SHANNON ROBOTICS

Principles of Robot Motion via Rust

The FCP Way

Complete mathematical foundations, interactive intuition, and working implementations in Rust — for robots that know where they are and must decide how to move.

Foundation · Conceptual · Practical

Principles of Robot Motion via Rust — The FCP Way

Print edition, generated from the interactive web edition.

This book takes Choset, Lynch, Hutchinson, Kantor, Burgard, Kavraki and Thrun's *Principles of Robot Motion* as its spine and brings it forward: asymptotically optimal sampling-based planners, trajectory optimization and model predictive control, multi-robot and manipulation planning, learning inside the planning loop, and an integrated planning stack. It is the sister volume of *Probabilistic Robotics via Rust*, with which it shares its robot, its simulator and its method; estimation is treated here only as far as a planner needs it, and cross-referenced there for the rest.

Every chapter of the web edition carries interactive simulations that run the same algorithms the text derives. In this print edition each appears as a described figure with its widget identifier, so the two editions can be read side by side.

Set in Space Grotesk and a Palatino-class serif. Composed with LaTeX.

How to read this book

Every chapter makes three passes over its subject, and they are meant to be read in this order.

Conceptual comes first, because intuition should precede formalism. In the web edition you meet an interactive figure before you meet an equation; here that figure is described, and the description tells you what it would show you.

Foundation is the mathematics, in full. Definitions are precise, assumptions are stated where they are made rather than buried, and derivations are carried to the end. Long algebra sits in marked derivation blocks, so you can take the result on faith the first time through and come back for the proof later.

Practical is the Rust. Not pseudocode dressed up as code: real types, real crates, code you could lift into a robot.

The running lab

Three robots and three worlds recur throughout, so that every new method can be compared against the last on identical ground. **Rusty** is a differential-drive rover with a range sensor — the same robot as in the sister volume — and lives in the **Apartment**, a 2-D floorplan with rooms, doorways and a long corridor; for most of Parts I–IV it is treated as a disc. **Reach** is a planar two-link arm (with a three-link variant) working at the **Workbench**, a table with polygonal obstacles; its configuration space is a torus, which is why it appears whenever topology matters, and its dynamics carry Part V. **Hitch** is a car-like robot that can pull a trailer and lives in the **Lot**; it cannot move sideways, which is the whole subject of Chapters 20 and 21. All three are built in Chapter 2 and act together in Chapter 23.

Colour carries meaning

One convention runs through the prose, the equations, the figures and the code comments alike. It is worth internalising before you start:

start · goal · obstacle · C-obstacle · roadmap / tree · path · robot

Green is always where you start and red is always where you want to be. Slate is a thing in the world; amber is the set of configurations that would make the robot

touch it. Blue is what a planner has explored — a roadmap, a tree, a wavefront. Purple is the path it chose. Orange is the robot itself and the trajectory it actually executed. Where a belief appears (Chapters 15–16 and the capstone) the sister volume's five estimation colours are used unchanged:

prior prediction measurement posterior ground truth

The teal used for chapter furniture is deliberately outside both sets, so it can never be mistaken for data.

Contents

How to read this book v

I Foundations: Robots, Worlds, and Configuration Space 1

1 Robots That Plan: The Piano Mover's Problem 3

2 The Lab: Three Robots, Three Worlds, One Simulator 23

3 Bug Algorithms: Planning with a Contact Sensor 47

4 Configuration Space I: Obstacles in the Space of Configurations 77

5 Configuration Space II: Topology, Manifolds, and Rigid Bodies 111

II Classical Planners: Search, Potentials, Roadmaps, Cells 137

6 Graph Search for Planners 139

7 Potential Functions 163

8 Roadmaps I: Visibility Graphs and the Generalized Voronoi Diagram 191

9 Roadmaps II: Sensor-Based Roadmaps and Silhouettes 215

10 Cell Decompositions and Coverage 247

III Sampling-Based Planning 279

11 Probabilistic Roadmaps 281

12 Tree Planners: EST, RRT, and Friends 307

13 Optimal Sampling-Based Planning 331

14 Planning for Many Bodies: Multi-Robot, Manipulation, and Kinodynamic Previews 357

IV Planning with Uncertainty 385

15 Kalman Filtering: The Geometric Observer 387

16 Bayesian Localization and Mapping 411

V Dynamics, Trajectories, and Constraints 439

17 Robot Dynamics 441

18 Trajectory Planning 463

19 Trajectory Optimization and MPC 491

20 Nonholonomic Systems I — Controllability 517

21 Nonholonomic Systems II — Steering Cars, Trailers, and Underactuated Systems 547

VI Frontiers and Integration 583

22 Learning and Planning 585

23 Capstone: A Planning Stack from Sensor to Motion 617

Part I

Foundations: Robots, Worlds, and Configuration Space

CHAPTER 1

Robots That Plan: The Piano Mover's Problem

FOUNDATIONS — ROBOTS, WORLDS, AND CONFIGURATION SPACE · Foundational · 40 min

> *The prototypical task is to find a path for a robot, whether it is a robot arm, a mobile robot, or a magically free-flying piano, from one configuration to another while avoiding obstacles.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), Chapter 1*

You already know how to make a robot move. This book is about making it decide *where* to move, and its thesis is that this is a discipline with theorems, not a bag of heuristics. The discipline rests on one idea that Choset and his co-authors put in their first paragraph and never let go of: a robot, however many joints it has, is a **point** in the right space. Motion planning does not happen in the room; it happens in configuration space, where the piano is a point and the walls have become shapes the point must avoid. Everything else in the book — bugs, potentials, roadmaps, samplers, trajectories, Lie brackets — is a way of finding a curve for that point.

This chapter sets the voice for the other twenty-two: hook first, playful when it helps, and every playful claim backed immediately by a figure you can drag, a number you can check, or a `cargo run` that prints that number. It is the one chapter where intuition outweighs mathematics, but nothing stated here is retracted later.

◎ AFTER THIS CHAPTER YOU CAN

- State the path-planning problem as Choset writes it — a curve c : [0,1] → Q_free — and say what each symbol means

- Explain why planning for a body is planning for a point in a different space, and what that reduction costs
- Place any planner on Choset's three-axis board: task, robot, algorithm, including the three kinds of completeness
- Run Bug1 by hand on a text world, reproduce the 28-step answer, and check it against the bound $d + 1.5 \Sigma p_i$
- Build and run the book's first Rust program, and know what the next twenty-two chapters will build on top of it

ASSUMED

- Fluency in some programming language (Rust not yet required — Appendix A is the on-ramp)
- Vectors and matrices at the "seen them before" level
- Comfort with the idea of a function

1.1 The problem

Below, a point robot with nothing but a contact sensor is crawling toward a goal it cannot see past. Its entire strategy is two sentences long: walk toward the goal; if you bump into something, go around it. Watch it arrive.

INTERACTIVE · w1.1

Bug Preview

A contact sensor and two behaviors give a complete planner with a length bound — no map, no heuristic.

Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch01-robots-that-plan#w1.1`

This is a provably complete planner, and it fits in forty lines.

"Complete" is a technical word and the whole book is careful about it: the planner finds a path whenever one exists and reports failure in finite time when none does. It needs no map. It never plans ahead. It has a *theorem* — the counter under the canvas never exceeds the bound printed beside it, and the bound is a formula, not a measurement. Drag the goal behind the larger obstacle: the walk around it gets longer and the bound grows with it, exactly as the formula says it must. Switch to Bug2, the greedy sibling that leaves an obstacle the first time it can, and the two totals tell you something that is true of search in general: greed is usually faster and sometimes disastrous. The "comb" preset is a world built so that greed loses.

Everything in this chapter is a first look at something a later chapter proves. The Bugs are Chapter 3's. The deeper claim — that the robot's shape can be made to disappear — is Chapter 4's, and it comes next.

1.2 Building intuition

1.2.1 The piano is a point

Choset's opening problem is the **piano mover's problem**: a rigid body, a known set of stationary obstacles, a start and a goal; find a collision-free path. In the plane it becomes the sofa mover's problem, and a sofa is enough to make the point.

> INTERACTIVE · w1.2
>
> **Piano Mover**
>
> The obstacle the planner fights is not the wall. It is the amber shape in configuration space, and it depends on the robot.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch01-robots-that-plan#w1.2

On the left is the room: a wall with a doorway, and a sofa you can drag and turn. On the right is something you have probably never looked at before. It is the *same* sofa, drawn as a single dot, in a space whose axes are the sofa's position — and the amber shapes around the dot are the sofa's *configurations* that would put it through a wall, at the heading it currently has.

Drag the sofa into the wall. The dot touches amber at the same instant. Not before, not after: the two pictures are the same fact drawn twice. Now turn the sofa and watch the amber reshape itself. The walls did not move. The *robot* changed, and the obstacle the planner fights changed with it. At one heading the doorway is a wide channel between two amber blobs; turn the sofa broadside and the channel closes. A sofa that only fits diagonally has a channel that exists at some headings and not at others — your first sighting of a narrow passage, the thing that makes Part III hard.

> THE ONE SENTENCE VERSION
>
> Planning for a body among walls is planning for a point among *C-obstacles* — the configurations whose footprint touches a wall — and the C-obstacle depends on the robot as much as on the wall.

Three vocabulary words have just been used informally, and the Foundation section pins them down. A **configuration** is a complete description of where

every point of the robot is: for the sofa, its position and heading, (x, y, θ). The **configuration space** $\mathcal{Q}$ is the set of all of them. A **C-obstacle** is the set of configurations at which the robot overlaps a workspace obstacle. The right-hand pane shows one slice of $\mathcal{Q}$ — fixed θ — because $\mathcal{Q}$ is three-dimensional and the screen is not. Toggle "stack all θ-slices" to see the slices pile into the translucent block of Choset's figure F.8, and notice the channel twisting as it climbs.

1.2.2 Choset's gallery

Choset's Chapter 1 opens with a gallery of problems, and each one carries a definition this book will need. The **piano mover** is the pure geometric problem: obstacles known and stationary, execution exact, planning finished before the robot moves — *offline* planning. NASA's **Mini AERCam**, a free-flying inspection camera with twelve cold-gas thrusters, is the same problem with one word changed: because it moves by applying forces, it needs not just a path but the speed along the path — a *trajectory*, and with it dynamics (Part V). The **CyCab** and the **Segway**, small vehicles that must parallel park, look like the sofa mover's problem until you notice that a car cannot slide sideways; that velocity constraint is called *nonholonomic*, the car is called nonholonomic too, and Chapters 20 and 21 are about it. **Demining** — passing a sensor over every point of a field — is the first time *completeness* is the whole point: a planner that might miss a mine is not a planner. The **painting arm** wants its coverage to be *time-optimal*, because robots in factories cost money by the second. **Snake robots** have more joints than their task needs — *redundant*, or hyper-redundant — and non-Euclidean configuration spaces of high dimension. The surgical robots, the digital actors, and the protein-folding problem each get a paragraph in Choset and a pointer here: Chapter 14 picks up the molecule as an articulated linkage.

Two of Choset's examples are handed to the sister book, which tells them properly. **RHINO**, the museum tour guide, is a localization story — a robot comparing its sensors against a stored map because exact execution is a fiction — and **Sojourner** on Mars is a mapping story. *Probabilistic Robotics via Rust* owns both; this book borrows their estimation only as far as a planner needs it, in Part IV.

1.2.3 The cast

Choset's protagonists were the robots of 2005. This book's are built in Chapter 2 and used until the end. **Rusty** is a disc-shaped differential-drive rover, shared with the sister book, whose home is **the Apartment** — five rooms off a long corridor. Its configuration is a planar pose, and for collision purposes a point of $\mathbb{R}^2$. **Reach** is a planar two-link arm, with a three-link variant for redundancy, bolted to **the Workbench**; its configuration is two angles, a point on a torus. **Hitch** is a car that may pull one trailer, living in **the Lot**; its configuration is a pose in SE(2), plus one more circle for the trailer. Three robots, three configuration

spaces of three different shapes, so that when the book says "a point in Q" it can mean three concretely different things.

1.2.4 The planner's taxonomy

What kind of thing is a planner? Choset's Table 1.1 answers with three axes, and the board below is that table with the chapters of this book placed on it.

> INTERACTIVE · w1.3
>
> **The Planner's Taxonomy**
>
> Motion planning is not 'find the shortest path'. Most of the board has nothing to do with optimality.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch01-robots-that-plan#w1.3

The **task** comes first: navigate (one configuration to another), cover (a tool over every point), localize (a map and sensors to a configuration), map (an unknown environment to a representation). Then the **robot**: how many degrees of freedom and what the configuration space looks like; whether it can move instantaneously in any direction of Q (*omnidirectional*) or is subject to velocity constraints (*nonholonomic*); whether velocities are its controls (*kinematic*) or forces are (*dynamic*). Then the **algorithm**: optimal or merely feasible; its computational complexity; which of three kinds of completeness it has; online or offline; sensor-based or working from a model.

Hover "Optimal" and count. Most of this book — most of the field — is not about shortest paths. It is about finding *a* path, with a guarantee about when you will find one, for a robot whose shape and constraints make "a path" hard to define. Hover "Complete" and notice that two of its chapters, 3 and 9, are *sensor-based* planners with no map at all. A planner without a map is not a heuristic. The hook above was one.

1.3 The mathematics

This chapter's mathematics is deliberately light, and nothing in it is retracted later. The symbols below are the book's global notation, making their first appearance; each is formalized in the chapter named.

$\mathcal{W} \subset \mathbb{R}^2$ or $\mathbb{R}^3$	Workspace: the ambient space the robot's body lives in. Chapter 2
$\mathcal{WO}_i$, $\mathcal{W}_{\text{free}} = \mathcal{W} \setminus \bigcup_i \mathcal{WO}_i$	The i-th workspace obstacle; the free workspace. Chapter 2

$\mathcal{Q}$, q	Configuration space; one configuration — a point the planner imagines. Chapters 4–5
$R(q) \subset \mathcal{W}$	Footprint: the set of workspace points the robot occupies at q. Chapter 2
$\mathcal{QO}_i$, $\mathcal{Q}_{\text{free}}$	C-obstacle, the configurations whose footprint meets WO_i; free configuration space. Chapter 4
$c : [0,1] \to \mathcal{Q}_{\text{free}}$	A path; a trajectory when the parameter is time. Chapter 18
q_{start}, q_{goal}	The query. Chapter 3
$d(x, y)$	Euclidean distance in the plane (the Bug algorithms); a metric in general. Chapter 5
p_i	Perimeter of workspace obstacle i (the Bug bounds). Chapter 3

1.3.1 Definitions

DEFINITION: CONFIGURATION

A **configuration** is a complete specification of the position of every point of the robot. The configuration space $\mathcal{Q}$ is the set of all configurations, and $\dim \mathcal{Q}$ — the number of independent parameters — is the robot's number of **degrees of freedom**. The piano has six: three for position, three for orientation.

DEFINITION: THE PATH-PLANNING PROBLEM (CHOSET EQ. 1.1)

Find a continuous map

$$c : [0,1] \to \mathcal{Q} \quad \text{with} \quad c(0) = q_{\text{start}}, \qquad c(1) = q_{\text{goal}}, \qquad c(s) \in \mathcal{Q}_{\text{free}} \ \ \forall s \in [0,1].$$

The unit interval is a convention; any parameterization serves.

DEFINITION: PATH AND TRAJECTORY

A **path** is a continuous curve in $\mathcal{Q}$ — a C^0 function. A **trajectory** is a path parameterized by time, and to have velocities and accelerations it must be at least C^2. Finding a path is *path planning*; finding a feasible trajectory is *trajectory planning*, which Choset also calls motion planning. This book

keeps his distinction: Parts I–IV plan paths; Part V plans trajectories.

DEFINITION: OFFLINE, ONLINE, SENSOR-BASED

An **offline** planner constructs the plan in advance from a known model and hands it to an executor. An **online** planner builds the plan incrementally while the robot executes; it can then be **sensor-based**, interleaving sensing, computation, and action. Choset calls the boundary "murky": an offline planner fast enough to re-run in a feedback loop *is* an online planner, and the primary distinction is computation time. This book keeps that remark because it anticipates model predictive control in Chapter 19.

DEFINITION: OMNIDIRECTIONAL AND NONHOLONOMIC; KINEMATIC AND DYNAMIC

A robot free to move instantaneously in any direction of $\mathcal{Q}$ (obstacles aside) is **omnidirectional** — the sofa. One subject to velocity constraints, like a car that cannot slide sideways, is **nonholonomic**, and so is the constraint — Hitch. A **kinematic** model takes velocities as controls; a **dynamic** model takes forces, and velocities become part of the state.

DEFINITION: THREE KINDS OF COMPLETENESS

A planner is **complete** if it finds a solution whenever one exists and otherwise reports failure in finite time. It is **resolution complete** if it finds a solution whenever one exists at the chosen discretization. It is **probabilistically complete** if the probability of finding a solution, when one exists, tends to one as running time grows. Optimality, completeness, and computational complexity trade off against one another; this book states which of the three a planner has, every time it introduces one.

1.3.2 The robot is a point

DERIVATION **The configuration-space reduction**

c is collision-free $\iff$ $c([0,1]) \subset \mathcal{Q}_{\text{free}}$

Statement. For any robot and any workspace obstacles, a curve c in $\mathcal{Q}$ is a collision-free motion of the robot's body if and only if it is a curve in $\mathcal{Q}_{\text{free}}$. Planning for a body is planning for a point in a different space.

Step 1 — the footprint is a function of q. By the definition of configuration, q fixes the position of every point of the robot, so the occupied set $R(q)$ is a function of q alone.
Step 2 — collision is membership. The robot at q collides with obstacle $\mathcal{WO}_i$ iff $R(q) \cap \mathcal{WO}_i \neq \emptyset$, which by definition is $q \in \mathcal{QO}_i$:

$$\mathcal{QO}_i = \{\, q \in \mathcal{Q} : R(q) \cap \mathcal{WO}_i \neq \emptyset \,\}.$$

Step 3 — along a curve. A motion c avoids every collision iff $c(s) \notin \mathcal{QO}_i$ for every s and every i, i.e. iff c avoids $\bigcup_i \mathcal{QO}_i$.
Step 4 — the complement. $\mathcal{Q}_{\text{free}} = \mathcal{Q} \setminus \bigcup_i \mathcal{QO}_i$, so "avoids every C-obstacle" is exactly "lies in $\mathcal{Q}_{\text{free}}$", which is the last clause of eq. (1.1). ■
Why the reduction is not free. Three things are hidden in the word "different". The C-obstacles are usually not computable in closed form — Chapter 4 does the cases that are, a disc and a polygon that translates, and the Piano Mover widget above computes its amber by brute force, asking the collision checker at every cell. The space $\mathcal{Q}$ is usually not $\mathbb{R}^n$: the sofa's θ wraps, Reach's two angles make a torus, and Chapter 5 is about what that costs. And the dimension is the enemy: the volume of $\mathcal{Q}$ grows exponentially with the number of joints, which is why Part III exists and why Appendix D says "PSPACE-hard".

DERIVATION **Dimension counting for the piano**

$\dim \mathcal{Q} = 6$ in $\mathbb{R}^3$, 3 in the plane

Statement. A free rigid body in $\mathbb{R}^3$ has six degrees of freedom; in the plane, three.
Step 1 — one point. Pick a reference point on the body. Placing it takes three coordinates in $\mathbb{R}^3$, two in the plane.
Step 2 — orientation. With one point fixed, the body can still rotate about it: three parameters in $\mathbb{R}^3$ (roll, pitch, yaw, or any chart of $SO(3)$), one in the plane.
Step 3 — rigidity. Once the reference point and the orientation are fixed, rigidity fixes every other point. $3 + 3 = 6$; $2 + 1 = 3$. ■
The warning. Six *coordinates* do not make $\mathcal{Q}$ equal to $\mathbb{R}^6$. Yaw by 2π returns the piano to where it was, so the orientation coordinates wrap, and some charts of $SO(3)$ become singular (the gimbal lock of Chapter 5's Rotation Zoo). Chapter 4 does the counting argument with holonomic constraints the way Choset's §3.3 does — points, then constraints, then the dimension that remains.

DERIVATION **Bug1 is complete and its path is bounded (quoted, not yet proved)**

$L_{\text{Bug1}} \le d(q_{\text{start}}, q_{\text{goal}}) + 1.5 \sum_i p_i$

Statement (Choset eq. 2.1). For a point robot with a contact sensor in a bounded planar workspace, the length of Bug1's path satisfies

$$L_{\text{Bug1}} \;\le\; d(q_{\text{start}}, q_{\text{goal}}) + 1.5 \sum_i p_i,$$

and Bug1 reports failure in finite time when no path exists.
Step 1 — straight segments. Between obstacles the robot moves straight toward the goal from a point that is strictly closer than its previous hit point; those segments total at most $d(q_{\text{start}}, q_{\text{goal}})$.
Step 2 — each obstacle. Meeting obstacle i costs one full circumnavigation (p_i) to survey the perimeter, plus the walk to the perimeter point nearest the goal — at most half the perimeter the shorter way round ($p_i/2$).
Step 3 — finitely many. A bounded workspace with finitely many obstacles is met finitely often; sum. ■
What is deferred. The proof that the leave point is well defined, the test that detects an unreachable goal, and Bug2's weaker bound $d + \frac{1}{2}\sum_i n_i p_i$ are Chapter 3's. Here the inequality is *checked*, numerically, on the text world below — with one grid-specific footnote stated where it arises.

DERIVATION **Why 'just drive toward the goal' is not a planner**

\text{greedy descent on } d(\cdot, \qgoal) \text{ has local minima}

Statement. A controller that always moves to reduce $d(\cdot, q_{\text{goal}})$ gets stuck in any workspace with a concave obstacle facing the start.
Step 1 — build the trap. Put a U-shaped obstacle between the start and the goal, open toward the start.
Step 2 — enter it. Driving toward the goal takes the robot into the U, because every step into the mouth reduces d.
Step 3 — the bottom. At the bottom of the U every admissible direction *increases* d. The controller stops. It has reached a local minimum of the function it descends, and by its own definition of success it has succeeded. ■
This is Choset's warning that "purely reactive gradient-following potential field approaches always run the risk of getting stuck in local minima", and the book's first instance of the local-minimum disease that Chapter 7 names and cures. Bug1 escapes the U because boundary following is a

> *second behavior* that does not descend d at all — it deliberately walks away from the goal, and a rule that only ever descends can never decide to do that.

1.4 The algorithm

Bug1 is stated informally here and formally as Choset's Algorithm 1 in Chapter 3. Two behaviors, one memory, one test.

> **ALGORITHM** `Bug1(W, q_start, q_goal) -- Choset Algorithm 1, informally` COST `path length ≤ d + 1.5 Σ p_i; each step is O(1) sensing`
>
> **in** a bounded planar workspace W seen only through a contact sensor; the robot measures d(x, y) and its own position exactly
>
> **out** a path to q_goal, or the conclusion that q_goal is unreachable

1: $i \leftarrow 1$, $q_0^L \leftarrow q_{\text{start}}$
2: repeat
3: **motion-to-goal:** from q_{i-1}^L move toward q_{goal} until it is reached or an obstacle is hit at q_i^H
4: if q_{goal} is reached, exit
5: **boundary-following:** follow the obstacle's boundary until q_i^H is re-encountered, recording the perimeter point q_i^L closest to q_{goal}
6: go to q_i^L the shorter way round
7: if moving toward q_{goal} from q_i^L would re-enter the obstacle, conclude q_{goal} is unreachable and exit
8: $i \leftarrow i + 1$

Line 5 is the exhaustive search: Bug1 walks the *whole* perimeter before committing to a leave point, which is why its leave point is optimal and its cost is a full p_i. Bug2, Choset's Algorithm 2, replaces lines 5–6 with a greedy exit: fix the **m-line** from q_{start} to q_{goal} once, and leave the boundary the first time the robot is back on the m-line, closer to the goal than the hit point, and able to step toward the goal. Bug2 is quick when obstacles are simple and can be arbitrarily worse than Bug1 when they are not.

1.4.1 The worked example, by hand

The book's first "numeric example equals unit test". A text world, nine columns by five rows, origin at the top left, 4-connected unit moves, a point robot that occupies one cell and whose contact sensor reports whether the next cell is #:

```
.........
...###...
S..###..G
...###...
.........
```

$q_{\text{start}} = (0, 2)$, $q_{\text{goal}} = (8, 2)$, one 3×3 obstacle in columns 3–5, rows 1–3. Its perimeter, measured as the ring of free cells the robot walks when going round it, is $p_1 = 16$.

1. *Motion-to-goal* along row 2: $(1, 2)$, $(2, 2)$ — 2 steps. The next cell $(3, 2)$ is #, so $q_1^H = (2, 2)$.
2. *Circumnavigate* the ring back to $(2, 2)$ with the obstacle on the right hand: 16 steps, recording $d(\cdot, q_{\text{goal}})$ at every cell. The unique minimum is $(6, 2)$, at distance 2.
3. *Walk to the leave point* $q_1^L = (6, 2)$ the shorter way round. The ring is symmetric about row 2, so both ways are 8 steps; the code breaks the tie clockwise. 8 steps.
4. *Motion-to-goal*: $(7, 2)$, $(8, 2)$ — 2 steps.

Total $L_{\text{Bug1}} = 2 + 16 + 8 + 2 = \mathbf{28}$ steps. The bound: $d + 1.5\, p_1 = 8 + 24 = 32 \geq 28$. For contrast, Bug2 on the same world leaves the ring the first time it is back on row 2 and closer to the goal — at $(6, 2)$, after 8 steps of boundary following — for $2 + 8 + 2 = \mathbf{12}$, which here is also the shortest 4-connected path.

The grid-specific footnote promised above. On a 4-connected grid "move straight toward the goal" is a staircase whose length is $|\Delta x| + |\Delta y|$, not the Euclidean d. The bound the Bug Preview prints, and the one the library checks on random worlds, is Choset's inequality with that grid distance d_1 in place of d; on the worked example the m-line is a row, so $d_1 = d = 8$ and the two bounds coincide. Chapter 3 does the continuous version, where the footnote disappears.

WORK IT OUT

Make the obstacle 4×4 (columns 3–6, rows 1–4) in a 10×6 world with S at $(0, 2)$ and G at $(9, 2)$. How many steps does Bug1 take?

Answer. 33 The ring around a 4×4 block is $6 \times 6 - 4 \times 4 = 20$ cells. The hit point is $(2, 2)$ again and the leave point is $(7, 2)$; but the block is no longer symmetric about row 2 — it extends two rows above and three below — so the shorter way round, over the top, is 9 steps, not 10. Total $2 + 20 + 9 + 2 = 33$, against a bound of $9 + 30 = 39$. The library check `integration lab` pins all three numbers. If you predicted 34, you assumed symmetry; the code did not.

1.5 Implementation in Rust

The first program is a dependency-free Rust binary, edition 2024, by design: your first build must finish in seconds and cannot fail on a native dependency. The workspace `Cargo.toml` already pins the book's stack (`nalgebra` 0.35, `parry2d` 0.30, `rand` 0.9, `petgraph` 0.8, `pathfinding` 4.15), but nothing in this chapter touches it.

Setup (CI-tested on Linux, macOS, Windows): install `rustup` and the stable toolchain (edition 2024 needs 1.85 or later); clone the book's repository; then

```
cargo run -p ch01_hello
cargo test -p ch01_hello
```

Optional, and deferred to Chapter 2: `rustup target add wasm32-unknown-unknown` for the browser cross-check, and Node 20.9 or later for the site in `web/`.

1.5.1 The world and the sensor

RUST **crates/ch01_hello/src/main.rs -- the world**

```rust
//! Bug1 on a text world. Zero dependencies, edition 2024.

/// '#' is obstacle, everything else is free. 4-connected unit moves.
struct TextWorld { rows: Vec<Vec<u8>> }

#[derive(Clone, Copy, PartialEq, Eq, Debug)]
struct Cell { x: i32, y: i32 }

/// Headings in screen coordinates (y grows downward): E, S, W, N. Turning right adds one.

#[derive(Clone, Copy, PartialEq, Eq, Debug)]
enum Dir { E, S, W, N }

/// The index into STEP back to a heading: 0 -> E, 1 -> S, 2 -> W, 3 -> N.
impl From<usize> for Dir {
    fn from(i: usize) -> Dir { [Dir::E, Dir::S, Dir::W, Dir::N][i % 4] }
}

impl Dir {
    fn right(self) -> Dir { Dir::from(self as usize + 1) }
    fn left(self) -> Dir { Dir::from(self as usize + 3) }
    fn back(self) -> Dir { Dir::from(self as usize + 2) }
}

impl Cell {
    /// One unit move; STEP is defined with the behaviors below.
    fn step(self, d: Dir) -> Cell {
        let (dx, dy) = STEP[d as usize];
        Cell { x: self.x + dx, y: self.y + dy }
    }
}

impl TextWorld {
    /// The contact sensor: is the cell blocked -- or outside the bounded workspace?
    /// Choset assumes W is bounded; here the border is simply a wall the sensor feels.
    fn blocked(&self, c: Cell) -> bool {
        if c.y < 0 || c.x < 0 { return true; }
        match self.rows.get(c.y as usize).and_then(|r| r.get(c.x as usize)) {
```

```
            Some(b'#') => true,
            Some(_) => false,
            None => true,
        }
    }

    /// Parse a map; S and G are the query.
    fn parse(s: &str) -> (Self, Cell, Cell) {
        let rows: Vec<Vec<u8>> = s.lines().filter(|l| !l.is_empty()).map(|l| l.bytes().
   collect()).collect();
        let find = |ch: u8| rows.iter().enumerate()
            .find_map(|(y, r)| r.iter().position(|&b| b == ch).map(|x| Cell { x: x as i32,
    y: y as i32 }))
            .expect("the map marks S and G");
        let (s, g) = (find(b'S'), find(b'G'));
        (TextWorld { rows }, s, g)
    }
}

/// Euclidean distance d(x, y) -- Bug1 may measure it, by assumption (Choset sec.2.1).
fn dist(a: Cell, b: Cell) -> f64 {
    (((a.x - b.x).pow(2) + (a.y - b.y).pow(2)) as f64).sqrt()
}
```

1.5.2 The two behaviors

RUST **crates/ch01_hello/src/main.rs -- motion-to-goal and boundary following**

```
const STEP: [(i32, i32); 4] = [(1, 0), (0, 1), (-1, 0), (0, -1)]; // E, S, W, N

/// One step of motion-to-goal: the 4-neighbor that most reduces d(., goal).
/// `Err(dir)` when that neighbor is blocked -- the robot has hit the obstacle,
/// pushing in direction `dir`.
fn motion_to_goal(w: &TextWorld, x: Cell, goal: Cell) -> Result<Cell, Dir> {
    let (best, dir) = (0..4)
        .map(|d| (Cell { x: x.x + STEP[d].0, y: x.y + STEP[d].1 }, d))
        .min_by(|a, b| dist(a.0, goal).total_cmp(&dist(b.0, goal)))
        .expect("four neighbors");
    if w.blocked(best) { Err(Dir::from(dir)) } else { Ok(best) }
}

/// One step of boundary following with the obstacle on the robot's right hand --
/// the grid analogue of the surface-normal rule of Choset sec.2.3.1. O(1) sensing.
///
/// Try to turn toward the wall (it may have turned away), else go straight,
/// else turn away, else reverse. On the ring around a block this traces the
/// 8-neighborhood of the obstacle, which is the perimeter the bound counts.
fn follow_boundary(w: &TextWorld, x: Cell, heading: Dir) -> (Cell, Dir) {
    for h in [heading.right(), heading, heading.left(), heading.back()] {
        let n = x.step(h);
        if !w.blocked(n) { return (n, h); }
    }
    (x, heading) // boxed in on all four sides: cannot happen from a free start
}

/// Walk the whole boundary from the hit point until it is re-encountered.
/// Returns the cells visited after the hit, ending with the hit itself --
```

```rust
/// so the length is the perimeter p_i as the robot measures it.
fn circumnavigate(w: &TextWorld, hit: Cell, toward: Dir) -> Vec<Cell> {
    let (mut cell, mut heading, mut ring) = (hit, toward.left(), Vec::new());
    loop {
        (cell, heading) = follow_boundary(w, cell, heading);
        ring.push(cell);
        if cell == hit { return ring; }
    }
}
```

1.5.3 Bug1 and the program

RUST **crates/ch01_hello/src/main.rs -- Bug1 and main**

```rust
enum Outcome { Reached(Vec<Cell>), Unreachable { at: Cell } }

/// Bug1, Choset Algorithm 1: motion-to-goal; on contact circumnavigate fully,
/// remembering the perimeter point closest to the goal; return to it the shorter
/// way; repeat. If leaving the leave point toward the goal would re-enter the
/// obstacle, the goal is unreachable.
fn bug1(w: &TextWorld, start: Cell, goal: Cell) -> (Outcome, Vec<usize>) {
    let (mut path, mut perimeters, mut x) = (vec![start], Vec::new(), start);
    while x != goal {
        match motion_to_goal(w, x, goal) {
            Ok(next) => { x = next; path.push(x); }
            Err(dir) => {
                let hit = x;
                let ring = circumnavigate(w, hit, dir);
                perimeters.push(ring.len());
                path.extend(&ring);
                // The leave point: closest to the goal; the first wins a tie.
                let (i, _) = ring.iter().enumerate()
                    .min_by(|a, b| dist(*a.1, goal).total_cmp(&dist(*b.1, goal))).unwrap
   ();
                // The shorter way round; a tie goes forward (clockwise).
                if i + 1 <= ring.len() - 1 - i { path.extend(&ring[..=i]); }
                else { path.extend(ring[i..ring.len() - 1].iter().rev()); }
                x = ring[i];
                if motion_to_goal(w, x, goal).is_err() {
                    return (Outcome::Unreachable { at: x }, perimeters);
                }
            }
        }
    }
    (Outcome::Reached(path), perimeters)
}

/// The map with the executed path marked `.`, and S and G on top.
fn render(w: &TextWorld, s: Cell, g: Cell, path: &[Cell]) -> String {
    let mut rows: Vec<Vec<char>> = w.rows.iter()
        .map(|r| r.iter().map(|&b| if b == b'#' { '#' } else { '.' }).collect())
        .collect();
    for c in path { rows[c.y as usize][c.x as usize] = '.'; }
    rows[s.y as usize][s.x as usize] = 'S';
    rows[g.y as usize][g.x as usize] = 'G';
    rows.iter().map(|r| r.iter().collect::<String>()).collect::<Vec<_>>().join("\n")
}
```

```
const WORLD: &str = "\
.........
...###...
S..###..G
...###...
.........
";

fn main() {
    let (w, s, g) = TextWorld::parse(WORLD);
    match bug1(&w, s, g) {
        (Outcome::Reached(path), perimeters) => {
            let steps = path.len() - 1;
            let p: usize = perimeters.iter().sum();
            println!("{}", render(&w, s, g, &path));
            println!("steps: {steps}  straight-line: {}  bound d + 1.5.Sigmap: {}", dist(
   s, g), dist(s, g) + 1.5 * p as f64);
        }
        (Outcome::Unreachable { at }, _) => println!("goal unreachable; detected at {at:?}
   "),
    }
}
```

```
.........
...###...
S..###..G
...###...
.........
steps: 28  straight-line: 8  bound d + 1.5.Sigmap: 32
```

(The rendering marks the executed path with ·: out along row 2, once around the block, then the shorter way back to the leave point. The Bug Preview above also marks the hit point `H` at $(2, 2)$ and the leave point `L` at $(6, 2)$.) The tests lock the number, and lock the failure mode:

RUST **crates/ch01_hello/tests/worked_example.rs**

```
#[test]
fn worked_example_ch01() {
    let (w, s, g) = TextWorld::parse(WORLD);
    let (Outcome::Reached(path), perimeters) = bug1(&w, s, g) else { panic!("reachable")
   };
    assert_eq!(path.len() - 1, 28);
    assert_eq!(perimeters, vec![16]);
    assert!(28.0 <= 8.0 + 1.5 * 16.0);
}

#[test]
fn unreachable_is_detected() {
    let (w, s, g) = TextWorld::parse(".........\n.....###.\nS....#G#.\n.....###.\n
   .........\n");
    let (outcome, perimeters) = bug1(&w, s, g);
    assert!(matches!(outcome, Outcome::Unreachable { .. }));
    assert_eq!(perimeters.len(), 1, "one circumnavigation, then give up");
}
```

```rust
/// Exercise 6: implement `bug2`, then `cargo test --features exercise-bug2`.
#[cfg(feature = "exercise-bug2")]
#[test]
fn bug2_text_world() {
    // Chapter 3 derives Bug2 properly; the expected count on this world is 12.
    let (w, s, g) = TextWorld::parse(WORLD);
    assert_eq!(bug2(&w, s, g).len() - 1, 12);
}
```

> **NOTE**
>
> The Bug Preview at the top of this chapter runs `web/lib/robots/bug-preview.ts`, a line-for-line port of this program onto a rasterized board, and the book's self-checks assert the same 28, the same 16, the same 12, and the same unreachable verdict on the same text worlds. The animation you watched and the number you printed are one computation.

1.6 Putting it together

The integration lab is your own machine. Clone the workspace, build, reproduce the 28. Then edit `WORLD` to the 4×4 block of the exercise above, *predict* the new length and the new bound before running, run, and check. The point is not the number; it is that you have just watched a guarantee stay true while the world changed under it, and that a test will tell you if a future edit ever breaks it.

Two conventions begin here and never change. **Every worked number in this book is reproduced by a test**: a Rust test in the crate and an invariant in the browser library, so that the prose, the program, and the animation cannot drift apart without the build going red. And **every chapter runs the same rhythm**, in the same colors.

> **INTERACTIVE · w1.4**
>
> **How to Read This Book**
>
> Every chapter runs the same rhythm, and every color means the same thing in a figure, an equation, and a line of Rust.
>
> Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch01-robots-that-plan#w1.4`

The color code is not decoration. Hover a tinted term in an equation on any page — try the QO_i in the first derivation above — and every figure on the page fades every other role. The amber in the Piano Mover, the amber `term-cobstacle` in the definition of QO_i, and a `// --rm-cobstacle` comment in a Rust listing are the same object seen three ways. Green is where you start, red is where you are

going (and, in Chapter 2's simulator, the flash of a collision — a collision is where a plan ends), slate is a wall, amber hatched is a wall seen from configuration space, blue is a roadmap or tree, purple is the path a planner found, orange is the robot and the path it actually drove.

A map of what comes next, by cell of the taxonomy board. Chapter 2 builds the cast and the collision checker that every planner asks its four questions of. Chapter 3 proves what this chapter only checked: Bug1, Bug2, and Tangent Bug, with their bounds. Chapters 4 and 5 (Chapter 5) are configuration space done properly — the amber computed exactly, the torus and SE(2) as manifolds. Part II is the classical planners for a point in $\mathcal{Q}_{\text{free}}$: search, potentials, roadmaps, cells. Part III trades completeness for speed and gets *probabilistic* completeness back. Part IV is what a planner must know about not knowing where it is. Part V puts time, dynamics, and Hitch's constraint back in. Part VI closes the loop the hook above opened: three robots, one planning stack, running live.

1.7 Exercises

1 FOUNDATION • **Classify the gallery**

Classify each of Choset's motivating examples — the piano mover, the Mini AERCam, the CyCab and Segway, RHINO, Sojourner, demining, the painting arm, snake robots, surgical robots, digital actors, protein folding — on all three axes of the taxonomy board. Which cells of Table 1.1 does the gallery leave empty? Name one robot deployed since 2020 that fills each empty cell.

2 FOUNDATION •• **Count the degrees of freedom**

The sofa in the Piano Mover has $\dim \mathcal{Q} = 3$. Count the degrees of freedom of (a) two sofas in the same room, (b) two sofas tied by a rope of fixed length pulled taut, (c) one sofa on casters that cannot roll sideways. For each, say whether the constraint changes $\dim \mathcal{Q}$, and why — this is the holonomic versus nonholonomic distinction of Choset §3.3, met early.

HINTS

<>Independent bodies add: (a) is $3+3$.</> <>A taut rope is one equation on the six coordinates — an equation on positions. Such a constraint removes a dimension.</> <>Casters that cannot slide constrain the sofa's velocity, not its position: every configuration in SE(2) is still reachable (a car can be parked anywhere), so $\dim \mathcal{Q}$ is unchanged. That is what nonholonomic means, and Chapter 20 proves the "still reachable".</>

3 CONCEPTUAL • **Predict the leave side**

In the Bug Preview, set the goal directly behind the larger of the two obstacles.

PREDICT FIRST

Before pressing play: which side will Bug1 leave the obstacle from, and will the counter land closer to d or to the bound?

- The side it hits on; closer to d
  Bug1 never leaves from the hit side unless the hit point is itself the perimeter point nearest the goal, which a goal behind the obstacle rules out.
- ✓ The far side, nearest the goal; closer to the bound
  The leave point is the perimeter point closest to the goal — on the far side — and reaching it costs a full circumnavigation plus up to half a perimeter, which is most of the 1.5 p_i term.
- Whichever side is shorter to walk; closer to d
  Bug1 chooses the leave point by distance to the goal, not by walking cost. Only the final approach to the chosen point uses the shorter way round.

Now switch to Bug2 and explain in one sentence why its counter differs: Bug2 commits to the first m-line crossing that is closer than the hit point, so it never pays for the survey — and on the comb preset, where the m-line crosses the obstacle many times, it pays for the lack of one.

4 CONCEPTUAL •• **The widest channel**

In the Piano Mover, shorten the sofa until it fits the doorway nose-first. Predict how the amber slice changes as you rotate the sofa by 90°, then verify. Which heading gives the widest channel through the doorway, and why is that heading *not* $\theta = 0$ when the doorway is in a side wall?

HINTS

<>The channel's width at a heading is the doorway's width minus the sofa's extent across the doorway at that heading.</> <>For a doorway in a vertical wall, the extent across it is the sofa's projection onto the y-axis: $L|\sin\theta| + W|\cos\theta|$ for length L and width W.</> <>That projection is smallest when the long axis points through the door — along x, so $\theta = 0$ for this room. Rotate the room's wall and the best heading rotates with it; "nose-first" is a statement about the doorway, not about the axes.</>

5 PRACTICAL •• **A C-shaped obstacle and a lying sensor**

Modify `ch01_hello` so the obstacle is a `C` shape opening toward the start. Predict the path length and the bound before running. Then make the contact sensor *lie* with probability 0.05, using a fixed seed and a tiny hand-rolled linear congruential generator — still zero dependencies — and report what breaks: the length, the guarantee, or both.

6 PRACTICAL ••• **Bug2, and a world where it loses**

Implement Bug2 in `ch01_hello` by changing only the exit condition of boundary following: leave when back on the fixed m-line, closer to the goal than the hit point, and able to step toward it. Make `bug2_text_world` pass with 12 — it compiles only when you ask for it, with `cargo test --features exercise-bug2`. Then construct a text world where Bug2's count exceeds Bug1's. The comb preset in the Bug Preview is one answer — a hook whose nearest-to-goal point sits high above the m-line, so that Bug1 leaves from there on a fresh line over a row of teeth that Bug2, welded to the m-line, must climb one by one. You have just drawn Choset's figure 2.4.

1.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

Chapter 1 is the source of the gallery, Table 1.1, eq. (1.1), and the notation this book adopts; §2.1 is the source of Bug1, Bug2, and the bound checked here. Lumelsky and Stepanov's 1987 Algorithmica paper, 'Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape', is the Bug algorithms' original, cited there as [301]. `link`

Lozano-Pé rez, T. (1983). Spatial Planning: A Configuration Space Approach. *IEEE Transactions on Computers C-32(2), 108–120*.

The paper that made 'the robot is a point' an algorithm: C-obstacles computed for polygons, the ancestor of the Piano Mover's amber. `doi:10.1109/TC.1983.1676196`

Latombe, J.-C. (1991). Robot Motion Planning. *Kluwer Academic Publishers (The Springer International Series in Engineering and Computer Science, vol. 124)*.

The reference text for the decade before Choset, and the source of the piano-mover framing of configuration space that Chapter 1 inherits; its author wrote Choset's foreword. `doi:10.1007/978-1-4615-4022-9`

Kavraki, L. E., Švestka, P., Latombe, J.-C., and Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation 12(4), 566–580*.
Where 'probabilistically complete' — the third kind of completeness on the taxonomy board — enters the field. Chapter 11. doi:10.1109/70.508439

Karaman, S. and Frazzoli, E. (2011). Sampling-based Algorithms for Optimal Motion Planning. *International Journal of Robotics Research 30(7), 846–894*.
The modern occupant of the board's 'Optimal' cell: asymptotic optimality for sampling-based planners. Chapter 13. doi:10.1177/0278364911406761

Şucan, I. A., Moll, M., and Kavraki, L. E. (2012). The Open Motion Planning Library. *IEEE Robotics & Automation Magazine 19(4), 72–82*.
What a planner is today: infrastructure, not a research artifact. The systems framing behind the taxonomy board's chapter links. doi:10.1109/MRA.2012.2205651

CHAPTER 2

The Lab: Three Robots, Three Worlds, One Simulator

FOUNDATIONS — ROBOTS, WORLDS, AND CONFIGURATION SPACE · Intermediate · 55 min

> *For the translating mobile robot described above, the workspace and the configuration space are both two-dimensional Euclidean spaces, but it is important to keep in mind that these are different spaces.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
> *Principles of Robot Motion (2005), §3.1*

Every later chapter of this book plans *for* somebody, *somewhere*. This chapter builds the somebody and the somewhere: three robots — Rusty the disc-shaped rover, Reach the two-link arm, Hitch the car that may pull a trailer — and three worlds — the Apartment, the Workbench, the Lot. It gives each robot a configuration whose *type* says which space it lives in, a footprint $R(q)$ that is a function of that configuration, and one collision checker that answers exactly four questions.

Two ideas carry the chapter. The pedagogical one: **a configuration is a typed value, and $R(q)$ is a function.** The compiler will not let you hand a point on a torus to a function that wants a rigid-body pose, and that refusal is the first theorem of Chapter 5, enforced by `rustc` three chapters early. The engineering one, inherited from the sister book: **determinism is a feature.** A seed and an input script determine a run to the last bit, natively and in the browser, and the book holds itself to that with a test.

Nothing here is a planner. Everything here is what a planner is made of.

◎ AFTER THIS CHAPTER YOU CAN

- Name the three robots and three worlds of this book and state the

configuration space of each as a type
- Write the forward kinematics of a planar revolute arm as a chain of rigid motions, and reproduce Choset's worked number
- State the four questions a planner may ask of geometry — intersects, distance, witness, swept — and nothing else
- Explain why a disc's collision query is a point query minus a radius, and why an arm's is not
- Reproduce one 60 s run of all three robots bit for bit from a seed, and say why that matters

ASSUMED

- Chapter 1: the cast, W, R(q), QO_i
- Rust basics (Appendix A); a cargo workspace that builds
- SE(2) poses as (x, y, θ) — the sister book's Chapter 3, used here as an API

2.1 The problem

Three cockpits, one engine. Below, Rusty is circling the Apartment's corridor, Reach is sweeping the Workbench, and Hitch is backing through the Lot. Each robot follows a scripted tour drawn from the seed in the transport bar, and each flashes red for a few frames when it touches something.

INTERACTIVE · w2.1

Three-Robot Dashboard

A configuration is a few typed numbers and R(q) follows from them; you drive the angles, not the tip.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch02-the-lab#w2.1

Take over. Click a pane, then drive. Rusty takes a forward speed and a turn rate from the arrow keys; Hitch takes a speed and a steering angle; Reach has no keys at all — you drag its elbow and its tip. Three things will happen to you, and each is a sentence of this chapter.

You will try to drive Reach's tip and fail. Drag the elbow handle and the tip traces a circle, whether you wanted a circle or not. Reach does not have a tip you can command; it has two angles, and the tip is where the angles put it. That is the forward kinematics map φ, and it is the first piece of mathematics below.

You will try to slide Hitch sideways into a bay and fail. No combination of arrow keys produces a lateral motion. The car's model has no input that does

that — not a missing feature, a constraint, and the subject of Chapter 20. For now it is physics the simulator enforces.

You will drive Rusty into a wall and stop short of it. Rusty halts a body radius before contact, not at contact. A disc at (x, y) collides when the wall is within r of its center, and the collision checker says so. The red flash is the only place in this book where red means "bad"; it borrows the goal's color because a collision is where a plan ends.

The readouts under the panes print each configuration *as its type*: `Pose2 { x, y, theta }` for Rusty, `T2 { th1, th2 }` for Reach, `HitchCfg { body, trailer }` for Hitch. Those are not display strings. They are the names of three different Rust types, and the rest of the book is written against them.

> **THE ONE SENTENCE VERSION**
>
> A robot is a function $R(q)$ from a typed configuration q to a set of workspace points, a world is a list of named segments and polygons, and a collision checker is the only thing allowed to put the two together — through four queries, and no others.

2.2 Building intuition

2.2.1 Three configurations, three types

Choset writes "we use q" and leaves the reader to remember what kind of thing q is. This book does not leave it to the reader. Rusty, considered as a disc for collision purposes, is a point of $\mathbb{R}^2$. Reach is a point of the torus $T^2 = S^1 \times S^1$: two angles, each wrapped to $(-\pi, \pi]$, with no joint limits so the links may pass over each other. Hitch's body is a pose in SE(2), and with a trailer the configuration picks up one more circle for the trailer's heading, $\mathrm{SE}(2) \times S^1$.

Those are four different spaces with four different shapes. T^2 is compact and has finite area; $\mathbb{R}^2$ is flat and infinite; SE(2) is three-dimensional and not compact; a circle wraps and a line does not. Chapter 5 makes these differences theorems. Chapter 2 makes them types, which is the version a compiler can check:

RUST `crates/robots/src/lib.rs -- the configuration types`

```rust
/// A point of R2: the disc's center. Rusty, as the planner sees it.
#[derive(Clone, Copy, Debug, PartialEq)]
pub struct R2(pub Point2<f64>);

/// A point of T2: two joint angles, each wrapped to (-pi, pi]. Reach.
#[derive(Clone, Copy, Debug, PartialEq)]
pub struct T2 { pub th1: f64, pub th2: f64 }

/// A point of SE(2) * S1 -- or SE(2) alone when there is no trailer. Hitch.
#[derive(Clone, Copy, Debug, PartialEq)]
pub struct HitchCfg { pub body: Pose2, pub trailer: Option<f64> }
```

Try to hand the wrong one to the arm and the compiler answers before the program runs:

```
error[E0308]: mismatched types
  --> examples/lab_tour.rs:31:20
   |
31 |     let tip = reach.fk(&HitchCfg { body, trailer: None });
   |                     -- ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected `&Tn<2>`, found `&
   HitchCfg`
```

The compiler just told you that $T^2 \neq \mathrm{SE}(2) \times S^1$. Chapter 5 tells you why.

2.2.2 What a collision checker returns

The second thing to unlearn is that a collision checker returns a boolean. Freeze one scene and ask it the four questions a planner is allowed to ask.

> **INTERACTIVE · w2.2**
>
> **Query Anatomy**
>
> A collision checker does not return a boolean. It returns geometry: a distance, two witness points, and a verdict on a whole motion.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch02-the-lab#w2.2

The default scene is the Workbench with Choset's own arm — link lengths $L_1 = L_2 = 1$, zero thickness — at his own configuration $q = (\pi/4, \pi/2)$ from Example 3.8.1. The numbers on the canvas are the chapter's worked numbers: the distance tab reads 0.0607 m to the block, and the witness tab puts b^* at the block's corner $(0.5, 1.0)$. Slide θ_2 and watch the distance fall to exactly zero at the instant the body turns red; keep sliding and it stays at zero while the footprint is inside. The distance query is not a smoothed boolean. It is a boolean *and* a number, and when the number is zero the boolean is what you have.

The witness pair is what Chapters 7 and 8 run on. Drag the tip around the block and watch b^* jump from the corner to an edge the moment the nearest feature changes; the potential field of Chapter 7 pushes along exactly that segment, and the Voronoi diagram of Chapter 8 is the set of configurations where two witnesses tie.

The swept tab is the one planners cannot live without. It ghosts a second configuration one chart step away and asks whether the *motion* between the two is free — not the endpoints, the motion. Set the step large and you will find pairs of perfectly free endpoints with a red ghost between them: the arm passes through the post on its way from one to the other. Every edge of every roadmap in Part II and Part III is checked with this query, and no planner in this book ever checks only the endpoints of anything.

2.2.3 A world is not a bitmap

The third thing to unlearn is that a map is a picture. The three worlds are lists of named geometry: wall segments with endpoints, obstacle polygons with vertices, and places with names.

> INTERACTIVE · w2.3
>
> **World Atlas**
>
> A world is not a bitmap. It is polylines and polygons with names, and every grid in the book is built from it.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch02-the-lab#w2.3

The Apartment is the sister book's floor plan, ported unchanged: five rooms off a long corridor, with rooms A and C mirror images of each other so that a range sensor cannot tell them apart. The Workbench is a 4.8 m square table with Reach bolted to its center and three obstacles inside the arm's reach: *the block*, which is the square $[0.5, 1.0] \times [1.0, 1.5]$ of the worked example, *the post*, a column in the arm's left-hand half, and *the shelf*, a bar along the near edge. The Lot is 16 m by 10 m, curbed, with four bays along the north side and two pillars Hitch must back around.

Because every wall is a segment and every obstacle a polygon, the distance from a point to the nearest wall is an exact number, a ray cast is an exact intersection, and the collision queries of the previous section are exact too. When Chapter 7 needs a grid it rasterizes these worlds at a chosen resolution; the grid is derived from the world, never the other way round. Click anywhere in the atlas to teleport the resident robot and the checker will refuse a configuration whose footprint overlaps anything — the same refusal, from the same function, that stops the dashboard's robots.

2.3 The mathematics

r	Rusty's body radius, 0.11 m, inherited from the sister book.
θ_1, θ_2 (θ_3)	Reach's joint angles, each in S^1; $q = (\theta_1, \theta_2) \in T^2$.
L_1, L_2 (L_3)	Reach's link lengths. $L_1 = L_2 = 1$ m for every worked number.
$\varphi : T^2 \to \mathbb{R}^2$	Reach's forward kinematics: joint angles to tip position.
(x, y, θ), ψ	Hitch's rear-axle pose in SE(2); the trailer heading in S^1.
ℓ, ϕ, d_h	Hitch's wheelbase; its steering angle (a control); the hitch length to the trailer axle.
$\rho_R(x, s_k)$	The range sensor's k-th ray from x, saturated at R. Full treatment in Chapter 3.

$\mathrm{dist}(A, B)$, (a^*, b^*)	Distance between two shapes, and a pair of points attaining it: the witnesses.

2.3.1 Definitions

DEFINITION: CONFIGURATION TYPE

A **configuration type** is a value type that *is* a chart of $\mathcal{Q}$: `R2` for a translating disc, `T2` for Reach (angles wrapped to $(-\pi, \pi]$), `Pose2` for a planar rigid body, `HitchCfg = (Pose2, S¹)` for a car and trailer. Mixing them is a compile error, not a runtime check.

DEFINITION: FOOTPRINT

The **footprint** $R(q) \subset \mathcal{W}$ is the compound shape the robot occupies at q. For Rusty it is Choset's disc (§3.1),

$$R(x, y) = \{\, (x', y') : (x - x')^2 + (y - y')^2 \le r^2 \,\},$$

for Reach it is two (or three) segments — or thin capsules of radius ρ — and for Hitch a rectangle for the body and one for the trailer. In every case the footprint is a *function* of q and nothing else.

DEFINITION: FORWARD KINEMATICS (CHOSET §3.8)

A **forward kinematics map** $\varphi : \mathcal{Q} \to M$ takes one representation of the configuration to another — here joint angles to the tip's position. It is in general not a homeomorphism even when $\dim \mathcal{Q} = \dim M$: for Reach the inverse is two-valued, elbow-up and elbow-down.

DEFINITION: THE COLLISION QUERY CONTRACT

For a footprint F and a scene S, a collision checker answers exactly four questions. *Intersects:* $F \cap S \neq \emptyset$. *Distance:* $\min_{a \in F,\, b \in S} \|a - b\|$, which is zero when they intersect. *Witness:* a minimizing pair (a^*, b^*). *Swept:* whether F, moving rigidly from q_a to q_b along a straight line in the chart, intersects S anywhere on the way. These four, and no others, are what a planner may ask.

DEFINITION: PLANNER-SIDE RANGE SENSOR

$\rho_R(x, s_k)$ for n equally spaced bearings s_k, by exact ray casting with *no noise*, reporting ∞ beyond the saturation range R. This is not the sister book's LiDAR, which adds Gaussian range noise and dropouts. Choset's Bug guarantees assume an ideal sensor, and the text of Chapter 3 says so where it uses them.

DEFINITION: REPRODUCIBLE RUN

A run is a pure function (seed, input script) $\mapsto$ trajectory. Each stochastic subsystem owns one random stream split from the seed, so adding a noisy sensor to a simulation never perturbs the trajectory of a robot that does not use it.

2.3.2 Reach's forward kinematics

Reach's base is pinned at the origin. Joint 1 rotates the whole arm by θ_1; joint 2, which rides at the end of link 1, rotates link 2 by a further θ_2 *relative to link 1*. That relativity is the whole derivation.

DERIVATION **Reach's forward kinematics (Choset Example 3.8.1)**

$\varphi(\theta_1, \theta_2) = (L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2),\ L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2))$

Statement. The tip of the 2R arm is at

$$\varphi(\theta_1, \theta_2) = \begin{pmatrix} L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2) \\ L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2) \end{pmatrix}.$$

Step 1 — the elbow. Link 1 leaves the origin at absolute angle θ_1 and has length L_1, so the elbow is at $L_1(\cos\theta_1, \sin\theta_1)$.
Step 2 — link 2's absolute heading. Joint angles are *relative*: θ_2 is measured from link 1's direction, so link 2 leaves the elbow at absolute heading $\theta_1 + \theta_2$.
Step 3 — add link 2. The tip is the elbow plus L_2 along that heading, which is the statement.
Step 4 — the 3R variant. A third link appends $L_3(\cos(\theta_1 + \theta_2 + \theta_3), \sin(\theta_1 + \theta_2 + \theta_3))$ to the tip, and the tip's heading is $\theta_1 + \theta_2 + \theta_3$ — the position-and-orientation map of Choset's problem 21. ■
The same map as a product of rigid motions. Write $T(\theta, L) \in \mathrm{SE}(2)$ for "rotate by θ, then translate L along the new x-axis". Then

$$\varphi(q) = \big[\, T(\theta_1, L_1)\, T(\theta_2, L_2) \,\big] \cdot \mathbf{0},$$

and for N links the product has N factors. This is how `robots::Reach::fk` computes it — one running heading and one running position, accumulated link by link — and it is why the same code serves the 2R arm, the 3R arm, and any `Reach<N>`.

Choset's number. With $L_1 = L_2 = 1$ and $q = (\pi/4, \pi/2)$: the elbow is $(\cos 45°, \sin 45°) = (0.7071, 0.7071)$; link 2 heads at 135°; so the tip is $(0.7071 + \cos 135°,\ 0.7071 + \sin 135°) = (0, \sqrt{2}) \approx (0, 1.4142)$. The library reproduces this to 10^{-12}, and the dashboard's Reach readout shows it when you drag the joints there.

2.3.3 The workspace is an annulus, not a configuration space

Choset's §3.1 makes a point that is easy to nod along to and hard to internalize: the set of positions the tip can reach is not the arm's configuration space, because knowing where the tip is does not tell you where the elbow is.

DERIVATION **Reach's workspace is an annulus with a two-valued inverse**

$\{x : |L_1 - L_2| \le \|x\| \le L_1 + L_2\}$

Statement. The reachable tip positions are the annulus $|L_1 - L_2| \le \|x\| \le L_1 + L_2$, and every interior point is reached by exactly two configurations.

Step 1 — the triangle inequality. The origin, the elbow, and the tip form a triangle with sides L_1, L_2, and $\|x\|$. A triangle exists iff $|L_1 - L_2| \le \|x\| \le L_1 + L_2$.

Step 2 — the law of cosines. In that triangle the angle at the elbow is $\pi - \theta_2$, so

$$\|x\|^2 = L_1^2 + L_2^2 - 2L_1L_2\cos(\pi - \theta_2) = L_1^2 + L_2^2 + 2L_1L_2\cos\theta_2,$$

which determines $\cos\theta_2$ from $\|x\|$ alone — and therefore θ_2 up to sign.

Step 3 — two arms. The two signs are the elbow-down and elbow-up solutions; θ_1 then follows from atan2 of the tip minus the direction of link 1 within the triangle. Two configurations, one tip position: the tip is *not* a complete description of the robot, so the annulus is not $\mathcal{Q}$. ■

Closed-form inverse kinematics (Choset problem 20), as `Reach::ik` computes it:

$$\cos\theta_2 = \frac{\|x\|^2 - L_1^2 - L_2^2}{2L_1L_2}, \qquad \theta_2 = \operatorname{atan2}\big(\pm\sqrt{1-\cos^2\theta_2},\ \cos\theta_2\big), \qquad \theta_1 = \operatorname{atan2}(x_2, x_1) - \operatorname{atan2}\big(L_2\sin\theta_2,\ L_1 + L_2\cos\theta_2\big).$$

On the annulus boundary the two solutions coincide; outside it there are none. The library check `ik_roundtrip` pushes 1,000 random configurations through φ, both inverses, and φ again, and requires the tip to return to within 10^{-9}.

2.3.4 A disc's collision is a point query

DERIVATION **Collision of a disc is a point query**

collides $\iff D(x, y) < r$, distance $= D - r$

Statement. Rusty at (x, y) collides iff $D(x, y) < r$, where D is the distance from the center to the nearest wall; the distance query returns $D - r$ and the witness on the obstacle is the same nearest point, with the witness on the robot shifted r toward it.
Step 1. A disc meets a set iff some point of the set is within r of the center.
Step 2. "Some point within r" is the statement $D < r$. (Touching, $D = r$, counts as collision in this book's closed-set convention, so the free test is the strict $D > r$.)
Step 3. The nearest wall point b^* realizes D; the disc point nearest to it is the center moved r along the direction to b^*, at distance $D - r$. ■
What this is. This is the Minkowski inflation of Choset §3.2.1 written in one line: growing the obstacles by r and shrinking the robot to a point are the same operation. It is also why `collide::DiscChecker` uses exact point-to-segment distance rather than a general convex-distance routine: for a disc the general machinery buys nothing. The general case — a polygon that rotates, an arm whose links sweep — is where the black box earns its keep, and Chapter 4 opens it.
The worked number. Rusty's center at $(2, 1)$, one wall from $(3, 0)$ to $(3, 3)$: the point distance is 1.0, so the distance query is $1.0 - 0.11 = 0.89$ with witness $(3, 1)$ on the wall.

2.3.5 Hitch's model is a definition, not yet a theorem

DERIVATION **Hitch's kinematic car with a trailer**

$\dot{\theta} = (v/\ell)\tan\phi, \quad \dot{\psi} = (v/d_h)\sin(\theta - \psi)$

Statement. The simulator integrates

$$\dot{x} = v\cos\theta, \qquad \dot{y} = v\sin\theta, \qquad \dot{\theta} = \frac{v}{\ell}\tan\phi, \qquad \dot{\psi} = \frac{v}{d_h}\sin(\theta - \psi),$$

with (x, y) the rear-axle midpoint, θ the heading, ϕ the steering angle, ψ the trailer heading, ℓ the wheelbase, and d_h the hitch length from the rear axle to the trailer axle.

Step 1 — rear-axle no-slip. The rear wheels roll without sliding, so the rear axle's velocity points along the heading: the first two equations, and no equation at all for a sideways velocity.

Step 2 — the front wheel sets the radius. The front axle is ℓ ahead and turned by ϕ; the instantaneous center of rotation lies where the two axle lines meet, at distance $\ell/\tan\phi$ from the rear axle, so $\dot{\theta} = v/(\ell/\tan\phi)$.

Step 3 — the hitch drags the trailer. The trailer axle is d_h behind the hitch point along ψ and also rolls without sliding, so only the component of the hitch's velocity perpendicular to the trailer turns it: $\dot{\psi} = (v/d_h)\sin(\theta - \psi)$. ■

How the code integrates it. For constant (v, ϕ) over one tick the body moves on an arc of constant curvature $\kappa = \tan\phi/\ell$, so the ported `boxplus` with the tangent vector $(v\,\Delta t,\ 0,\ \kappa v\,\Delta t)$ integrates the body *exactly*. The trailer does not move on a circle, so ψ is integrated by a fixed-step RK4 inside the tick. Both are pure functions of $(q, u, \Delta t)$, which is what the replay test needs.

What this is not yet. Each of these equations is a Pfaffian constraint $\omega(q)\dot{q} = 0$ — "no sideways velocity" written as a one-form. Chapter 20 derives that view, proves that sideways *displacement* is nonetheless reachable by a Lie bracket of the two inputs, and Chapter 21 turns the bracket into a parking maneuver. Here it is the simulator's physics, stated.

2.3.6 The worked example, assembled

Put the pieces together on the Workbench. Reach with $L_1 = L_2 = 1$ and zero-thickness links at $q = (\pi/4, \pi/2)$; the block $[0.5, 1.0] \times [1.0, 1.5]$.

The elbow is at $(0.7071, 0.7071)$ and the tip at $(0, 1.4142)$. Link 2 runs from the elbow to the tip along the line $x + y = \sqrt{2} \approx 1.4142$. Every point of the block has $x + y \geq 1.5$, so the block lies entirely on one side of that line and the nearest pair is the block's corner closest to the line against the foot of its perpendicular:

$$b^* = (0.5, 1.0), \qquad \text{dist} = \frac{1.5 - \sqrt{2}}{\sqrt{2}} = \frac{1.5}{\sqrt{2}} - 1 \approx 0.0607, \qquad a^* = b^* - 0.0607\,(0.7071, 0.7071) \approx (0.4571, 0.9571).$$

The foot a^* lies between elbow and tip, so it is on the link and not clipped to an endpoint; *intersects* is false. These are the numbers the Query Anatomy widget

shows at its default pose, and the library check `reach_distance_to_square` holds them to 10^{-9} for the distance and 10^{-6} for the witnesses.

One honest footnote. Zero-thickness links make the number exact and the arm unrealistic. The default Workbench arm wears 3 cm capsules, and the same query then reads $0.0607 - 0.03 = 0.0307$ m. Toggle "3 cm capsule links" in the widget and watch the number drop by exactly the radius.

WORK IT OUT

With 3 cm capsule links, what distance does the same query return (in metres)?

Answer. 0.0307 m The capsule's surface is 0.03 closer to everything than its axis is, so every distance drops by exactly the radius: $0.0607 - 0.03 = 0.0307$. The witness b^* does not move; a^* slides 0.03 toward it. The check `capsule links` in the library pins this.

2.4 The algorithm

What a collision checker *computes* is stated here as a contract and left sealed. The Rust crate delegates every geometric primitive to `parry2d`; the TypeScript port that runs the widgets hand-rolls them. Chapter 4 opens the box for polygons, Appendix C documents what the engine does, and Appendix F.5 of Choset — the GJK algorithm of Gilbert, Johnson, and Keerthi — is the name of the idea inside.

ALGORITHM `Collision::{intersects, distance, witness, swept}(fp)` COST `O(k log n)` per query for k footprint parts, plus the engine's iterations (GJK, shape-cast)

in a footprint fp = R(q) as a list of (pose, shape) parts; a scene of n obstacles behind a bounding-volume tree

out a boolean; a distance ≥ 0; a witness pair or None; a boolean for the sweep

1: `intersects(fp)`: for each part of `fp`, ask the tree for candidate obstacles; return true on the first part–obstacle pair whose shapes intersect
2: `distance(fp)`: for each part and each candidate, take the engine's distance; return the minimum (0 if any pair intersects)
3: `witness(fp)`: as `distance`, but keep the pair (a^*, b^*) attaining the minimum
4: `swept(fp_a, fp_b)`: for each part, cast its shape from the pose in `fp_a` to the pose in `fp_b` against the candidates; return true if any cast reports a time of impact in $[0, 1]$

The browser port has no shape-cast, so it marches:

ALGORITHM `edge_free(a, b) -- conservative sphere marching` COST

`O(log(1/clearance))` distance queries on a clear edge; `O(L / step_min)` at worst

in configurations a, b; distance(q); a Lipschitz bound L on how far any footprint point moves over the whole chart line a → b

out true iff every configuration on the straight chart line from a to b is free

1: if not `is_free(a)` or not `is_free(b)`: return false
2: $s \leftarrow 0$
3: while $s < 1$:
4: $d \leftarrow$ `distance(interpolate(a, b, s))`; if $d \le 0$ return false
5: $s \leftarrow s + \max(d/L,\ \text{step}_{\min})$ — nothing within d of the footprint is occupied, and no point of it moves faster than L per unit s, so no collision can begin before $s + d/L$
6: return true

The bound L is what makes the march *conservative* rather than hopeful. For a disc it is the Euclidean length of the edge. For Reach, a point on any link moves at most $(L_1 + L_2)$ per radian of either joint, so $L = (L_1 + L_2)(|\Delta\theta_1| + |\Delta\theta_2|)$ with the differences taken the short way round the circle. For Hitch it is the translation plus the body's circumscribed radius times the rotation. The library check `swept` compares the march against dense sampling at 2 mm chart spacing on 264 random edges across all three robots and finds no disagreement.

The remaining named routines of the chapter are small enough to state in a line each:

Name	Signature	Cost
`Reach::fk`	`(&self, q: &Tn<N>) -> [Point2; N + 1]` — base, joints, tip	$O(N)$
`Hitch::step`	`(&self, q: &HitchCfg, u: &CarCmd, dt: f64) -> HitchCfg`	$O(1)$
`RangeSensor::scan`	`(&self, scene: &Scene, x: Point2) -> Vec<f64>`, ∞ beyond R	$O(n_{\text{rays}} \log n_{\text{seg}})$
`Run::record` / `Run::replay`	`(cfg, script) -> Log` / `(&Log) -> Vec<Frame>`	$O(\text{ticks})$

The sensor is worth one remark now and a chapter later. Its scan is Choset's saturated raw distance function ρ_R sampled at n bearings, and $\min_k \rho_R(x, s_k)$ is a sensor's estimate of $D(x)$, the distance to the nearest obstacle. It can never be *smaller* than $D(x)$ — a ray cannot hit something closer than the closest thing —

and it equals $D(x)$ exactly when a ray happens to point at the nearest point. From Rusty's $(2, 1)$ with the wall at $x = 3$, ray 0 does, and the library check `scan_is_rho_R` sees 1.000000.

2.5 Implementation in Rust

This section is the book's architectural contract. Later chapters add crates; none restructures what is fixed here.

Crates. `nalgebra` 0.35 for `Point2` and `Isometry2`; `parry2d` 0.30 as `parry2d-f64` for shapes, the `Bvh` broad phase, and `query::{distance, closest_points, intersection_test, cast_shapes}` — WASM-clean, no `rayon`; `rand` 0.9 with `SmallRng` for seeded streams; `serde` for the replay log. Geometry is the one thing this book does not hand-roll.

```
crates/
  prob/          # PORTED: Rng = seeded SmallRng wrapper
  geom/          # PORTED: Pose2 = SE(2)
  edt/           # PORTED: exact Euclidean distance transform
  sim/           # PORTED: World (Apartment), Rusty (diff-drive step, encoders), ray cast
  robots/        # THIS CHAPTER: Reach<N> (FK, IK), Hitch (car + trailer), Workbench, Lot
  collide/       # THIS CHAPTER: trait Collision over parry2d; Footprint; Scene;
   RangeSensor
  widget-kit/    # THIS CHAPTER: Role enum (the only place a color is named), capture,
   replay
  ch01_hello/    # Chapter 1 (std-only)
  bugs/ cspace/ manifold/ search/ ...                    # added by their chapters
web/lib/
  prob/ geom/ sim/ mapping/edt                          # ported unchanged
  robots/ collide/                                      # this chapter's TypeScript
   mirrors
```

2.5.1 The arm

RUST **crates/robots/src/reach.rs**

```rust
use nalgebra::Point2;

/// A point of T^n: N joint angles, each wrapped to (-pi, pi].
#[derive(Clone, Copy, Debug, PartialEq)]
pub struct Tn<const N: usize>(pub [f64; N]);
pub type T2 = Tn<2>;

/// Planar revolute arm with N links. `Reach<2>` is the book's Reach; `Reach<3>`
/// is the redundant variant of Exercise 5 and Chapter 14.
pub struct Reach<const N: usize> {
    pub base: Point2<f64>,
    pub lengths: [f64; N],
    /// Capsule radius of each link. 0 gives exact segments -- every worked number.
    pub link_radius: f64,
}

impl<const N: usize> Reach<N> {
    /// Choset's Example 3.8.1: L_1 = L_2 = 1 at the origin, zero-thickness links.
    pub fn choset() -> Reach<2> {
```

```rust
        Reach { base: Point2::origin(), lengths: [1.0, 1.0], link_radius: 0.0 }
    }

    /// phi as the chain T_0_1(theta_1) T_1_2(theta_2) ?: joint positions base..tip.
    ///
    /// One running heading and one running position. Joint angles are relative,
    /// so the absolute heading of link k is theta_1 + ? + theta_k -- that single fact is
    /// the whole derivation, and it is why the same loop serves any N.
    pub fn fk(&self, q: &Tn<N>) -> Vec<Point2<f64>> {
        let mut pts = Vec::with_capacity(N + 1);
        let (mut heading, mut p) = (0.0, self.base);
        pts.push(p);
        for i in 0..N {
            heading += q.0[i];
            p += nalgebra::Vector2::new(heading.cos(), heading.sin()) * self.lengths[i];
            pts.push(p);
        }
        pts
    }

    pub fn tip(&self, q: &Tn<N>) -> Point2<f64> {
        *self.fk(q).last().expect("N + 1 points")
    }

    /// R(q): one capsule per link, as parry2d shapes with their poses.
    pub fn footprint(&self, q: &Tn<N>) -> Footprint {
        let pts = self.fk(q);
        Footprint::capsules(pts.windows(2).map(|w| (w[0], w[1])), self.link_radius)
    }
}

impl Reach<2> {
    /// Closed-form IK (Choset problem 20): [elbow-down, elbow-up]; None outside the
   annulus.
    pub fn ik(&self, tip: Point2<f64>) -> [Option<T2>; 2] {
        let [l1, l2] = self.lengths;
        let d = tip - self.base;
        let c2 = (d.norm_squared() - l1 * l1 - l2 * l2) / (2.0 * l1 * l2);
        if !(-1.0..=1.0).contains(&c2) {
            return [None, None];
        }
        let s2 = (1.0 - c2 * c2).sqrt();
        let solve = |sign: f64| {
            let th2 = (sign * s2).atan2(c2);
            let th1 = d.y.atan2(d.x) - (l2 * th2.sin()).atan2(l1 + l2 * th2.cos());
            Tn([wrap(th1), wrap(th2)])
        };
        [Some(solve(1.0)), Some(solve(-1.0))]
    }
}
```

The browser runs `lib/robots/reach.ts`, a line-for-line port with `Tn` as a readonly number array; `fmtTn` is what prints `T2 { th1:  0.785, th2:  1.571 }` in the dashboard.

2.5.2 The car

RUST **crates/robots/src/hitch.rs**

```rust
/// SE(2) * S1 with a trailer; SE(2) without.
#[derive(Clone, Copy, Debug, PartialEq)]
pub struct HitchCfg { pub body: Pose2, pub trailer: Option<f64> }

/// Speed and steering angle. phi is a *control*: it is not part of the configuration.
#[derive(Clone, Copy, Debug)]
pub struct CarCmd { pub v: f64, pub phi: f64 }

pub struct Hitch {
    pub wheelbase: f64,
    pub max_steer: f64,
    pub body: (f64, f64),            // length, width of the chassis
    pub hitch_len: Option<f64>,      // rear axle -> trailer axle
}

impl Hitch {
    /// One tick. The constraint is *in* the model: no input produces a lateral
    /// velocity, so no command sequence slides the car sideways. (Chapter 20:
    /// a sideways *displacement* is still reachable, by a Lie bracket.)
    pub fn step(&self, q: &HitchCfg, u: &CarCmd, dt: f64) -> HitchCfg {
        let phi = u.phi.clamp(-self.max_steer, self.max_steer);
        let kappa = phi.tan() / self.wheelbase;
        let ds = u.v * dt;
        // Exact arc: the tangent vector (ds, 0, kappa ds) pushed through exp on SE(2).
        let body = q.body.boxplus(&Vector3::new(ds, 0.0, kappa * ds));
        let trailer = match (q.trailer, self.hitch_len) {
            (Some(psi0), Some(dh)) => {
                // psi_dot = (v/d_h) sin(theta(t) - psi), theta(t) = theta_0 + kappa v t
   known in closed form;
                // fixed-step RK4 over the tick keeps the run a pure function.
                let f = |t: f64, psi: f64| (u.v / dh) * (q.body.theta + kappa * u.v * t -
    psi).sin();
                Some(wrap(rk4(f, psi0, 0.0, dt, 4)))
            }
            _ => None,
        };
        HitchCfg { body, trailer }
    }

    /// R(q): the chassis rectangle, and the trailer's when attached. Both poses
    /// are functions of (x, y, theta, psi) -- the content of Exercise 2.
    pub fn footprint(&self, q: &HitchCfg) -> Footprint { /* two oriented rectangles */ }
}
```

2.5.3 The black box

RUST **crates/collide/src/collision.rs**

```rust
use nalgebra::{Isometry2, Point2};
use parry2d::math::{Pose, Vector};
use parry2d::query::{self, ClosestPoints, ShapeCastOptions};

```

```rust
pub struct Witness { pub on_robot: Point2<f64>, pub on_obstacle: Point2<f64>, pub dist: f64 }

/// The four questions, and no others.
pub trait Collision {
    fn intersects(&self, fp: &Footprint) -> bool;
    /// 0.0 when intersecting.
    fn distance(&self, fp: &Footprint) -> f64;
    fn witness(&self, fp: &Footprint) -> Option<Witness>;
    /// Rigid straight-line sweep in the chart from fp_a's poses to fp_b's.
    fn swept(&self, fp_a: &Footprint, fp_b: &Footprint) -> bool;
}

/// Walls as thin segments, obstacles as convex polygons, all under one Bvh. Obstacles are
/// stored in parry's own types; only the robot's footprint arrives in nalgebra's.
pub struct Scene { obstacles: Vec<Obstacle>, bvh: Bvh, pub bounds: Aabb }

/// parry >= 0.26 does its math in glam; the rest of the workspace speaks nalgebra.
/// The two meet here and nowhere else.
fn to_parry(p: &Isometry2<f64>) -> Pose {
    Pose::new(Vector::new(p.translation.x, p.translation.y), p.rotation.angle())
}
fn from_parry(v: Vector) -> Point2<f64> { Point2::new(v.x, v.y) }

impl Collision for Scene {
    // The only module in the workspace that imports parry2d::query.

    fn intersects(&self, fp: &Footprint) -> bool {
        fp.parts.iter().any(|(pose, shape)| {
            let pose = to_parry(pose);
            self.candidates(&pose, shape).any(|o| {
                query::intersection_test(&pose, &**shape, &o.pose, &*o.shape).unwrap_or(true)
            })
        })
    }

    fn distance(&self, fp: &Footprint) -> f64 {
        self.witness(fp).map_or(f64::INFINITY, |w| w.dist)   // 0.0 on contact; inf in an empty scene
    }

    fn witness(&self, fp: &Footprint) -> Option<Witness> {
        let mut best: Option<Witness> = None;
        for (pose, shape) in &fp.parts {
            let pose = to_parry(pose);
            for o in self.candidates(&pose, shape) {
                // GJK, inside the engine (Choset App. F.5). We never see the simplex.
                match query::closest_points(&pose, &**shape, &o.pose, &*o.shape, f64::MAX).ok()? {
                    ClosestPoints::Intersecting => return Some(Witness::contact(&pose)),
                    ClosestPoints::WithinMargin(a, b) => {
                        let dist = a.distance(b);
                        if best.as_ref().is_none_or(|w| dist < w.dist) {
                            best = Some(Witness { on_robot: from_parry(a), on_obstacle: from_parry(b), dist });
                        }
                    }
                    ClosestPoints::Disjoint => {} // cannot happen: the margin is infinite
```

```
                }
            }
        }
        best
    }

    fn swept(&self, fp_a: &Footprint, fp_b: &Footprint) -> bool {
        fp_a.parts.iter().zip(&fp_b.parts).any(|((pa, shape), (pb, _))| {
            let (pa, pb) = (to_parry(pa), to_parry(pb));
            let vel = pb.translation - pa.translation;   // over unit time
            self.candidates(&pa, shape).any(|o| {
                // A shape-cast reports a time of impact in [0, 1] if the swept volume hits.
                query::cast_shapes(&pa, vel, &**shape, &o.pose, Vector::ZERO, &*o.shape,
                                   ShapeCastOptions::with_max_time_of_impact(1.0))
                    .map(|hit| hit.is_some()).unwrap_or(true)
            })
        })
    }
}
```

> **NOTE**
>
> The widgets on this page run `web/lib/collide`, which hand-rolls the same four queries over segments and polygons — point-in-polygon, segment–segment, a separating-axis test for convex pairs, and the sphere march for the sweep. The library check `polygons` compares the separating-axis test against an edge-and-containment brute force on 400 random convex pairs; every other check on this page holds the port to the same numbers as the Rust.

2.5.4 The worked example as a program

RUST **crates/robots/examples/lab_tour.rs**

```
fn main() {
    let arm = Reach::<2>::choset();
    let q = Tn([FRAC_PI_4, FRAC_PI_2]);
    let pts = arm.fk(&q);
    println!("elbow ({:.4}, {:.4})  tip ({:.4}, {:.4})", pts[1].x, pts[1].y, pts[2].x, pts[2].y);

    // A scratch scene with the worked example's square -- which is also the Workbench's block.
    let scene = Scene::empty(WORKBENCH_BOUNDS).with_obstacle("the block", Cuboid::from_aabb(0.5, 1.0, 1.0, 1.5));
    let fp = arm.footprint(&q);
    let w = scene.witness(&fp).expect("a witness exists");
    println!("intersects: {}  distance: {:.4}", scene.intersects(&fp), scene.distance(&fp));
    println!("witness a*=({:.4}, {:.4}) b*=({:.4}, {:.4})", w.on_robot.x, w.on_robot.y, w.on_obstacle.x, w.on_obstacle.y);
```

```rust

    // Rusty against one wall.
    let wall = Scene::of_walls(&[Segment::new(point![3.0, 0.0], point![3.0, 3.0])]);
    let rusty = Footprint::disc(point![2.0, 1.0], RUSTY.body_radius);
    println!("distance: {:.4}", wall.distance(&rusty));

    // The 60 s tour at BOOK_SEED, recorded and replayed.
    let log = Run::record(TourConfig::default(), BOOK_SEED);
    assert_eq!(Run::replay(&log), log.frames, "replay must be bit-identical");
    print_final(&log);
    svg::write("figures/ch02_tour.svg", &log);
}
```

```
elbow (0.7071, 0.7071)  tip (0.0000, 1.4142)
intersects: false  distance: 0.0607
witness a*=(0.4571, 0.9571) b*=(0.5000, 1.0000)
distance: 0.8900
```

Every line of that output is locked by a test in the crate and by a check in the browser library:

RUST `crates/robots/tests/worked_example.rs`

```rust
#[test]
fn fk_matches_choset_example() {
    let tip = Reach::<2>::choset().tip(&Tn([FRAC_PI_4, FRAC_PI_2]));
    assert_abs_diff_eq!(tip.x, 0.0, epsilon = 1e-12);
    assert_abs_diff_eq!(tip.y, SQRT_2, epsilon = 1e-12);
}

#[test]
fn reach_distance_to_square() {
    let (scene, fp) = worked_scene();
    assert_abs_diff_eq!(scene.distance(&fp), 1.5 / SQRT_2 - 1.0, epsilon = 1e-9);
    let w = scene.witness(&fp).unwrap();
    assert_abs_diff_eq!(w.on_obstacle, point![0.5, 1.0], epsilon = 1e-6);
    assert_abs_diff_eq!(w.on_robot, point![0.4571, 0.9571], epsilon = 1e-4);
}

#[test]
fn rusty_wall_distance() { assert_abs_diff_eq!(wall_scene().distance(&rusty_at(2.0, 1.0)),
    0.89, epsilon = 1e-12); }

#[test]
fn ik_roundtrip() { /* 1,000 seeded q: fk(ik(fk(q))) == fk(q) to 1e-9, both elbows */ }

#[test]
fn swept_catches_midpoint() {
    // Two free configurations whose straight sweep passes link 1 through the post.
    let c = ChainChecker::new(Scene::workbench(), Reach::<2>::workbench());
    let (a, b) = (Tn([2.0, 0.0]), Tn([3.1, 0.0]));
    assert!(c.is_free(&a) && c.is_free(&b));
    assert!(!c.edge_free(&a, &b));
}

#[test]
fn tour_replays_bitwise() {
```

```
    let (a, b) = (Run::record(TourConfig::default(), BOOK_SEED), Run::record(TourConfig::default(), BOOK_SEED));
    assert_eq!(a.frames, b.frames); // every f64 equal exactly, not approximately
}

#[test]
fn scan_is_rho_r() {
    // min over 360 rays from (2, 1) equals the point distance to the wall at x = 3.
    let scan = RangeSensor { n_rays: 360, max_range: 4.0 }.scan(&wall_scene(), point![2.0, 1.0]);
    assert_abs_diff_eq!(scan.iter().cloned().fold(f64::INFINITY, f64::min), 1.0, epsilon = 1e-9);
}
```

2.6 Putting it together

The integration lab is the tour itself. Sixty seconds at 20 Hz, three robots, one seed: Rusty pursues a loop of waypoints out of room A, down the corridor, into room C and back, with the sister book's wheel slip as the run's one stochastic subsystem; Reach follows two incommensurate sinusoids whose amplitudes, frequencies, and phases the seed chose; Hitch runs a six-leg reversing script with the seed's steering angles. Every tick, each robot's footprint goes to its own `Collision` implementation; a contact flashes, is counted, and — for Rusty and Hitch, whose motion is integrated — holds the body where it is while the wheels keep turning, which is exactly what a stuck robot's encoders do.

Run it twice and compare every frame. The library check `tour_replays_bitwise` does: 1,200 frames, every coordinate equal to the last bit, and a different seed gives a different log. After the full minute at `BOOK_SEED = 2005` the browser port reports

```
Rusty  Pose2 { x: 6.212, y: 4.466 }        contacts: 0
Reach  T2 { th1: 0.186, th2: 1.324 }       contacts: 9
Hitch  Pose2 { x: 6.245, y: 1.351, theta: 0.272 }   contacts: 12
```

and the check `tour at BOOK_SEED` pins those digits. Press re-roll on the dashboard and every one of them changes; type the seed back and every one of them returns. Rusty's zero contacts are not luck: the waypoint controller closes its loop on ground truth, which — as the sister book is careful to say wherever it uses the same trick — is a quantity no real robot has.

One honesty item about the two implementations. The geometry agrees across them to the digit, and each replays itself bit for bit. The *tours* do not agree across them, because `SmallRng` and the browser's generator draw different numbers from the same seed, and Rusty's slip and the scripts' parameters come from those draws. The claim this book makes is the one it can test: that each implementation is a pure function of its seed, and that both compute the same geometry.

2.6.1 The anatomy of a widget

Every interactive figure in this book is built from the same parts, fixed here and never renegotiated. A `WidgetFrame` carries the id from the chapter's design manifest, the title, one line naming the misconception the widget kills, and a color key. Inside it, one or more `SimCanvas` instances draw in metres and radians and let the framework map to pixels, resize, and re-read the palette when the reader flips theme. A `useSimulation` hook runs a fixed-rate clock from a seed; `Transport` shows play, step, reset, re-roll, and the seed itself, because reproducibility is part of what is being taught. Colors are never named in a widget — they come from the palette the canvas hands to `draw`, which is how hovering a tinted term in an equation can fade every other role on the page without any widget knowing the feature exists. Each widget autoplays a seeded default, foregrounds one parameter, and hides the rest behind a disclosure. The frame also carries a `<noscript>` slot for a static description, so a reader without JavaScript is told what the simulation would have drawn rather than shown a blank.

Which chapter opens which box: Chapter 3 builds the Bug algorithms on `RangeSensor` and the Apartment; Chapter 4 opens `collide` and adds Reach's Jacobian beside the forward kinematics defined here; Chapter 5 implements `Manifold` *for* the configuration types defined here; every planner from Chapter 7 to Chapter 21 takes a `&dyn Collision` and a robot from `robots`; and Appendix F is this section, expanded into a reference.

2.7 Exercises

1 FOUNDATION •• **Inverse kinematics for general link lengths**

Derive Reach's inverse kinematics for general L_1, L_2 using the law of cosines and atan2 (Choset problem 20). State the reachable annulus and show exactly which tip positions have one solution rather than two. Then explain why `Reach::ik` returns `[Option<T2>; 2]` rather than `Option<[T2; 2]>` — what does it mean for exactly one entry to be `None`?

HINTS

<>Place the triangle: origin, elbow, tip, with sides L_1, L_2, $\|x\|$. The law of cosines at the elbow gives $\cos\theta_2$; a cosine has two preimages in $(-\pi, \pi]$.</> <>The two preimages coincide when $\cos\theta_2 = \pm 1$, i.e. on the annulus boundary — both circles. Inside, two; on the boundary, one; outside, none.</> <>With the library's convention the two entries are both `Some` or both `None`, so a single `None` cannot occur — the type is more permissive than the mathematics. Chapter 5 revisits this when `Manifold` gives the annulus boundary its own treatment.</>

2 FOUNDATION •• **R(q) for a car and trailer**

Write $R(q)$ as a set for Hitch with the trailer: show it is the union of two rectangles whose poses are functions of (x, y, θ, ψ), and give those two poses explicitly in terms of the wheelbase, the overhangs, and d_h. Then explain why ψ is a configuration variable but the steering angle ϕ is not. (The compiler already knows: ϕ lives in `CarCmd`, not in `HitchCfg`.)

3 CONCEPTUAL •• **Predict a witness, then watch it jump**

Put Reach at $q = (0, \pi/2)$ and imagine the square obstacle of side 0.2 centered on $(1, 1)$.

PREDICT FIRST

What does the distance query return for this configuration?

- About 0.1 — the half-side of the square, measured from its edge to the tip
  That would be the distance from the square's center to its edge. But the tip is not outside the square: $\phi(0, \pi/2) = (1 + \cos 90°, \sin 90°) = (1, 1)$, which is the square's center.
- ✓ Exactly 0 — the configuration is in collision
  The elbow is at (1, 0) and the tip at (1, 1), the center of the square. Link 2 ends inside the obstacle, so intersects is true, the distance is 0, and the witness is a contact point rather than a nearest pair.
- About 0.9 — the distance from the elbow to the square
  The query minimizes over the whole footprint, not over the joints. Link 2 reaches the square even though the elbow is far from it.

Now slide θ_2 to $\pi/2+0.3$. The tip swings to about $(0.70, 0.96)$, outside the square, and link 2 runs from $(1, 0)$ past the square's lower-left corner. The nearest obstacle point is that corner, $(0.9, 0.9)$, at a distance you can compute by projecting the corner onto the link.

WORK IT OUT

Distance from link 2 to the square at q = $(0, \pi/2 + 0.3)$, in metres

Answer. 0.1704 m Link 2's direction is $(-\sin 0.3, \cos 0.3) \approx (-0.2955, 0.9553)$. The corner $(0.9, 0.9)$ minus the elbow $(1, 0)$ is $(-0.1, 0.9)$, whose projection onto the link is at parameter $0.889 \in [0, 1]$ — so the foot lies on the link — giving the foot $(0.7373, 0.8493)$ and a distance of 0.1704. The library check `exercise 3` pins it, with $b^* = (0.9, 0.9)$. Keep sliding toward π and watch for the moment the nearest feature stops being that corner.

4 CONCEPTUAL •• **One metre to the left**

In the dashboard, turn the trailer on and try to make Hitch's trailer end up exactly one metre to the left of where it started, with the same heading. Describe the maneuver you find and count its direction reversals.

HINTS

<>A pure translation sideways has zero lateral velocity at every instant — and the model has no input for that.</> <>Steer left and drive forward, then steer right and drive backward. Where did the body end up relative to where it started?</> <>Zero reversals is impossible; Chapter 20 proves it with the Lie bracket of the drive and steer fields, and Chapter 21 computes the minimal maneuver.</>

5 PRACTICAL •• **Redundancy: a tip that does not move**

Add `Reach<3>` with `lengths = [0.6, 0.5, 0.4]` to the dashboard's Workbench pane and show that the tip can stay fixed while the arm moves (redundancy, Choset §3.3). Write a test that takes 200 seeded configurations with the same tip — generate them by a one-dimensional walk in the null space of the tip map, re-solving the last two joints by the 2R inverse kinematics after each step of θ_1 — and asserts `tip` constant to 10^{-9}.

6 PRACTICAL ••• **Lie to the Bug (stretch)**

Replace the exact `RangeSensor` by the sister book's noisy LiDAR port (`lib/sim/rusty.ts`, `raycastScan`) in the Bug1 stub of Chapter 1 and measure, over 100 seeds, how often the leave point is misjudged. Write one paragraph on why Chapter 3's guarantee needs the ideal sensor, and what a real robot does instead (the sister book's Chapter 10 is the pointer).

2.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

§3.1 for the circular robot's R(q) and the arm's torus; §3.8, Example 3.8.1, for the forward kinematics and its worked numbers; Appendix F.5 for GJK, the name of what the black box does. `link`

Gilbert, E. G., Johnson, D. W., and Keerthi, S. S. (1988). A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation 4(2), 193–203.*

The GJK distance algorithm, Choset's Algorithm 23: the iteration inside every distance and witness query the Rust crate delegates to parry2d. `doi:10.1109/56.2083`

Lozano-Pé rez, T. (1983). Spatial Planning: A Configuration Space Approach. *IEEE Transactions on Computers C-32(2), 108–120*.
Where the disc's one-line Minkowski inflation comes from, and where Chapter 4 picks it up for polygons. doi:10.1109/TC.1983.1676196

Pan, J., Chitta, S., and Manocha, D. (2012). FCL: A general purpose library for collision and proximity queries. *IEEE International Conference on Robotics and Automation (ICRA), 3859–3866*.
The collision library behind MoveIt and OMPL's default checkers, with the same four-query vocabulary — intersection, distance, closest points, continuous collision — that this chapter's Collision trait adopts. doi:10.1109/ICRA.2012.6225337

Şucan, I. A., Moll, M., and Kavraki, L. E. (2012). The Open Motion Planning Library. *IEEE Robotics & Automation Magazine 19(4), 72–82*.
A planner that takes a state validity checker and a state space as abstract interfaces: the architecture this chapter's Collision trait object and typed configurations mirror. doi:10.1109/MRA.2012.2205651

Dimforge (2024). parry: 2D and 3D collision-detection library for the Rust programming language. *Documentation and source*.
The crate behind crates/collide: shapes, the Bvh broad phase, and query::{distance, closest_points, intersection_test, cast_shapes}. Appendix C documents what it does. link

Thrun, S., Burgard, W., and Fox, D. (2005). Probabilistic Robotics. *MIT Press*.
The source of Rusty's differential-drive model, encoders, and LiDAR, ported here from the sister book's Chapter 4 and never re-derived. link

CHAPTER 3

Bug Algorithms: Planning with a Contact Sensor

FOUNDATIONS — ROBOTS, WORLDS, AND CONFIGURATION SPACE · Foundational · 55 min

> *These algorithms require two behaviors: move on a straight line and follow a boundary.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), Chapter 2*

Choset opens *Principles of Robot Motion* with the Bug algorithms because they prove something surprising with almost no mathematics. A point robot that knows only where the goal is, how far it has walked, and whether it is touching something can be *guaranteed* to reach any reachable goal, with a path whose length is bounded before it sets off. No map, no search, no configuration space yet — just two behaviors and a rule for switching between them.

This chapter keeps that order of business and sharpens it. The claim to carry away is this: **a planner is two behaviors and a switching rule, and the whole difference between a heuristic and a theorem is the switching rule.** Bug1 switches exhaustively — circumnavigate, then leave from the best point — and pays for its certainty. Bug2 switches greedily — leave the first time the fixed m-line is closer — and can be made arbitrarily worse than Bug1 by an obstacle shaped like a comb. Tangent Bug buys shorter paths with a range sensor by *seeing* the discontinuities of the raw distance function and heading for them.

Underneath all three sits a problem the rest of the book keeps meeting: how do you follow a curve you have not been given? Choset's answer — treat the offset curve as the zero set of $G(x) = D(x) - W^*$ and trace it with a predictor–corrector justified by the implicit function theorem — becomes this chapter's owned artifact, `trace_curve`. Chapter 7 uses it to follow level sets, Chapter 8 the Voronoi diagram, Chapter 9 the sensor-based GVG, and Chapter 10 the slices of a

coverage decomposition.

AFTER THIS CHAPTER YOU CAN

- State Bug1, Bug2 and Tangent Bug precisely, as two behaviors and a switching rule each
- Prove the Bug1 bound $L \leq d + 1.5 \Sigma p_i$ and the Bug2 bound $L \leq d + \frac{1}{2} \Sigma n_i p_i$, and build the obstacle on which greedy Bug2 loses
- Read a range scan the way Tangent Bug does: intervals of continuity, their endpoints O_i, and the heuristic that chooses among them
- Derive $D(x)$ and ∇D from range data, and the tangent $(\nabla D)^\perp$ of the offset curve
- Trace the offset curve $G(x) = D(x) - W^* = 0$ with a predictor–corrector, and say exactly which theorem guarantees each half of it
- Implement all of it in Rust, with the worked example reproduced to machine precision and both bounds property-tested

ASSUMED

- Bug1 informally, from Chapter 1
- The Apartment, rayCast and the point/disc reduction of Chapter 2
- Gradients and the chain rule; the implicit function theorem is restated here

3.1 A blindfolded walker with one hand on the wall

Imagine walking across an unfamiliar room blindfolded. You know where the door is — you can point at it — and you can tell when your outstretched hand touches a wall. Nothing else. Could you promise to reach the door?

You could. Walk straight toward it. When you touch something, keep one hand on it and walk all the way around, noting as you go the point where you were closest to the door. Come back to that point, step off toward the door, and repeat. If the point of the wall closest to the door is one from which stepping toward the door walks you straight back into the wall, the door is on the other side of a closed room and you can stop: it is not reachable. That is Bug1, and it is a complete planner.

Three robots run that idea below, on the same map, toward the same goal. The first is Bug1. The second, Bug2, is impatient: it leaves the wall the first time it gets back to the straight line from start to goal at a point closer than where it hit. The third, Tangent Bug, carries a range sensor and heads for the corners it can see.

INTERACTIVE · w3.1

Bug Arena

Greedy is faster on average and unboundedly slower in the worst case — the switching rule is the whole difference between a heuristic and a theorem.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch03-bug-algorithms#w3.1

Watch the first race through. Bug1 walks 34 units, Bug2 walks 16, and the counters stop against their bounds — 37 and 28 — with room to spare. Greedy wins by a wide margin, and every intuition says it should: why walk all the way around an obstacle when the far side is right there?

Then the map changes to a spiral, Choset's figure 2.4 made concrete, and Bug2 turning right walks 132 units to Bug1's 125. Not because the spiral is large — Bug1 pays for it once — but because the m-line crosses the spiral's boundary ten times, and almost every crossing is on the *wrong side*: stepping toward the goal from it would re-enter the wall. Bug2 can only leave at a crossing that is closer than its hit point *and* free, and with its fixed turn direction it has to walk nearly the whole boundary it has already seen to find the next one. Flip the turn direction and Bug2 wins again, at 52. The algorithm did not change. The obstacle did, and with it the cost of being greedy.

THE ONE SENTENCE VERSION

A sensor-based planner is two behaviors — move toward the goal, follow the boundary — and a rule for switching between them. Bug1's rule is exhaustive and buys a bound that holds everywhere; Bug2's is greedy and buys speed on simple obstacles at the price of an unbounded worst case; Tangent Bug's is informed by range data and buys shorter paths without a proof of completeness at all.

3.2 Building intuition

3.2.1 Two behaviors

Every Bug in this chapter has exactly two behaviors.

Motion-to-goal is gradient descent on the distance to a point: toward q_{goal}, or, for Tangent Bug, toward whichever sensed point n minimizes $d(x, n) + d(n, q_{\text{goal}})$. It is the behavior you would write first, and it is the one that fails first — it stops the moment it touches anything.

Boundary-following is the behavior that repairs it. The robot keeps the obstacle at a fixed distance and walks along it. It sounds trivial — "keep your hand on the wall" — and the second half of the chapter is about why it is not: the

robot does not know the wall's shape, it only knows the nearest point and the direction to it, and from those two facts it has to *construct* the curve as it goes. That construction is root-finding, and it has hypotheses.

What distinguishes the three algorithms is only *when they switch*. Watch the arena with that lens: the orange path is identical for all three until the first hit point; after that the hit and leave points — the switches — are the whole story.

3.2.2 What a range sensor sees

Tangent Bug replaces the contact sensor with a ring of range finders: for each bearing θ it knows $\rho(x, \theta)$, the distance to the first obstacle along that ray, saturated at the sensor's range R. The sensor does not return a map. It returns a function of one angle, and the only things in that function worth planning on are its *discontinuities*: the bearings where one obstacle ends and empty space (or a farther obstacle) begins. Those are the corners of what the robot can see, and in the figure below they are the points O_i.

> **INTERACTIVE · w3.3**
>
> **Tangent Scope**
>
> A range sensor does not give you the map. It gives you a few discontinuities — and that is enough to plan with, until it is not.
>
> Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch03-bug-algorithms#w3.3`

Two things the scope makes visible. First, the heuristic $d(x, O_i) + d(O_i, q_{\text{goal}})$ is a guess about a path it cannot see: "if I go to that corner, how far is the goal from there as the crow flies?" Second, that guess can be wrong in a way the robot can sometimes detect and sometimes cannot. In Choset's figure 2.8, loaded by default, the corner that looks best with a 3 m sensor is one from which the straight line to the goal cuts through a second obstacle. With a 6 m sensor the robot can see that obstacle, assigns the corner infinite cost, and chooses differently. More range is more information; it is never a map.

3.2.3 The curve you were not given

Boundary following at a safe distance W^* means walking along the set of points whose distance to the nearest obstacle is exactly W^* — the *offset curve*. The robot can measure that distance $D(x)$ (it is the shortest range in its scan) and its gradient ∇D (the direction of that shortest ray, reversed). It does not know the curve. The widget in the algorithm section below shows how it constructs the curve anyway: predict a step along the tangent, fall off, correct back with Newton's method, repeat — and what happens when the curve it is chasing has a cusp.

3.3 The mathematics

q_i^H, q_i^L	The i-th hit point and leave point.
m-line	Bug1: the segment from the last leave point q^L_{i-1} to the goal. Bug2: the fixed segment from q_start to q_goal.
p_i, n_i	Perimeter of WO_i; number of times the Bug2 m-line crosses the boundary of WO_i.
L_{Bug1}, L_{Bug2}	Path lengths.
$B_r(x)$	The open ball of radius r about x. The workspace is bounded: W ⊂ B_r(x) for some finite r.
$\rho(x,\theta)$, $\rho_R(x,\theta)$	Raw distance along the ray from x at bearing θ (Choset eq. 2.3), and its saturated version: ∞ when $\rho \geq$ R.
O_i, T	Endpoints of the intervals of continuity of ρ_R(x, ·); the point of the sensing circle on the segment x q_goal.
d_{reach}, d_{followed}	Tangent Bug's two bookkeeping distances: goal distance of the closest visible point of the followed obstacle, and of the closest point sensed so far.
$D(x)$, $\nabla D(x)$	Distance from x to the nearest obstacle (eq. 2.4) and its gradient, the unit vector along the minimizing ray, away from the obstacle.
W^*	The safety offset of the followed curve.
$G(x) = D(x) - W^*$, DG, $D_Y G$	The offset curve's defining function, its differential, and the differential on the correcting line Y.
$\Delta\lambda$	The predictor step along the tangent.

3.3.1 Definitions

DEFINITION: CONTACT SENSOR, RAW DISTANCE, SATURATED RAW DISTANCE

A **contact (tactile) sensor** reports whether the point robot touches $\partial\mathcal{WO}_i$; it is a range sensor with $R = 0$. The **raw distance function** (Choset eq. 2.3) is

$$\rho(x,\theta) \;=\; \min\{\, \lambda \geq 0 \;:\; x + \lambda\,[\cos\theta, \sin\theta]^{\mathsf{T}} \in \textstyle\bigcup_i \mathcal{WO}_i \,\},$$

the distance to the first obstacle along the ray at bearing θ. The **saturated** version is $\rho_R(x,\theta) = \rho(x,\theta)$ when $\rho(x,\theta) < R$ and ∞ otherwise.

DEFINITION: INTERVAL OF CONTINUITY AND ITS ENDPOINTS

For fixed x, an **interval of continuity** is a connected set of boundary points $x + \rho(x,\theta)[\cos\theta, \sin\theta]^{\mathsf{T}}$ over which $\rho_R(x,\cdot)$ is finite and continuous in θ. Its endpoints O_i arise in two ways: one obstacle occludes another — the

ray is tangent to the nearer one — or the boundary leaves sensing range.

DEFINITION: DISTANCE TO THE NEAREST OBSTACLE AND THE OFFSET CURVE

$D(x) = \min_{c \in \bigcup_i \mathcal{WO}_i} d(x, c)$ (Choset eq. 2.4). Wherever the minimizer c^* is unique, D is differentiable and $\nabla D(x) = (x - c^*)/\|x - c^*\|$: the unit vector pointing from the nearest obstacle point to x. The **offset curve** at distance W^* is $\{x : D(x) = W^*\} = G^{-1}(0)$ with $G = D - W^*$; its tangent at x is $(\nabla D(x))^\perp$.

DEFINITION: FOLLOWED AND BLOCKING OBSTACLE

Tangent Bug's **followed obstacle** $\mathcal{WO}_f$ is the one whose boundary it is tracking; the **blocking obstacle** is the nearest in-range obstacle intersecting the segment from x to q_{goal}. When boundary-following begins they coincide. With $\Lambda = \{y \in \partial\mathcal{WO}_f : \text{the segment } xy \text{ is free}\}$ the visible part of the followed obstacle, $d_{\text{reach}} = \min_{c \in \Lambda} d(q_{\text{goal}}, c)$.

3.3.2 Bug1 is complete, and its path is bounded

$$L_{\text{Bug1}} \leq d(q_{\text{start}}, q_{\text{goal}}) + 1.5 \sum_{i=1}^{n} p_i \tag{2.1}$$

Statement. In a bounded workspace with finitely many obstacles, Bug1 reaches q_{goal} whenever a path exists, otherwise halts with "unreachable" in finite time, and its path obeys (2.1).

DERIVATION **Bug1 is complete and L ≤ d + 1.5 Σ p_i (Choset eq. 2.1)**

$L_{Bug1} \leq d(q_{start}, q_{goal}) + 1.5 \sum_i p_i$

Step 1 — leave points get strictly closer. The leave point q_i^L is, by construction, the perimeter point of $\mathcal{WO}_i$ closest to the goal. The hit point q_i^H is also on that perimeter, so $d(q_i^L, q_{\text{goal}}) \leq d(q_i^H, q_{\text{goal}})$, with equality only if the hit point *is* the closest perimeter point. Motion-to-goal from q_i^L moves along the straight line to the goal, so every later point of the path is closer to the goal than q_i^L — unless the line immediately re-enters $\mathcal{WO}_i$, which is exactly the failure test on line 13.

Step 2 — no obstacle is hit twice. Suppose $\mathcal{WO}_i$ were hit again later, at q_j^H. By Step 1, $d(q_j^H, q_{\text{goal}}) < d(q_i^L, q_{\text{goal}})$. But $q_j^H \in \partial\mathcal{WO}_i$ and q_i^L was

the *closest* point of $\partial\mathcal{WO}_i$ to the goal — contradiction. Hence at most n circumnavigations occur, and the algorithm terminates: each loop is a finite walk, and there are finitely many.

Step 3 — the cost of one obstacle. A full circumnavigation costs exactly p_i. Returning to q_i^L the shorter way round costs at most $p_i/2$ — the robot knows the perimeter it just walked and the arc length at which it recorded the best point, so it can choose. Total per obstacle: at most $1.5\,p_i$.

Step 4 — the straight-line budget. Every motion-to-goal segment moves straight toward the goal from a point closer to it than the start of the previous segment (Step 1). Laid end to end, those segments total at most $d(q_{\text{start}}, q_{\text{goal}})$: each one spends goal distance, and there is only $d(q_{\text{start}}, q_{\text{goal}})$ of it to spend.

Step 5 — sum. $L_{\text{Bug1}} \le d(q_{\text{start}}, q_{\text{goal}}) + \sum_i 1.5\,p_i$. The sum ranges over *all* obstacles, including ones the robot never meets, which is why (2.1) is an upper bound and not an estimate.

Step 6 — the failure test is sound. If the robot stands at q_i^L — the perimeter point closest to the goal — and a step toward the goal re-enters $\mathcal{WO}_i$, then the goal direction points *into* the obstacle at the point of the boundary nearest the goal. That happens only if the goal is enclosed by $\mathcal{WO}_i$: otherwise there would be a boundary point even closer to the goal, on the far side. So Bug1 reports "unreachable" only when it is. ■

Why grazing needs no following. If the line to the goal merely touches $\partial\mathcal{WO}_i$ at a point, the robot can continue: the contact is a point of tangency, there is no direction in which the obstacle blocks the goal, and Choset says so explicitly (p. 18). Our implementation treats a move collinear with an edge as a miss for the same reason.

In the worked example the slack is exactly 3: the bound budgets the straight-line distance 10 but the robot never walks the 2 + 1 units of m-line *inside* the two rectangles. The return legs (4 and 5) are the worst cases $p_i/2$ — both obstacles are symmetric about the m-line, so both ways round are equal.

3.3.3 Bug2's bound, and why greed can lose

$$L_{\text{Bug2}} \;\le\; d(q_{\text{start}}, q_{\text{goal}}) \;+\; \tfrac{1}{2}\sum_{i=1}^{n} n_i\, p_i \tag{2.2}$$

Statement. L_{Bug2} obeys (2.2), where n_i is the number of times the line through q_{start} and q_{goal} crosses $\partial\mathcal{WO}_i$; and for every K there is a workspace with $L_{\text{Bug2}} > K \cdot L_{\text{Bug1}}$.

DERIVATION **Bug2's bound and the comb that defeats it (Choset eq. 2.2)**

$L_{Bug2} \le d + \frac{1}{2}\sum_i n_i p_i$, and L_{Bug2}/L_{Bug1} is unbounded

Step 1 — candidates. Bug2 may leave $\mathcal{WO}_i$ only at a point of the fixed m-line that lies on $\partial\mathcal{WO}_i$. The m-line crosses that boundary n_i times, so there are at most n_i candidate leave points on $\mathcal{WO}_i$ in the whole run.
Step 2 — half of them are on the wrong side. The m-line enters and exits the obstacle alternately along its length. At an *entry* crossing, stepping toward the goal re-enters the obstacle, so the third leave condition fails: at most $n_i/2$ crossings are valid leave points.
Step 3 — each leave may cost nearly a perimeter. Between hitting $\mathcal{WO}_i$ and reaching a valid leave point the robot follows the boundary in its fixed turn direction. In the worst case the next valid crossing is almost a full loop away: nearly p_i per leave, and $n_i/2$ leaves, for a total of at most $\frac{1}{2}n_i p_i$ on $\mathcal{WO}_i$.
Step 4 — add the straight-line budget. Motion-to-goal runs along the m-line, so its segments total at most $d(q_{\text{start}}, q_{\text{goal}})$. Summing gives (2.2). As with (2.1), the sum is over all obstacles, so this is an upper bound.
Step 5 — the comb. Nothing in (2.2) stops n_i from being large. Take a boundary that the m-line crosses n_1 times — the spiral in the arena, where $n_1 = 10$ — arranged so that from every hit point the turn direction leads first past wrong-side crossings and only then to a valid one. Bug2 pays close to a perimeter per crossing, roughly $n_1 p_1/2$ in total, while Bug1 pays p_1 once and at most $p_1/2$ to walk back: $L_{\text{Bug2}}/L_{\text{Bug1}} \gtrsim n_1/3$, which grows without bound as the comb grows more teeth. ■
The exhaustive-versus-greedy reading (Choset p. 22). For each obstacle Bug1 performs an *exhaustive* search for the best leave point — it looks at the entire perimeter and is certain to have found the optimum. Bug2 is *opportunistic*: it commits to the first leave point better than its hit point. When obstacles are simple the greedy choice pays off at once; when they are complex the conservative choice wins. This is the book's first instance of a trade-off that returns, dressed as best-first versus uniform-cost search, in Chapter 6.
What the spiral actually does. Our spiral has one obstacle of perimeter $p_1 = 85.2$ and the m-line crosses it $n_1 = 10$ times, so (2.2) promises $L_{\text{Bug2}} \le 6.5 + 426 = 432.5$ and (2.1) promises $L_{\text{Bug1}} \le 6.5 + 127.8 = 134.3$. The runs come in at 132.4 and 124.9. Both bounds hold; one of them is tight to within eight percent and the other is a factor three loose, which is the honest picture of what (2.2) is: a statement about the worst arrangement of crossings, not a prediction.

3.3.4 Distance is the minimum of the rays, and the gradient is the minimizing ray

Statement (Choset problem 2.1). $D(x) = \min_s \rho(x, s)$. Wherever the minimizer s^* is unique, $\nabla D(x) = -[\cos s^*, \sin s^*]^\mathsf{T}$.

DERIVATION **D(x) = min$_s$ ρ(x, s) and ∇D is the minimizing ray (Choset eq. 2.4)**

$D(x) = \min_s \rho(x, s), \quad \nabla D(x) = -[\cos s^*, \sin s^*]^\mathsf{T}$

Step 1 — every obstacle point is on some ray. For $c \in \bigcup_i \mathcal{WO}_i$, let s be the bearing of $c - x$. The first obstacle point along that ray is at distance $\rho(x, s) \le d(x, c)$, so $\min_s \rho(x, s) \le D(x)$.
Step 2 — every ray's first hit is an obstacle point. $x + \rho(x, s)[\cos s, \sin s]^\mathsf{T}$ lies in some $\mathcal{WO}_i$ by definition of ρ, so $D(x) \le \rho(x, s)$ for every s, hence $D(x) \le \min_s \rho(x, s)$. With Step 1, equality.
Step 3 — the envelope argument. Write $D(x) = \min_{c \in C} \|x - c\|$ over the closed set $C = \bigcup_i \mathcal{WO}_i$. Where the minimizing c^* is unique, the minimum is attained by a single smooth function of x in a neighborhood, and the derivative of a minimum of smooth functions is the derivative of the active one: $\nabla D(x) = (x - c^*)/\|x - c^*\|$. That unit vector points from c^* to x, i.e. *away* from the obstacle along the minimizing ray — $-[\cos s^*, \sin s^*]^\mathsf{T}$. ■
Non-uniqueness. Where two obstacle points tie for nearest, D is continuous but not differentiable: ∇D jumps. That set — the points equidistant from two or more obstacles — is the *medial axis*, and Chapter 8 builds the generalized Voronoi diagram out of exactly it. For this chapter it is where the tracer's hypotheses fail, and the widget below walks into it on purpose.
What a finite scan gives you. With n rays the minimum is attained on a grid of bearings, so the estimate never undershoots D and overshoots it by $D(1 - \cos(\pi/n))$ on a flat wall — 10^{-5} of D with 720 rays. Near a corner seen edge-on no finite set of rays can see the nearest point, and the estimate can be badly high. The self-check prints both facts: at 90 % of a thousand seeded Apartment points the 720-ray estimate is within 0.11 % of D and ∇D within 0.19°, and the worst case is off by 65 %. Problem 2.1 is a statement about infinite resolution.

3.3.5 The offset curve is a curve where $D_Y G$ is invertible

Statement (Choset §2.3.3 and Theorem D.1.1). Write $x = (\lambda, y)$ in coordinates along the tangent and the normal of the offset curve at a point x_0 with $G(x_0) = 0$. If $D_Y G(x_0) \neq 0$, there is a neighborhood of x_0 in which the zero set of G is the graph of a unique smooth function $y(\lambda)$.

DERIVATION **The offset curve is locally the graph y(λ) (implicit function theorem, App. D.1)**

$G(\lambda, y(\lambda)) = 0$ near x_0 whenever $D_Y G(x_0) \neq 0$

Theorem D.1.1 (Implicit Function Theorem). Let $f : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^n$ be smooth and suppose $D_y f(x_0, y_0)$ is invertible. Then there exist neighborhoods X_0 of x_0 and Z_0 of $f(x_0, y_0)$ and a unique smooth $g : X_0 \times Z_0 \to \mathbb{R}^n$ with $f(x, g(x, z)) = z$.

Step 1 — D is smooth off the medial axis with $|\nabla D| = 1$. By the previous derivation, ∇D is a unit vector wherever the nearest point is unique, and D is as smooth as the obstacle boundary there.

Step 2 — the differential in the (λ, y) frame. Along the tangent λ the distance does not change to first order, so $\partial G/\partial\lambda = 0$; along the normal y it changes at unit rate, so $\partial G/\partial y = \pm 1$. Hence $DG = [\,0 \quad D_Y G\,]$ with $D_Y G = \pm 1 \neq 0$ on the curve.

Step 3 — apply the theorem with $m = n = 1$. Take $f(\lambda, y) = G(\lambda, y), z = 0$. The hypothesis is $D_Y G \neq 0$, which Step 2 gives; the conclusion is a unique smooth $y(\lambda)$ with $G(\lambda, y(\lambda)) = 0$ near x_0.

Step 4 — the graph is the curve. Points of the zero set near x_0 are exactly the graph points, by the uniqueness clause. So the offset curve is, locally, a smooth curve parameterized by arc length along the tangent — the object the predictor–corrector traces. ■

The general version Chapters 8–9 use. For $G : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^n$ — equidistance to several obstacles at once — the same theorem produces an m-dimensional zero set wherever $D_Y G$ has full rank. And because the nonsingular matrices form an open set, full rank at a point persists in a neighborhood: Choset's reason (p. 37) that the corrector can be run at every step, not just on the curve.

3.3.6 The corrector converges quadratically

Statement (Theorem D.2.1). Let $f : \mathbb{R}^n \to \mathbb{R}^n$ with $f(y^*) = 0$. Suppose $Df(y^*)$ is nonsingular with $\|(Df(y^*))^{-1}\| \leq \beta$, and Df is γ-Lipschitz on $B_\rho(y^*)$ with $\gamma \leq 2/(\rho\beta)$. Then from any $y_0 \in B_\rho(y^*)$ the sequence $y_{h+1} = y_h - (Df(y_h))^{-1} f(y_h)$ stays in the ball and converges quadratically: $\|y_{h+1} - y^*\| \leq a \|y_h - y^*\|^2$ with $a = \beta\gamma/(2(1 - \rho\beta\gamma)) < 1/\rho$.

DERIVATION **Newton–Raphson converges quadratically (App. D.2), and what it means for eq. 2.5**

$\|y_{h+1} - y^*\| \leq a \|y_h - y^*\|^2$

Step 1 — Taylor about y_h. Since $f(y^*) = 0$, $0 = f(y^*) = f(y_h) + Df(y_h)(y^* - y_h) + r_h$ with remainder $\|r_h\| \le \frac{\gamma}{2}\|y^* - y_h\|^2$ by the Lipschitz bound on Df.
Step 2 — subtract the Newton step. $y_{h+1} - y^* = y_h - y^* - (Df(y_h))^{-1} f(y_h) = (Df(y_h))^{-1}\left[Df(y_h)(y_h - y^*) - f(y_h)\right] = (Df(y_h))^{-1} r_h$.
Step 3 — bound the inverse. On $B_\rho(y^*)$, $\|Df(y_h) - Df(y^*)\| \le \gamma\rho$, and a standard perturbation bound gives $\|(Df(y_h))^{-1}\| \le \beta/(1 - \rho\beta\gamma)$ — finite because $\gamma \le 2/(\rho\beta)$ keeps the denominator positive. Multiply: $\|y_{h+1} - y^*\| \le \frac{\beta}{1-\rho\beta\gamma} \cdot \frac{\gamma}{2}\|y_h - y^*\|^2 = a\|y_h - y^*\|^2$, and $a\rho < 1$ keeps every iterate inside the ball. ■
The specialization in the tracer (eq. 2.5). On the correcting line Y through the predicted point, $f = G$ restricted to Y is a scalar function of one variable, and $Df = D_Y G = \nabla G \cdot n$ with n the unit normal of the line — a number the robot computes from the distance gradient, i.e. from its sensor. The Newton step becomes

$$y_{h+1} \;=\; y_h \;-\; \frac{G(y_h)}{D_Y G(y_h)}\, n. \tag{2.5}$$

What quadratic looks like. On the kidney obstacle of the widget, at $W^* = 0.5$ with a long predictor step $\Delta\lambda = 0.4$, the residuals $|G(y_h)|$ run $3.96 \times 10^{-2} \to 6.47 \times 10^{-5} \to 2.02 \times 10^{-10} \to 5.6 \times 10^{-17}$. The ratios $|G(y_{h+1})|/|G(y_h)|^2$ are 0.041 and 0.048 — the same constant a twice, which is what "quadratic" means and what the self-check asserts.
When $D_Y G \to 0$. $\beta = 1/|D_Y G|$ blows up, the admissible γ shrinks to nothing, and the theorem promises nothing. Our tracer refuses to divide: below $|D_Y G| < 0.2$ it stops and reports `singular`. The widget shows the moment.

3.3.7 The length of a convex obstacle's offset curve

Statement. For a convex polygon of perimeter p, the offset curve at distance W^* has length $p + 2\pi W^*$.

DERIVATION **Offset length p + 2πW* (the Steiner formula), the tracer's test**

$\text{len}\{D = W^*\} = p + 2\pi W^*$

Step 1 — edges translate. Each edge of the polygon contributes a parallel segment of the same length at distance W^*: total p.
Step 2 — vertices become arcs. At each vertex the offset curve is a circular arc of radius W^* whose angle equals the vertex's exterior turning angle.

Step 3 — the turning angles sum to 2π. A simple closed convex curve turns through exactly 2π, so the arcs total $2\pi W^*$. Adding Step 1 gives $p + 2\pi W^*$. ■

This is the Steiner formula for the perimeter of a Minkowski sum with a disc — the same geometry that Chapter 4 uses to inflate obstacles into C-obstacles. Reflex vertices contribute no arc (the two offset edges meet at a corner), and when W^* exceeds a concave radius of curvature the offset *set* self-intersects — the cusp the tracer must survive.

The number the tracer must hit. For $\mathcal{WO}_1 = [2,4] \times [-1,1]$, $p = 8$ and $W^* = 0.5$ give $8 + \pi = 11.1416$. Our tracer with $\Delta\lambda = 0.02$ closes the curve after 558 steps with length 11.1414 — the inscribed polygon undershoots the arcs by 2×10^{-4} — and every sample satisfies $|G| < 10^{-13}$.

3.4 The algorithm

Choset numbers the three Bugs as Algorithms 1–3. We state them in his form, then make the one line that all of them hide — "follow the boundary" — into an algorithm of its own.

ALGORITHM BUG1(sensor: Contact, q_start, q_goal) -- C Alg. 1 COST L ≤ d(q_start, q_goal) + 1.5 Σ p_i; O(1) sensing per step

in a point robot with a tactile sensor; q_start, q_goal

out a path to q_goal, or the conclusion that none exists

1: **while** forever **do**
2: **repeat** from q_{i-1}^L, move toward q_{goal} **until** q_{goal} is reached or an obstacle is encountered at q_i^H
3: **if** goal is reached **then** exit
4: **repeat** follow the obstacle boundary **until** q_{goal} is reached or q_i^H is re-encountered
5: determine the point q_i^L on the perimeter with the shortest distance to the goal
6: go to q_i^L (the shorter way round)
7: **if** moving toward the goal from q_i^L would re-enter the obstacle **then** conclude q_{goal} is not reachable and exit
8: **end while**

Line 5 is the exhaustive search and line 7 is the completeness test; everything else is bookkeeping. Our implementation records, on every straight piece of the boundary walk, the closest point of that piece to the goal and its arc length from

the hit point, so line 5 costs nothing extra and line 6 knows which way is shorter.

ALGORITHM BUG2(sensor: Contact, q_start, q_goal) -- C Alg. 2 COST $\le d(q\_start, q\_goal) + \frac{1}{2} \sum n_i p_i$

in a point robot with a tactile sensor; the fixed m-line from q_start to q_goal

out a path to q_goal, or the conclusion that none exists

1: **while** true **do**
2: **repeat** from q_{i-1}^L, move toward q_{goal} along the m-line **until** q_{goal} is reached or an obstacle is encountered at q_i^H
3: turn left (or right)
4: **repeat** follow the boundary **until**
5: q_{goal} is reached, **or** q_i^H is re-encountered (conclude unreachable), **or**
6: the m-line is re-encountered at a point m with $m \neq q_i^H$, $d(m, q_{\text{goal}}) < d(q_i^H, q_{\text{goal}})$, and a move toward the goal from m would not hit the obstacle
7: $q_i^L \leftarrow m$; increment i
8: **end while**

The three conditions on line 6 are the greedy leave rule. Our implementation evaluates them on each straight piece of the follower's move, intersects the piece with the m-line analytically, and seeks the follower back to the exact crossing — so a leave point is *on* the m-line, not a step past it, and the worked example lands on 16 to machine precision rather than 16.03.

ALGORITHM TANGENT BUG(sensor: RangeSensor(R), q_start, q_goal) -- C Alg. 3 COST $O(n\_rays)$ per step for $\{O_i\}$, d_reach, d_followed; completeness is Choset's problem 2.9

in a point robot with a range sensor of radius R

out a path to q_goal, or the conclusion that none exists

1: **while** true **do**
2: **repeat** continuously move toward the $n \in \{T, O_i\}$ minimizing $d(x, n) + d(n, q_{\text{goal}})$
3: **until** the goal is encountered, **or** that direction begins to increase $d(x, q_{\text{goal}})$ — a local minimum M
4: choose the boundary-following direction that continues the most recent motion-to-goal direction

5: **repeat** continuously update d_{reach}, d_{followed} and $\{O_i\}$; move toward the O_i in the chosen direction
6: **until** the goal is reached, **or** a cycle around the obstacle is completed (conclude unreachable), **or** $d_{\text{reach}} < d_{\text{followed}}$
7: **end while**

Two implementation facts the pseudocode hides. First, with a finite ray count an endpoint created by the *range limit* is a horizon, not a feature: it recedes as the robot approaches, and two such horizons either side of the goal ray can trade places every step. Our planner keeps its current side of the goal ray unless a rival wins by a full step. Second, d_{followed} — the goal distance of the closest boundary sensed *so far* — is kept across behaviors, so that "$d_{\text{reach}} < d_{\text{followed}}$" is a statement of progress rather than a trigger that fires on the first step; and the robot leaves only when motion-to-goal from the current point would actually decrease $d(x, q_{\text{goal}})$, which is what stops the two behaviors chattering at the same point.

3.4.1 Boundary following as curve tracing

All three Bugs say "follow the boundary". Choset's §2.3 asks the three questions an implementation must answer — what information does the robot need, how does it infer it from sensors, how does it use it — and answers them in order: the **tangent** $(\nabla D)^{\perp}$ (§2.3.1); the tangent from the **shortest ray** of the scan (§2.3.2); and a **continuation method** that predicts along the tangent and corrects back onto $G^{-1}(0)$ (§2.3.3). That continuation method is this chapter's owned artifact.

ALGORITHM `trace_curve(G, DG, x₀, direction, Δλ) -- App. D predictor-corrector, eq. 2.5` COST O(steps × Newton iterations); the corrector converges quadratically (Theorem D.2.1)

in a scalar field G with gradient DG; a point x_0 near the zero set; a direction; the predictor step $\Delta\lambda$

out samples on $G^{-1}(0)$, and a stop reason: closed, reached, singular, budget

1: $x \leftarrow$ Newton-correct x_0 along $\nabla G(x_0)$ until $|G(x)| < \text{tol}$
2: **repeat**
3: $n \leftarrow \nabla G(x)/\|\nabla G(x)\|$; $t \leftarrow \pm n^{\perp}$ with the sign fixed by `direction` (which side the obstacle is kept on)
4: **predict** $y_0 \leftarrow x + \Delta\lambda\, t$
5: **correct** for $h = 0, 1, 2$: $D_Y G \leftarrow \nabla G(y_h) \cdot n$; **if** $|D_Y G| < \epsilon$ **then** stop with **singular**; $y_{h+1} \leftarrow y_h - G(y_h)/D_Y G \cdot n$

6: $x \leftarrow y_{h+1}$; record x and the residuals $|G(y_h)|$
7: **until** x returns within $\Delta\lambda$ of the start (**closed**), `stop(x)` fires (**reached**), or the step budget is spent

The tracer "hovers" around the curve, as Choset's figure 2.20 draws it: it produces *samples* of the curve, not the curve, and its guarantee is local — conditioned on $D_Y G \neq 0$ in a neighborhood. The widget below is the algorithm box running, one line at a time.

INTERACTIVE · w3.2

Boundary Follower

Following a wall is root-finding: predict along the tangent, correct with Newton, and respect the hypotheses of the theorem that makes it work.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch03-bug-algorithms#w3.2

Three things to try. Drag $\Delta\lambda$ down to 0.05: the prediction barely leaves the curve and the first Newton iterate already lands within 10^{-6}. Drag it up to 0.6: the prediction falls visibly off the curve, the first correction is large, the second is tiny, and the third is machine precision — the exponents doubling in the residual strip are Theorem D.2.1 in numbers. Then press the cusp preset. The offset curve now enters the dent on the obstacle's west side, whose bottom has a radius of curvature of about 0.05 — far smaller than $W^* = 0.25$ — so the offset *set* $\{D = 0.25\}$ has a cusp there. Approaching it, ∇D swings through nearly 180° within a single predictor step, the correcting line becomes almost tangent to the level set, $|D_Y G|$ drops to 0.002, and the tracer stops after 52 steps and says `singular`. It could have divided by that 0.002 and landed somewhere on the far branch, and a less honest tracer would have. Ours reports that the hypothesis of Theorem D.1.1 has failed and lets the caller decide.

NOTE

The Bugs' boundary follower is built on the same tracer, with one pragmatic addition the pure tracer refuses to make: at an *inside* corner of the Apartment's walls, where ∇D jumps by 90° and the fixed-plane corrector is exactly singular, the follower corrects along the current gradient instead and turns the corner. The tracer's contract is to report the failure; the follower's is to survive it. Both behaviors are deliberate, and the Rust below keeps them in separate functions for that reason.

3.5 Implementation in Rust

The `bugs` crate sits on Chapter 2's `collide` (the `Scene` and its distance and witness queries) and `robots` (the `RangeSensor`). Four modules: `scan.rs` for everything that can be computed from a scan alone, `trace.rs` for the owned tracer, and `bug1.rs`, `bug2.rs`, `tangent.rs` for the planners. The TypeScript in `web/lib/bugs/` is a port of the same code, module for module, and the arena above runs it.

RUST **crates/bugs/src/scan.rs**

```rust
use nalgebra::{Point2, Vector2};

/// One ray of rho_R(x, .): the raw distance, or +inf when the ray saturates at R.
/// A scan is sensor data and nothing else -- no scene access anywhere in this file.
pub struct Scan {
    pub x: Point2<f64>,
    pub angles: Vec<f64>,
    pub ranges: Vec<f64>, // f64::INFINITY where rho >= R
    pub max_range: f64,
}

/// A maximal run of rays over which rho_R is finite and continuous (C sec.2.2).
/// `to < from` means the run wraps past bearing 0.
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub struct Interval { pub from: usize, pub to: usize, pub count: usize }

impl Scan {
    /// D(x) = min_s rho(x, s): the shortest ray (C eq. 2.4, problem 2.1). With finitely many
    /// rays this never undershoots D and overshoots by D(1 - cos(pi/n)) on a flat wall.
    pub fn distance(&self) -> f64 {
        self.ranges.iter().cloned().fold(f64::INFINITY, f64::min)
    }

    /// ?D(x): the unit vector along the shortest ray, pointing *away* from the obstacle.
    pub fn gradient(&self) -> Vector2<f64> {
        let (k, _) = self.argmin();
        let s = self.angles[k];
        -Vector2::new(s.cos(), s.sin())
    }

    /// (?D)? with its sign fixed by the side the obstacle is kept on. `Turn::Left`
    /// keeps it on the robot's right (clockwise around a convex obstacle).
    pub fn tangent(&self, turn: Turn) -> Vector2<f64> {
        let g = self.gradient();
        let t = Vector2::new(-g.y, g.x);
        match turn { Turn::Left => -t, Turn::Right => t }
    }

    /// Two consecutive hits belong to one surface when their endpoints are closer than
    /// the spacing a wall at steep incidence would produce: kappa.rho.Deltatheta, with kappa = 8 good to
    /// an incidence of ~83?, floored at `jump_abs` so a hairline step is not a corner.
    fn continuous(&self, k: usize, l: usize, kappa: f64, jump_abs: f64) -> bool {
        let (rk, rl) = (self.ranges[k], self.ranges[l]);
        if !rk.is_finite() || !rl.is_finite() { return false; }
        let dtheta = std::f64::consts::TAU / self.ranges.len() as f64;
        let gap = (self.point(k) - self.point(l)).norm();
        gap <= jump_abs.max(kappa * rk.min(rl) * dtheta)
```

```
    }

    /// The intervals of continuity. O(n_rays). Runs straddling the 0/2pi seam are merged;
    /// an all-finite, all-continuous scan (inside a closed room) is one interval with no
    /// endpoints -- the robot sees no corner to aim for.
    pub fn continuity_intervals(&self, kappa: f64, jump_abs: f64) -> Vec<Interval> {
        let n = self.ranges.len();
        let wrap = |k: isize| ((k % n as isize + n as isize) % n as isize) as usize;
        // Start at a break so wrapped runs come out whole.
        let Some(start) = (0..n).find(|&k| !self.continuous(wrap(k as isize - 1), k, kappa, jump_abs)) else {
            return if self.ranges[0].is_finite() { vec![Interval { from: 0, to: n - 1, count: n }] } else { vec![] };
        };
        let (mut out, mut k, mut visited) = (Vec::new(), start, 0);
        while visited < n {
            if !self.ranges[k].is_finite() { k = wrap(k as isize + 1); visited += 1; continue; }
            let mut count = 1;
            while visited + count < n
                && self.continuous(wrap(k as isize + count as isize - 1), wrap(k as isize + count as isize), kappa, jump_abs)
            { count += 1; }
            out.push(Interval { from: k, to: wrap(k as isize + count as isize - 1), count });
            k = wrap(k as isize + count as isize);
            visited += count;
        }
        out
    }

    fn point(&self, k: usize) -> Point2<f64> {
        let (r, a) = (self.ranges[k], self.angles[k]);
        self.x + r * Vector2::new(a.cos(), a.sin())
    }
}
```

The endpoints O_i of those intervals are refined with the sensor: starting from the last ray inside the interval, the bearing is marched toward the first ray outside with a step that doubles after each accepted move and halves after each rejection, accepting a move while the hit stays continuous with the one before — a short hop on a smooth surface, a vanishing hop at a corner, or a hit collinear with the last two on a wall seen edge-on. On a polygon world it converges on the silhouette vertex itself to 10^{-8}, which is what the tangent-line self-check compares against brute force.

RUST **crates/bugs/src/trace.rs -- THE OWNED ARTIFACT**

```
use nalgebra::{Point2, Vector2};

#[derive(Clone, Copy, Debug)]
pub enum Turn { Left, Right }

pub struct TraceParams {
    /// Deltalambda, the predictor step along the tangent.
```

```rust
    pub step: f64,
    /// Newton iterations per correction; three is usually one too many.
    pub newton_iters: usize,
    /// Stop correcting once |G| falls below this.
    pub tol: f64,
    /// |D_Y G| below this means the correcting line is tangent to the level set.
    pub singular_tol: f64,
    pub max_steps: usize,
}

#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum TraceStop { Closed, Reached, Singular, Budget, Diverged }

/// One predictor-corrector step, kept whole so a figure can draw every hop.
pub struct TraceRecord {
    pub from: Point2<f64>,
    pub tangent: Vector2<f64>,
    pub predicted: Point2<f64>,
    pub corrections: Vec<Point2<f64>>,
    /// |G(y_h)| for h = 0, 1, 2, ... -- the squares Theorem D.2.1 promises.
    pub residuals: Vec<f64>,
    pub dy_g: f64,
}

pub struct Trace {
    pub points: Vec<Point2<f64>>,
    pub records: Vec<TraceRecord>,
    pub stop: TraceStop,
    pub length: f64,
    pub max_residual: f64,
}

/// Newton along a fixed direction n -- the corrector of eq. 2.5 with D_Y G = ?G.n.
/// `singular` is set when the directional derivative vanished before the tolerance was
   met.
fn correct_along(
    g: &impl Fn(Point2<f64>) -> f64,
    dg: &impl Fn(Point2<f64>) -> Vector2<f64>,
    y0: Point2<f64>,
    n: Vector2<f64>,
    p: &TraceParams,
) -> (Point2<f64>, Vec<Point2<f64>>, Vec<f64>, f64, bool) {
    let (mut y, mut corr, mut res) = (y0, Vec::new(), vec![g(y0).abs()]);
    let dy_g0 = dg(y0).dot(&n);
    for _ in 0..p.newton_iters {
        let gy = g(y);
        if gy.abs() <= p.tol { break; }
        let dy_g = dg(y).dot(&n);
        if dy_g.abs() < p.singular_tol {
            return (y, corr, res, dy_g0, true); // Theorem D.1.1's hypothesis has failed:
    say so
        }
        y -= n * (gy / dy_g);                    // eq. 2.5
        corr.push(y);
        res.push(g(y).abs());
    }
    (y, corr, res, dy_g0, false)
}

/// Trace the zero set of G from x_0 until it closes, `stop(x)` fires, the corrector goes
/// singular, or the budget runs out. Generic over G: Chapters 7-10 pass their own.
```

```rust
pub fn trace_curve(
    g: impl Fn(Point2<f64>) -> f64,
    dg: impl Fn(Point2<f64>) -> Vector2<f64>,
    x0: Point2<f64>,
    direction: Turn,
    p: TraceParams,
    stop: impl Fn(Point2<f64>) -> bool,
) -> Trace {
    // x_0 need not be on the curve: Newton along the full gradient first.
    let mut x = x0;
    for _ in 0..8 {
        let (gx, grad) = (g(x), dg(x));
        if gx.abs() <= p.tol || grad.norm_squared() < 1e-18 { break; }
        x -= grad * (gx / grad.norm_squared());
    }
    let start = x;
    let (mut points, mut records, mut length) = (vec![start], Vec::new(), 0.0);
    let mut max_residual = g(start).abs();
    let mut outcome = TraceStop::Budget;

    for _ in 0..p.max_steps {
        let grad = dg(x);
        let n = grad / grad.norm();
        let perp = Vector2::new(-n.y, n.x);
        let t = match direction { Turn::Left => -perp, Turn::Right => perp };
        let predicted = x + t * p.step;                                    // predict
        let (y, corr, res, dy_g, singular) = correct_along(&g, &dg, predicted, n, &p); // correct
        records.push(TraceRecord { from: x, tangent: t, predicted, corrections: corr, residuals: res, dy_g });
        if singular { outcome = TraceStop::Singular; break; }
        let hop = (y - x).norm();
        if !hop.is_finite() || hop > 4.0 * p.step { outcome = TraceStop::Diverged; break; }
        length += hop;
        x = y;
        points.push(x);
        max_residual = max_residual.max(g(x).abs());
        if stop(x) { outcome = TraceStop::Reached; break; }
        if length > 3.0 * p.step && (x - start).norm() < 0.75 * p.step {
            length += (x - start).norm();
            points.push(start);
            outcome = TraceStop::Closed;
            break;
        }
    }
    Trace { points, records, stop: outcome, length, max_residual }
}

/// Convenience: the offset curve of a scene at W*, with G = D - W* and ?G = ?D from
/// `Collision::witness` -- the nearest wall point and the unit vector away from it.
pub fn offset_curve(scene: &Scene, w_star: f64, x0: Point2<f64>, dir: Turn, p: TraceParams) -> Trace {
    trace_curve(
        |x| scene.distance(x) - w_star,
        |x| scene.witness(x).gradient,
        x0, dir, p, |_| false,
    )
}
```

The planners are state machines so the arena can animate them. Bug1 below is the whole of Algorithm 1; Bug2 and Tangent Bug follow the same shape with their own switching rules. The `Contact` trait is the two behaviors as an interface — one implementation walks polygon edges exactly (zero clearance, exact lengths), the other follows the offset curve at W^* on any scene through `trace_curve`, which is how Rusty, reduced to a point by Chapter 2's disc argument, runs the Bugs in the Apartment.

RUST **crates/bugs/src/bug1.rs**

```rust
use nalgebra::Point2;
use crate::follow::{Contact, Follower, Piece};

#[derive(Clone, Copy, PartialEq, Eq)]
enum Phase { ToGoal, Circumnavigate, Return }

pub struct BugState {
    pub x: Point2<f64>,
    pub path: Vec<Point2<f64>>,
    pub length: f64,
    pub hits: Vec<Point2<f64>>,
    pub leaves: Vec<Point2<f64>>,
    pub done: bool,
    pub failed: bool,
}

/// C Alg. 1. `step` is the motion resolution; the leave point and the perimeter are
/// exact regardless of it, because the bookkeeping is done on the follower's pieces.
pub struct Bug1<'c, C: Contact> {
    contact: &'c C,
    goal: Point2<f64>,
    step: f64,
    turn: Turn,
    pub state: BugState,
    phase: Phase,
    follower: Option<C::Follower>,
    /// Best leave candidate on this obstacle: point, arc length from the hit, goal distance.
    best: (Point2<f64>, f64, f64),
    remaining: f64,
}

impl<'c, C: Contact> Bug1<'c, C> {
    pub fn step(&mut self) {
        if self.state.done || self.state.failed { return; }
        match self.phase {
            Phase::ToGoal => {
                // Lines 2-3: toward the goal until we touch something.
                let heading = (self.goal - self.state.x).normalize();
                let (x, hit) = self.contact.move_toward(self.state.x, self.goal, self.step);
                self.push(x);
                if (x - self.goal).norm() < 1e-9 { self.state.done = true; return; }
                if hit {
                    self.state.hits.push(x);
                    self.follower = Some(self.contact.follow(x, heading, self.turn));
                    self.best = (x, 0.0, (x - self.goal).norm());
                    self.phase = Phase::Circumnavigate;
```

```
                }
            }
            Phase::Circumnavigate => {
                // Line 4: a full loop, remembering the closest perimeter point (line 5).
                let f = self.follower.as_mut().unwrap();
                for Piece { a, b, s0 } in f.advance(self.step) {
                    let (q, t, d) = closest_on_segment(self.goal, a, b);
                    if d < self.best.2 - 1e-12 { self.best = (q, s0 + t * (b - a).norm(),
    d); }
                    self.push(b);
                }
                if f.closed() {
                    // Line 6: the shorter way round.
                    let p = f.perimeter().unwrap();
                    let (fwd, back) = (self.best.1, p - self.best.1);
                    if back < fwd { f.reverse(); }
                    self.remaining = fwd.min(back);
                    self.phase = Phase::Return;
                }
            }
            Phase::Return => {
                let f = self.follower.as_mut().unwrap();
                let ds = self.step.min(self.remaining);
                for Piece { b, .. } in f.advance(ds) { self.push(b); }
                self.remaining -= ds;
                if self.remaining <= 1e-12 {
                    self.push(self.best.0);
                    self.state.leaves.push(self.best.0);
                    // Line 7: would heading for the goal re-enter the obstacle?
                    if self.contact.blocked(self.best.0, self.goal) { self.state.failed =
    true; return; }
                    self.follower = None;
                    self.phase = Phase::ToGoal;
                }
            }
        }
    }

    fn push(&mut self, p: Point2<f64>) {
        self.state.length += (p - self.state.x).norm();
        self.state.path.push(p);
        self.state.x = p;
    }
}
```

3.5.1 The worked example, and the test that pins it

RUST **crates/bugs/examples/two_rectangles.rs**

```
fn main() {
    // WO_1 = [2,4]*[-1,1] (p_1 = 8), WO_2 = [6,7]*[-2,2] (p_2 = 10); q_start = (0,0),
   q_goal = (10,0).
    let scene = Scene::from_polygons(&[rect(2.0, -1.0, 4.0, 1.0), rect(6.0, -2.0, 7.0,
   2.0)]);
    let (start, goal) = (Point2::new(0.0, 0.0), Point2::new(10.0, 0.0));
    let contact = PolygonContact::new(&scene);

```

```
    let b1 = bug1(&contact, start, goal, 0.05, Turn::Left);
    let b2 = bug2(&contact, start, goal, 0.05, Turn::Left);
    let d = (goal - start).norm();
    let p: Vec<f64> = scene.polygons().iter().map(perimeter).collect();
    let n: Vec<usize> = scene.polygons().iter().map(|o| line_crossings(o, start, goal)).collect();

    println!("Bug1   length {:.4}  bound d + 1.5.Sigmap = {:.4}   hits {}  leaves {}",
             b1.length(), d + 1.5 * p.iter().sum::<f64>(), fmt(&b1.hits), fmt(&b1.leaves));
    println!("Bug2   length {:.4}  bound d + ?.Sigma n?p? = {:.4}   hits {}  leaves {}",
             b2.length(), d + 0.5 * n.iter().zip(&p).map(|(n, p)| *n as f64 * p).sum::<f64>(),
             fmt(&b2.hits), fmt(&b2.leaves));

    // Tangent Bug's first decision with infinite range: the sqrt5 + sqrt65 tie.
    let tb = TangentBug::new(&scene, start, goal, TangentOptions { range: f64::INFINITY, clearance: 0.1, ..Default::default() });
    let e = tb.evaluate(start);
    println!("TBuginf  first subgoal: tie {:.4} between O=(2,1) and O=(2,-1) -> {:?}; length {:.4}",
             e.candidates[0].h, Turn::Left, tangent_bug(&scene, start, goal, 0.05, f64::INFINITY).length());

    // The tracer against Derivation 6: p + 2piW* = 8 + pi.
    let tr = offset_curve(&scene.only(0), 0.5, Point2::new(1.4, 0.0), Turn::Left,
                          TraceParams { step: 0.02, newton_iters: 3, tol: 1e-12, singular_tol: 0.2, max_steps: 5000 });
    println!("offset curve of WO1 at W*=0.5: {:?} after {} steps of Deltalambda=0.02, length {:.4} (8+pi = {:.4}), max |G| {:.1e}",
             tr.stop, tr.points.len() - 1, tr.length, 8.0 + std::f64::consts::PI, tr.max_residual);
}
```

```
Bug1   length 34.0000  bound d + 1.5.Sigmap = 37.0000   hits (2,0) (6,0)  leaves (4,0) (7,0)
Bug2   length 16.0000  bound d + ?.Sigma n?p? = 28.0000   hits (2,0) (6,0)  leaves (4,0) (7,0)
TBuginf  first subgoal: tie 10.2983 between O=(2,1) and O=(2,-1) -> Left; length 11.2128
offset curve of WO1 at W*=0.5: Closed after 558 steps of Deltalambda=0.02, length 11.1414 (8+pi = 11.1416), max |G| 3.0e-14
```

Read those four lines against the §3 arithmetic. Bug1: $2+8+4+2+10+5+3 = 34$, against a bound of $10 + 1.5(8 + 10) = 37$. Bug2: $2 + 4 + 2 + 5 + 3 = 16$, against $10 + \frac{1}{2}(2 \cdot 8 + 2 \cdot 10) = 28$. Tangent Bug with infinite range sees both corners of $\mathcal{WO}_1$ at $\sqrt{5} + \sqrt{65} = 10.2983$, breaks the tie by its turn direction, and walks 11.21 — shorter than Bug2's 16 because it cuts the corners it can see. And the tracer closes the offset curve of $\mathcal{WO}_1$ at $8 + \pi$ to two parts in 10^5.

RUST **crates/bugs/tests/bounds.rs**

```
use proptest::prelude::*;

#[test]
fn bug1_two_rectangles() {
    let (scene, start, goal) = two_rectangles();
```

```
    let s = bug1(&PolygonContact::new(&scene), start, goal, 0.05, Turn::Left);
    assert_relative_eq!(s.length(), 34.0, epsilon = 1e-6);
    assert_eq!(s.hits, vec![Point2::new(2.0, 0.0), Point2::new(6.0, 0.0)]);
    assert_eq!(s.leaves, vec![Point2::new(4.0, 0.0), Point2::new(7.0, 0.0)]);
}

#[test]
fn bug2_two_rectangles() {
    let (scene, start, goal) = two_rectangles();
    assert_relative_eq!(bug2(&PolygonContact::new(&scene), start, goal, 0.05, Turn::Left).
   length(), 16.0, epsilon = 1e-6);
}

proptest! {
    /// Both bounds, on seeded fields of 1-6 disjoint convex polygons with random queries.

    /// `Unreachable` must come back iff the goal's component differs from the start's.
    #[test]
    fn bounds_hold(seed in 0u64..500) {
        let field = random_convex_field(SmallRng::seed_from_u64(seed));
        let contact = PolygonContact::new(&field.scene);
        let d = (field.goal - field.start).norm();
        let p: f64 = field.scene.polygons().iter().map(perimeter).sum();
        let np: f64 = field.scene.polygons().iter()
            .map(|o| line_crossings(o, field.start, field.goal) as f64 * perimeter(o)).
   sum();
        for turn in [Turn::Left, Turn::Right] {
            match bug1(&contact, field.start, field.goal, 0.05, turn) {
                BugOutcome::Reached(path) => prop_assert!(path.length() <= d + 1.5 * p +
   1e-9),
                BugOutcome::Unreachable(_) => prop_assert!(!field.connected()),
            }
            match bug2(&contact, field.start, field.goal, 0.05, turn) {
                BugOutcome::Reached(path) => prop_assert!(path.length() <= d + 0.5 * np +
    1e-9),
                BugOutcome::Unreachable(_) => prop_assert!(!field.connected()),
            }
        }
    }
}

#[test]
fn offset_square_length() {
    let tr = offset_curve(&Scene::from_polygons(&[rect(2.0, -1.0, 4.0, 1.0)]), 0.5,
                          Point2::new(1.4, 0.0), Turn::Left, TraceParams::fine(0.02));
    assert_eq!(tr.stop, TraceStop::Closed);
    assert_relative_eq!(tr.length, 8.0 + std::f64::consts::PI, epsilon = 1e-3);
    assert!(tr.max_residual < 1e-8);
}

#[test]
fn tracer_reports_singular() {
    // Into the kidney's dent at W* = 0.25: the offset set has a cusp, D_Y G -> 0.
    let tr = offset_curve(&kidney(), 0.25, Point2::new(0.0, -1.75), Turn::Left,
   TraceParams::fine(0.05));
    assert_eq!(tr.stop, TraceStop::Singular);
}
```

The TypeScript port's `__checks_ch03__.ts` asserts the same thirteen facts

— 34, 16, the tie, 558 steps and $8 + \pi$, the squares in the residuals, `singular` at $W^* = 0.25$, the bounds over forty seeded fields, and the Apartment lab below — and `npm run check` runs them every time the book is built.

3.6 Putting it together: the Apartment lab

Everything so far ran on polygons the planner could walk exactly. Rusty's world is the Apartment of Chapter 2: forty wall segments, doorways, a kitchen counter, no polygons at all, and a robot with a radius. Chapter 2's disc reduction says to shrink Rusty to a point and inflate every wall by r; here that means the point robot keeps $W^* = 0.11$ from every wall, hits when $D(x)$ reaches W^*, and follows the offset curve $G = D - 0.11 = 0$ with `trace_curve`. The query is the one every chapter of this book runs: from room A at $(1, 1)$ to the bedroom at $(11, 8)$.

Planner	Path length	Hits / local minima	What it did
Bug1	173.37 m	1	Hit the party wall of room A, followed the offset curve of the *entire* connected wall network — every room, both sides of every interior wall — found the perimeter point closest to the bedroom, walked back to it, and went in.
Bug2	112.34 m	2	Hit the same wall, followed it until the fixed m-line was crossed closer to the goal with a free step ahead, left, hit once more in the corridor, left again, arrived.
Tangent Bug ($R = 3$)	13.51 m	0	Saw the doorway as a discontinuity of ρ_R from the start, steered through it, never switched behaviors. The straight-line distance is 12.21 m.

Three honest observations. First, both contact-sensor Bugs *arrive*, and never come closer than W^* to a wall — the self-check asserts a minimum clearance of exactly 0.1100 along all three paths. Completeness is real. Second, Bug1's number is absurd, and correctly so: the Apartment's walls are one connected obstacle whose offset curve is about 115 m long, and the exhaustive rule walks all of it. There is no "$\sum p_i$" to quote here — the perimeter of a wall network viewed from inside is not a quantity Choset's bound was written for — which is itself a lesson about what "bounded workspace with finitely many obstacles" assumes. Third, Tangent Bug is 1.1 times the straight line. A range sensor with a

3 m horizon turns a 173 m ordeal into a 13.5 m stroll, and it does so with exactly the information Choset's §2.3 says a robot needs: discontinuities, the shortest ray, and its direction.

The design doc's phrase for the chapter is *a blindfolded walker with one hand on the wall*. The lab is where the metaphor earns its keep: with the blindfold on, you can still promise to arrive — Bug1 — and with a cane that reaches three metres you can promise it in a tenth of the distance. Neither promise needs a map.

3.6.1 What the Bugs assume, and what breaks without it

Every guarantee in this chapter rests on four assumptions Choset states up front: a point robot; perfect position (the robot knows $d(x, q_{\text{goal}})$ exactly, recognizes when it is back at q_i^H, and can return to q_i^L); a bounded workspace; finitely many obstacles. Drop perfect position and Bug1's "re-encountered q_i^H" becomes a tolerance band, Bug2's "closer than the hit point" a hysteresis — Exercise 6 asks you to put them in and measure the failure rate. Drop the point robot and the disc-to-point reduction of Chapter 2 is what you reach for; our lab did exactly that. And Tangent Bug's completeness is not proved anywhere in this chapter — Choset leaves it as problem 2.9, and we leave it there too: our implementation terminates on an enclosed goal by recognizing a completed cycle, and carries a step budget besides, because an algorithm without a proof should not be allowed to loop forever in a figure.

Production stacks do not navigate with Bugs. Nav2's local planners and the MPC of Chapter 19 replaced them long ago, with maps and replanning at costmap rate. But look inside any recovery behavior — back up, spin, follow the wall until the corridor opens — and the two-behavior architecture is still there, switching rule and all. The Bugs are not practical navigation. They are the shortest honest route to a complete sensor-based planner, and the chapter where the curve tracer that Chapters 8 and 9 (Chapter 9) build roadmaps with was born. Rusty's unknown-Apartment navigation in the capstone (Chapter 23) starts from these behaviors before replanning takes over.

3.7 Exercises

1 FOUNDATION •• **Tighten the Bug1 bound (C problem 2.7)**

Prove that in Bug1 the robot never contacts an obstacle lying entirely outside the disk of radius $d(q_{\text{start}}, q_{\text{goal}})$ centered at q_{goal}. Use Step 1 of the completeness derivation: every point of the path after the first leave point is strictly closer to the goal than the start was. Conclude that the sum in (2.1) may be taken over the obstacles that meet that disk, and compute the tightened bound for the worked example — then say in one sentence why it does *not* change there.

2 FOUNDATION •• **Write out D_Y G (C problems 2.11 and 2.12)**

For planar boundary following at offset W^*, with n the unit normal of the correcting line, write $D_Y G$ explicitly in terms of ∇D and show it equals ± 1 exactly on the offset curve. Then explain why $G_1 = D + 1$ and $G_2 = D + 2$ have the same Jacobian, and what that means for a tracer that only ever sees ∇G. Finally, name the precise hypothesis of Theorem D.2.1 that fails at a point of the medial axis, and relate it to the number 0.002 printed by the cusp preset.

HINTS

<>G and D differ by a constant. What does differentiation do to constants?</> <>Theorem D.2.1 asks for $\|(Df(y^*))^{-1}\| \le \beta$. Which quantity is Df here, and what happens to β as it tends to zero?</> <>On the medial axis ∇D is not defined — two nearest points tie — so no neighborhood on which Df is Lipschitz with any finite γ exists. The tracer's 0.002 is $|D_Y G|$ one step before that.</>

3 CONCEPTUAL • **Predict the spiral**

Load the spiral preset in the Bug Arena and pause before it finishes.

WORK IT OUT

How many times does the line through q_start and q_goal cross the spiral's boundary — Choset's n_1 in eq. (2.2)?

Answer. 10 Each of the five left arms of the spiral is crossed twice, once per face: $n_1 = 10$. With $p_1 = 85.2$ the bound (2.2) is $6.5 + \frac{1}{2} \cdot 10 \cdot 85.2 = 432.5$, and Bug2 turning right actually walks 132.4 — the bound is a factor three loose, because only the arms the robot *reaches* cost it anything.

PREDICT FIRST

Now flip the turn direction to left and reset. Which Bug walks the shortest path?

- Bug1
  Bug1 does not care about the turn direction beyond which way it circumnavigates: it still walks the whole perimeter once (124.9 or 131.3). It only wins when Bug2 is forced to re-walk the boundary.
- ✓ Bug2
  Turning left, Bug2 goes up the outer arm, round the outside, and spirals inward along faces whose m-line crossings are all wrong-side — until the innermost one, which is valid. One pass, 52.2 units, and it never re-walks anything. Greed won because the geometry let it.
- Tangent Bug
  Tangent Bug does well here (it sees the channels), but its path still has to wind into the spiral; it is not shorter than Bug2 turning left on this map.

The algorithm did not change between the two runs. The obstacle and the turn direction decided whether the greedy leave rule paid a perimeter per crossing or nothing at all — which is exactly what (2.2) says: the bound is about the worst arrangement of crossings, and a lucky arrangement beats it by a mile.

4 **CONCEPTUAL** •• **Count the Newton iterations**

In the Boundary Follower set $W^* = 0.5$ and $\Delta\lambda = 0.4$ — roughly a third of the obstacle's convex radius of curvature — and step once.

WORK IT OUT

From the first residual $|G(y_0)| \approx 4 \times 10^{-2}$, how many Newton iterations does the quadratic rate predict before $|G| < 10^{-8}$?

Answer. 2 With $a \approx 0.045$, the residuals go $4 \times 10^{-2} \to 7 \times 10^{-5} \to 2 \times 10^{-10}$: two iterations. Now raise W^* past the dent's radius of curvature by pressing the cusp preset, and read what the tracer did: it stopped and reported singular at the bottom of the dent, where $|D_Y G| = 0.002$. Theorem D.1.1 needs $D_Y G$ invertible in a neighborhood; at a cusp of the offset set there is no such neighborhood, and the tracer declines to pretend otherwise.

5 PRACTICAL •• **A limited field of view (C problem 2.10)**

Give `TangentBug` a sensor that sees only a ±90° wedge about its heading. Which O_i can it still find, and which of d_{reach} and d_{followed} becomes unsafe — i.e. can be *wrong* rather than merely stale — when the robot cannot see behind it while following a boundary? Add a fourth lane to the arena with your wedge sensor and compare lengths over 100 seeded random fields against the full-circle sensor. Report the mean ratio and the number of seeds on which the wedge sensor failed to arrive.

6 PRACTICAL ••• **A robust Bug1 (C problem 2.6)**

Replace perfect position with the sister book's odometry drift and noisy LiDAR (its Chapter 4) and make Bug1 robust: a tolerance ϵ for "re-encountered q_i^H", a hysteresis band for the leave test so the robot does not leave on noise. Over 200 seeds in the Apartment, measure the failure rate (did not arrive, or declared unreachable) as a function of the noise level, and report the smallest ϵ that keeps the measured path length within 10 % of the noise-free 173.37 m. State which of the four assumptions in "What the Bugs assume" you had to relax, and which one you could not.

3.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G. A., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

The spine of this chapter and the source of its epigraph. Chapter 2 is Bug1, Bug2 and Tangent Bug with Algorithms 1–3 and eqs. 2.1–2.5; Appendix D is the implicit function theorem and the Newton–Raphson convergence theorem. Its bibliography carries Tangent Bug's original papers (Kamon, Rivlin and Rimon, 1996 and 1998) and the continuation-methods literature behind §2.3.3. `link`

Lumelsky, V. J. and Stepanov, A. A. (1987). Path-Planning Strategies for a Point Mobile Automaton Moving Amidst Unknown Obstacles of Arbitrary Shape. *Algorithmica 2(1), 403–430*.

Bug1 and Bug2, with the completeness proofs and the path-length bounds this chapter restates as eqs. 2.1 and 2.2. The original already contains the observation that Bug2 can be made arbitrarily worse than Bug1. `doi:10.1007/BF01840369`

Khatib, O. (1986). Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. *The International Journal of Robotics Research 5(1), 90–98*.
Motion-to-goal is gradient descent on d(·, n), and Chapter 7 turns that observation into a potential field — with the local minima the Bugs' boundary following is designed to escape. doi:10.1177/027836498600500106

Rimon, E. and Koditschek, D. E. (1992). Exact Robot Navigation Using Artificial Potential Functions. *IEEE Transactions on Robotics and Automation 8(5), 501–518*.
The other way to a complete sensor-free planner: a navigation function with a single minimum. Chapter 7 contrasts it with the Bugs' two-behavior architecture. doi:10.1109/70.163777

Allgower, E. L. and Georg, K. (1990). Numerical Continuation Methods: An Introduction. *Springer Series in Computational Mathematics 13, Springer*.
The predictor–corrector family trace_curve belongs to, including the implicit-function-theorem argument of Appendix D and the step-length control this chapter's tracer leaves to the reader. doi:10.1007/978-3-642-61257-2

Macenski, S., Moore, T., Lu, D. V., Merzlyakov, A., and Ferguson, M. (2023). From the Desks of ROS Maintainers: A Survey of Modern and Capable Mobile Robotics Algorithms in the Robot Operating System 2. *Robotics and Autonomous Systems 168, 104493*.
What production navigation actually does instead of Bugs — map-based planners replanning at costmap rate — and where the two-behavior idea survives: in the recovery behaviors. doi:10.1016/j.robot.2023.104493

CHAPTER 4

Configuration Space I: Obstacles in the Space of Configurations

FOUNDATIONS — ROBOTS, WORLDS, AND CONFIGURATION SPACE · Intermediate · 60 min

> *Although the example in figure 3.5 is quite simple, the main point is that it is easier to think about points moving around than bodies with volume.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
> *Principles of Robot Motion (2005), §3.2.1*

Chapter 1 promised that the piano is a point. This chapter pays. It answers the four questions Choset opens his Chapter 3 with — how much information specifies a robot's position, how to represent it, what its mathematical properties are, and how obstacles in the workspace become constraints on it — for every case that can be computed exactly, and it opens the collision box that Chapter 2 deliberately sealed.

The idea the whole chapter turns on is one sentence long and easy to say wrong. A configuration space obstacle is **not a picture of the workspace obstacle**. It is the set of configurations q at which a *function* of the configuration, $R(q)$, meets the obstacle. For a disc that set happens to look like the obstacle, grown by the radius — and Choset warns on his page 44 that this is the one case where it does. For a polygon that rotates, it is a Minkowski difference computed one vertex at a time, and the vertices re-pair every time the robot turns. For a two-joint arm it is a blob on a torus that nobody can draw by hand and that a planner must nonetheless search. The widget at the top of this page draws it live; the mathematics below says why it has the shape it has; the Rust at the end computes it, and the tests hold every printed number to the code.

Two things from Choset's §3.8 ride along, because every later planner for Reach — the potential fields of Chapter 7, the dynamics of Chapter 17, the steering of Chapter 21 — needs them already: the Jacobian $J(q) = \partial\varphi/\partial q$ that turns joint velocities into tip velocities, and the configurations $\sin\theta_2 = 0$ where it loses rank. The topology that makes the torus wrap, and why the square is only a chart of it, waits for Chapter 5.

AFTER THIS CHAPTER YOU CAN

- Define a configuration, the configuration space, and the degrees of freedom of a robot, and count them by constraints — three for a planar body, six in space, Grü bler's formula for closed chains
- Define the C-obstacle QO_i and the free space Q_free, and compute QO exactly for a translating disc (Minkowski inflation) and for a translating convex polygon (the star algorithm), and approximately for the two-joint arm (Choset's pixel method)
- Open the collision box: the Type A / Type B contact test of Appendix F.2, the star algorithm of F.3, SE(2) as stacked slices (F.4), and GJK (F.5) — each verified against a brute-force oracle
- Differentiate Reach's forward kinematics: the Jacobian J(q), det J = $L_1 L_2 \sin\theta_2$, and what a singular configuration does and does not mean
- Say honestly what a rasterized C-obstacle certifies — a picture, not a proof — and what thickening buys
- Implement all of it in the cspace crate, with the five hand-checked cells, the 29-cell row, the five Minkowski vertices and Choset's Jacobian pinned by tests

ASSUMED

- Chapter 2: Reach's forward kinematics $\phi(\theta_1, \theta_2)$, the T2 and Pose2 configuration types, the Scene and the Collision contract
- Chapter 3: the distance D(x) to the nearest obstacle — GJK is how a collision engine computes it
- Chapter 1: the piano-mover reduction, "the robot is a point in the space of its configurations"
- Elementary vector geometry: dot products, outward normals, convexity (Choset Appendix B, reviewed as needed)

4.1 The problem: a block, and its shadow

Below, a slate block slides across the Workbench on a seeded path while Reach, the two-joint arm, sweeps through its configurations. Beside it, the square $[-\pi, \pi]^2$ of joint angles — the torus T^2 cut open and flattened — with an amber region that stretches, splits, and rejoins as the block moves. Before reading on: what is the amber region?

> INTERACTIVE · w4.1
>
> **C-Space Morph**
>
> A C-obstacle is not a picture of the obstacle: it is the set of configurations where R(q) meets it — a blob on a torus that moves the other way.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch04-configuration-space-i#w4.1

It is the set of joint-angle pairs (θ_1, θ_2) at which some point of the arm lies inside the block. Choset calls it the **configuration space obstacle** QO, and three things about it are the whole chapter.

It is not the block. The block is a square in the plane. Its shadow on the torus is a curved blob whose shape depends on where the block is, how long the links are, and which link can reach it. Slide the block toward the base and a vertical band appears — link 1 now hits it for *every* θ_2, so a whole column of the square is forbidden. Slide it the other way and the blob shrinks toward a point. Drag it in one direction and the blob can move in the other. The misconception this widget exists to kill is that the C-obstacle "looks like" the obstacle. It is a different shape in a different space.

It is the preimage of a function. Write $R(q)$ for the set of workspace points the arm occupies at configuration q. The amber region is $\{q : R(q) \cap \text{block} \neq \emptyset\}$, the set of inputs for which a function of q meets a fixed set. Every property of the blob — its convexity or lack of it, its connectedness, whether it wraps across the seam — is a property of that function. This is also why the same code draws the disc's C-space in the next section: nothing about the rasterizer knows what an arm is. It asks a `Collision` checker one question per cell.

It is a picture, not a certificate. The amber cells are the configurations the raster *tested*. A path through the white cells passes through configurations no one tested. Choset's §3.2.2 says this plainly — the grid "only encodes collision information for the discrete set of points lying on the grid" — and offers the remedy the widget's *link thickness* slider implements: thicken the robot so that a free sample certifies a free neighborhood. The planner is then conservative and only *resolution complete*. We will come back to this.

The readout under the torus counts cells and components. Load the *micro-example square* preset and the $\theta_1 = 0°$ row shows exactly 29 occupied cells. That number is the chapter's handshake between hand, raster and code: the next

section derives it with a protractor, the Rust at the end prints it, and a test asserts it.

> **THE ONE SENTENCE VERSION**
>
> A configuration space obstacle is the set of configurations at which the robot's footprint $R(q)$ meets the workspace obstacle. For a disc that is the obstacle grown by r; for a convex polygon at fixed heading it is a Minkowski difference, built vertex by vertex from the two bodies' edge normals; for an arm it is a region of the torus that only a collision checker can draw. In every case the obstacle lives in the space of configurations, and planning happens there.

4.2 Building intuition

4.2.1 A disc only grows

Start where the picture *is* easy. Choset's figure 3.5 puts three circular robots of increasing radius in one room with a doorway and asks each to cross it. The widget below is that figure with numbers: a 4×5 m room, a 0.6 m doorway in the dividing wall, one convex crate, and three discs — a point, Rusty at $r = 0.11$ m, and a third whose radius you set.

> **INTERACTIVE · w4.4**
>
> **Disc Inflation**
>
> For a disc, growing the obstacles by r and planning for a point is exactly right — and right only for discs.
>
> Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch04-configuration-space-i#w4.4`

For a disc translating in the plane the configuration is the center (x, y), so $\mathcal{Q} = \mathbb{R}^2$ and it is tempting to think of configuration space as the workspace. It is not — they are different spaces that happen to have the same coordinates — but the C-obstacle of a disc does have a closed form. The disc of radius r at q meets the crate iff some crate point lies within r of q, so $\mathcal{QO}$ is the crate with every point pushed outward by r: edges slide out by r, corners become quarter-circles, and walls — zero-width obstacles — become capsules of width $2r$. Choset's phrase is that "we 'grow' the polygon outward and the walls inward." The point robot then plans in the grown map.

Three numbers to check against the readouts. The point robot's free area is the room less the crate, $20 - 0.4 = 19.60$ m^2. For Rusty the crate's C-obstacle has area $|\mathcal{WO}| + r\,p + \pi r^2 = 0.4 + 0.11 \cdot 2.6 + \pi \cdot 0.0121 = 0.7240$ m^2 — the original, a

strip of width r along the perimeter p, and corner arcs that together make exactly one disc. And the doorway, 0.6 m wide, narrows to $0.6 - 2r$: Rusty passes with 0.19 m to spare on each side, the free space is one component, and the purple path goes through. Raise the third radius past 0.30 m and the two capsules meet in the doorway; Q_{free} falls into **two** components and the query has no answer — which is exactly what Choset's figure 3.5(c) shows and what no amount of clever search can fix. The check `doorway` pins the component counts 1/1/2 for $r = 0, 0.11, 0.35$ and the free area 19.60 m^2.

Now the honesty item this widget is really about. The disc inflation is *correct* — "inflate by the radius and plan for a point" is exactly right — and it is right **only for discs**. A disc has no orientation, so its footprint is the same set translated; for any body that rotates, $R(q)$ changes shape with θ, and the "grow the obstacle" picture fails at the first corner. The Minkowski Studio in the mathematics section shows what replaces it.

4.2.2 An arm's C-obstacle is a shape in a different space

Here is the micro-example that runs through the rest of the chapter, chosen so that every number can be checked with a protractor. Reach has $L_1 = L_2 = 1$, zero-thickness links, base at the origin; the only obstacle is a small square $\mathcal{WO} = [1.4, 1.6] \times [-0.1, 0.1]$ — side 0.2, centered at $(1.5, 0)$. Link 1 has length 1 and the square begins at $x = 1.4$, so link 1 can never touch it: the C-obstacle comes from **link 2 alone**.

Five configurations by hand.

- **(a)** $q = (0°, 0°)$. Link 2 runs from the elbow $(1, 0)$ to the tip $(2, 0)$, straight through the square's center. **Collision.**
- **(b)** $q = (0°, 13°)$. Link 2 leaves $(1, 0)$ at 13°; at $x = 1.4$ its height is $0.4 \tan 13° = 0.092 < 0.1$, inside the square's left edge. **Collision.**
- **(c)** $q = (0°, 15°)$. Height at $x = 1.4$ is $0.4 \tan 15° = 0.107 > 0.1$: the link clears the corner $(1.4, 0.1)$. **Free.**
- **(d)** $q = (45°, -90°)$. Elbow at $(0.7071, 0.7071)$, link 2 at absolute heading −45°, tip at $(\sqrt{2}, 0) = (1.4142, 0)$ — inside the square. **Collision.**
- **(e)** $q = (90°, \theta_2)$ for any θ_2. Elbow at $(0, 1)$; the nearest point of the square is the corner $(1.4, 0.1)$ at distance $\sqrt{1.96 + 0.81} = 1.664 > L_2$. **Free for every** θ_2.

So the slice of $\mathcal{QO}$ at $\theta_1 = 0$ is exactly $|\theta_2| \le \arctan(0.1/0.4) = \arctan \frac{1}{4} = 0.2450$ rad = 14.04°, the angle the near corner subtends from the elbow. On a 1° raster whose cells are centered at integer degrees, that is the cells $\theta_2 = -14°, \ldots, 14°$: $2 \cdot 14 + 1 = 29$ cells out of 360 — the number the widget's readout shows for the preset. Generalizing (e), the θ_1-extent of the blob is the set of elbow positions within $L_2 = 1$ of the square: $(1.4 - \cos\theta_1)^2 + (\sin\theta_1 - 0.1)^2 \le 1$, i.e. $2.8\cos\theta_1 + 0.2\sin\theta_1 \ge 1.97$, i.e. $|\theta_1| \lesssim 0.864$ rad = 49.5°. On the raster, the rows $\theta_1 = \pm 49°$ hold one cell each and the rows ±50° are empty; the blob is one

connected region of 2,015 cells (1.55% of T^2) and touches no seam. The check `cspace` asserts all of it, and bisects the exact segment test along the $\theta_1 = 0$ row to recover $\arctan \frac{1}{4}$ to 10^{-9}.

Compare this with the square itself: a 0.2×0.2 box. Its shadow on the torus is a curved lens about 99° wide and 29° tall at its waist, with no straight edges and no right angles. That is the picture to carry into the mathematics.

4.2.3 Why "grow the obstacle" fails for a polygon

One more experiment before the definitions. Suppose the robot is a convex polygon that translates at a fixed heading θ among a convex obstacle. Its C-obstacle is again convex, and it is again built from the obstacle and the robot — but *not* by growing the obstacle: it is the obstacle with the *reflected* robot swept along its boundary, the **Minkowski difference** $\mathcal{WO} \ominus R$. Turn the robot and the sweep changes; the C-obstacle's vertex count can change with it. The Minkowski Studio, placed beside the star algorithm below, computes it one vertex at a time. The misconception it kills is that Minkowski sums are expensive. For convex polygons they are a *merge of two sorted lists*.

4.3 The mathematics

$\mathrm{cl}(\cdot)$, $\mathrm{int}(\cdot)$, ∂	closure, interior, boundary of a set (Choset App. B)
$A \oplus B$, $A \ominus B$	Minkowski sum {a + b} and Minkowski difference {a − b}
M, N, k, n, f_i	mobility; ambient DOF (3 planar, 6 spatial); links; joints; DOF of joint i (Grü bler)
$g(q,t) = 0$, $g(q,\dot{q},t) = 0$	a holonomic constraint; a nonholonomic (velocity) constraint
$h(x,y) = ax + by - c$, h^-, h^+	a line with unit normal (a, b) and its negative / positive half-planes (App. F.1)
r_i, E_i^R, $n_i^R(\theta)$; o_j, E_j^W, n_j^W	robot vertices, edges, outward normals (θ-dependent); obstacle vertices, edges, normals
$f_{ij}^R(q)$, $f_{ij}^W(q)$	the half-plane constraints of a Type A and a Type B contact (eqs. F.4, F.6)
$J(q) = \partial\varphi/\partial q$	the Jacobian of the forward kinematics; $\dot{x}$ = J(q) $\dot{q}$
$h_Z(x)$, $s_Z(x)$	GJK support value max z·x and support point of a polytope Z in direction x (eqs. F.11–F.12)

4.3.1 Configuration, configuration space, degrees of freedom

DEFINITIONS (CHOSET §3.1) — CONFIGURATION, C-SPACE, DEGREES OF FREEDOM

The **configuration** of a robot system is a complete specification of the position of every point of that system. The **configuration space** $\mathcal{Q}$ is the space of all configurations; a configuration q is a point of it. The number of **degrees of freedom** is $\dim \mathcal{Q}$ — equivalently, the minimum number of parameters needed to specify a configuration. $R(q) \subset \mathcal{W}$ denotes the set of workspace points the robot occupies at q.

For the disc, $q = (x, y)$ and $R(x, y) = \{(x', y') : (x - x')^2 + (y - y')^2 \le r^2\}$, so $\mathcal{Q} = \mathbb{R}^2$. For the two-joint arm, $q = (\theta_1, \theta_2)$ with each $\theta_i \in S^1$, so $\mathcal{Q} = S^1 \times S^1 = T^2$, the torus. Choset's figure 3.2 cuts the torus along $\theta_1 = 0$ and $\theta_2 = 0$ and flattens it to a square whose opposite edges are identified — the square the widgets draw, with its seams marked. The *workspace* of the arm, in the sense of the set of points the end effector reaches, is an annulus; it is **not** a configuration space, because every interior point is reached by two configurations (elbow-up and elbow-down), so a tip position does not specify where every point of the robot is. Chapter 2's `Reach::ik` returns both.

4.3.2 C-obstacles, free space, and the two kinds of path

DEFINITIONS (CHOSET §3.2) — C-OBSTACLE, FREE SPACE, FREE AND SEMIFREE PATHS

The **configuration space obstacle** of workspace obstacle $\mathcal{WO}_i$ is $\mathcal{QO}_i = \{q \in \mathcal{Q} : R(q) \cap \mathcal{WO}_i \neq \emptyset\}$. The **free space** is $\mathcal{Q}_{\text{free}} = \mathcal{Q} \setminus \bigcup_i \mathcal{QO}_i$, assumed open unless noted. A **free path** is a continuous $c : [0, 1] \to \mathcal{Q}_{\text{free}}$; a **semifree path** is a continuous $c : [0, 1] \to \text{cl}(\mathcal{Q}_{\text{free}})$, which may touch obstacle boundaries.

$$\mathcal{QO}_i = \{\, q \in \mathcal{Q} : R(q) \cap \mathcal{WO}_i \neq \emptyset \,\}, \qquad \mathcal{Q}_{\text{free}} = \mathcal{Q} \setminus \bigcup_i \mathcal{QO}_i .$$

Hover a tinted term and every figure on the page mutes the other roles: the amber of the C-obstacle is the amber of the torus, the slate of the workspace obstacle is the slate of the block. The definition is a *preimage*: $\mathcal{QO}_i$ is the set of q that the map $q \mapsto R(q)$ sends into the sets that meet $\mathcal{WO}_i$. Two consequences follow at once and both matter for code. First, obstacles union: $\mathcal{QO}(\mathcal{WO}_i \cup \mathcal{WO}_j) = \mathcal{QO}_i \cup \mathcal{QO}_j$, because $R(q)$ meets a union iff it meets a member (Choset problem 3.5) — so a rasterizer may ask one checker about a whole `Scene` rather than one obstacle at a time. Second, for a *convex* robot translating against a *convex* obstacle, $\mathcal{QO}$ is convex (problem 3.4; the star algorithm below is its constructive proof).

DERIVATION **Derivation 1 — The disc's C-obstacle is a Minkowski inflation (Choset §3.2.1)**

$$\mathcal{QO} = \mathcal{WO} \oplus B_r(0), \qquad |\mathcal{WO} \oplus B_r| = |\mathcal{WO}| + r\,p + \pi r^2$$

Statement. For a disc of radius r translating in the plane, with configuration its center q, $\mathcal{QO}_i = \mathcal{WO}_i \oplus B_r(0) = \{o + b : o \in \mathcal{WO}_i,\ \|b\| \le r\}$. For a convex polygon $\mathcal{WO}$ with perimeter p the inflated area is $|\mathcal{WO}| + r\,p + \pi r^2$.
Step 1 — the preimage, unpacked. $R(q) = q + B_r(0)$. Then $R(q) \cap \mathcal{WO} \neq \emptyset$ iff there is an $o \in \mathcal{WO}$ with $\|q - o\| \le r$ iff $q = o + b$ for some $o \in \mathcal{WO}$, $\|b\| \le r$, iff $q \in \mathcal{WO} \oplus B_r$. Nothing about the disc's shape entered except that it is a ball about its reference point: the sum is exactly the obstacle "grown" by r.
Step 2 — the boundary. The boundary of $\mathcal{WO} \oplus B_r$ for a convex polygon consists of each edge offset outward by r along its normal, joined at each vertex by a circular arc of radius r sweeping from the incoming edge's normal to the outgoing edge's normal. `inflateDisc` builds exactly this: `arcSegments` chords per full turn, so the chord polygon approaches the true inflation from the inside.
Step 3 — the area. The offset strips contribute r times the perimeter p. The arcs at the vertices have angles that sum to the total turning of a convex polygon, 2π, so together they make one full disc, πr^2. For $\mathcal{WO} = [1,2]^2$ and $r = 0.5$: $1 + 0.5 \cdot 4 + \pi \cdot 0.25 = \mathbf{3.7854}$. The check `inflate` asserts the closed form and that the chord polygon's area climbs toward it from below — 3.7804 with 32 chords, 3.7854 with 720. ■
Choset's warning (his page 44): "although both the workspace and the configuration space for this system can be represented by $\mathbb{R}^2$, and the obstacles appear to simply 'grow' in this example, the configuration space and workspace are different spaces, and the transformation from workspace obstacles to configuration space obstacles is not always so simple." A rotating body's footprint is not a ball about any point; Derivations 3 and 4 are what "not so simple" costs.

4.3.3 Dimension: counting by constraints

DEFINITIONS (CHOSET §3.3) — HOLONOMIC AND NONHOLONOMIC CONSTRAINTS; MOBILITY; REDUNDANCY

A **holonomic constraint** is an equation on the configuration variables, $g(q,t) = 0$. Each linearly independent holonomic constraint reduces $\dim \mathcal{Q}$ by one: n coordinates with m independent holonomic constraints give an $(n-m)$-dimensional configuration space. A **nonholonomic constraint** is

a velocity constraint $g(q, \dot{q}, t) = 0$ and removes no dimension — Chapter 20's subject. For a **closed chain** of k links and n joints with f_i degrees of freedom at joint i, in an ambient space with N rigid-body DOF (3 planar, 6 spatial), the **mobility** is Grü bler's $M = N(k - n - 1) + \sum_i f_i$, valid when the joint constraints are independent. A robot with more DOF than its task needs is **redundant**.

DERIVATION **Derivation 2 — Dimension by point counting (Choset §3.3)**

\dim \Q = 3 \text{ (planar body)},\; 6 \text{ (spatial body)},\; 12 - 10 = 2 \text{ (Reach)}

Statement. A planar rigid body has three degrees of freedom and a spatial one six; a system of n coordinates with m independent holonomic constraints has $n - m$.
Step 1 — place A freely. Fix three non-collinear points A, B, C on the body. A's position is free: 2 coordinates in the plane, 3 in space.
Step 2 — B is constrained to a circle (sphere). Rigidity fixes $d(A, B)$, so B lies on a circle about A in the plane — one angle θ remains — or on a sphere in space, where two angles remain.
Step 3 — C follows, up to a discrete choice. Two fixed distances $d(A, C)$, $d(B, C)$ place C in the plane up to a reflection: zero continuous freedoms remain. In space, three distances leave C on a circle about the axis AB: one angle remains. Every other point of the body is then fixed by its three distances. Totals: $2 + 1 = 3$ and $3 + 2 + 1 = 6$, with configuration spaces $\mathbb{R}^2 \times S^1$ and $\mathbb{R}^3 \times SO(3)$. ■
Reach, counted the indirect way (Choset's own example). Treat each link as a spatial rigid body: 12 coordinates. Six constraints confine the two links to the table plane, two pin the first joint to a point, and — once link 1's angle is chosen — two pin the second joint to the end of link 1: $12 - 10 = 2$. The direct count is shorter: one revolute joint, one degree of freedom, twice. *A closed chain*: Choset's figure 3.9 has six links and seven revolute joints in the plane, so $M = 3(6 - 7 - 1) + 7 = 1$ — a one-parameter mechanism, like the four-bar linkage of problem 3.7.

Choset's table 3.1 collects the configuration spaces this book plans in, and the inequalities beside it are the ones a planner written against coordinates cannot see:

Robot	Q
Mobile robot translating in the plane (Rusty as a disc)	$\mathbb{R}^2$

Robot	Q
Mobile robot translating and rotating in the plane (Rusty, Hitch's chassis)	$SE(2)$ or $\mathbb{R}^2 \times S^1$
Rigid body translating in space	$\mathbb{R}^3$
A spacecraft	$SE(3)$ or $\mathbb{R}^3 \times SO(3)$
An n-joint revolute arm (Reach)	T^n
A planar mobile robot with an attached n-joint arm	$SE(2) \times T^n$

$S^1 \times \cdots \times S^1 = T^n \neq S^n$; $S^1 \times S^1 \times S^1 \neq SO(3)$; $SE(2) \neq \mathbb{R}^3$; $SE(3) \neq \mathbb{R}^6$. The torus is compact and the plane is not; this chapter *uses* the difference (the blob wraps, the seam is a lie of the chart) and Chapter 5 proves it.

4.3.4 Polygons as intersections of half-planes

DEFINITIONS (CHOSET APP. F.1) — HALF-PLANES, CONVEX POLYGONAL REGION, POLYGON

A line $h(x, y) = ax + by - c = 0$ with unit normal (a, b) splits the plane into the **positive half-plane** $h^+ = \{h \geq 0\}$ and the **negative half-plane** $h^- = \{h \leq 0\}$. A **convex polygonal region** is an intersection of finitely many negative half-planes (by convention; multiply h by -1 to flip a side). A **polygonal region** is a finite union of convex ones, not necessarily bounded or connected. A **polygon** is a closed, simply connected polygonal region — a polygonal region homeomorphic to the closed unit disc.

The convention is what makes the rest of the appendix mechanical. List a convex polygon's vertices counterclockwise; edge i runs from r_i to r_{i+1}; its outward unit normal n_i is the edge direction rotated clockwise by 90°; its half-plane is $h_i(x, y) = n_i \cdot ((x, y) - r_i) \leq 0$. A point is inside iff every $h_i \leq 0$. Choset's figure F.2 is the triangle $h_1 = -x + y - 3$, $h_2 = -y$, $h_3 = x$, whose negative half-planes intersect in the region $x \geq 0$, $y \geq 0$, $y \leq x + 3$ — wait: $x \leq 0$. Read the signs: $h_3 = x \leq 0$ is the *left* half-plane. The convention is easy to say and easy to get backwards, which is why `halfPlaneOfEdge` fixes it once and every polygon in the module reads the same way.

4.3.5 Contacts, and the half-planes they define

DEFINITION (CHOSET APP. F.2) — TYPE A AND TYPE B CONTACTS

Fix the robot's heading θ. Two convex polygons touch without overlapping — eq. F.2, $R(q) \cap W \neq \emptyset$ and $\operatorname{int} R(q) \cap \operatorname{int} W = \emptyset$ — in exactly two ways. A

Type A contact: an edge E_i^R of the robot contains a vertex o_j of the obstacle. A **Type B contact**: an edge E_j^W of the obstacle contains a vertex r_i of the robot. (Both at once, figure F.4(c), is an edge lying along an edge.)

DERIVATION **Derivation 3 — Contact half-planes characterize collision for convex polygons (App. F.2)**

$q \in QO(\theta) \iff f_{ij}^R(q) \le 0 \ \forall$ applicable A, $\quad f_{ij}^W(q) \le 0 \ \forall$ applicable B

Statement. For fixed θ, $QO(\theta) = \{(x, y) : (x, y, \theta) \in QO\}$ is a convex polygon, and $q \in QO(\theta)$ iff every *applicable* contact's half-plane inequality holds. A Type A pair (E_i^R, o_j) is applicable (eq. F.3) when

$$(o_{j-1} - o_j) \cdot n_i^R(\theta) \ge 0 \quad \text{and} \quad (o_{j+1} - o_j) \cdot n_i^R(\theta) \ge 0,$$

equivalently when the *negated* robot normal $-n_i^R$ lies between the normals of the two obstacle edges meeting at o_j; its half-plane is (eq. F.4) $f_{ij}^R(q) = n_i^R(\theta) \cdot (o_j - r_i(x, y, \theta)) \le 0$. Symmetrically a Type B pair (E_j^W, r_i) is applicable (F.5) when $(r_{i-1} - r_i) \cdot n_j^W \ge 0$ and $(r_{i+1} - r_i) \cdot n_j^W \ge 0$, with half-plane (F.6) $f_{ij}^W(q) = n_j^W \cdot (r_i(x, y, \theta) - o_j) \le 0$.

Step 1 — the boundary is where the bodies touch. If the interiors overlap at q, every nearby q' is also in $QO(\theta)$, so q is interior to $QO(\theta)$. Boundary configurations are therefore exactly those satisfying F.2: contact without overlap.

Step 2 — only Type A and Type B realize contact. Two convex polygons whose interiors are disjoint meet in a point or a segment of their boundaries; the meeting set contains a vertex of one body lying on an edge of the other (or an edge along an edge, which is both). That is the dichotomy.

Step 3 — each applicable pair gives a supporting half-plane. Take an applicable Type A pair. The condition F.3 says both obstacle edges leaving o_j point into or along the robot normal's half-space, so the whole obstacle lies on the side $n_i^R \cdot (o - r_i) \ge n_i^R \cdot (o_j - r_i)$; the robot edge E_i^R can touch the obstacle only at o_j. Translating the robot, the contact persists while $n_i^R \cdot (o_j - r_i(q)) = 0$ — a line in the (x, y)-plane — and the robot overlaps the obstacle exactly on the side $f_{ij}^R \le 0$. So $QO(\theta) \subseteq \{f_{ij}^R \le 0\}$ and the line is a *supporting line* of $QO(\theta)$. Type B is the same with roles swapped. Non-applicable pairs give nothing: a robot edge whose normal does not point at o_j cannot touch it from outside.

Step 4 — a convex set is the intersection of its supporting half-planes. $QO(\theta)$ is convex (it is a Minkowski difference of convex sets, Derivation

4), and every edge of its boundary is a contact line from Step 3, so $QO(\theta)$ equals the intersection of the applicable half-planes. Hence $q \in QO(\theta)$ iff all applicable inequalities hold — and *one* failing inequality is a certificate of separation. ■

Why n_i^R depends on θ only. The robot's normals rotate with it and do not move with it; applicability is a function of θ alone, and the star algorithm exploits this by sorting normals once per heading.

Nonconvex bodies. Decompose both into convex pieces $\{R_l\}$, $\{W_k\}$ and test every pair; the union of the pairwise QO's is the C-obstacle (the union property above).

Choset's figure F.7 shows a triangle robot and a four-sided obstacle in two placements and tabulates seven inequalities: all satisfied in (a), one violated in (b). His figure carries no coordinates, so here is the same table with numbers the check `contacts` pins. Robot $r_1 = (0,0)$, $r_2 = (0.6, 0.1)$, $r_3 = (0.15, 0.5)$; obstacle the square $[1,2] \times [0,1]$ with $o_1 = (1,0)$ counterclockwise; heading $\theta = 0$. Three Type A pairs and four Type B pairs are applicable:

Contact pair	**half-plane**	**at $q_a = (0.8, 0.3)$**	**at $q_b = (0.3, 0.3)$**
E_1^R, o_4	$n_1^R \cdot (o_4 - r_1) \le 0$	-0.658 yes	-0.575 yes
E_2^R, o_1	$n_2^R \cdot (o_1 - r_2) \le 0$	-0.565 yes	-0.233 yes
E_3^R, o_2	$n_3^R \cdot (o_2 - r_3) \le 0$	-1.236 yes	-1.715 yes
E_1^W, r_3	$n_1^W \cdot (r_3 - o_1) \le 0$	-0.800 yes	-0.800 yes
E_2^W, r_1	$n_2^W \cdot (r_1 - o_2) \le 0$	-1.200 yes	-1.700 yes
E_3^W, r_1	$n_3^W \cdot (r_1 - o_3) \le 0$	-0.700 yes	-0.700 yes
E_4^W, r_2	$n_4^W \cdot (r_2 - o_4) \le 0$	-0.400 yes	$+0.100$ **no**

At q_a all seven hold: collision. At q_b the obstacle's left edge E_4^W (normal $(-1, 0)$, through $o_4 = (1,1)$) separates the robot's vertex r_2 from the square by 0.1: no collision, and that one line is the proof. The same check runs this test against Chapter 2's separating-axis box on 5,000 seeded poses of random convex pairs — 470 collisions among them — and finds no disagreement. The black box is now a verified box.

4.3.6 The star algorithm

INTERACTIVE · w4.2

Minkowski Studio

For convex polygons the Minkowski difference is a merge of two sorted lists of edge normals — one vertex per step, n_R + n_W steps.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch04-configuration-space-i#w4.2

DERIVATION **Derivation 4 — The star algorithm computes QO(θ) = WO $\ominus$ R(0,0,θ) in linear time after sorting (App. F.3)**

$QO(\theta) = \mathcal{WO} \ominus R(0,0,\theta) = \text{hull}\{\, o_j - r_i(\theta)\,\}$

Statement. Negate the robot's edge normals; merge-sort them with the obstacle's by angle; scan the merged list once. Each time a negated robot normal $-n_i^R$ falls between adjacent obstacle normals, a Type A contact of E_i^R with the vertex o_j between those normals is applicable, and it contributes the vertices $o_j - r_i$ and $o_j - r_{i+1}$; each time an obstacle normal n_j^W falls between adjacent negated robot normals, a Type B contact of E_j^W with r_i is applicable and contributes $o_j - r_i$ and $o_{j+1} - r_i$. The emitted points, in scan order, are the vertices of $\mathcal{QO}(\theta)$ counterclockwise, and $\mathcal{QO}(\theta)$ equals the Minkowski difference (eq. F.7) $\mathcal{WO} \ominus R = \{o - r : o \in \mathcal{WO},\ r \in R\}$.

Step 1 — contacts persist along edges, and their extremes are vertices. Fix an applicable Type A pair. Slide the robot so that o_j stays on edge E_i^R: the configuration traces a segment of the boundary of $\mathcal{QO}(\theta)$. At one extreme o_j coincides with r_i, at the other with r_{i+1}; the reference point is then at $o_j - r_i(0,0,\theta)$ and $o_j - r_{i+1}(0,0,\theta)$. These are two vertices of $\mathcal{QO}(\theta)$, and the edge between them is a translate of E_i^R — with outward normal $-n_i^R$, since the robot is reflected through its reference point. Symmetrically a Type B pair gives the vertices $o_j - r_i$ and $o_{j+1} - r_i$, joined by a translate of E_j^W with normal n_j^W.

Step 2 — angular order of normals is boundary order. The boundary of a convex polygon, walked counterclockwise, has outward normals that rotate monotonically through 2π. The boundary of $\mathcal{QO}(\theta)$ is made of translates of robot edges (normals $-n_i^R$) and obstacle edges (normals n_j^W), so walking it counterclockwise visits those edges in the angular order of $\{-n_i^R\} \cup \{n_j^W\}$. Merging the two already-sorted lists *is* that walk.

Step 3 — the merged scan visits every applicable pair exactly once. Keep a current robot vertex i and obstacle vertex j, starting from the pair extreme in the $+x$ direction. Consuming a robot entry means walking the (reflected)

robot edge E_i^R while o_j stays put — a Type A contact — and advancing i; consuming an obstacle entry means walking E_j^W while r_i stays put — Type B — and advancing j. Each entry is consumed once, so each applicable contact is emitted once and the scan closes on its start after $n_R + n_W$ steps. Cost: $O((n_R+n_W)\log(n_R+n_W))$ for a general sort, $O(n_R+n_W)$ here because each list is already in order.

Step 4 — equality with the hull of all pairwise differences. $\mathcal{WO} \ominus R = \mathcal{WO} \oplus (-R)$ is a Minkowski sum of convex polygons, hence convex, and its extreme points are differences of extreme points, so it is the convex hull of the $n_R n_W$ points $o_j - r_i$. The star algorithm's vertices are among these points and its edges have the correct normals, so the two polygons coincide. That hull is `minkowskiBruteForce`, the oracle the check `star` holds the scan to on 500 seeded convex pairs at random headings: identical polygons, worst area gap 2×10^{-15}. ■

The preset, worked. Robot $(0,0), (0.2,0), (0,0.2)$ at $\theta = 0$; obstacle $[1,2]^2$. The merged list is $-n_3^R$ at 0°, n_1^W at 0°, $-n_1^R$ at 90°, n_2^W at 90°, n_3^W at 180°, $-n_2^R$ at 225°, n_4^W at 270° — seven entries, seven steps. The emitted points are $(2, 0.8), (2, 1), (2, 2), (1.8, 2), (0.8, 2), (0.8, 1), (1, 0.8)$; two of them, $(2, 1)$ and $(1.8, 2)$, lie on edges because a robot edge is parallel to an obstacle edge, and dropping collinear vertices leaves the five the design doc names — $(1, 0.8), (2, 0.8), (2, 2), (0.8, 2), (0.8, 1)$ — with area $1 + 0.2 + 0.2 + 0.02 =$ **1.42**: the square, two strips, and the reflected triangle in the corner. Turn the robot to $\theta = 30°$ and no edges are parallel: **seven** vertices. The check `star` pins all of it.

4.3.7 $SE(2)$: the C-obstacle as stacked slices

A polygon that also rotates has $\mathcal{Q} = SE(2)$, three-dimensional, and its C-obstacle is a solid. Choset's App. F.4 visualizes it the only honest way there is: fix θ, compute the convex slice $\mathcal{QO}(\theta)$ by the star algorithm, and stack the slices along a vertical θ-axis — his figure F.8, a triangle against a five-sided obstacle. The *stack* toggle of the Minkowski Studio does this with 24 slices. Each slice is exact; the stack is a picture. The gap between two slices is the same honesty item as the arm's raster: a path that passes between θ_k and θ_{k+1} visits configurations no slice tested. Chapter 8's visibility graphs and Chapter 10's trapezoidal decomposition need exactly these polygonal slices, and import `se2Slices` for them.

4.3.8 GJK: distance as the distance from a Minkowski difference to the origin

DERIVATION **Derivation 5 — GJK computes the distance between convex polytopes from support functions alone (App. F.5, Algorithm 23)**

$$d(A,B) = \min_{z \in A \ominus B} \|z\|, \qquad h_{A\ominus B}(x) = h_A(x) + h_B(-x), \quad s_{A\ominus B}(x) = s_A(x) - s_B(-x)$$

Statement. $d(A,B) = \min_{a\in A, b\in B} \|a-b\| = \min_{z\in Z} \|z\|$ with $Z = A \ominus B$ (eqs. F.8–F.10); the minimizer z^* is unique; and Gilbert, Johnson and Keerthi's iteration finds it using only the support value $h_Z(x) = \max_{z\in Z} z \cdot x$ and support point $s_Z(x)$ (F.11–F.12), which never require Z to be built.

Step 1 — rewrite over the Minkowski difference. Every $a - b$ is a point of Z and every point of Z is some $a - b$, so the two minima are the same number (F.10). The distance between two bodies is the distance from one convex set to the origin.

Step 2 — uniqueness. Z is convex (Derivation 4, Step 4) and $\|\cdot\|$ is a strictly convex function away from the origin, so $\arg\min_{z\in Z} \|z\|$ is a single point z^*. The witness pair (a^*, b^*) with $a^* - b^* = z^*$ need *not* be unique — two parallel edges facing each other have a whole segment of nearest pairs — and Choset says so after F.10. Uniqueness is of the difference.

Step 3 — the simplex iteration. Keep a working set V_k of at most three points of Z (a point, a segment or a triangle in the plane) and let x_k be the point of hull(V_k) nearest the origin. Ask Z for its support point in direction $-x_k$: $z_k = s_Z(-x_k)$. If $h_Z(-x_k) = -\|x_k\|^2$ — no point of Z projects farther toward the origin than x_k itself — then $x_k = z^*$ and the iteration stops (Algorithm 23, step 4). Otherwise z_k is strictly closer to the origin in the direction $-x_k$, so hull$(V_k \cup \{z_k\})$ contains points nearer than x_k; keep the edge of the old simplex that contained x_k plus z_k (step 6) and repeat. The distance to the origin strictly decreases, the working set never revisits a vertex of Z in exact arithmetic, and Z has at most $n_A + n_B$ vertices, so the loop terminates in at most that many steps. If the origin lands inside the triangle, $d = 0$ and the bodies intersect.

Step 4 — support functions compose, so Z is never built. $h_{A\ominus B}(x) = \max\{(a-b)\cdot x\} = \max_a a \cdot x + \max_b b \cdot (-x) = h_A(x) + h_B(-x)$ (F.13), and if a^*, b^* attain those maxima then $s_{A\ominus B}(x) = a^* - b^*$ (F.14). For a polygon the support point is the vertex with the largest dot product — a scan of n vertices — so one GJK step costs $O(n_A + n_B)$ and the whole query is linear with a tiny constant. ■

What the check pins. `gjk` runs the two-dimensional iteration against Chapter 2's brute-force polygon distance on 500 seeded convex pairs (22 of them intersecting): the distances agree to 4×10^{-16}, the witnesses realize the

distance, the intersection verdict matches the separating-axis test, and no query needs more than 5 iterations — well under the $n_A + n_B$ bound. *What `parry2d` adds.* The engine the Rust crate delegates to runs this same loop with a bounding-volume hierarchy to skip far pairs, EPA (expanding polytope algorithm) to recover the *penetration* depth when the origin is inside Z, and shape-casting — GJK along a motion — for the swept query of Chapter 2's `Collision::swept`. Choset's one-sentence extension to polyhedra is to start from four points and keep a face instead of an edge.

4.3.9 Reach's Jacobian and its singularities

INTERACTIVE · w4.3

Jacobian Lens

A singularity is not where the arm breaks: a direction of tip motion becomes unavailable, and asking for it asks for infinite joint speed.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch04-configuration-space-i#w4.3

Chapter 2 gave Reach's forward kinematics $\varphi : T^2 \to \mathbb{R}^2$, $\varphi(q) = (L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2),\ L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2))$. Choset's §3.8 asks how velocities transform: as the arm moves, $\dot{x} = \frac{\partial \varphi}{\partial q}\dot{q} = J(q)\dot{q}$, where J is the Jacobian of φ, also called its differential $D\varphi$ — the linear map that Chapter 5 will show is independent of the chart.

DEFINITION (CHOSET §3.8) — SINGULAR CONFIGURATION

A configuration q is **singular** when $J(q)$ loses rank. At such a q the n-dimensional set of joint velocities $\dot{q}$ maps to a lower-dimensional set of end-point velocities $\dot{x}$: instantaneous end-point motion is impossible in some direction.

DERIVATION **Derivation 6 — Reach's Jacobian, det J = L_1L_2 sin θ_2, and Example 3.8.1 (Choset §3.8)**

$$J(q) = \begin{bmatrix} -L_1 \sin\theta_1 - L_2 \sin(\theta_1 + \theta_2) & -L_2 \sin(\theta_1 + \theta_2) \\ L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2) & L_2 \cos(\theta_1 + \theta_2) \end{bmatrix}, \quad \det J = L_1 L_2 \sin\theta_2$$

Statement. The matrix above is $\partial\varphi/\partial q$; its determinant is $L_1 L_2 \sin\theta_2$; the arm is singular exactly when $\sin\theta_2 = 0$ — straight or folded. Choset's

numbers: $L_1 = L_2 = 1$, $q = (\pi/4, \pi/2)$, $\dot{q} = (1, 0)^\mathsf{T}$ give $J = \begin{bmatrix} -\sqrt{2} & -\sqrt{2}/2 \\ 0 & -\sqrt{2}/2 \end{bmatrix}$, $\det J = 1$, and $\dot{x} = (-\sqrt{2}, 0)^\mathsf{T}$.

Step 1 — differentiate entrywise. $\partial\varphi_1/\partial\theta_1 = -L_1 \sin\theta_1 - L_2 \sin(\theta_1 + \theta_2)$; $\partial\varphi_1/\partial\theta_2 = -L_2 \sin(\theta_1 + \theta_2)$; $\partial\varphi_2/\partial\theta_1 = L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2)$; $\partial\varphi_2/\partial\theta_2 = L_2 \cos(\theta_1 + \theta_2)$. Read the columns: column i is the tip velocity when joint i alone turns at unit rate — every link beyond joint i swings about it, so the column is the sum over those links of L_k times the link's heading rotated by 90°. That sentence is the implementation of `jacobian`, and it serves any number of links.

Step 2 — the determinant. Write $s_1 = \sin\theta_1$, $c_1 = \cos\theta_1$, $s_{12} = \sin(\theta_1 + \theta_2)$, $c_{12} = \cos(\theta_1 + \theta_2)$. Then $\det J = (-L_1 s_1 - L_2 s_{12})(L_2 c_{12}) - (-L_2 s_{12})(L_1 c_1 + L_2 c_{12}) = -L_1 L_2 s_1 c_{12} + L_1 L_2 s_{12} c_1 = L_1 L_2 \sin((\theta_1 + \theta_2) - \theta_1) = L_1 L_2 \sin\theta_2$ by the sine difference identity.

Step 3 — what rank loss means. At $\sin\theta_2 = 0$ the two columns are parallel (both are multiples of $(-\sin\theta_1, \cos\theta_1)$ when the arm is straight), so the image of $\dot{q} \mapsto J\dot{q}$ is a line perpendicular to the arm. Motion *along* the arm is instantaneously impossible — the tip is at the edge of the annulus and can only move tangentially. The arm has not broken; one direction is unavailable, and a planner that asks for it asks for $\dot{q} = J^{-1}\dot{x}$ with $\det J \to 0$: infinite joint speed. Chapters 17–18 meet this as a velocity limit that goes to zero; Chapter 21 as a rank condition.

Step 4 — Choset's numbers. At $q = (\pi/4, \pi/2)$: $\theta_1 + \theta_2 = 3\pi/4$, $s_1 = c_1 = \sqrt{2}/2$, $s_{12} = \sqrt{2}/2$, $c_{12} = -\sqrt{2}/2$. So $J_{11} = -\sqrt{2}/2 - \sqrt{2}/2 = -\sqrt{2}$, $J_{12} = -\sqrt{2}/2$, $J_{21} = \sqrt{2}/2 - \sqrt{2}/2 = 0$, $J_{22} = -\sqrt{2}/2$; $\det J = \sin(\pi/2) = 1$; and $J(1, 0)^\mathsf{T} = (-\sqrt{2}, 0)^\mathsf{T} = (-1.4142, 0)$ — the tip, at $(0, \sqrt{2})$, moves straight left when the shoulder alone turns, as his figure 3.21 draws. The check `jacobian` asserts the four entries, the determinant and $\dot{x}$ to 10^{-12}, and holds `jacobian` to central differences of `Reach::fk` on 300 seeded arms to 10^{-7}. ■

The velocity ellipse. The Jacobian Lens maps the unit circle of joint velocities through J: an ellipse with semi-axes the singular values $\sigma_1 \geq \sigma_2$ of J and area $\pi\sigma_1\sigma_2 = \pi|\det J|$ — Yoshikawa's manipulability. At $\theta_2 = 10°$ (any θ_1, any $L_1 = L_2 = 1$): $\sigma_1\sigma_2 = \sin 10° = 0.1736$, $\sigma_1^2 + \sigma_2^2 = \|J\|_F^2 = 3 + 2\cos 10° = 4.9696$, so $\sigma_1 = 2.228$, $\sigma_2 = 0.078$ and the ellipse is **28.6** times longer than it is wide; at $\theta_2 = 0$ it is a segment. Exercise 4 asks for this number before the widget shows it.

Example 3.8.2, the polygon. A point fixed at $r = (r_1, r_2)$ on a body with configuration $q = (q_1, q_2, q_3) \in \mathbb{R}^2 \times S^1$ sits at $x = (q_1, q_2) + R(q_3)\, r$, with Jacobian

$$J(q) = \begin{bmatrix} 1 & 0 & -r_1 \sin q_3 - r_2 \cos q_3 \\ 0 & 1 & r_1 \cos q_3 - r_2 \sin q_3 \end{bmatrix},$$

a 2×3 matrix of rank 2 *always*: a rigid body in the plane has no singular configurations, and φ^{-1} is one-to-many because $\dim Q > \dim M$. The third column is the lever arm rotated by 90°, the velocity of the point under pure rotation. `bodyPointJacobian` implements it and the same check differences `bodyPointFk` against it.

J^T is the force map. Power is coordinate-free: $\tau \cdot \dot{q} = f \cdot \dot{x} = f \cdot J\dot{q}$ for every $\dot{q}$, so $\tau = J^\mathsf{T} f$. The widget's J^T toggle shows a unit tip force becoming joint torques; Chapter 7 lifts workspace potentials through this map and Chapter 17 makes it dynamics.

4.4 The algorithm

Four algorithms, each exact for the bodies it covers, in the order the chapter met them. Choset numbers only the last; the others are App. F.2, F.3 and §3.2.2 written as procedures.

ALGORITHM `RASTERIZE-C-OBSTACLE (Choset §3.2.2, the pixel method)` COST $O(n_1 n_2)$ collision queries -- 129,600 at 1° on T^2

in a Collision checker for the robot in its scene; a 2-D chart of Q with $n_1 \times n_2$ cell centers q(i, j) and a flag per axis saying whether it wraps

out a BitGrid: cell (i, j) set iff q(i, j) ∈ QO; its connected components on the torus

1: for $i = 0 \ldots n_1 - 1$, $j = 0 \ldots n_2 - 1$ do
2: grid$[i, j] \leftarrow \neg$ isFree$(q(i, j))$ — one query per cell center; the checker decides what "the robot" is
3: components ← flood fill over 4-neighbors, wrapping across the seams the chart marks periodic — connectivity measured, not assumed
4: return grid, components — a free cell is a free *point*: thicken the footprint by the cell's reach if free cells must certify paths between them

ALGORITHM `POLYGON-INTERSECTS (Choset App. F.2)` COST `O(n_R n_W)` as written; `O(n_R + n_W)` with the normals merged as in STAR

in convex robot R (vertices r_i CCW, normals n^R_i), configuration q = (x, y, θ), convex obstacle W (vertices o_j, normals n^W_j)

out true iff R(q) ∩ W ≠ ∅

1: for each robot edge E_i^R and obstacle vertex o_j: if $(o_{j-1} - o_j) \cdot n_i^R(\theta) \geq 0$ and $(o_{j+1} - o_j) \cdot n_i^R(\theta) \geq 0$ — applicable Type A (F.3)
2: if $n_i^R(\theta) \cdot (o_j - r_i(q)) > 0$ return **false** — one violated half-plane (F.4) separates the bodies
3: for each obstacle edge E_j^W and robot vertex r_i: if $(r_{i-1} - r_i) \cdot n_j^W \geq 0$ and $(r_{i+1} - r_i) \cdot n_j^W \geq 0$ — applicable Type B (F.5)
4: if $n_j^W \cdot (r_i(q) - o_j) > 0$ return **false** — (F.6)
5: return **true** — every applicable inequality holds: $q \in QO(\theta)$ (Derivation 3)

ALGORITHM `STAR (Choset App. F.3): QO(θ) = WO ⊖ R(0, 0, θ)` COST `O((n_R + n_W) log(n_R + n_W)); O(n_R + n_W) since each list is already sorted`

in convex robot R with reference point at the origin, heading θ, convex obstacle W, both CCW
out the vertices of QO(θ) counterclockwise, each tagged with the contact that produced it

1: rotate R by θ; compute normals $n_i^R(\theta)$, n_j^W
2: $L_R \leftarrow$ the angles of $-n_i^R$; $L_W \leftarrow$ the angles of n_j^W; each cyclically sorted — rotate so each starts at its smallest angle
3: merge L_R and L_W by angle into one list of $n_R + n_W$ entries
4: $i \leftarrow$ the robot edge first in L_R, $j \leftarrow$ the obstacle edge first in L_W; emit $o_j - r_i$ — the pair extreme in $+x$
5: for each entry of the merged list, in order:
6: if it is robot edge i: $i \leftarrow i + 1$; emit $o_j - r_i$ — Type A: E_i^R slid along o_j (Derivation 4, Step 1)
7: else it is obstacle edge j: $j \leftarrow j + 1$; emit $o_j - r_i$ — Type B: E_j^W slid along r_i
8: drop the closing duplicate and any collinear vertex; return the list — equals hull$\{o_j - r_i\}$, the brute-force oracle

ALGORITHM `GJK (Choset Algorithm 23) on Z = A ⊖ B` COST `each step O(n_A + n_B); at most n_A + n_B steps in exact arithmetic -- 5 on the chapter's 500 random pairs`

in convex polygons A, B through their support functions h, s; a tolerance
out d(A, B) = min_{z ∈ Z} ‖z‖, the unique minimizer z, *and a witness pair (a*, b*)

1: $V_0 \leftarrow \{s_Z(d_0)\}$ for any direction d_0 — Choset seeds with three vertices of Z; one support point works equally and never needs Z listed
2: $k \leftarrow 0$
3: $x_k \leftarrow \arg\min_{x \in \text{hull}(V_k)} \|x\|$ — nearest point of a point, segment or triangle; if the origin is inside the triangle return $d = 0$
4: if $\|x_k\|^2 = h_Z(-x_k) \cdot (-1)$, i.e. $h_Z(-x_k) = -\|x_k\|^2$ (within tolerance): return $\|x_k\|$ with the witnesses from x_k's barycentric weights
5: $z_k \leftarrow s_Z(-x_k) = s_A(-x_k) - s_B(x_k)$ — (F.14); the point of Z farthest toward the origin
6: $V_{k+1} \leftarrow$ the edge (or point) of hull(V_k) containing x_k, plus z_k
7: $k \leftarrow k + 1$; go to 3

Line 4 of GJK is the step worth a sentence. $h_Z(-x_k) = \max_z z \cdot (-x_k)$; if that maximum is attained at x_k itself the value is $-\|x_k\|^2$, and no point of Z lies on the origin's side of the supporting line through x_k perpendicular to x_k. That line is a certificate — the same kind of certificate as the single violated half-plane in POLYGON-INTERSECTS, and the same geometry: a supporting half-plane of a convex set.

4.5 Implementation in Rust

Crates. `nalgebra` 0.35 for `Point2`, `Vector2`, `Matrix2`, `Isometry2`; `parry2d` 0.30 as `parry2d-f64`, still the black box for GJK/EPA and shape-casting, now *explained*; `collide` and `robots` from Chapter 2; `rand` 0.9 `SmallRng` for seeded polygon fields; `proptest` as a dev dependency for the star-versus-brute-force property test; `k` and `urdf-rs` as **test-only** dependencies that load a URDF of Reach and cross-check `jacobian()`.

```
crates/cspace/
  src/lib.rs
  src/polygon.rs      # App. F.1: CCW convex polygons, half-planes, normals derived never
    stored
  src/contact.rs      # App. F.2: applicable contacts, the half-plane test, the F.7 table
  src/star.rs         # App. F.3-F.4: star algorithm, brute-force oracle, inflate_disc,
   se2_slices
  src/raster.rs       # sec.3.2.2: FootprintAt<Q>, TorusGrid, BitGrid,
   rasterize_cobstacle
  src/jacobian.rs     # sec.3.8: jacobian, det_jacobian, body_point_jacobian
  examples/reach_square.rs      # the micro-example, printed
  examples/workbench_map.rs     # the integration lab
  tests/{star_vs_brute.rs, jacobian_vs_k.rs, contact_table_f7.rs, black_box_agrees.rs}
web/lib/cspace/{polygon,star,gjk,raster,jacobian,examples}.ts   # the port the widgets
   run
```

4.5.1 Polygons and contacts

The polygon type enforces the one convention everything depends on — counterclockwise, convex — in its constructor, and derives normals on demand so they can never go stale after a transform.

RUST `crates/cspace/src/polygon.rs`

```rust
use nalgebra::{Point2, Vector2};
use geom::Pose2;

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct NotConvexOrNotCcw;

/// A convex polygon as Choset's App. F.1 writes it: an intersection of negative
/// half-planes, stored as its counterclockwise vertex list. Edge i runs r_i -> r_{i+1}.
#[derive(Debug, Clone, PartialEq)]
pub struct Polygon { verts: Vec<Point2<f64>> }

impl Polygon {
    /// The only constructor. Rejecting clockwise or nonconvex input here means no
    /// function below ever has to re-check the winding it relies on.
    pub fn new_ccw(verts: Vec<Point2<f64>>) -> Result<Self, NotConvexOrNotCcw> {
        if verts.len() < 3 || signed_area(&verts) <= 0.0 || !is_convex(&verts) {
            return Err(NotConvexOrNotCcw);
        }
        Ok(Polygon { verts })
    }
    pub fn len(&self) -> usize { self.verts.len() }
    pub fn vertex(&self, i: usize) -> Point2<f64> { self.verts[i % self.verts.len()] }

    /// Outward unit normal of edge i: the edge direction rotated clockwise by 90?.
    /// Derived, never stored -- a stored normal is stale after the next `transformed`.
    pub fn edge_normal(&self, i: usize) -> Vector2<f64> {
        let e = self.vertex(i + 1) - self.vertex(i);
        Vector2::new(e.y, -e.x).normalize()
    }

    /// h_i(x, y) = n_i . ((x, y) - r_i): negative inside, by the F.1 convention.
    pub fn half_plane(&self, i: usize, p: &Point2<f64>) -> f64 {
        self.edge_normal(i).dot(&(p - self.vertex(i)))
    }

    /// r_i(x, y, theta) = R(theta) r_i + (x, y): the robot's vertices at configuration q.

    pub fn transformed(&self, pose: &Pose2) -> Polygon {
        Polygon { verts: self.verts.iter().map(|v| pose.act(v)).collect() }
    }
}
```

The contact test is written out so the reader sees the box's insides: applicability first (F.3, F.5), then one half-plane value per applicable pair (F.4, F.6). It is $O(n_R n_W)$ and makes no attempt to be fast; its job is to make `parry2d`'s answer a *verified* answer.

RUST `crates/cspace/src/contact.rs`

```rust
use super::polygon::Polygon;
use geom::Pose2;

/// The two ways convex bodies touch without overlapping (Choset F.2, figure F.4).
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum ContactKind {
    /// Robot edge E^R_i contains obstacle vertex o_j.
    TypeA { robot_edge: usize, obs_vertex: usize },
    /// Obstacle edge E^W_j contains robot vertex r_i.
    TypeB { obs_edge: usize, robot_vertex: usize },
}

/// Eqs. F.3 and F.5. Only theta enters: the robot's normals rotate with it, its position does not.
pub fn applicable_contacts(robot: &Polygon, theta: f64, obs: &Polygon) -> Vec<ContactKind> {
    let r = robot.transformed(&Pose2::new(0.0, 0.0, theta));
    let mut out = Vec::new();
    for i in 0..r.len() {
        let n = r.edge_normal(i);
        for j in 0..obs.len() {
            let (o, prev, next) = (obs.vertex(j), obs.vertex(j + obs.len() - 1), obs.vertex(j + 1));
            // Both obstacle edges leaving o_j point along or away from the robot normal:
            // -n^R_i lies between the normals of the edges adjacent to o_j.
            if (prev - o).dot(&n) >= 0.0 && (next - o).dot(&n) >= 0.0 {
                out.push(ContactKind::TypeA { robot_edge: i, obs_vertex: j });
            }
        }
    }
    for j in 0..obs.len() {
        let n = obs.edge_normal(j);
        for i in 0..r.len() {
            let (v, prev, next) = (r.vertex(i), r.vertex(i + r.len() - 1), r.vertex(i + 1));
            if (prev - v).dot(&n) >= 0.0 && (next - v).dot(&n) >= 0.0 {
                out.push(ContactKind::TypeB { obs_edge: j, robot_vertex: i });
            }
        }
    }
    out
}

/// f^R_ij(q) = n^R_i(theta) . (o_j - r_i(q))  or  f^W_ij(q) = n^W_j . (r_i(q) - o_j)   (F.4, F.6).
pub fn contact_value(c: ContactKind, robot: &Polygon, q: &Pose2, obs: &Polygon) -> f64 {
    let r = robot.transformed(q);
    match c {
        ContactKind::TypeA { robot_edge: i, obs_vertex: j } => r.edge_normal(i).dot(&(obs.vertex(j) - r.vertex(i))),
        ContactKind::TypeB { obs_edge: j, robot_vertex: i } => obs.edge_normal(j).dot(&(r.vertex(i) - obs.vertex(j))),
    }
}

/// App. F.2: q in QO(theta) iff every applicable half-plane constraint holds. One violation
/// is a certificate of separation -- the supporting line that keeps the bodies apart.
pub fn polygon_intersects(robot: &Polygon, q: &Pose2, obs: &Polygon) -> bool {
    applicable_contacts(robot, q.theta, obs)
        .into_iter()
```

```
        .all(|c| contact_value(c, robot, q, obs) <= 0.0)
}
```

4.5.2 The star algorithm, and its oracle

This is the chapter's owned artifact — Chapters 8 and 10 import it and never reimplement it — so it is written to be checked, not merely used. Every emitted vertex carries the contact that produced it, and the brute-force hull sits beside it in the same file.

RUST **crates/cspace/src/star.rs**

```rust
use nalgebra::{Point2, Vector2};
use super::polygon::{convex_hull, Polygon};
use geom::Pose2;

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum Source { Robot, Obstacle }

/// One entry of the merged normal list: whose edge, which edge, at what angle in [0, 2pi
   ).
#[derive(Debug, Clone, Copy)]
struct Normal { source: Source, edge: usize, angle: f64 }

fn angle_of(v: Vector2<f64>) -> f64 {
    let a = v.y.atan2(v.x);
    if a < 0.0 { a + std::f64::consts::TAU } else { a }
}

/// Cyclic rotation so a monotone-but-wrapped list starts at its smallest angle.
fn start_at_min(mut list: Vec<Normal>) -> Vec<Normal> {
    let k = (0..list.len()).min_by(|&a, &b| list[a].angle.total_cmp(&list[b].angle)).
   unwrap_or(0);
    list.rotate_left(k);
    list
}

/// QO(theta) = WO ? R(0, 0, theta) by Choset's star algorithm (App. F.3).
///
/// WO ? R = WO ? (-R), and the boundary of a Minkowski sum of convex polygons is
/// made of both bodies' edges in order of outward normal. -R's normals are R's
/// negated, so: negate, merge by angle, walk the merged list. Consuming a robot
/// entry slides E^R_i along the current obstacle vertex (Type A) and advances i;
/// consuming an obstacle entry slides E^W_j along the current robot vertex (Type B)
/// and advances j. Both lists are already sorted, so this is one merge: O(n_R + n_W).
pub fn star_algorithm(robot: &Polygon, theta: f64, obs: &Polygon) -> Polygon {
    let r = robot.transformed(&Pose2::new(0.0, 0.0, theta));
    let rs = start_at_min((0..r.len()).map(|i| Normal { source: Source::Robot, edge: i,
   angle: angle_of(-r.edge_normal(i)) }).collect());
    let os = start_at_min((0..obs.len()).map(|j| Normal { source: Source::Obstacle, edge:
    j, angle: angle_of(obs.edge_normal(j)) }).collect());

    // The merge.
    let mut merged = Vec::with_capacity(rs.len() + os.len());
    let (mut a, mut b) = (0, 0);
    while a < rs.len() || b < os.len() {
```

```
        if b >= os.len() || (a < rs.len() && rs[a].angle <= os[b].angle) { merged.push(rs[a]); a += 1; }
        else { merged.push(os[b]); b += 1; }
    }

    // The scan. (i, j) start at the vertex pair extreme in +x; o_j - r_i is a vertex of QO(theta).
    let (mut i, mut j) = (rs[0].edge, os[0].edge);
    let diff = |j: usize, i: usize| Point2::from(obs.vertex(j) - r.vertex(i).coords);
    let mut verts = vec![diff(j, i)];
    for e in &merged {
        match e.source {
            // Type A: E^R_i slides along o_j, from o_j - r_i to o_j - r_{i+1}.
            Source::Robot => { i = (e.edge + 1) % r.len(); verts.push(diff(j, i)); }
            // Type B: E^W_j slides along r_i, from o_j - r_i to o_{j+1} - r_i.
            Source::Obstacle => { j = (e.edge + 1) % obs.len(); verts.push(diff(j, i)); }
        }
    }
    verts.pop(); // the scan closes on its start
    Polygon::new_ccw(remove_collinear(verts)).expect("the merge-scan of convex inputs is convex and CCW")
}

/// Eq. F.7 read literally: the convex hull of all n_R . n_W pairwise differences o_j - r_i.
/// O(n_R n_W log) -- wasteful in a planner, exactly right as the oracle `star_algorithm` must equal.
pub fn minkowski_bruteforce(robot: &Polygon, theta: f64, obs: &Polygon) -> Polygon {
    let r = robot.transformed(&Pose2::new(0.0, 0.0, theta));
    let pts: Vec<Point2<f64>> = (0..obs.len()).flat_map(|j| (0..r.len()).map(move |i| (j, i)))
        .map(|(j, i)| Point2::from(obs.vertex(j) - r.vertex(i).coords)).collect();
    convex_hull(&pts)
}

/// sec.3.2.1: WO ? B_r for a convex polygon -- edges offset by r, vertices replaced by arcs.
pub fn inflate_disc(obs: &Polygon, r: f64, arc_segments: usize) -> Polygon { /* as in the TypeScript port */ }

/// App. F.4: QO ? SE(2) as n_theta exact slices QO(theta_k). The stack is a picture; each slice is a theorem.
pub fn se2_slices(robot: &Polygon, obs: &Polygon, n_theta: usize) -> Vec<(f64, Polygon)> {
    (0..n_theta).map(|k| {
        let theta = -std::f64::consts::PI + std::f64::consts::TAU * k as f64 / n_theta as f64;
        (theta, star_algorithm(robot, theta, obs))
    }).collect()
}
```

4.5.3 The rasterizer: any footprint, any chart

Choset's pixel method, generic over the configuration type. The rasterizer never learns what an arm is; it asks a `Collision` checker one question per cell. `thicken` is his §3.2.2 footnote made a parameter — zero for the micro-example's exact

segments, the capsule radius for the Workbench.

RUST `crates/cspace/src/raster.rs`

```rust
use collide::{Collision, Footprint};

/// Any robot that can produce a footprint from a chart point. Reach<2> implements it
/// over T2; a disc over R2; nothing in this file knows which.
pub trait FootprintAt<Q> {
    fn footprint_at(&self, q: &Q) -> Footprint;
}

/// Cells centered on a uniform grid of [-pi, pi)2: with n = 360 the centers sit at integer
/// degrees, which is what makes "29 cells in the theta_1 = 0? row" a statement about this grid.
pub struct TorusGrid { pub n1: usize, pub n2: usize }

impl TorusGrid {
    pub fn point(&self, i: usize, j: usize) -> T2 {
        T2::new(-PI + i as f64 * TAU / self.n1 as f64, -PI + j as f64 * TAU / self.n2 as f64)
    }
}

/// One bit per cell: set iff the cell's configuration is in Q0.
pub struct BitGrid { pub n1: usize, pub n2: usize, bits: Vec<u64> }

impl BitGrid {
    pub fn get(&self, i: usize, j: usize) -> bool { let k = i * self.n2 + j; self.bits[k / 64] >> (k % 64) & 1 == 1 }
    pub fn row_count(&self, i: usize) -> usize { (0..self.n2).filter(|&j| self.get(i, j)).count() }
    /// 4-connected components of the occupied (or free) cells with both seams identified:
    /// connectivity *measured* before Chapter 5 states what the torus is.
    pub fn components_torus(&self, occupied: bool) -> usize { /* flood fill, wrapping both axes */ }
}

/// Choset's pixel method. A free cell is a free *point*, not a free square: `thicken` > 0
/// inflates the footprint so a free cell certifies a free neighborhood. Conservative --
/// the planner is then resolution complete, never complete (his footnote 3, p. 47).
pub fn rasterize_cobstacle<Q>(
    robot: &dyn FootprintAt<Q>, scene: &dyn Collision, grid: &TorusGrid,
    chart: impl Fn(usize, usize) -> Q, thicken: f64,
) -> BitGrid {
    let mut out = BitGrid::new(grid.n1, grid.n2);
    for i in 0..grid.n1 {
        for j in 0..grid.n2 {
            let fp = robot.footprint_at(&chart(i, j)).thickened(thicken);
            if scene.intersects(&fp) { out.set(i, j); }
        }
    }
    out
}
```

4.5.4 The Jacobian

RUST **crates/cspace/src/jacobian.rs**

```rust
use nalgebra::{Matrix2, Matrix2x3, Vector2};
use robots::{Reach, T2};

/// J(q) = ?phi/?q for the 2R arm (Choset Example 3.8.1). Column i is the tip velocity
   when
/// joint i alone turns at unit rate: every link beyond joint i swings about it, so the
/// column is Sigma_{k >= i} L_k . (heading of link k rotated by 90?).
pub fn jacobian(reach: &Reach<2>, q: &T2) -> Matrix2<f64> {
    let [l1, l2] = reach.lengths;
    let (s1, c1) = q.th1().sin_cos();
    let (s12, c12) = (q.th1() + q.th2()).sin_cos();
    Matrix2::new(
        -l1 * s1 - l2 * s12, -l2 * s12,
         l1 * c1 + l2 * c12,  l2 * c12,
    )
}

/// det J = L_1 L_2 sin theta_2 -- computed from the matrix; the identity is a test, not
   an assumption.
pub fn det_jacobian(reach: &Reach<2>, q: &T2) -> f64 { jacobian(reach, q).determinant() }

/// Singular iff J loses rank: for 2R, sin theta_2 = 0 -- arm straight or folded.
pub fn is_singular(reach: &Reach<2>, q: &T2, tol: f64) -> bool { det_jacobian(reach, q).
   abs() <= tol }

/// Example 3.8.2: a point r on a planar rigid body. Rank 2 always -- never singular.
pub fn body_point_jacobian(r: Vector2<f64>, q3: f64) -> Matrix2x3<f64> {
    let (s, c) = q3.sin_cos();
    Matrix2x3::new(1.0, 0.0, -r.x * s - r.y * c,
                   0.0, 1.0,  r.x * c - r.y * s)
}
```

4.5.5 The worked example as a program

RUST **crates/cspace/examples/reach_square.rs**

```rust
use collide::Scene;
use cspace::{jacobian, rasterize_cobstacle, TorusGrid};
use nalgebra::{Point2, Vector2};
use robots::{Reach, T2};
use std::f64::consts::{FRAC_PI_2, FRAC_PI_4};

fn main() {
    let arm = Reach::<2>::choset();                       // L_1 = L_2 = 1, link_radius =
    0
    let scene = Scene::empty().with_rect("the square", 1.4, -0.1, 1.6, 0.1);
    let verdict = |q: T2| if scene.intersects(&arm.footprint_at(&q)) { "collision" } else
    { "free" };

    let deg = |d: f64| d.to_radians();
    println!("q=(0?,0?) {}   q=(0?,13?) {}   q=(0?,15?) {}", verdict(T2::new(0.0, 0.0)),
   verdict(T2::new(0.0, deg(13.0))), verdict(T2::new(0.0, deg(15.0))));
```

```
    let tip = arm.fk(&T2::new(FRAC_PI_4, -FRAC_PI_2)).tip();
    println!("q=(45?,-90?) {} (tip ({:.4}, {:.4}) inside)   q=(90?,*) {} (nearest {:.3} > L2)",
        verdict(T2::new(FRAC_PI_4, -FRAC_PI_2)), tip.x, tip.y, verdict(T2::new(FRAC_PI_2, 1.0)), (1.4f64.powi(2) + 0.9f64.powi(2)).sqrt());

    let grid = TorusGrid { n1: 360, n2: 360 };            // cells centered at integer degrees
    let bits = rasterize_cobstacle(&arm, &scene, &grid, |i, j| grid.point(i, j), 0.0);
    let half = (0.1f64 / 0.4).atan();
    println!("slice theta1=0?: {} occupied cells; analytic half-width atan(1/4) = {:.4} rad = {:.2}?", bits.row_count(180), half, half.to_degrees());
    let rows: Vec<i64> = (0..360).filter(|&i| bits.row_count(i) > 0).map(|i| i as i64 - 180).collect();
    println!("theta1 extent: rows 0..+/-{}? occupied, +/-{}? empty; analytic bound {:.3} rad = {:.1}?", rows.iter().max().unwrap(), rows.iter().max().unwrap() + 1, cspace::THETA1_EXTENT, cspace::THETA1_EXTENT.to_degrees());
    println!("occupied: {} of {} cells ({:.2} %)", bits.count(), 360 * 360, 100.0 * bits.count() as f64 / 129_600.0);
    println!("components on T2: {} ({})", bits.components_torus(true), if bits.touches_seam() { "wraps" } else { "does not wrap" });

    let q = T2::new(FRAC_PI_4, FRAC_PI_2);
    let j = jacobian(&arm, &q);
    let xdot = j * Vector2::new(1.0, 0.0);
    println!("Jacobian at (45?,90?): [[{:.4}, {:.4}],[{:.0}, {:.4}]]  det = {:.4}  J.(1,0) = ({:.4}, {:.0})", j[(0,0)], j[(0,1)], j[(1,0)], j[(1,1)], j.determinant(), xdot.x, xdot.y);
}
```

```
q=(0?,0?) collision   q=(0?,13?) collision   q=(0?,15?) free
q=(45?,-90?) collision (tip (1.4142, 0.0000) inside)   q=(90?,*) free (nearest 1.664 > L2)
slice theta1=0?: 29 occupied cells; analytic half-width atan(1/4) = 0.2450 rad = 14.04?
theta1 extent: rows 0..+/-49? occupied, +/-50? empty; analytic bound 0.864 rad = 49.5?
occupied: 2015 of 129600 cells (1.55 %)
components on T2: 1 (does not wrap)
Jacobian at (45?,90?): [[-1.4142, -0.7071],[0, -0.7071]]  det = 1.0000  J.(1,0) = (-1.4142, 0)
```

The tests beside the example are the chapter's contract. `reach_square_samples` asserts the five hand-checked cells; `slice_row_count_is_29` and `extent_rows` assert the raster counts; `slice_halfwidth_analytic` bisects the exact segment test along $\theta_1 = 0$ and asserts $\arctan\frac{1}{4}$ to 10^{-9}; `jacobian_matches_choset_example` asserts J, $\det J = 1$ and $\dot{x} = (-\sqrt{2}, 0)$ to 10^{-12}; `jacobian_is_fk_derivative` checks central differences of `Reach::fk`; `jacobian_vs_k` loads `reach.urdf` through the `k` crate and asserts agreement at 1,000 seeded q to 10^{-9}; `star_equals_bruteforce` is a `proptest` over 500 seeded convex pairs and headings; `star_triangle_square` asserts the five vertices and area 1.42; `inflate_area` asserts 3.7854; `contact_table_f7` reproduces the two seven-row tables above; and `black_box_agrees` asserts `Scene::intersects` — `parry2d`'s GJK — equals `polygon_intersects` on 10,000 seeded poses. The TypeScript port in `web/lib/cspace/` runs the same seventeen checks; every widget on this page is that port,

and every number printed above was produced by it.

> **NOTE**
>
> The web port hand-rolls GJK (`gjk.ts`) because the browser has no `parry2d`; the Rust crate does not, because it has. Both are held to the same brute-force polygon distance. That is the division of labor this book keeps throughout: geometry engines are *delegated and explained*, planners are *written*. The `cspace` crate is the explanation.

4.6 Putting it together: a Workbench whose free space is in two pieces

The integration lab asks one question of the full Workbench: is Reach's free space connected? Load the *lab* preset of the C-Space Morph. It takes the Workbench of Chapter 2 — block, post, shelf — with the block slid toward the base to $(0.75, 0.35)$ so that link 1 can reach it, and adds a **clamp**, a 0.3 m square at $(-0.55, 0.55)$ on the far side of the base. The arm has its 3 cm capsule links, and the raster runs at 1° in the check (`workbenchLabRaster`) and 2° in the widget.

Two things appear on the torus that the micro-example never showed. First, each near obstacle cuts a **vertical band**: link 1 collides with the block for $\theta_1 \in [8°, 47°]$ and with the clamp for $\theta_1 \in [118°, 152°]$, and a link-1 collision does not care about θ_2, so those columns are forbidden in full. Second, the two bands together do what one band cannot: a single band removes an annulus from the torus and leaves a connected annulus behind, but two bands cut that annulus into two strips. The readout confirms it — $\mathcal{QO}$ has 3 components (33.5% of the torus), and $\mathcal{Q}_{\text{free}}$ has **2**. Switch the lab obstacles off and $\mathcal{Q}_{\text{free}}$ is **1** again; the block's band alone, the post's and the shelf's blobs, all merge into one free component.

The lab's query is $q_{\text{start}} = (90°, 57°)$ and $q_{\text{goal}} = (-90°, 57°)$, both free, both drawn, and in different components. No planner in this book will connect them. Not Chapter 6's A*, which is complete and will say so after expanding every free cell; not Chapter 7's potential field, which will sit in a local minimum against a band; not Chapter 11's PRM, whose roadmap will have two components forever. The first thing a planner must respect is the connectivity of $\mathcal{Q}_{\text{free}}$, and this page has now *measured* it, on a raster, before Chapter 5 says what connectivity means on a torus. The check `lab` pins the band edges, the component counts with and without the clamp, the start and goal labels, and the occupancy.

A last honesty item, because the lab makes it concrete. The Workbench as Chapter 2 ships it — block at $[0.5, 1.0] \times [1.0, 1.5]$, post, shelf — keeps every obstacle beyond link 1's reach: its nearest corner is 1.12 m from the base and $L_1 + 0.03 = 1.03$. Its $\mathcal{Q}_{\text{free}}$ is **connected** (the default *Workbench* preset shows one free component), and a design that promised a disconnected Workbench would

have been wrong by exactly the amount this chapter teaches: whether a C-space is in one piece is not visible from the workspace picture. You have to compute it.

4.7 Exercises

1 FOUNDATION •• **Convexity, and why one checker serves a whole scene**

Prove that the C-obstacle of a convex robot translating in the plane against a convex obstacle is convex (Choset problem 3.4), and that the union operator propagates from the workspace to the configuration space, $\mathcal{QO}(\mathcal{WO}_i \cup \mathcal{WO}_j) = \mathcal{QO}_i \cup \mathcal{QO}_j$ (problem 3.5). Then say in one sentence why the second fact lets `rasterize_cobstacle` take a whole `Scene` rather than one obstacle at a time, and why the first fact does *not* extend to a rotating robot.

> **HINTS**
>
> <>For convexity, write $\mathcal{QO} = \{q : (q + R) \cap \mathcal{WO} \neq \emptyset\} = \mathcal{WO} \oplus (-R)$ and show that a Minkowski sum of convex sets is convex: if $q = o + r$ and $q' = o' + r'$ then $\lambda q + (1-\lambda)q' = (\lambda o + (1-\lambda)o') + (\lambda r + (1-\lambda)r')$.</> <>For the union, $R(q)$ meets $\mathcal{WO}_i \cup \mathcal{WO}_j$ iff it meets $\mathcal{WO}_i$ or meets $\mathcal{WO}_j$ — a statement about sets, with nothing geometric in it. The scene's `intersects` is that "or".</> <>For rotation: $\mathcal{QO} \subset SE(2)$ is a union over θ of convex slices $\mathcal{QO}(\theta) \times \{\theta\}$, and a union of convex sets need not be convex. The Minkowski Studio's stack toggle shows slices whose shapes differ; take two configurations in different slices and find the midpoint.</>

2 FOUNDATION •• **Count the degrees of freedom**

Give $\dim \mathcal{Q}$ for (a) two planar rigid bodies tied together by a taut rope of fixed length; (b) two planar rigid bodies connected rigidly by a bar; (c) a train on tracks, with and without the wheel angles, where the wheels roll without slipping; (d) a sheet of paper (Choset problem 3.2, parts b, c, e, i). For (c), name the constraint that is nonholonomic and explain why it removes no dimension.

HINTS

<>Start from the unconstrained count — 3 per planar body — and subtract one per independent holonomic constraint $g(q) = 0$. A taut rope fixes one distance; a rigid bar fixes a distance *and* two relative angles.</> <>A train's position along a fixed track is one number. Adding k wheel angles adds k coordinates — but rolling without slipping ties each wheel's *rate* to the train's speed: $r\dot{\phi}_i = \dot{s}$. That is $g(q, \dot{q}) = 0$, a nonholonomic constraint. It happens to be integrable here, so it is really holonomic in disguise — $\phi_i = s/r + \text{const}$ — and the configuration space is still one-dimensional. Chapter 20 is about the constraints that do not integrate.</> <>A sheet of paper can bend without stretching: its configuration space is infinite-dimensional. Choset's point is that "robot" is a modeling choice, and the count depends on what you let move.</>

3 CONCEPTUAL •• **Predict the band, then verify**

PREDICT FIRST

In the C-Space Morph, load the micro-example preset (zero-thickness links, 1° raster) and drag the square to be centered at (0.95, 0). Before releasing: what appears on the torus, and how wide is it in θ_1?

- The lens widens but stays a lens: no band
  Link 1 has length 1 and the square now spans x ∈ [0.85, 1.05]: link 1 passes through it whenever $|\theta_1|$ is small enough, for every θ_2. A link-1 collision is a vertical band.
- ✓ A vertical band of half-width arctan(0.1/0.85) = 6.71°, i.e. the rows $\theta_1 = -6°, \ldots, 6°$ fully occupied
  Right. Link 1 is the segment from the origin at heading θ_1; it crosses the square's left edge x = 0.85 at height 0.85 tan θ_1, which is inside the square iff $|\theta_1| \le$ arctan(0.1/0.85) = 6.71°. On the 1° raster that is 13 fully occupied rows, −6° to 6°; the check `exercise 3` pins it.
- A vertical band of half-width arcsin(0.1) = 5.74°
  That is the condition for the link's *tip* (at radius 1) to be inside the square. But the link is a segment, not a point: it enters the square through its left edge before the tip does, so the band is wider.
- A band of half-width arctan(0.1/0.4) = 14.04°, as in the $\theta_1 = 0$ slice
  The 14.04° half-width was the angle the near corner subtends from the *elbow* at (1, 0), when the square was at x = 1.4. Now the square is near the base and link 1 does the colliding; the relevant vertex is the origin and the relevant corner is (0.85, 0.1).

Now set the link thickness to 0.03 m. The band widens to the rows $-8°, \ldots, 8°$ — 17 rows. Explain why the edge is not at arctan(0.1/0.85) any more: the capsule of radius 0.03 touches the corner (0.85, 0.1) when the link's *axis* passes within 0.03 of it, which adds arcsin$(0.03/\sqrt{0.85^2 + 0.1^2})$ = 2.01° to the half-width: 8.72°, so 8° is in and 9° is out. This is Choset's "thickening" footnote in a single number, and the price of a conservative raster: 17 forbidden rows where 13 are truly blocked.

4 **CONCEPTUAL** •• **How flat is the ellipse?**

In the Jacobian Lens set $\theta_2 = 10°$ with $L_1 = L_2 = 1$. The velocity ellipse has semi-axes $\sigma_1 \ge \sigma_2$, the singular values of J. Predict their ratio from two facts you can compute by hand — $\sigma_1\sigma_2 = |\det J| = \sin\theta_2$ and $\sigma_1^2 + \sigma_2^2 = \|J\|_F^2 = L_1^2 + 2L_2^2 + 2L_1L_2\cos\theta_2$ — then read it off the widget. Then switch on J^T and find a configuration where a unit tip force needs the *largest* joint torque; explain why it is the same q at which the ellipse is

longest.

WORK IT OUT

σ_1 / σ_2 at $\theta_2 = 10°$, $L_1 = L_2 = 1$ (any θ_1), to one decimal

Answer. 28.6 With $p = \sigma_1\sigma_2 = \sin 10° = 0.1736$ and $s = \sigma_1^2 + \sigma_2^2 = 3+2\cos 10° = 4.9696$, the two values solve $\sigma^4 - s\sigma^2 + p^2 = 0$: $\sigma_1 = 2.228$, $\sigma_2 = 0.078$, ratio 28.6. The ratio does not depend on θ_1 — rotating the whole arm rotates the ellipse without reshaping it — and the check `jacobian` asserts both facts. For the torque question: $\tau = J^{\mathsf{T}} f$ has the same singular values as J, so the largest torque for a unit force is σ_1, attained when f points along the ellipse's long axis; σ_1 is largest when the arm is straight ($\theta_2 = 0$), where $\|J\|_F^2 = 5$ and $\sigma_1 = \sqrt{5} = 2.236$ — the longest lever.

5 PRACTICAL • • • **A three-dimensional raster**

Implement `rasterize_cobstacle` for `Reach<3>` on T^3 — a `BitGrid3` with `components_torus` wrapping all three axes — and render three θ_3-slices of the Workbench as images. Verify with a test that setting $L_3 = 0$ and taking the $\theta_3 = 0$ slice reproduces the `Reach<2>` grid of this chapter bit for bit. Then measure: how many collision queries does 1° on T^3 cost, and how long does `parry2d` take per query on your machine? That product is the reason Part III exists.

6 PRACTICAL • • • **Choset problem 3.18: translation paths at some headings, none at others**

Write a program that reads a convex robot (counterclockwise vertices, reference point at the origin) and a set of convex obstacles from files, takes a heading θ, computes $QO(\theta)$ for each obstacle with `star_algorithm`, unions the slices, and flood-fills the complement to decide whether a translation path exists between two given points. Show one heading at which the path exists and one at which it does not. Cross-check the union against `rasterize_cobstacle` on $\mathbb{R}^2$ with a `FootprintChecker` at the same heading: every occupied cell center must lie in some slice, and every cell center in a slice must be occupied.

4.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

§3.1–3.3 for configuration, C-obstacles, the pixel method and the dimension count; §3.7 for Table 3.1; §3.8 for the Jacobian and Examples 3.8.1–3.8.2; Appendix F in full — half-planes, Type A/B contacts, the star algorithm, SE(2) slices and GJK as Algorithm 23. link

Lozano-Pé rez, T. (1983). Spatial Planning: A Configuration Space Approach. *IEEE Transactions on Computers C-32(2), 108–120*.

The paper that made configuration space the planner's space and computed polygonal C-obstacles from vertex–edge contacts; the star algorithm and the SE(2) slices descend from it. doi:10.1109/TC.1983.1676196

Gilbert, E. G., Johnson, D. W., and Keerthi, S. S. (1988). A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation 4(2), 193–203*.

GJK — Choset's Algorithm 23 and Derivation 5: distance as the distance from the Minkowski difference to the origin, computed from support functions alone. What parry2d runs inside every distance query. doi:10.1109/56.2083

Yoshikawa, T. (1985). Manipulability of Robotic Mechanisms. *International Journal of Robotics Research 4(2), 3–9*.

The velocity ellipsoid and the manipulability measure $|\det J| = \sigma_1\sigma_2$ the Jacobian Lens draws; singularities as the ellipsoid's collapse. doi:10.1177/027836498500400201

Latombe, J.-C. (1991). Robot Motion Planning. *Kluwer Academic Publishers*.

Chapter 3 is the fullest classical treatment of configuration space obstacles for translating and rotating polygons, including the contact-based boundary construction this chapter follows. doi:10.1007/978-1-4615-4022-9

Lynch, K. M. and Park, F. C. (2017). Modern Robotics: Mechanics, Planning, and Control. *Cambridge University Press*.

Chapters 2 and 5: degrees of freedom by Grü bler's formula and the manipulator Jacobian, manipulability ellipsoids and singularities, by one of Choset's co-authors. link

Montaut, L., Le Lidec, Q., Petrik, V., Sivic, J., and Carpentier, J. (2022). Collision Detection Accelerated: An Optimization Perspective. *Robotics: Science and Systems (RSS)*.

GJK re-read as a Frank–Wolfe method on the Minkowski difference, with Nesterov acceleration — the modern view of Algorithm 23, and why its support-function formulation is the right one. link

Dimforge (2024). parry: 2D and 3D collision-detection library for the Rust programming language. *Documentation and source*.
The engine behind crates/collide: GJK, EPA for penetration depth, a Bvh broad phase and shape-casting — what this chapter's Appendix F explains and the black_box_agrees test verifies.
link

CHAPTER 5

Configuration Space II: Topology, Manifolds, and Rigid Bodies

FOUNDATIONS — ROBOTS, WORLDS, AND CONFIGURATION SPACE · Intermediate · 55 min

> *It is not surprising that people once believed the world was flat — they were only looking at their neighborhoods!*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), §3.4.1*

Chapter 4 drew Reach's configuration space as a square with amber blobs and kept saying "the edges are identified." This chapter says what that means, why a planner must care, and how to make the compiler enforce it.

The reason not to skip Choset's most abstract pages is concrete. Every planner in the rest of this book needs three things from a configuration space: a *distance* between two configurations, an *interpolation* between them, and a way to *sample* them. All three are wrong if you pretend the torus T^2 is the plane $\mathbb{R}^2$, or that SE(2) is $\mathbb{R}^3$ — not wrong by a little, wrong by the width of the whole space, as the first worked example shows with two numbers. The cure is one sentence, and it is a Rust trait: **a configuration space is a manifold — local coordinates, usually no global ones — so a planner must be written against the operations the manifold provides, `dist`, `interpolate`, `sample`, and never against the coordinates.**

Along the way the chapter collects what Choset's §3.4–3.6 and Appendices B, C and E say about mappings, charts, connectedness, compactness, metric spaces, matrix groups and rotations, keeps the examples that carry the pedagogy — the circle, the ellipse and the racetrack; the four charts of S^1; the three frames A, B, C worked by hand — and closes by reading Reach's Jacobian as the differential $D\varphi$,

a linear map that does not care which chart you used.

AFTER THIS CHAPTER YOU CAN

- Say precisely what it means for two configuration spaces to be "the same" — homeomorphic, diffeomorphic — and tell the torus from the plane by a topological invariant
- Define manifolds, charts and atlases, and show with Choset's four charts why the circle has no global coordinate
- Write a metric, a geodesic interpolation and a uniform sampler for S^1, T^2, SE(2) and SO(3), and verify the metric axioms
- Compose rigid-body transforms by subscript cancellation, and tell a body-frame displacement from a world-frame one
- Explain gimbal lock as a lost chart, and why unit quaternions need four charts but never lose one
- Implement all of it behind one Rust trait, Manifold, that every planner from Chapter 6 on is generic over

ASSUMED

- The configuration types of Chapter 2: T2 for Reach, Pose2 for Rusty, HitchCfg
- Chapter 4: the unwrapped torus, C-obstacles, and Reach's Jacobian J(q)
- Open and closed sets, continuity, the chain rule (Choset App. B, C — reviewed as needed)

5.1 The problem: a path in two pieces

Here is the Chapter 4 picture of Reach's configuration space again — the square $[-\pi, \pi]^2$ with the C-obstacles of three disc obstacles on the Workbench — and a purple path between two configurations. On the square the path is two disconnected pieces. Watch the square fold.

INTERACTIVE · w5.1

Torus Unwrapper

T^2 is not a square with periodic boundary conditions. The square is one chart; the torus is the space, and the seam is where the chart lies.

Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch05-configuration-space-ii#w5.1`

When the square has become a doughnut the two pieces are one curve, and a short one. Nothing about the configurations changed; the picture of the space changed, from a *chart* to the *space*. The readout underneath says the same thing in numbers. For the default endpoints the chart's Euclidean distance is about six, the torus distance well under one, and the difference is entirely the seam: the path crosses the dashed edge where $\theta_1 = \pi$ becomes $\theta_1 = -\pi$.

Three things in the widget are the whole chapter.

The seam is an artifact of the chart, not a feature of the space. The square is one way of assigning two numbers to a configuration of a two-joint arm. It is a good way almost everywhere and a bad way along four edges, and a planner that works with the numbers inherits the bad edges. Choset's §3.4.2 names what the square is — a *chart* — and what the torus needs instead — an *atlas*.

Distance belongs to the space. `dist` in the readout is d_{T^2}, the metric this chapter defines on the torus. It is shorter than the chart distance exactly when the short way crosses a seam. Chapter 11's nearest-neighbor queries, Chapter 12's tree extensions and Chapter 13's rewiring radii all call this function, and all of them would quietly plan around a wall that does not exist if they called the chart's distance instead.

Joint limits change the topology. Switch them on. The square stops wrapping, the surface on the right is a disc, and the component count in the readout changes: the band of C-obstacle that cut the square into two components, which the torus had glued back into one, now separates them for good. Choset's §3.4.3 says this in one line — *compact and noncompact spaces cannot be diffeomorphic* — and the lab at the end of the chapter measures it.

> **THE ONE SENTENCE VERSION**
>
> A configuration space is a manifold: locally like $\mathbb{R}^k$, usually with no global coordinates. A planner must ask the space for distances, interpolations and samples, and must never do arithmetic on the coordinates — because every chart has a seam, and only the space knows where it is.

5.2 Building intuition

5.2.1 An angle is not a number

Start with the simplest manifold that is not a Euclidean space: the circle S^1, the configuration space of one revolute joint. Choset's §3.4.2 covers it with four charts — the open half-circles $U_1 = \{x_2 > 0\}$, $U_2 = \{x_2 < 0\}$, $U_3 = \{x_1 > 0\}$, $U_4 = \{x_1 < 0\}$, each coordinatized by whichever Cartesian coordinate varies monotonically on it. The widget hands a point from chart to chart.

INTERACTIVE · w5.3

Atlas of the Circle

An angle is not a number. An angle is a point of S^1; a number is its coordinate in one chart, and no single chart covers the circle.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch05-configuration-space-ii#w5.3

Every point of the circle lies in at least one chart — the four domains *cover*. Wherever a point lies in two, both coordinates exist and are related by a smooth function, the *transition map*; on $U_1 \cap U_3$ it is $\varphi_1 \circ \varphi_3^{-1}(z) = \sqrt{1 - z^2}$, drawn at lower right. That smoothness is what lets a planner, or a derivative, pass from one chart to the next without a jolt. The two-chart angle atlas is the one `S1` uses in code: the angle measured from $+x$ on the circle minus $(-1, 0)$, and the angle measured from $-x$ on the circle minus $(1, 0)$, whose transition map is a translation by π.

Then try the single-chart attempt, $\theta \in [0, 2\pi)$. It assigns a number to every point, and it is not a chart: drag through $\theta = 0$ and the coordinate tears from 6.28 to 0. A chart's domain has to be *open*, so that every point has a neighborhood on which the map is continuous, and no neighborhood of $(1, 0)$ maps continuously to an interval. Chapter 4's "single angle in $[0, 2\pi)$" was a chart with a seam, not an atlas, and the seam is where its arithmetic lies.

5.2.2 A transform is not a position plus an angle

The second intuition is about rigid bodies. Choset's §3.5.1 works one example completely — three frames A, B, C on a unit grid and a point w — and it is the one place in his book where the group SE(2) can be checked by hand. The widget keeps his numbers as its reset state.

INTERACTIVE · w5.4

Frame Composer

A transform is not a position plus an angle. It is a group element, and the order of multiplication is the whole content: T_AB T_1 and T_1 T_AB place B differently.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch05-configuration-space-ii#w5.4

Two rules generate everything in the readout. *Subscript cancellation*: $T_{AB}\, T_{BC} = T_{AC}$, and $T_{BC}\, w_C = w_B$ — the inner subscripts annihilate, and the result is the outer pair. And *non-commutativity*: the same transform T_1 multiplied on the right of T_{AB} moves frame B in its own coordinates (a *body-frame* transformation, B'), multiplied on the left it moves B in A's coordinates (a *world-frame* transformation, B''), and the two frames land in different places. Autoplay sweeps T_1's heading so you can watch B' and B'' diverge; reset returns to Choset's T_1 and the two

placements $(-2, -1, 0)$ and $(2, 1, 0)$. A pose is an element of a group. The order of multiplication is the whole content.

The third intuition — that gimbal lock is a missing chart — waits for the rotation section, where it has its own widget.

5.3 The mathematics

$\phi : S \to T$, $\phi(S)$, $\phi^{-1}(T)$	a mapping; its image; the preimage of a set
(U, ϕ); $\psi \circ \phi^{-1}$	a chart — an open U ⊂ Q with a diffeomorphism ϕ onto an open set of $\mathbb{R}^k$ — and a transition map between two charts
$B_\epsilon(p)$, nbhd(p); C^0, C^k, C^∞	open ball; neighborhood; continuous, k times continuously differentiable, smooth
$T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix}$, T_{AB}, w_A	a homogeneous transform; frame B expressed in A; the point w in A's coordinates
d_{S^1}, d_{T^2}, $d^w_{SE(2)}$, $d_{SO(3)}$	the chapter's metrics; w is the rotation weight in metres per radian w has no canonical value
(ϕ, θ, ψ)	Z-Y-Z Euler angles (Choset §3.6)
$T_q\mathcal{Q}$, $D\varphi_q : T_q\mathcal{Q} \to T_{\varphi(q)}M$	tangent space (informal here, formal in Chapter 20); the differential of the forward map

5.3.1 Mappings: onto, one-to-one, and the two kinds of "the same"

Let $\phi : S \to T$ be a mapping. It is **surjective** (onto) if $\phi(S) = T$, **injective** (one-to-one) if $\phi(s_1) = \phi(s_2)$ forces $s_1 = s_2$, and **bijective** if both. A bijection has an inverse everywhere on T, and this is what lets us move back and forth between a configuration space, whose geometry may be complicated, and a Euclidean space, whose geometry is not. Choset's example: $\sin : (-\pi/2, \pi/2) \to (-1, 1)$ is bijective; $\sin : \mathbb{R} \to [-1, 1]$ is only surjective.

> **DEFINITION 3.4.1 AND 3.4.2 — HOMEOMORPHISM, DIFFEOMORPHISM**
>
> A bijection $\phi : S \to T$ with ϕ and ϕ^{-1} both continuous is a **homeomorphism**, and S, T are **homeomorphic**. A bijection with ϕ and ϕ^{-1} both smooth (C^∞) is a **diffeomorphism**, and S, T are **diffeomorphic**. Every diffeomorphism is a homeomorphism; the converse fails.

The converse fails on Choset's three curves (his figure 3.12): the unit circle $M_c = \{x^2 + y^2 = 1\}$, the ellipse $M_e = \{x^2/4 + y^2 = 1\}$, and the *racetrack* M_r — two half-circles of radius one joined by straight segments, $f_r(x, y) = 0$ with

$$f_r(x,y) = \begin{cases} x-1 & -1 \le y \le 1,\ x > 0 \\ (y+1)^2 + x^2 - 1 & y < -1 \\ (y-1)^2 + x^2 - 1 & y > 1 \\ x+1 & -1 \le y \le 1,\ x < 0. \end{cases}$$

All three are homeomorphic: radial projection $\phi(x,y) = (x,y)/\sqrt{x^2+y^2}$ carries the ellipse onto the circle and is continuous both ways. The circle and ellipse are diffeomorphic — that same ϕ is smooth both ways. But no diffeomorphism reaches the racetrack: its curvature jumps at $(\pm 1, \pm 1)$, where straight meets round, and a diffeomorphism would have to carry the circle's constant curvature onto it smoothly. Topology sees one closed curve; differential structure sees two kinds.

5.3.2 Neighborhoods, and what "locally" means

A **neighborhood** of a point p in a metric space is any set $U \ni p$ such that every point of U has an open ball $B_\epsilon(p') = \{p'' : d(p', p'') < \epsilon\}$ inside U; an open ball is itself a neighborhood. S is **locally diffeomorphic** to T if every point of S has a neighborhood diffeomorphic to an open subset of T. The sphere is locally diffeomorphic to the plane — which is Choset's remark about the flat earth in this chapter's epigraph — and not globally: there is no bijection from the sphere onto the plane that is continuous both ways.

The two robots of Chapter 4 sit on either side of this distinction. A disc translating in the plane has configuration space $\mathbb{R}^2$, globally diffeomorphic to its workspace by the identity. The two-joint arm has T^2, which is *locally* diffeomorphic to $\mathbb{R}^2$ — every configuration has a neighborhood that looks like a patch of the plane — and not globally, because T^2 is compact and $\mathbb{R}^2$ is not (§3.4.3 below). Give the arm strict joint limits $\theta_i^\ell < \theta_i < \theta_i^u$ and its configuration space becomes an open rectangle, which *is* globally diffeomorphic to $\mathbb{R}^2$: stretch each open interval over the line with tan. Joint limits are not a detail. They change the topology.

5.3.3 Manifolds, charts, atlases

DEFINITIONS 3.4.3–3.4.5 — MANIFOLD, CHART, C∞-RELATED CHARTS, ATLAS

A set S is a **k-dimensional manifold** if it is locally homeomorphic to $\mathbb{R}^k$: every point has a neighborhood homeomorphic to an open set of $\mathbb{R}^k$. A **chart** is a pair (U, ϕ) with U open in S and ϕ a diffeomorphism from U onto an open set of $\mathbb{R}^k$; ϕ^{-1} is a *parameterization*. Two charts (U, ϕ), (V, ψ) are **C^∞-related** if the transition maps $\psi \circ \phi^{-1}$ and $\phi \circ \psi^{-1}$ are smooth on the images of $U \cap V$. A family of pairwise C^∞-related charts whose domains cover S is an **atlas**, also called the *differentiable structure*; S with an atlas is a

differentiable manifold.

The configuration spaces this book plans in are all differentiable manifolds, and the parameterizations of Chapter 4 — "the configuration of the arm is (θ_1, θ_2)" — were charts without the name. When no single chart covers the space, as for the torus, there are three options (Choset §3.5): use one chart anyway and suffer its seam; use an atlas; or *embed* the space in a higher-dimensional Euclidean space with constraints — the unit circle in $\mathbb{R}^2$, the doughnut in $\mathbb{R}^3$, rotation matrices in $\mathbb{R}^9$. The code in this chapter uses all three, each where it is best.

DERIVATION **S^1 needs at least two charts, and Choset's four suffice**

$\{(U_i, \phi_i)\}_{i=1}^4$ is an atlas

Statement. No single chart covers S^1. The charts $U_1 = \{x_2 > 0\}, \phi_1 = x_1$; $U_2 = \{x_2 < 0\}, \phi_2 = x_1$; $U_3 = \{x_1 > 0\}, \phi_3 = x_2$; $U_4 = \{x_1 < 0\}, \phi_4 = x_2$, with parameterizations $\phi_1^{-1}(z) = (z, \sqrt{1-z^2})$, $\phi_2^{-1}(z) = (z, -\sqrt{1-z^2})$, $\phi_3^{-1}(z) = (\sqrt{1-z^2}, z)$, $\phi_4^{-1}(z) = (-\sqrt{1-z^2}, z)$, form an atlas.

Step 1 — one chart is impossible. A single chart would be a homeomorphism from S^1 onto an open subset of $\mathbb{R}$, which is a union of open intervals; being the image of a connected set it is one interval. Removing a point from an open interval disconnects it; removing a point from S^1 does not. A homeomorphism preserves connectedness of the complement of a point, so none exists.

Step 2 — cover. Every point of S^1 has $x_1 \neq 0$ or $x_2 \neq 0$, so lies in some U_i. Each U_i is open, each ϕ_i is the restriction of a coordinate projection, a diffeomorphism onto $(-1, 1)$ with the inverse shown.

Step 3 — the transition maps. $U_1 \cap U_2 = U_3 \cap U_4 = \emptyset$, so there are four overlaps to check, each a quarter-circle. On $U_1 \cap U_3$, $\phi_1 \circ \phi_3^{-1}(z) = \sqrt{1-z^2}$ for $z \in (0, 1)$ and $\phi_3 \circ \phi_1^{-1}(z) = \sqrt{1-z^2}$ for $z \in (0, 1)$. The other three overlaps give $\pm\sqrt{1-z^2}$ on $(-1, 0)$ or $(0, 1)$.

Step 4 — smoothness. $\sqrt{1-z^2}$ is smooth on any open subinterval of $(-1, 1)$ — the only non-smooth points, $z = \pm 1$, are excluded because the overlaps are open quarter-circles. Hence the charts are pairwise C^∞-related and the four form an atlas. ■

Two charts suffice (Choset problem 3.9): $V_1 = S^1 \setminus \{(-1, 0)\}$ with the angle from $+x$ in $(-\pi, \pi)$, and $V_2 = S^1 \setminus \{(1, 0)\}$ with the angle from $-x$. Their transition map is $z \mapsto z \pm \pi$ on each component of the overlap: a translation, with derivative 1. This is the atlas `S1` uses, and the derivative-one fact is what makes the Jacobian chart-independent in the last derivation of this

chapter.

5.3.4 Connectedness and compactness

A manifold is **connected** if any two of its points are joined by a path. $\mathbb{R}^n$, S^n, T^n are; so are SO(3) and SE(2). Obstacles break $\mathcal{Q}_{\text{free}}$ into **connected components**, its maximal connected subsets, and the first fact every planner must respect is that there is no solution when q_{start} and q_{goal} lie in different components. The Torus Unwrapper's component count is this definition evaluated on a raster.

A space is **compact** if it is a closed and bounded subset of some $\mathbb{R}^n$. $[0, 1]$ is compact and $[0, 1)$ is not; S^n, T^n and SO(3) are compact; $\mathbb{R}^n$ and SE(2) are not. Products of compact spaces are compact, and a non-compact product $M_1 \times M_2$ with M_1 compact has M_1 as its *compact factor* — $\text{SE}(2) \cong \mathbb{R}^2 \times S^1$ has S^1. Two facts matter for planning: compact and non-compact spaces are never diffeomorphic, which is the invariant that separates T^2 from $\mathbb{R}^2$; and a sampler can be uniform on a compact space but needs a box on a non-compact one, which is why `Se2::sample` takes bounds and `S1::sample` does not.

DERIVATION **Joint limits change the topology**

$T^2 \not\cong \mathbb{R}^2, \quad (\theta^\ell, \theta^u)^2 \cong \mathbb{R}^2$

Statement. The two-joint arm's configuration space T^2 is not diffeomorphic to $\mathbb{R}^2$; with strict joint limits it becomes an open rectangle, which is.
Step 1. Continuous images of compact sets are compact. T^2 is compact (a closed, bounded subset of $\mathbb{R}^3$ via the doughnut embedding) and $\mathbb{R}^2$ is unbounded, so no continuous bijection carries one onto the other — let alone a diffeomorphism.
Step 2. With limits $\theta_i^\ell < \theta_i < \theta_i^u$, each joint ranges over an open interval and the space is the open rectangle $(\theta_1^\ell, \theta_1^u) \times (\theta_2^\ell, \theta_2^u)$.
Step 3. $t \mapsto \tan\!\left(\pi \frac{t-\theta^\ell}{\theta^u-\theta^\ell} - \frac{\pi}{2}\right)$ is a diffeomorphism from (θ^ℓ, θ^u) onto $\mathbb{R}$, and a product of diffeomorphisms is a diffeomorphism, so the rectangle is diffeomorphic to $\mathbb{R}^2$. ■
Closed limits $\theta_i^\ell \le \theta_i \le \theta_i^u$ give a compact space again — but a manifold *with boundary*, which Choset's §3.4.4 warns is not a manifold in the sense above. And some one-degree-of-freedom parallel mechanisms have a configuration space shaped like a figure eight, which is no manifold at all: at the crossing there are two distinct motion directions. "If you cannot show it to be a manifold, it may not be."

5.3.5 Metric spaces, and the contract for `dist`

DEFINITION C.2.1 — METRIC

A **metric** on a set M is a function $d : M \times M \to \mathbb{R}_{\geq 0}$ with, for all m_1, m_2, m_3: (1) *definiteness*, $d(m_1, m_2) = 0$ iff $m_1 = m_2$; (2) *symmetry*, $d(m_1, m_2) = d(m_2, m_1)$; (3) the *triangle inequality*, $d(m_1, m_3) \leq d(m_1, m_2) + d(m_2, m_3)$. The open balls of a metric generate a topology, and two metrics are equivalent when each ball of one contains a ball of the other.

Non-negativity is not an axiom; it follows: $0 = d(m_1, m_1) \leq d(m_1, m_2) + d(m_2, m_1) = 2d(m_1, m_2)$. A function $f : S \to T$ between metric spaces is **continuous** (Definition C.4.1) when preimages of open sets are open, equivalently when for every $\epsilon > 0$ there is a $\delta > 0$ with $d(x, s) < \delta \Rightarrow d(f(x), f(s)) < \epsilon$. A path is a C^0 map $c : [0, 1] \to \mathcal{Q}_{\text{free}}$; a trajectory must be C^k for $k \geq 1$ so that velocity exists.

Here are the four metrics this chapter ships, with the chart's wrong answer written beside each right one. The seam example from the hook: angles $a = 0.1$ and $b = 6.2$ radians.

$$d_{S^1}(a, b) = \min\left(|a - b|,\ 2\pi - |a - b|\right) \qquad d_{S^1}(0.1, 6.2) = 2\pi - 6.1 = 0.1832 \quad \text{(chart: } |a - b| = 6.1\text{)}$$

The geodesic midpoint `interpolate(a, b, 0.5)` goes the short way, *downward* from 0.1 by half of 0.1832: it lands at 0.0084 after wrapping, not at the chart midpoint 3.15. On the torus, with $p = (0.1, 3.0)$ and $q = (6.2, 3.5)$:

$$d_{T^2}(p, q) = \sqrt{d_{S^1}(0.1, 6.2)^2 + d_{S^1}(3.0, 3.5)^2} = \sqrt{0.1832^2 + 0.5^2} = \mathbf{0.5325}.$$

On SE(2) the metric needs an *exchange rate* w between metres and radians, and with $w = 0.5$ m/rad the distance from $(0, 0, 0)$ to $(1, 0, \pi)$ is

$$d^w_{SE(2)} = \sqrt{\Delta x^2 + \Delta y^2 + w^2\, d_{S^1}(\theta, \theta')^2} = \sqrt{1 + 0.25\pi^2} = \mathbf{1.8621}.$$

On SO(3) the distance between two rotations is the angle of the rotation that carries one to the other, $d_{SO(3)}(R, R') = \arccos\left((\operatorname{tr}(R^{\mathsf{T}}R') - 1)/2\right)$, which is also the axis–angle θ of $R^{\mathsf{T}}R'$ (App. E.3) and, for unit quaternions, $2\arccos|\langle Q, Q'\rangle|$.

DERIVATION **The chapter's metrics are metrics, and interpolate follows geodesics**

$d_{S^1}, d_{T^2}, d^w_{SE(2)}, d_{SO(3)}$ satisfy C.2.1

Statement. Each of the four functions above satisfies Definition C.2.1, and `interpolate(t)` moves along the shortest arc at constant speed: $d(a, \gamma(t)) =$

$t\,d(a,b)$.

Step 1 — S^1 is a quotient. Identify S^1 with $\mathbb{R}/2\pi\mathbb{Z}$. For classes $[a],[b]$ define $d([a],[b]) = \min_k |a-b+2\pi k|$, the shortest representative of the difference — which is exactly $\min(|a-b|, 2\pi-|a-b|)$ for representatives in a window of width 2π. Definiteness and symmetry are inherited from $|\cdot|$. For the triangle inequality pick k_1, k_2 attaining the two minima on the right; then $a-c+2\pi(k_1+k_2)$ is *some* representative of $[a]-[c]$, so $d([a],[c]) \le |a-b+2\pi k_1| + |b-c+2\pi k_2|$.

Step 2 — products. For metrics d_1, d_2, the function $\sqrt{d_1^2+d_2^2}$ is a metric: definiteness and symmetry are immediate, and the triangle inequality is Minkowski's inequality for the Euclidean norm applied to the vectors $(d_1(a,b), d_2(a,b))$ and $(d_1(b,c), d_2(b,c))$, using the triangle inequalities of d_1 and d_2 componentwise. This gives d_{T^2}, d_{T^n}, and $d^w_{SE(2)}$ for any $w>0$ (a positive scalar multiple of a metric is a metric).

Step 3 — the weight has no canonical value. Scaling w changes which pose is "nearer": with $w=0.5$ a half-turn in place costs 1.57 m-equivalents, with $w=2$ it costs 6.28. Both are metrics; neither is right. Chapter 11 measures what a PRM does as w varies, and this book never hides w in a default.

Step 4 — SO(3). $d(R,R')$ is the angle $\theta\in[0,\pi]$ of the relative rotation $R^{\mathsf T}R'$. Definiteness: $\theta=0$ iff $R^{\mathsf T}R'=I$. Symmetry: $R'^{\mathsf T}R$ is the inverse rotation, same angle. Triangle inequality: composing a rotation by α with a rotation by β yields a rotation by at most $\alpha+\beta$ (the angle is the geodesic distance on the unit 3-sphere of quaternions, folded by the double cover, and great-circle distance on a sphere satisfies the triangle inequality). The quaternion form $2\arccos|\langle Q,Q'\rangle|$ is the same number because $\langle Q,Q'\rangle = \cos(\theta/2)$ up to sign, and the absolute value is what identifies Q with $-Q$.

Step 5 — geodesics. On S^1, $\gamma(t) = a + t\,\mathrm{wrap}(b-a)$ moves along the shorter arc with $d(a,\gamma(t)) = t|\mathrm{wrap}(b-a)| = t\,d(a,b)$. On a product, the componentwise geodesic is the geodesic of the ℓ^2 product metric because the factors do not interact. On SO(3), spherical linear interpolation between Q and the sign-flipped $\pm Q'$ nearer to Q traverses the shorter great-circle arc at constant speed. ■

The antipodal ambiguity. At $d_{S^1}=\pi$ exactly, or $\theta=\pi$ on SO(3), the geodesic is not unique. `S1::interpolate` takes the representative $+\pi$ that `wrap` returns — counter-clockwise — and `So3::interpolate` the sign-flipped endpoint; both deterministic, both documented, neither hidden.

5.3.6 Embeddings: $SO(n)$ and $SE(n)$

The alternative to an atlas is an embedding with constraints. A rotation in $\mathbb{R}^3$ is a 3×3 matrix whose columns are the body axes $\tilde{x},\tilde{y},\tilde{z}$ expressed in a stationary

frame: nine numbers, six constraints (three unit lengths, three orthogonalities), three degrees of freedom.

§3.5.1 — THE MATRIX GROUPS

$$SO(n) = \{R \in \mathbb{R}^{n\times n} : RR^{\mathsf{T}} = I,\ \det R = 1\}, \qquad SE(n) = \left\{\begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} : R \in SO(n),\ p \in \mathbb{R}^n\right\}.$$

Both are groups under matrix multiplication and smooth submanifolds of $\mathbb{R}^{m^2}$ — Lie groups, in Choset's footnote: the group operation and inversion are C^∞. $\dim SO(2) = 1$, $\dim SO(3) = 3$, $\dim SE(2) = 3$, $\dim SE(3) = 6$. $R^{\mathsf{T}} = R^{-1}$.

A matrix in $SE(n)$ is used three ways (§3.5.1): to *represent* a configuration — then it is a *frame*; to *change the reference frame* of a configuration or a point; and to *displace* one — then it is a *transform*. The three frames of the Frame Composer are Choset's figure 3.17 restricted to the plane, with T_{AB} having $R = \mathrm{rot}(\pi)$, $p = (-2, 0)$ and T_{BC} having $R = \mathrm{rot}(-\pi/2)$, $p = (-4, -1)$.

DERIVATION **SE(n) composes by subscript cancellation; body versus world transformations**

$T_{AB}T_{BC} = T_{AC}$, $\quad T_{AB}T_1 \neq T_1T_{AB}$

Statement. $T_{AB}T_{BC} = T_{AC}$ changes the frame in which C is expressed; $w_A = T_{AB}w_B$ changes a point's frame; $T_{AB}w_A$ *displaces* the point by the motion that carries A to B; and for a transform T_1, $T_{AB}T_1$ acts in B's frame (body) while T_1T_{AB} acts in A's (world), with different results.

Step 1 — block multiply.

$$\begin{bmatrix} R_1 & p_1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} R_2 & p_2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_1R_2 & R_1p_2 + p_1 \\ 0 & 1 \end{bmatrix}.$$

$R_1R_2 \in SO(n)$ since $(R_1R_2)(R_1R_2)^{\mathsf{T}} = I$ and determinants multiply, so $SE(n)$ is closed under the product. The inverse is $T^{-1} = \begin{bmatrix} R^{\mathsf{T}} & -R^{\mathsf{T}}p \\ 0 & 1 \end{bmatrix}$, as one checks by multiplying. With the identity, $SE(n)$ is a group.

Step 2 — the constraint count. $R \in SO(3)$ has 9 entries and 6 independent constraints, so $\dim SO(3) = 3$; adding p gives $\dim SE(3) = 6$, the six degrees of freedom of a rigid body in space. In the plane, $\dim SO(2) = 1$

and $\dim SE(2) = 3$.

Step 3 — the micro-example. With $T_{AB} = (x, y, \theta) = (-2, 0, \pi)$ and $T_{BC} = (-4, -1, -\pi/2)$: $R_{AC} = \mathrm{rot}(\pi)\,\mathrm{rot}(-\pi/2) = \mathrm{rot}(\pi/2)$ and $p_{AC} = R_{AB}\,p_{BC} + p_{AB} = (4, 1) + (-2, 0) = (2, 1)$, so $T_{AC} = (2, 1, \pi/2)$, Choset's $\begin{bmatrix} 0 & -1 & 2 \\ 1 & 0 & 1 \end{bmatrix}$. The point $w_C = (-2, 1)$: $w_B = R_{BC}w_C + p_{BC} = (1, 2) + (-4, -1) = (-3, 1)$ and $w_A = R_{AB}w_B + p_{AB} = (3, -1) + (-2, 0) = (1, -1)$, both as in his text. Displacing instead: $T_{AB}w_A = (-3, 1)$ — the subscripts do not cancel, and the result is a *new* point in A's frame, rotated about A's origin by R_{AB} and translated by p_{AB} (figure 3.18).

Step 4 — the two orderings. With $T_1 = (0, 1, \pi)$: $T_{AB}T_1 = \begin{bmatrix} R_{AB}R_1 & R_{AB}p_1 + p_{AB} \\ 0 & 1 \end{bmatrix} = (-2, -1, 0)$ rotates B about *its own* origin by R_1 then translates by p_1 *in the original B frame*; $T_1T_{AB} = \begin{bmatrix} R_1R_{AB} & R_1p_{AB} + p_1 \\ 0 & 1 \end{bmatrix} = (2, 1, 0)$ rotates B about *A's* origin and translates in A. They differ because matrix multiplication does not commute. n body-frame transformations stack on the right, $T_{AB}T_1T_2 \cdots T_n$; world-frame ones on the left, $T_n \cdots T_2T_1T_{AB}$. ■

Choset problem 3.23. SE(2) and $\mathbb{R}^2 \times SO(2)$ with the product operation $(x_1, R_1)(x_2, R_2) = (x_1 + x_2, R_1R_2)$ are homeomorphic as spaces — the same three coordinates — but not isomorphic as groups: the product group is commutative and SE(2) is not. `Pose2::compose` is SE(2)'s product, with the R_1p_2 coupling; the commutative one is what you get by adding poses componentwise, which is wrong for a robot.

5.3.7 $D\varphi$ is chart-independent; the Jacobian is its matrix

Chapter 4 computed Reach's Jacobian $J(q) = \partial\varphi/\partial q$ for the forward map $\varphi : T^2 \to \mathbb{R}^2$. Appendix C.5 calls the same matrix the **differential** $D\varphi$ and adds the one footnote that matters here: the differential is really a linear map from the tangent space of the domain to the tangent space of the range, $D\varphi_q : T_q\mathcal{Q} \to T_{\varphi(q)}M$, and the matrix is its expression in charts. Along a curve $c(t)$ in $\mathcal{Q}$, $\frac{d}{dt}\varphi(c(t)) = D\varphi_{c(t)}\,\dot{c}(t)$ — velocities are columns, pushed forward by J. Forces live in the cotangent space and are rows, pulled back by J^T (Choset's remark on p. 486), which is why Chapter 7 will write $\tau = J^\mathsf{T}F$.

DERIVATION **Dϕ is chart-independent; the Jacobian is its matrix**

```
D(\varphi \circ h) = D\varphi \cdot Dh,\quad Dh = I \text{ for the angle atlas}
```

Statement. Under a change of chart h, the matrix of $D\varphi$ transforms by the chain rule $D(\varphi \circ h) = D\varphi \cdot Dh$. For the torus's angle atlas the transition maps are translations by 2π, $Dh = I$, and Chapter 4's J is the same matrix in every chart.

Step 1 — the differential along curves. Let $c : \mathbb{R} \to Q$ be smooth and set $x(t) = \varphi(c(t))$. Then $\dot{x} = \frac{\partial \varphi}{\partial q}\big|_{c(t)} \dot{c} = D\varphi_{c(t)}\, \dot{c}(t)$: $D\varphi$ is the linear map that takes the velocity of the configuration to the velocity of the image.

Step 2 — the chain rule (C.5). If the curve is given in y-coordinates and h maps y-coordinates to x-coordinates, $\frac{d}{dt}(\varphi \circ h \circ c) = D_h\varphi \cdot D_c h \cdot \dot{c}$. Choset's Example C.5.1 is the worked check: $c(t) = (2, 2\pi t)$ in polar coordinates, h polar-to-Cartesian, $g(x) = x_1$, and the chain rule gives $[1\,0] \begin{bmatrix} \cos 2\pi t & -2\sin 2\pi t \\ \sin 2\pi t & 2\cos 2\pi t \end{bmatrix} \begin{bmatrix} 0 \\ 2\pi \end{bmatrix} = -4\pi \sin 2\pi t$, which direct differentiation of $2\cos 2\pi t$ confirms.

Step 3 — the torus. The transition maps of the angle atlas are $h(\theta_1, \theta_2) = (\theta_1 \pm 2\pi, \theta_2)$ and the like, with $Dh = I$. So the matrix of $D\varphi_q$ is the same $J(q)$ whichever chart the configuration is read in; shifting θ_1 by a full turn changes nothing — not the entries, not the singularities. At Choset's $q = (\pi/4, \pi/2)$ with $L_1 = L_2 = 1$ and $\dot{q} = (1, 0)$:

$$J(q)\dot{q} = \begin{bmatrix} -\sqrt{2} & -\sqrt{2}/2 \\ 0 & -\sqrt{2}/2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -\sqrt{2} \\ 0 \end{bmatrix} = \begin{bmatrix} -1.4142 \\ 0 \end{bmatrix},$$

and `dphi_chart_independent` asserts the matrix is unchanged, to machine precision, under $\theta_1 \mapsto \theta_1 + 2\pi$. ■

5.3.8 Rotations in three dimensions, briefly

Everything above generalizes from $SO(2)$ to SO(3) except the coordinates. Nine matrix entries with six constraints leave three degrees of freedom, so SO(3) is a three-dimensional manifold and can be *locally* parameterized by three numbers — Euler angles — but, just as for the circle, not globally.

Choset's choice is **Z-Y-Z Euler angles** (ϕ, θ, ψ) (§3.6): rotate about z by ϕ, then about the new y by θ, then about the new z by ψ, so that

$$R = R_{z,\phi} R_{y,\theta} R_{z,\psi} = \begin{bmatrix} c_\phi c_\theta c_\psi - s_\phi s_\psi & -c_\phi c_\theta s_\psi - s_\phi c_\psi & c_\phi s_\theta \\ s_\phi c_\theta c_\psi + c_\phi s_\psi & -s_\phi c_\theta s_\psi + c_\phi c_\psi & s_\phi s_\theta \\ -s_\theta c_\psi & s_\theta s_\psi & c_\theta \end{bmatrix} \qquad (3.6\text{–}3.8).$$

INTERACTIVE · w5.2

Rotation Zoo

Gimbal lock is not mechanical. It is a missing chart: Euler angles are coordinates on SO(3) minus a set, and the body simply drove off the map.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch05-configuration-space-ii#w5.2

DERIVATION **Euler angles lose a chart; quaternions provide an atlas**

$U_{\text{Euler}} = \{R : R_{33} \neq \pm 1\}$; $\{U_i\}_{i=0}^{3}$ cover $SO(3)$

Statement. The Z-Y-Z map is a chart only on $\{R : R_{33} \neq \pm 1\}$. At $R_{33} = 1$ only $\phi+\psi$ is determined (Choset eq. E.2, E.9). Unit quaternions $Q = (\cos\frac{\theta}{2}, n\sin\frac{\theta}{2})$ cover SO(3) with four charts $U_i = \{|q_i| \text{ largest}\}$.

Step 1 — the collapse. Put $\theta = 0$ in (3.8). Then $c_\theta = 1$, $s_\theta = 0$ and

$$R = \begin{bmatrix} c_{\phi+\psi} & -s_{\phi+\psi} & 0 \\ s_{\phi+\psi} & c_{\phi+\psi} & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{(E.2)},$$

a rotation about z by $\phi + \psi$: infinitely many (ϕ, ψ) pairs give the same R, and only $\phi + \psi = \operatorname{atan2}(R_{21}, R_{11})$ is defined (E.9). The same happens at $R_{33} = -1$, where $\theta = \pi$ and only $\phi - \psi$ survives (E.10–E.11). Off those two rotations, $\theta = \operatorname{atan2}(\sqrt{1 - R_{33}^2}, R_{33})$, $\phi = \operatorname{atan2}(R_{23}, R_{13})$, $\psi = \operatorname{atan2}(R_{32}, -R_{31})$ (E.3, E.5, E.6) invert the map smoothly, so it is a chart there. *Gimbal lock* is the name for having driven off this chart; nothing mechanical is involved.

Step 2 — unit quaternions. For a rotation by θ about a unit axis n, $Q = (\cos\frac{\theta}{2}, n_x\sin\frac{\theta}{2}, n_y\sin\frac{\theta}{2}, n_z\sin\frac{\theta}{2})$ (E.26) has $\|Q\|^2 = \cos^2\frac{\theta}{2} + \sin^2\frac{\theta}{2}(n_x^2 + n_y^2 + n_z^2) = 1$ (E.27), and the matrix $R(Q)$ of (E.28) is a rotation; Q and $-Q$ give the same R — the double cover — and quaternion multiplication is rotation composition.

Step 3 — four charts cover. Some component of a unit quaternion has the largest magnitude, at least $\frac{1}{2}$. On $U_i = \{|q_i| \geq |q_j| \ \forall j\}$ (interiors, to make them open) the map $\varphi_i(Q) = (q_j/|q_i|)_{j\neq i}$ is a smooth bijection onto an open subset of $\mathbb{R}^3$, and the U_i cover. Where an Euler chart ends, the quaternion simply moves to a different U_i — which is what the Rotation Zoo's smooth quaternion curve shows while ϕ and ψ swing. ■

Kept for Appendix B: the full inverse (E.3–E.11), roll–pitch–yaw (E.12) and its lock at pitch $\pm\pi/2$, the axis–angle matrix (E.18) and the chart $\varphi(R) = \theta k$ on

{tr $R \neq \pm 1$} (E.20–E.21), the quaternion product and conjugate (E.31–E.35). The exponential and logarithm maps that turn axis–angle into a Lie-group coordinate, and the ⊞/⊟ operators estimators need, are the sister book's Chapter 3, whose `So3` the Rust `exp/log` of Exercise 6 reuses.

One more honesty item about SO(3), because Chapter 11's samplers depend on it. *Uniform on* SO(3) means uniform with respect to the Haar measure, the one invariant under rotation. A normalized Gaussian 4-vector is uniform on the quaternion 3-sphere and therefore Haar-uniform on SO(3). Uniform *Euler angles* are not: the rotation angle of a Haar-uniform rotation has density $(1-\cos\theta)/\pi$ on $[0,\pi]$, mean $\pi/2 + 2/\pi = 2.2074$, and only $(0.5 - \sin 0.5)/\pi = 0.65\%$ of rotations lie within half a radian of the identity; drawing (ϕ, θ, ψ) uniformly puts about three times that mass near the identity. `So3::sample` is the Gaussian version, and a check pins both numbers.

5.4 The algorithm

This chapter's algorithms are operations, not planners: the three methods a manifold must provide, and the first generic planner-shaped function built on them.

ALGORITHM `GEODESIC-PATH(M, a, b, n)` COST `O(n · dim M)`

in a manifold M with dist and interpolate, endpoints a, b ∈ M, a step count n
out n + 1 configurations along the shortest geodesic from a to b

1: for $i = 0, 1, \ldots, n$ do
2: $\quad t \leftarrow i/n$
3: $\quad q_i \leftarrow M.\text{interpolate}(a, b, t)$ — the short way, by the metric's own geodesic
4: return $(q_0, q_1, \ldots, q_n)$ — with $q_0 = a$, $q_n = b$, and $M.\text{dist}(a, q_i) = t_i\, M.\text{dist}(a, b)$

The per-manifold operations, in the order the Rust implements them:

ALGORITHM `S1 · dist, interpolate, sample` COST `O(1)`

in angles a, b ∈ $(-\pi, \pi]$; a parameter t ∈ [0, 1]; a seeded Rng
out a distance in $[0, \pi]$; an angle on the short arc; a uniform angle

1: dist$(a, b) \leftarrow |\text{wrap}(a - b)|$ — $= \min(|a - b|, 2\pi - |a - b|)$
2: interpolate$(a, b, t) \leftarrow \text{wrap}\left(a + t \cdot \text{wrap}(b - a)\right)$ — antipodal tie: wrap returns $+\pi$, counter-clockwise
3: sample() $\leftarrow$ Uniform$(-\pi, \pi)$
4: T^n: apply 1–3 componentwise; dist $\leftarrow \sqrt{\sum_i d_{S^1}(a_i, b_i)^2}$
5: SE(2) with weight w: dist $\leftarrow \sqrt{\Delta x^2 + \Delta y^2 + w^2 d_{S^1}(\theta, \theta')^2}$; straight in (x, y), short way in θ; sample a box for (x, y) and Uniform$(-\pi, \pi)$ for θ

ALGORITHM `SO3 · dist, slerp, Haar sample` COST $O(1)$

in unit quaternions Q, Q′; $t \in [0, 1]$; a seeded Rng
out the rotation angle between them; the geodesic rotation; a Haar-uniform rotation

1: $d \leftarrow \langle Q, Q' \rangle$; if $d < 0$ then $Q' \leftarrow -Q'$, $d \leftarrow -d$ — the double cover: $-Q'$ is the same rotation, nearer on S^3
2: dist $\leftarrow 2 \arccos d$, computed as $4\,\text{atan2}(\|Q - Q'\|, \|Q + Q'\|)$ — arccos near 1 loses eight digits
3: $\omega \leftarrow \arccos d$; $\text{slerp}(t) \leftarrow \dfrac{\sin((1-t)\omega)}{\sin \omega} Q + \dfrac{\sin(t\omega)}{\sin \omega} Q'$, normalized; linear blend when $\omega \approx 0$
4: sample() $\leftarrow (g_0, g_1, g_2, g_3)/\|g\|$ with $g_i \sim \mathcal{N}(0, 1)$ — uniform on S^3, hence Haar on SO(3)
5: Euler Z-Y-Z: if $|R_{33}| = 1$ return Err(GimbalLock) carrying $\phi \pm \psi$; else (E.3, E.5, E.6)

Line 2 of the SO(3) box is a numerical honesty item worth a sentence: the first implementation of `dist` used $2 \arccos d$ directly, and `dist(Q, Q)` came out as 6×10^{-8}, which failed the definiteness check. The half-angle identity $\arccos x = 2\,\text{atan2}(\sqrt{1-x}, \sqrt{1+x})$ gives the same number to full precision, and a metric that is not definite to machine precision breaks every equality test built on it.

5.5 Implementation in Rust

The trait is the chapter. Everything else is four implementations of it, a product, and one generic function.

RUST `crates/manifold/src/lib.rs`

```rust
use prob::Rng;

/// THE OWNED ARTIFACT. Every planner from Chapter 6 on is generic over this trait
```

```rust
/// and never touches coordinates directly. `DIM` is the manifold dimension -- the
/// number of local coordinates -- not the embedding dimension (3 for SO(3), not 9).
pub trait Manifold: Copy + PartialEq + core::fmt::Debug {
    const DIM: usize;
    /// Sampling box for non-compact factors; `()` for compact spaces, which have
    /// a uniform distribution of their own and need no box.
    type Bounds;

    /// A metric in the sense of Definition C.2.1 -- property-tested, not assumed.
    fn dist(&self, other: &Self) -> f64;
    /// The geodesic from `self` to `other` at `t in [0, 1]`, taken the short way.
    fn interpolate(&self, other: &Self, t: f64) -> Self;
    /// Uniform with respect to the natural measure: Lebesgue on boxes, Haar on groups.
    fn sample(rng: &mut Rng, bounds: &Self::Bounds) -> Self;
}

/// The first generic planner-shaped function in the book: a geodesic local path.
/// Chapter 11's `trait Steer` defaults to exactly this.
pub fn geodesic_path<M: Manifold>(a: &M, b: &M, n: usize) -> Vec<M> {
    (0..=n).map(|i| a.interpolate(b, i as f64 / n as f64)).collect()
}
```

`S1` is a newtype around an `f64`, and the newtype is the point: the constructor wraps, so no value outside $(-\pi, \pi]$ can exist, and the metric and interpolation never see a seam.

RUST **crates/manifold/src/s1.rs**

```rust
use core::f64::consts::{PI, TAU};

/// S1 as a wrapped angle. The chart is (-pi, pi]; the type guarantees the representative.

#[derive(Clone, Copy, Debug, PartialEq)]
pub struct S1(f64);

impl S1 {
    /// Wrap into (-pi, pi]. This is the only way to build an S1, so the invariant holds
   everywhere.
    pub fn new(theta: f64) -> Self {
        let mut x = (theta + PI).rem_euclid(TAU);
        if x <= 0.0 { x += TAU; }
        S1(x - PI)
    }
    pub fn angle(self) -> f64 { self.0 }
}

impl Manifold for S1 {
    const DIM: usize = 1;
    type Bounds = ();

    /// min(|a - b|, 2pi - |a - b|): the quotient metric of R by 2pi?.
    fn dist(&self, o: &Self) -> f64 {
        let d = (self.0 - o.0).abs();
        d.min(TAU - d)
    }

    /// Travel the short way. At the antipode wrap() returns +pi: deterministic,
   documented.
```

```
    fn interpolate(&self, o: &Self, t: f64) -> Self {
        S1::new(self.0 + t * S1::new(o.0 - self.0).0)
    }

    fn sample(rng: &mut Rng, _: &()) -> Self {
        S1::new(rng.uniform(-PI, PI))
    }
}
```

The torus is a product of circles, and SE(2) is the ported `Pose2` with a weight attached. Note what `Se2` is *not*: it does not redefine the pose type. `Pose2::compose`, `::inverse` and `::act` already exist from Chapter 2 (the sister book derived them); this chapter only adds the planner's view of them.

RUST `crates/manifold/src/torus.rs, se2.rs`

```rust
/// T^n as an array of circles, with the l2 product metric. Reach's `T2` from Chapter 2
/// is `Torus<2>` through a `From` impl, so no code in `cspace` changes.
#[derive(Clone, Copy, Debug, PartialEq)]
pub struct Torus<const N: usize>(pub [S1; N]);

impl<const N: usize> Manifold for Torus<N> {
    const DIM: usize = N;
    type Bounds = ();
    fn dist(&self, o: &Self) -> f64 {
        self.0.iter().zip(&o.0).map(|(a, b)| a.dist(b).powi(2)).sum::<f64>().sqrt()
    }
    fn interpolate(&self, o: &Self, t: f64) -> Self {
        let mut out = *self;
        for i in 0..N { out.0[i] = self.0[i].interpolate(&o.0[i], t); }
        out
    }
    fn sample(rng: &mut Rng, _: &()) -> Self {
        Torus(core::array::from_fn(|_| S1::sample(rng, &())))
    }
}

/// SE(2) is the ported Pose2. The metric needs an exchange rate between metres and
/// radians, and that rate has no canonical value -- so it is a field, never a default.
#[derive(Clone, Copy, Debug, PartialEq)]
pub struct Se2 { pub pose: Pose2, pub w: f64 }

impl Manifold for Se2 {
    const DIM: usize = 3;
    type Bounds = Aabb; // sampling box for (x, y); theta is uniform on S1

    fn dist(&self, o: &Self) -> f64 {
        let dth = S1::new(self.pose.theta).dist(&S1::new(o.pose.theta));
        (self.pose.x - o.pose.x).hypot(self.pose.y - o.pose.y).hypot(self.w * dth)
    }
    fn interpolate(&self, o: &Self, t: f64) -> Self {
        let th = S1::new(self.pose.theta).interpolate(&S1::new(o.pose.theta), t);
        Se2 { w: self.w, pose: Pose2 {
            x: self.pose.x + t * (o.pose.x - self.pose.x),
            y: self.pose.y + t * (o.pose.y - self.pose.y),
            theta: th.angle(),
        } }
```

```
    }
    fn sample(rng: &mut Rng, b: &Aabb) -> Self {
        Se2 { w: 0.0, pose: Pose2 { x: rng.uniform(b.min.x, b.max.x), y: rng.uniform(b.
min.y, b.max.y),
                                    theta: rng.uniform(-PI, PI) } }
    }
}
```

`So3` wraps `nalgebra`'s `UnitQuaternion`, which already gives the product, the conjugate, and `slerp`. What the chapter adds is the metric with the double cover accounted for, Haar sampling, and the Euler inverse that *returns an error* at the lost chart instead of a number.

RUST **crates/manifold/src/so3.rs**

```rust
use nalgebra::{Quaternion, SVector, UnitQuaternion};

#[derive(Clone, Copy, Debug, PartialEq)]
pub struct So3(pub UnitQuaternion<f64>);

#[derive(Debug, Clone, Copy, PartialEq)]
pub struct GimbalLock { /// The one quantity that survives at R_3_3 = +/-1: phi + psi (or
    phi - psi).
                        pub sum: f64 }

impl So3 {
    /// Z-Y-Z Euler angles, eqs. E.3, E.5, E.6, on the chart {R_3_3 != +/-1}. Off the
chart
    /// there is no answer, and the type says so.
    pub fn euler_zyz(&self) -> Result<[f64; 3], GimbalLock> {
        let r = self.0.to_rotation_matrix();
        let r33 = r[(2, 2)].clamp(-1.0, 1.0);
        if 1.0 - r33.abs() < 1e-9 {
            let sum = if r33 > 0.0 { r[(1, 0)].atan2(r[(0, 0)]) }        // E.9: phi +
psi
                      else { (-r[(0, 1)]).atan2(-r[(0, 0)]) };          // E.11: phi -
psi
            return Err(GimbalLock { sum });
        }
        let theta = (1.0 - r33 * r33).sqrt().atan2(r33);
        Ok([r[(1, 2)].atan2(r[(0, 2)]), theta, r[(2, 1)].atan2(-r[(2, 0)])])
    }

    /// Eqs. E.19-E.21: axis and angle, theta in [0, pi], with the sign fixed by w >= 0.
    pub fn axis_angle(&self) -> (SVector<f64, 3>, f64) {
        let q = if self.0.w < 0.0 { -self.0.into_inner() } else { self.0.into_inner() };
        let angle = 2.0 * q.w.clamp(-1.0, 1.0).acos();
        let v = q.imag();
        let n = v.norm();
        (if n < 1e-12 { SVector::z() } else { v / n }, angle)
    }
}

impl Manifold for So3 {
    const DIM: usize = 3;
    type Bounds = ();

```

```rust
    /// 2.acos|?q, q'?|, via 4.atan2(||q - q'||, ||q + q'||) after a sign flip: the same angle,
    /// but acos near 1 loses eight digits and a metric must be definite to machine precision.
    fn dist(&self, o: &Self) -> f64 {
        let (a, mut b) = (self.0.into_inner(), o.0.into_inner());
        if a.dot(&b) < 0.0 { b = -b; }
        4.0 * (a - b).norm().atan2((a + b).norm())
    }
    /// Spherical linear interpolation toward the nearer of +/-q' -- the short way on S3.
    fn interpolate(&self, o: &Self, t: f64) -> Self {
        let b = if self.0.dot(&o.0) < 0.0 { UnitQuaternion::new_unchecked(-o.0.into_inner()) } else { o.0 };
        So3(self.0.slerp(&b, t))
    }
    /// Haar-uniform: a normalized Gaussian 4-vector is uniform on S3.
    fn sample(rng: &mut Rng, _: &()) -> Self {
        let q = Quaternion::new(rng.normal(), rng.normal(), rng.normal(), rng.normal());
        So3(UnitQuaternion::from_quaternion(q))
    }
}

/// A * B with the l2 product metric -- SE(2) * S1 for Hitch, Chapter 14's composites.
#[derive(Clone, Copy, Debug, PartialEq)]
pub struct Product<A: Manifold, B: Manifold>(pub A, pub B);
```

5.5.1 The type system knows that $T^2 \neq SE(2)$

The compile error is a feature. Choset's §3.7 lists the configuration spaces of common robots and then a list of inequalities — $S^1 \times S^1 \times S^1 \neq SO(3)$, $SE(2) \neq \mathbb{R}^3$, $SE(3) \neq \mathbb{R}^6$ — that a planner written against coordinates cannot see. One written against `Manifold` cannot *fail* to see them:

RUST **crates/manifold/tests/ui/mix_spaces.rs (checked by trybuild: must NOT compile)**

```rust
use manifold::{geodesic_path, Se2, Torus};

fn main() {
    let q_reach: Torus<2> = reach_home();          // Reach lives on T2
    let rusty: Se2 = Se2 { pose: rusty_pose(), w: 0.5 }; // Rusty lives on SE(2)
    let _path = geodesic_path(&q_reach, &rusty, 10);
    //                                ^^^^^^ error[E0308]: mismatched types
    //                                expected `&Torus<2>`, found `&Se2`
}
```

`trybuild` asserts that this file fails to compile with exactly that message. There is no runtime check anywhere in the planners for "are these two configurations in the same space?", because there does not need to be.

5.5.2 The worked example, and its printed output

RUST `crates/manifold/examples/seams.rs`

```rust
fn main() {
    let (a, b) = (S1::new(0.1), S1::new(6.2));
    println!("S1 dist(0.1, 6.2) = {:.4}  chart |a-b| = {:.4}  midpoint = {:.4} (chart midpoint {:.4})",
        a.dist(&b), (0.1f64 - 6.2).abs(), a.interpolate(&b, 0.5).angle(), (0.1 + 6.2) / 2.0);
    let (p, q) = (Torus([S1::new(0.1), S1::new(3.0)]), Torus([S1::new(6.2), S1::new(3.5)]));
    println!("T2 dist = {:.4}", p.dist(&q));
    let w = 0.5;
    let (o, r) = (Se2 { pose: Pose2::new(0.0, 0.0, 0.0), w }, Se2 { pose: Pose2::new(1.0, 0.0, PI), w });
    println!("SE2 (w=0.5) dist((0,0,0),(1,0,pi)) = {:.4}", o.dist(&r));
}
```

```
S1 dist(0.1, 6.2) = 0.1832  chart |a-b| = 6.1000  midpoint = 0.0084 (chart midpoint 3.1500)
T2 dist = 0.5325
SE2 (w=0.5) dist((0,0,0),(1,0,pi)) = 1.8621
```

```
T_AC = Pose2 { 2, 1, 1.5708 }  w_B = (-3, 1)  w_A = (1, -1)  body: (-2,-1,0)  world: (2,1,0)  Dphi.(1,0) = (-1.4142, 0)
```

The tests beside these examples are the chapter's contract. `s1_seam_distance`, `s1_midpoint_short_way`, `t2_distance` and `se2_weighted_distance` assert the four numbers against their closed forms to 10^{-12}; `choset_frames` asserts T_{AC}, w_B, w_A and the body/world placements exactly; `metric_axioms` is a `proptest` asserting definiteness, symmetry and the triangle inequality on 10,000 seeded triples for each of `S1`, `Torus<2>`, `Se2` and `So3`; `interpolate_geodesic` asserts the endpoints and $d(a, \gamma(t)) = t\, d(a, b)$ to 10^{-9}; `euler_gimbal_lock` asserts `Err(GimbalLock)` at $R_{33} = 1$ with the right $\phi + \psi$ and an exact round trip elsewhere; `so3_dist_is_angle` asserts `dist` equals the axis–angle of $R^{\mathsf{T}}R'$; `dphi_chart_independent` asserts the Jacobian is unchanged under a 2π chart shift. The TypeScript port in `web/lib/manifold/` runs the same checks — every widget on this page is that port, and every number printed above was produced by it.

> **NOTE**
>
> Why the weight w is a field and not a constant: with $w = 0.5$ the two poses in the example are 1.86 apart; with $w = 2$ they are 6.36 apart and a half-turn in place costs more than driving six metres. Both are metrics. Which one a PRM should use depends on the robot and the task, and Chapter 11 measures the consequences. Hiding w in a default would hide a modelling decision.

5.6 Putting it together: components on the square and on the torus

The integration lab turns §3.4.3 into a measured quantity. Take Chapter 4's raster of Reach's C-space on the Workbench — here a 48×48 grid over $[-\pi, \pi]^2$ for the arm among three disc obstacles, built inside this chapter's module because the general raster lands with Chapter 4 — and count connected components of the free cells twice: once as a square, with 4-connectivity and no wrapping, and once as a torus, with the opposite edges identified.

The square has **2** components; the torus has **1**. One of the discs lies within reach of the first link, and a first-link collision does not care about θ_2, so its C-obstacle is a band across the whole chart. The band cuts the square in two. On the torus the two halves meet through the seam at $\theta_1 = \pm\pi$, and `seam_connection` finds the closest pair of cells that the square separates and the torus joins: two cells at $d_{T^2} = 0.2618$ — two raster cells apart — whose chart distance is 6.0214. That is the purple path of the hook, measured.

Monotonicity is the invariant the lab asserts: identifying edges can only *merge* components, never split them, so the torus count is at most the square count, on every raster. With joint limits $|\theta_i| < 3.0$ the outermost ring of cells is forbidden, the seam becomes a wall, and both counts are **2**: the topology changed, and the count noticed.

The last thread to tie is Chapter 4's Jacobian. At $q = (\pi/4, \pi/2)$ the matrix of $D\varphi_q$ sends $\dot{q} = (1, 0)$ to $(-\sqrt{2}, 0) = (-1.4142, 0)$, Choset's Example 3.8.1, and shifting θ_1's chart by 2π changes no entry by more than 10^{-15}. The Jacobian is the matrix of a chart-independent linear map between tangent spaces; Chapter 20 makes tangent spaces formal and Chapter 7 uses J^{T} to pull forces back from the workspace to the joints.

Everything downstream is now generic. Chapter 6 builds graphs whose edge costs are `dist`; Chapter 7 needs the tangent spaces this chapter named informally; Chapter 11's kd-tree splits on `Chart::coords` with `periodic` axes and its samplers call `sample`; Chapters 12–14's tree planners and composite spaces are generic over `Manifold`; Chapter 17 needs SO(3)'s inertia; Chapters 20–21 live in SE(2) as the car's group. None of them will do arithmetic on coordinates.

5.7 Exercises

1 FOUNDATION • **Two charts for the circle, two for the sphere**

Find two charts for S^1 and prove they form an atlas (Choset problem 3.9). Then explain why latitude–longitude is not a global chart on S^2 — what happens at the poles, and along the date line? — and exhibit a two-chart atlas for the sphere (problem 3.10). Which chart does `manifold::S1` use, and where does its domain end?

HINTS

<>A chart's domain must be open and the map must be a diffeomorphism onto an open set of $\mathbb{R}^k$. The two angle charts of Derivation 1 each omit one point.</> <>At a pole every longitude names the same point: the map is not injective there. Along the date line the longitude jumps by 2π: not continuous. Neither can be fixed by changing the convention.</> <>Stereographic projection from the north pole covers everything but the pole; from the south pole, everything but the south pole. The transition map is inversion in the unit circle, smooth away from the origin.</>

2 **FOUNDATION** •• **Homeomorphic, not isomorphic**

Show that SE(2) and $\mathbb{R}^2 \times SO(2)$ with the product operation $(x_1, R_1)(x_2, R_2) = (x_1 + x_2, R_1 R_2)$ are homeomorphic as spaces but not isomorphic as groups (Choset problem 3.23). Which one is commutative? Which one is `Pose2::compose`? Give a concrete pair of planar motions for which the two products disagree, and say which product describes what Rusty actually does.

3 **CONCEPTUAL** • **Predict the seam, then verify**

PREDICT FIRST

In the Torus Unwrapper, place the path endpoints at $(\theta_1, \theta_2) = (-3.0, 0)$ and $(3.0, 0)$. Before reading the readout: what is dist, and where does the purple path cross the seam?

- dist ≈ 6.0, no seam crossing: the path runs straight across the square
  That is the chart distance. The torus distance is the shorter of the two arcs on each circle factor, and $|-3.0 - 3.0| = 6.0$ is longer than $2\pi - 6.0 = 0.283$.
- ✓ dist ≈ 0.28, crossing the $\theta_1 = \pm\pi$ seam
  Right. d_{S^1}(−3.0, 3.0) $= 2\pi - 6.0 = 0.2832$, with θ_2 unchanged, so d_{T^2} = 0.2832 and the geodesic goes through $\theta_1 = \pi \equiv -\pi$. On the square it is two short stubs at opposite edges; on the torus one short curve.
- dist ≈ 3.0, crossing both seams
  Only θ_1 differs between the endpoints; $\theta_2 = 0$ on both, so the θ_2 seam is never approached.

Now turn joint limits on with $|\theta_1| < 3.1$. The endpoints are still legal configurations, but the space is an open rectangle: no path may cross the seam, so the shortest legal path is the long way across the square at chart distance 6.0 — and the component count changes, because the band of C-obstacle that the seam had bypassed now separates the two halves.

4 CONCEPTUAL •• **Read the lost chart**

In the Rotation Zoo, scrub to the frame where R_{33} is closest to 1 and read ϕ and ψ. Predict their sum from the quaternion readout using eq. (E.9), $\phi + \psi = \text{atan2}(R_{21}, R_{11})$ with $R_{21} = 2(q_1 q_2 + q_0 q_3)$ and $R_{11} = 2(q_0^2 + q_1^2) - 1$ from (E.28); verify against the dashed curve. Then explain why a controller that commands $\dot{\phi}$ and $\dot{\psi}$ separately misbehaves there, and what it should command instead.

WORK IT OUT

With the default motion (miss = 0.08 rad), what is the smallest value R_{33} reaches, to four decimals?

Answer. 0.9968 The body z-axis passes within 0.08 rad of the world z-axis, so $R_{33} = \cos\theta_{\min} = \cos 0.08 = 0.9968$. The Euler polar angle θ never reaches zero, so the chart is never *lost* — but the swing of ϕ and ψ is already nearly a half-turn each, in opposite directions, over a short interval. Near a lost chart the coordinate velocities blow up even though the body's angular velocity is constant; command the angular velocity, or the quaternion rate, instead.

5 PRACTICAL •• **The sphere is not the torus**

Implement `Manifold` for the sphere S^2 as a unit vector in $\mathbb{R}^3$: great-circle `dist`, `slerp` as `interpolate`, and a uniform `sample` (normalize a Gaussian 3-vector). Property-test the metric axioms and the geodesic property as `metric_axioms` does. Then write a test that takes identical-looking coordinates (θ_1, θ_2), interprets them once as a point of `Torus<2>` and once — via latitude and longitude — as a point of your `S2`, and asserts that `dist` to a second such pair *differs*. Make the test's name say what it proves: $T^2 \neq S^2$.

6 PRACTICAL ••• **Exponential coordinates**

Add `exp/log` to `Se2` and `So3`, reusing the sister book's `Pose2` exponential and `nalgebra`'s quaternion logarithm, and implement a second interpolation $\gamma(t) = a \cdot \exp\big(t \log(a^{-1}b)\big)$. Show with a test that on SE(2) it coincides with the chart-based `interpolate` only when the translation and rotation are decoupled — straight-line translation with constant heading — and report the maximum discrepancy in `dist` over 1,000 seeded pairs otherwise, as a function of w. The micro-Lie theory paper in the references is the map for this exercise.

5.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

§3.4–3.6 and Appendices B, C and E are this chapter's source: the definitions in their order, the circle/ellipse/racetrack, the four charts of S^1, the three-frame example pinned by the tests, and the Euler-angle inverse. `link`

Lozano-Pé rez, T. (1983). Spatial Planning: A Configuration Space Approach. *IEEE Transactions on Computers C-32(2), 108–120*.
The paper that made configuration space the planner's space; the C-obstacles on the torus in the hook descend from its figures. doi:10.1109/TC.1983.1676196

Shoemake, K. (1985). Animating Rotation with Quaternion Curves. *ACM SIGGRAPH Computer Graphics 19(3), 245–254*.
Spherical linear interpolation — the geodesic on SO(3) that So3::interpolate implements — and the case for quaternions over Euler angles, made with the gimbal-lock argument this chapter repeats. doi:10.1145/325334.325242

Solà, J., Deray, J., and Atchuthan, D. (2018). A micro Lie theory for state estimation in robotics. *arXiv:1812.01537*.
The exponential-coordinates view of SE(2) and SO(3) that Exercise 6 asks for, written for roboticists; the ⊞/⊟ operators the sister book uses come from here. link

Lynch, K. M. and Park, F. C. (2017). Modern Robotics: Mechanics, Planning, and Control. *Cambridge University Press*.
Chapter 3 is the fullest modern treatment of SO(3), SE(3), exponential coordinates and the body/world distinction, by one of Choset's co-authors. link

Lee, J. M. (2012). Introduction to Smooth Manifolds. *Springer, Graduate Texts in Mathematics 218, 2nd edition*.
Where to go when 'locally homeomorphic to $\mathbb{R}^k$' is not enough: charts, atlases, tangent spaces and the differential done properly, in the notation this chapter borrows. doi:10.1007/978-1-4419-9982-5

Thrun, S., Burgard, W., and Fox, D. (2005). Probabilistic Robotics. *MIT Press*.
The sister volume's source; its Chapter 3 web treatment of SE(2) as a Lie group, with exp/log and ⊞/⊟, is the cross-link this chapter defers to for rotation machinery. link

Part II

Classical Planners: Search, Potentials, Roadmaps, Cells

CHAPTER 6

Graph Search for Planners

CLASSICAL PLANNERS — SEARCH, POTENTIALS, ROADMAPS, CELLS · Foundational · 60 min

> *To distinguish between these forms of optimality, let us reserve the term optimality to measure the path and efficiency to measure the search, i.e., the number of nodes visited to determine the path.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), Appendix H.2*

Every planner in Parts II and III ends the same way: with a graph and a question — *which vertices lead from here to there, cheapest?* This chapter builds the one tool they all share. The reader knows what $\mathcal{Q}_{\text{free}}$ is and how to test a configuration for collision. None of that tells Rusty which way to drive.

The idea the chapter unpacks fits in one sentence. **Best-first search is one algorithm with one knob.** Priority by edge count gives breadth-first search; by cost-to-come g, Dijkstra; by a guess h alone, greedy search; by $g + h$ with an *optimistic* h, A* — optimal and lazy at once. Push the knob past optimism and optimality quietly breaks, and the chapter shows exactly where, on a graph small enough to trace by hand.

Two consequences follow and get their own widgets. A search can be *repaired* rather than redone when the world changes — D, *and its modern descendant D* Lite — and a search run *backward from the goal* yields a plan for every vertex at once, the reader's first value function, which the sister book turns into Markov decision processes.

◎ AFTER THIS CHAPTER YOU CAN

- Write breadth-first search, Dijkstra, greedy search and A* as one best-first loop with one priority rule, and say which guarantee each rule buys
- Prove that A* terminates, is complete, and is optimal when the heuristic

is admissible — and show with a four-node graph exactly where the proof fails when it is not
- Explain why a consistent heuristic never reopens a node, and what weighted A* trades for its speed
- Repair a search after the world changes with D *and D* Lite, and say which states a repair must touch
- Turn a backward Dijkstra into a universal plan, and recognize its labels as a Bellman fixed point
- State what complete, resolution complete and probabilistically complete mean, and where motion planning sits among P, NP and PSPACE

ASSUMED

- Rusty, the Apartment and the simulator (Chapter 2)
- Configuration space and the rasterized free space of a disc robot (Chapter 4)
- A metric on a configuration space, so that edges have costs (Chapter 5)

6.1 The problem: a goal is not a plan

Put Rusty in room A of the Apartment and the goal in room B, and write the first controller anyone writes: drive toward the goal.

The goal-seeking controller. Rusty drives straight at a goal two rooms away and wedges against the party wall between them. Nothing in 'drive toward the goal' knows that the doorway is behind it.

It wedges into the first wall. The goal is 2.2 m away through the plaster, the doorway is two metres behind and to the left, and nothing in the controller can know that, because the controller only looks at where it is and where it wants to be. Something must *look ahead* — must consider positions Rusty is not in, order them by how promising they are, and remember which one led to which. That something is a graph search, and the rest of the chapter is what the words "promising" and "remember" mean precisely.

6.2 Building intuition: one search, one knob

Rasterize the Apartment's south rooms and corridor into a grid of half-metre cells, connect each cell to its eight neighbors, and search it four ways at once.

INTERACTIVE · w6.1

Search Theater

A stronger heuristic means fewer expansions only while it stays optimistic. Past that, it buys speed with path quality — a better heuristic does not always mean a better path.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch06-graph-search#w6.1

The four lanes run the *same* loop. Pop the most promising node, mark it done, look at its neighbors, push the ones not yet done. The only thing that differs is what "most promising" means:

- **Breadth-first** ranks by *link length*, the number of edges from the start. It floods outward in rings, and the first time it reaches the goal it has found the path with the fewest edges — which, on a grid where a diagonal step costs 1.4 and an axial one costs 1, is not the cheapest path.
- **Dijkstra** ranks by g, the cost of the cheapest known path from the start. It floods in *cost* rings, and the first time it pops the goal it has the cheapest path — at the price of expanding every cell cheaper than the goal.
- **Greedy** ranks by h, a guess at the cost from each node to the goal. It darts toward the goal and often arrives in a handful of expansions with a path that is visibly bad.
- **A*** ranks by $f = g + \varepsilon h$. With $\varepsilon = 1$ it expands only the cells it must and matches Dijkstra's cost to the last digit.

Now slide ε. Below one, the A *lane's cost does not move; only its expansion count does, climbing toward Dijkstra's as* $\varepsilon \to 0$ *and the heuristic switches off. Above one the expansion count keeps falling — and the path cost drifts above Dijkstra's. The badge marks the first* ε *at which that happens. This is the misconception the widget exists to kill:* a better heuristic always means a better path. *A stronger heuristic means fewer expansions* only while it stays optimistic*; past that, it buys speed with path quality, and the chapter's derivations say how much of each.

THE ONE SENTENCE VERSION

Breadth-first, Dijkstra, greedy and A* are one best-first loop with four priority rules. The rule decides the guarantee: link-optimal, cost-optimal, no guarantee, or cost-optimal if — and only as long as — the heuristic never overestimates.

6.3 The mathematics

$G = (V, E),\ c(n_1, n_2) \geq 0$	a graph with non-negative edge costs (weights)
$\text{Star}(n)$	the set of nodes adjacent to n — Choset's name for the neighbor set
$g(n),\ h(n),\ f(n)$	cost-to-come along the back-pointer path; heuristic estimate of the cost-to-go; the priority book-wide
$O,\ C$	the open set (priority queue) and the closed set (expanded nodes)
$b(n)$	back pointer: the parent of n on the best known path from q_start
$\ell(\pi)$	link length of a path — number of edges, weights ignored
$k(X),\ t(X),\ r(X, Y)$	D*: minimum key, tag ∈ {NEW, OPEN, CLOSED}, measured arc cost
$rhs(s),\ k_m$	D* Lite: one-step lookahead value, key modifier
$\varepsilon \geq 1$	heuristic inflation factor (weighted A, *ARA*)
$\pi^\star : V \to V$	universal plan (policy): the successor to take from every vertex
$O(\cdot), o(\cdot), \Omega(\cdot), \Theta(\cdot)$	asymptotic bounds (Choset Def. G.1.1)

6.3.1 Graphs, grids, and two kinds of length

DEFINITIONS — GRAPH, PATH, CYCLE, TREE (CHOSET §H.1)

A **graph** $G = (V, E)$ is a set of nodes and a set of edges, directed or undirected, each with a non-negative **weight** c. A **path** is a sequence of nodes with consecutive nodes adjacent; a **cycle** is a path whose first and last nodes coincide; a **tree** is a connected directed graph without cycles, with a unique **root** that has no incoming edge. A **grid** induces a graph whose nodes are cells; **four-point connectivity** joins N/S/E/W neighbors, **eight-point** adds the diagonals.

The weights on a grid are the chapter's first honesty item. Choset's §H.2.4 charges 1 for an axial step and 1.4 for a diagonal — his approximation of $\sqrt{2}$, chosen so that hand arithmetic comes out in tenths. This is *not* the Euclidean metric: two cells sideways and one up costs 2.4, not $\sqrt{5} = 2.24$. It is a perfectly good metric on the graph, and the design of this chapter keeps it, because it is what makes the examples traceable. Code that wants $\sqrt{2}$ changes one constant.

The **link length** $\ell(\pi)$ of a path is its number of edges, weights ignored. Breadth-first search minimizes link length; Dijkstra minimizes weighted cost; the two coincide only when all weights are equal. On the 1/1.4 grid they disagree in a concrete way: four axial steps cost 4.0 and three diagonals cost 4.2, so a path with *more* links can be *cheaper*. This is why the Search Theater's breadth-first lane reports a different cost from Dijkstra's, and why the Rust port relaxes a node by depth in breadth-first mode and by cost everywhere else.

6.3.2 Expansion, open and closed, and what a heuristic promises

Expanding a node n (Choset §H.2.1) moves it from the open set O to the closed set C and, for every $x \in \mathrm{Star}(n) \setminus C$, either inserts x into O with $g(x) = g(n) + c(n, x)$ and $b(x) = n$, or — if x is already open and the new route is cheaper — updates $g(x)$ and $b(x)$. The closed set is what stops a search from walking in circles; the open set is the frontier between what is known and what is not. In the colors every figure on this page uses:

$f(n) = g(n) + \varepsilon\, h(n)$, $O \cup C$ is what the search has looked at, $q_{\text{start}} = b^k(q_{\text{goal}}), \ldots, b(q_{\text{goal}}), q_{\text{goal}}$ is what it returns.

The cost-to-come g is measured from the green start; the heuristic h is an estimate toward the red goal; the blue explored set is the price of the search; the purple back-pointer path is its product. Point at a term and the matching layer in the Search Theater comes forward.

DEFINITIONS — ADMISSIBLE AND CONSISTENT HEURISTICS

Let $h^\star(n)$ be the true cost of the cheapest path from n to q_{goal}. A heuristic is **admissible** (Choset: *optimistic*) if $h(n) \le h^\star(n)$ for every n. It is **consistent** if $h(n) \le c(n, x) + h(x)$ for every edge (n, x) and $h(q_{\text{goal}}) = 0$. Consistency implies admissibility: sum the inequality along an optimal path from n to the goal.

Consistency is not in Choset; it is the standard strengthening that makes the no-reopening theorem below true, and the octile heuristic — $h = 1.4 \min(\Delta x, \Delta y) + |\Delta x - \Delta y|$, the exact cost-to-go on an *empty* 1/1.4 grid — has it. Exercise 1 asks for the proof, and for the slightly embarrassing fact that the same formula is not admissible if diagonals cost $\sqrt{2}$.

DERIVATION **Breadth-first search is optimal in link length**

depth at first dequeue $= \min \ell(\pi)$

Statement. With a FIFO queue, a node's depth when it is first dequeued is its shortest link length from q_{start} — on a grid, the Manhattan (4-connected) or chessboard (8-connected) shortest path (Choset §H.1–H.2).
Step 1 — the queue invariant. At every moment the queue holds nodes of some depth d followed by nodes of depth $d + 1$, and nothing else: the start is at depth 0, and dequeuing a depth-d node enqueues only depth-$(d + 1)$ nodes behind the remaining depth-d ones.
Step 2 — first discovery. A node is enqueued the first time it is seen, from a parent at depth d, and is assigned depth $d + 1$; later sightings are ignored.

Step 3 — no shorter path exists. Suppose x were reachable in $d' < d + 1$ links. Then its parent on that path has depth $d' - 1 \leq d - 1$ and, by the invariant, was dequeued *before* any depth-d node — at which moment x would have been enqueued at depth $d' \leq d$. Contradiction.
Step 4 — grids. On a unit-weight grid link length *is* the grid metric, so the first path to the goal is also cheapest. With 1/1.4 weights it is not: see the paragraph above. ■
Why depth-first is different. A stack pushes and pops on the same end, so the search commits to a branch and backs up only at a dead end. It is complete on a finite graph — every node is visited at most once — and optimal in nothing (Choset's figures H.8–H.9). The chapter's check confirms the first claim on two hundred random grids and does not bother with the second.

DERIVATION **A* terminates and is complete**

\text{pops } q_{goal} \text{ or empties } O \text{ — finitely}

Statement. On a finite graph, A* (Alg. 24) either pops q_{goal} or empties O; in the latter case no path exists (Choset §H.2.2).
Step 1. Each expansion moves a node into C. Under a consistent heuristic a node is never removed from C again, so there are at most $|V|$ expansions. Without consistency a closed node may be *reopened* when a cheaper path to it is found; but each reopening strictly decreases its g, and g takes values among the costs of acyclic paths, of which a finite graph has finitely many. Choset's phrasing: A* searches a tree, a tree has finitely many acyclic paths, so the work is finite.
Step 2. V is finite, so by Step 1 the number of expansions is finite and the loop terminates.
Step 3. While a path from q_{start} to q_{goal} exists, O contains some node on it: the start is on it and in O initially; whenever a path node is expanded, its successor on the path is inserted or already present. So O cannot empty while a path exists, and if O empties, no path exists. ■

DERIVATION **A* is optimal under admissibility**

g(q_{goal}) = c^\star \text{ when first popped}

Statement. If h is admissible, then the first time q_{goal} is popped, $g(q_{\text{goal}}) = c^\star$, the cost of a cheapest path (formalizing Choset §H.2.2).
Step 1 — suppose not. Assume q_{goal} is popped with $g(q_{\text{goal}}) > c^\star$.

Step 2 — a frontier node on an optimal path. Fix an optimal path $q_{\text{start}} = v_0, v_1, \ldots, v_m = q_{\text{goal}}$. Let v be the first node on it not yet expanded. Its predecessor has been expanded with the optimal g (induction along the path), so $v \in O$ with $g(v) = g^\star(v)$.
Step 3 — bound its priority. $f(v) = g^\star(v) + h(v) \le g^\star(v) + h^\star(v) = c^\star$, using admissibility and the fact that v lies on an optimal path.
Step 4 — compare with the goal. $f(q_{\text{goal}}) = g(q_{\text{goal}}) + 0 > c^\star \ge f(v)$.
Step 5 — contradiction. The queue pops the smallest f, so it would pop v before q_{goal}. Hence the assumption fails and $g(q_{\text{goal}}) = c^\star$. ■
Reconciling two stopping rules. Alg. 24 exits when the goal is *popped*. Choset's §H.2 walk-through keeps expanding nodes whose f is below the pushed goal's and discards the rest. These are the same rule: a node is popped only when nothing in O beats it, so the walk-through's "discard everything with $f \ge$ the goal's" is what the pop does implicitly.

DERIVATION **Consistency means no node is ever reopened**

f \text{ non-decreasing along pops} \Rightarrow g \text{ optimal at closing}

Statement. With a consistent h, the popped f-values are non-decreasing, and every node's g is already optimal when it enters C.
Step 1. For an edge (n, x), $f(x) = g(n) + c(n, x) + h(x) \ge g(n) + h(n) = f(n)$ by consistency. A node's children enter the queue with priority at least their parent's.
Step 2. Hence the sequence of popped f-values is monotone non-decreasing: whatever is popped next was inserted by some earlier pop with no smaller f.
Step 3. Suppose a closed node x later received a cheaper g via some node y popped after x. Then $f(x)_{\text{new}} = g_{\text{new}}(x) + h(x) < f(x)_{\text{old}} \le f(y)$, contradicting monotonicity. So closed nodes stay closed and their g is optimal. ■
Consequences. Octile and Euclidean heuristics are consistent, so the "reopened" column in the Search Theater reads zero for them, and the chapter's check asserts it on two hundred random grids. An inflated εh with $\varepsilon > 1$ is not consistent — but it still satisfies $c \le \varepsilon\, c^\star$, since the argument of the previous derivation goes through with $f(v) \le g^\star(v) + \varepsilon h^\star(v) \le \varepsilon c^\star$. That is the weighted-A *bound, which anytime planners such as ARA* (Likhachev, Gordon and Thrun, 2003; see the literature) exploit by running a decreasing schedule of ε and reusing the open set between runs.

6.3.3 Where optimism fails: the four-node counterexample

DERIVATION **The non-optimistic counterexample**

$A^\star$ returns cost 8; optimum 5

Statement (Choset §H.2.5, figure H.20, with the numbers fixed). Edges S–A: 4, A–G: 4, S–B: 3, B–G: 2. Heuristic $h(S) = 2$, $h(A) = 3$, $h(B) = 10$, $h(G) = 0$. A* returns S, A, G at cost 8; the optimum S, B, G costs 5.
Step 1 — expand S. $f(A) = g(A) + h(A) = 4 + 3 = 7$; $f(B) = 3 + 10 = 13$.
Step 2 — expand A, the smaller f. $f(G) = (4 + 4) + 0 = 8$.
Step 3 — pop G. $8 < 13$, so the goal is popped before B is ever expanded. A* returns the back-pointer path S, A, G with $g(G) = 8$.
Step 4 — the culprit. $h(B) = 10 > h^\star(B) = 2$. In Step 3 of the optimality proof the inequality $f(v) \le c^\star$ needed $h(v) \le h^\star(v)$ at the frontier node $v = B$ on the optimal path; here $f(B) = 13 > 5 = c^\star$, and the chain of inequalities snaps. One bad estimate, anywhere on an optimal path, suffices. ■
Repair $h(B)$ to its true value 2 and A* pops B at $f = 5$ before G at 8, finds G via B at $f = 5$, and returns the optimum; the chapter's check runs both versions.

6.3.4 The hand trace: five pops on a 4×3 grid

Here is the micro-example the tests pin, in Choset's §H.2.4 style. An 8-connected 4×3 grid, cells (col, row) with col $\in \{1, \dots, 4\}$ and row $\in \{1, 2, 3\}$; obstacle cells $(2, 1)$ and $(2, 2)$ are *deleted* from the graph; $q_{\text{start}} = (1, 1)$, $q_{\text{goal}} = (4, 1)$; axial step 1, diagonal 1.4; octile heuristic. Ties in f break toward smaller h, then lexicographically.

pop	expanded	g	h	f	open set after expansion (f: cell)
1	$(1, 1)$	0	3.0	3.0	4.4: $(1, 2)$
2	$(1, 2)$	1.0	3.4	4.4	5.2: $(2, 3)$ · 5.8: $(1, 3)$
3	$(2, 3)$	2.4	2.8	5.2	5.2: $(3, 2)$ · 5.8: $(1, 3)$, $(3, 3)$
4	$(3, 2)$	3.8	1.4	5.2	5.2: $(4, 1)$ · 5.8: $(1, 3)$, $(3, 1)$, $(3, 3)$, $(4, 2)$ · 7.2: $(4, 3)$
5	$(4, 1)$	5.2	0	5.2	goal popped — done

Back pointers give $(1, 1) \to (1, 2) \to (2, 3) \to (3, 2) \to (4, 1)$ at cost $1 + 1.4 +$

$1.4 + 1.4 = \mathbf{5.2}$ in **five** expansions. Dijkstra on the same grid, $h \equiv 0$, expands all **ten** free cells — every other free cell has $g < 5.2$, so all of them are popped before the goal — and returns the same cost. Greedy, $f = h$, happens to find the same path here in five expansions, with no guarantee that it would anywhere else. All three facts are asserted by `astar_trace_matches_text` and the TypeScript check that mirrors it.

6.3.5 Repairing a search: D*, and universal plans

DERIVATION **D* repairs locally; backward Dijkstra is a universal plan**

h(X) = \min_Y [c(X,Y) + h(Y)];\ \text{repair touches only chains through the change}

Statement (Choset §H.3–H.4). Dijkstra from q_{goal} labels every node with its cost-to-goal $h(X)$ and a back pointer $b(X)$ — a policy. When an arc cost changes, only nodes whose best path used that arc need new labels.
Step 1. Backward Dijkstra is forward Dijkstra on the reversed graph; for the symmetric graphs this chapter searches, the same graph.
Step 2 — the fixed point. Its labels satisfy $h(X) = \min_Y [c(X, Y) + h(Y)]$ with $h(q_{\text{goal}}) = 0$: a *deterministic Bellman equation*. The policy is the arg min.
Step 3 — raised states. Increase the cost of an arc (X_c, Y). Only nodes whose back-pointer chain passes through that arc have an h that is now too small. D* re-inserts the affected states; when such a state is popped its key k — the smallest h it has had since insertion — is less than its current h. Choset calls it a **RAISE** state: its old route got worse.
Step 4 — lowered states. A neighbor whose chain avoided the change has a trustworthy h and can offer a better route: in Choset's gate example, cell $(3, 2)$ is raised when the gate $(4, 3)$ closes, and its neighbor $(4, 1)$, whose path never used the gate, offers $h(4, 1) + 1.4 = 6.2 + 1.4 = 7.6$ (figure H.26). A state popped with $k = h$ is a **LOWER** state: it has good news for its descendants.
Step 5 — when to stop. Propagation stops when the smallest key on the open list is at least the robot's own $h(X_c)$ (Alg. 25, line 20): nothing left on the queue could lower the robot's cost, so its back-pointer path is optimal again. ■
PROCESS-STATE (Alg. 31) is the case analysis of Steps 3–4 made exact; the Rust `dstar.rs` and the TypeScript port follow Stentz's original structure, in which a state that is lowered by the RAISE fix-up still propagates in the same call (Choset's typesetting has an `else if` there — see the honesty notes). *D* Lite* (Koenig and Likhachev) reaches the same behavior from Lifelong Planning A*: two values per vertex, g and a one-step lookahead rhs, a queue of* inconsistent* vertices, and a key modifier k_m that absorbs the

robot's own motion so the heap never needs re-keying. In the sister book's Chapter 21 the min in Step 2 becomes an expectation and the plan becomes a value function; nothing else changes.

6.3.6 Complexity and completeness

Choset Appendix G in one picture. Left: the complexity classes, nested, with the inclusions that are proven and the strictness that is open. Right: the three completeness notions this book uses as law; hover for the definitions.

Running time is measured asymptotically (Choset Def. G.1.1): $f \in O(g)$ if $f(n) \leq c\, g(n)$ for all large n; Ω is the lower bound, Θ both, o and ω the strict versions. A* with a binary heap runs in $O((V + E) \log V)$ in the worst case, the same as Dijkstra, and the heuristic buys a smaller constant — fewer expansions — not a better exponent.

DEFINITIONS G.2.5, G.2.7, G.2.11 — P, NP, PSPACE

A decision problem is in **P** if some algorithm decides it in polynomial time; in **NP** if a polynomial-length certificate for a "yes" instance can be verified in polynomial time; in **PSPACE** if some algorithm decides it in polynomial space. $P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$ (Choset §G.2). Shortest paths, and A* on an explicit graph, are in P. Generalized motion planning with many bodies is PSPACE-hard, which Chapter 9 needs when it quotes Canny's bound.

DEFINITIONS G.3.1–G.3.3 — THREE KINDS OF COMPLETE

An algorithm is **complete** if in finite time it finds a solution when one exists and otherwise reports that none does; **resolution complete** if it does so for a fixed resolution step $\epsilon > 0$; **probabilistically complete** if the probability of finding an existing solution tends to one as running time tends to infinity. A* is complete. A grid planner is resolution complete: it answers correctly about the grid. PRM and RRT, in Part III, are probabilistically complete and cannot report that no solution exists.

6.4 The algorithm

Choset's Algorithm 24 is the master template; everything else in this chapter is a special case, a repair of it, or a run of it in reverse.

ALGORITHM BEST-FIRST(G, q_start, q_goal, h, priority) -- Choset Alg. 24 with the knob exposed COST O((V + E) log V) with a binary heap; the heuristic shrinks the constant

in a graph G with Star(n) and costs c ≥ 0, start and goal, a heuristic h, a priority rule

out a back-pointer path from q_start to q_goal and its cost, or failure when O empties

1: $g(q_{\text{start}}) \leftarrow 0$; $O \leftarrow \{q_{\text{start}}\}$ keyed on f; $C \leftarrow \emptyset$
2: **repeat**
3: pick n_{best} from O with $f(n_{best}) \le f(n)$ for all $n \in O$ — ties: smaller h, then the graph's order
4: remove n_{best} from O and add it to C
5: **if** $n_{best} = q_{\text{goal}}$ **then** EXIT with the back-pointer path
6: **for all** $x \in \text{Star}(n_{best})$ not in C **do**
7: **if** $x \notin O$ **then** $g(x) \leftarrow g(n_{best}) + c(n_{best}, x)$; $b(x) \leftarrow n_{best}$; add x to O
8: **else if** $g(n_{best}) + c(n_{best}, x) < g(x)$ **then** update $g(x)$, $b(x)$ *and* x's priority — Alg. 24 prints only the back pointer
9: **until** O is empty — report that no path exists
10: **where** $f(n)$ = link length (breadth-first) · $g(n)$ (Dijkstra) · $h(n)$ (greedy) · $g(n) + \varepsilon h(n)$ (A, *weighted A*)

Two implementation notes that the proofs depend on. The open set is a *lazy* binary heap: improving a node pushes a fresh entry and the stale one is discarded when it surfaces, so line 8 never needs a decrease-key. And if a node already in C receives a cheaper g — which Derivation 4 says cannot happen under a consistent heuristic, and which does happen with $\varepsilon > 1$ — it is *reopened*: removed from C and pushed again. Correctness over speed, and a counter the widget shows.

ALGORITHM PROCESS-STATE(O, L) -- the heart of D* (Choset Alg. 31, Stentz's structure) COST O(deg · log |O|) per call; a repair runs it until k_min ≥ h(X_c)

in the open list O keyed on k, all states L with h, k, t, b

out the new k_min, with h, b and O updated

1: $X \leftarrow$ the state in O with minimum k; $k_{old} \leftarrow k(X)$; delete X from O, $t(X) \leftarrow$ CLOSED
2: **if** $k_{old} < h(X)$ **then** — RAISE: the old route got worse; look for a rescuer

3: **for each** neighbor Y with $h(Y) \le k_{old}$ and $h(X) > h(Y) + c(Y, X)$: $b(X) \leftarrow Y$; $h(X) \leftarrow h(Y) + c(Y, X)$
4: **if** $k_{old} = h(X)$ **then** — LOWER: good news for the descendants
5: **for each** neighbor Y: **if** $t(Y)$ = NEW, or $b(Y) = X$ and $h(Y) \neq h(X) + c(X, Y)$, or $b(Y) \neq X$ and $h(Y) > h(X) + c(X, Y)$ **then** $b(Y) \leftarrow X$; INSERT$(Y, h(X) + c(X, Y))$
6: **else** — still RAISE after the fix-up
7: **for each** neighbor Y: **if** $t(Y)$ = NEW, or $b(Y) = X$ and $h(Y) \neq h(X) + c(X, Y)$ **then** $b(Y) \leftarrow X$; INSERT$(Y, h(X) + c(X, Y))$
8: **else if** $b(Y) \neq X$ and $h(Y) > h(X) + c(X, Y)$ **then** INSERT$(X, h(X))$ — re-open X: it can still improve Y later
9: **else if** $b(Y) \neq X$ and $h(X) > h(Y) + c(X, Y)$ and $t(Y)$ = CLOSED and $h(Y) > k_{old}$ **then** INSERT$(Y, h(Y))$ — re-open a possible rescuer
10: **return** k_{min} of O — INSERT (Alg. 27): $k(X) \leftarrow h_{new}$ if NEW, $\min(k, h_{new})$ if OPEN, $\min(h, h_{new})$ if CLOSED; then $h(X) \leftarrow h_{new}$, $t(X) \leftarrow$ OPEN

ALGORITHM D*-LITE -- Koenig & Likhachev's ComputeShortestPath, backward from the goal COST each vertex expanded at most twice per repair; repair cost proportional to the affected region

in g, rhs tables (∞ except rhs(goal) = 0), a queue U keyed on [min(g,rhs) + h(start, s) + k_m ; min(g,rhs)]

out g(start) = cost-to-goal, with the greedy descent of g an optimal path

1: **while** TopKey$(U) \le$ Key(s_{start}) **or** $rhs(s_{start}) \neq g(s_{start})$ **do** — ties processed too, so the whole $f \le g(s_{start})$ band is consistent
2: $u \leftarrow$ Pop(U), $k_{old} \leftarrow$ its key
3: **if** $k_{old} <$ Key(u) **then** re-insert u with Key(u) — stale because k_m grew
4: **else if** $g(u) > rhs(u)$ **then** $g(u) \leftarrow rhs(u)$; UpdateVertex on every predecessor — overconsistent: commit the good news
5: **else** $g(u) \leftarrow \infty$; UpdateVertex on every predecessor and on u — underconsistent: forget, re-derive
6: **UpdateVertex**(u): **if** $u \neq s_{goal}$: $rhs(u) \leftarrow \min_{s'}[c(u, s') + g(s')]$; remove u from U; **if** $g(u) \neq rhs(u)$ insert u with Key(u)
7: **on edge changes**: $k_m \leftarrow k_m + h(s_{last}, s_{start})$; $s_{last} \leftarrow s_{start}$; UpdateVertex on both endpoints of every changed arc; run lines 1–5
8: **move**: $s_{start} \leftarrow \arg\min_{s'}[c(s_{start}, s') + g(s')]$

```
ALGORITHM UNIVERSAL-PLAN(G, q_goal) -- Choset §H.4    COST one full Dijkstra,
O((V + E) log V); every later query is a lookup
```

in a graph with symmetric non-negative costs and a goal

out cost-to-go h(X) and a successor π^*(X) for every reachable X

1: $h(q_{\text{goal}}) \leftarrow 0$; push q_{goal}; all other $h \leftarrow \infty$
2: **while** the queue is non-empty **do** pop the cheapest X
3: **for each** $Y \in \text{Star}(X)$: **if** $h(X) + c(X, Y) < h(Y)$ **then** $h(Y) \leftarrow h(X) + c(X, Y)$; $\pi^\star(Y) \leftarrow X$; push Y
4: **return** $(h, \pi^\star)$ — h satisfies $h(X) = \min_Y[c(X, Y) + h(Y)]$ everywhere; following $\pi^\star$ from any X costs exactly $h(X)$

6.4.1 Replanning, watched

INTERACTIVE · w6.2

D* Replanner

Replanning does not mean searching again. Most of the old search is still correct after the world changes, and D* proves which part — then fixes only the rest.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch06-graph-search#w6.2

With Choset's gate world the numbers are the ones in his figures. The initial backward Dijkstra labels the start $(2, 1)$ with $h = 7.0$ — five diagonals — and $(4, 1)$ with $h = 6.2$, and Rusty sets off along $(2, 1)$ $(3, 2)$ $(4, 3)$ $(5, 4)$ $(6, 5)$ $(7, 6)$. The gate closes; at $(3, 2)$ Rusty is adjacent and notices. D *re-inserts the gate and its neighbors, pops* $(3, 2)$ *as a RAISE state — its h has jumped above Choset's obstacle cost of* 10000 *while its key stays at the old* 7.0 *— and the LOWER neighbor* $(4, 1)$ *rescues it at* $6.2 + 1.4 = 7.6$. *The repair pops* ***15*** *states, five of them raised, and Rusty drives* $(3, 2)$ $(4, 1)$ $(5, 2)$ $(6, 3)$ $(7, 4)$ $(7, 5)$ $(7, 6)$ *at cost* 7.6. *Redoing the backward Dijkstra from scratch on the changed map would pop* ***32****; a from-scratch A** from $(3, 2)$ needs only **7** expansions on this tiny grid, which is Exercise 4's point — repair is not always cheaper than redo, and the raised/lowered classification says when it is.

6.4.2 A plan that is a field

INTERACTIVE · w6.3

Universal Plan Field

A plan is not a path. A plan can be a field — a successor for every cell — and a field is

what survives being pushed off course.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch06-graph-search#w6.3

A path is a plan for one start. Run the backward Dijkstra to exhaustion and every free cell of the Apartment carries a cost-to-go and an arrow; kick Rusty anywhere and the arrows bring it home. The equal-cost contours are wave fronts — Chapter 7's wave-front planner is this picture with unit costs — and the panel under the canvas evaluates $h(X) = \min_Y [c(X, Y) + h(Y)]$ at the cell under the cursor, term by term, with the minimizing Y being the arrow. The chapter's check asserts the equation holds at every labelled cell to 10^{-15} and that following the arrows from sixty random cells reaches the goal at exactly Dijkstra's cost for that start.

6.5 Implementation in Rust

The `search` crate is introduced here and imported by Chapters 7, 8, 10, 16, 18, 21 and 22. Its interface is the smallest thing best-first search needs.

RUST `crates/search/src/space.rs`

```rust
/// Anything best-first search can walk. Costs are non-negative edge weights
/// (Choset App. H.1); `Node` must be hashable so open/closed sets are maps.
pub trait SearchSpace {
    type Node: Copy + Eq + std::hash::Hash;
    /// Star(n) with edge costs c(n, x) -- Choset's notation, made an iterator.
    fn neighbors(&self, n: Self::Node) -> impl Iterator<Item = (Self::Node, f64)> + '_;
}

/// 8-connected lattice over a rasterized Q_free (Chapter 4's `cspace::Raster`). `diag` is
/// 1.4 for Choset's grid metric (figure H.16) or SQRT_2 for the exact one. Obstacle cells
/// are deleted from the graph: a neighbor that is occupied is simply not yielded.
pub struct Grid8<'a> { pub raster: &'a cspace::Raster, pub diag: f64 }

impl SearchSpace for Grid8<'_> {
    type Node = (i32, i32);
    fn neighbors(&self, (i, j): (i32, i32)) -> impl Iterator<Item = ((i32, i32), f64)> + '_ {
        const STEPS: [(i32, i32); 8] = [(1,0), (-1,0), (0,1), (0,-1), (1,1), (1,-1), (-1,1), (-1,-1)];
        STEPS.iter().filter_map(move |&(di, dj)| {
            let n = (i + di, j + dj);
            if !self.raster.is_free(n) { return None; }
            let diag = di != 0 && dj != 0;
            Some((n, if diag { self.diag } else { 1.0 } * self.raster.cell_size()))
        })
    }
}
```

The engine is one function. `Priority` is the knob; `Trace` records the pop order so the examples can print Choset-style tables and the widgets can paint the frontier.

RUST **crates/search/src/best_first.rs**

```rust
use std::cmp::Ordering;
use std::collections::{BinaryHeap, HashMap, HashSet};

/// The one knob: how a node's priority is formed from g, h and depth.
#[derive(Clone, Copy, Debug, PartialEq)]
pub enum Priority { LinkLength, CostToCome, HeuristicOnly, Sum { eps: f64 } }

impl Priority {
    fn f(self, g: f64, h: f64, depth: usize) -> f64 {
        match self {
            Priority::LinkLength => depth as f64,
            Priority::CostToCome => g,
            Priority::HeuristicOnly => h,
            Priority::Sum { eps } => g + eps * h,
        }
    }
}

/// One row of the trace table and the result of a search.
pub struct Pop<N> { pub node: N, pub g: f64, pub h: f64, pub f: f64 }
pub struct Trace<N> { pub pops: Vec<Pop<N>>, pub path: Vec<N>, pub cost: f64, pub reopened: usize }

/// Heap entries ordered by (f, h, seq): ties toward smaller h, then insertion order.
/// `Reverse` semantics are written out so the comparison is readable.
struct Entry<N> { f: f64, h: f64, seq: u64, g: f64, node: N }
impl<N> PartialEq for Entry<N> { fn eq(&self, o: &Self) -> bool { self.cmp(o) == Ordering::Equal } }
impl<N> Eq for Entry<N> {}
impl<N> PartialOrd for Entry<N> { fn partial_cmp(&self, o: &Self) -> Option<Ordering> { Some(self.cmp(o)) } }
impl<N> Ord for Entry<N> {
    fn cmp(&self, o: &Self) -> Ordering {
        // BinaryHeap is a max-heap: reverse every component so the smallest f pops first.
        o.f.total_cmp(&self.f).then(o.h.total_cmp(&self.h)).then(o.seq.cmp(&self.seq))
    }
}

/// BFS, Dijkstra, greedy, and (weighted) A* are this function with four `Priority` values.
/// Returns `None` when the open set empties -- the completeness half of Derivation 2.
pub fn best_first<S: SearchSpace>(
    space: &S, start: S::Node, goal: S::Node,
    h: impl Fn(S::Node) -> f64, prio: Priority,
) -> Option<Trace<S::Node>> {
    let mut g: HashMap<S::Node, f64> = HashMap::from([(start, 0.0)]);
    let mut depth: HashMap<S::Node, usize> = HashMap::from([(start, 0)]);
    let mut parent: HashMap<S::Node, S::Node> = HashMap::new();
    let mut closed: HashSet<S::Node> = HashSet::new();
    let mut open = BinaryHeap::new();
    let (mut seq, mut pops, mut reopened) = (0u64, Vec::new(), 0usize);
    open.push(Entry { f: prio.f(0.0, h(start), 0), h: h(start), seq, g: 0.0, node: start });
```

```
    while let Some(Entry { node, g: gn, .. }) = open.pop() {
        // Lazy deletion: a stale entry (better g pushed later, or already closed) costs nothing.
        if closed.contains(&node) || gn > g[&node] + 1e-12 { continue; }
        closed.insert(node);
        let d = depth[&node];
        pops.push(Pop { node, g: gn, h: h(node), f: prio.f(gn, h(node), d) });
        if node == goal {
            let mut path = vec![goal];
            while let Some(&p) = parent.get(path.last().unwrap()) { path.push(p); }
            path.reverse();
            return Some(Trace { pops, path, cost: gn, reopened });
        }
        for (x, c) in space.neighbors(node) {
            let tentative = gn + c;
            // Breadth-first relaxes on link length (first discovery wins); the rest on cost.
            let better = match prio {
                Priority::LinkLength => d + 1 < *depth.get(&x).unwrap_or(&usize::MAX),
                _ => tentative < *g.get(&x).unwrap_or(&f64::INFINITY) - 1e-12,
            };
            if !better { continue; }
            if closed.remove(&x) { reopened += 1; }       // only an inconsistent h gets here
            g.insert(x, tentative); depth.insert(x, d + 1); parent.insert(x, node);
            seq += 1;
            open.push(Entry { f: prio.f(tentative, h(x), d + 1), h: h(x), seq, g: tentative, node: x });
        }
    }
    None
}

pub fn astar<S: SearchSpace>(s: &S, a: S::Node, b: S::Node, h: impl Fn(S::Node) -> f64) -> Option<Trace<S::Node>> {
    best_first(s, a, b, h, Priority::Sum { eps: 1.0 })
}
pub fn dijkstra<S: SearchSpace>(s: &S, a: S::Node, b: S::Node) -> Option<Trace<S::Node>> {
    best_first(s, a, b, |_| 0.0, Priority::CostToCome)
}
```

D* Lite is the default replanner. The whole algorithm is the key, the two update cases, and the modifier k_m; everything else is bookkeeping.

RUST **crates/search/src/dstar_lite.rs**

```
/// Incremental replanner (Koenig & Likhachev 2002/2005): g/rhs tables plus the key
/// modifier k_m, so the robot's own motion never forces a heap rebuild.
pub struct DStarLite<S: SearchSpace> {
    space: S,
    g: HashMap<S::Node, f64>,
    rhs: HashMap<S::Node, f64>,
    queue: KeyedHeap<S::Node, [f64; 2]>,   // lazy; stale entries detected by key mismatch
    start: S::Node, last: S::Node, goal: S::Node,
    km: f64,
```

```rust
    h: fn(S::Node, S::Node) -> f64,
}

impl<S: SearchSpace> DStarLite<S> {
    fn key(&self, s: S::Node) -> [f64; 2] {
        let m = self.g[&s].min(self.rhs[&s]);
        [m + (self.h)(self.start, s) + self.km, m]
    }

    fn update_vertex(&mut self, u: S::Node) {
        if u != self.goal {
            let best = self.space.neighbors(u).map(|(v, c)| c + self.g[&v]).fold(f64::INFINITY, f64::min);
            self.rhs.insert(u, best);
        }
        self.queue.remove(u);
        if self.g[&u] != self.rhs[&u] { let k = self.key(u); self.queue.insert(u, k); }
    }

    pub fn compute_shortest_path(&mut self) -> usize {
        let mut pops = 0;
        // Process ties with the start's key as well: a stale vertex left at an equal key
        // can win a tie in the greedy descent and misreport the path's cost.
        while self.queue.top_key().is_some_and(|k| k <= self.key(self.start)) || self.rhs[&self.start] != self.g[&self.start] {
            let Some((u, k_old)) = self.queue.pop() else { break };
            pops += 1;
            let k_new = self.key(u);
            if k_old < k_new { self.queue.insert(u, k_new); }
            else if self.g[&u] > self.rhs[&u] {
                self.g.insert(u, self.rhs[&u]);                       // overconsistent: commit
                for (p, _) in self.space.neighbors(u) { self.update_vertex(p); }
            } else {
                self.g.insert(u, f64::INFINITY);                      // underconsistent: re-derive
                for (p, _) in self.space.neighbors(u) { self.update_vertex(p); }
                self.update_vertex(u);
            }
        }
        pops
    }

    /// Report observed cost changes; returns how many vertices' values moved (w6.2's statistic).
    pub fn update_edges(&mut self, changed: &[(S::Node, S::Node)]) -> usize {
        self.km += (self.h)(self.last, self.start);
        self.last = self.start;
        let mut touched = HashSet::new();
        for &(u, v) in changed { for s in [u, v] { self.update_vertex(s); touched.insert(s); } }
        self.compute_shortest_path();
        touched.len()
    }

    pub fn next_step(&mut self) -> Option<S::Node> {
        if self.g[&self.start].is_infinite() { return None; }
        let (s, _) = self.space.neighbors(self.start)
            .map(|(v, c)| (v, c + self.g[&v]))
            .min_by(|a, b| a.1.total_cmp(&b.1))?;
        self.start = s;
```

```
        Some(s)
    }
}
```

The universal plan is Dijkstra with the early exit removed and the direction reversed.

RUST **crates/search/src/policy.rs**

```rust
/// Choset sec.H.4: Dijkstra run backward from the goal yields cost-to-go and a successor
/// for *every* node -- a universal plan. The sister book's Chapter 21 generalizes `min`
   to E[.].
pub struct UniversalPlan<N> { pub cost_to_go: HashMap<N, f64>, pub next: HashMap<N, N> }

pub fn universal_plan<S: SearchSpace>(space: &S, goal: S::Node) -> UniversalPlan<S::Node>
    {
    let mut cost_to_go = HashMap::from([(goal, 0.0)]);
    let mut next = HashMap::from([(goal, goal)]);
    let mut closed = HashSet::new();
    let mut heap = BinaryHeap::new();
    heap.push(MinEntry { key: 0.0, node: goal });
    while let Some(MinEntry { key, node: x }) = heap.pop() {
        if !closed.insert(x) || key > cost_to_go[&x] + 1e-12 { continue; }
        for (y, c) in space.neighbors(x) {              // symmetric edges: neighbors =
   predecessors
            let cand = cost_to_go[&x] + c;
            if cand < *cost_to_go.get(&y).unwrap_or(&f64::INFINITY) - 1e-12 {
                cost_to_go.insert(y, cand);
                next.insert(y, x);
                heap.push(MinEntry { key: cand, node: y });
            }
        }
    }
    UniversalPlan { cost_to_go, next }
}

/// The deterministic Bellman residual: max |h(X) - min_Y [c(X,Y) + h(Y)]|, zero for a
   correct plan.
pub fn bellman_residual<S: SearchSpace>(space: &S, plan: &UniversalPlan<S::Node>) -> f64
   {
    plan.cost_to_go.iter().filter(|(x, _)| plan.next[x] != **x).map(|(&x, &hx)| {
        let best = space.neighbors(x).filter_map(|(y, c)| plan.cost_to_go.get(&y).map(|hy
   | c + hy)).fold(f64::INFINITY, f64::min);
        (hx - best).abs()
    }).fold(0.0, f64::max)
}
```

6.5.1 The worked example, and its printed output

RUST **crates/search/examples/astar_trace.rs**

```rust
fn main() {
    let raster = cspace::Raster::from_ascii(&["....", ".#..", ".#.."]);   // first row is
    the top
```

```rust
    let grid = Grid8 { raster: &raster, diag: 1.4 };
    let (start, goal) = ((0, 0), (3, 0));                                    // (col,row)
    = (1,1), (4,1)
    let h = |(i, j): (i32, i32)| { let (dx, dy) = ((3 - i).abs() as f64, (0 - j).abs() as
    f64); 1.4 * dx.min(dy) + (dx - dy).abs() };
    let t = astar(&grid, start, goal, h).expect("a path exists");
    println!("pop  node   g    h    f");
    for (k, p) in t.pops.iter().enumerate() {
        println!("{:>3}  ({},{}) {:.1}  {:.1}  {:.1}", k + 1, p.node.0 + 1, p.node.1 + 1,
    p.g, p.h, p.f);
    }
    println!("path: {}", t.path.iter().map(|(i, j)| format!("({},{})", i + 1, j + 1)).
   collect::<Vec<_>>().join(" "));
    println!("cost = {:.1}", t.cost);
    let d = dijkstra(&grid, start, goal).unwrap();
    println!("dijkstra: expansions = {}, cost = {:.1}", d.pops.len(), d.cost);
}
```

```
pop  node   g    h    f
  1  (1,1) 0.0  3.0  3.0
  2  (1,2) 1.0  3.4  4.4
  3  (2,3) 2.4  2.8  5.2
  4  (3,2) 3.8  1.4  5.2
  5  (4,1) 5.2  0.0  5.2
path: (1,1) (1,2) (2,3) (3,2) (4,1)
cost = 5.2
dijkstra: expansions = 10, cost = 5.2
```

```
A* returned cost 8 via A; optimal is 5 via B
initial path: (2,1) (3,2) (4,3) (5,4) (6,5) (7,6)  cost 7.0
gate (4,3) closed at (3,2): repaired k(3,2) = h(3,2) = 7.6 via (4,1)  --  15 states
   popped (5 raised) vs 32 for a fresh backward Dijkstra
```

`astar_trace_matches_text` asserts the pop order, the $g/h/f$ triples to 10^{-9}, the path, and both expansion counts. `astar_equals_dijkstra_on_random_grids` runs two hundred seeded `SmallRng` grids and asserts equal costs, zero reopenings, and equality with `pathfinding::prelude::astar` (the crate is a test dependency only; no search crate is in the library path). `nonoptimistic_loses` asserts 8 and 5. `dstar_gate_repairs_to_7_6` asserts the initial path, $h(4,1) = 6.2$, and the repaired $k(3,2) = 7.6$ with back pointer $(4,1)$; the work counts are regression-locked, as the design doc asks, because they depend on the choice to re-insert both endpoints of a changed arc. `dstar_lite_equals_fresh_astar` flips random cells on forty grids as the robot walks and asserts, after every flip and every step, that D *Lite's cost-to-goal equals a fresh A*'s and that its greedy descent has that cost. `universal_plan_is_bellman_fixed_point` asserts the residual is below 10^{-9} on the Apartment and that following the policy from sixty random cells reaches the goal at Dijkstra's cost. The TypeScript port in `web/lib/search/` runs the same eleven checks, and every widget on this page is that port.

> **NOTE**
>
> The D *Lite check caught a real bug worth recording. Koenig and Likhachev's loop stops when the top key is no longer* less than *the start's. A vertex whose key* ties* the start's — common on a grid, where keys are sums of 1s and $\sqrt{2}$s — can be left stale, and on one grid in five hundred the greedy descent stepped onto such a vertex and reported a cost g had never promised. Processing ties, with a tolerance for the ulp-level rounding of those sums, fixed it. The loop in the listing and in the port both say $\leq$.

6.6 Putting it together: a door closes

The integration lab is the Apartment scenario of the D *Replanner. Rasterize the Apartment at half-metre cells for disc-Rusty with the Chapter 4 inflation of* 0.25 *m, plan from room A to the bedroom with* `astar` *under Choset's* 1/1.4 *metric, and let Rusty drive. At the chosen step the cell on the plan about halfway along — in the Apartment that is almost always a doorway — becomes an obstacle, a door closing while Rusty is en route. When Rusty is adjacent it notices, and D* Lite repairs: it updates the cells around the door, propagates until the band of keys below Rusty's own is consistent again, and hands back a path through a different door. The blue footprint is the repair; the readout compares its size with the expansions a from-scratch A* needs from the same cell. In the open corridor the two are comparable, because the repair has to re-derive the whole region whose cost-to-go flowed through the closed door; deep inside a room, where a closed door changes only a few cells' routes, the repair is a tenth of the redo. Exercise 4 asks you to find the other kind of case.

Three pointers forward close the chapter. Chapter 7's brushfire and wave-front planners are breadth-first search from a different seed set — all obstacle cells, or the goal — and its navigation functions are universal plans with a smoothness requirement. Chapter 8 runs A* on the visibility graph and the generalized Voronoi diagram, and Chapter 10 on a cell decomposition's adjacency graph; none of them re-implement anything on this page. And the sister book's Chapter 21 takes the deterministic Bellman equation of the universal plan, replaces the minimum over successors by an expectation over outcomes, and arrives at value iteration — the same table, now a value function.

6.7 Exercises

> 1 **FOUNDATION** •• **Octile is consistent — for the right diagonal**
>
> Prove that $h(n) = 1.4\min(\Delta x, \Delta y) + |\Delta x - \Delta y|$ is consistent on the 8-connected grid with Choset's 1/1.4 metric: show $h(n) \leq c(n, x) + h(x)$ for each of the eight moves

and $h(q_{\text{goal}}) = 0$. Then give a two-cell example showing that the same formula is *not* admissible if diagonals cost $\sqrt{2}$ instead of 1.4.

> **HINTS**
>
> <>Consistency is one inequality per move type. An axial move changes one of Δx, Δy by one; check how much h can drop.</> <>A diagonal move can reduce both Δx and Δy by one, dropping h by exactly 1.4 — the cost of the move. An axial move drops h by at most 1.</> <>For inadmissibility with √2: start one diagonal step from the goal. h says 1.4; the true cost is $\sqrt{2} = 1.414\ldots$, so h < h*, fine. Now try two diagonal steps along the same diagonal, and compare 2.8 with $2\sqrt{2}$.</>

2 FOUNDATION •• **Obstacles as 10000-cost nodes**

Choset's grid A* (§H.2.4) keeps obstacle pixels in the graph as nodes that cost 10000 to enter. Show that on any grid with fewer than 10000/1.4 cells this returns the same path as deleting them. Then exhibit a grid where the two differ — and say which answer a robot should prefer.

3 CONCEPTUAL • **Predict the weighted-A* bound, then verify**

PREDICT FIRST

In the Search Theater set $\varepsilon = 2$. Before pressing Run: can the A* lane's path cost exceed Dijkstra's by more than a factor of 2 on any layout?

- Yes — once the heuristic is inadmissible, anything goes
  Inadmissible, but not arbitrary. Weighted A *with $\varepsilon \cdot h$ keeps the proof of Derivation 3 alive with one extra factor: $f(v) \le g(\mathrm{v}) + \varepsilon\ \mathrm{h}(v) \le \varepsilon\ c$*, so the goal is never popped above ε c*.
- ✓ No — the cost is at most 2 × Dijkstra's, and usually much closer
  Right. The ε-suboptimality bound $\mathrm{c} \le \varepsilon$ c* holds for every layout, and on grids the realized ratio is typically 1.0–1.2: the bound is loose, the loss is real but small, and the expansion savings are large.
- No — the cost never changes at all, only the expansion count does
  That is true for $\varepsilon \le 1$. Past 1 the heuristic overestimates, Derivation 5's failure mode applies, and the cost does drift — the badge marks the first ε at which it does.

Run ten seeds and record the ratio column. You should see values between 1.00 and about 1.2, never above 2. Then set $\varepsilon = 3$ and look for the layouts where the ratio gets largest: they are the ones where the heuristic's straight line toward the goal points into a room with no second door.

4 CONCEPTUAL •• **When repair costs more than redo**

In the D *Replanner, find a gate position — on Choset's grid or in the Apartment — for which D* Lite touches *more* cells than a from-scratch A* expands. Explain, using the raised/lowered classification, why this is rare: which cells must a repair touch, which cells must a redo touch, and when is the first set the larger?

WORK IT OUT

On Choset's gate grid, how many states does the D* repair pop after the gate closes with Rusty at (3,2)?

Answer. 15 Fifteen: the gate cell and its neighbors are re-inserted at their current h and popped as LOWER states, $(3, 2)$ and the cells behind it are popped as RAISE states, and the loop stops as soon as the smallest key reaches 7.6. A from-scratch A *from* $(3, 2)$ *expands only seven cells here, because the grid is tiny and the heuristic nearly exact; a from-scratch backward Dijkstra — what D* would have to redo to recover a plan for *every* cell — pops thirty-two.

5 PRACTICAL •• **A true FIFO**

Implement `Priority::LinkLength` as a true first-in-first-out queue — not a heap with constant keys — and add a property test that breadth-first search and Dijkstra agree on every unit-weight grid over 200 seeded trials: same link length, same cost. Then switch the weights to 1/1.4 and write the test that shows where they disagree, with the four-axial-versus-three-diagonal example as its smallest witness.

6 PRACTICAL ••• **ARA***

Implement anytime repairing A *in* `src/ara.rs` *with the schedule* $\varepsilon = 3, 2, 1.5, 1$, *reusing the open set between iterations and keeping the* inconsistent* list that the algorithm uses to seed the next round. Add a fifth lane to the Search Theater that shows each iterate's path, and assert in a test that the final cost equals Dijkstra's. The paper is Likhachev, Gordon and Thrun's; the design is three pages, and Derivation 4 is the part of this chapter you will need.

6.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

Appendices G and H are this chapter's source: the by-example pedagogy, Algorithm 24, the non-optimistic figure H.20, the D *gate walk-through and Algorithms 25–31, and the universal plans of §H.4. D* itself is Stentz's (Choset's reference [397], ICRA 1994). `link`

Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik 1, 269–271*.

The cost-to-come search; three pages. The priority rule f = g in this chapter's one loop. `doi:10.1007/BF01386390`

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics 4(2), 100–107*.

A*, with the admissibility condition and the optimality theorem that Derivation 3 restates; consistency appears there as the 'consistency assumption'. `doi:10.1109/TSSC.1968.300136`

Koenig, S. and Likhachev, M. (2005). Fast Replanning for Navigation in Unknown Terrain. *IEEE Transactions on Robotics 21(3), 354–363*.

D* Lite, the chapter's default replanner: g, rhs, the queue of inconsistent vertices, and the key modifier k_m. The AAAI 2002 paper of the same authors introduced it; this is the journal version. `doi:10.1109/TRO.2004.838026`

LaValle, S. M. (2006). Planning Algorithms. *Cambridge University Press*.
Chapter 2 presents discrete planning as the same forward-search template with a swappable priority queue, and is the best second reading for this chapter's 'one search, one knob'. link

Russell, S. and Norvig, P. (2020). Artificial Intelligence: A Modern Approach. *Pearson, 4th edition*.
Chapter 3 has the cleanest textbook statements of admissibility, consistency and the no-reopening property, with the proofs this chapter follows. link

Thrun, S., Burgard, W., and Fox, D. (2005). Probabilistic Robotics. *MIT Press*.
Where the universal plan goes next: value iteration over MDPs, which the sister volume's Chapter 21 derives as this chapter's backward Dijkstra with an expectation in place of the minimum. link

CHAPTER 7

Potential Functions

CLASSICAL PLANNERS — SEARCH, POTENTIALS, ROADMAPS, CELLS · Intermediate · 60 min

> *The potential function approach directs a robot as if it were a particle moving in a gradient vector field.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), Chapter 4*

Chapter 6 planned on a graph someone had already built. This chapter asks what a robot can do with nothing but a goal and a sense of distance: treat the goal as a valley, the obstacles as hills, and roll downhill. The construction is beautifully local. The gradient of the potential at q needs only the goal and the nearest obstacle, so Rusty can compute it from a range scan and Reach from a few points on its links — no configuration space is ever built.

The "aha" comes in two halves. The first is a failure: a sum of one valley and some hills can have a second valley that is not the goal, and gradient descent rolls into it and stops. The widget that opens the chapter shows a bead doing exactly that in the mouth of a horseshoe, and then — this is the part people get wrong — between two convex discs. The disease is in the *sum*, not in concavity. The second half is the cure, and it is not a tweak to the hills. A *distance-to-goal* field has exactly one minimum, and on a grid the breadth-first search the reader already owns computes it: wave-front is BFS rooted at the goal, brushfire is BFS rooted at the obstacles. Navigation functions show that the single minimum can even be had *smoothly*, on a restricted family of worlds, so the local-minimum problem is a fact about additive potentials, not about potentials.

Everything here is incomplete except the wave-front planner, which is resolution complete and exponential in dimension; the chapter says so each time. What survives is the *field* idea: a scalar function whose gradient the robot follows. Chapter 8 collides brushfire fronts on the generalized Voronoi diagram, Chapter 19's CHOMP descends a distance field, and Chapter 23's reactive layer is this

chapter running at a hundred hertz.

AFTER THIS CHAPTER YOU CAN

- Write the attractive/repulsive potential of Choset §4.1 with its two honesty items — the blended attractive term whose gradient is continuous and bounded, and the per-obstacle repulsive sum whose gradient does not jump — and run gradient descent on it
- Compute the distance a repulsive term needs three ways: from geometry, from the local minima of a range scan, and from a brushfire grid; say what each one gets wrong
- Prove that brushfire and wave-front are the breadth-first search of Chapter 6 with a different seed set, and that the wave-front descent is a shortest grid path with exactly one minimum
- Explain why additive potentials have spurious minima — with two convex discs, no concavity required — and classify a stuck point by the eigenvalues of the Hessian
- State what a navigation function is, read Rimon and Koditschek's sphere-world construction, count its critical points as κ grows, and carry it to star worlds through a diffeomorphism
- Lift a workspace force at a control point to a configuration-space torque with J^T, sum the torques in configuration space, and say why repelling a few points does not keep a body safe

ASSUMED

- Rusty, Reach, the Apartment and the Workbench, and the distance queries of the collide module (Chapter 2)
- The raw distance function ρ, D(x) and ∇D from a range scan (Chapter 3)
- C-obstacles, the torus of a two-joint arm, and Reach's Jacobian (Chapter 4)
- Breadth-first search on a grid graph (Chapter 6)

7.1 The problem: roll downhill, and ask who built the hill

Put a bead at the left edge of a table and a goal at the right. Define a scalar field U that is small at the goal and large near obstacles, and let the bead move with velocity $-\nabla U$. With a single disc between them, the bead swerves around the disc and arrives. Replace the disc by a horseshoe whose mouth faces the bead, and it rolls into the mouth and stops.

INTERACTIVE · w7.1

Potential Well

Local minima are a property of sums, not of concave obstacles — two convex discs suffice. A cost-to-go field has exactly one minimum.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch07-potential-functions#w7.1

Nothing is broken. The badge under the canvas reports what the mathematics says: the gradient has vanished at a point that is not the goal, and both eigenvalues of the Hessian there are positive. The bead sits at the bottom of a genuine bowl. The goal's pull and the base's push cancel exactly on the axis of the horseshoe, and the two arms push back toward the axis whenever the bead drifts off it, so the point is stable in every direction. Choset's figure 4.10 draws this and his figure 4.11 makes the sharper point: switch the widget to *two convex discs* and the bead still stops, between them. Concavity has nothing to do with it.

Now switch on **wave-front mode**. The heightfield changes to the breadth-first cost-to-goal of Chapter 6, computed on a tenth-of-a-metre raster of the same obstacles, and the bead escapes along a grid-shortest path. That field has one minimum, provably, because it *is* distance to the goal. The rest of the chapter is the difference between those two heightfields.

7.2 Building intuition

7.2.1 Why a sum has extra valleys

The additive potential is $U = U_{att} + U_{rep}$: a bowl centred at the goal plus a spike on each obstacle. Far from any obstacle only the bowl matters and the bead heads for the goal. Near an obstacle the spike takes over and pushes the bead off. In between, the two gradients can cancel, and the only question is whether the cancellation point is *stable*. Between two obstacles symmetric about the line to the goal it is: the repulsive push is purely axial on the axis by symmetry, it grows without bound toward the gap while the attractive pull stays bounded, so somewhere on the axis the two cancel; and off the axis both spikes push back toward it. Derivation 3 writes this out. The Potential Well's $Q^\star$ slider changes where the cancellation happens but cannot make it go away; the ζ and η sliders in the disclosure move it along the axis. Only the obstacle layout can remove a minimum, and the planner does not control the layout.

THE ONE SENTENCE VERSION

An additive potential is a valley plus hills; gradient descent finds *a* valley, and nothing in the sum promises it is the one with the goal in it. A

distance-to-goal field is a single valley by construction — and on a grid it is one breadth-first search.

7.2.2 Fronts on a grid

The grid cure is older than the word "potential". Label every obstacle pixel 1, its free neighbours 2, their unlabelled neighbours 3, and so on: a fire spreading outward from every obstacle at once. When it is done, each pixel's label is one more than its link length to the nearest obstacle. That is **brushfire**, Choset §4.3.2, and it is a multi-source breadth-first search.

INTERACTIVE · w7.4

Brushfire Fronts

Brushfire is breadth-first search from every obstacle pixel; it computes a grid metric, not Euclidean distance; where two fronts meet is the grid GVD.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch07-potential-functions#w7.4

Two things to watch. First, the *shape* of a front: under 4-point connectivity the fronts are diamonds, because the label counts Manhattan steps; under 8-point they are squares, because the label counts chessboard steps. Neither is a circle. Brushfire computes a grid metric, and the hover readout compares it with the exact Euclidean distance transform ported from the sister book: the two agree only along the axes. Second, the solid pixels where two fronts meet. On the micro-grid that is column 4, equidistant from both obstacles; each of its three pixels was reached by both fronts and keeps *two back pointers*. Chapter 8 will call that column the grid generalized Voronoi diagram. Seed the same search from the goal pixel instead, labelled 2, and you have the **wave-front** planner of §4.5: a pixel's label is 2+ its grid distance to the goal, and "step to any neighbour labelled one less" is a shortest grid path.

7.2.3 One minimum, smoothly

Grids are exponential in dimension. Is the single minimum available without one? Rimon and Koditschek's answer is yes, on a *sphere world* — a disc with disjoint disc obstacles — and on anything diffeomorphic to one.

INTERACTIVE · w7.2

Navigation Function Sculptor

Potentials do not have to have spurious minima: on a sphere world ϕ has exactly one for large enough κ, and a diffeomorphism carries that onto star worlds.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch07-potential-functions#w7.2

The field is $\varphi = d^2/[d^{2\kappa} + \beta]^{1/\kappa}$, where d is distance to the goal and β is a product that vanishes on every obstacle boundary. At $\kappa = 2$ the Sculptor finds three minima; at $\kappa = 3$ two; from $\kappa = 7$ exactly one. Slide κ and watch the spurious minima slide toward their obstacles and turn into saddles. This is the point the widget exists to make: a potential with no spurious minima *exists*, and it is smooth. The price is the family of worlds it works on and a sharpening exponent whose threshold the theorem only promises exists. Star mode shows the trick that widens the family: a map h_λ that squashes each star-shaped obstacle onto a disc, with $\varphi \circ h_\lambda$ inheriting the single minimum because critical points correspond one-to-one under a diffeomorphism.

7.2.4 A body is not a point

Reach's configuration space is a torus; there is no "goal point" in the workspace to put a bowl at. Choset's §4.7 writes the potential in the *workspace*, at a few control points on the arm, and lifts each workspace force to a joint torque with the transposed Jacobian of that point.

INTERACTIVE · w7.3

Arm in a Field

$u = J^T f$: workspace forces at control points become joint torques summed in configuration space — and repelling a few fixed points does not keep the body safe.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch07-potential-functions#w7.3

Two red control points on the end effector pull it to the goal pose; slate points on the links push away from the block, the post and the shelf. The bars under the canvases are the lifted torques $J_j^\top f_j$, one row per control point, and their sum — the gradient the arm descends. Watch the default run twice. Without r_{float} link 1 drives straight through the post *between* its two repulsive points, which are far enough from the post that their push is weak; with r_{float} — a floating control point at whichever link point is nearest any obstacle — the same start reaches the goal. Repelling the vertices of a body does not protect its edges (Choset's figure 4.20); r_{float} reduces the risk and does not eliminate it, and the widget's alternate mode lets you find starts where even that is not enough.

7.3 The mathematics

U, U_{att}, U_{rep}, ∇U	potential, its attractive and repulsive parts, and the gradient the robot reads as a velocity book-wide
ζ, η	attractive and repulsive gains
$d^{\star}_{goal}$	radius where the attractive potential switches from quadratic to conic
$Q^{\star}$, $Q^{\star}_i$	repulsive distance of influence — global, or per obstacle
$d_i(q)$, $D(q)$, c	distance to QO_i, to the nearest obstacle, and the closest point on QO_i, so ∇d_i = (q − c)/d_i book-wide
$\alpha(i)$, ϵ	descent step size at iteration i; the gradient tolerance in $\|\nabla U\| < \varepsilon$
$\nabla^2 U$	Hessian; nondegenerate critical point ⇔ nonsingular Hessian
$\beta_i(q)$, $\beta(q)$	obstacle functions (< 0 inside, 0 on ∂QO_i, > 0 outside); their product
γ_κ, σ_λ, ξ_κ	goal term d^{2κ}; analytic switch x/(λ + x); sharpening x^{1/κ}
$\varphi(q)$	navigation function (Choset Def. 4.6.1), ϕ = ξ_κ ∘ σ_1 ∘ γ_κ/β
h_λ, T_i, ν_i, s_i	star-to-sphere diffeomorphism; translated scaling maps; their scale factors; analytic switches
$r_j(q)$, $J_j(q)$, r_{float}	control point j and its Jacobian, u = J_j^T f; the floating control point nearest any obstacle

7.3.1 Potentials, critical points, and the Hessian

DEFINITIONS — POTENTIAL FUNCTION; GRADIENT DESCENT PATH (CHOSET CH. 4 INTRO)

A **potential function** is a differentiable $U : \mathbb{R}^m \to \mathbb{R}$. Its gradient $\nabla U(q) = [\partial U/\partial q_1, \ldots, \partial U/\partial q_m]^\top$ points in the direction of locally steepest increase. For the first-order robots of §4.1–4.6 the gradient is read as a **velocity**; in §4.7 it is a **force**. The robot follows $\dot{c} = -\nabla U(c)$ — **gradient descent** — and stops at a $q^\star$ with $\nabla U(q^\star) = 0$, a **critical point**.

DEFINITIONS — CRITICAL POINT TYPE; NONDEGENERACY

At a critical point the **Hessian** $\nabla^2 U$ decides the type: positive definite is a **minimum**, negative definite a **maximum**, indefinite a **saddle**. The point is **nondegenerate** if the Hessian is nonsingular, in which case it is isolated. Gradient descent generically ends only at a minimum: a maximum or a saddle is unstable under any perturbation, a minimum is stable under every one. The full Morse vocabulary — index, level-set topology — is Chapter 8's definition box; this chapter needs only "nondegenerate", which is what Choset means when he says a navigation function "is Morse".

In the colours every figure on this page uses, the descent is

$$\dot{c}(t) \;=\; -\nabla U(c(t)), \qquad U \;=\; \tfrac{1}{2}\zeta\, d^2(q, q_{\text{goal}}) \;+\; \sum_i \tfrac{1}{2}\eta\left(\tfrac{1}{d_i(q)} - \tfrac{1}{Q_i^\star}\right)^2.$$

The orange bead is $c(t)$, the red goal sits at the bottom of the bowl, and every slate obstacle contributes one spike through its distance d_i. Point at a term and the matching layer in the Potential Well comes forward.

DEFINITION — DISTANCE TO AN OBSTACLE (CHOSET EQ. 4.6)

$d_i(q) = \min_{c \in QO_i} d(q, c)$ is the distance from q to C-obstacle i, and $D(q) = \min_i d_i(q)$ the distance to the nearest one. For a point robot in the plane these are the Chapter 2 and Chapter 3 quantities; for Reach they are workspace distances from control points (§4.7).

DERIVATION **The blended attractive potential has a continuous, bounded gradient**

| \nabla U_{att} | \le \zeta d^\star_{goal},\ \text{continuous at } d = d^\star_{goal}

Statement (Choset eqs. 4.2–4.3). With

$$U_{att}(q) = \begin{cases} \tfrac{1}{2}\zeta\, d^2(q, q_{\text{goal}}), & d \le d^\star_{goal}, \\ d^\star_{goal}\,\zeta\, d(q, q_{\text{goal}}) - \tfrac{1}{2}\zeta (d^\star_{goal})^2, & d > d^\star_{goal}, \end{cases}$$

∇U_{att} is continuous and its magnitude never exceeds $\zeta d^\star_{goal}$.

Step 1 — inside. $\nabla \tfrac{1}{2}\zeta d^2 = \zeta(q - q_{\text{goal}})$, of magnitude $\zeta d \le \zeta d^\star_{goal}$, vanishing linearly at the goal (eq. 4.1).

Step 2 — outside. $\nabla[d^\star_{goal}\zeta d] = d^\star_{goal}\zeta\,(q - q_{\text{goal}})/d$, of magnitude exactly $\zeta d^\star_{goal}$ everywhere.

Step 3 — the seam. At $d = d^\star_{goal}$ both expressions equal $\zeta(q - q_{\text{goal}})$; the constant $-\tfrac{1}{2}\zeta(d^\star_{goal})^2$ makes the *values* agree too, so U_{att} is C^1. ■

Why blend at all. The pure conic potential ζd has gradient of magnitude ζ everywhere and no direction at the goal — descent chatters around it. The pure quadratic vanishes nicely at the goal but asks for a velocity proportional to distance, which a far start turns into an absurd one. The blend takes each where it is good. The chapter's check pushes two hundred random seams and finds a gradient jump of 10^{-8} (the finite width of the test) and a bound excess of 10^{-16}.

DERIVATION **Gradient of distance to a convex obstacle**

$\nabla d_i(q) = (q - c)/\|q - c\|$

Statement (Choset eq. 4.7). If c is the unique closest point of a convex QO_i to $q \notin QO_i$, then $\nabla d_i(q) = (q - c)/\|q - c\|$, a unit vector pointing away from the obstacle.
Step 1 — uniqueness. $\|q - c\|$ is strictly convex in c and QO_i is convex and closed, so the minimiser c is unique.
Step 2 — the envelope theorem. $d_i(q) = \min_c \|q - c\|$; at a unique minimiser the derivative of the minimum with respect to q is the partial derivative of the objective with c held fixed.
Step 3 — differentiate. $\partial_q \|q - c\| = (q - c)/\|q - c\|$.
Step 4 — where convexity enters. A nonconvex obstacle can have *two* closest points, and d_i is then not differentiable: the gradient jumps as q crosses the set where they tie. Choset's §4.1 warns that a repulsive term written in $D(q)$ oscillates where the nearest obstacle changes; a nonconvex obstacle does the same thing to d_i *inside one term*. ■
The two cures. Sum the repulsion per convex obstacle, $U_{rep} = \sum_i U_{rep_i}$, and decompose nonconvex obstacles into convex pieces. The Potential Well's horseshoe is three rectangles for exactly this reason; the disclosure toggle gives you the single nonconvex polygon instead, and the bead chatters on its axis — the badge says "stalled" with a gradient of order one, because ∇d flips sign every step. The check compares the analytic ∇U_{rep} over discs, a polygon and a wall segment with central differences at 352 random points: worst relative error 6×10^{-10}. Choset's §4.3.1 reading of d_i as a *local minimum of $\rho(q, \theta)$ over the bearing* is the sensor version of this derivation, and it is where the Apartment's thin walls expose a limit — see the algorithm section.

7.3.2 The local-minimum disease

DERIVATION **Additive potentials admit non-goal minima — two convex discs suffice**

$\nabla U(q^\star) = 0,\ \nabla^2 U(q^\star) \succ 0,\ q^\star \neq q_{\text{goal}}$

Statement (Choset §4.4, figures 4.10–4.11). For two disc obstacles placed symmetrically about the line from the start to the goal, there is a point on that line with $\nabla U = 0$ and positive-definite Hessian.
Step 1 — symmetry. On the axis of symmetry the two repulsive gradients

are mirror images, so their sum is axial. The attractive gradient is axial too. So ∇U is axial on the axis.

Step 2 — unbounded push, bounded pull. As the point on the axis moves toward the gap, each d_i falls toward its minimum and $\eta(1/d_i - 1/Q^\star)/d_i^2$ grows without bound if the gap is narrower than $2Q^\star$; the attractive pull is bounded by $\zeta d^\star_{goal}$ (Derivation 1). Far from the discs, beyond $Q^\star$, the push is zero and the pull is toward the goal.

Step 3 — a zero. The axial component of ∇U is continuous, negative (goalward) far from the gap and positive (away) close to it; the intermediate value theorem gives $q^\star$ with $\nabla U(q^\star) = 0$.

Step 4 — transversal stability. Off the axis, the nearer disc pushes harder than the farther one and both push toward the axis; so the transverse second derivative is positive. The axial second derivative is positive because the push grows as the gap is approached. The Hessian is positive definite: $q^\star$ is a minimum. ■

The numbers. Discs of radius 0.5 at $(1, \pm 0.8)$, goal $(4, 0)$, $\zeta = \eta = 1$, $Q^\star = 1.5$, $\alpha = 0.01$. From $(-2, 0.05)$ the descent stops after **160** steps at $q^\star = (0.2409, 0.0000)$ with Hessian eigenvalues **14.15** and **16.16**, both positive. The chapter's check pins all of it. The horseshoe case is the same argument with the base supplying the push; Barraquand and Latombe's RPP escapes such minima with random walks and tries again, which is the honest fix of 1991 and the ancestor of Part III's sampling. The method is not complete.

7.3.3 Brushfire and wave-front are breadth-first search

DERIVATION **Brushfire computes grid distance; wave-front computes grid cost-to-goal; both are BFS**

```
\text{label} = 1 + \ell(\text{nearest obstacle}),\quad 2 + \ell(\text{goal})
```

Statement (Choset §4.3.2, §4.5; Chapter 6 Derivation 1). Labelling obstacle pixels 1, their free neighbours 2, theirs 3, ... assigns each free pixel 1+ its link length to the nearest obstacle — Manhattan distance under 4-point connectivity, chessboard under 8-point. Seeding the goal pixel with 2 instead assigns 2+ link length to the goal, and "step to a neighbour labelled one less" is a shortest grid path.

Step 1 — it is a BFS. Brushfire is breadth-first search on Chapter 6's grid graph with *every obstacle pixel in the initial queue at depth* 0; wave-front is the same search with the goal alone at depth 0. Labels are depth plus the seed label.

Step 2 — depth is link length. Chapter 6, Derivation 1: a node's depth when first dequeued is its shortest link length from the seed set. On a unit-cost 4-connected grid that is the Manhattan distance; on an 8-connected one, the chessboard distance.
Step 3 — descent reaches the goal. On a bounded grid every reachable free pixel receives a label. A non-goal pixel labelled $k > 2$ was labelled from a neighbour labelled $k - 1$, so such a neighbour exists; stepping to it strictly decreases the label and the walk ends at the goal after exactly $k - 2$ steps. The field has one minimum. ■
Two fronts meeting. A pixel reached by fronts from two different obstacles keeps two back pointers (Choset §5.2.5); the set of such pixels is the grid GVD of Chapter 8, and the micro-grid's column 4 is its smallest instance. On an even gap the collision lands on a pixel; on an odd gap it lands on the edge between two pixels, and the port flags both so the ridge has no holes. *Grid versus Euclidean.* On forty random grids totalling 7020 free cells the 4-point and 8-point labels equal the exact Manhattan and chessboard distances cell for cell, the exact Euclidean distance lies between them, 99.5% of cells are within one cell of it, and the worst gap is 2.789 cells — along a diagonal, where Manhattan overshoots by $\sqrt{2}$. The exact Euclidean distance transform is a different algorithm, ported from the sister book's map chapter as `mapping/edt`, and Exercise 5 asks you to run the two side by side. *Dimension.* Pixels become voxels, 4-point becomes 6-point, 8-point becomes 26-point, and the cell count is exponential in the dimension of Q — which is why Part III exists.

7.3.4 Navigation functions

DEFINITION 4.6.1 — NAVIGATION FUNCTION

A function $\varphi : Q_{\text{free}} \to [0, 1]$ is a **navigation function** if it is smooth (or at least C^2), has a unique minimum at q_{goal} in the connected component of Q_{free} containing it, is uniformly maximal on ∂Q_{free}, and is Morse (all critical points nondegenerate). Gradient descent on a navigation function reaches the goal from every start except a set of measure zero — the saddles' stable manifolds.

DEFINITIONS — SPHERE WORLD; STAR-SHAPED SET (CHOSET §4.6.1–4.6.2)

A **sphere world** is a disc of radius r_0 with disjoint disc obstacles inside it; its boundary functions are $\beta_0 = r_0^2 - d^2(q, q_0)$ and $\beta_i = d^2(q, q_i) - r_i^2$. A set is **star-shaped** if some point of it sees every other point along a segment

inside the set; every convex set is star-shaped, and a star-shaped obstacle has a centre q_i from which the boundary is a function $\rho(\theta)$ of the bearing.

DERIVATION **The sphere-world navigation function**

$$\varphi(q) = \frac{d^2(q, q_{\text{goal}})}{\left[d^{2\kappa}(q, q_{\text{goal}}) + \beta(q)\right]^{1/\kappa}}$$

Statement (Choset §4.6.1, eqs. 4.8–4.13; Rimon and Koditschek). On a sphere world, $\varphi = \xi_\kappa \circ \sigma_1 \circ (\gamma_\kappa/\beta)$ is a navigation function for sufficiently large κ, where $\gamma_\kappa = d^{2\kappa}(q, q_{\text{goal}})$, $\beta = \prod_{i=0}^{n} \beta_i$, $\sigma_\lambda(x) = x/(\lambda + x)$ and $\xi_\kappa(x) = x^{1/\kappa}$.

Step 1 — the raw ratio. γ_κ/β is zero only at the goal and $+\infty$ on every obstacle boundary, where some $\beta_i = 0$. It is positive in between.

Step 2 — large κ pushes the goal term ahead. Away from obstacles, $\partial\gamma_\kappa/\partial q$ dominates $\partial\beta/\partial q$ once κ is large, so the gradient of the ratio points goalward and no critical point survives there.

Step 3 — near an obstacle, only it matters. The ratio is then γ_κ/β_i times a slowly varying factor, and a critical point can lie only on the radial line from the obstacle to the goal; on that line the ratio *decreases* from the boundary toward the goal, so the Hessian cannot be positive definite. Spurious minima become saddles (Choset cites [239] for the proof).

Step 4 — squash. σ_1 is strictly increasing, maps $[0, \infty)$ onto $[0, 1)$, and so moves no critical point and changes no type; it bounds the function.

Step 5 — sharpen. ξ_κ is strictly increasing on $(0, \infty)$ and makes the critical points nondegenerate (Morse); composing gives eq. (4.13). ■

Counting instead of eyeballing. Because σ_1, ξ_κ and log are strictly increasing, the critical points of φ off the goal are those of $\psi = \log(\gamma_\kappa/\beta) = 2\kappa \log d - \sum_i \log \beta_i$, whose gradient is of order one where φ's is 10^{-8} — the **plateau problem** Choset's last paragraph of §4.6.1 admits: φ is flat near the goal and far from it and steep in between, which makes gradient descent on it numerically miserable. The Sculptor's beads descend the *normalized* gradient for that reason. The port finds critical points by damped Newton on $\nabla\psi$ from a lattice of seeds and classifies them by the Hessian of ψ. On the five-disc world the sweep reads:

κ	**2**	**3**	**4**	**6**	**7**	**8**	**10**
minima	3	2	2	2	**1**	1	1
saddles	7	6	6	6	5	5	5

Choset's own five-disc figure 4.15 has three minima at $\kappa = 3$ and one at $\kappa = 10$; our discs sit elsewhere and lose their last spurious minimum at $\kappa = 7$. In every column minima − saddles = −4: on a disc with five holes the Euler characteristic is $1 - 5 = -4$, and Morse theory makes that an invariant of the count — each spurious minimum that disappears takes exactly one saddle with it. The chapter's check pins the whole table. Nothing in the theorem says *which* κ suffices; it is existential, and in practice one sweeps.

DERIVATION **Navigation functions pull back under diffeomorphisms; the star-to-sphere map**

$$h_\lambda = s_{q_{\text{goal}}}\,\mathrm{id} + \sum_{i=0}^{n} s_i\, T_i, \qquad \varphi \circ h_\lambda \text{ is a navigation function}$$

Statement (Choset §4.6.2, eqs. 4.14–4.19). If φ is a navigation function on M and $h : F \to M$ is a diffeomorphism — smooth, bijective, smooth inverse — then $\varphi \circ h$ is a navigation function on F. For a star world F the map

$$T_i(q) = v_i(q)\,(q - q_i) + p_i, \qquad v_i(q) = \frac{(1 + \beta_i(q))^{1/2}\, r_i}{d(q, q_i)},$$

$$s_i(q, \lambda) = \frac{\gamma_\kappa \bar\beta_i}{\gamma_\kappa \bar\beta_i + \lambda \beta_i}, \quad \bar\beta_i = \prod_{j \neq i} \beta_j, \quad s_{q_{\text{goal}}} = 1 - \sum_{i=0}^{n} s_i, \quad h_\lambda(q) = s_{q_{\text{goal}}}(q)\, q + \sum_{i=0}^{n} s_i(q)\, T_i(q)$$

is such a diffeomorphism onto the model sphere world for suitable λ.

Step 1 — critical points correspond. $\nabla(\varphi \circ h) = J_h^{\top}\,(\nabla\varphi \circ h)$, and J_h is invertible, so $\nabla(\varphi \circ h)(q) = 0$ iff $\nabla\varphi(h(q)) = 0$. The Hessians are congruent at critical points, so nondegeneracy and type survive. Minima map to minima: one on M, one on F.

Step 2 — each T_i maps its star boundary onto a circle. On ∂QO_i, $\beta_i = 0$, so $T_i(q) = r_i(q - q_i)/d(q, q_i) + p_i$: the point at bearing θ from q_i lands at distance r_i from p_i at the same bearing. A star boundary of radius $\rho(\theta)$ is squashed radially by $r_i/\rho(\theta)$.

Step 3 — the switches switch. On ∂QO_i, $\beta_i = 0$ gives $s_i = 1$ and every $\bar\beta_j$ with $j \neq i$ contains the factor $\beta_i = 0$, so $s_j = 0$ and $s_{q_{\text{goal}}} = 0$. Hence $h_\lambda = T_i$ exactly on ∂QO_i. At the goal $\gamma_\kappa = 0$, every $s_i = 0$, and $h_\lambda(q_{\text{goal}}) = q_{\text{goal}}$.

Step 4 — suitable λ. Smoothness and bijectivity for some λ beyond a threshold are quoted from Rimon and Koditschek. The port measures the local half: $\det J_{h_\lambda} > 0$ on a 48×48 lattice of free points. ■

What the port had to learn. Three things about this construction are easy to get wrong, and the chapter's check caught all three. First, $\gamma_\kappa \bar\beta_i$ is

astronomically larger than $\lambda\beta_i$ for any λ of order one — on our world $\lambda_0 \approx 2 \times 10^{18}$ at $\kappa = 6$ — so a $\lambda = 1$ pins every switch at one *and* turns a rounding error of -10^{-16} in a boundary β_i into garbage for the other switches; the boundary is now handled exactly. Second, the model disc must be *inscribed* in its star: if a disc poked outside its star, $T_i - \mathrm{id}$ would point outward in the star's valleys while ∇s_i points inward, and $\det J_h$ went negative in the switch collar for every large λ we tried. With inscribed discs, $\min \det J_h = 0.984$ at λ_0 over 1450 lattice points. Third, "suitable" is not decoration: at $10^{-6}\lambda_0$ the minimum determinant is -55 and the map folds. The Sculptor's λ slider spans both regimes. *Virtual work*, the other half of this derivation's collapsible, follows in §4.7 below.

7.3.5 Lifting forces to configuration space

DEFINITION — CONTROL POINT (CHOSET §4.7.2)

A **control point** r_j is a point fixed on the robot whose workspace potential is lifted to a configuration-space force by $J_j^\top$, where $J_j = \partial r_j / \partial q$. Enough control points must be chosen to *pin the body down* — two for a rigid body in the plane. Attractive points carry an $U_{att,j}$ toward $r_j(q_{\text{goal}})$; repulsive points carry $U_{rep_i,j}$ for each workspace obstacle $\mathcal{WO}_i$ (eqs. 4.24–4.25). The **floating control point** r_{float} is the point of the robot's boundary nearest any obstacle, recomputed at every q.

DERIVATION **Virtual work: a workspace force is a configuration-space force through $J^\top$**

$$u = J^\top f, \qquad u(q) = \sum_j J_j^\top(q)\, f_j$$

Statement (Choset §4.7.1, eq. 4.26). Power is coordinate-independent: $f^\top \dot{x} = u^\top \dot{q}$ for every $\dot{q}$. With $\dot{x} = J\dot{q}$, $f^\top J\dot{q} = u^\top \dot{q}$ for all $\dot{q}$, so $u = J^\top f$. The lifted forces of several control points are **summed in configuration space**, never in the workspace.

Example 4.7.1. A vertex $a = (a_x, a_y)$ of a planar rigid body at $q = (x, y, \theta)$ sits at $\varphi(q) = (x + a_x \cos\theta - a_y \sin\theta,\ y + a_x \sin\theta + a_y \cos\theta)$ (eq. 4.20), with Jacobian

$$J = \begin{bmatrix} 1 & 0 & -a_x \sin\theta - a_y \cos\theta \\ 0 & 1 & a_x \cos\theta - a_y \sin\theta \end{bmatrix}$$

(eq. 4.21), so $u = J^\top f = (f_x,\ f_y,\ -f_x(a_x \sin\theta + a_y \cos\theta) + f_y(a_x \cos\theta - a_y \sin\theta))$ (eq. 4.23), and $u_\theta = (r \times f)_z$ with r the world vector from the body origin to the vertex. The check verifies this identity at a hundred random configurations to 10^{-12} against Chapter 4's `bodyPointJacobian`.
Reach. A point at fraction t along link k is the tip of a shorter arm, so its Jacobian is Chapter 4's `jacobian` with the columns for joints beyond k zero and the last link shortened to tL_k; at $t = 1$ on the last link it *is* `jacobian`, and the check says so. Then $u(q) = \sum_j J_j^\top(q)\,\nabla U_j(r_j(q))$ is the exact gradient of $\sum_j U_j(r_j(q))$ by the chain rule — including the floating point, by the envelope theorem of Derivation 2 — and the check compares it with central differences at 213 configurations: worst relative error 2×10^{-9}. ■
Why summing in the workspace is wrong. Two control points have two Jacobians; $J_1^\top f_1 + J_2^\top f_2 \neq J^\top(f_1 + f_2)$ for any single J. Control points approximate the body; they are the finite sample that stands in for an integral over its area, and figure 4.20's collision between them is the price. r_{float} reduces the price and does not eliminate it.

7.4 The algorithm

Choset's Algorithm 4 is the whole method; brushfire and wave-front are Chapter 6's breadth-first search with a different seed set; navigation functions are a formula; and lifting is a sum.

ALGORITHM GRADIENT DESCENT -- Choset Alg. 4, with the sign the text intends COST O(number of potential terms) per step; the number of steps depends on α and the field

in a means to compute ∇U(q) at a point q; a step schedule α(i); a tolerance ε
out the iterates {q(0), q(1), . . . , q(i)}, and why the loop stopped

1: $q(0) \leftarrow q_{\text{start}}$; $i \leftarrow 0$
2: **while** $\|\nabla U(q(i))\| \geq \epsilon$ **do**
3: $\quad q(i+1) \leftarrow q(i) - \alpha(i)\,\nabla U(q(i))$
4: $\quad i \leftarrow i + 1$
5: **end while**
6: **report**: at the goal; or a local minimum (print the Hessian's eigenvalue signs); or a step limit

Honesty. Line 4 of Algorithm 4 as printed reads $q(i+1) = q(i) + \alpha(i)\nabla U(q(i))$, while the text says "take a small step in the direction opposite the gradient" and $\dot{c} = -\nabla U$. It is a sign typo; the code uses the minus. The port adds

a cap on the step length (a repulsive spike must not fling the bead across the table), a collision test (so an α too large for the spike is reported, not ignored), and a *stall* detector for the chatter of Derivation 2's nonsmooth seam, where $\|\nabla U\|$ never falls below ϵ and the bead goes nowhere.

ALGORITHM `BRUSHFIRE -- Choset §4.3.2 · WAVE-FRONT -- Choset §4.5` COST `O(#cells): one multi-source breadth-first search`

in an occupancy grid (0 free, 1 obstacle); 4- or 8-point connectivity; for wave-front, the goal pixel
out a label per pixel (1 + grid distance to the nearest obstacle, or 2 + grid distance to the goal); back pointers; the pixels where two fronts met

1: **Brushfire:** label every obstacle pixel 1; enqueue every free pixel adjacent to an obstacle with label 2 and the obstacle's component as its source
2: **Wave-front:** enqueue the goal pixel with label 2
3: **while** the queue is not empty: pop c; for each free neighbour x of c (Chapter 6's `GridGraph.neighbors`):
4: if x is unlabelled: label it label(c) + 1, point its back pointer at c, inherit c's source, enqueue it
5: else if (brushfire) x's source differs from c's and label(x) ≥ label(c): record a second back pointer — two fronts met
6: **Grid gradient** at a pixel: a neighbour with the lowest label (ties to the first in scan order E, W, N, S, then diagonals)
7: **Wave-front descent** from q_{start}: step to a neighbour labelled one less until the label is 2

Reading distance off the grid: (label − 1) · cellSize is the grid distance, and the gradient points from the back-pointer pixel to the robot's — four or eight possible directions, "just as the grid is an approximation of the workspace, so is the computed gradient an approximation of the actual distance gradient" (Choset). The micro-example's gradient at $(5, 3)$ points to $(6, 3)$: both $(6, 3)$ and $(5, 2)$ carry label 3 and east comes first.

ALGORITHM `SENSOR-BASED DISTANCE -- Choset §4.3.1` COST `O(n)`

in a range scan ρ(q, θ_k) on n bearings (Chapter 3's rangeScan)
out one (d_i, ∇d_i) per visible obstacle: the local minima of ρ over θ, with ∇d_i = −(cos θ, *sin* θ)

1: **for** each bearing k with a finite reading: if $\rho_k \le \rho_{k-1}$ and $\rho_k \le \rho_{k+1}$

and at least one is strict, k is a local minimum (a plateau counts once, at its first ray)

2: emit $d_i = \rho_k$ and $\nabla d_i = -(\cos\theta_k, \sin\theta_k)$: the ray faces the closest point of its obstacle, so the gradient points back along it

What the Apartment taught the check. Each ray reads the exact distance along itself, so the smallest local minimum is never *below* $D(q)$. Broadside — the foot of the perpendicular on the nearest wall at least $D\tan(\pi/n)$ from both ends — the ray nearest the perpendicular hits within π/n of it and overestimates D by at most $1/\cos(\pi/n) - 1$, which is 3.8×10^{-5} for 360 rays. But a zero-thickness wall seen *end-on* subtends no angle, and a fan of 360 rays can miss its end entirely; the nearest *visible* thing is then farther away. In 80 seeded poses, 70 were broadside and met the bound (the Apartment's axis-aligned walls are hit exactly perpendicularly by the fan, so the overestimate there is 10^{-16}), and 10 were end-on with a worst overestimate of **8.3%**. No sensor-only estimator can do better; Choset's remark that "an obstacle distance function may be incorrect if the obstacle is partially occluded by another" is the same limit from the other side.

ALGORITHM NAVIGATION FUNCTION -- Choset eq. (4.13), star worlds via h_λ (eq. 4.19) COST O(n) per evaluation for n obstacles, plus a 2 × 2 finite-difference Jacobian of h_λ

in a sphere world (centre, r_0, discs) or a star world; q_goal; κ; for stars, λ

out ϕ(q) and $\nabla\phi$(q) by the quotient rule; for stars, ($\phi \circ$ h_λ)(q) and J_h$^\mathsf{T}$ $\nabla\phi$ by the chain rule

1: $A \leftarrow d^2(q, q_{\text{goal}})$; $\beta \leftarrow \prod_{i=0}^{n} \beta_i(q)$; $B \leftarrow A^\kappa + \beta$

2: $\varphi \leftarrow A\,B^{-1/\kappa}$; $\nabla\varphi \leftarrow \nabla A\,B^{-1/\kappa} - (A/\kappa)\,B^{-1/\kappa-1}\,(\kappa A^{\kappa-1}\nabla A + \nabla\beta)$

3: **critical points:** damped Newton on $\nabla\psi$, $\psi = \kappa \log A - \sum_i \log\beta_i$, from a lattice of free seeds; polish, deduplicate, classify by the Hessian of ψ; the goal is the minimum it always is

4: **star world:** $h_\lambda(q) = s_{q_{\text{goal}}} q + \sum_{i=0}^{n} s_i T_i(q)$ with a $\beta_i \le 0$ sending q to $T_i(q)$ outright; $(\varphi \circ h_\lambda)(q)$ and $J_h^\top \nabla\varphi(h_\lambda(q))$

5: **suitable λ:** λ_0 = max over a lattice of free points, at least a quarter-radius clear of obstacle i, of $\gamma_\kappa \bar{\beta}_i/\beta_i$ — so every switch is $\le \frac{1}{2}$ outside that collar

The check verifies the quotient-rule gradient against central differences at 210 points (worst relative error 1.6×10^{-7}) and that $\varphi \in [0, 1]$ on the free space.

ALGORITHM CONTROL-POINT LIFTING -- Choset eq. (4.26) COST O(#points · dim Q) per evaluation, plus one segment-scene distance per link for r_float

in Reach and its scene; control points (link, fraction t, role); the goal configuration; whether to add r_float

out u(q) = Σ_j J_j(q)^T ∇U_j(r_j(q)), the configuration-space gradient; per-term forces and torques for drawing

1: **for** each control point j: $r_j \leftarrow$ its world position from Chapter 2's FK; $f_j \leftarrow \nabla U_j(r_j)$ with U_j attractive toward $r_j(q_{\text{goal}})$ or repulsive from every obstacle within $Q^\star$
2: $J_j \leftarrow$ the control-point Jacobian (Chapter 4's `jacobian` for a shortened arm); $u \leftarrow u + J_j^\top f_j$
3: **if** floating: $r_{float} \leftarrow$ the link point nearest any obstacle (one segment–scene distance per link); treat it as one more repulsive control point at its (k, t)
4: **descend** $q \leftarrow q - \alpha\, u(q)$ with Algorithm 4, in joint space

7.5 Implementation in Rust

The `potential` crate is introduced here; `brushfire` is the owned grid distance transform that Chapters 8 and 10 import, and `Potential` is the interface Chapter 19's CHOMP reads a distance field through. Configurations are `SVector<f64, N>` so one `Attractive` serves a point in the plane and a control point on Reach.

RUST **crates/potential/src/lib.rs**

```rust
use nalgebra::SVector;

/// A differentiable scalar field over an N-dimensional configuration space. The gradient is
/// read as a *velocity* for first-order robots (Choset Ch. 4's convention through sec.4.6) and
/// as a *force* once it is lifted through J^T (sec.4.7).
pub trait Potential<const N: usize> {
    fn value(&self, q: &SVector<f64, N>) -> f64;
    fn gradient(&self, q: &SVector<f64, N>) -> SVector<f64, N>;
}

/// Choset eqs. (4.2)-(4.3): quadratic within `d_goal_star`, conic beyond. The two agree in value
/// and gradient at the seam (Derivation 1), so descent never sees it; `f64::INFINITY` gives the
/// pure quadratic of eq. (4.1).
pub struct Attractive<const N: usize> { pub goal: SVector<f64, N>, pub zeta: f64, pub d_goal_star: f64 }

impl<const N: usize> Potential<N> for Attractive<N> {
    fn value(&self, q: &SVector<f64, N>) -> f64 {
```

```
        let d = (q - self.goal).norm();
        if d <= self.d_goal_star { 0.5 * self.zeta * d * d }
        else { self.d_goal_star * self.zeta * d - 0.5 * self.zeta * self.d_goal_star *
    self.d_goal_star }
    }
    fn gradient(&self, q: &SVector<f64, N>) -> SVector<f64, N> {
        let diff = q - self.goal;
        let d = diff.norm();
        if d <= self.d_goal_star { self.zeta * diff }               // eq. (4.1): zeta(q
    - q_goal)
        else { (self.d_goal_star * self.zeta / d) * diff }          // eq. (4.3):
    magnitude zeta d*
    }
}

/// One repulsive term per obstacle (Choset sec.4.1, after eq. 4.7), so the gradient does
    not jump
/// when the nearest obstacle changes. `dist` is the Chapter 4 query: d_i(q) and its
    closest point.
pub struct Repulsive<'w, const N: usize> { pub dist: &'w dyn cspace::DistanceQuery<N>,
    pub eta: f64, pub q_star: f64 }

impl<const N: usize> Potential<N> for Repulsive<'_, N> {
    fn value(&self, q: &SVector<f64, N>) -> f64 {
        self.dist.distances(q).iter().filter(|o| o.d <= self.q_star)
            .map(|o| 0.5 * self.eta * (1.0 / o.d - 1.0 / self.q_star).powi(2))      // eq.
    (4.4)
            .sum()
    }
    fn gradient(&self, q: &SVector<f64, N>) -> SVector<f64, N> {
        let mut g = SVector::zeros();
        for o in self.dist.distances(q).iter().filter(|o| o.d <= self.q_star) {
            // eq. (4.5): eta (1/Q* - 1/d) (1/d2) ?d, with ?d = (q - c)/d (Derivation 2).
            g += self.eta * (1.0 / self.q_star - 1.0 / o.d) / (o.d * o.d) * o.grad;
        }
        g
    }
}

/// U = Sigma terms -- the whole method and its local minima.
pub struct Additive<const N: usize>(pub Vec<Box<dyn Potential<N>>>);
```

Gradient descent returns the iterates *and the reason it stopped*, with the Hessian's eigenvalues when it stopped at a critical point that is not the goal — the stuck badge of the Potential Well.

RUST **crates/potential/src/descent.rs**

```
use nalgebra::{SMatrix, SVector};

pub enum DescentOutcome { Goal, LocalMinimum { hessian_eigs: Vec<f64> }, StepLimit,
    Collision, Stalled }

/// Choset Algorithm 4 with the sign the text intends: q(i+1) = q(i) - alpha(i) ?U(q(i)).
/// `alpha` is a schedule ("often made on an ad hoc or empirical basis", Choset); `eps`
    the
/// forgiving termination ||?U|| < epsilon; `max_step` caps a repulsive spike's fling.
pub fn gradient_descent<const N: usize>(
```

```rust
    u: &dyn Potential<N>, q_start: SVector<f64, N>, alpha: impl Fn(usize) -> f64,
    eps: f64, max_steps: usize, goal: Option<(SVector<f64, N>, f64)>, max_step: f64,
) -> (Vec<SVector<f64, N>>, DescentOutcome) {
    let mut path = vec![q_start];
    for i in 0..max_steps {
        let q = *path.last().unwrap();
        if let Some((g, tol)) = goal { if (q - g).norm() <= tol { return (path,
   DescentOutcome::Goal); } }
        let grad = u.gradient(&q);
        if grad.norm() < eps {
            // Not at the goal, gradient gone: classify the critical point by its Hessian.

            let h = hessian(u, &q, 1e-4);
            let eigs = h.symmetric_eigenvalues().iter().copied().collect();
            return (path, DescentOutcome::LocalMinimum { hessian_eigs: eigs });
        }
        let mut step = -alpha(i) * grad;
        if step.norm() > max_step { step *= max_step / step.norm(); }
        path.push(q + step);
    }
    (path, DescentOutcome::StepLimit)
}

/// ?2U by central differences of the gradient, symmetrized -- Choset reads the type of a
/// critical point off its signature, and Chapter 8 reads the Morse index off the same
   matrix.
pub fn hessian<const N: usize>(u: &dyn Potential<N>, q: &SVector<f64, N>, h: f64) ->
   SMatrix<f64, N, N> {
    let mut m = SMatrix::<f64, N, N>::zeros();
    for j in 0..N {
        let (mut qp, mut qm) = (*q, *q);
        qp[j] += h; qm[j] -= h;
        m.set_column(j, &((u.gradient(&qp) - u.gradient(&qm)) / (2.0 * h)));
    }
    0.5 * (m + m.transpose())
}
```

Brushfire is the owned artifact. It is Chapter 6's breadth-first search with every obstacle pixel as a source, and it keeps the second back pointer that Chapter 8 reads as the grid GVD.

RUST **crates/potential/src/brushfire.rs**

```rust
use std::collections::VecDeque;

#[derive(Clone, Copy)] pub enum Conn { Four, Eight }

/// Choset's labels: obstacle 1, free pixels 2, 3, ...; `back2` is set where a second
   front from a
/// *different* obstacle arrived -- sec.5.2.5's "two different back pointers", the grid
   GVD.
pub struct DistGrid { pub label: Vec<u32>, pub back: Vec<Option<usize>>, pub back2: Vec<
   Option<usize>>, pub source: Vec<i32>, pub w: usize, pub h: usize }

pub fn brushfire(occ: &cspace::Raster, conn: Conn) -> DistGrid {
    let (w, h) = (occ.width(), occ.height());
    let comp = obstacle_components(occ);                    // a front's source is an
   obstacle, not a pixel
```

```
    let mut g = DistGrid { label: vec![0; w * h], back: vec![None; w * h], back2: vec![None; w * h], source: vec![-1; w * h], w, h };
    let mut queue = VecDeque::new();
    for k in 0..w * h {
        if occ.is_occupied(k) { g.label[k] = 1; continue; }
        // Step 1 of the text: free pixels neighbouring an obstacle are labelled 2.
        if let Some(o) = occ.lattice_neighbors(k, conn).find(|&n| occ.is_occupied(n)) {
            g.label[k] = 2; g.back[k] = Some(o); g.source[k] = comp[o]; queue.push_back(k);
        }
    }
    while let Some(c) = queue.pop_front() {
        let (lc, sc) = (g.label[c], g.source[c]);
        for x in occ.free_neighbors(c, conn) {              // Star(c) on Chapter 6's grid graph
            if g.label[x] == 0 {
                g.label[x] = lc + 1; g.back[x] = Some(c); g.source[x] = sc; queue.push_back(x);
            } else if g.source[x] != sc && g.source[x] >= 0 && g.label[x] >= lc {
                // Two fronts meet. Even gap: x was labelled lc + 1 by the other front and now gets
                // its second back pointer. Odd gap: both pixels carry lc and the collision falls on
                // their shared edge -- both are flagged so the ridge has no holes.
                if g.label[x] == lc + 1 && g.back2[x].is_none() { g.back2[x] = Some(c); }
                g.mark_ridge(x); if g.label[x] == lc { g.mark_ridge(c); }
            }
        }
    }
    g
}

impl DistGrid {
    /// Pixels two fronts from different obstacles reached -- Chapter 8's grid GVD.
    pub fn ridge_cells(&self) -> Vec<usize> { (0..self.w * self.h).filter(|&k| self.is_ridge(k)).collect() }
    /// "Point to a lowest neighbor" (Choset sec.4.3.2); ties to the first in scan order.
    pub fn grid_gradient(&self, cell: usize, conn: Conn) -> Option<usize> { /* lowest-label neighbour */ todo_lowest(self, cell, conn) }
}

/// Wave-front (sec.4.5): the same propagation from the goal pixel alone, seeded with 2.
pub fn wavefront(occ: &cspace::Raster, goal: usize, conn: Conn) -> DistGrid { propagate(occ, &[(goal, 2, 0)], conn) }
```

The navigation function is eq. (4.13) with its analytic gradient, and the lifted potential for Reach is a sum of $J^T f$ terms.

RUST `crates/potential/src/navigation.rs · src/lifted.rs`

```
/// Eq. (4.13) on a sphere world; `Potential<2>` via the quotient rule.
pub struct NavigationFunction<'w> { pub world: &'w SphereWorld, pub goal: Point2<f64>, pub kappa: u32 }

impl Potential<2> for NavigationFunction<'_> {
    fn value(&self, q: &Vector2<f64>) -> f64 {
        let a = (q - self.goal.coords).norm_squared();                  // gamma_1 = d2
        let b = a.powi(self.kappa as i32) + self.world.beta(q);         // d^{2kappa} +
```

```
    beta
        if b <= 0.0 { 1.0 } else { a * b.powf(-1.0 / self.kappa as f64) }
    }
    fn gradient(&self, q: &Vector2<f64>) -> Vector2<f64> {
        let k = self.kappa as f64;
        let diff = q - self.goal.coords;
        let a = diff.norm_squared();
        let (beta, grad_beta) = self.world.beta_and_gradient(q);
        let b = a.powf(k) + beta;
        let bm = b.powf(-1.0 / k);
        let da = 2.0 * diff;
        let db = k * a.powf(k - 1.0) * da + grad_beta;
        da * bm - (a / k) * (bm / b) * db                              // quotient rule
    }
}

/// sec.4.7.2-4.7.3: control points on Reach, each with a workspace potential; `gradient` is
/// u(q) = Sigma_j J_j(q)^T ?U_j(r_j(q)), summed in C-space (eq. 4.26) -- never in the workspace.
pub struct LiftedPotential<'r> { pub arm: &'r robots::Reach, pub scene: &'r collide::Scene, pub points: Vec<ControlPoint>, pub goal: Tn<2>, pub floating: bool }

impl Potential<2> for LiftedPotential<'_> {
    fn value(&self, q: &Vector2<f64>) -> f64 { self.terms(q).iter().map(|t| t.value).sum() }
    fn gradient(&self, q: &Vector2<f64>) -> Vector2<f64> {
        let mut u = Vector2::zeros();
        for t in self.terms(q) {
            let j = cspace::control_point_jacobian(self.arm, q, t.link, t.t);   // Chapter 4's jacobian, shortened
            u += j.transpose() * t.force;                                       // u = J^T f
        }
        u
    }
}
```

7.5.1 The worked example, and its printed output

RUST **crates/potential/examples/descent_step.rs · brushfire_7x3.rs · kappa_sweep.rs**

```
fn main() {
    // Part B: one evaluation and one step, as in the text.
    let goal = Vector2::new(4.0, 3.0);
    let att = Attractive { goal, zeta: 1.0, d_goal_star: 10.0 };
    let disc = cspace::DiscObstacle { center: Point2::new(0.0, -2.0), radius: 1.0 };
    let rep = Repulsive { dist: &disc, eta: 1.0, q_star: 2.0 };
    let q = Vector2::zeros();
    println!("U_att = {}  grad = {:?}", att.value(&q), att.gradient(&q));
    println!("U_rep = {}  grad = {:?}", rep.value(&q), rep.gradient(&q));
    let u = Additive(vec![Box::new(att), Box::new(rep)]);
    println!("U = {}  grad = {:?}", u.value(&q), u.gradient(&q));
    println!("q1 = {:?}", q - 0.1 * u.gradient(&q));

    // Part A: the 7 * 3 micro-grid.
```

```
    let occ = cspace::Raster::from_ascii(&[".......", "#.....#", "......."]);  // first
    row is the top
    let d = brushfire(&occ, Conn::Four);
    for row in d.rows_top_first() { println!("{}", row.iter().map(u32::to_string).collect
    ::<Vec<_>>().join(" ")); }
    println!("ridge: {}", d.ridge_cells().iter().map(|&k| d.col_row(k)).map(|(c, r)|
    format!("({c},{r})")).collect::<Vec<_>>().join(" "));

    // The kappa sweep on the five-disc world.
    for kappa in [2, 3, 4, 6, 7, 8, 10] {
        let c = critical_points(&FIVE_DISC_WORLD, FIVE_DISC_GOAL, kappa);
        println!("kappa={kappa}: minima={} saddles={}", c.minima, c.saddles);
    }
}
```

```
U_att = 12.5  grad = [-4.0, -3.0]
U_rep = 0.125  grad = [0.0, -0.5]
U = 12.625  grad = [-4.0, -3.5]
q1 = [0.4, 0.35]
2 3 4 5 4 3 2
1 2 3 4 3 2 1
2 3 4 5 4 3 2
ridge: (4,1) (4,2) (4,3)
kappa=2: minima=3 saddles=7
kappa=3: minima=2 saddles=6
kappa=4: minima=2 saddles=6
kappa=6: minima=2 saddles=6
kappa=7: minima=1 saddles=5
kappa=8: minima=1 saddles=5
kappa=10: minima=1 saddles=5
```

Read Part B against the text. $U_{att} = \frac{1}{2} \cdot 25 = 12.5$ with gradient $q - q_{\text{goal}} = (-4, -3)$. The disc's closest point is $c = (0, -1)$, so $d_1 = 1$ and $\nabla d_1 = (0, 1)$; with $\eta = 1$ and $Q^\star = 2$, $U_{rep} = \frac{1}{2}(1 - \frac{1}{2})^2 = 0.125$ and $\nabla U_{rep} = (\frac{1}{2} - 1) \cdot 1 \cdot (0, 1) = (0, -0.5)$. The total is $U = 12.625$, $\nabla U = (-4, -3.5)$, and one step with $\alpha = 0.1$ lands at $q(1) = (0.4, 0.35)$ — away from the obstacle and toward the goal. `descent_step_matches_text` pins these to 10^{-12}; `brushfire_7x3_labels` asserts all 21 labels and the ridge set; `brushfire_equals_bfs_depth_plus_one` checks on seeded grids that `brushfire` equals `search::bfs` multi-source depth +1 and that the wave-front descent has the Dijkstra cost under unit weights; `kappa_sweep` pins exactly one minimum from $\kappa = 7$; `reach_field` lifts a unit force at Reach's tip and asserts $u_\theta = (r \times f)_z$. The TypeScript port in `web/lib/potential/` runs the same thirteen checks, and every widget on this page is that port.

> **NOTE**
>
> Two of the port's checks failed on first writing, and both failures were mathematics, not typos. The sensor-based distance check assumed the ray fan overestimates D by at most the ray quantisation, and the Apartment's free-standing wall stubs, seen end-on, broke it by 8%: a zero-thickness wall

subtends no angle, and the bound only holds broadside. The star-world check assumed h_λ maps every star boundary onto its circle for $\lambda = 1$, which is true in exact arithmetic and false by six units in floating point once a boundary β_i rounds to -10^{-16}, because $\gamma_\kappa \vec{\beta}_i$ is 10^{17} and the switches cannot see a λ of order one. Both are now stated as the theorems actually say them.

7.6 Putting it together: Rusty in a doorway, Reach at the bench

The integration lab runs Rusty in the Apartment under a *sensor-based* potential: the repulsive term is fed by the local minima of a 72-ray scan of radius 3 m (Algorithm SENSOR-BASED DISTANCE), the attractive term by the goal in the corridor, and the descent is Algorithm 4 at $\alpha = 0.02$. From the middle of room A Rusty heads for the door, and in the alcove beside the jamb — where the goal pulls through the wall and the two wall faces push toward the centre of the alcove — it stops, with the stuck badge reading a positive-definite Hessian. The same raster the Chapter 6 planner used, run through `wavefront` from the goal cell at 4-point connectivity, gives a field with one minimum, and the grid descent walks Rusty out of the alcove and through the door — hugging the jamb at one cell's clearance, which is Choset's "dangerously close to obstacles" and the reason Chapter 8's Voronoi roadmap exists. The Potential Well's wave-front toggle is this lab on a table; the Apartment version is Exercise 4's.

Reach's half of the lab is the Arm in a Field widget run to the end: the same start descended twice, with the first collision logged. Without r_{float} link 1 meets the post after a dozen steps between its two fixed repulsive points; with it the arm reaches the goal pose in about five hundred. Move the goal into the block's shadow and you will find starts where it stalls in a local minimum of the lifted potential instead — the additive disease, now on the torus — and starts where even r_{float} fails, because a floating point repels the nearest link point and nothing else.

Three pointers close the chapter. Brushfire's double-back-pointer pixels are the grid generalized Voronoi diagram, and Chapter 8 proves the exact version is a roadmap, with the Morse vocabulary this chapter's "nondegenerate critical point" was borrowing from. Chapter 19's CHOMP descends a potential whose obstacle term is a distance field exactly like the brushfire one, over a whole trajectory at once, and inherits every local minimum on this page. And Part III's planners are the field's eventual answer to incompleteness: when the valley you roll into is not the goal's, sample.

7.7 Exercises

1 FOUNDATION •• **Convexity of d_i and the seams of U_rep**

Prove that $d_i(q) = \min_{c \in QO_i} \|q - c\|$ is a convex function of q when QO_i is convex (Choset Ch. 5, Problem 19, brought forward). Conclude that with the per-obstacle sum $U_{rep} = \sum_i U_{rep_i}$, the only places ∇U_{rep} can fail to be continuous are the spheres $d_i = Q_i^\star$ — and show that in fact it is continuous there too, so the seams that remain are those of Derivation 2's Step 4: where a *nonconvex* obstacle has two closest points.

> **HINTS**
>
> <>A minimum over c of a jointly convex function of (q, c) is convex in q. Is ‖q − c‖ jointly convex? Is the constraint set convex?</> <>For the seam at d_i = Q: *evaluate* $\eta(1/Q - 1/d\_i)/d\_i^2$ as d_i → Q* from below.</> <>For the nonconvex case, take the horseshoe of the Potential Well and a point on its axis: name the two closest points and the two limiting gradients.</>

2 FOUNDATION •• **Wave-front paths are L_1-shortest**

Show that the path produced by wave-front descent under 4-point connectivity is shortest in the L_1 (Manhattan) metric among all grid paths from the start to the goal (Choset Ch. 4, Problem 2). Then give a 3×3 example where the 8-point descent path — shortest in link length — is *not* Euclidean-shortest among 8-connected paths, and say by how much.

WORK IT OUT

On an empty 3×3 grid from the bottom-left to the top-right cell under 8-point connectivity, the wave-front path has two links. What is the Euclidean length of a two-link path that goes diagonal, diagonal, divided by the length of one that goes east, east, then north, north? (Give the ratio as a decimal.)

Answer. 0.7071 Two diagonals measure $2\sqrt{2} \approx 2.83$ against four axial steps' 4: the ratio is $\sqrt{2}/2 \approx 0.707$. Both have the same wave-front label difference under 8-point connectivity only if you count links, which is the point: the wave-front field measures the grid metric, and under 8-point connectivity that metric is chessboard distance, which calls the two-diagonal path and a two-link axial path (which does not exist here) equally short. Choset's 1/1.4 weights of Chapter 6 repair the worst of this; brushfire with those weights is a Dijkstra, not a BFS — Exercise 5.

3 **CONCEPTUAL** • **Predict the κ threshold, then move an obstacle**

PREDICT FIRST

In the Navigation Function Sculptor, read the critical-point count at $\kappa = 6$: two minima, six saddles. Without moving the slider: what is the smallest κ at which the count will read one minimum?

- ✓ $\kappa = 7$
  The last spurious minimum sits near the obstacle at $(-2.3, 1.6)$, on its radial line to the goal; at $\kappa = 7$ the goal term finally dominates $\partial\beta$ there and the minimum merges with its saddle. The check pins the table: 2, 2, 2, 1, 1, 1 for $\kappa = 4, 6, 7, 8, 10$.
- $\kappa = 10$, as in Choset's figure 4.15
  Choset's figure is for his obstacle layout, not ours. The theorem says a large enough κ exists; the threshold depends on where the discs are, and ours lose their last spurious minimum earlier.
- It never reads one — some layout always keeps a spurious minimum
  That is the additive potential's disease, which the navigation function is built to avoid: on any sphere world a finite κ makes every spurious minimum a saddle. The widget's counter shows the moment it happens.

Verify, then drag the goal toward the obstacle at $(-2.3, 1.6)$ — the one that held the last spurious minimum. The threshold moves *down*: a goal closer to an obstacle makes the goal term steeper on that obstacle's radial line sooner. Drag it away and the threshold rises. Then switch on star mode and confirm the count does not change for any $\lambda \geq \lambda_0$ — and find the λ at which it does.

4 CONCEPTUAL •• **A world of only convex discs, and the price of the escape**

In the Potential Well choose *two convex discs*, drag the goal and the discs until the bead gets stuck, and record the Hessian eigenvalues the badge prints. Then enable wave-front mode and estimate how close the escape path passes to the discs — Choset's "dangerously close to obstacles" in §4.5. Explain why the wave-front path *must* graze: among all grid paths of the same link length to the goal it has no reason to prefer clearance, and the one found first by "step to any neighbour labelled one less" is whichever the scan order produced.

HINTS

<>Clearance on the grid is a brushfire label; the wave-front label is a different BFS. Which one does "step to a lower neighbour" consult?</> <>Chapter 8 will combine the two: the GVD is the ridge of the brushfire field, and a path that follows it maximizes clearance while the wave-front minimizes length.</>

5 PRACTICAL •• **Brushfire as Dijkstra, within one cell of the EDT**

Implement an 8-point `brushfire` with diagonal cost 1.4 (Chapter 6's grid metric) as a Dijkstra rather than a BFS: labels become real-valued distances and the queue becomes a heap. Property-test it on 200 seeded grids against the exact Euclidean distance transform ported from the sister book (`mapping/edt`): assert that every cell's label is within one cell of the EDT, and record the worst gap. Then repeat with diagonal cost $\sqrt{2}$ and report whether the worst gap improves — and why it cannot reach zero (the octile metric is not the Euclidean one).

6 PRACTICAL ••• **Wave-front on the torus**

Implement Choset Ch. 4 Problem 5: a wave-front planner for Reach on the T^2 raster of Chapter 4, with the goal seeded at the goal configuration and adjacency *wrapping across the seam* at $\pm\pi$ in both axes. Animate the arm on the Workbench along the descent path, and assert in a test that the path's link length equals the Chapter 6 breadth-first distance on the same torus graph — then compare the number of cells the wave-front labels with the number A* expands for the same query, and say which planner you would run twice.

7.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

Chapter 4 is this chapter's source: the additive potential and its honesty items, Algorithm 4, sensor-based and brushfire distance, the local-minimum figures 4.10–4.11, the wave-front planner, Definition 4.6.1 and the sphere- and star-world constructions, and the virtual-work lifting of §4.7 with Example 4.7.1 and figure 4.20. `link`

Khatib, O. (1986). Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. *International Journal of Robotics Research 5(1), 90–98*.
The artificial potential field for manipulators: attractive goal, repulsive obstacles, and forces at points on the links lifted through the Jacobian — §4.7 in its original form, written for control rather than planning. doi:10.1177/027836498600500106

Rimon, E. and Koditschek, D. E. (1992). Exact Robot Navigation Using Artificial Potential Functions. *IEEE Transactions on Robotics and Automation 8(5), 501–518*.
Navigation functions: Definition 4.6.1, the sphere-world formula (4.13) with its existence-in-κ theorem, and the star-to-sphere diffeomorphism h_λ with its switches — Derivations 5 and 6. doi:10.1109/70.163777

Barraquand, J. and Latombe, J.-C. (1991). Robot Motion Planning: A Distributed Representation Approach. *International Journal of Robotics Research 10(6), 628–649*.
The Randomized Path Planner: gradient descent on a grid potential with random walks to escape local minima — Choset's reference [37], the honest 1991 fix and the ancestor of Part III's sampling. doi:10.1177/027836499101000604

Latombe, J.-C. (1991). Robot Motion Planning. *Kluwer Academic Publishers*.
Chapter 7 is the classical textbook treatment of potential-field methods, including the wave-front (numerical navigation function) planner and the local-minimum analysis this chapter condenses. link

Ratliff, N., Zucker, M., Bagnell, J. A., and Srinivasa, S. (2009). CHOMP: Gradient Optimization Techniques for Efficient Motion Planning. *IEEE International Conference on Robotics and Automation (ICRA), 489–494*.
Forward pointer only: the obstacle term of CHOMP is a distance field like brushfire's, descended over a whole trajectory, and it inherits this chapter's local minima. Chapter 19 derives it. doi:10.1109/ROBOT.2009.5152817

Thrun, S., Burgard, W., and Fox, D. (2005). Probabilistic Robotics. *MIT Press*.
The sister volume's map representations chapter ports the exact Euclidean distance transform this chapter compares brushfire against; the two are different algorithms computing different metrics. link

CHAPTER 8

Roadmaps I: Visibility Graphs and the Generalized Voronoi Diagram

CLASSICAL PLANNERS — SEARCH, POTENTIALS, ROADMAPS, CELLS · Intermediate · 60 min

> *Robots use roadmaps in much the same way people use highway systems.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), Chapter 5*

Chapters 6 and 7 answered one query at a time. This chapter builds something to keep: a network of one-dimensional curves in $\mathcal{Q}_{\text{free}}$ that captures everything a planner needs about the space, so that every later query is "get on, ride, get off". The surprise is that *two very different curves qualify*. The **visibility graph** strings obstacle vertices together along lines of sight, and it contains the shortest path — one that grazes every corner it turns. The **generalized Voronoi diagram** is the set of points equally far from the two nearest obstacles — the path that stays as far from everything as it can. One optimizes length, the other clearance; both are roadmaps because both satisfy the same three-word contract.

Proving the contract for the GVD brings in two ideas that run through the rest of Part II. A **deformation retraction** melts the free space onto its skeleton without tearing, so every loop in the free space survives as a loop on the skeleton — which is why the skeleton is connected wherever the space is. The **preimage theorem** says that one equation in the plane cuts out a curve wherever its gradient does not vanish, which is why the GVD is one-dimensional and also why its definition needs a refinement that a range sensor would have insisted on anyway. The theorem arrives with the Morse vocabulary — nondegenerate critical points, their index, the Euler characteristic — introduced in a definition box here and

used without ceremony in Chapters 9 and 10.

Everything on this page runs. The visibility graph is built by Choset's rotational sweep and checked against the brute-force definition at sixteen hundred vertices; the exact Voronoi diagram of the chapter's slab is traced by Chapter 3's predictor–corrector to 10^{-12} and its meet point lands on $(2, 1.5)$ to nine digits; the Apartment's GVD is a brushfire ridge, accessed by gradient ascent from a hundred and forty-five seeded points, every one of which arrives.

AFTER THIS CHAPTER YOU CAN

- State the roadmap contract of Choset Def. 5.0.2 — accessibility, departability, connectivity — and check it for a given set of curves
- Define the visibility graph, prove it contains a Euclidean-shortest path, build it by rotational plane sweep in $O(n^2 \log n)$, and prune it to supporting and separating lines without losing that path
- Define the generalized Voronoi diagram through two-equidistant faces, access it by gradient ascent of the distance function, and say why the surjectivity refinement SS_ij is what makes the definition unique
- Explain what a deformation retraction is, why the GVD is one, and why that makes it connected and a faithful record of the free space's loops
- Use the preimage theorem to show the planar GVD is one-dimensional, and carry away the Morse vocabulary — nondegenerate critical point, index, Euler characteristic — that Chapters 9 and 10 use
- Build the GVD three ways — traced from distances, as brushfire ridges, and from polygon features — and answer a query on either roadmap with Chapter 6's A*

ASSUMED

- Worlds, polygons and distance queries (Chapter 2); range scans, D(x) and the curve tracer traceCurve (Chapter 3)
- Polygonal C-obstacles for a disc robot and the T^2 raster for Reach (Chapter 4); manifolds, tangent spaces and the differential (Chapter 5)
- A* on a graph (Chapter 6); brushfire and its two-back-pointer pixels (Chapter 7)

8.1 The problem: one query, three answers

Put Rusty in room A and the goal in the bedroom, and solve the query three ways: A* on the Chapter 6 lattice, the visibility-graph path on the inflated walls, and the path down the middle of the Voronoi diagram.

One Apartment query, three planners. Orange: A* on a quarter-metre lattice inflated by Rusty's radius. Purple: the visibility-graph path, which hugs every corner it turns. Blue: the GVD path, which keeps to the middle of every room and doorway. The readout prints each path's length and minimum clearance; the three discs drive at the same speed.

The lattice path is Chapter 6's answer and it is fine. The visibility path is shorter: 6.7 m on average over fifty seeded queries in the chapter's check, against 10.7 m for the Voronoi path. The Voronoi path is safer: its minimum clearance averages 0.48 m against 0.22 m — the visibility path runs exactly at Rusty's radius from the walls, because that is what "shortest" means for a disc. Rusty, being a disc with odometry error and a controller that overshoots, might prefer the second — or might not, if the corridor is long and the battery is low. The point of the chapter is not that one is right. It is that both are *roadmaps*: built once, searched many times, with the same three guarantees.

8.2 Building intuition

8.2.1 The highway contract

DEFINITION 5.0.2 — ROADMAP

A union of one-dimensional curves RM is a **roadmap** if for every q_{start} and q_{goal} in $\mathcal{Q}_{\text{free}}$ that can be connected by a path: (1) **accessibility** — there is a path from q_{start} to some $q'_{start} \in RM$; (2) **departability** — there is a path from some $q'_{goal} \in RM$ to q_{goal}; (3) **connectivity** — there is a path *in* RM from q'_{start} to q'_{goal}.

That is Los Angeles. You plan a path onto the 110, ride to the 105 and the 405, and get off at the beach; the side streets are never searched. For a robot the payoff is that the search happens in a one-dimensional set, however many dimensions $\mathcal{Q}$ has, and that building the roadmap can be done once — or, with a range sensor, incrementally while exploring. The visibility graph satisfies (1) and (2) by definition: every point of a polygonal free space sees some vertex. The GVD satisfies them by *gradient ascent*: walk away from the nearest obstacle until a second one is equally near. Connectivity is the theorem in each case.

8.2.2 A lighthouse

INTERACTIVE · w8.1

Rotational Sweep

Visibility does not need O(n) tests per pair: a turning half-line keeps the crossed edges sorted, and only the nearest one can occlude. The reduced graph drops needless edges and loses no shortest path.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch08-roadmaps-i#w8.1

The obvious way to decide which vertices v can see is to test the segment to each one against every obstacle edge — $O(n^2)$ per vertex, $O(n^3)$ for the graph. The sweep is a different idea: a beam of light turning about v illuminates, at any moment, the nearest thing in its direction, and the nearest thing changes only when the beam passes a vertex. So pause only at vertex angles, keep the list S of edges the beam currently crosses sorted by distance, and ask of each vertex a single question: *is it nearer than the nearest edge in S?* The table beside the canvas is Choset's Table 5.1 filling in as the beam turns; three vertices pass the test and the other five are behind something. When the sweep finishes, the full graph appears with the A* path from v to the goal, and the *reduced* toggle fades every edge that does not lie on a supporting or separating line — the path length does not move. That is the misconception this widget kills: visibility does not cost $O(n)$ tests per pair, and most of the edges it finds are never needed.

8.2.3 Where fronts collide

INTERACTIVE · w8.2

Voronoi Carver

Where brushfire fronts meet is the GVD — and between a vertex and an edge the exact diagram is a parabola, not a line. Grid ridges are jagged and converge as the cells shrink.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch08-roadmaps-i#w8.2

Chapter 7 grew brushfire fronts from every obstacle and noticed that a pixel reached by two fronts from different obstacles keeps two back pointers. The Voronoi Carver colours those pixels solid as the fronts arrive and overlays the exact diagram in purple. On the slab — two pegs and a wall — the exact diagram is the vertical bisector of the pegs below their meet point and two *parabolas* above it: each parabola has a peg as its focus and the wall as its directrix, because "equidistant from a point and a line" is the definition of a parabola. The meet point is $(2, 1.5)$, with clearance 2.5. Slide the resolution: at one-metre cells the

ridge is a staircase whose meet cell is a cell and a half from the exact one, because the chessboard metric meets at $(2, 2)$ and the Manhattan metric at $(2, 1)$; at five centimetres it hugs the curves. The misconception killed: the Voronoi diagram of polygons is *not* made of straight lines.

8.2.4 Melting the candy

> INTERACTIVE · w8.3
>
> **Retract Candy**
>
> A skeleton is not any thinner version of the space. Only a deformation retract keeps the holes — and gradient ascent onto the GVD is one.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch08-roadmaps-i#w8.3

Imagine a doughnut-shaped candy dissolving. What remains at the end is a ring, far smaller than the candy but with the same hole. The Retract Candy does this to the Apartment: every free point slides along its own gradient-ascent path onto the GVD, and the scrub is the time t of the homotopy $H(q, t)$. Two loops ride along — one around a kitchen island, one in an empty room — and their winding numbers about the island are printed at $t = 0$ and now: $1 \to 1$ and $0 \to 0$, at every t. Then flip to *retract to a point*. That map is also continuous and also fixes its image; it is a retraction. But the orange loop has to cross the island to reach the point — the red marks — and its winding number is lost. A deformation retraction is a retraction that is homotopic to the identity, and that is the property that keeps the holes.

8.3 The mathematics

RM	a roadmap: a union of one-dimensional curves in Q_free satisfying Def. 5.0.2
v_i, e_{ij}	visibility-graph nodes (obstacle vertices, start, goal) and line-of-sight edges
l, α_i, E, S	the sweep half-line, vertex angles, the sorted angle list, the active edge list (Alg. 5)
n, N	number of obstacles; number of obstacle vertices
F_i	generalized Voronoi region of QO_i (eq. 5.1)
S_{ij}, SS_{ij}, F_{ij}	two-equidistant surface; its surjective subset (∇d_i $\neq \nabla$d_j); two-equidistant face
F_{ijk}	meet point: equidistant to three closest obstacles; a boundary point has D = 0

$H(x,t)$, $f \sim g$, $\pi_1(X, x_0)$	homotopy; homotopic maps; the first fundamental group (loop classes through x_0)
$G^{-1}(n)$, DG, $\Sigma(G)$, $T_m M$	preimage; differential; critical set; tangent space of M at m

8.3.1 Visibility

DEFINITIONS — VISIBILITY GRAPH; SUPPORTING AND SEPARATING LINES; REDUCED GRAPH (CHOSET §5.1.1)

In a polygonal planar $\mathcal{Q}_{\text{free}}$, the **visibility graph** has as nodes q_{start}, q_{goal} and every vertex of every C-obstacle, and an edge e_{ij} between v_i and v_j iff $sv_i + (1-s)v_j \in \text{cl}(\mathcal{Q}_{\text{free}})$ for all $s \in [0,1]$. Obstacle edges are edges of the graph. A **supporting line** is tangent to two obstacles with both on one side; a **separating line** is tangent to two obstacles with them on opposite sides; for a polygon, tangency at a vertex v_i is the nonsmooth kind, $B_\epsilon(v_i) \cap l \cap \mathcal{QO}_i = \{v_i\}$. The **reduced visibility graph** keeps only edges on supporting or separating lines. A boundary is **locally convex** at $\bar{c}$ if the tangent line there separates the nearby boundary points; at a reflex vertex no line is tangent, so the reduction removes every edge at it.

$e_{ij} \neq \emptyset \iff s\,v_i + (1-s)\,v_j \in \text{cl}(\mathcal{Q}_{\text{free}}) \;\; \forall s \in [0,1]$, shortest path = A* on (V, E) from q_{start} to q_{goal}.

DERIVATION The visibility graph contains a shortest path

\text{bends only at obstacle vertices} \Rightarrow \text{a path in the visibility graph}

Statement (Choset §5.1.1). In a polygonal planar $\mathcal{Q}_{\text{free}}$ a Euclidean-shortest path from q_{start} to q_{goal} is a polyline whose interior vertices are obstacle vertices; hence it is a path in the visibility graph, and A* with the Euclidean heuristic finds it.

Step 1 — locally shortest means straight. Any sub-arc of a shortest path is shortest between its endpoints; in the open free space the shortest curve between two points is the segment, so the path is straight wherever it is not touching an obstacle.

Step 2 — bends live on the boundary. A bend at a point interior to the free space could be shortcut by a chord; so bends occur only on $\partial\mathcal{Q}_{\text{free}}$.

Step 3 — not on an edge's interior. A bend at a point interior to an obstacle edge could be shortcut too: the chord between two nearby path points stays on the free side of the edge's line. So bends occur only at vertices.

Step 4 — consecutive bends see each other. Between two bends the path

is a segment in $\mathrm{cl}(\mathcal{Q}_{\text{free}})$, which is the definition of a visibility edge. ■
Connectivity is Choset's Problem 1 and this chapter's Exercise 1: within a component of $\mathcal{Q}_{\text{free}}$, every vertex sees some vertex, and the obstacle edges close the chain. *What breaks in* $\mathbb{R}^3$*:* a shortest path around a polyhedron bends on edges, not at vertices, so the visibility graph of vertices misses it (Choset §5.1.1; Exercise 1). *Taut paths use only supporting and separating lines:* at a bend the path wraps the vertex, so both polygon neighbours of the vertex lie on one side of the path's line — the nonsmooth tangency above. That is why the reduction loses no shortest path, and the check confirms it: on fifty seeded fields the full and reduced graphs give the same start-to-goal length every time, with 3,875 edges in place of 8,386. The same check samples 405 random collision-free polylines from start to goal and finds none shorter than A* on the graph.

DERIVATION **Rotational plane sweep is correct and runs in O(n² log n)**

v_i \text{ visible} \iff |v v_i| < \text{dist to the nearest edge in } S \text{ and } l \text{ does not enter the obstacle at } v

Statement (Choset §5.1.2, Algorithm 5). Sweep a half-line l from v through $[0, 2\pi)$, pausing at the vertex angles α_i in increasing order, and maintain the set S of edges crossing l sorted by distance from v. Then v_i is visible iff the segment vv_i does not cross the nearest edge of S and l does not pass through the interior of the obstacle at v.
Step 1 — nothing happens between vertices. The set of edges crossing l changes only when l passes an endpoint of an edge, that is, a vertex. So pausing at the α_i loses nothing.
Step 2 — only the nearest crossed edge can occlude. Every edge in S crosses l at some distance; if the nearest crossing lies beyond v_i the segment vv_i crosses none of them, and no edge outside S crosses l at all. Two exceptions are handled by hand: edges incident to v_i reach it exactly and do not occlude it, and when v is itself a vertex, directions into its own polygon are blocked whatever S says.
Step 3 — bookkeeping is logarithmic. At α_i the two edges at v_i either end (delete from S) or begin (insert, by distance along l). Edges in S do not cross one another, so their order by distance is invariant while they are in S; with a balanced tree each operation is $O(\log n)$.
Step 4 — the count. n events per vertex, each $O(\log n)$, plus an $O(n \log n)$ sort: $O(n \log n)$ per vertex and $O(n^2 \log n)$ for the graph. ■
Table 5.1, reproduced. For Fig. 5.8 — rectangle 1 on $[2, 4] \times [-1, 1]$, rectangle

2 on $[6, 10] \times [-2, 4]$, v at the origin — the initial list is $S = \{E_4, E_2, E_8, E_6\}$, the events come in the order $\alpha_3, \alpha_7, \alpha_4, \alpha_8, \alpha_1, \alpha_5, \alpha_2, \alpha_6$, and the edges added are exactly $(v, v_4), (v, v_8), (v, v_1)$. The port's row for α_1 reads $\{E_4, E_1\}$ where Choset prints $\{E_1, E_4\}$; his next row, $\{E_4, E_1, E_8, E_5\}$, agrees with the port, and the two cannot both be sorted by distance — just past α_1 the beam meets E_4 (the line $x = 2$) before E_1 (the line $y = -1$). *Two honesty items:* the TypeScript port keeps S as a sorted array, $O(n)$ per update rather than $O(\log n)$, which changes the constant and not the idea; and general position — no three vertices collinear — is assumed, as Choset assumes it, with the perturbation he suggests as the fix. The check runs the sweep and the brute-force definition at 1,598 vertices over fifty seeded convex fields and finds no disagreement.

8.3.2 The generalized Voronoi diagram

DEFINITIONS — VORONOI REGION, TWO-EQUIDISTANT SETS, THE GVD (CHOSET §5.2.1, §5.2.4)

The **generalized Voronoi region** of QO_i is $F_i = \{q \in Q_{\text{free}} : d_i(q) \le d_h(q)\ \forall h \ne i\}$ (eq. 5.1). The **two-equidistant surface** is $S_{ij} = \{q : d_i(q) - d_j(q) = 0\}$; its **surjective** subset is $SS_{ij} = \{q \in S_{ij} : \nabla d_i(q) \ne \nabla d_j(q)\}$; the **two-equidistant face** is $F_{ij} = \{q \in SS_{ij} : d_i(q) \le d_h(q)\ \forall h\}$. The **GVD** is $\bigcup_i \bigcup_j F_{ij}$. In the plane the F_{ij} are **GVD edges**, ending at **meet points** F_{ijk} (equidistant to three closest obstacles) or **boundary points** (where $D = 0$, the ends of the diagram's spokes).

DEFINITIONS — RETRACTION, HOMOTOPY, DEFORMATION RETRACTION, π_1 (CHOSET §5.2.3)

For a manifold X and $A \subset X$, a **retraction** is a continuous $f : X \to A$ with $f(a) = a$ for all $a \in A$; A is the **retract**. A **homotopy** between $f, g : U \to V$ is a continuous $H : U \times [0, 1] \to V$ with $H(\cdot, 0) = f$ and $H(\cdot, 1) = g$; then $f \sim g$. A **deformation retraction** is a homotopy $H : X \times [0, 1] \to X$ with $H(x, 0) = x$, $H(x, 1) \in A$ and $H(a, t) = a$ for $a \in A$ — a retraction homotopic to the identity — and A is a **deformation retract**. The **first fundamental group** $\pi_1(X, x_0)$ is the group of homotopy classes of loops through x_0 under concatenation; a deformation retraction preserves it: $\pi_1(X, x_0) = \pi_1(A, f(x_0))$.

DERIVATION **Accessibility of the GVD**

\dot c = \nabla D(c) \text{ reaches the GVD from any } q \in \Qfree

Statement (Choset Lemma 5.2.1). In an obstacle-bounded environment, gradient ascent of the distance function, $\dot{c}(t) = \nabla D(c(t))$ with $c(0) = q_{\text{start}}$ (eq. 5.2), traces a path from any free point to the GVD.

Step 1 — the flow is ∇d_i. Let QO_i be the unique closest obstacle at q. Nearby, $D = d_i$ and the ascent follows ∇d_i, the unit vector away from the closest point (Chapter 7, Derivation 2).

Step 2 — d_i grows, something else must stop it. Along the flow d_i increases at unit rate. The environment is bounded, so d_i cannot grow forever; by continuity of the distance functions some d_j must stop exceeding d_i.

Step 3 — the tie. There is a $\bar{t}$ with $d_i(c(\bar{t})) = d_j(c(\bar{t}))$, a point of S_{ij}. ■

Two refinements the port needs. First, the tie must be in SS_{ij}: two collinear wall pieces sharing an endpoint tie at every point beyond the endpoint with $\nabla d_i = \nabla d_j$, and that is not a Voronoi edge — the ascent skips such ties and continues with the new nearest obstacle. Second, the crossing is detected as a change of nearest obstacle between two steps and then bisected, so the landing point is on S_{ij} to 10^{-15} rather than within a step of it. *Departability* is accessibility run backward from the goal. Choset adds that every free point sees some GVD point along a segment, so a robot that can sense the goal may simply drive at it once in sight. *The numbers.* From 145 seeded free points of the Apartment every ascent lands, with $|d_i - d_j|$ at most 3×10^{-15}, and every landing point lies within 0.11 m of the 0.1 m grid GVD below; on the open slab, 107 of 120 starts land within one trace step of the traced diagram and the other 13, below the pegs, walk out of the window — the lemma's boundedness hypothesis is not decoration.

DERIVATION **The GVD is a deformation retract, hence connected, hence a roadmap**

$RM \sim \mathrm{id}, \quad \pi_1(Q_{\text{free}}) = \pi_1(\text{GVD})$

Statement (Choset §5.2.2–5.2.3). The map $RM : Q_{\text{free}} \to \text{GVD}$ that sends q to the landing point of its gradient ascent (and fixes the GVD) is a retraction homotopic to the identity. Consequently each connected component of Q_{free} contains exactly one connected component of the GVD, and the GVD has the same loop classes as the free space.

Step 1 — it is a retraction. $RM(q) = q$ on the GVD by definition; elsewhere it is the ascent endpoint, which exists by Lemma 5.2.1.

Step 2 — it is continuous. Nearby starts follow nearby flow lines of the

piecewise-smooth field ∇D and land nearby (Choset cites [340] for the proof).

Step 3 — it is homotopic to the identity. $H(q,t) = c_q(t)$, the point at fraction t of q's own ascent path, is continuous in (q,t), equals q at $t = 0$, equals $RM(q)$ at $t = 1$, and fixes the GVD throughout. That is a deformation retraction.

Step 4 — connectivity. The image of a connected set under a continuous map is connected, so RM carries each component of $\mathcal{Q}_{\text{free}}$ onto a connected piece of the GVD; and since the GVD lies in $\mathcal{Q}_{\text{free}}$ and RM fixes it, that piece is a whole component.

Step 5 — loops survive. Deformation retracts preserve π_1, so a loop in $\mathcal{Q}_{\text{free}}$ that cannot be shrunk to a point — one around an island — retracts to a loop on the GVD that cannot either. ■

Retract versus deformation retract. A point is a retract of the filled doughnut (send everything to it) but not a deformation retract: the straight-line homotopy would have to pass through the hole. The Retract Candy shows exactly this — the loop around the island keeps winding number 1 about it at $t = 0$, $\frac{1}{2}$ and 1 under the gradient-ascent homotopy, every one of its 48 points lands on the GVD and none enters the island, while the straight-line retraction to a corridor point pierces an obstacle on 14 of the 48 rays. The 0.1 m grid GVD of the Apartment has one connected component, as the free space does.

DEFINITIONS — REGULAR VALUE, CRITICAL POINT, CRITICAL VALUE (CHOSET §5.2.4)

For smooth $G : M \to N$, a point m is a **critical point** if the differential $DG(m)$ is not surjective; its image $G(m)$ is a **critical value**; $\Sigma(G)$ is the set of critical points. A value $n \in N$ is **regular** if $DG(m)$ is surjective for every $m \in G^{-1}(n)$. For a real-valued G the differential is the row vector $(\nabla G)^\top$, surjective iff $\nabla G \neq 0$.

DEFINITION BOX — MORSE FUNCTION (INTRODUCED HERE; IMPORTED BY CHAPTERS 9–10)

A smooth real-valued f is **Morse** if every critical point is **nondegenerate** (nonsingular Hessian). Nondegenerate critical points are isolated, and each has an **index**: the number of negative eigenvalues of the Hessian — 0 a minimum, $\dim M$ a maximum, anything else a saddle. The topology of the level sets $f^{-1}(\lambda)$ changes only as λ crosses a critical value, and $\sum(-1)^{\text{index}}$ over the critical points is the Euler characteristic of M. *Example:* $h(\theta_1, \theta_2) = \cos\theta_1 + \cos\theta_2$ on Reach's torus has four critical points, of index $2, 1, 1, 0$ — a maximum at $(0,0)$, saddles at $(\pi,0)$ and $(0,\pi)$, a minimum

at (π, π) — and $1 - 2 + 1 = 0 = \chi(T^2)$; the check computes the indices from Chapter 7's Hessian. Chapter 7's navigation functions are Morse by definition. The distance function D is *not* smooth, which is why Chapter 9 introduces the generalized gradient.

DERIVATION **The planar GVD consists of one-dimensional manifolds**

$SS_{ij} \subset (d_i - d_j)^{-1}(0)$ is a 1-manifold, $\quad T_q SS_{ij} = \ker(\nabla d_i - \nabla d_j)^\top$

Statement (Choset Theorem 5.2.2, the preimage theorem). Let $G : M \to N$ be smooth and n a regular value. Then $G^{-1}(n)$ is a closed submanifold of M of dimension $\dim M - \dim N$, with tangent space $T_m G^{-1}(n) = \ker DG(m)$. For $G = d_i - d_j : Q \to \mathbb{R}$ the set SS_{ij} is one-dimensional in the plane.
Step 1 — the differential. $D(d_i - d_j) = (\nabla d_i - \nabla d_j)^\top$, a 1×2 row.
Step 2 — surjective iff the gradients differ. A 1×2 row is surjective onto $\mathbb{R}$ iff it is nonzero, i.e. iff $\nabla d_i(q) \neq \nabla d_j(q)$.
Step 3 — that is the definition of SS_{ij}. The word *surjective* in "two-equidistant surjective surface" is this condition. On SS_{ij}, 0 is a regular value of $d_i - d_j$, so SS_{ij} is a submanifold of dimension $2 - 1 = 1$, with tangent perpendicular to $\nabla d_i - \nabla d_j$.
Step 4 — faces are submanifolds. $F_{ij} \subset SS_{ij}$ is cut out by the closed conditions $d_i \leq d_h$; it is a submanifold with boundary, and the GVD is a finite union of them. ■
The warm-up. $f(x, y) = x^2 + y^2$ has $Df = [2x, 2y]$, nonzero everywhere on $f^{-1}(91{,}538)$ — the check finds $|Df| = 605.1$ there — so that preimage is a one-manifold: a circle. *Why the refinement is needed.* Cut a nonconvex obstacle into two convex pieces in two different ways (Choset Fig. 5.10) and you get two different S_{ij} — but the same SS_{ij}, because the parts that differ are exactly those where the two closest points coincide and the gradients line up. A robot at q_2 *outside* the concavity sees one local minimum of ρ, not two (Fig. 5.11): the sensor could never see the cut, and SS_{ij} is the definition that agrees with the sensor. *The tangent.* $\ker(\nabla d_i - \nabla d_j)^\top$ is perpendicular to the chord joining the two closest points (Fig. 5.13): on the slab, at the meet point $(2, 1.5)$, $\nabla d_{P_1} = (0.8, 0.6)$ and $\nabla d_W = (0, -1)$, so the tangent along $P_1|W$ is $\propto (1, -0.5)$ — slope -0.5, and $(1, -0.5) \cdot (2, 4) = 0$ against the chord from P_1 to the wall's foot $(2, 4)$. That tangent is also what Chapter 3's tracer follows.

DERIVATION **Size and construction of the polygonal GVD**

$\frac{3(n+1)}{2} \le \#\text{edges} \le 6N + 3n - 3, \qquad \frac{n+5}{2} \le \#\text{nodes} \le 4N - n - 2$

Statement (Choset §5.2.5). In a polygonal world with n obstacles and N vertices, the GVD's edges are straight (vertex–vertex, edge–edge) or parabolic (vertex–edge), and their number lies between $\frac{3(n+1)}{2}$ and $6N+3n-3$; the nodes number between $\frac{n+5}{2}$ and $4N - n - 2$ [Choset ref. 359].
Step 1 — three bisectors. Points equidistant from two points lie on a line; from two lines, on their angle bisectors; from a point and a line, on a parabola with that focus and directrix.
Step 2 — partition by feature pair. The plane splits into regions by which pair of features (vertex or edge) is closest (Fig. 5.14); on each region the GVD edge is one of the three curves.
Step 3 — count. The GVD is a planar graph; Euler's formula with the bounds on how many feature pairs can be adjacent gives the counts. ■

> The bisector zoo (Choset Fig. 5.14), computed by the module's analytic bisectors. Vertex against vertex and edge against edge give lines; vertex against edge gives a parabola with the vertex as focus and the edge as directrix.

Brushfire. Choset's third construction is Chapter 7's: a pixel the brushfire reaches by two fronts with different back pointers is a GVD pixel, and `ridge_cells` returns them. The grid version of the SS_{ij} refinement matters here too: the Apartment's walls are one connected obstacle, so "two different obstacles" finds nothing until the back pointer is the *obstacle pixel* of origin and a pixel is called a ridge pixel where a neighbour's origin lies far from its own with a different direction — the discrete form of two distinct closest points with distinct gradients. The algorithm section says how the port does it and what it costs.

8.4 The algorithm

ALGORITHM `ROTATIONAL PLANE SWEEP -- Choset Alg. 5` COST `O(n log n) per vertex with S a balanced tree (the port: a sorted array, O(n) per update); O(n^2 log n) for the graph`

in a set of vertices {v_i} whose edges do not intersect, and a vertex v
out the subset of {v_i} within line of sight of v

1: **for** each vertex v_i: compute α_i, the angle from the horizontal axis to vv_i
2: create the vertex list E, the α_i sorted in increasing order
3: create the active list S: the edges crossing the horizontal half-line from v, sorted by distance
4: **for all** α_i **do**
5: **if** v_i is visible to v — $|vv_i|$ is less than the distance to the nearest edge of S not incident to v_i, and l does not enter the obstacle at v — **then** add (v, v_i) to the graph
6: **if** v_i is the beginning of an edge not in S **then** insert it into S by distance along l
7: **if** v_i is the end of an edge in S **then** delete it
8: **end for**

Reduced graph: keep (v_i, v_j) iff the line through it is tangent at both ends — at a polygon vertex, both polygon neighbours on one side of the line (free nodes impose nothing). *Brute force*, for the check: (v_i, v_j) is an edge iff no polygon edge not incident to either endpoint properly crosses the segment and its midpoint is not strictly inside any polygon; the second clause is what rejects a chord of a convex obstacle.

ALGORITHM `GVD BY RETRACTION AND TRACING -- Choset eq. (5.2), §5.2.5 (sensor-based construction)` `COST retraction: O(#obstacles) per step; tracing: one predictor-corrector step per sample`

in a distance query giving d_i and ∇d_i for every obstacle (geometry here; Chapter 9 feeds it from range data); a window
out landing points on SS_ij, traced edges F_ij with their meet and boundary points

1: **access** (`retract`): from q, step along ∇d_i for the nearest i; when the nearest obstacle changes to j and $\angle(\nabla d_i, \nabla d_j) > 0$, bisect $d_i - d_j$ on the last step to 10^{-12} — a point of SS_{ij}; a tie with equal gradients is skipped
2: **seed** (`projectToGvd`): for lattice points, Newton along $\nabla d_i - \nabla d_j$ onto S_{ij} for the two nearest sites; keep the point only if they are still the two nearest there and the gradients differ
3: **trace** (`traceEdge`): Chapter 3's `traceCurve` on $G = d_i - d_j$ with tangent $(\nabla d_i - \nabla d_j)^{\perp}$, both ways from the seed, until a third site ties (polish the **meet point** by Newton on $(d_i - d_j,\ d_i - d_k)$), D vanishes (a **boundary point**), or the window is left
4: **collect** (`exactGvd`): skip seeds within a step of an edge already traced for the same pair; each edge carries its samples, its ends, and its analytic

type per sample — *line* when both closest features are of the same kind, *parabola* when one is a vertex and the other an edge

5: **ride**: `Roadmap` is a `Graph` for Chapter 6's A*; a query is `retract(q_s)`, `retract(q_g)`, A*

What this is and is not. The Rust crate delegates the exact Voronoi diagram of segments to a library kernel (Boost.Polygon's algorithm); no such kernel exists in TypeScript, and the port does not pretend to be one. Tracing is exact to the tracer's tolerance — 4.8×10^{-13} against the slab's parabolas — and comfortable for a dozen sites; it is Choset's sensor-based construction with the sensor replaced by geometry, which is also why Chapter 9 can reuse it unchanged.

ALGORITHM `GRID GVD -- Choset §5.2.5 (brushfire method), with the SS_ij refinement in grid form` COST `O(#cells) brushfire; O(#cells log #cells) feature transform; A* on the ridge`

in an occupancy grid; Chapter 7's brushfire; for connected obstacles, a Euclidean feature transform

out ridge cells, a ridge graph for A*, an access map from any free cell

1: **separate obstacles** (the slab, a room with furniture): Chapter 7's `brushfire` and its `ridge` — pixels two fronts from different obstacle components reach

2: **connected obstacles** (the Apartment's walls): propagate each pixel's nearest obstacle *pixel* under the Euclidean distance (Danielsson's vector distance transform, a Dijkstra over 8-neighbours keyed on the true distance to the propagated origin); pixel p is a ridge pixel when a neighbour's origin lies at least $\gamma = 2.5$ cells from p's, the two directions differ by more than 20°, and p is no nearer the obstacles than the neighbour

3: **ridge graph**: 8-adjacent ridge cells with Euclidean step costs; `thinRidge` (one Zhang–Suen pass, heuristic) when a one-pixel skeleton is wanted

4: **access**: climb the brushfire labels to a ridge cell; **ride**: A* on the ridge graph; **depart**: the goal's climb, reversed

Why not brushfire origins for step 2: brushfire's labels are a grid metric and its origin pixels are tie-broken by queue order, so along a long wall two neighbouring pixels can carry origins many cells apart — a false ridge. The Euclidean features are the quantity Chapter 7 compared brushfire against, and the ridge they give is within 0.11 m of every exact landing point in the

> Apartment check. *Grid ridges are jagged* (Choset Problem 14): on the slab at 0.25 m the 8-point ridge passes 0.18 m from the exact meet point and the 4-point ridge 0.40 m, and the grid meet cells sit at $y \approx 2$ (chessboard) and $y \approx 1.1$ (Manhattan), bracketing $y = 1.5$.

8.5 Implementation in Rust

The `roadmap` crate is introduced here; Chapter 9 adds `gvg` as a submodule. Its spine is a graph embedded in the free space together with the access map of Def. 5.0.2.

RUST **crates/roadmap/src/lib.rs**

```rust
use nalgebra::Point2;
use petgraph::graph::{NodeIndex, UnGraph};

/// A roadmap: a graph embedded in Q_free with the geometry of each edge sampled, plus the
/// access map of Def. 5.0.2. "Ride" is `search::astar` on `graph` -- nothing is re-implemented.
pub struct Roadmap {
    pub graph: UnGraph<Point2<f64>, f64>,
    /// One polyline per edge, in edge-index order: a chord for the visibility graph, a traced curve for the GVD.
    pub polylines: Vec<Vec<Point2<f64>>>,
}

/// Accessibility as a trait: where does q get on, and along what path?
pub trait Access {
    fn access(&self, q: Point2<f64>) -> Option<(NodeIndex, Vec<Point2<f64>>)>;
}

pub struct QueryResult { pub path: Vec<Point2<f64>>, pub length: f64, pub min_clearance: f64 }

/// Get on, ride, get off. `None` when start and goal land in different roadmap components --
/// which, for a roadmap, means they are in different components of Q_free.
pub fn query(rm: &Roadmap, acc: &dyn Access, q_s: Point2<f64>, q_g: Point2<f64>, d: &dyn Fn(Point2<f64>) -> f64) -> Option<QueryResult> {
    let (ns, on) = acc.access(q_s)?;
    let (ng, off) = acc.access(q_g)?;
    let ride = search::astar_graph(&rm.graph, ns, ng, |n| (rm.graph[n] - rm.graph[ng]).norm())?;
    let mut path = on;
    for w in ride.path.windows(2) {
        let e = rm.graph.find_edge(w[0], w[1]).unwrap();
        path.extend(rm.polylines[e.index()].iter().skip(1));
    }
    path.extend(off.into_iter().rev().skip(1));
    let length = path.windows(2).map(|w| (w[1] - w[0]).norm()).sum();
    let min_clearance = path.iter().map(|p| d(*p)).fold(f64::INFINITY, f64::min);
    Some(QueryResult { path, length, min_clearance })
}
```

The sweep keeps S in a `BTreeMap` keyed by distance along the current ray — the balanced tree the $O(n \log n)$ bound assumes.

RUST **crates/roadmap/src/visibility.rs**

```rust
use std::collections::BTreeMap;
use ordered_float::OrderedFloat;

/// Choset Alg. 5 for one vertex `v`. `polys` are the C-obstacles (general position assumed).
pub fn rotational_sweep(polys: &[cspace::Polygon], v: Point2<f64>) -> Vec<Point2<f64>> {
    let (nodes, edges) = index_vertices_and_edges(polys);
    // Step 1-2: angles alpha_i in [0, 2pi), sorted (ties broken by distance).
    let mut events: Vec<(f64, usize)> = nodes.iter().enumerate()
        .filter(|(_, n)| n.p != v)
        .map(|(i, n)| (angle_from(v, n.p), i)).collect();
    events.sort_by(|a, b| a.0.partial_cmp(&b.0).unwrap());
    // Step 3: S = edges crossing the horizontal half-line, keyed by crossing distance.
    let mut s: BTreeMap<OrderedFloat<f64>, usize> = edges.iter().enumerate()
        .filter(|(_, e)| !incident(e, v) && crosses_half_line(e, v))
        .map(|(i, e)| (OrderedFloat(ray_distance(v, 0.0, e)), i)).collect();
    let mut visible = Vec::new();
    for (alpha, i) in events {
        let vi = nodes[i].p;
        let d = (vi - v).norm();
        // Step 5: nearest crossed edge not incident to v_i; the sweep line must not enter the obstacle at v.
        let nearest = s.iter().filter(|(_, &e)| !incident_to(&edges[e], i))
            .map(|(_, &e)| ray_distance(v, alpha, &edges[e])).fold(f64::INFINITY, f64::min);
        if d < nearest - 1e-9 && !enters_obstacle_at(polys, v, vi - v) && !midpoint_buried(polys, v, vi) {
            visible.push(vi);
        }
        // Steps 6-7: an incident edge already in S ends here; otherwise it begins. Key by the
        // distance just past alpha so two edges starting at v_i are ordered the way the beam meets them.
        for &e in &nodes[i].incident {
            if let Some(k) = s.iter().find(|(_, &x)| x == e).map(|(k, _)| *k) { s.remove(&k); }
            else { s.insert(OrderedFloat(ray_distance(v, alpha + 1e-7, &edges[e])), e); }
        }
    }
    visible
}

/// Keep only edges on supporting or separating lines: tangent (both polygon neighbours on one
/// side) at both ends. At a reflex vertex nothing is tangent -- the local-convexity test of sec.5.1.1.
pub fn reduced(polys: &[cspace::Polygon], g: &Roadmap) -> Roadmap { /* filter edges by tangent_at(a, b) && tangent_at(b, a) */ filter_tangent(polys, g) }
```

Tracing the GVD is Chapter 3's tracer on $G = d_i - d_j$; a meet point is two equations in two unknowns.

RUST **crates/roadmap/src/gvd/retract.rs · src/gvd/mod.rs**

```rust
/// Eq. (5.2): follow ?D until two obstacles tie with distinct gradients (SS_ij). A tie with
/// equal gradients -- collinear wall pieces beyond their shared endpoint -- is not the GVD and is
/// skipped. The crossing is bisected so the landing point is on S_ij to 1e-12.
pub fn retract(dist: &dyn cspace::DistanceQuery<2>, q0: Point2<f64>, step: f64, bounds: Aabb) -> Retraction {
    let mut q = q0; let mut path = vec![q0];
    let mut ds = sorted(dist.distances(&q));
    for _ in 0..MAX_STEPS {
        let (i, g) = (ds[0].index, ds[0].grad);
        let next = q + step * g;
        if !bounds.contains(next) { return Retraction::left_window(path); }
        let ds_next = sorted(dist.distances(&next));
        let j = ds_next[0].index;
        if j != i && angle(dist.grad_of(i, &q), dist.grad_of(j, &q)) > GRAD_TOL {
            let land = bisect(|t| { let p = q.lerp(&next, t); dist.d_of(i, &p) - dist.d_of(j, &p) }, q, next, 60);
            path.push(land);
            return Retraction::landed(path, [i, j]);
        }
        q = next; ds = ds_next; path.push(q);
    }
    Retraction::budget(path)
}

/// A meet point F_ijk: Newton on F(q) = (d_i - d_j, d_i - d_k) with J = [?d_i - ?d_j; ?d_i - ?d_k].
/// Rejected unless i, j, k are the three *closest* sites at the solution.
pub fn meet_point(sites: &[Site], i: usize, j: usize, k: usize, seed: Point2<f64>) -> Option<MeetPoint> { newton_2x2(sites, [i, j, k], seed, 60, 1e-12) }

/// Sensor-based edge following (sec.5.2.5): Chapter 3's predictor-corrector on G = d_i - d_j, tangent
/// (?d_i - ?d_j)^? -- perpendicular to the chord between the two closest points (Fig. 5.13).
pub fn trace_edge(sites: &[Site], i: usize, j: usize, q0: Point2<f64>, turn: Turn, step: f64, bounds: Aabb) -> (Vec<Point2<f64>>, EdgeEnd) {
    let g = |q: Point2<f64>| sites[i].distance(q).d - sites[j].distance(q).d;
    let dg = |q: Point2<f64>| sites[i].distance(q).grad - sites[j].distance(q).grad;
    let stop = |q: Point2<f64>| !bounds.contains(q) || clearance(sites, q) < step || third_site_ties(sites, i, j, q);
    let tr = bugs::trace_curve(&g, &dg, q0, turn, bugs::TraceParams { step, newton_iters: 4, tol: 1e-12, ..Default::default() }, Some(&stop));
    (tr.points.clone(), classify_end(sites, i, j, &tr))
}
```

The grid GVD reuses Chapter 7's brushfire for separate obstacles and adds the Euclidean feature transform for connected walls.

RUST **crates/roadmap/src/gvd/grid.rs**

```rust
/// Pixel p is on the grid GVD when a neighbour's nearest obstacle pixel lies >= gamma cells from p's,
/// in a direction differing by more than `min_angle`, and p is no nearer the obstacles than the
```

```rust
/// neighbour. Features come from a Euclidean vector distance transform (Dijkstra keyed on the
/// true distance to the propagated origin pixel); brushfire's grid-metric origins are tie-broken
/// by queue order and would call long straight walls ridges.
pub fn ridge_from_features(occ: &cspace::Raster, ft: &FeatureTransform, gamma: f64, min_angle: f64) -> Vec<bool> {
    let mut mask = vec![false; occ.len()];
    for p in occ.free_pixels() {
        let o = ft.origin[p];
        for q in occ.neighbors8(p).filter(|&q| !occ.is_occupied(q) && ft.origin[q] != o && ft.dist2[q] <= ft.dist2[p]) {
            let oq = ft.origin[q];
            if occ.pixel_distance(o, oq) < gamma { continue; }
            if occ.direction_angle(p, o, oq) < min_angle { continue; }
            mask[p] = true; break;
        }
    }
    mask
}
```

8.5.1 The worked example, and its printed output

RUST **crates/roadmap/examples/three_obstacle_gvd.rs · sweep_table.rs**

```rust
fn main() {
    // The slab: pegs at (0, 0) and (4, 0), a wall along y = 4, a window [-2, 6] * [-1, 4].
    let sites = [Site::point(0.0, 0.0), Site::point(4.0, 0.0), Site::segment((-12.0, 4.0), (16.0, 4.0))];
    let gvd = exact_gvd(&sites, Aabb::new(-2.0, -1.0, 6.0, 4.0), 0.02);
    for m in &gvd.meet_points { println!("meet point ({:.3}, {:.3}) clearance {:.3}", m.q.x, m.q.y, m.clearance); }
    for e in &gvd.edges {
        let kind = if e.types.iter().all(|t| *t == EdgeType::Parabola) { "parabola" } else { "line" };
        println!("edge {:?}: {} ({} samples, length {:.2})", e.sites, kind, e.points.len(), e.length);
    }
    let t = tangent_at(&sites, 0, 2, Point2::new(2.0, 1.5));
    println!("tangent along P1|W at the meet point: slope {:.3}", t.y / t.x);
    let occ = cspace::Raster::from_sites(&sites, 0.25, Aabb::new(-2.0, -1.0, 6.0, 4.0));
    for conn in [Conn::Four, Conn::Eight] {
        let d = potential::brushfire(&occ, conn);
        let ridge = RidgeGraph::new(&d);
        let c = ridge.nearest(occ.cell_at(2.0, 1.5));
        println!("{conn:?}: nearest ridge cell to (2, 1.5) at {:?}, {} ridge cells", occ.center(c), ridge.len());
    }
    // Fig. 5.8 / Table 5.1.
    let (polys, v) = fig_5_8();
    let sw = rotational_sweep_trace(&polys, v);
    println!("S0 = {}", sw.initial_s_names());
    for row in &sw.events { println!("{:<3} {:<22} {}", row.vertex, row.s_names(), row.actions()); }
}
```

```
meet point (2.000, 1.500) clearance 2.500
edge [0, 2]: parabola (210 samples, length 4.18)      y = 2 - x2/8,  x <= 2
edge [0, 1]: line (127 samples, length 2.51)          x = 2,  y < 1.5
edge [1, 2]: parabola (210 samples, length 4.17)      y = 2 - (x-4)2/8,  x >= 2
tangent along P1|W at the meet point: slope -0.500
Four: nearest ridge cell to (2, 1.5) at (2.125, 1.125), 64 ridge cells
Eight: nearest ridge cell to (2, 1.5) at (2.125, 1.625), 86 ridge cells
S0 = {E4, E2, E8, E6}
v3  {E4, E3, E8, E6}      Delete E2. Add E3.
v7  {E4, E3, E8, E7}      Delete E6. Add E7.
v4  {E8, E7}              Delete E3. Delete E4.  ADD (v, v4)
v8  {}                    Delete E7. Delete E8.  ADD (v, v8)
v1  {E4, E1}              Add E4. Add E1.        ADD (v, v1)
v5  {E4, E1, E8, E5}      Add E8. Add E5.
v2  {E4, E2, E8, E5}      Delete E1. Add E2.
v6  {E4, E2, E8, E6}      Delete E5. Add E6.
```

`meet_point_is_2_1p5` asserts the meet point to 10^{-9}, the clearance, the tangent slope -0.5, that the traced samples satisfy the two parabola equations and $x = 2$ to 10^{-9}, and that the 8-point grid ridge passes within one cell of $(2, 1.5)$. `sweep_table_matches_fig_5_8` asserts the initial list, the event order, every row of S, and that the added edges are exactly $(v, v_4), (v, v_8), (v, v_1)$. `sweep_equals_brute_force` compares the sweep with the $O(n^3)$ definition at every vertex of fifty seeded polygon fields. `apartment_roadmaps` builds both roadmaps for disc-Rusty and answers fifty seeded queries, logging mean length and mean minimum clearance — 6.68 m and 0.22 m for the visibility graph, 10.70 m and 0.48 m for the GVD in the port's run — and asserting only the inequalities: the visibility path is never longer, the GVD path never less clear. The TypeScript port in `web/lib/roadmap/` runs the same ten checks, and every widget on this page is that port.

> **NOTE**
>
> Two things the port had to learn about grids. First, a wall lying exactly on the window's edge — the Apartment's shell — vanishes from a raster whose occupancy test is "centre within half a cell", because the centres are exactly half a cell away; with the shell gone the free space is unbounded, the feature transform is off by metres along the border, and the "GVD" is nonsense. The inflation is now a hair over half a cell. Second, the brushfire ridge of Chapter 7 names *obstacle components*, and the Apartment is one component; the GVD of a single nonconvex obstacle is empty until SS_{ij} asks for two distinct closest points, and the grid needs the Euclidean feature transform to ask it reliably. Neither is a bug in Choset; both are places where the discrete object needs the same refinement the continuous definition did.

8.6 Putting it together: fifty queries, two highways

The integration lab is the hook run to statistics. Both roadmaps are built once for disc-Rusty in the Apartment: the visibility graph on the walls inflated by 0.22 m — 150 nodes, built by the sweep — and the GVD from the 0.1 m raster's feature ridge, 1,592 cells in one connected component. Fifty seeded start–goal pairs at least two metres apart are answered on each: retract, ride, depart. Every pair is answered on both. The visibility path is shorter in every case and 6.68 m on average; the GVD path is 10.70 m on average and its minimum clearance, 0.48 m, is more than twice the visibility path's 0.22 m — which is Rusty's radius, exactly, because the shortest path touches the inflated walls. The check pins the means to two decimals and asserts the two inequalities pair by pair.

Reach's half of the lab is the GVD of its configuration space: the Chapter 4 raster of the Workbench C-obstacle on T^2, run through the same feature transform with wrap-around adjacency, gives the skeleton of the free torus — the curves along which the arm stays as far from the block, the post and the shelf as it can in *joint* space. Exercise 6 builds it; Chapter 10 uses it.

Three pointers close the chapter. The planar GVD is one-dimensional because one equation in two unknowns cuts out a curve; in $\mathbb{R}^3$ the same equation cuts out two-dimensional *sheets*, and the roadmap must be the intersection of sheets — the generalized Voronoi graph of Chapter 9, which also builds the GVD from range data with the tracer this chapter already used. Chapter 10's Morse decomposition takes the definition box literally: the cells of a coverage plan are the pieces between critical values of a sweep function. And Chapter 11 will call "sample near the GVD" medial-axis sampling, and will state the roadmap contract one more time, for a graph built by throwing darts.

8.7 Exercises

1 FOUNDATION •• **Connectivity in the plane, and a counterexample in space**

Prove that the visibility graph is connected within each connected component of Q_{free} (Choset Ch. 5, Problem 1): show every point of the component sees some vertex, and that the obstacle edges link the vertices of one obstacle while a line of sight between two obstacles exists whenever the component is connected. Then give an example in $\mathbb{R}^3$ with one box obstacle where the shortest path from start to goal bends on an *edge* of the box and is therefore not a path in the visibility graph of its vertices (Problem 2).

HINTS

<>For the plane: from any free point, sweep a ray until it hits an obstacle; which vertex does the hit edge give you?</> <>For two obstacles in the same component, take a shortest path between a vertex of each and apply Derivation 1 to it.</> <>In space, put the start and goal on opposite sides of a box, offset so the straight line pierces a face; the shortest detour wraps a box edge at a point that is not a vertex.</>

2 **FOUNDATION** •• **The preimage theorem on a cone**

For $f(x, y, z) = x^2 + y^2 - z^2$ (Choset Problem 18): for which c is $f^{-1}(c)$ a manifold, of what dimension, and when is it connected? Explain why $c = 0$ fails, and relate the failure to the definition of SS_{ij}: what plays the role of $\nabla d_i = \nabla d_j$?

WORK IT OUT

Along the GVD edge P1 | W of the slab, at the meet point (2, 1.5), what is | ∇d_P1 − ∇d_W |, the norm of the differential of G = d_P1 − d_W? (Three decimals.)

Answer. 1.789 $\nabla d_{P_1} = (2, 1.5)/2.5 = (0.8, 0.6)$ and $\nabla d_W = (0, -1)$, so $DG = (0.8, 1.6)$ with norm $\sqrt{0.64 + 2.56} = \sqrt{3.2} \approx 1.789$ — comfortably nonzero, so 0 is a regular value there and the edge is a one-manifold with tangent $\perp (0.8, 1.6)$, slope -0.5. On the cone, $c = 0$ fails because $Df = (2x, 2y, -2z)$ vanishes at the apex, which lies on the level set; the apex is to the cone what a point with $\nabla d_i = \nabla d_j$ is to S_{ij}.

3 **CONCEPTUAL** • **Predict the meet point, then move the wall**

PREDICT FIRST

In the Voronoi Carver, drag the wall from y = 4 to y = 6 (or set it in the disclosure). Before releasing: where does the meet point of the two pegs and the wall go?

- To (2, 2.5)
  Halfway between the pegs' level and the wall sounds right but is not: the meet point is equidistant to a peg and the wall, $\sqrt{(4 + y^2)} = 6 - y$, which gives $12y = 32$, $y = 8/3 \approx 2.667$, not 2.5.
- ✓ To (2, 8/3 ≈ 2.667), with clearance 10/3
  On x = 2 the distance to either peg is $\sqrt{(4 + y^2)}$ and to the wall is $6 - y$; equating and squaring gives $4 + y^2 = 36 - 12y + y^2$, so $y = 32/12 = 8/3$, and the clearance is $6 - 8/3 = 10/3 \approx 3.333$.
- It stays at (2, 1.5); only the parabolas move
  The meet point is on all three edges at once, so it must move with the wall: the vertical edge x = 2 just gets longer.

Verify in the widget, then predict what happens as the wall goes to infinity: the parabolas flatten, the meet point climbs without bound, and the diagram tends to the single vertical bisector of the pegs — the ordinary Voronoi diagram of two points.

4 CONCEPTUAL •• **Which loop survives?**

In the Retract Candy, drop the green loop around the kitchen island (click just beside the orange one) and another in the middle of room A. Predict which keeps a nonzero winding number about the island after the retraction and why the "winding about the island" counters for a loop can never disagree between $t = 0$ and $t = 1$ under gradient ascent — but can under "retract to a point". Then explain, in one sentence each, why the straight-line map is still a retraction, and what property it lacks.

HINTS

<>A winding number is an integer that depends continuously on the loop as long as the loop never touches the point it winds about. What would have to happen for it to change?</> <>Gradient ascent of D never decreases D, so no point of a loop can enter an obstacle along the way. A straight line to a far point can.</>

5 PRACTICAL •• **Reduced graphs for nonconvex polygons**

Implement `reduced` for nonconvex polygons using the local-convexity test of §5.1.1: at a reflex vertex no line is tangent, so every edge there is dropped, while at a convex vertex the tangency test is the one for convex obstacles. Property-test on 100 seeded worlds that contain L-shaped and U-shaped obstacles that the shortest-path length from start to goal on the reduced and full graphs agree, and record how many edges the reduction removes.

6 PRACTICAL ••• **The sensor-based GVD, end to end**

Implement Choset's §5.2.5 construction with Rusty's Chapter 3 range sensor: access the GVD by `retract` driven by the local minima of the scan (Chapter 7's `SensorDistance`), trace edges with `trace_edge` on $G = d_i - d_j$ with the two smallest local minima standing in for d_i and d_j, detect a meet point as "a sudden change in one of the two closest obstacles", build the graph incrementally — mark the direction you came from, explore every new edge from each meet point, turn around at boundary points — and compare the edge count with `exact` on a room of the Apartment. Chapter 9 does this in $\mathbb{R}^3$; the planar version is three pages.

8.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

Chapter 5, §5.0–5.2, is this chapter's source: Definition 5.0.2, the visibility graph and Algorithm 5 with Fig. 5.8 and Table 5.1, the GVD through F_ij and SS_ij, Lemma 5.2.1, the deformation-retract argument, the preimage theorem (Theorem 5.2.2) and the three constructions of §5.2.5. `link`

Lozano-Pé rez, T. and Wesley, M. A. (1979). An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles. *Communications of the ACM 22(10), 560–570*.

The visibility graph as a motion-planning roadmap, with the shortest-path argument of Derivation 1 and the observation that it fails in three dimensions. `doi:10.1145/359156.359164`

Lozano-Pé rez, T. (1983). Spatial Planning: A Configuration Space Approach. *IEEE Transactions on Computers C-32(2), 108–120*.

The C-obstacles on which the visibility graph lives: the inflated walls of this chapter's Apartment are its Minkowski sums for a disc. `doi:10.1109/TC.1983.1676196`

Choset, H. and Burdick, J. (2000). Sensor-Based Exploration: The Hierarchical Generalized Voronoi Graph. *International Journal of Robotics Research 19(2), 96–125*.
The GVD and its three-dimensional successor as roadmaps built from range data by gradient ascent and edge tracing — the sensor-based construction of §5.2.5 in full, and Chapter 9's subject. doi:10.1177/02783640022066770

Aurenhammer, F. (1991). Voronoi Diagrams — A Survey of a Fundamental Geometric Data Structure. *ACM Computing Surveys 23(3), 345–405*.
The survey behind the polygonal construction: bisectors of points, lines and segments, the size bounds, and the sweep and divide-and-conquer algorithms that the exact kernel the Rust crate delegates to implements. doi:10.1145/116873.116880

Danielsson, P.-E. (1980). Euclidean Distance Mapping. *Computer Graphics and Image Processing 14(3), 227–248*.
The vector distance transform: propagating each pixel's nearest obstacle pixel under the Euclidean distance, which the grid GVD uses to apply the SS_ij refinement to the Apartment's connected walls. doi:10.1016/0146-664X(80)90054-4

Hatcher, A. (2002). Algebraic Topology. *Cambridge University Press*.
Chapter 0 defines homotopy, retraction and deformation retraction exactly as used here, and §1.1 the fundamental group; the standard reference for the fact that a deformation retraction induces an isomorphism on π_1. link

CHAPTER 9

Roadmaps II: Sensor-Based Roadmaps and Silhouettes

CLASSICAL PLANNERS — SEARCH, POTENTIALS, ROADMAPS, CELLS · Advanced · 70 min

> *Since $\nabla G(q)$ is a function of distance gradients, the planner can compute $\nabla G(q)$ solely from range sensor information.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), §5.3.3*

Chapter 8 ended on a cliff. In the plane, "equidistant to the two nearest obstacles" is a curve, and the curves form a roadmap. In $\mathbb{R}^3$ the same condition is one equation in three unknowns and cuts out a *sheet* — and a union of sheets is not a roadmap: the problem is only one dimension smaller than before. This chapter climbs down two ways, and they turn out to be one idea. The first way intersects more equations: equidistance to *three* obstacles in $\mathbb{R}^3$ is a curve again, the **generalized Voronoi graph**, provided the sheets cross cleanly — *transversally*. The curves need not connect, and patching them, the hierarchical GVG, is the real work. The second way sweeps a slice through the space and tracks the extrema of a function on it: **Canny's silhouette curves**, recursing wherever the trace breaks. Both obey Chapter 8's Morse vocabulary: critical points are exactly where a roadmap can change connectivity.

The practical surprise is that a robot can build a roadmap *without a map*. The quantity $d_1 - d_2$ is something a range sensor measures directly, and so is its gradient; a control law that drives $\Gamma = \frac{1}{2}G^\top G$ to zero — a Lyapunov function, the book's first stability proof — lets Rusty trace the generalized Voronoi graph with no predictor, no corrector, and no equation of the curve. Exploration is then depth-first search over a graph the robot is still drawing.

The price of exactness closes the chapter. Canny's roadmap is complete and its running time is singly exponential in the dimension; the generalized mover's

problem is PSPACE-hard. Both facts are why Part III gives completeness up for speed. Everything on this page runs: the Apartment's graph is built from a simulated range ring and lands on all twenty-six meet points of Chapter 8's exact diagram, the sphere's silhouette is its equator to 10^{-6}, and the chapter's one-step Lyapunov example prints $\dot{\Gamma} = -0.7918$ to four decimals in both the TypeScript port and the Rust.

AFTER THIS CHAPTER YOU CAN

- Define the generalized Voronoi graph in R^m as the set of points equidistant to m closest obstacles with distinct gradients, and prove its edges are one-dimensional from the preimage theorem plus the transversality assumption
- Explain why the GVG is generically disconnected in three dimensions — no one-dimensional deformation retract of a punctured 3-space exists — and how second-order edges, periods and the projected gradient patch it into the hierarchical GVG
- State the definition of a Lyapunov function, derive $\dot{\Gamma} = \beta G^{\mathsf{T}} G$ for the tracing law $\dot{q} = \alpha v + \beta(DG)\dagger G$, and say exactly where the derivation stops holding
- Build a roadmap of an unknown environment from a range ring: access, trace, home onto meet points, and run depth-first search over a graph the robot is still drawing
- Compute a silhouette as the critical set of a projection (Lemmas 5.5.1–5.5.2 and the slice lemma), locate critical slices, and connect the pieces by recursion — Canny's roadmap — and its cousin the opportunistic path planner with its generalized gradient
- Quote Canny's singly exponential bound and the PSPACE-hardness of the generalized mover's problem, and explain why they motivate the sampling-based planners of Part III

ASSUMED

- The range ring, ρ, D(x) and ∇D, and the predictor–corrector curve tracer (Chapter 3)
- Manifolds, tangent spaces, the differential and codimension (Chapter 5); A* on a graph (Chapter 6)
- The distance function D, its nonsmoothness, and the local minima of a scan as distances to obstacles (Chapter 7)
- The GVD, SS_ij, the preimage theorem, deformation retracts and the Morse definition box (Chapter 8)

9.1 The problem: a map nobody drew

Drop Rusty into the Apartment with no floor plan, only its range ring. Twenty seconds into the replay — under three minutes of simulated driving — a graph of the whole floor exists: every room, every doorway, the corridor, drawn by the robot driving along it.

> Rusty (orange) in an Apartment it has never seen. It drives away from the nearest wall until two range readings tie, then follows the curve along which they stay tied, drawing it in blue; where a third reading ties it has found a meet point and records how many curves leave it. The purple rays are the two shortest returns of the ring. The replay runs the same code as the GVG Explorer below, at ten simulation steps per frame.

Nothing in that animation knows where a wall is. Each step the robot reads its 360 rays, finds the local minima of the range — the distances d_i to the nearby obstacles and, from their bearings, the gradients ∇d_i — and commands one velocity. The blue curve is Chapter 8's generalized Voronoi diagram; the robot does not compute it, it *lives on it*, because staying on it is a condition the sensor can check: two of the ranges are equal. The run ends when every curve leaving every meet point has been driven: 27 meet points, 33 boundary points where a curve runs into a corner, 62 edges and 103.4 m of roadmap after 205.0 m of driving.

Two questions organize the rest of the chapter. *Why does this work* — what makes a velocity law converge onto a curve it cannot see? And *what happens in three dimensions*, where "two readings tie" is a surface and the robot would be wandering on sheets? The first answer is a Lyapunov function. The second is a story about transversality, disconnection, and slices — and it ends in a complexity bound that sends us to sampling.

9.2 Building intuition

9.2.1 Driving the graph

INTERACTIVE · w9.1

GVG Explorer

A roadmap is a sensor quantity: three ranges tie, and the robot is on it. Exploration is depth-first search over a graph the robot is still drawing.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch09-roadmaps-ii#w9.1

Watch the explorer in phases. **Access**: from wherever it starts, Rusty backs

away from the nearest return — gradient ascent of D, exactly Chapter 8's Lemma 5.2.1 — until a second return is as short as the first. **Trace**: now it holds the difference of the two shortest returns at zero while moving perpendicular to the chord between their two closest points. The Γ sparkline under the floor plan is the square of that difference: it spikes when the robot is pushed off the curve (the parabolic edges near door jambs curve faster than the law can follow at a gentle gain) and decays back. **Home**: when a third return closes in, a meet point is near; the robot switches law and converges on the point where all three tie. **Branch**: each meet point in the plane has three departing edges. The hollow node's number counts those not yet driven. **Backtrack**: when a node has none left, the robot drives along edges it already knows to the nearest node that still has one — depth-first search over a graph whose nodes are being discovered by the search itself. **Terminate**: when no node has an undriven edge, the graph is complete.

The faint blue curves underneath are Chapter 8's exact GVD of the same walls, traced from geometry. The sensor's graph lands on them: every meet point the robot homes onto is three-way equidistant by exact wall geometry to 10^{-6}, and all twenty-six exact meet points are found within five centimetres. Two more — in the corridor at $(6.00, 4.30)$ and in the bedroom at $(7.85, 6.79)$ — are genuine meet points that Chapter 8's lattice-seeded tracer missed, which the same exact test confirms. The headline control is β, the stiffness with which the robot is pulled back onto the curve. The misconception this widget kills is the one in the chapter title: *you do not need a map to build a roadmap*. The roadmap is a sensor quantity.

9.2.2 From sheets to curves

In the plane one equation, $d_i = d_j$, leaves one free direction: a curve. In $\mathbb{R}^3$ it leaves two: a sheet. Imagine the Apartment extruded upward into a box-shaped room with a floor and a ceiling. The set of points equally far from the floor and the ceiling is the horizontal plane halfway up; equally far from the floor and the west wall is a tilted plane; and so on. These sheets form Chapter 8's GVD in three dimensions, and it is a perfectly good *two*-dimensional retract — but a planner searching a two-dimensional set has not gained much.

So intersect more. Points equidistant to the floor, the ceiling *and* the west wall lie on two sheets at once, and two planes in $\mathbb{R}^3$ meet in a line. That is a generalized Voronoi graph edge: think of a ball rolling along the room, touching the floor, the ceiling and one wall at once; its centre traces the edge. Where a fourth obstacle ties, four sheets meet at a point — a meet point. Two conditions must hold for the picture to be right: the sheets must actually cross rather than graze, and the resulting curves must connect. The first is transversality; the second, it turns out, fails.

9.2.3 A slice through a doughnut

Canny's idea comes at the same problem from the other side. Sweep a plane through the configuration space and look at what the plane cuts out: a few closed curves. On each curve mark the extreme points in some fixed direction. As the plane moves those marks move, and their traces are curves — *silhouettes* — that a planner can follow. On a torus lying flat, a plane sweeping across it first touches the outer rim, cuts one oval, and then, as it reaches the hole, the oval pinches and splits into two. That moment is a *critical slice*: the silhouette breaks there, and the algorithm repairs it by solving the same problem one dimension down, inside the critical slice. The Silhouette Sweep in the mathematics section plays this out.

9.3 The mathematics

S_{ijk}, SS_{ijk}, F_{ijk}	triple-equidistant set; its subset with pairwise distinct gradients; a GVG edge in R^3 (eq. 5.3)
codim; $T_xM_1 + T_xM_2 = T_xM$	codimension (the GVD has codimension 1, the GVG dimension 1); transversal intersection at x
$F_{kl}\|_{F_{ij}}$, $F_{klp}\|_{F_{ij}}$; $\pi_{TF_{ij}}$	second-order GVG edge and meet point on the sheet F_ij (eq. 5.4); projection onto its tangent space
$G(q)$, DG, $(DG)^\dagger$	constraint vector, e.g. $[\mathrm{d}_1 - \mathrm{d}_2, \mathrm{d}_1 - \mathrm{d}_3]^\mathsf{T}$; its differential; the pseudoinverse $\mathrm{DG}^\mathsf{T}(\mathrm{DG}\,\mathrm{DG}^\mathsf{T})^{-1}$
α, β, $v \in \mathrm{Null}(DG)$; $\Gamma = \frac{1}{2}G^\top G$	tracing gains and unit tangent (eq. 5.5); the Lyapunov function of the tracer
RF_{ij}, RF_{ijk}, J_{ijk}	rod two-equidistant face, rod-GVG edge, junction region (§5.4)
Q_λ, π_1, π_2, π_{12}; $\Sigma(\cdot)$, $D(f,h)$	the slice at λ and coordinate projections; critical set; the stacked differential of Lemma 5.5.2
$\tilde{d}_i(p;\lambda)$, $\tilde{D}$	distance restricted to a slice and its minimum over obstacles (eqs. 5.7–5.8)
$\partial D(q^\star)$, Co, $Z(q^\star)$	generalized gradient, convex hull, index set of the closest obstacles
p, w	number and maximum degree of the polynomials bounding the obstacles (Canny's bound)

9.3.1 The generalized Voronoi graph

DEFINITION — GENERALIZED VORONOI GRAPH (CHOSET §5.3.1, EQ. 5.3)

In $\mathbb{R}^3$ let $S_{ijk} = \{q : (d_i - d_j)(q) = 0 \text{ and } (d_i - d_k)(q) = 0\}$ — the third equation $d_j = d_k$ follows — and $SS_{ijk} = SS_{ij} \cap SS_{ik} \cap SS_{jk}$, the points of S_{ijk} where the three gradients $\nabla d_i, \nabla d_j, \nabla d_k$ are pairwise distinct. A **GVG edge** is $F_{ijk} = \{q \in SS_{ijk} : d_i(q) \le d_h(q)\ \forall h\}$ and the **GVG** is $\bigcup_i \bigcup_j \bigcup_k F_{ijk}$. In $\mathbb{R}^m$ the GVG is the set of points equidistant to m closest obstacles with

distinct gradients; it is one-dimensional, and its edges end at **meet points** (equidistant to $m+1$) and **boundary points**. In the plane the GVG and the GVD coincide.

Note what the definition does *not* say: SS_{ijk} is not $SS_{ij} \cap SS_{ik}$. Transitivity of equality gives $d_j = d_k$ for free, but not $\nabla d_j \neq \nabla d_k$, and all three gradients must differ for the preimage theorem to apply. Choset's Table 5.2 compares the two structures in $\mathbb{R}^m$:

Structure	Dimension	Codimension	Equidistant to	Symbol
GVD	$m-1$	1	2	$F_{i_1 i_2}$
GVG	1	$m-1$	m	$F_{i_1 \ldots i_m}$

DEFINITION — TRANSVERSALITY (CHOSET §5.3.1)

Submanifolds $M_1, M_2 \subset M$ intersect **transversally** if $T_x M_1 + T_x M_2 = T_x M$ at every point x of the intersection. Then codimensions add: $\operatorname{codim}(T_x M_1 \cap T_x M_2) = \operatorname{codim} T_x M_1 + \operatorname{codim} T_x M_2$. Transversal intersections are *stable* — a small perturbation of either manifold leaves an intersection of the same dimension — and a non-transversal intersection becomes transversal after a generic small perturbation. The transversality assumption generalizes the general-position assumption of computational geometry.

The transversality trio (Choset Fig. 5.17). Two lines in the plane meet not at all, at a point, or along a line. Hover a panel to perturb the teal line: the parallel and overlapping pairs change the dimension of their intersection; the crossing pair still meets at a point. The number in each panel is the smallest singular value of the matrix whose rows are the two lines' normals — the transversality margin the GVG code computes for its sheets — and it is zero exactly in the unstable cases.

Two lines in the plane each have codimension one; meeting transversally, their intersection has codimension two — a point. Two planes in $\mathbb{R}^3$ meet transversally in a line, a plane and a line in a point; two lines in $\mathbb{R}^3$ can never meet transversally, because a generic nudge separates them. Two planes in $\mathbb{R}^4$, codimension two each, meet in a point. That arithmetic is the whole content of Table 5.2.

$$G(q) = \begin{bmatrix} d_i - d_j \\ d_i - d_k \end{bmatrix}(q), \qquad DG(q) = \begin{bmatrix} (\nabla d_i - \nabla d_j)^\top \\ (\nabla d_i - \nabla d_k)^\top \end{bmatrix}, \qquad SS_{ijk} \subset G^{-1}(0) \text{ is a 1-manifold where } \operatorname{rank} DG = 2.$$

DERIVATION **GVG edges in $\mathbf{R}^3$ are one-dimensional**

$SS_{ijk} = G^{-1}(0) \cap \{\text{distinct gradients}\}, \quad \dim = 3 - 2 = 1$

Statement (Choset §5.3.1). With $G = [d_i - d_j,\ d_i - d_k]^\top : \mathbb{R}^3 \to \mathbb{R}^2$, the set SS_{ijk} is a one-dimensional manifold, provided the sheets SS_{ij} and SS_{ik} intersect transversally.

Step 1 — each row is nonzero. On SS_{ij} the first row $\nabla d_i - \nabla d_j$ is nonzero by definition (Chapter 8: that is what the extra S in SS means); likewise the second row on SS_{ik}.

Step 2 — rank loss means proportional rows. A 2×3 matrix with nonzero rows loses rank exactly when $\nabla(d_i - d_j) = \alpha\, \nabla(d_i - d_k)$ for some $\alpha \in \mathbb{R}$.

Step 3 — proportional rows are a non-transversal crossing. The rows are the normals of the two sheets. Proportional normals mean equal tangent planes, $T_q SS_{ij} = T_q SS_{ik}$, whose sum is a plane and not $\mathbb{R}^3$: the sheets are tangent there. Under the transversality assumption this does not happen, and if it did, a small perturbation of either sheet would remove it.

Step 4 — the preimage theorem. DG has full rank two on SS_{ijk}, so 0 is a regular value there and SS_{ijk} is a submanifold of dimension $3 - 2 = 1$, with tangent $\ker DG$ — the cross product of the two rows. ■

The numbers. The chapter's synthetic room — a $10 \times 8 \times 3$ box, Choset's Fig. 5.15 — is explored by the same tracer that runs in the Apartment, with exact plane distances in place of a scan. From one start it finds the four meet points $(1.5, 1.5, 1.5)$, $(8.5, 1.5, 1.5)$, $(8.5, 6.5, 1.5)$, $(1.5, 6.5, 1.5)$ — each four-way equidistant to 10^{-9} and of degree four — and eight spokes into the corners, thirteen edges in all. Every edge sample is three-way equidistant to 10^{-6}, and the smallest singular value of DG along the edges is at least $0.87 = \sqrt{3 - \sqrt{5}}$, the value on the rectangle's long edges: comfortably transversal everywhere.

9.3.2 Why the GVG falls apart, and the hierarchical GVG

In the room the GVG is connected. Put a box in the middle of the room — Choset's Fig. 5.19 — and it is not. The outer network is unchanged; but around the box there is a second component, a *halo*: the loop of points equally far from the floor, the ceiling and the box. No edge of the outer network touches it.

DEFINITIONS — GVG CYCLE, SECOND-ORDER EDGE, PERIOD, HGVG (CHOSET §5.3.2, EQ. 5.4)

A **GVG cycle** is a GVG edge homeomorphic to S^1. A **second-order GVG edge** on the sheet F_{ij} is $F_{kl}\big|_{F_{ij}} = \{q : d_i = d_j \le d_k = d_l \le d_h \ \forall h \neq i,j,k,l,\ \nabla d_i \neq \nabla d_j,\ \nabla d_k \neq \nabla d_l\}$: QO_i, QO_j are the closest pair and QO_k, QO_l the second-closest. Second-order edges end at **second-order meet points** $F_{klp}\big|_{F_{ij}}$. A **period** is a closed loop of second-order edges sharing a common second-closest obstacle. The **hierarchical GVG** (HGVG) is the union of the GVG and the second-order GVG.

DERIVATION **The GVG is generically disconnected, and the HGVG repairs it**

$\dot{c} = -\pi_{TF_{ij}} \nabla d_k(c)$ from a period reaches $d_i = d_j = d_k$

Statement (Choset §5.3.2). In a punctured three-dimensional free space there is in general no one-dimensional deformation retract [Choset ref. 63], so the GVG can be disconnected. The two-dimensional GVD is connected; GVG edges lie on the boundaries of its sheets, $F_{ijk} = \partial F_{ij} \cap \partial F_{ik} \cap \partial F_{jk}$; and a path that keeps $d_i = d_j$ while decreasing d_k leads from a period to the missing cycle.

Step 1 — the hole. In the box-in-a-room, the floor–ceiling sheet F_{fc} is the plane halfway up, with a hole where the box is nearer than the floor and the ceiling. The boundary of that hole is the halo: the GVG edge F_{fcb}, a cycle.

Step 2 — the trouble is on sheet boundaries. The GVD is a two-dimensional deformation retract of the free space and therefore connected. If the boundary of every sheet were connected, the GVG would be too. The floor–ceiling sheet's boundary is not: its outer rim belongs to the outer network, its hole to the halo.

Step 3 — draw a planar GVG on the sheet. On F_{fc}, look at which obstacles are *second* closest and where two of them tie. Those are the second-order edges of eq. (5.4) — planar GVG edges drawn on the sheet — and they are one-manifolds by the preimage theorem, because of the gradient conditions on the second line of (5.4).

Step 4 — a period announces a cycle. Around the box the second-order edges whose tied pair includes the box close into a loop: a period. Its common second-closest obstacle, the box, is the clue that a GVG component involving the box exists somewhere inside it.

Step 5 — descend to it. Starting on the period, follow $\dot{c} = -\pi_{T_c F_{ij}} \nabla d_k(c)$: decrease the distance to the box while staying on the sheet. Since $d_k > d_i = d_j$ at the start, the path reaches $d_i = d_j = d_k$ — a GVG edge — unless the projected gradient vanishes first, in which case no such edge exists and the

robot returns to the period. ■

Honesty. Full HGVG connectivity is "tedious and challenging" in Choset's words and is proved in Choset and Burdick [Choset ref. 106]; this book cites it and does not prove it. The HGVG is *retract-like*, not a retract. *The numbers.* The check builds Fig. 5.19 with a unit box at the centre of the $10 \times 8 \times 3$ room. The outer network never touches the box. The explorer, run on the floor–ceiling sheet with the sheet's own nearest-obstacle query, traces nine second-order edges; the four whose tied pair includes the box close into a period around it; descending $-\pi \nabla d_{box}$ from the period reaches $d_{box} = d_{floor} = d_{ceiling} = 1.5$, and from there the halo traces as a closed GVG cycle 13.42 m long, three-way equidistant to 10^{-6}.

9.3.3 Tracing without a predictor: the Lyapunov law

Chapter 3 traced curves with a discrete predictor–corrector: step along the tangent, then Newton back onto the curve. Choset §5.3.3 replaces both by one velocity,

$$\dot{q} = \alpha\, v + \beta\, (DG)^{\dagger} G, \qquad v \in \mathrm{Null}(DG),\ \|v\| = 1, \qquad (DG)^{\dagger} = DG^{\top}\left(DG\, DG^{\top}\right)^{-1},$$

with $G = d_1 - d_2$ in the plane and $G = [d_1 - d_2,\ d_1 - d_3]^{\top}$ in $\mathbb{R}^3$ (eq. 5.5). On the GVG $G = 0$ and the law reduces to αv: move along the tangent. Off it, the second term pulls back — Choset calls α and β "spring constants": α is the speed along the edge, β the stiffness of the return. Both terms need only G and DG, and DG's rows are differences of distance gradients — unit vectors pointing back along the shortest rays. That is the epigraph: the planner can compute ∇G solely from range sensor information.

DEFINITION BOX — LYAPUNOV FUNCTION (INTRODUCED HERE; APPENDIX E HAS THE FULL TREATMENT)

Let $\dot{q} = X(q)$ be a dynamical system and A a closed invariant set — here, the GVG. A C^1 function $\Gamma \geq 0$ defined on a neighborhood U of A is a **Lyapunov function** for A if $\Gamma = 0$ exactly on A and $\dot{\Gamma} = \nabla\Gamma \cdot X < 0$ on $U \setminus A$. Then A is **asymptotically stable**: trajectories that start close enough to A stay close and converge to it. Choset's gloss: think of a Lyapunov function as "an error function whose minimal value is zero". The guarantee is *local* — it holds only where $\dot{\Gamma} < 0$ has been shown.

INTERACTIVE · w9.3

Lyapunov Funnel

The controller does not need the GVG's equation — only G and DG, both read off the range ring — and $\dot{\Gamma} = \beta G^{\mathsf{T}} G$ makes the edge attract.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch09-roadmaps-ii#w9.3

The funnel is the chapter's micro-world: two point obstacles at $(0, 0)$ and $(4, 0)$, whose GVG is the bisector $x = 2$, and Rusty at $(2.5, 1)$. The shading is $\Gamma = \frac{1}{2}G^2$ — a trough along the bisector — and the robot slides down its wall while advancing along its floor. The sparkline plots $\Gamma(t)$ on the curve $\Gamma_0 e^{2\beta t}$, which the derivation below says it must follow; slide β above zero and the robot climbs out instead. The misconception this widget kills: *the controller needs the GVG's equation*. It needs G and DG at the point where the robot is, and nothing else.

DERIVATION **The tracing law is Lyapunov-stable**

\dot\Gamma = \beta\, G^{\top} G \le 0 \text{ for } \beta < 0,\ \text{strict off the GVG}

Statement (Choset §5.3.3, eq. 5.5). Along $\dot{q} = \alpha v + \beta (DG)^{\dagger} G$ with $v \in \mathrm{Null}(DG)$, the function $\Gamma = \frac{1}{2} G^{\mathsf{T}} G$ satisfies $\dot{\Gamma} = \beta\, G^{\mathsf{T}} G$. For $\beta < 0$ it is a Lyapunov function for the GVG wherever $DG\, DG^{\mathsf{T}}$ is invertible, so the robot converges onto the GVG.

Step 1 — the chain rule. $\dot{\Gamma} = G^{\mathsf{T}} \dot{G} = G^{\mathsf{T}} DG\, \dot{q}$.

Step 2 — the tangent term vanishes. $DG\,(\alpha v) = \alpha\, DG\, v = 0$, because v lies in the null space of DG. Moving along the edge does not change G to first order.

Step 3 — the pseudoinverse is a right inverse. $DG\,(DG)^{\dagger} = DG\, DG^{\mathsf{T}} (DG\, DG^{\mathsf{T}})^{-1} = I$, so $G^{\mathsf{T}} DG\, \beta (DG)^{\dagger} G = \beta\, G^{\mathsf{T}} G$.

Step 4 — the sign. $\dot{\Gamma} = \beta\, G^{\mathsf{T}} G = 2\beta\, \Gamma \le 0$ for $\beta < 0$, with equality only where $G = 0$, on the GVG. In fact $\Gamma(t) = \Gamma(0)\, e^{2\beta t}$ exactly along the continuous flow.

Step 5 — the neighborhood. All of this needs $(DG\, DG^{\mathsf{T}})^{-1}$. Near the interior of a GVG edge the rows of DG are independent (Derivation 1), and they stay independent in a neighborhood [Choset ref. 108] — the neighborhood the definition box asks for. ■

Where it stops holding. The argument needs two things: a fixed set of closest obstacles, so that G is smooth, and an invertible $DG\, DG^{\mathsf{T}}$. Both fail at meet points. There a further obstacle ties, the identity of the "closest" ones switches, G is not even continuous across the switch, and the Lyapunov argument has nothing to say; and wherever two gradients become parallel — a non-transversal point — $DG\, DG^{\mathsf{T}}$ is singular and the pseudoinverse does not exist. The tracer's guarantee is local, and at meet points a different

law, homing, takes over. *The sign of v* is the planner's choice (Choset's footnote 4): the null space is a line and the tracer keeps the direction that continues the previous heading. *Sensing* ∇G: fit a codimension-one plane through the n closest points on the n closest obstacles; the tangent is its normal. In the plane, $\nabla d_1 - \nabla d_2$ is perpendicular to the chord through the two closest points, and the tangent is along the chord's perpendicular — Chapter 8's Fig. 5.13.

The chapter's micro-example, by hand

Take the funnel's opening position: sites $c_1 = (0, 0)$, $c_2 = (4, 0)$, Rusty at $q = (2.5, 1)$, gains $\alpha = 1$, $\beta = -1$.

$$
\begin{aligned}
d_1 &= \sqrt{7.25} = 2.6926, \qquad d_2 = \sqrt{3.25} = 1.8028, \qquad G = d_1 - d_2 = 0.8898,\\
DG &= \nabla d_1 - \nabla d_2 = (0.9285,\ 0.3714) - (-0.8321,\ 0.5547) = (1.7605,\ -0.1833),\\
(DG)^{\dagger} &= DG^{\top}/(DG\,DG^{\top}) = (1.7605,\ -0.1833)/3.1333 = (0.5619,\ -0.0585),\\
v &= (0.1036,\ 0.9946) \quad (\perp DG,\ \text{oriented upward}),\\
\beta(DG)^{\dagger}G &= (-0.5000,\ 0.0521), \qquad \dot{q} = \alpha v + \beta(DG)^{\dagger}G = (-0.3964,\ 1.0467),\\
\Gamma &= \tfrac{1}{2}G^2 = 0.3959, \qquad \dot{\Gamma} = \beta G^2 = -0.7918 < 0.
\end{aligned}
$$

The velocity points left, toward $x = 2$, while advancing along the edge. Look at the correction's x-component: exactly -0.5, minus Rusty's offset from the bisector. For two point sites the Newton-like step $(DG)^{\dagger}G$, projected on the line through the sites, is *exactly* the signed distance to the bisector — not approximately, everywhere — and Exercise 1 asks why. The check pins all of these numbers to 10^{-4} and the identity to 10^{-12} at twenty seeded points around twenty seeded pairs of sites. Integrating the law with explicit Euler steps of 0.01 s, Γ never increases, and at $t = 2$ it is within 3 % of $e^{-4}\Gamma_0$, the continuous-time prediction; after 200 steps of 0.05 s the robot is on $x = 2$ to 10^{-3}, having driven up the edge. With $\beta = +1$, Γ more than triples in 0.6 s.

9.3.4 Meet-point homing

DERIVATION **Homing is the same law with an empty null space**

\dot q = \beta\, (DG)^{-1} G \;\to\; \text{the centre of the circle (sphere) through the closest points}

Statement (Choset §5.3.3). With $G = [d_1 - d_2,\ d_1 - d_3]^{\top}$ in the plane (three rows, $[d_1 - d_2, d_1 - d_3, d_1 - d_4]^{\top}$, in $\mathbb{R}^3$), DG is square, $\mathrm{Null}(DG) = \{0\}$, and $\dot{q} = \beta(DG)^{\dagger}G = \beta\, DG^{-1}G$ converges to the meet point.

Step 1 — Derivation 3 with $v = 0$. The computation of $\dot{\Gamma}$ never used v except to kill it; with $v = 0$ it reads $\dot{\Gamma} = \beta\, G^\top G$ unchanged.
Step 2 — $G = 0$ only at the meet point. Two independent equations in the plane, three in $\mathbb{R}^3$, cut out isolated points: the meet points, where $m + 1$ obstacles tie.
Step 3 — the geometry of the step. Linearize each d_i about q: the step $-DG^{-1}G$ goes to the point where the linearized distances agree, the centre of the circle through the three closest points (the sphere through four in $\mathbb{R}^3$). Choset: "The velocity vector points toward the center of this sphere." ■

Higher-order laws come from swapping G: a second-order edge uses $G = [d_i - d_j,\ d_k - d_l]^\top$, a second-order meet point $[d_i - d_j,\ d_k - d_l,\ d_k - d_p]^\top$.
The numbers. On Chapter 8's slab — pegs at $(0, 0)$ and $(4, 0)$, a wall along $y = 4$ — homing from three seeds converges to the meet point $(2, 1.5)$ with clearance 2.5 to 10^{-9}. For the wall the linearization is exact; for the pegs it is first order, so the aim sharpens with distance: from 0.57 m away the homing velocity points within 45° of the meet point, from 0.06 m away within 3°.

9.3.5 Beyond a point robot: the rod

DEFINITIONS — ROD, ROD DISTANCE, R-EDGE, PIECEWISE RETRACT (CHOSET §5.4)

A **rod** is a line segment moving in the plane; its configuration space is $SE(2)$, three dimensional. Distance is measured *in the workspace*: $d_i(q) = \min_{r \in R(q),\ c \in \mathcal{WO}_i} d(r, c)$, which is what sensors on the rod measure. The **rod two-equidistant face** is $RF_{ij} = \{q \in SE(2) : d_i(q) = d_j(q),\ \nabla d_i(q) \neq \nabla d_j(q)\}$, and a **rod-GVG edge** is $RF_{ijk} = RF_{ij} \cap RF_{ik} \cap RF_{jk}$. An **R-edge** is the set of rod placements tangent to the planar point-GVG. The **rod-HGVG**, rod-GVG edges joined by R-edges, is a **piecewise retract**: a union of deformation retracts of *junction regions* J_{ijk}, linked.

The rod-GVG edges play the role of meet points, but each is a whole curve of orientations: a small rod can spin a full turn while staying three-way equidistant, so RF_{ijk} is homeomorphic to S^1 (Choset's footnote 5). The R-edges inherit adjacency from the plane. The chapter keeps the rod to a definition and a check — Exercise 6 builds it — and the check confirms the first claim: a 0.3 m rod centred near the Workbench's widest meet point (clearance 1.539) stays three-way equidistant through all 36 orientations sampled, Newton in (x, y) at fixed θ converging to 10^{-10}, and 130 R-edge placements are sampled from Chapter 8's point-GVD.

9.3.6 Silhouettes: Canny's roadmap

DEFINITIONS — SLICE, CRITICAL SLICE, SILHOUETTE (CHOSET §5.5, §5.5.1)

The **slice** at λ is $Q_\lambda = \pi_1^{-1}(\lambda) = \{q : q_1 = \lambda\}$; sweeping λ sweeps it through Q. On each slice take the extrema of $\pi_2(q) = q_2$ restricted to the free-space boundary in the slice. The traces of those extrema as λ varies are the **silhouette curves**. A **critical slice** is one where the number of silhouette curves changes; its λ is a **critical value**, and the points where a silhouette curve is tangent to the critical slice are **critical points**. On each critical slice the algorithm recurses one dimension down, sweeping along q_2 and extremizing q_3, until no critical points remain or the slice is one-dimensional — and "the roadmap of a one-dimensional set is the set itself". The union of all silhouettes is **Canny's roadmap**.

LEMMA 5.5.1 (LAGRANGE MULTIPLIER) AND LEMMA 5.5.2 (GENERALIZED, CANNY)

If $S = f^{-1}(c) \subset \mathbb{R}^{n+1}$ with $\nabla f \neq 0$ on S, and p is an extremum of a smooth h on S, then $\nabla h(p) = \mu \nabla f(p)$ for some μ. For vector-valued $f : \mathbb{R}^m \to \mathbb{R}^p$ and $h : \mathbb{R}^m \to \mathbb{R}^n$, x is a critical point of $h|_{f^{-1}(\cdot)}$ if and only if the stacked differential $D(f, h)(x)$ — the rows of Df, then the rows of Dh — loses rank. **Slice lemma:** $\Sigma(\pi_{12}|_S) = \bigcup_\lambda \Sigma\big(\pi_2|_{\pi_1^{-1}(\lambda)}\big)$ — the silhouette is the critical set of π_{12}.

INTERACTIVE · w9.2

Silhouette Sweep

The silhouette is not the outline: it is the critical set of a projection, and the roadmap's connectivity changes only at critical slices — where the algorithm recurses.

Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch09-roadmaps-ii#w9.2`

The sweep makes the definitions concrete on three bodies. The **sphere** has one silhouette, its equator — traced as two curves, the maxima and the minima of q_2 — born at $\lambda = -r$ and dying at $\lambda = r$. The **torus** lying flat ($R = 2$, $r = 0.7$) is Choset's Figs. 5.31–5.32: the slice is one closed curve for $|\lambda| > R - r$ and two for $|\lambda| < R - r$, so the critical values are ± 2.7 (birth and death) and ± 1.3 (pinch). Its silhouette is four arcs — the $q_2 > 0$ and $q_2 < 0$ halves of the outer and inner equators — and they are not connected to one another: four components. The linking curves on the two pinch slices join them into one. The **ellipsoid with a bent hole** is Fig. 5.25 simplified: an ellipsoid with semi-axes 3, 2 and 1.5 and a hole of radius 0.7 drilled along q_3 at $q_1 = 0.6$, its axis bending as $y = 0.5z^2$. The bend matters. With a straight hole the wall is a cylinder, q_2 is constant along its

vertical lines, and the extrema are degenerate — not Morse. Bent, each slice's wall curve is a parabola with one nondegenerate extremum. Choset describes the silhouette as the ellipsoid's equator, the perimeter of the hole and the two curves along its side; the sweep finds it in six pieces, and the linking curves join them into one. That is the misconception the widget kills: *the silhouette is not the outline* of the body as you see it. It is the critical set of a projection, and it lives where the surface's tangent plane contains the q_3 direction.

$$D(f,\pi_{12})(q) = \begin{bmatrix} 2q_1 & 2q_2 & 2q_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \qquad \det D(f,\pi_{12}) = 2q_3 = 0 \iff q_3 = 0 \text{ (the equator).}$$

DERIVATION **Silhouettes are critical sets, and connectivity changes only at critical points**

\Sigma(\pi_{12}|_S) = \textstyle\bigcup_\lambda \Sigma(\pi_2|_{Q_\lambda}),\quad \text{slice topology constant between adjacent critical values}

Statement (Choset §5.5.1; Lemmas 5.5.1–5.5.2, the slice lemma, Morse theory [Choset ref. 315]). For $S = f^{-1}(0)$, a point $x \in S$ is on the silhouette iff $D(f,\pi_{12})(x)$ loses rank; the silhouette is the union over λ of the critical points of π_2 on each slice; and between adjacent critical values the slice topology does not change, so the silhouette's connectivity can change only at critical slices.

Step 1 — Lagrange. At an extremum p of h on $f^{-1}(c)$, every direction tangent to the surface leaves h stationary to first order, so $\nabla h(p)$ has no tangential component: $\nabla h(p) = \mu \nabla f(p)$ (Lemma 5.5.1).

Step 2 — stacking. For vector-valued constraints and functions the same condition says the rows of Dh are combinations of the rows of Df on the tangent space — rank loss of the stacked matrix $D(f,h)$ (Lemma 5.5.2). For two real-valued functions, rank loss of a two-row matrix *is* parallelism, and Lemma 5.5.1 is recovered.

Step 3 — the slice lemma. Fixing $q_1 = \lambda$ and extremizing q_2 on the slice is the same as asking that the map $q \mapsto (q_1, q_2)$ restricted to S fail to be a submersion: the rows e_1 (the slice) and e_2 (the extremized coordinate) together with ∇f lose rank. So $\Sigma(\pi_{12}|_S) = \bigcup_\lambda \Sigma(\pi_2|_{\pi_1^{-1}(\lambda)})$.

Step 4 — the sphere. For $f = q_1^2 + q_2^2 + q_3^2 - r^2$ the stacked matrix above has determinant $2q_3$, zero exactly on the equator. The check sweeps the unit sphere: every silhouette sample has $|q_3| < 10^{-6}$ and $|\det| = 2|q_3| < 10^{-6}$ there, while at $(0, 0.6, 0.8)$, off the equator, $|\det| = 1.6$; the only critical slices are birth and death at $\lambda = \pm 1$.

Step 5 — critical slices are where the roadmap's tangent lies in the slice.

At a critical point of π_1 restricted to the roadmap, the roadmap's tangent is orthogonal to $\nabla\pi_1 = e_1$: it lies in the slice. There the number of silhouette pieces can change, and by Morse theory it can change *only* there: between adjacent critical values a diffeomorphism carries one slice's intersection with S to the next (Choset Fig. 5.33). ■

The recursion. Each critical slice is a lower-dimensional problem of the same kind: sweep it along q_2, extremize q_3, find its critical slices, recurse, down to dimension one. Accessibility and departability come from treating the slices through q_{start} and q_{goal} as critical, and connectivity is proved by induction on dimension [Choset refs. 90–92]. The demonstrator in this chapter recurses once: in a surface in $\mathbb{R}^3$ a critical slice meets the surface in curves, and those curves *are* their own roadmap — the linking curves. *The numbers.* The torus sweep finds its critical values at $-2.7, -1.3, 1.3, 2.7$ to 2×10^{-3}, with two slice curves and four extrema just inside $|\lambda| = 1.3$ and one curve with two extrema just outside, and four silhouette components that the linking curves join into one. The bent-hole ellipsoid has six critical values, at -2.999, -0.122, -0.109, 1.307, 1.322 and 2.992. The slice-curve count changes exactly twice, where the hole punches through at $0.6 - 0.7$ and closes at $0.6 + 0.7$; the smoothing of the rim (the solid is a smooth maximum of the ellipsoid and the tube, so that Newton has a gradient) adds a "dimple" critical slice just outside each. Midway through the hole the slice is two curves with six extrema. Six silhouette components, one with the linking curves.

Choset's ellipsoid has four critical points, Cp_1 to Cp_4, because his hole goes down and comes back up, punching through the slice twice on the way in and twice on the way out. Ours goes straight through, so it has two; the honest bookkeeping above is what the sweep finds, not what the figure shows.

9.3.7 The opportunistic path planner and the generalized gradient

Canny's silhouettes extremize a coordinate. Canny and Lin's **opportunistic path planner** (OPP) extremizes something a robot cares about: the distance to the nearest obstacle, restricted to the slice.

DEFINITIONS — FREEWAY, CHANNEL, BRIDGE CURVE, INTERESTING CRITICAL POINT (CHOSET §5.5.2, EQS. 5.7–5.8)

Write $q = [\lambda, p]^\top$ with p the slice coordinates. The **slice-restricted distances** are $\tilde{d}_i(p;\lambda) = d_i(q)$ and $\tilde{D}(p;\lambda) = \min_i \tilde{d}_i(p;\lambda)$. A **freeway** is the trace of the local maxima of $\tilde{D}$ on the slices as λ sweeps. A **channel** is a connected component of the free part of a slice. An **interesting critical point** is where

channels join or split; a **bridge curve** joins neighbouring freeways near an interesting critical point, or leads away from a **bifurcation point** where a freeway ends in free space. The union of freeways and bridges is the OPP roadmap.

DEFINITION — GENERALIZED GRADIENT (CHOSET §5.5.2, AFTER CLARKE)

Where $D = \min_i d_i$ is not differentiable — at ties — its **generalized gradient** is the convex hull of the active gradients, $\partial D(q^\star) = \mathrm{Co}\{\nabla d_i(q^\star) : i \in Z(q^\star)\}$, with $Z(q^\star)$ the indices of the closest obstacles. If $0 \in \partial D(q^\star)$ then $q^\star$ is a local maximum; if $\partial D(q^\star) = \{0\}$, a local minimum. No second derivative is needed: first-order information characterizes the extremum.

DERIVATION **OPP freeways are ties, characterized by the generalized gradient**

$q^\star$ a slice maximum of $\tilde{D} \iff 0 \in \mathrm{Co}\{\partial \tilde{d}_i/\partial p : i \in Z(q^\star)\}$

Statement (Choset §5.5.2). Slice maxima of $\tilde{D}$ trace freeways joined by bridges near interesting critical points. Since $\tilde{D}$ is nonsmooth at ties, a maximum is characterized by $0 \in \partial\tilde{D}$; for convex obstacles every freeway point is a tie, hence a GVD point.

Step 1 — smooth between ties. On a slice, $\tilde{D} = \tilde{d}_i$ wherever QO_i is the unique closest obstacle, and it inherits the smoothness of $\tilde{d}_i$ there (Choset Fig. 5.37).

Step 2 — at a tie, the convex hull. Approaching a tie from either side gives the limiting gradients $\nabla\tilde{d}_i$ of the tied obstacles; the generalized gradient is their convex hull.

Step 3 — a maximum means no ascent direction. At a slice maximum no direction along the slice increases every active $\tilde{d}_i$; by a separating-hyperplane argument that is exactly $0 \in \mathrm{Co}\{\nabla\tilde{d}_i\}$. On a one-dimensional slice: the active slopes bracket zero.

Step 4 — bridges at merges and splits. Freeways are traces of maxima *within a channel*; where two channels merge or split — where the slice's normal ∇h is parallel to an obstacle's surface normal, the rank loss of $D(f, h)$ again (Choset Fig. 5.35) — maxima appear or vanish, and a bridge along the slice joins them. ■

Problem 12, for convex obstacles. Each d_i is convex when QO_i is (Choset Problem 19), so its restriction to a slice is convex and has no interior maximum; a maximum of $\tilde{D}$ must therefore be a tie of two or more obstacles with slopes of both signs — a point of the GVD, for *any* slice

direction. Exercise 2 asks for the general proof. *The numbers.* In the chapter's tilted 10×6 room with two convex obstacles, the sweep finds 181 freeway points on 4 freeways; at every one of them the active slopes bracket zero and two or more obstacles tie. The channel count changes at the interesting critical values $\lambda = 2.2, 4.6, 6.0, 8.2$, and the 10 bridges there and at the bifurcation points connect the four freeways into one roadmap. The room's walls are tilted on purpose: a slice parallel to a wall makes $\tilde{D}$ constant along it, every point a degenerate maximum, and the Morse assumption fails — the tilt is the general-position fix.

9.3.8 What exactness costs

Canny's algorithm was the first general roadmap algorithm, and with it came the complexity bound that made roadmap theory a theory: for a configuration space of dimension n whose obstacles are bounded by p polynomials of degree at most w, any path-planning query can be answered in

$$p^{\,n}(\log p)\, w^{O(n^4)}$$

time [Choset refs. 90–92]. That is *singly* exponential in n, where the earlier general method of Schwartz and Sharir — cylindrical algebraic decomposition — was doubly exponential. It is still enormous. Taking the constant in the $O(\cdot)$ as one, $p = 10$ and $w = 2$, the base-ten logarithm of the bound is 7.2 for $n = 2$, 27.7 for $n = 3$, 81.4 for $n = 4$ and 396.5 for $n = 6$ — a six-joint arm among quadric obstacles: more operations than there are atoms in the observable universe, by over three hundred orders of magnitude. And no algorithm can do fundamentally better in general: Reif showed that the generalized mover's problem — many independently moving parts — is PSPACE-hard (Appendix D). Exact, complete planning in high dimensions is not going to get cheap.

Canny's roadmap is complete, and it has never been a practical planner; the opportunistic planner, which is, gives up nothing of the theory but runs on a sweep that still scales with the volume of the space. The honest reading is that this chapter's exact structures are tools for low dimensions and for *understanding*: the GVG for a mobile robot that must explore, silhouettes as the reason sweep-based coverage works (Chapter 10), critical points as the place where everything interesting happens. For a six-joint arm, Part III trades completeness for probabilistic completeness and the problem becomes tractable.

9.4 The algorithm

ALGORITHM SENSOR-BASED GVG EXPLORATION -- Choset §5.2.5, §5.3.3 steps (1)-(5) COST one traversal per GVG edge plus the backtracking of a depth-first search; one scan and one 2×2 (3×3) solve per step

in a nearest-obstacle query (a range ring's local minima of ρ with their bearings, or exact geometry); gains $\alpha > 0$, $\beta < 0$; step dt; tolerances meetTol, boundaryTol, nodeTol

out a graph whose nodes are meet, access and boundary points and whose edges are the driven polylines

1: **access**: drive along ∇d_1 until $d_2 - d_1 <$ `accessTol`; in $\mathbb{R}^3$ continue along the two-equidistant sheet, increasing d, until a third reading ties. Record an access node with two departure slots
2: **depart**: pick an unexplored slot at the current node, step `departStep` along it, start a new edge
3: **trace**: apply the Lyapunov law $\dot{q} = \alpha v + \beta(DG)^{\dagger}G$ with G from the m nearest readings and v signed to continue the heading; append q to the edge; log Γ
4: **detect**: if the $(m+1)$-th reading comes within `meetTol` of the first — "a sudden change in one of the two closest obstacles" — go to **home**; if $d_1 <$ `boundaryTol`, close the edge at a boundary point and go to **backtrack**
5: **home**: Newton on $G = [d_1 - d_2, \ldots, d_1 - d_{m+1}]$, i.e. the homing law with $\beta\, dt = -1$, to `homeTol`; if a node lies within `nodeTol`, close the edge there and mark the arriving slot explored; otherwise create a meet node with the $m+1$ departure directions (each omits one obstacle, signed so the omitted one falls behind) and close the edge there
6: **branch**: if the node has an unexplored slot go to **depart**; else **backtrack**
7: **backtrack**: breadth-first search over known edges to the nearest node with an unexplored slot; drive its polylines; go to **depart**. If no such node exists: **done**

Rejected ties. A homing that converges somewhere other than a three-way (four-way) tie, or a tie whose gradients coincide, is reported as spurious and tracing resumes after a short cooldown — the "sudden change" detector is a heuristic, and with noisy ranges it fires early, late or twice. *Degenerate meet points.* When four or more readings tie to within the sensor's resolution — the Apartment's mirrored doorways produce meet points two millimetres apart — the node gets one departure per *adjacent* pair of closest points around the robot.

ALGORITHM LYAPUNOV TRACER AND MEET-POINT HOMING -- Choset eq. (5.5), §5.3.3 COST $O(m^3)$ for the $(m-1)\times(m-1)$ or $m\times m$ inverse

in the m (edge) or m + 1 (homing) nearest readings d_i, ∇d_i; gains α, β; the previous heading
out the commanded velocity $\dot{q}$ and $\Gamma = \frac{1}{2}G^\top G$

1: $G \leftarrow [d_1 - d_2, \ldots, d_1 - d_k]$, $DG \leftarrow$ rows $(\nabla d_1 - \nabla d_j)^\top$
2: $(DG)^\dagger \leftarrow DG^\top (DG\, DG^\top)^{-1}$; **if** $DG\, DG^\top$ is singular **return** "undefined" (a meet point for the edge law, or parallel gradients)
3: **if** $k = m$: $v \leftarrow$ unit vector of Null(DG) — the perpendicular of the row in the plane, the cross product of the rows in $\mathbb{R}^3$ — negated if $v\cdot$ heading < 0; **else** $v \leftarrow 0$
4: **return** $\dot{q} = \alpha v + \beta(DG)^\dagger G$ and $\Gamma = \frac{1}{2}G^\top G$; along it $\dot{\Gamma} = \beta G^\top G$

ALGORITHM HGVG PERIOD FOLLOWING -- Choset §5.3.2 COST one planar exploration of the sheet; one projected-gradient descent

in a sheet F_ij (here the floor–ceiling plane z = h/2) with its nearest-obstacle query; a candidate obstacle k
out a period around k, a link to the GVG cycle, the cycle

1: explore the planar GVG of the sheet — the second-order edges $F_{kl}|_{F_{ij}}$ — with the exploration algorithm above, using the sheet's own query (obstacles i, j left out)
2: collect the second-order edges whose tied pair includes k; if they close into a loop around k (angular gaps small), it is a **period**
3: from a point of the period, follow $\dot{c} = -\pi_{TF_{ij}} \nabla d_k(c)$ until $d_k = d_i = d_j$ (stop: a GVG edge) or $\|\pi_{TF_{ij}} \nabla d_k\| \to 0$ (stop: no edge; return to the period)
4: trace the GVG edge reached; if it closes on itself, it is the missing **GVG cycle**; link it to the period

ALGORITHM CANNY'S ROADMAP (demonstrator) -- Choset §5.5.1 COST general algorithm: $p^n (\log p)\, w^{O(n^4)}$; the demonstrator: one 2-D curve trace per slice

in an implicit surface $S = f^{-1}(0)$ in R^3 with ∇f; a box; the number of slices
out silhouette curves, critical slices with their before/after signatures, linking curves; component counts

1: **for** λ on a grid over the q_1-range: trace every closed curve of $f(\lambda, p) = 0$ with Chapter 3's predictor–corrector, seeded from a lattice and deduplicated
2: on each curve, locate the extrema of q_2 as the sign changes of $\partial f / \partial q_3$ (the tangent $(-f_3, f_2)$ has no q_2-component there — Lemma 5.5.2), refined by bisection along the chord with Newton projection
3: **signature** of a slice = (number of curves, number of extrema); between grid values with different signatures, bisect on λ to locate the **critical value**
4: chain the extrema across slices into **silhouette curves** (nearest neighbour, maxima with maxima)
5: **recurse once**: on each critical slice the roadmap of the one-dimensional intersection is the intersection itself — take the slice curves just either side of $\lambda^\star$ as **linking curves**
6: count connected components of the silhouette alone and with the linking curves

ALGORITHM OPPORTUNISTIC PATH PLANNER -- Choset §5.5.2 (Canny and Lin)
COST one 1-D maximization per slice; bisection on the channel count
in a planar polygonal world in general position; a slice direction
out freeways, interesting critical values, bridge curves

1: **for** each slice $x = \lambda$: split the slice into **channels**; on each, find the strict local maxima of $\tilde{D}$ by sampling and golden-section refinement
2: at each maximum record the active set Z and the slopes $\partial \tilde{d}_i / \partial y$, and test $0 \in \mathrm{Co}\{\cdot\}$
3: chain maxima across slices into **freeways**
4: locate **interesting critical values** by bisection where the channel count changes; on the merged side, add the slice segment joining the freeway points of the splitting channels as a **bridge**
5: where a freeway ends in free space (a bifurcation point), bridge it along its last slice to the nearest freeway point in the same channel
6: access and departure: slice-constrained gradient ascent of $\tilde{D}$ from q_{start} and q_{goal}

9.5 Implementation in Rust

This chapter adds the `gvg` submodule to Chapter 8's `roadmap` crate: `scan.rs` turns a range ring into the k nearest obstacles, `tracer.rs` is eq. (5.5), `explore.rs`

the five-step exploration, `hgvg.rs` the period following, `canny.rs` the silhouette demonstrator and `opp.rs` the freeways. The tracer is written once for any number of readings; const generics carry the sizes, and the pseudoinverse returns `None` exactly where the Lyapunov argument stops holding.

RUST **crates/roadmap/src/gvg/tracer.rs**

```rust
use nalgebra::{SMatrix, SVector, Vector2};

/// The k nearest hits in a range ring: distance and unit direction away from each (C sec .4.3.1).
/// `obstacle_id` is `None` from a scan -- the sensor cannot name what it sees, and the law never asks.
pub struct Nearest<const K: usize> {
    pub d: [f64; K],
    pub grad: [Vector2<f64>; K],
    pub obstacle_id: [Option<usize>; K],
}

pub struct Twist { pub v: Vector2<f64> }
pub struct GvgTracer { pub alpha: f64, pub beta: f64 }

/// DG? = DG^T (DG DG^T)-1 for a full-row-rank differential. `None` where the Gram matrix is singular:
/// two sheet normals parallel (a non-transversal crossing) or, for the edge law, a meet point --
/// exactly where Lyapunov's neighborhood ends and homing has to take over.
pub fn pseudo_inverse<const R: usize, const C: usize>(dg: &SMatrix<f64, R, C>) -> Option<SMatrix<f64, C, R>> {
    let gram = dg * dg.transpose();
    gram.try_inverse().map(|inv| dg.transpose() * inv)
}

/// G and DG from the first R + 1 readings: rows d_1 - d_j and ?d_1 - ?d_j.
fn constraint<const K: usize, const R: usize>(n: &Nearest<K>) -> (SVector<f64, R>, SMatrix<f64, R, 2>) {
    let g = SVector::<f64, R>::from_fn(|j, _| n.d[0] - n.d[j + 1]);
    let dg = SMatrix::<f64, R, 2>::from_fn(|j, c| n.grad[0][c] - n.grad[j + 1][c]);
    (g, dg)
}

impl GvgTracer {
    /// Edge tracing, C eq. (5.5): q_dot = alpha v + beta (DG)? G with v in Null(DG), ||v|| = 1, signed to keep
    /// going the way we were going (Choset's footnote 4). Also returns Gamma = ? G^TG; Gamma_dot = beta G^TG.
    pub fn trace_step(&self, n: &Nearest<2>, heading: Vector2<f64>) -> Option<(Twist, f64)> {
        let (g, dg) = constraint::<2, 1>(n);
        let dg_pinv = pseudo_inverse(&dg)?;
        let row = dg.row(0);
        let mut v = Vector2::new(-row[1], row[0]).normalize();
        if v.dot(&heading) < 0.0 { v = -v; }
        let correction = self.beta * (dg_pinv * g);
        Some((Twist { v: self.alpha * v + correction }, 0.5 * g.norm_squared()))
    }

    /// Meet-point homing: DG is square, Null(DG) = {0}, so the law is a Newton-like step
    /// beta DG-1 G toward the centre of the circle through the three closest points.
    pub fn home_step(&self, n: &Nearest<3>) -> Option<(Twist, f64)> {
```

```
        let (g, dg) = constraint::<3, 2>(n);
        let dg_pinv = pseudo_inverse(&dg)?;
        Some((Twist { v: self.beta * (dg_pinv * g) }, 0.5 * g.norm_squared()))
    }

    /// Access (Ch. 8, Lemma 5.2.1): climb ?d_1 until the second reading ties. Unit speed, so the
    /// ascent takes d_2 - d_1 seconds at most from any start in a bounded room.
    pub fn access_step(&self, n: &Nearest<2>) -> Twist {
        Twist { v: self.alpha * n.grad[0] }
    }
}
```

The tracer returns `Option` rather than a bare `(Twist, f64)` so that the one place the derivation fails — a singular $DG\,DG^{\top}$ — is a value the caller must handle, not a `NaN` that drives the robot into a wall. The explorer is a state machine around the tracer, and its graph is `petgraph`'s.

RUST **crates/roadmap/src/gvg/explore.rs**

```
use petgraph::graph::{NodeIndex, UnGraph};

pub struct MeetNode { pub q: Vector2<f64>, pub clearance: f64, pub departures: Vec<Vector2<f64>>, pub explored: Vec<bool> }
pub struct Edge { pub polyline: Vec<Vector2<f64>>, pub length: f64 }
/// The incrementally built topological map; `frontier` lists (node, slot) pairs not yet driven.
pub struct GvgGraph { pub graph: UnGraph<MeetNode, Edge>, pub frontier: Vec<(NodeIndex, usize)> }
pub enum Event { OnEdge, MeetPoint(NodeIndex), BoundaryPoint, Done }
enum Phase { Access, Depart, Trace, Home, Backtrack(Vec<Vector2<f64>>), Done }

pub struct Explorer<'r> {
    pub robot: &'r mut sim::Rusty,
    pub tracer: GvgTracer,
    pub graph: GvgGraph,
    phase: Phase, heading: Vector2<f64>, edge: Vec<Vector2<f64>>, from: NodeIndex,
    pub gamma_log: Vec<f64>,
}

impl Explorer<'_> {
    /// One tick: scan; access / trace / home by phase; DFS to the next unexplored slot.
    pub fn tick(&mut self, dt: f64) -> Event {
        let near: Nearest<3> = scan::nearest_from_scan(&self.robot.scan());
        match &self.phase {
            Phase::Access => {
                if near.d[1] - near.d[0] > ACCESS_TOL { self.drive(self.tracer.access_step(&near.first2()), dt); return Event::OnEdge; }
                self.from = self.add_node(self.robot.position(), access_departures(&near));
                self.phase = Phase::Depart;
                Event::OnEdge
            }
            Phase::Trace => {
                // "A sudden change in one of the two closest obstacles": a third reading closes in.
                if near.d[2] - near.d[0] < MEET_TOL { self.phase = Phase::Home; return
```

```
    Event::OnEdge; }
                if near.d[0] < BOUNDARY_TOL { return self.close_at_boundary(); }
                let Some((tw, gamma)) = self.tracer.trace_step(&near.first2(), self.
    heading) else {
                    self.phase = Phase::Home; // DG DG^T singular: we are at (or past) a
    meet point
                    return Event::OnEdge;
                };
                self.gamma_log.push(gamma);
                self.drive(tw, dt);
                self.edge.push(self.robot.position());
                Event::OnEdge
            }
            Phase::Home => match newton_home(&mut *self.robot, &self.tracer, HOME_TOL) {
                Some(q) => {
                    let node = self.node_near(q, NODE_TOL).unwrap_or_else(|| self.
    add_meet_node(q, &near));
                    self.close_edge(node);
                    self.next_from(node);
                    Event::MeetPoint(node)
                }
                None => { self.phase = Phase::Trace; Event::OnEdge } // spurious tie:
    carry on
            },
            Phase::Depart => self.depart(dt),
            Phase::Backtrack(_) => self.follow_backtrack(dt),
            Phase::Done => Event::Done,
        }
    }

    /// DFS bookkeeping: an unexplored slot here, or BFS over known edges to the nearest
    node with one.
    fn next_from(&mut self, n: NodeIndex) {
        self.phase = if self.graph.graph[n].explored.iter().any(|e| !e) { Phase::Depart }
            else { match self.path_to_frontier(n) { Some(p) => Phase::Backtrack(p), None
    => Phase::Done } };
    }
}
```

Canny's demonstrator works on implicit surfaces. Each slice is a planar curve-tracing problem, so Chapter 3's tracer does the work; the extrema are the sign changes of one partial derivative.

RUST **crates/roadmap/src/gvg/canny.rs**

```
use nalgebra::{SVector, Vector2};

pub struct Silhouette {
    pub curves: Vec<Vec<SVector<f64, 3>>>,
    pub critical_points: Vec<SVector<f64, 3>>,
    pub linking: Vec<Vec<SVector<f64, 3>>>,
    pub depth: usize,
}
struct Slice { lambda: f64, pieces: Vec<Vec<Vector2<f64>>>, extrema: Vec<(SVector<f64, 3>,
    bool)> }

/// Sweep pi_1 slices; pi_2-extrema where D(f, pi_1_2) = [?f; e_1; e_2] loses rank, i.e.
   ?f/?q_3 = 0
```

```
/// (Lemma 5.5.2); recurse once on the critical slices, where the (curves, extrema) count changes.
pub fn canny(f: &dyn Fn(SVector<f64, 3>) -> f64, grad_f: &dyn Fn(SVector<f64, 3>) -> SVector<f64, 3>,
             bounds: [(f64, f64); 3], n_slices: usize) -> Silhouette {
    let lambdas: Vec<f64> = (0..=n_slices).map(|k| bounds[0].0 + (bounds[0].1 - bounds[0].0) * k as f64 / n_slices as f64).collect();
    let slices: Vec<Slice> = lambdas.iter().map(|&l| slice(f, grad_f, l, bounds)).collect();
    let mut sil = Silhouette { curves: chain_extrema(&slices), critical_points: vec![], linking: vec![], depth: 1 };
    for w in slices.windows(2) {
        if signature(&w[0]) == signature(&w[1]) { continue; }
        // Bisect on lambda for the critical value; the slice there is pinched, so the linking curves
        // are taken just either side -- the recursion's roadmap of a one-dimensional set is the set.
        let l_star = bisect_signature(f, grad_f, bounds, w[0].lambda, w[1].lambda, 8);
        let eps = 0.5 * (w[1].lambda - w[0].lambda);
        for l in [l_star - eps, l_star + eps] {
            sil.linking.extend(slice(f, grad_f, l, bounds).pieces.into_iter().map(|c| lift(l, &c)));
        }
        sil.critical_points.push(changed_extremum(&w[0], &w[1], l_star));
        sil.depth = 2;
    }
    sil
}

/// The curves of f(lambda, .) = 0 by Chapter 3's predictor-corrector, and the sign changes of ?f/?q_3
/// along each: there the tangent (-f_3, f_2) has no q_2-component, so q_2 is extremal.
fn slice(f: &dyn Fn(SVector<f64, 3>) -> f64, grad_f: &dyn Fn(SVector<f64, 3>) -> SVector<f64, 3>,
         lambda: f64, bounds: [(f64, f64); 3]) -> Slice {
    let g = |p: Vector2<f64>| f(SVector::<f64, 3>::new(lambda, p.x, p.y));
    let dg = |p: Vector2<f64>| { let n = grad_f(SVector::<f64, 3>::new(lambda, p.x, p.y)); Vector2::new(n[1], n[2]) };
    let pieces = trace_all_closed(&g, &dg, bounds[1], bounds[2], bugs::TraceParams { step: 0.03, ..Default::default() });
    let f3 = |p: &Vector2<f64>| grad_f(SVector::<f64, 3>::new(lambda, p.x, p.y))[2];
    let extrema = pieces.iter().flat_map(|c| sign_changes(c, f3).map(|(p, is_max)| (lift_point(lambda, p), is_max))).collect();
    Slice { lambda, pieces, extrema }
}
```

The opportunistic planner's one new idea fits in a function: a slice maximum of a nonsmooth minimum is a convex-hull test on first derivatives.

RUST **crates/roadmap/src/gvg/opp.rs**

```
/// 0 in ?D(q*) = Co{?d_i/?y : i in Z(q*)} on a vertical slice: the active slopes bracket zero.
/// For one active obstacle this is the smooth condition ?d/?y = 0; at a tie it is a kink maximum.
pub fn generalized_gradient_has_zero(slopes: &[f64], tol: f64) -> bool {
    let lo = slopes.iter().cloned().fold(f64::INFINITY, f64::min);
    let hi = slopes.iter().cloned().fold(f64::NEG_INFINITY, f64::max);
```

```
    lo <= tol && hi >= -tol
}

pub struct FreewayPoint { pub q: Point2<f64>, pub d: f64, pub active: Vec<usize>, pub slopes: Vec<f64>, pub channel: usize }

/// Strict local maxima of D(.; lambda) on each channel of the slice x = lambda, golden-section refined.
pub fn slice_maxima(world: &OppWorld, lambda: f64, samples: usize) -> Vec<FreewayPoint> {
    channels(world, lambda).iter().enumerate().flat_map(|(ci, &(lo, hi))| {
        let d = |y: f64| world.nearest(Point2::new(lambda, y)).d;
        strict_maxima(d, lo, hi, samples).into_iter().map(move |y| world.freeway_point(lambda, y, ci))
    }).collect()
}
```

9.5.1 The worked example, and its printed output

RUST **crates/roadmap/examples/lyapunov_step.rs**

```
use nalgebra::Vector2;
use roadmap::gvg::{GvgTracer, Nearest};

/// Two point sites; Nearest in *site* order (site 1 at the origin), as the text labels them.
fn nearest(q: Vector2<f64>, sites: &[Vector2<f64>; 2]) -> Nearest<2> {
    let d = sites.map(|c| (q - c).norm());
    let grad = sites.map(|c| (q - c).normalize());
    Nearest { d, grad, obstacle_id: [Some(0), Some(1)] }
}

fn main() {
    let sites = [Vector2::new(0.0, 0.0), Vector2::new(4.0, 0.0)];
    let tracer = GvgTracer { alpha: 1.0, beta: -1.0 };
    let q = Vector2::new(2.5, 1.0);
    let n = nearest(q, &sites);
    let g = n.d[0] - n.d[1];
    let dg = n.grad[0] - n.grad[1];
    let dg_pinv = dg / dg.norm_squared();
    let v = Vector2::new(-dg.y, dg.x).normalize();
    let (tw, gamma) = tracer.trace_step(&n, Vector2::new(0.0, 1.0)).unwrap();
    println!("G = {g:.4}  DG = ({:.4}, {:.4})  DG+ = ({:.4}, {:.4})  v = ({:.4}, {:.4})",
    dg.x, dg.y, dg_pinv.x, dg_pinv.y, v.x, v.y);
    println!("correction = ({:.4}, {:.4})  qdot = ({:.4}, {:.4})  Gamma = {gamma:.4}  dGamma = {:.4}",
             tracer.beta * dg_pinv.x * g, tracer.beta * dg_pinv.y * g, tw.v.x, tw.v.y, tracer.beta * g * g);

    // The Lyapunov box: integrate, watch Gamma. The edge x = 2 is vertical, so "keep going up" is the
    // heading for the whole run.
    let heading = Vector2::new(0.0, 1.0);
    let (mut q, mut prev) = (q, f64::INFINITY);
    for _ in 0..200 {
        let (tw, gamma) = tracer.trace_step(&nearest(q, &sites), heading).unwrap();
        assert!(gamma <= prev + 1e-12, "Gamma must not increase");
        prev = gamma;
```

```
        q += tw.v * 0.05;
    }
    println!("200 steps of dt = 0.05: Gamma monotone, |x - 2| < 1e-3: {}", (q.x - 2.0).
    abs() < 1e-3);
}
```

```
G = 0.8898  DG = (1.7605, -0.1833)  DG+ = (0.5619, -0.0585)  v = (0.1036, 0.9946)
correction = (-0.5000, 0.0521)  qdot = (-0.3964, 1.0467)  Gamma = 0.3959  dGamma =
   -0.7918
200 steps of dt = 0.05: Gamma monotone, |x - 2| < 1e-3: true
```

`lyapunov_step_matches_text` pins every printed number to 10^{-4}, the correction's x-component to -0.5 to 10^{-12}, and the monotone descent of Γ. Two hundred steps of 0.01 s would not land the robot: at $t = 2$ s the offset has shrunk only by e^{-2}, to about 0.07. So the test integrates to $t = 10$ s, where $|x - 2| < 10^{-3}$ holds, and checks the $t = 2$ s value of Γ against $e^{-4}\Gamma_0$ instead. The other two examples print the chapter's larger numbers:

```
explore: done after 5779 ticks, 205.0 m driven
         27 meet points, 33 boundary points, 62 edges, 103.4 m of roadmap
         every homed meet point three-way equidistant by exact geometry to < 1e-6
exact GVD (Chapter 8): 26 meet points, 26 matched within 0.05 m, 2 extra: (6.00, 4.30)
   (7.85, 6.79)
fifty seeded queries, 29 pass the filter, 29 answered
         mean length: GVG 10.88 m   shortest (disc, r = 0.22) 6.68 m   ratio 1.62 mean,
   2.46 worst
         mean min clearance: GVG 0.48 m   shortest 0.22 m

torus R = 2, r = 0.7: critical lambda -2.700 -1.300 1.300 2.700; silhouette components 4
   -> 1
ellipsoid with a bent hole: critical lambda (curves/extrema before -> after)
  -2.999   0/0  -> 1/2
  -0.122   1/2  -> 1/10
  -0.109   1/10 -> 2/6
   1.307   2/6  -> 1/6
   1.322   1/6  -> 1/2
   2.992   1/2  -> 0/0
  silhouette components 6 -> 1 with linking curves
sphere r = 1: silhouette |q3| < 1e-6 at every sample; critical lambda -1.000 1.000
```

`explored_graph_matches_exact_gvd` asserts that every exact meet point is matched within five centimetres and every homed meet point is a three-way tie by exact geometry; `sphere_silhouette_is_equator` asserts $|q_3| < 10^{-6}$ (Derivation 5, step 4); `torus_pinches_at_r_minus_r` and `ellipsoid_critical_slices` pin the values above. The TypeScript port in `web/lib/roadmap/` runs the same twelve checks, and every widget on this page is that port.

> **NOTE**
>
> Three things the port had to learn from the range ring. First, a ring of 360 rays reports the local minima of ρ *on its rays*, not on the obstacles: the closest point can sit half a degree away, and the homing then converges to a

point a centimetre off the true meet point. The port refines each minimum's bearing by golden-section search between its neighbouring rays — Chapter 3's "more rays where they matter" — and the meet points then land on the exact ones to 10^{-6}. Second, one wall seen across the $0/2\pi$ seam, or a flat minimum split in two, produces two readings with the same gradient: DG is singular and the meet detector fires on nothing. Readings within 0.05 rad and 10^{-6} m of each other are merged. Third, transversality says exactly degenerate meet points have probability zero; the Apartment's mirrored doorways produce two meet points two millimetres apart all the same, and the robot cannot tell them apart. Such a node gets one departure per adjacent pair of closest points, which is what the geometry would give if the two points were merged.

9.6 Putting it together: the floor plan Rusty drew

The integration lab uses the explored graph as what Chapter 8 said a roadmap is for. Rusty explores the Apartment from room A with the ideal ring, and the graph it draws — 27 meet points, 62 edges — is turned into a roadmap: every polyline sample a node, consecutive samples joined, meet points shared. Getting on is Chapter 8's gradient ascent, the same access the explorer used; the landing point is on the GVD, and the nearest explored sample is never more than 2 cm away, which is the honest measure of how completely the sensor-built graph covers the exact diagram. Riding is Chapter 6's A*. Getting off is the goal's ascent, reversed.

Chapter 8's fifty seeded start–goal pairs — the same generator, the same filter, twenty-nine pairs surviving it — are answered on the explored GVG and on the visibility graph for a 0.22 m disc. All twenty-nine are answered; the explored graph is one connected component. The GVG paths average 10.88 m against 6.68 m for the shortest paths: 1.62 times as long on average and 2.46 times in the worst case — the price of riding the middle of every room and doorway. They average 0.48 m of minimum clearance against 0.22 m — the shortest path grazes the inflated walls, which is what shortest means. The sensor-built graph's mean path length is within two percent of Chapter 8's grid-GVD paths for the same queries (10.70 m): the robot's map is as good a highway as the geometer's.

The sister book's frontier exploration (*Probabilistic Robotics via Rust*, Chapter 24) is the probabilistic cousin of this lab: it drives toward the boundary between known and unknown cells of an occupancy grid, while the GVG explorer drives along a topological skeleton and needs no grid at all — but it trusts its range readings, and the GVG Explorer's noise slider shows what that trust costs (Exercise 4). What a range ring actually returns, and how noisy, is the sister book's Chapter 10. Maps that keep a Euclidean distance field (the sister book's Chapter 19 surveys map representations) yield the GVD as the field's ridge, exactly as Chapter 8's

brushfire did; and Choset and Nagatani's topological SLAM, which localizes a robot by where it is *on* the explored GVG, is the direct descendant of this chapter's explorer.

Three pointers close Part II's roadmaps. Chapter 10 takes the critical points of this chapter literally: the boustrophedon decomposition's cell boundaries are the slices through Canny's critical points, found by a robot with Lemma 5.5.2 in its sensor loop. The complexity bound sends us to Chapter 11, where a roadmap is built by throwing darts and the contract of Definition 5.0.2 is restated with probability one. And the capstone, Chapter 23, explores an unknown Apartment again.

9.7 Exercises

1 FOUNDATION •• **The Newton step lands on the bisector — exactly, and only for points**

For two point sites c_1, c_2 and $G = d_1 - d_2$, prove that the component of $(DG)^{\dagger}G$ along $\hat{u} = (c_2 - c_1)/\|c_2 - c_1\|$ equals the signed distance from q to the bisector, for every q off the line through the sites (Choset's micro-example has it equal to 0.5). Then repeat the computation numerically for a point site and a segment site (a parabolic edge) and explain what you find: is the step still exact, and if not, what is it?

WORK IT OUT

With sites (0, 0) and (4, 0), Rusty at (3, −1) and $\beta = -1$, what is the x-component of the correction β(DG)†G?

Answer. −1 The signed distance from $(3, -1)$ to the bisector $x = 2$ is 1, so the correction's component along the site axis is $\beta \cdot 1 = -1$. The proof: put $x = q - c_1$ and $y = q - c_2$, so that $x - y = L\hat{u}$ with $L = \|c_2 - c_1\|$. Then $d_1^2 - d_2^2 = (x - y) \cdot (x + y)$ is $2L$ times the signed distance s, and $G = 2Ls/(d_1+d_2)$. With $DG = x/d_1 - y/d_2$, the product $L\,\hat{u} \cdot DG = (x-y)\cdot(x/d_1 - y/d_2)$ simplifies to $(d_1+d_2)(1 - x\cdot y/(d_1 d_2))$, while $\|DG\|^2 = 2(1 - x\cdot y/(d_1 d_2))$; so $L\,\hat{u}\cdot DG = (d_1+d_2)\|DG\|^2/2$. Since $(DG)^{\dagger}G = DG\,G/\|DG\|^2$, its component along $\hat{u}$ is $\frac{d_1+d_2}{2L} \cdot \frac{2Ls}{d_1+d_2} = s$. For a point and a line the zero set is a parabola, and the Newton step misses it by an amount of second order in the offset: exact only on the edge itself. The check confirms the identity to 10^{-12} at twenty seeded points.

HINTS

<>Write G as $(d_1^2 - d_2^2)/(d_1 + d_2)$. What kind of function of q is $d_1^2 - d_2^2$?</> <>The numerator is affine and vanishes on the bisector, so it is 2L times the signed distance s.</> <>For a parabolic edge the zero set of G curves away from the Newton line; expand to second order in the offset.</>

2 FOUNDATION • • • **OPP is a subset of the GVG, for any slice direction**

Prove Choset's Problem 12: for any choice of slice direction, every point of the opportunistic path planner's roadmap lies on the GVG. Use the generalized-gradient characterization of slice maxima, and say where convexity of the obstacles enters (Problem 19: d_i is convex when QO_i is). Then explain why bridge curves need not lie on the GVG, and whether that matters for the claim.

HINTS

<>Restrict d_i to a line: a convex function on a line has no strict interior maximum.</> <>So a strict slice maximum of $\tilde{D} = \min \tilde{d}$_i cannot be a smooth point of a single $\tilde{d}$_i: at least two obstacles are active there.</> <>At a tie with $0 \in$ Co of the slopes, two of the active slopes have opposite signs, so the gradients differ: SS_ij. Now check the "closest" condition of F_ij.</>

3 CONCEPTUAL • **Predict the landing time, then check**

PREDICT FIRST

In the Lyapunov Funnel with the micro-example's sites, set $\beta = -3$ and put Rusty at (3, 0) heading up. With explicit Euler steps of dt = 0.05 s, how many steps until $|x - 2| < 0.01$? Predict from $\dot{\Gamma} = \beta G^2$ before you run anything.

- About 15
  That is half the time needed: the offset decays like $\exp(\beta t)$ and must shrink a hundredfold, so $t \approx \ln 100 / 3 \approx 1.5$ s — about 31 steps of 0.05 s in continuous time, not 15.
- ✓ 29 without the tangent term, 30 with $\alpha = 1$
  At (3, 0) the tangent is vertical and the offset obeys $x - 2 \leftarrow (1 + \beta dt)(x - 2) = 0.85(x - 2)$, so $\lceil \ln 0.01 / \ln 0.85 \rceil = 29$ steps. With $\alpha = 1$ the robot climbs the edge, where DG tilts and v acquires an x-component of the wrong sign, and it takes one more: 30. The check pins both.
- About 300
  That would be the count at $\beta = -0.3$: the offset shrinks by $(1 + \beta dt)$ per step, so a gain ten times smaller needs about ten times as many steps.

Explain the discrepancy between the discrete count and the continuous prediction $\ln 100/3 \approx 1.54$ s ≈ 31 steps: Euler's factor 0.85 per step decays slightly faster than $e^{-0.15} = 0.861$. Then explain why αv, which is perpendicular to DG and so cannot change G to first order, still costs a step: G's level sets are not parallel to the bisector away from the sites' axis. The widget integrates with $dt = 0.02$ s, so verify there by time rather than by step count: the offset should fall below 0.01 after about 1.5 s.

4 CONCEPTUAL •• **Break the meet-point detector**

In the GVG Explorer, raise the range noise σ step by step until the exploration no longer terminates cleanly — it reports spurious ties, revisits a meet point as a new node, or misses a doorway. From the event log and the meet-point count, decide for one failure whether a meet point was *spurious* (announced where no third obstacle is near) or *missed* (driven through), and relate it to Choset's detector, "a sudden change in one of the two closest obstacles". Which tolerance would you widen to fix each kind, and what does widening it break?

HINTS

<>The detector fires when the third reading comes within meetTol of the first. Noise of σ on each of two readings moves their difference by about $\sqrt{2}\,\sigma$.</> <>A spurious meet fires when noise closes the gap; a missed one when noise opens it as the robot passes. The node tolerance decides whether a re-homed point is "the same node".</>

5 PRACTICAL • • • **Follow a period to the halo**

Implement `follow_period` in `hgvg.rs` for the box-in-a-room world of Choset Fig. 5.19, extruded from analytic planes: explore the second-order GVG on the floor–ceiling sheet with the sheet's own nearest-obstacle query, detect the period whose common second-closest obstacle is the box, descend $-\pi_{TF_{fc}} \nabla d_{box}$ from it, and trace the halo. Assert that the descent stops on a three-way tie of the floor, the ceiling and the box at clearance 1.5, and that the halo closes. Then move the box off-centre vertically and explain what happens to the floor–ceiling sheet's hole.

6 PRACTICAL • • • **The rod-GVG in the Workbench (stretch)**

Implement the rod-GVG of Choset §5.4: $d_i(q)$ for a rod at $q = (x, y, \theta)$ as segment–polygon distance with `parry2d`, its gradient in (x, y, θ) from the closest pair of points, and RF_{ijk} traced with the Lyapunov law over $SE(2)$ ($G \in \mathbb{R}^2$, DG a 2×3 matrix). Sample R-edges as rod placements tangent to Chapter 8's planar GVD, and render Choset Fig. 5.22's panels for the Workbench: the rod-GVG edges as swept placements, the R-edges, and the union in $SE(2)$. Check that a short rod's RF_{ijk} is a single closed curve over θ (footnote 5) and find the rod length at which it splits.

9.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

Chapter 5, §5.3–5.5, is this chapter's source: the GVG and transversality (Fig. 5.17, Table 5.2), the HGVG and eq. (5.4), the Lyapunov tracing law eq. (5.5) and homing, the rod-HGVG, Canny's roadmap with Lemmas 5.5.1–5.5.2 and the slice lemma, and the OPP with the generalized gradient. Canny's roadmap and complexity bound are from Canny's The Complexity of Robot Motion Planning (MIT Press, 1988) and the OPP from Canny and Lin (1990), Choset's refs. 90–93. `link`

Choset, H. and Burdick, J. (2000). Sensor-Based Exploration: The Hierarchical Generalized Voronoi Graph. *International Journal of Robotics Research 19(2), 96–125*.
The HGVG in full: the GVG in R^m, its disconnection, second-order edges and periods, the connectivity argument this chapter cites rather than proves, and the sensor-based exploration procedure. doi:10.1177/02783640022066770

Choset, H. and Nagatani, K. (2001). Topological Simultaneous Localization and Mapping (SLAM): Toward Exact Localization Without Explicit Localization. *IEEE Transactions on Robotics and Automation 17(2), 125–137*.
The explored GVG used as a topological map for localization: what the integration lab's graph becomes when odometry drifts. doi:10.1109/70.928558

Schwartz, J. T. and Sharir, M. (1983). On the "Piano Movers" Problem II: General Techniques for Computing Topological Properties of Real Algebraic Manifolds. *Advances in Applied Mathematics 4(3), 298–351*.
The first general complete algorithm, by cylindrical algebraic decomposition, doubly exponential in the dimension — the bound Canny's roadmap improved to singly exponential. doi:10.1016/0196-8858(83)90014-3

Reif, J. H. (1979). Complexity of the Mover's Problem and Generalizations. *20th Annual Symposium on Foundations of Computer Science (FOCS), 421–427*.
PSPACE-hardness of the generalized mover's problem: the lower bound that makes 'exact planning is expensive' a theorem rather than an observation (Appendix D). doi:10.1109/SFCS.1979.10

Clarke, F. H. (1990). Optimization and Nonsmooth Analysis. *SIAM Classics in Applied Mathematics 5 (reprint of the 1983 Wiley edition)*.
The generalized gradient as the convex hull of limiting gradients, and the first-order characterization of extrema of nonsmooth functions used for the OPP's slice maxima (Choset's ref. 114). doi:10.1137/1.9781611971309

CHAPTER 10

Cell Decompositions and Coverage

CLASSICAL PLANNERS — SEARCH, POTENTIALS, ROADMAPS, CELLS · Intermediate · 65 min

> *A coverage path planner determines a path that passes an effector (e.g., a robot, a detector, etc.) over all points in a free space.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), Chapter 6*

Every planner so far answered "how do I get *there*?" This chapter asks "how do I visit *everywhere*?" — the question of a vacuum cleaner, a demining robot, a lawnmower, a wiping pad on a bench. One object answers both. Carve Q_{free} into **cells** simple enough that a back-and-forth motion sweeps each one, and record which cells touch in an **adjacency graph**: planning is then a search on that graph (Chapter 6), and coverage is a walk that visits every node.

The interesting question is *where the cell walls go*. The trapezoidal decomposition puts one at every vertex — correct, simple, and over-chopped. The boustrophedon decomposition puts one only where a sweeping slice changes connectivity: at the **critical points of the slice function on the obstacle boundary**, which is the Morse vocabulary of Chapter 8 read by the rank condition of Chapter 9. Those critical points can be *sensed* — the range ring's shortest ray lines up with the sweep — so coverage becomes an online algorithm that explores an unknown room while it mows it, with a completeness guarantee and a path length linear in the area, the number of critical points, and the perimeter. A short coda carves the plane by *visibility* instead, and uses the cells to find an evader.

Everything on the page runs. The sweep on the chapter's micro-world produces the 12 trapezoids and 13 adjacencies the design promises and its list of crossed edges matches a brute-force re-sort at every event; Euler's count holds on 300 seeded worlds, convex and star-shaped; the online coverer re-discovers the

six critical points of the offline sweep from range data alone, with its shortest ray horizontal to within a hundredth of a degree; and the information-graph search proves — by exhausting it — that one pursuer cannot clear three pockets.

◎ AFTER THIS CHAPTER YOU CAN

- Define an exact cell decomposition and its adjacency graph, and reduce planning to graph search and coverage to an exhaustive walk (Choset Ch. 6 introduction)
- Build the trapezoidal decomposition by a vertical sweep: the four event types of Choset Fig. 6.4, the list L of crossed edges, extensions only into free space, O(n log n) — and count its cells, V + N_ob − 1 for any polygonal free space in general position
- Prove midpoint–centroid paths through trapezoids are collision-free, and plan with Chapter 6's A* on the adjacency graph
- Define critical points of a slice function on the obstacle boundary, Morse decompositions (Def. 6.2.1) and their duals, Reeb graphs; prove Euler's count N_ce = N_cp + N_ob − 1 and see where it stops applying (brushfire)
- Cover a cell decomposition with laps of width 2r, bound the area a lawnmower misses and Choset's path length $\Delta^2/2r + 4\Delta N\_ce + 5P\_total$, and measure both
- Sense critical points from a range ring while covering an unknown space, explain why a raster scan misses them, and run the cycle algorithm that does not
- State the pursuit–evasion problem, compute visibility polygons and gap-edge labels, and search the information graph for a strategy

ASSUMED

- Polygonal worlds, the Apartment and the Workbench, Rusty's range ring (Chapter 2); the curve tracer and the offset follower (Chapter 3)
- Disc C-obstacles by inflation (Chapter 4)
- A*, BFS and DFS on a graph (Chapter 6); brushfire and wave-front (Chapter 7)
- The Morse box — critical points, index, Euler characteristic (Chapter 8); Lemma 5.5.2 and slices (Chapter 9)

10.1 The problem: Rusty mows the living room

Room C of the Apartment, furnished for this chapter with a sofa against the north wall and a coffee table, is the free space of disc-Rusty: the room shrunk by Rusty's

radius, the furniture grown by it (Chapter 4's inflation). Rusty's effector — the brush — is its own diameter, $2r = 0.44$ m. Put it down in the south-west corner of the room and ask it to cover the room twice, from the same start, knowing nothing about the room but what its range ring tells it.

Left: a conventional raster scan — lap until contact, follow whichever boundary the lap ended on for one lap width, lap back — senses critical points only on the boundary it happens to follow. It finds the room's first and last points, concludes the room is one cell, and stops after 21.7 m with 79.7 % covered; the hatched strip behind the sofa, the floor between the coffee table and the south wall, and a sliver along the east wall were never visited. Right: the cycle algorithm closes a loop around every gap between laps, senses all six critical points (blue dots), and covers 100 % in 85.7 m. Purple is covered ground, at the 3 cm resolution of the chapter's checks.

The raster scan *looked* done. Every lap it drove was a full lap, every turn was neat, and the algorithm terminated by its own rule — no unexplored openings in the cell it thought it was covering. But it never followed the bottom of the sofa at the moment the sofa ended, so it never learned that the free space split there; the strip behind the sofa was another cell, and nothing told the robot it existed. That is Choset Fig. 6.22 in a living room, and the rest of the chapter is the machinery that makes the failure impossible: cells, the points where cells begin and end, how to sense those points, and a motion pattern that cannot walk past one.

10.2 Building intuition

10.2.1 Cells, walls, and a graph

DEFINITIONS — EXACT CELL DECOMPOSITION, ADJACENCY GRAPH, COVERAGE (CHOSET CH. 6 INTRODUCTION)

An **exact cell decomposition** of Q_{free} is a partition into simple regions — **cells** c_i — whose union is exactly Q_{free}. Two cells are **adjacent** if they share a boundary; the **adjacency graph** has a node per cell and an edge per shared boundary. **Planning** with a decomposition is two steps: locate the cells containing q_{start} and q_{goal}, then search the adjacency graph. A **coverage path** passes an effector of width $2r$ over every point of Q_{free}; with cells it is an **exhaustive walk** through the adjacency graph (every node visited) plus a pattern inside each cell.

The cells have to be "simple" in a specific sense: something a robot can cover without thinking. A cell bounded left and right by vertical segments and above and below by obstacle boundary — a *floor* and a *ceiling* that never double back in

x — is covered by vertical laps one effector width apart, stepping over along the floor or ceiling between laps. That is the boustrophedon ("ox-turning") pattern of a plough, and it is why every decomposition in this chapter is built by sweeping a vertical line.

10.2.2 Where the walls go: the trapezoidal sweep

> INTERACTIVE · w10.3
>
> **Trapezoid Sweep**
>
> Not every vertex gets two extensions: only those with free space both above and below — the boustrophedon critical points — get both; the rest get one or none, and the totals land on 12 cells and 13 adjacencies.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch10-cell-decompositions#w10.3

The chapter's micro-world — the **bench** — is a convex quadrilateral plot $B = (0, 0.2), (10, 0), (10.4, 6), (0.3, 6.2)$ with a triangle $T = (2, 2), (4, 1.5), (3, 4)$ and a quadrilateral $K = (6, 1), (8, 0.8), (8.6, 4), (6.5, 4.4)$; its eleven vertex abscissae are distinct. A vertical line sweeps it from left to right and stops only at vertices. Between stops nothing can happen: the edges the line crosses stay in the same vertical order, and the free intervals between them only stretch or shrink. The widget's strip is that ordered list, L. At a stop, the two edges meeting at the vertex are inserted into L, deleted from it, or one replaces the other — Choset's four event types — and the vertex shoots a **vertical extension** up to the next edge of L above it and down to the next below, *if free space lies there*.

Watch the totals. The eleven vertices do not produce twenty-two extensions; they produce thirteen, because most vertices have obstacle on one side. The pass-through vertices — $(0.3, 6.2)$ on the plot's upper chain, T's top $(3, 4)$, K's $(6.5, 4.4)$, $(8, 0.8)$, and the plot's corner $(10, 0)$ — each get one. The leftmost and rightmost vertices of T and K get two. The plot's own leftmost and rightmost vertices get none. Cells: twelve trapezoids (some degenerate into triangles), and the adjacency graph has thirteen edges, one per extension.

10.2.3 Fewer walls: the boustrophedon decomposition

> INTERACTIVE · w10.1
>
> **Boustrophedon Mower**
>
> More cells is not finer coverage: cell walls belong only at the critical points, and every extra cell costs a turnaround and a transit — the trapezoidal mower is longer, not better.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/

```
ch10-cell-decompositions#w10.1
```

Now keep only the extensions that go *both* up and down. There are four — at the leftmost and rightmost points of T and of K — and with the slice's birth at $(0, 0.2)$ and death at $(10.4, 6)$ they are the six places where the free part of the sweep line changes its number of pieces: one interval, then two (above and below T), then one, two (around K), one, none. Merge the trapezoids that the other extensions separated and you have seven cells. The Reeb graph docked beside the canvas is the dual picture: a node per critical point, an edge per cell. Six nodes, seven edges, two holes:

$$N_{ce} = N_{cp} + N_{ob} - 1, \qquad 7 = 6 + 2 - 1,$$

and the widget re-prints it live while you drag any vertex — Euler's count is a theorem, not a coincidence of this world.

Then Rusty mows. With a 0.5 m effector, the first cell (from $x = 0$ to the triangle's tip at $x = 2$) takes $\lceil 2/0.5 \rceil = 4$ laps. From a start at $(1.5, 3)$ the boustrophedon mower finishes the bench in 158.4 m; switch to the trapezoidal decomposition and the same mower needs 174.4 m — one more lap and 14 m more of transits, because twelve cells mean twelve entries and twelve exits. In the living room the gap is wider: 45.9 m against 79.1 m, because the octagonal corners of Rusty's C-obstacles shatter the room into 29 trapezoids. **More cells is not finer coverage; it is more turnarounds.**

10.2.4 Six slices, one definition

Nothing in the boustrophedon decomposition depends on the slices being vertical lines. Sweep circles out from a centre, or diamonds, or rays, or the fronts of a brushfire, and the same rule — a cell wall wherever the swept slice splits or merges — gives a different decomposition and a different pattern of laps (Choset Figs. 6.12–6.18). The gallery computes all six on the bench by one raster algorithm.

> The slice-function gallery on the bench (8 cm pixels). Cells are tinted, the critical slices I* are dark, critical points are blue dots, and thin purple curves are a few level sets h = λ — the laps a robot would drive. Boustrophedon, spike (rays from a centre off the plot, below left), and wave-front (path length from the green point) each give 6 critical points and 7 cells; the spiral and squarel slices about (5, 3) give 12 and 13, and 15 and 16 — all obeying N_ce = N_cp + N_ob − 1. The brushfire slice gives a tree: 6 critical points, 5 cells. Hover a panel for what to notice.

Two things in the gallery are worth a second look. The spiral's circles touch the outer wall many times, yet not every touch is a critical point: the first touch turns

a closed circle into an arc — one piece before, one piece after — and *connectivity* is what Definition 6.2.1 counts. And the brushfire decomposition is a tree, not a graph with a loop per obstacle: its slices are fronts *around* each obstacle, so each obstacle is itself a critical set where a cell is born, and Euler's count takes a different form (Derivation 5).

10.3 The mathematics

c_i	a cell; the adjacency graph has a node per cell and an edge per shared boundary
e_{lower}, e_{upper}; e_{LOWER}, e_{UPPER}; L	the two edges at an event vertex v (e_lower is the one whose other end is lower); the edges of L just below and above v; the sweep line's list of crossed edges, bottom to top (C §6.1)
Q_λ, $Q_{free\,\lambda}$, $Q^j_{free\,\lambda}$	a slice $h^{-1}(\lambda)$, its free part, and the free part's j-th connected interval (a slice interval)
$I^\star$	the slice intervals that contain a critical point; cells are the components of Q_free \ I*
$h : Q \to \mathbb{R}$; m	the slice function (x for boustrophedon; |q − c| spiral; |x| + |y| squarel; D brushfire); an implicit function whose zero set is the obstacle boundary
N_{cp}, N_{ce}, N_{ob}; V	numbers of critical points, cells, obstacles (holes); number of polygon vertices
Δ, $2r$, P_{total}	diameter of the smallest disk containing Q_free; effector width = interlap spacing; perimeter of the obstacles plus the outer boundary
$V(x)$, $B(x)$; $H(\mathcal{W}_{free})$	visibility polygon of x; the vector of gap-edge labels (0 clear, 1 contaminated); the least number of pursuers that guarantees capture

10.3.1 Cell decompositions by sweeping

DEFINITION — THE TRAPEZOIDAL DECOMPOSITION (CHOSET §6.1)

Let Q_{free} be bounded by a polygon, with polygonal holes, in **general position**: no two vertices share an x-coordinate. At every vertex v draw the **upper vertical extension** from v up to the first edge above it and the **lower vertical extension** down to the first edge below — each only if it lies in Q_{free}. The extensions cut Q_{free} into trapezoids (some with a zero-length side: triangles). Two trapezoids are adjacent iff they share an extension.

DERIVATION **The trapezoidal sweep: four event types, O(n log n)**

L changes only at vertices; each event is an insertion, a deletion or a replacement next to

e_LOWER, e_UPPER

Statement (Choset §6.1). Sort the n vertices by x. Sweep a vertical line through them keeping the list L of edges it crosses, ordered by height. With L in a balanced tree, each event updates L and finds e_{LOWER}, e_{UPPER} in $O(\log n)$, so the whole decomposition costs $O(n \log n)$ — against the naive $O(n^2)$ of intersecting a vertical line through each vertex with every edge.
Step 1 — nothing happens between vertices. Edges do not cross (the polygons are simple and disjoint), so two edges both crossing the line keep their vertical order until one of them ends, which happens at a vertex. Between events L is frozen.
Step 2 — at an event, only the two edges at v change. Let e_{lower}, e_{upper} be the edges containing v, labelled so that the other end of e_{lower} is lower. Four cases (Choset Fig. 6.4): both to the right of the line — insert both between e_{LOWER} and e_{UPPER}; both to the left — delete both; e_{lower} left, e_{upper} right — e_{upper} takes e_{lower}'s slot; e_{lower} right, e_{upper} left — e_{lower} takes e_{upper}'s slot. Every update happens *at the position of v in L*, so finding e_{LOWER} and e_{UPPER} is one search.
Step 3 — the extensions. The extension from v upward ends on e_{UPPER} at height $e_{UPPER}(v_x)$, the downward one on e_{LOWER}; each exists exactly when the direction $\pm\hat{y}$ at v points into free space, which the angular sector of free space at v decides in $O(1)$.
Step 4 — the count. $O(n \log n)$ to sort, n events at $O(\log n)$ each. ■
The bench's trace. Number the edges $e_0, \ldots, e_3$ around the plot, e_4, e_5, e_6 around T and $e_7, \ldots, e_{10}$ around K, each counter-clockwise from the first listed vertex. The sweep's list is

$$\emptyset \to \{e_0, e_3\} \to \{e_0, e_2\} \to \{e_0, e_4, e_6, e_2\} \to \{e_0, e_4, e_5, e_2\} \to \{e_0, e_2\} \to \{e_0, e_7, e_{10}, e_2\} \to \{e_0, e_7, e_9, e_2\} \to \{e_0, e_8, e_9, e_2\} \to \{e_0, e_2\} \to \{e_1, e_2\} \to \emptyset,$$

the same shape as Choset's $L : \emptyset \to \{e_8, e_{13}\} \to \{e_8, e_0, e_3, e_{13}\} \to \ldots$ for his Fig. 6.1. The check re-sorts the crossed edges by brute force between every pair of events and finds no disagreement. *Two honesty items.* The TypeScript port keeps L as a sorted array — $O(n)$ per update, which changes the constant and not the idea. And general position is an assumption: the Apartment's rectilinear walls violate it on every wall, so the port shears rectilinear worlds by $x \leftarrow x + \sigma y$ with $\sigma = \sqrt{2} \cdot 10^{-3}$, which keeps every edge straight and every polygon simple, moves nothing by more than six millimetres in these rooms, and turns each vertical wall's two corners into one critical point and one pass-through. *Point location* (C §6.1, end): vertical lines through all events cut the plane into slabs; a binary search finds a query's slab and a second search its floor and ceiling, $O(\log n)$ per query

[C ref. 124].

How many trapezoids? Each event opens new cells: a vertex with extensions both ways (a **split**) closes one cell and opens two; a vertex where two edges leave and free space lies *between* them (a **birth**) opens one; a pass-through closes one and opens one; a vertex where two edges arrive with obstacle between (a **merge**) closes two and opens one; a vertex where two edges arrive with free space between (a **death**) only closes. Counting cells opened per vertex gives

$$\#\text{trapezoids} = V + \#\text{split} - \#\text{death}.$$

For a convex m-gon boundary with convex k_i-gon holes, each hole has one split and no death and the boundary has one death and no split, so the count is $(m-1) + \sum_i (k_i + 1)$ — on the bench, $3+4+5 = 12$. The surprise is that the closed form survives non-convexity: $\#\text{split} - \#\text{death} = N_{ob} - 1$ for *every* polygonal free space in general position (Exercise 1 proves it from Euler's count), so

$$\#\text{trapezoids} = V + N_{ob} - 1, \qquad \#\text{adjacencies} = V + 2N_{ob} - 2,$$

which the check confirms on 200 convex and 100 star-shaped seeded worlds — 69 of the latter have a notch tip that opens or closes a slice — and which gives the living room's $28 + 2 - 1 = 29$ and the Workbench top's $16 + 3 - 1 = 18$.

DERIVATION **Midpoint–centroid paths are collision-free**

a path in the adjacency graph ⇒ a polyline in Q_free

Statement (Choset §6.1). Let $c_0, c_1, \ldots, c_k$ be a path in the adjacency graph from the cell containing q_{start} to the cell containing q_{goal}. The polyline q_{start} → midpoint of the extension $c_0|c_1$ → centroid of c_1 → midpoint of $c_1|c_2$ → … → midpoint of $c_{k-1}|c_k$ → q_{goal} lies in $\mathcal{Q}_{\text{free}}$.

Step 1 — trapezoids are convex. A trapezoid is the region between two non-crossing segments over a common x-interval: an intersection of four half-planes.

Step 2 — each piece joins two points of one cell. q_{start} and the first midpoint lie in c_0 (the midpoint is on its right or left wall); the midpoint of $c_{i-1}|c_i$ and the centroid of c_i lie in c_i; and so on. By convexity each segment stays in its cell.

Step 3 — consecutive pieces meet on shared walls, which are in $\mathcal{Q}_{\text{free}}$. ■

In practice the adjacency graph is a Chapter 6 `Graph`, costed by the length of each centroid–midpoint–centroid hop, and A* with the centroid-to-centroid straight line as heuristic (admissible by the triangle inequality) finds the route; the check plans 120 random queries over the bench, the

living room and the Workbench top and samples every path at 1 cm — all inside Q_{free}. The argument fails for the non-convex cells of the next section, which is why the coverage code routes its *transits* between boustrophedon cells through the trapezoids, which refine them.

10.3.2 Morse decompositions

DEFINITIONS — SLICES, CRITICAL POINTS, MORSE DECOMPOSITION (CHOSET §6.2.2, DEF. 6.2.1 [C REF. 12

)] Let $h : Q \to \mathbb{R}$ be a **slice function** and $Q_\lambda = h^{-1}(\lambda)$ a **slice**; $Q_{free\,\lambda} = Q_\lambda \cap Q_{\text{free}}$ is its free part, whose connected components $Q^j_{free\,\lambda}$ are the **slice intervals**. A **critical point** is a point of ∂Q_{free} where $Q_{free\,\lambda}$ changes connectivity as λ passes h of it — for $h = x$, a vertex with extensions both up and down, or the slice's birth or death. Let $I^\star$ be the set of slice intervals containing a critical point. A **Morse decomposition** is the exact cell decomposition whose cells are the connected components of $Q_{\text{free}} \setminus I^\star$; the **boustrophedon decomposition** [C ref. 110] is the case $h(x, y) = x$.

DERIVATION **Between critical slices the free slice keeps its topology**

Q_free,λ_1 ≅ Q_free,λ_2 when no critical value lies between λ_1 and λ_2

Statement (Choset §6.2.2; the Morse box of Chapter 8; Lemma 5.5.2 of Chapter 9). If h restricted to ∂Q_{free} is Morse, then for $\lambda_1 < \lambda_2$ with no critical value between them a diffeomorphism carries $Q_{free\,\lambda_1}$ onto $Q_{free\,\lambda_2}$: slice intervals stretch or shrink, but never split, merge, appear or vanish. Hence each cell is swept by a *single* interval that varies continuously — exactly what a lawnmower needs.

Step 1 — connectivity changes only at tangency. An interval's endpoints lie on ∂Q_{free}. Moving λ, an endpoint slides along the boundary as long as the slice crosses the boundary transversally; intervals can only be created, destroyed, cut or joined where the slice is *tangent* to ∂Q_{free}.

Step 2 — tangency is a rank condition. With $\partial Q_{\text{free}} = m^{-1}(0)$, the slice is tangent at x iff $\nabla h(x) \parallel \nabla m(x)$, i.e. the stacked differential $D(h, m)(x)$ loses rank — Lemma 5.5.2.

Step 3 — those are the critical points of $h|_{\partial Q_{\text{free}}}$. On the one-dimensional boundary, rank loss of $D(h, m)$ means the derivative of h along the boundary vanishes.

Step 4 — Morse theory between critical values. For a Morse function the

topology of the level sets changes only across critical values [C ref. 315] — the Chapter 8 box applied to $h|_{\partial Q_{\text{free}}}$, with the interior of each slice interval carried along. ■

Consequences. A critical point lies at an *endpoint* of a slice interval, never inside one — inside, the slice meets free space on both sides and is not tangent to anything (Exercise 2). *Not every tangency is critical.* A circle centred in free space first touches the outer wall from inside: a closed loop becomes an arc, one piece before and one after. That is a topology change of the slice but not a *connectivity* change, and Definition 6.2.1 counts connectivity — which is why the spiral panel of the gallery has fewer critical points than its circles have tangencies. *Nonsmooth slices* — $|x| + |y|$, the distance D — use Chapter 9's generalized gradient: for the squarel a corner of the diamond is critical when the obstacle's normal lies in the convex hull of the two flat sides meeting there (C §6.2.3). *Degenerate critical points* — a vertical wall seen by $h = x$ — are what rectilinear rooms are made of; Butler et al. [C ref. 86] cover them directly, and this book shears them away offline and merges them online (Derivation 4).

 lap 1 · x = 0.25 lap 2 · x = 0.75 lap 3 · x = 1.25 lap 4 · x = 1.75

Choset's boustrophedon cell as Definition 6.2.1 sees it, with the bench's numbers: the first cell is swept by one slice interval from its birth at (0, 0.2) to the critical slice through T's tip at x = 2, where the interval splits around the triangle. Four laps at 2r = 0.5 cover x ∈ [0, 2]; the abscissae are 0.25, 0.75, 1.25, 1.75 — the outer laps r inside the cell's ends, so every point is within r of a lap horizontally.

10.3.3 Sensing a critical point

Choset's observation (§6.2.4) is that a critical point can be *felt*. A robot following an obstacle boundary at standoff δ knows the direction to the nearest obstacle point — its range ring's shortest ray — and that direction is $-\nabla D$, the negative of the distance gradient (Chapter 7, Derivation 2). The boundary normal is $\nabla m \parallel \nabla D$ there. So the rank condition of Derivation 3 becomes one angle comparison.

DERIVATION **Critical points are sensed as 'shortest ray parallel to the sweep'**

for h = x: the range ring's minimum ray points straight ahead or straight behind

Statement. While boundary-following, the robot is at a critical point of h exactly when its range ring's global minimum ray points along $\pm\nabla h$ — for $h = x$, along $\pm\hat{x}$.

Step 1 — the ray is the normal. The nearest obstacle point c satisfies $x - c = D(x)\,\nabla D(x)$, and $\nabla D(x)$ is the outward normal of the boundary at c (the offset curve's normal too).

Step 2 — rank loss is parallelism. $D(h, m)$ loses rank iff $\nabla h \parallel \nabla m$ (Lemma 5.5.2), i.e. iff the normal at c is horizontal when $h = x$.

Step 3 — one comparison. The minimum ray is $-\nabla D$, so the test is whether its bearing is 0 or π. Equivalently: the follower's tangent is vertical, so its x-velocity changes sign — the boundary *turns back* in the sweep direction. ■

Classification from the same ray. Moving $+x$, a turn-back with the shortest ray pointing *back* (obstacle behind, $\nabla D = +\hat{x}$) is a convex extremity bulging into the robot's path — a **merge**; with the ray pointing ahead it is a corner that pinches the slice off — a **death**. Moving $-x$ the same two cases are a **split** and a **birth**. *The port's sensing* (`online.rs`) watches the follower's x-progress and declares an event once it has fallen back by 0.3δ after a maximum; on a convex extremity it then steps along the arc of radius δ to the point level with the obstacle vertex, where the minimum ray is exactly horizontal, and re-reads the ring. The check finds the minimum ray within $0.01°$ of $\pm\hat{x}$ at every split and merge on the bench, and within a tenth of a degree — the general-position shear — in the sheared living room. *Degenerate critical points* are where Choset's sensing fails: along an axis-aligned wall the shortest ray is horizontal for the wall's whole length. The port's **merged-event rule** — beyond Choset — reports the stretch where the maximum x was attained once, at its middle, so a vertical wall's two corner critical points become one event. *Why a raster scan misses them* (C Fig. 6.22): the test fires only on the boundary being followed, and a raster follows the ceiling only on every other gap.

10.3.4 The Reeb graph and Euler's count

DEFINITIONS — REEB GRAPH, CLOSING CRITICAL POINT (CHOSET §6.2.4 [C REF. 154

)] The **Reeb graph** of a Morse decomposition is its dual: a node per critical point, an edge per cell, joining the critical point where the cell opens to the one where it closes. It is connected and planar. While a robot covers a

cell it starts from one of the cell's defining critical points; the other one — whose discovery completes the cell — is the cell's **closing critical point**.

$$N_{ce} = N_{cp} + N_{ob} - 1 \qquad \Longleftarrow \qquad v - e + f = 2, \quad v = N_{cp},\ e = N_{ce},\ f = N_{ob} + 1.$$

DERIVATION **Euler's count: N_ce = N_cp + N_ob − 1**

cells = critical points + obstacles − 1 (boustrophedon); a tree for brushfire

Statement (Choset §6.2.5). For a boustrophedon decomposition of a bounded planar free space, $N_{ce} = N_{cp} + N_{ob} - 1$, where the outer boundary's leftmost and rightmost points count as critical points (the slice's birth and death).
Step 1 — the Reeb graph is connected and planar, with $v = N_{cp}$ and $e = N_{ce}$: draw each node at its critical point and each edge through its cell.
Step 2 — its faces are the obstacles plus the outside. Each hole is encircled by the cells that pass above and below it, so it is a face; the unbounded face is the outside. $f = N_{ob} + 1$.
Step 3 — Euler's formula for a connected plane graph [C ref. 57]: $v - e + f = 2$.
Step 4 — rearrange: $N_{cp} - N_{ce} + N_{ob} + 1 = 2$. ■
Checks. Choset Fig. 6.25: $21 + 2 - 1 = 22$ cells. The bench: $6 + 2 - 1 = 7$, with Reeb nodes at $x = 0, 2, 4, 6, 8.6, 10.4$. The check runs the sweep on 200 convex and 100 star-shaped seeded worlds and the identity never fails; cells grow *linearly* in the critical points discovered. *The same fact from the Morse side.* Chapter 8's box counts critical points with signs: for a manifold with boundary and $h = x$, $\chi(\mathcal{Q}_{\text{free}})$ is the sum of $(-1)^{\text{index}}$ over the boundary critical points where ∇h points *into* $\mathcal{Q}_{\text{free}}$ — births (index 0) and merges (index 1). On the bench that is $1 - 2 = -1 = 1 - N_{ob}$, and the check computes it with Chapter 8's `eulerCharacteristicFromCriticalPoints`. *Where the count changes form.* For the brushfire slice $h = D$ every obstacle boundary is a whole critical set at $D = 0$ — the fronts start *around* each obstacle rather than passing on both sides of it — so the obstacles are nodes, not faces, and the Reeb graph is a tree: $N_{ce} = N_{cp} - 1$ with each obstacle counted once among the critical points. The gallery's brushfire panel (6 critical points, 5 cells) is exactly that.

10.3.5 The complexity of coverage

DERIVATION **Coverage path length is linear in area, critical points and perimeter**

$length < \Delta^2/2r + 4\Delta N_c e + 5P_t otal = \Delta^2/2r + 4\Delta(N_c p + N_o b) + 5P_t otal - 4\Delta$

Statement (Choset §6.2.5). The sensor-based boustrophedon coverage path is shorter than $\Delta^2/(2r) + 4\Delta N_{ce} + 5P_{total}$, where Δ is the diameter of the smallest disk containing the space and P_{total} the perimeter of the obstacles and the outer boundary.

Step 1 — laps. Each lap is at most Δ long. At spacing $2r$ the laps across the space need about $\lceil \Delta/2r \rceil$ slice positions, plus one extra first lap per cell (C Fig. 6.26): lap length $\leq \Delta\lceil \Delta/2r \rceil + \Delta N_{ce}$, which is at most $\Delta^2/2r + \Delta N_{ce}$ up to the rounding Choset drops.

Step 2 — boundary following. The cycle algorithm follows the whole floor and ceiling of each cell, and undoing a reverse follow retraces part of it: between $1.5P_{cell}$ and $2P_{cell}$ per cell (C Fig. 6.27), less than $2P_{total}$ in all.

Step 3 — backtracking to the closing critical point of a cell with uncovered neighbours: at most $P_{cell} + \Delta$ each, $P_{total} + \Delta N_{ce}$ in all (C Fig. 6.28).

Step 4 — discovery and traversal. Discovering a cell's defining critical point costs at most P_{cell} more (P_{total} in all); a depth-first search over the Reeb graph traverses each covered cell at most once on the way to an uncovered one [C ref. 118], at most $P_{cell} + 2\Delta$ each — $P_{total} + 2N_{ce}\Delta$ in all (C Figs. 6.29–6.30).

Step 5 — sum: $\Delta^2/2r + \Delta N_{ce} + 2P_{total} + P_{total} + \Delta N_{ce} + P_{total} + P_{total} + 2N_{ce}\Delta = \Delta^2/2r + 4\Delta N_{ce} + 5P_{total}$, and Euler's count turns N_{ce} into $N_{cp} + N_{ob} - 1$. ■

How loose is it? The bench has $\Delta = 11.908$ m and $P_{total} = 49.955$ m; at $2r = 0.5$ and $N_{ce} = 7$ the bound is 866.8 m. The offline mower needs 158.4 m and the online cycle algorithm 272.4 m. In the living room ($\Delta = 4.899$, $P_{total} = 23.434$, $2r = 0.44$) the bound is 308.9 m against 45.9 m offline and 85.7 m online. The bound is a guarantee of *linearity*, not an estimate. *One caution the port taught us:* the lap *count* $\lceil \Delta/2r \rceil + N_{ce}$ is not a bound when cells are stacked — the bench's mower uses 32 laps where the formula allows 31, because the cells above and below each obstacle each need their own laps at the same abscissae. What the argument needs, and what holds, is the lap *length*: stacked laps share one slice, whose total length is at most Δ.

How much can a lawnmower *miss*? Inside a cell with floor $f(x)$ and ceiling $c(x)$, the laps sit at most r apart horizontally from every point, so a point (x, y) escapes its nearest lap at x_j only if $y < f(x_j)$ or $y > c(x_j)$ — a sliver under a dip of the floor or over a bump of the ceiling, of height at most the variation of f or c between x_j and x. Integrating over the half-gaps gives a bound that is *not* in

Choset, and that the check holds the mower to:

$$\text{missed area} \;\le\; r \sum_{\text{cells}} \big(\mathrm{TV}(f) + \mathrm{TV}(c)\big) \;\le\; r\,P_{total},$$

with TV the total vertical variation of a chain. On the bench it is 6.150 m^2, and the mower misses 0.67 m^2 (98.7 % covered on a 3 cm raster) — all of it under the slanted edges of the plot and of T and K. In the rectilinear living room the bound is 2.350 m^2 and the mower misses almost nothing (99.87 %, the octagonal corners of the furniture). The online coverer, which follows *both* floor and ceiling of every gap, covers 100.0 % in all three worlds.

10.3.6 Coda: decomposing by information — pursuit and evasion

The last decomposition in Choset Chapter 6 is carved by visibility, not by a sweep, and it is a workspace problem: the pursuers and the evader are points in $\mathcal{W}_{free}$, not configurations of a robot (C Problem 7).

DEFINITIONS — VISIBILITY POLYGON, GAP EDGES, INFORMATION STATE (CHOSET §6.3, AFTER SUZUKI & YAMASHITA [C REF. 403

and LaValle, Guibas et al. [C refs. 172, 273])] The **visibility polygon** $V(x) \subset \mathcal{W}_{free}$ is the star-shaped set of points in line of sight of x. An evader $e(t)$ is caught at the first t with $e(t) \in V(\gamma_i(t))$ for some pursuer path γ_i. A region that may contain the evader is **contaminated**, otherwise **clear**; cleared and then contaminated again is **recontaminated**. Edges of $V(x)$ that lie in free space are **gap edges**, labelled 0 if they bound a clear region and 1 if contaminated; $B(x)$ is the vector of labels and $(x, B(x))$ the **information state**. A connected set on which the gap structure — hence $B(x)$ — cannot change is **conservative**; conservative regions form an exact cell decomposition. $H(\mathcal{W}_{free})$ is the least number of pursuers that guarantees capture.

Crossing from one conservative region to another changes the gaps in one of four ways, and the labels follow (Choset §6.3): a gap edge **disappears** — do nothing; a gap edge **appears** — label it 0, since what it now hides was just visible; gap edges **merge** — the new label is the OR of the old; a gap edge **splits** — the pieces inherit its label. The **information graph** G_i has a node per (conservative region, label vector) and an edge per transition the rules allow; a pursuer strategy is a path in G_i to any node whose labels are all zero.

DERIVATION How many pursuers: O(log n), and O(√h + log n) with holes

H(W_free) ≤ k + 1 when a chord held by one static pursuer splits W_free into parts needing

k

Statement (Choset §6.3). A simply connected polygonal $\mathcal{W}_{free}$ with n edges can be cleared by $O(\log n)$ pursuers; with h holes, by $O(\sqrt{h} + \log n)$.
Step 1 — a static pursuer splits the problem. If a chord between two boundary vertices splits $\mathcal{W}_{free}$ into $\mathcal{W}_1$ and $\mathcal{W}_2$ with $H(\mathcal{W}_1), H(\mathcal{W}_2) \le k$, then $H(\mathcal{W}_{free}) \le k + 1$: park one "static pursuer" on the chord and clear $\mathcal{W}_1$, then $\mathcal{W}_2$, with the same k.
Step 2 — balanced chords give logarithmic depth. A simple polygon has a chord leaving at least a third of the edges on each side, so recursive splitting reaches triangles (clearable by one pursuer) after $O(\log n)$ levels, each holding one static pursuer.
Step 3 — holes. Triangulate; remove the $O(h)$ trichromatic triangles (those touching three boundary components), which leaves simply connected pieces; the dual graph of the trichromatic triangles is planar, so the planar separator theorem splits it with $O(\sqrt{h})$ static pursuers.
Step 4 — recurse: $O(\sqrt{h} + \sqrt{2h/3} + \dots) = O(\sqrt{h})$ static pursuers, plus $O(\log n)$ for the simply connected pieces. ■
The U-worlds of C Fig. 6.40 make the logarithm tight: a 12-edge U needs one pursuer, three U's joined at a hub (36 edges) need two, and so on by threes. The coda's worlds are miniatures of that construction — a square hub with two or three U-shaped pockets (20 and 28 edges) — and the widget below runs the information-graph search on them.

INTERACTIVE · w10.2

Pursuit-Evasion

Seeing a region once does not clear it: information, not geometry, is the state. A pocket cleared and then left unwatched is recontaminated, and in three pockets no single pursuer's path ever drives every gap label to zero.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch10-cell-decompositions#w10.2

The port computes $V(x)$ by an angular sweep — one exact ray through every vertex and one a hair to either side — and the check compares it with 1 800 brute-force rays at each of 24 random points in three worlds: the largest disagreement is below 10^{-13} m. Contamination is kept on an 12.5 cm raster: what the pursuers see is cleared, and contamination then floods every hidden pixel connected to a contaminated one, because the evader is arbitrarily fast. The four transition rules are not coded; they *emerge* from the flood, and the gap labels are read off it. The conservative regions are computed by grouping pixels by

which polygon vertices they cannot see (plus, with a static pursuer, the corners of its view) — a refinement of Choset's regions, since $B(x)$ cannot change while that set is fixed — and labels are carried across a region boundary by overlap: a shadow on the far side is contaminated iff it overlaps a contaminated shadow on the near side. Breadth-first search on the resulting G_i finds a strategy for the two-pocket world after 286 information states, and the replay through the flood leaves nothing contaminated. In the three-pocket world it exhausts all 693 reachable states without an all-zero node — one pursuer *cannot* do it — while a second pursuer parked at the hub (the static pursuer of Step 1) brings a solution after 2 480 states, which the replay confirms. The naive patrol the widget shows instead — visit each pocket's far leg in turn — leaves 10.25 m^2 contaminated: every pocket it clears is recontaminated through the hub while it clears the next.

10.4 The algorithm

ALGORITHM TRAPEZOIDAL SWEEP -- Choset §6.1, Fig. 6.4 COST O(n log n) with L a balanced tree (the port: a sorted array, O(n) per update)

in a polygonal free space in general position (boundary and holes)
out trapezoids, the adjacency graph with one edge per vertical extension

1: sort the vertices by x; $L \leftarrow \emptyset$; open cells keyed by their floor edge
2: **for** each vertex v in order **do**
3: $e_{lower}, e_{upper} \leftarrow$ the edges at v; $e_{LOWER}, e_{UPPER} \leftarrow$ the edges of L just below and above v
4: **if** both edges go right: insert both; **if** free above and below — *split*: close the cell between e_{LOWER} and e_{UPPER}, open two — **else** *birth*: open one between them
5: **if** both edges go left: delete both; **if** free above and below — *merge*: close two, open one — **else** *death*: close the cell between them
6: **else** replace the leaving edge by the arriving one; close the cell whose floor or ceiling it was and open its continuation
7: draw the extensions from v to e_{UPPER} and e_{LOWER} that lie in free space; each joins the closed cell to the opened one
8: **end for**

Boustrophedon is the same loop with one change: at a pass-through (step 6) the cell is not closed — the vertex is appended to its floor or ceiling chain — so walls appear only at births, splits, merges and deaths, the critical points of $h = x$.

ALGORITHM BOUSTROPHEDON COVERAGE (OFFLINE) -- Choset §6.2.1, Fig. 6.10

COST O(V + E) for the walk; one A* per transit; O(laps) per cell

in a boustrophedon decomposition, effector width 2r, a start point

out a coverage path: laps, boundary steps, transits

exhaustive walk: depth-first search of the adjacency graph from the start cell, with backtracking — on the bench $0-1-3-2-3-4-6-5-6-4-3-1-0$; cover cells in first-visit order
for each cell $c = [x_l, x_r]$ in that order **do**
  laps at $\lceil (x_r - x_l)/2r \rceil$ abscissae, the outer ones r inside the ends, the rest evenly spaced
   enter at the corner nearest the robot; transit there by A* on the *trapezoids* with midpoint–centroid chaining
  lap floor → ceiling, step along the ceiling to the next lap, lap ceiling → floor, step along the floor, . . .
end for

ALGORITHM CYCLE ALGORITHM (SENSOR-BASED COVERAGE) -- Choset §6.2.4, Fig. 6.23 [C ref. 11] COST online; path length within the bound of Derivation 6

in a robot with a contact sensor, a boundary follower at standoff δ, a range ring; effector width 2r

out a coverage path and the Reeb graph, built incrementally

seek: move −x to contact, follow the boundary −x until a turn-back — the first critical point; push it on the DFS stack
while some node on the stack has an unexplored side **do**
  travel to it over covered ground (A* on the robot's own trail plus range-ring line-of-sight shortcuts); step $r - \delta$ into the new cell
  repeat for each gap between laps x_k and $x_{k+1} = x_k + 2r$:
    forward: follow the boundary the last lap ended on, forward, one lap width (stop at a turn-back event)
     lap to the other boundary; *reverse*: follow it backward one lap width (stop at an event)
    closing: if that did not return to the previous lap's far end, lap toward it and follow forward to the critical point that explains why
    undo: retrace the reverse follow; **until** an event closes the cell
  classify the event from the range ring (split, merge, birth, death); match it to a known node or add one; mark the arriving side explored; for a split or merge, drive one full lap through it (the closing lap)

10: **end while**

A *raster scan* is steps 5 and the first half of 6 alone — lap, follow forward, lap, follow forward on the other side — which is why it never follows a ceiling backward and never sees a split.

ALGORITHM INFORMATION-GRAPH SEARCH -- Choset §6.3, Figs. 6.38-6.39 COST

BFS over (region, labels): O(regions · 2^gaps) nodes in the worst case

in a polygonal W_free, a pursuer start (optionally a static second pursuer)

out a pursuer path whose information state reaches all zeros, or proof that none exists

1: conservative regions: group free points by the set of vertices they cannot see; regions are the connected components
2: start node: the start's region, every label 1 (everything unseen is contaminated)
3: **for** each crossing $A \to B$: compute the shadows (hidden components) just inside A and just inside B; a shadow in B is contaminated iff it overlaps a contaminated shadow in A — rules 1–4 of §6.3
4: breadth-first search to any node with all labels 0; replay the region path through the contamination flood to confirm

10.5 Implementation in Rust

The `cells` crate is introduced here. Its spine is a decomposition — cells with floor and ceiling chains, and the adjacency graph as a `petgraph` graph whose edges carry the shared wall — and every algorithm above produces or consumes one.

RUST **crates/cells/src/lib.rs**

```rust
use nalgebra::Point2;
use petgraph::graph::UnGraph;

/// A planar polygonal free space: Choset sec.6.1's standing assumption. For disc-Rusty the
/// polygons are Chapter 4 C-obstacles; for Reach's pad they are the Workbench's objects.
#[derive(Clone, Debug)]
pub struct PolyWorld {
    pub boundary: Vec<Point2<f64>>,       // counter-clockwise
    pub obstacles: Vec<Vec<Point2<f64>>>, // holes, counter-clockwise, disjoint
}

/// The wall two adjacent cells share: a vertical extension from an event vertex.
#[derive(Clone, Copy, Debug)]
pub struct Boundary {
    pub segment: (Point2<f64>, Point2<f64>),
}
```

```rust

/// A cell: everything between a floor chain and a ceiling chain over an x-interval.
/// Trapezoids have two-point chains; boustrophedon cells bend at pass-through vertices.
#[derive(Clone, Debug)]
pub struct Cell {
    pub floor: Vec<Point2<f64>>,
    pub ceiling: Vec<Point2<f64>>,
    pub x_range: (f64, f64),
}

pub struct Decomposition {
    pub cells: Vec<Cell>,
    pub adjacency: UnGraph<usize, Boundary>,
}

impl Decomposition {
    /// Faces of the planar adjacency graph by Euler, f = 2 - V + E: the holes plus the outside.
    pub fn euler_faces(&self) -> i64 {
        2 - self.adjacency.node_count() as i64 + self.adjacency.edge_count() as i64
    }
}

impl PolyWorld {
    /// Choset assumes distinct vertex abscissae; a rectilinear room violates that on every wall.
    /// The shear x <- x + sigmay keeps edges straight and polygons simple, and moves nothing by more
    /// than sigma.height -- a few millimetres in a room of the Apartment.
    pub fn in_general_position(mut self) -> Self {
        const SIGMA: f64 = 1.0e-3 * std::f64::consts::SQRT_2;
        let mut xs: Vec<f64> = self.vertices().map(|p| p.x).collect();
        xs.sort_by(f64::total_cmp);
        if xs.windows(2).all(|w| w[1] - w[0] > 1e-7) {
            return self;
        }
        for p in self.vertices_mut() {
            p.x += SIGMA * p.y;
        }
        self
    }
}
```

The sweep keeps L as a vector sorted by height at the current abscissa — the bookkeeping is easier to read than a tree keyed by a moving comparator, and the event logic is the same. The interesting part is the `match`, which is Choset Fig. 6.4.

RUST **crates/cells/src/trapezoid.rs**

```rust
use std::collections::HashMap;

#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum EventKind { BothRight, BothLeft, LowerLeftUpperRight, LowerRightUpperLeft }

#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum Rule { Trapezoidal, Boustrophedon }

pub fn sweep(world: &PolyWorld, rule: Rule) -> Decomposition {
```

```
    let edges = world.edges();
    let mut events: Vec<Vertex> = world.vertex_list();
    events.sort_by(|a, b| a.p.x.total_cmp(&b.p.x));
    assert!(events.windows(2).all(|w| w[1].p.x > w[0].p.x), "general position: distinct x
");

    let mut l: Vec<usize> = Vec::new();                    // crossed edges, bottom to
top
    let mut open: HashMap<usize, OpenCell> = HashMap::new(); // open cells keyed by floor
    edge
    let mut out = Builder::new(rule);

    for v in &events {
        let (e_prev, e_next) = world.incident_edges(v);
        let rest: Vec<usize> = l.iter().copied().filter(|&e| e != e_prev && e != e_next).
collect();
        // e_LOWER and e_UPPER: one search at v's height, because every update happens
there.
        let i = rest.partition_point(|&e| edges[e].y_at(v.p.x) < v.p.y);
        let (lo, up) = (i.checked_sub(1).map(|j| rest[j]), rest.get(i).copied());
        let (free_up, free_down) = world.free_vertical(v); // the extensions that exist
        match world.event_kind(v) {
            EventKind::BothRight => {
                let (a, b) = world.order_rightward(v, e_prev, e_next); // lower edge
first
                l = [&rest[..i], &[a, b], &rest[i..]].concat();
                if free_up && free_down {
                    // Split: the slice interval (lo, up) is cut by the new obstacle.
                    let c = open.remove(&lo.unwrap()).unwrap();
                    out.close(c, v.p.x);
                    out.open_two(&mut open, (lo.unwrap(), a), (b, up.unwrap()), v, true);
                } else {
                    out.open(&mut open, a, b, v); // birth: free space starts between a
and b
                }
            }
            EventKind::BothLeft => {
                let (a, b) = world.order_leftward(v, e_prev, e_next);
                l = rest;
                if free_up && free_down {
                    let (below, above) = (open.remove(&lo.unwrap()).unwrap(), open.remove
(&b).unwrap());
                    out.merge(below, above, &mut open, lo.unwrap(), up.unwrap(), v); //
merge
                } else {
                    out.close(open.remove(&a).unwrap(), v.p.x); // death
                }
            }
            _ => {
                // One edge leaves the line, the other arrives in its slot.
                let (leaving, arriving) = world.left_right(v, e_prev, e_next);
                for e in l.iter_mut() {
                    if *e == leaving { *e = arriving; }
                }
                let floor_side = free_up; // v lies on the floor of the cell above it
                out.pass_through(&mut open, v, leaving, arriving, floor_side, lo, rule);
            }
        }
    }
    out.finish()
}
```

The Morse layer adds the slice function and the Reeb graph. For $h = x$ the boustrophedon decomposition *is* the sweep with `Rule::Boustrophedon`; the trait is there so that the variable slices of §6.2.3 share one interface (the raster construction behind the gallery implements it for the curved ones).

RUST **crates/cells/src/morse.rs · src/reeb.rs**

```rust
use nalgebra::{Point2, Vector2};
use petgraph::graph::UnGraph;

pub trait SliceFn {
    fn h(&self, q: Point2<f64>) -> f64;
    fn grad_h(&self, q: Point2<f64>) -> Vector2<f64>;
}
pub struct Vertical;
impl SliceFn for Vertical {
    fn h(&self, q: Point2<f64>) -> f64 { q.x }
    fn grad_h(&self, _: Point2<f64>) -> Vector2<f64> { Vector2::x() }
}

#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum Kind { Birth, Split, Merge, Death }

#[derive(Clone, Copy, Debug)]
pub struct CriticalPoint { pub q: Point2<f64>, pub kind: Kind }

/// Nodes are critical points, edges are cells (the dual of the adjacency graph).
pub struct ReebGraph { pub graph: UnGraph<CriticalPoint, usize> }

impl ReebGraph {
    /// Euler's count for a boustrophedon decomposition: e = v + N_ob - 1.
    pub fn euler_check(&self, n_obstacles: usize) -> bool {
        self.graph.edge_count() + 1 == self.graph.node_count() + n_obstacles
    }

    /// chi(Q_free) from Chapter 8's Morse count with boundary: births (index 0) and
   merges
    /// (index 1) are the boundary critical points where ?h points into the free space.
    pub fn morse_chi(&self) -> i64 {
        self.graph.node_weights().map(|c| match c.kind {
            Kind::Birth => 1,
            Kind::Merge => -1,
            Kind::Split | Kind::Death => 0,
        }).sum()
    }
}

pub fn boustrophedon(world: &PolyWorld) -> (Decomposition, ReebGraph) {
    let d = sweep(world, Rule::Boustrophedon);
    let reeb = ReebGraph::from_cells(&d); // one node per critical event, one edge per
   cell
    (d, reeb)
}
```

Coverage: the exhaustive walk is a DFS, the laps are a closed-form set of

abscissae, and the transits are Chapter 6's A* on the trapezoids.

RUST **crates/cells/src/coverage.rs**

```rust
pub struct CoveragePath {
    pub points: Vec<Point2<f64>>,
    pub laps: usize,
    pub lap_len: f64, pub boundary_len: f64, pub transit_len: f64,
}

/// ?width/2r? laps, the outer ones r inside the cell's ends: every point of the cell is
/// within r of a lap horizontally, which is what the missed-area bound needs.
pub fn lap_xs(x_l: f64, x_r: f64, two_r: f64) -> Vec<f64> {
    let n = ((x_r - x_l) / two_r - 1e-9).ceil().max(1.0) as usize;
    if n == 1 { return vec![0.5 * (x_l + x_r)]; }
    let r = 0.5 * two_r;
    let step = (x_r - x_l - two_r) / (n - 1) as f64;
    (0..n).map(|k| x_l + r + k as f64 * step).collect()
}

pub fn coverage_path(d: &Decomposition, world: &PolyWorld, two_r: f64, start: Point2<f64
   >) -> CoveragePath {
    let trapezoids = sweep(world, Rule::Trapezoidal); // convex: safe transits (
   Derivation 2)
    let order = dfs_first_visits(&d.adjacency, d.locate(start).unwrap_or(0));
    let mut path = PathBuilder::at(start);
    for c in order {
        let cell = &d.cells[c];
        let xs = lap_xs(cell.x_range.0, cell.x_range.1, two_r);
        let pattern = Lawnmower::new(cell, &xs).entered_nearest(path.here());
        path.transit(&trapezoids, pattern.first()); // A* + midpoint-centroid chaining
        path.extend(pattern);
    }
    path.finish()
}

/// Delta2/2r + 4DeltaN_ce + 5P_total -- Choset sec.6.2.5.
pub fn bound(world: &PolyWorld, two_r: f64, n_cells: usize) -> f64 {
    let delta = world.min_enclosing_circle().radius * 2.0;
    delta * delta / two_r + 4.0 * delta * n_cells as f64 + 5.0 * world.total_perimeter()
}
```

The online coverer is a state machine around Chapter 3's follower. Its `tick` advances one motion primitive and returns the phase, so the widget and a test harness drive the same code.

RUST **crates/cells/src/online.rs**

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum Phase { Forward, Reverse, Closing, Backtrack, Done }

pub struct Coverer<'r> {
    pub robot: &'r mut sim::Rusty,
    pub reeb: ReebGraph,
    pub two_r: f64,
    pub phase: Phase,
    gap_start: Point2<f64>,   // where the current gap's first lap began
    far_end: Point2<f64>,     // the previous lap's other end, T_k
```

```
    best_progress: f64,       // max x-progress along the current follow
}

impl Coverer<'_> {
    /// One step of the cycle algorithm. Critical points come from the range ring alone.
    pub fn tick(&mut self, scan: &bugs::RangeScan, dt: f64) -> Phase {
        let progress = self.sweep_dir() * (self.robot.pose().x - self.gap_start.x);
        self.best_progress = self.best_progress.max(progress);
        // Derivation 4: the boundary turned back ? the tangent went vertical ? the minimum
        // ray was parallel to the sweep. Classify by where that ray points.
        if self.following() && progress < self.best_progress - 0.3 * self.standoff() {
            let g = scan.gradient();                        // ?D = -(minimum ray)
            let convex = g.x * self.moving_dir() > 0.3;
            let node = self.reeb.add_or_match(self.robot.position(), self.classify(convex));
            self.close_cell(node);
            self.phase = Phase::Backtrack;                  // DFS picks the next unexplored edge
            return self.phase;
        }
        self.phase = match self.phase {
            Phase::Forward if progress >= self.two_r => { self.lap_to_other_side(); Phase::Reverse }
            Phase::Reverse if progress <= 0.0 => {
                if (self.robot.position() - self.far_end).norm() > self.close_tol() {
                    self.lap_toward(self.far_end);          // an obstacle lies between: find why
                    Phase::Closing
                } else {
                    self.undo_reverse();                    // retrace, then the next gap
                    Phase::Forward
                }
            }
            Phase::Backtrack => self.travel_to_next_unexplored(),
            p => { self.robot.follow_boundary(dt); p }
        };
        self.phase
    }
}
```

Pursuit is short once visibility is exact: a polygon from rays, and a BFS whose transitions carry labels by overlap.

RUST **crates/cells/src/pursuit.rs**

```
pub fn visibility_polygon(world: &PolyWorld, x: Point2<f64>) -> Vec<Point2<f64>> {
    const EPS: f64 = 1e-5;
    let mut bearings: Vec<f64> = world.vertices()
        .flat_map(|v| { let a = (v.y - x.y).atan2(v.x - x.x); [a - EPS, a, a + EPS] })
        .collect();
    bearings.sort_by(f64::total_cmp);
    bearings.into_iter().map(|a| x + world.ray_cast(x, a) * Vector2::new(a.cos(), a.sin())).collect()
}

pub struct InfoState { pub cell: usize, pub labels: Vec<u8> }

```

```rust
/// BFS on G_i from (start region, all ones) to any all-zero node -- Choset Fig. 6.39.
pub fn clear(world: &PolyWorld, start: Point2<f64>, static_pursuer: Option<Point2<f64>>) -> Option<Vec<usize>> {
    let regions = conservative_regions(world, static_pursuer);
    let s0 = InfoState { cell: regions.locate(start)?, labels: regions.all_contaminated(start) };
    bfs(s0, |s| regions.crossings(s.cell).map(|t| t.carry(&s.labels)), |s| s.labels.iter().all(|&b| b == 0))
}
```

10.5.1 The worked example, and its printed output

RUST **crates/cells/examples/workbench_counts.rs**

```rust
fn main() {
    let bench = PolyWorld::bench(); // B, T, K of the sec.3 micro-example
    let t = sweep(&bench, Rule::Trapezoidal);
    println!("trapezoidal: {} cells, {} adjacency edges, euler faces = {}",
             t.cells.len(), t.adjacency.edge_count(), t.euler_faces());
    let (b, reeb) = boustrophedon(&bench);
    println!("boustrophedon: {} critical points, {} cells, reeb nodes = {}, reeb edges = {}, euler_check = {}",
             reeb.graph.node_count(), b.cells.len(), reeb.graph.node_count(),
             reeb.graph.edge_count(), reeb.euler_check(bench.obstacles.len()));
    println!("morse chi = {}", reeb.morse_chi());
    println!("laps in cell 0 at 2r = 0.5: {}", lap_xs(0.0, 2.0, 0.5).len());
}

#[test]
fn workbench_counts_match_text() {
    let bench = PolyWorld::bench();
    let t = sweep(&bench, Rule::Trapezoidal);
    assert_eq!((t.cells.len(), t.adjacency.edge_count(), t.euler_faces()), (12, 13, 3));
    let (b, reeb) = boustrophedon(&bench);
    assert_eq!((reeb.graph.node_count(), b.cells.len()), (6, 7));
    assert!(reeb.euler_check(2));
    assert_eq!(lap_xs(0.0, 2.0, 0.5), vec![0.25, 0.75, 1.25, 1.75]);
}
```

```
trapezoidal: 12 cells, 13 adjacency edges, euler faces = 3
boustrophedon: 6 critical points, 7 cells, reeb nodes = 6, reeb edges = 7, euler_check = true
morse chi = -1
laps in cell 0 at 2r = 0.5: 4
```

RUST **crates/cells/examples/mow_apartment.rs · pursue_u.rs**

```rust
fn main() {
    let room = PolyWorld::living_room().in_general_position(); // Room C, sofa and coffee table
    let (d, _) = boustrophedon(&room);
    let start = Point2::new(9.0, 0.8);
    let off = coverage_path(&d, &room, 0.44, start);
```

```
    println!("offline: length {:.2} m, covered {:.2} %", off.length(), 100.0 * covered(&
room, &off.points, 0.22, 0.03));
    for policy in [Policy::Cycle, Policy::Raster] {
        let run = Coverer::run(&room, start, 0.44, policy);
        println!("{policy:?}: length {:.2} m, covered {:.2} %, {} critical points, bound
{:.1} m",
                 run.length(), 100.0 * covered(&room, &run.points, 0.22, 0.03),
                 run.reeb.graph.node_count(), bound(&room, 0.44, run.reeb.graph.
edge_count()));
    }
}
```

```
offline: length 45.86 m, covered 99.87 %
Cycle: length 85.74 m, covered 100.00 %, 6 critical points, bound 308.9 m
Raster: length 21.70 m, covered 79.66 %, 2 critical points, bound 191.3 m

two pockets (20 edges), 1 pursuer: cleared after 286 information states
three pockets (28 edges), 1 pursuer: no all-zero node in 693 states
three pockets (28 edges), 2 pursuers (one static at the hub): cleared after 2480 states
```

The tests pin all of these: `workbench_counts_match_text` as above; `euler_holds_on_random_worlds` asserts $N_{ce} = N_{cp} + N_{ob} - 1$ and $V + N_{ob} - 1$ trapezoids on 300 seeded worlds; `coverage_is_complete` asserts `covered == 100 %` at a 3 cm raster and `length <= bound` for the cycle algorithm in the living room, on the Workbench top and on the bench. Every number above is reproduced by the TypeScript port that runs this page, and the chapter's checks hold it to the printed digits.

10.6 Putting it together: three free spaces, three mowers

The integration lab covers three free spaces three ways: **offline** (the map is known: sweep, walk, mow), **online** with the cycle algorithm (the map is unknown: sense, mow, build the Reeb graph), and online with a conventional **raster** scan. The living room is mowed by disc-Rusty with its own diameter as effector; the Workbench top is wiped by Reach's end-effector pad, 20 cm wide, around the block, the post and the shelf (reachability is ignored here — Chapter 4 maps it); the bench is the §3 micro-world.

world, effector	method	length	covered	critical points / cells	bound
living room, $2r = 0.44$	offline	45.86 m	99.87 %	6 / 7	308.9 m
	cycle	85.74 m	100.00 %	6 / 7	308.9 m
	raster	21.70 m	79.66 %	2 / 1	—
Workbench top, $2r = 0.2$	offline	170.21 m	99.98 %	8 / 10	631.2 m
	cycle	264.41 m	100.00 %	8 / 10	631.2 m
	raster	212.86 m	99.95 %	4 / 5	—
bench, $2r = 0.5$	offline	158.44 m	98.70 %	6 / 7	866.8 m

world, effector	method	length	covered	critical points / cells	bound
	cycle	272.40 m	100.00 %	6 / 7	866.8 m
	raster	190.09 m	86.36 %	4 / 5	—

Three things to read off the table. *Knowing the map is worth a factor of 1.5 to 1.9 in length*: the cycle algorithm follows both boundaries of every gap and retraces its reverse follows, drives a closing lap through every split and merge, and travels between cells over ground it has already covered rather than along the shortest route. *Not knowing it costs nothing in coverage* — the cycle algorithm is the only method at 100 % everywhere, because it follows both the floor and the ceiling of every gap, while the offline lawnmower misses the slivers the missed-area bound allows. *A raster scan's failure is not a constant*: on the Workbench its lucky phase finds the merges that matter and it misses 0.05 %; in the living room it misses a fifth of the room. A method whose completeness depends on the phase of its laps is not complete.

Part II closes here, and its scorecard is worth stating. Every planner in Chapters 6–10 is **complete** — it finds a path when one exists and says so when none does — and most are **exact**: the visibility graph, the GVD, Canny's roadmap, the cell decompositions represent $\mathcal{Q}_{\text{free}}$ without approximation. The price is dimension. A trapezoidal sweep is $O(n \log n)$ in the plane; Canny's roadmap is singly exponential in the dimension of $\mathcal{Q}$; the decompositions of this chapter have no practical analogue for Reach's six-joint cousins. Chapter 11 gives the guarantee up for a probabilistic one and gets the dimension back: it will measure its roadmaps against exactly the kind of decomposition built here, on spaces small enough for both. Chapter 14 reuses the adjacency graph as the discrete layer of task planning, and the sister book's exploration chapter replaces the coverage of a *known-to-be-there* space with the coverage of an unknown one under a belief — frontiers instead of critical points.

> **NOTE**
>
> Two modern pointers, both surveys. Galceran and Carpin (2013) organize coverage path planning since Choset's survey of 2001 — cellular decompositions, grid-based methods, multi-robot coverage — and are the place to start for anything beyond this chapter. Choset's own 2001 survey is the bridge from the boustrophedon decomposition to the Morse decompositions and sensor-based coverage of §6.2.

10.7 Exercises

1 FOUNDATION •• **Counting trapezoids, convex or not**

(a) Derive the trapezoid count $(m-1)+\sum_i(k_i+1)$ for a convex m-gon boundary with convex k_i-gon holes in general position, by counting the cells each vertex opens. (b) Now notch K: replace it by $(6,1),(8.6,0.8),(7.3,2.4),(8.8,4),(6.5,4.4)$, whose vertex $(7.3,2.4)$ is reflex and is the tip of a notch opening to the right. Classify every vertex of the notched K (split, merge, birth, death, pass) and count again. (c) Prove the general rule: trapezoids $= V +$ #split $-$ #death, and #split $-$ #death $= N_{ob}-1$ for every polygonal free space in general position, so the count is always $V+N_{ob}-1$ — reflex vertices do not break it.

WORK IT OUT

How many trapezoids does the bench have with the notched K (five vertices)? (The sweep in this chapter's code agrees with your count.)

Answer. 13 $V = 4+3+5 = 12$, $N_{ob} = 2$: $12+2-1 = 13$. Per vertex, the notched K has a split at $(6,1)$ (+2), a pass-through at $(6.5,4.4)$ (+1), a *birth* at the notch tip $(7.3,2.4)$ (+1: free space starts between two edges that both leave to the right), and *merges* at both lips $(8.6,0.8)$ and $(8.8,4)$ (+1 each) — six, the same as a convex pentagon's $k+1$. The boustrophedon decomposition changes more: $N_{cp}=8$, $N_{ce}=9=8+2-1$.

HINTS

<>Count cells *opened* at each event: split 2, birth 1, pass 1, merge 1, death 0. So trapezoids = V + #split − #death.</> <>Along one closed boundary loop the local minima and maxima of x alternate, so #split + #birth = #merge + #death on every loop.</> <>Derivation 5's Morse count says #birth − #merge = χ(Q_free) = 1 − N_ob. Combine with the previous hint summed over loops.</>

2 FOUNDATION •• **Where a critical point can sit**

Prove that a critical point cannot lie in the interior of a slice interval (Choset §6.2.2): at an interior point the slice meets Q_{free} on both sides. Then, for the squarel $h=|x|+|y|$, show that a corner of the diamond $\{h=\lambda\}$ touching an obstacle vertex is a critical point exactly when the obstacle's normal cone at that vertex meets the convex hull of the normals of the two flat sides meeting at the corner (C §6.2.3) — the generalized-gradient condition of Chapter 9.

HINTS

<>A point of ∂Q_free cannot be in the interior of an interval of Q_free,λ: intervals are open in the slice, and their endpoints are where the slice leaves the free space.</> <>At a corner of the diamond the generalized gradient of h is the convex hull of the two one-sided gradients, e.g. (1, 1) and (1, −1) at the right corner.</> <>Criticality is "0 in the generalized gradient of h restricted to the boundary", i.e. some element of ∂h is parallel to some normal of the obstacle at the point.</>

3 CONCEPTUAL • **Predict, then drag**

PREDICT FIRST

In the Boustrophedon Mower (bench), drag the triangle's top vertex from (3, 4) down to (3, 1.6), just below the chord from (2, 2) to (4, 1.5). Before releasing: what happens to the two cell counts, trapezoidal 12 and boustrophedon 7?

- Trapezoidal drops to 11, boustrophedon stays 7
  The vertex does not lose its extension: it still has free space on one side. What changes is which side — its extension now points down instead of up.
- ✓ Both unchanged: 12 and 7
  The triangle is still a triangle in general position. (3, 1.6) is a pass-through on the lower hull now instead of the upper, so its single extension flips from up to down; V + N_ob − 1 = 12 and N_cp + N_ob − 1 = 7 do not move.
- Boustrophedon rises to 8, because (3, 1.6) becomes critical
  A critical point needs free space both above and below, i.e. both incident edges on the same side of the sweep. One edge of (3, 1.6) goes left and one goes right: it is a pass-through.

Now drag K's vertex $(8.6, 4)$ to $(7.0, 2.6)$. It becomes a reflex vertex, and the counts *still* do not move — but one critical point does: K's merge jumps from $(8.6, 4)$ to $(8, 0.8)$, the new rightmost vertex. Which vertices are critical depends on the shape; how many cells there are depends only on how many.

4 CONCEPTUAL •• **Three pockets, one pursuer**

PREDICT FIRST

In the Pursuit-Evasion widget, pick three pockets with one pursuer. The search exhausts G_i without an all-zero node. Which statement explains why no strategy exists?

- The pockets are too deep to see into from the hub
  Depth alone is not the obstacle: in the two-pocket world the pursuer also has to walk into each pocket, and it succeeds.
- ✓ While the pursuer clears one pocket, the hub is unwatched and the third pocket recontaminates the one already cleared
  That is the recontamination cycle in G_i: every path that drives one pocket's gap label to 0 passes through states where the hub's gap labels return to 1. A static pursuer at the hub holds those labels at 0, and the search then succeeds.
- One pursuer can never clear a polygon with more than 20 edges
  Edge count is not the criterion; a convex polygon with a thousand edges is cleared by one pursuer standing anywhere.

Turn on the second pursuer and replay the solution. Watch which gap labels the static pursuer keeps at 0: the openings of the pockets the mobile pursuer is not in. That is Step 1 of Derivation 7, and the reason three U's need two pursuers and nine need three.

5 PRACTICAL •• **Spiral and squarel critical points on polygons**

Implement `Radial` and `Squarel` slice functions for polygons in `morse.rs`: the critical points of $|q - c|$ restricted to an edge are its endpoints and the foot of the perpendicular from c (if inside the edge); for $|x - c_x| + |y - c_y|$, the endpoints and the points where the edge crosses $x = c_x$ or $y = c_y$. Keep only the candidates where the free slice's *number of intervals* changes (a circle first touching the outer wall does not count), build the Reeb graph by sweeping λ, and assert Euler's count. Render Choset Figs. 6.12 and 6.14 for the bench with $c = (5, 3)$ and compare your counts with the gallery's raster ones (12 and 15 critical points at 8 cm pixels).

6 PRACTICAL ••• **The brushfire decomposition, end to end**

Build the brushfire decomposition (C §6.2.3) from Chapter 7's `brushfire` and Chapter 8's GVD: cells are born at each obstacle, and change where fronts first collide — at saddles of D on the GVD (C Fig. 6.16). Cover each cell by following the obstacle boundary at increasing offsets $D = r, 3r, 5r, \ldots$ with Chapter 3's offset follower,

switching to the GVD where the robot becomes doubly equidistant (C Fig. 6.17). Assert that the Reeb graph is a tree with $N_{ce} = N_{cp} - 1$, and compare the path length with the boustrophedon mower's on the living room. Which pattern would you choose for a robot whose odometry drifts?

10.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

Chapter 6 is this chapter's source: the trapezoidal sweep and its four events (Figs. 6.1–6.8), Morse decompositions (Def. 6.2.1) and the variable slices of Figs. 6.12–6.18, sensor-based coverage and the cycle algorithm (§6.2.4), Euler's count and the path-length bound (§6.2.5), and pursuit–evasion after Suzuki and Yamashita and LaValle, Guibas et al. (§6.3). `link`

Choset, H. (2000). Coverage of Known Spaces: The Boustrophedon Cellular Decomposition. *Autonomous Robots 9(3), 247–253*.

The boustrophedon decomposition and its coverage path: cells only where the sweep line changes connectivity, and the comparison with the trapezoidal decomposition that the Boustrophedon Mower reproduces. `doi:10.1023/A:1008958800904`

Acar, E. U., Choset, H., Rizzi, A. A., Atkar, P. N., and Hull, D. (2002). Morse Decompositions for Coverage Tasks. *International Journal of Robotics Research 21(4), 331–344*.

Definition 6.2.1 in full: critical points of a slice function on the obstacle boundary, and the spiral, spike, squarel and brushfire patterns of the gallery. `doi:10.1177/027836402320556359`

Acar, E. U. and Choset, H. (2002). Sensor-Based Coverage of Unknown Environments: Incremental Construction of Morse Decompositions. *International Journal of Robotics Research 21(4), 345–366*.

Critical-point sensing from range data, the incremental Reeb graph, and the cycle algorithm whose forward, reverse and closing phases the online coverer implements. `doi:10.1177/027836402320556368`

Choset, H. (2001). Coverage for Robotics — A Survey of Recent Results. *Annals of Mathematics and Artificial Intelligence 31, 113–126*.

The survey that frames coverage as exhaustive walks on cell decompositions and places the boustrophedon and Morse decompositions among heuristic, approximate and exact methods. `doi:10.1023/A:1016639210559`

Galceran, E. and Carpin, S. (2013). A Survey on Coverage Path Planning for Robotics. *Robotics and Autonomous Systems 61(12), 1258–1276*.
The modern map of the field: cellular decompositions, grid-based and graph-based coverage, multi-robot coverage, and coverage on 3-D surfaces — where to go after this chapter. doi:10.1016/j.robot.2013.09.004

LaValle, S. M. (2006). Planning Algorithms. *Cambridge University Press*.
Chapter 6 treats cell decompositions (vertical decomposition, cylindrical decomposition) and §12.4 visibility-based pursuit–evasion with information spaces — the decomposition by information of this chapter's coda, at full length. link

Part III

Sampling-Based Planning

CHAPTER 11

Probabilistic Roadmaps

SAMPLING-BASED PLANNING · Intermediate · 75 min

> *PRM fully exploits the fact that it is cheap to check if a single robot configuration is in Q_free or not.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), Chapter 7*

Every planner in Part II built its roadmap by *constructing* $\mathcal{Q}_{\text{free}}$: visibility graphs from obstacle vertices, Voronoi diagrams from equidistance, cells from critical points. Each needs an explicit description of the free space, and each dies as the dimension climbs. Choset's figure 7.1 is a ten-degree-of-freedom arm threading clutter — a problem none of them can touch.

This chapter makes the turn that defined the field after 1996. Stop describing $\mathcal{Q}_{\text{free}}$ and *ask it questions*. A collision checker answers "is q free?" in microseconds. A roadmap built from coarse random samples (the nodes) and fine sampling along local paths (the edges) captures the connectivity of a space nobody wrote down. The price is a weaker promise: the planner is complete only *probabilistically*. The surprise is that the weakness is quantifiable — PRM's failure probability decays *exponentially* in the sample count, with constants computable from the length and clearance of a path you will never see.

The sister book's Chapter 20 previews PRM in a dozen lines. This is the canonical treatment it links to, and it owns the sampling machinery — samplers, the nearest-neighbor structure, the `Steer` trait — that the rest of Part III builds on.

◎ AFTER THIS CHAPTER YOU CAN

- Build a roadmap of a configuration space nobody wrote down, from one predicate — "is q free?" — and two resolutions of random sampling, and state exactly what the resulting planner promises

- Define probabilistic completeness as a limit statement, and prove the three Choset §7.4 results that make it quantitative: ball tiling along a known path, $(\varepsilon, \alpha, \beta)$-expansiveness, and abstract path tiling
- Compute the constants: for a path of length 1.2 and clearance 0.1 in a unit square, 892 uniform samples bound the failure probability below 1 percent — and the expansiveness bound asks for 15 552
- Choose a metric on T^2 and SE(2), build a kd-tree whose pruning bound respects periodic axes, and run a straight-line local planner with the incremental or the subdivision check schedule
- Say where the Gaussian, bridge, OBPRM and medial-axis samplers put their mass, why none of them wins everywhere, and what a finite collision-check step can miss
- Write the sampling crate — Sampler, NearestNeighbors, Steer, Prm generic over Chapter 5's Manifold — that Chapters 12, 13, 14, 21 and 22 import without change

ASSUMED

- The collision checker and the three worlds (Chapter 2) — the only question this chapter asks of the world
- C-obstacles as sets that exist but are never computed here; Reach's torus (Chapter 4)
- Manifolds with dist, interpolate and sample on S^1, T^2 and SE(2) (Chapter 5)
- Dijkstra and A* for the query phase (Chapter 6); roadmap properties — accessibility, connectivity, departability (Chapter 8)

11.1 The problem: a slot nobody can describe

Put two bars on the Workbench, leaving a half-metre gap between them, and ask Reach to bring its forearm from the home pose to a pose inside the gap.

INTERACTIVE · `w11.0`

Reach through the slot

Explicit geometry fails where sampling succeeds: six hundred random configurations and a straight-line local planner thread an arm through a slot that no visibility vertex or cell boundary describes.

Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch11-prm#w11.0`

Think about what Part II would do with this. Chapter 8's visibility graph wants obstacle *vertices* in configuration space, and the C-obstacle of the bars on the torus has no vertices — it is a curved amber region we can only sample, as the right-hand panel does. Chapter 10's exact decomposition of the torus would run to thousands of cells for one slot, and every new object on the bench would rebuild all of them. Neither can start until someone has written $\mathcal{Q}_{\text{free}}$ down.

The planner in the figure never did. It drew six hundred configurations at random, kept the ones the Chapter 2 checker called free, joined each to its six nearest neighbors with a straight line in joint space when the checker approved every point along it, and asked Chapter 6's Dijkstra for a path. The only thing it knows about the world is the answer to one yes/no question, asked a few tens of thousands of times.

That is the whole idea. The rest of the chapter is about making it precise — what the planner promises, why the promise is weaker than Part II's, and exactly how much weaker.

11.2 Building intuition: coarse rain, fine rain

A probabilistic roadmap samples at two resolutions. **Coarse rain**: configurations fall at random, and the ones that land in $\mathcal{Q}_{\text{free}}$ become nodes. **Fine rain**: for each node and each of its k nearest neighbors, a *local planner* proposes a short path — here, a straight line — and walks it in small steps, asking the checker at each one. If every step is free, the edge is added. Everything the chapter does is a choice about where one of the two rains falls.

> INTERACTIVE · w11.1
>
> **Roadmap Rain**
>
> A PRM is a random graph, but whether it connects is not luck: the component count collapses the same way on every seed — luck lives in the constants, not the shape.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch11-prm#w11.1

Watch the component counter, not the dots. At first every node is its own component; then components swallow one another in bursts; then the count sits at two or three for a long time while the rain fills in regions that were already connected. The frame where q_{start} and q_{goal} first share a component is marked. Now re-roll the seed. The dots are all different. The curve is the same shape, and the marker lands in the same neighborhood of n. **A PRM is a random graph, but whether it connects is not luck.** The randomness lives in the constants — how many samples it takes — and the constants are the subject of the Foundation section.

Two more things to notice. The planner is *metered* in two currencies: calls

to the local planner Δ (how many edges were tried) and collision checks (how many points were asked about). The first is the coarse rain's cost, the second the fine rain's, and every comparison in Part III is in the second unit, because that is the unit Choset's quotation above is about. And the torus tab runs *the same code* on Reach's T^2 as the Apartment tab runs on Rusty's $\mathbb{R}^2$ — the planner asks the manifold for distances and interpolants and never does arithmetic on coordinates, which is what Chapter 5 was for.

> **THE ONE SENTENCE VERSION**
>
> A probabilistic roadmap is a graph whose nodes are random free configurations and whose edges are local paths a checker approved, built from the single predicate "is q free?"; it is complete only in the limit, and the rate at which the failure probability vanishes is exponential in the number of samples with constants set by the clearance and length of a path that exists.

11.3 The mathematics

$\Delta(q, q')$	the local planner: a collision-free local path from q to q', or NIL Choset §7.1.1
$\mathrm{dist}(q, q')$, $\mathrm{emb}(q)$	a distance function on $\mathcal{Q}$; the embedding of q by p robot points into $\mathbb{R}$^{p·dim $\mathcal{W}$}
N_q	the k closest neighbors of q among the nodes V under dist
$\mathrm{clr}(\gamma)$, L	clearance and length of a known path γ
$\mu(A)$, $B_\rho(x)$, σ	Lebesgue measure; the open ball of radius ρ at x; $\sigma = \mu(\mathrm{B}_1)$ / (2^d μ($\mathcal{Q}$_free)) Theorem 7.4.1
$\mathrm{reach}(S)$, $\mathrm{lookout}_\beta(S)$	the set of points Δ connects to some point of S; the part of S that sees at least a β fraction of the rest of its component Defs. 7.4.2–7.4.4
$x\ R\ y$, R^ℓ	the abstract local-planner relation; its ℓ-fold iterate Theorem 7.4.3
$D(P, \mathcal{R})$, $\delta(P, \rho)$	discrepancy and dispersion of a point set P Defs. 7.1.1–7.1.2
n, k, $(\epsilon, \alpha, \beta)$	sample count; neighbors per node; expansiveness constants book-wide

11.3.1 Two phases and one predicate

DEFINITION — PROBABILISTIC ROADMAP, LEARNING AND QUERY PHASES (CHOSET §7.1.1)

A **probabilistic roadmap** is an undirected graph $G = (V, E)$ with $V \subset Q_{\text{free}}$ a set of sampled configurations and $(q, q') \in E$ iff $q' \in N_q$ and $\Delta(q, q') \neq$ NIL. The **learning phase** builds G (Choset Algorithm 6); the **query phase** connects q_{start} and q_{goal} to G by the same Δ and searches (Algorithm 7). The phases may interleave: a roadmap that fails a query can be grown and asked again.

Three decisions hide inside that definition, and §7.1.2 of Choset is a list of them: what `dist` means, so that "nearest" is defined; what Δ is, so that an edge can be proposed; and how Δ decides, so that an edge can be accepted. The chapter takes them in order after saying what the planner promises.

11.3.2 What "probabilistically complete" means

Part II's planners were *complete*: a path exists if and only if they find one, in finite time. Chapter 6's grid search was *resolution complete*: complete relative to a discretization, so that a path thinner than a cell can be missed. This chapter's planners make a third kind of promise.

DEFINITION — PROBABILISTIC COMPLETENESS (CHOSET §7.4)

A sampling-based planner is **probabilistically complete** if, for every solvable query (a, b),

$$\lim_{n \to \infty} \Pr\left[\text{the planner reports FAILURE on } (a, b) \text{ after } n \text{ samples}\right] = 0.$$

Resolution completeness is the deterministic analogue — the failure probability is zero or one, and reaches zero once the sampling resolution is fine enough.

Read the quantifiers carefully, because every honest sentence about PRM depends on them. The statement is about a *limit*. At any finite n the planner can report "no path" when one exists, and nothing in the definition says how often. The content of §7.4 is that, under hypotheses on the space, the limit is approached *exponentially fast* — and the content of this chapter's honesty items is that the exponent's constants can be hopeless.

11.3.3 Metrics, embeddings, and a kd-tree that knows about seams

"Nearest" needs a metric, and the metric is the first place a planner can quietly go wrong. On $\mathbb{R}^2$ the Euclidean distance is the obvious choice. On Reach's torus the right choice is Chapter 5's d_{T^2}, the ℓ^2 norm of per-joint circular distances, so that $\theta = 3.1$ and $\theta = -3.1$ are 0.08 apart, not 6.2. On Hitch's SE(2) the chapter uses the weighted metric $w_t\|X' - X''\| + w_r|\theta' - \theta''|_{S^1}$ with an exchange rate between metres and radians that is *a parameter, not a fact* — Choset's §7.1.2 is explicit that no natural metric exists on SE(2), and the book keeps it visible.

Choset's alternative is the **embedding**: pick p points on the robot, map a configuration to the $p \cdot \dim \mathcal{W}$ vector of their workspace positions, and use Euclidean distance there. It is a metric on the robot's actual geometry, it is what Kavraki's original planner used, and it costs a forward kinematics evaluation per distance. The book's `Manifold::dist` is the default; `emb` is an exercise.

Given a metric, N_q asks for the k nearest of up to a thousand nodes, a thousand times. Choset recommends the kd-tree of de Berg, van Kreveld and Overmars: $O(d\, n \log n)$ to build, $O(n^{1-1/d} + m)$ per range query returning m points. A textbook kd-tree splits on coordinates and prunes a subtree when the query's distance to the subtree's box exceeds the current best. On a torus that bound is wrong: a box touching the seam at $\theta = \pi$ is *adjacent* to a query at $\theta = -3.1$, and a flat bound would prune the very subtree the answer lives in.

> **INTERACTIVE · `f11.3`**
>
> **kd-tree over the torus**
>
> A kd-tree splits on chart coordinates but must measure with the manifold's metric: a neighbor across the seam is a real neighbor, and the periodic pruning bound is what keeps it from being thrown away.
>
> Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch11-prm#f11.3`

The fix is one line. The distance from a query coordinate c to a box edge s along a periodic axis of period T is $\min(|c - s|,\ T - |c - s|)$, and the bound on the box is the ℓ^2 sum of per-axis gaps, each scaled by the metric's weight on that axis. The bound never exceeds the true distance, so pruning never discards a real neighbor — the check `kd-tree (periodic axes) equals brute force on` $\mathbb{R}^2$ `and` T^2 compares two thousand torus points and two hundred queries, seam-crossing neighbors included, against a linear scan.

11.3.4 The local planner and its two schedules

$\Delta(q, q')$ is the fine rain. The straight-line planner follows the manifold's own geodesic — `interpolate`, which on T^2 goes the short way through the seam — and samples it at a resolution `step` far finer than the node sampling. Two orders

of asking are in Choset's figure 7.6.

- **Incremental**: $q_1, q_2, \ldots$ in order from one end; stop at the first collision.
- **Subdivision**: the midpoint first, then the quarter points, then the eighths — breadth-first over intervals. A blocked edge is usually blocked somewhere in its middle, and the midpoint finds that first; a free edge of length ℓ is declared free after $\lceil \ell/\text{step} \rceil$ checks either way.

Hover any edge in Roadmap Rain to see the numbered checks of the active schedule. Subdivision tends to win [C refs. 162, 367] because an edge that is going to fail fails sooner. Both share a flaw the prose must keep: **a finite step can miss a collision thinner than the step.** An obstacle 0.2 m wide between two checks 0.25 m apart is invisible to both schedules. Only a distance-based schedule — march by the checker's *distance* to the nearest obstacle, so that consecutive checks are guaranteed to overlap, as Chapter 2's `edgeFree` does — cannot miss, at the price of a distance query instead of a boolean one. The book's `StraightLine` is the fixed-step version because that is what Choset analyzes; the chapter says so every time it matters.

After a path is found, **shortcutting** (figure 7.7) tries to connect q_{start} straight to q_{goal}, then to the configuration before it, and so on, replacing jagged pieces by geodesics between their own points. It never lengthens a path and never changes its homotopy class. Chapter 13 will show that this is why it is postprocessing and not optimization.

11.3.5 The sampler zoo

Uniform sampling is the baseline, and Theorem 7.4.1 is about it. Everything else in §7.1.3 is a bias toward where uniform sampling is poor: near obstacles, and inside narrow passages.

- **Gaussian** [C ref. 59, Boor, Overmars and van der Stappen]: draw q_1 uniformly, q_2 a Gaussian step of scale σ from it, and keep the free one only if the other is in collision.
- **Bridge test** [C ref. 193, Hsu, Jiang, Reif and Sun]: draw two configurations a random length apart; if *both* are in collision and their midpoint is free, keep the midpoint.
- **OBPRM** [C ref. 18, Amato et al.]: from a configuration in collision, march in a random direction until free, then bisect back toward the surface.
- **Medial-axis retraction** [C ref. 427, Wilmarth, Amato and Stiller]: push a free sample up the clearance function until a second obstacle becomes equidistant — onto Chapter 8's generalized Voronoi diagram.
- **Quasirandom sequences**: Halton points replace the random generator with a deterministic low-discrepancy sequence; the planner becomes resolution complete rather than probabilistically complete, and two runs at the same n give the same roadmap.

Where do the biased samplers put their mass? The question has a clean answer.

DERIVATION **Where the Gaussian and bridge samplers put their mass**

\text{Gaussian: within } O(\sigma) \text{ of } \partial\QO;\quad \text{bridge: inside corridors narrower than the bridge}

Step 1 — acceptance as an integral. A two-point sampler accepts q with probability equal to the integral of the collision indicator over the partner distribution centered at q. For the Gaussian sampler, the density of accepted samples at a free q is

$$p_{\text{acc}}(q) \;\propto\; \int_{\mathcal{QO}} \mathcal{N}(q'; q, \sigma^2 I)\, dq',$$

the probability that a σ-ball around q *straddles* $\partial\mathcal{QO}$.

Step 2 — a half-space obstacle. If $\mathcal{QO}$ is the half-space at distance $\text{dist}(q)$ from q, the integral is $\Phi(-\text{dist}(q)/\sigma)$ with Φ the standard normal CDF: one half at the boundary, 0.16 at one σ, 0.02 at two. The mass concentrates within $O(\sigma)$ of every boundary — the walls of a narrow passage, and also the outer wall of an empty room.

Step 3 — both piers in collision. The bridge test accepts the midpoint of q_1, q_2 when both endpoints are in collision and the midpoint is free. In a corridor of width w between two obstacles, a bridge of length $\ell > w$ laid across it has both piers in collision for a large fraction of directions; in open space, a bridge has both piers in collision in almost no direction, because the nearest obstacle is one-sided. The sampler is therefore nearly *silent* in the open and loud exactly in corridors narrower than its bridge — which is why it is run as a hybrid with uniform sampling, as its authors recommend and as this chapter's widgets do.

Step 4 — OBPRM lands within one step. Bisection between a colliding and a free configuration halves the interval each round, so after r rounds the accepted sample is within 2^{-r} of the initial step of $\partial\mathcal{QO}$, on the free side.

The medial-axis sampler is the odd one out: it pushes mass *away* from boundaries, toward the set of maximal clearance, and in a corridor that set is the center line. Choset closes §7.1.3 with the honest summary — hybrids $\pi = (1 - w)\,\pi_A + w\,\pi_B$ and samplers used as *filters* on one another exist because no single strategy wins everywhere, and pathological instances exist for every one. ■

DEFINITIONS 7.1.1–7.1.2 — DISCREPANCY AND DISPERSION

For a point set $P \subset [0,1]^d$ and a family $\mathcal{R}$ of ranges, the **discrepancy** $D(P, \mathcal{R}) = \sup_{R \in \mathcal{R}} \left| |P \cap R|/|P| - \mu(R) \right|$ measures how far P is from counting volume. The **dispersion** $\delta(P, \rho) = \sup_x \min_{p \in P} \rho(x, p)$ is the radius of the largest empty ball. Under L_∞ the Sukharev grid achieves $\delta = 1/(2\lfloor N^{1/d} \rfloor)$, and no N-point set does better — so reaching dispersion δ^* needs $N \geq (1/2\delta^*)^d$ points. Dispersion costs exponentially in dimension, random or not.

11.3.6 Connection strategies

The k-nearest rule is the default. Three refinements are in §7.1.3–7.1.4.

- **Sparse roadmaps**: connect a new node only to the nearest node of each *other* component plus a few redundant neighbors. The result is nearly a forest; it has the connectivity and few of the edges.
- **Connection sampling**: after the first pass, pick nodes with probability inversely related to their degree, sample around them, and connect as usual. It never creates a component on its own and tends to join the ones that remain.
- **Lazy evaluation** [C ref. 53, Bohlin and Kavraki]: add every candidate edge *unchecked*, search, and verify only the edges on the returned path, coarse-to-fine; delete a blocked edge and search again. Collision checks are spent only where a path wanted to go.

Each is a `Connect` variant in the implementation, and Exercise 5 asks for the lazy one's accounting.

11.3.7 Theorem 7.4.1 — the ball tiling

The first analysis assumes nothing about the space except that a solution exists with some room around it.

THEOREM 7.4.1 (KAVRAKI, KOLOUNTZAKIS AND LATOMBE; CHOSET §7.4.1)

Let $\mathcal{Q}_{\text{free}} \subset [0,1]^d$ and let γ be a path of length L and clearance $\rho > 0$ joining $a, b \in \mathcal{Q}_{\text{free}}$. After n uniform samples, with Δ the straight-line planner and every pair of samples within ρ of each other tried,

$$\Pr[\text{FAILURE}] \;\leq\; \left\lceil \frac{2L}{\rho} \right\rceil e^{-\sigma \rho^d n}, \qquad \sigma = \frac{\mu(B_1)}{2^d\, \mu(\mathcal{Q}_{\text{free}})}.$$

DERIVATION Theorem 7.4.1 — tiling a path with balls

$\Pr[\text{FAILURE}] \leq m\,(1 - \sigma\rho^d)^n \leq m\, e^{-\sigma \rho^d n}$

Step 1 — tile the path. Place $m = \lceil 2L/\rho \rceil$ points $x_1 = a, x_2, \ldots, x_m = b$ along γ so that consecutive points are less than $\rho/2$ apart. Around each put the ball $B_{\rho/2}(x_i)$. Since $\mathrm{clr}(\gamma) = \rho$, every such ball lies inside the larger free ball $B_\rho(x_i) \subset \mathcal{Q}_{\text{free}}$.
Step 2 — any two consecutive balls see each other. Take $y_i \in B_{\rho/2}(x_i)$ and $y_{i+1} \in B_{\rho/2}(x_{i+1})$. Both lie in $B_\rho(x_i)$: the first trivially, the second because $|y_{i+1} - x_i| \le |y_{i+1} - x_{i+1}| + |x_{i+1} - x_i| < \rho/2 + \rho/2$. A ball is convex, so the segment $\overline{y_i\, y_{i+1}}$ lies in $B_\rho(x_i) \subset \mathcal{Q}_{\text{free}}$ and the straight-line Δ succeeds on it.
Step 3 — one sample per ball is a roadmap path. If every ball $B_{\rho/2}(x_i)$ contains at least one sample, Step 2 chains them into a sequence of roadmap edges from a sample near a to a sample near b; a and b themselves connect to their balls' samples by the same argument. The query succeeds.
Step 4 — union bound. FAILURE therefore requires some ball to be empty:

$$\Pr[\text{FAILURE}] \;\le\; \sum_{i=1}^{m} \Pr\left[B_{\rho/2}(x_i) \cap V = \emptyset\right].$$

Step 5 — one empty ball. A uniform sample over $\mathcal{Q}_{\text{free}}$ lands in $B_{\rho/2}(x_i)$ with probability $\mu(B_{\rho/2})/\mu(\mathcal{Q}_{\text{free}}) = \mu(B_1)\,(\rho/2)^d/\mu(\mathcal{Q}_{\text{free}}) = \sigma\rho^d$. The n samples are independent, so the ball is empty with probability $(1 - \sigma\rho^d)^n$.
Step 6 — the exponential. $(1-\beta)^n \le e^{-\beta n}$ for $\beta \in [0,1]$, giving the theorem. ■

Where the k-nearest rule breaks the proof. Step 2 assumes every pair of samples within ρ is tried. The k-nearest rule tries only k pairs per node, and nothing stops all k from lying in the *same* ball while the neighbor in the next ball goes untried. Exercise 1 asks for the picture. The repair is either a radius rule — connect everything within r — or a k that grows with n; Chapter 13's PRM* radius r_n is exactly that repair, and the chapter proves it is also the smallest one that keeps the per-node work logarithmic. Kavraki, Kolountzakis and Latombe [C ref. 223] extend the theorem to paths of varying clearance by tiling with balls of varying radius.

Now the numbers, because the shape of a bound is cheap and its constants are the point. Take $d = 2$, a free space occupying $\mu(\mathcal{Q}_{\text{free}}) = 0.9$ of the unit square, a path of length $L = 1.2$ with clearance $\rho = 0.1$. Then $\sigma = \pi/(4 \cdot 0.9) = 0.8727$, $\sigma\rho^2 = 0.008727$, and $m = \lceil 24 \rceil = 24$ balls. The bound is $24\, e^{-0.008727\, n}$:

n	**bound**
500	0.306
892	0.0100 — the smallest n at or below one percent

n	**bound**
1000	0.0039

Exponential in shape; nine hundred samples for a two-dimensional problem a child could solve by eye. Theorem 7.4.1 is a *proof* that PRM works, not a *budget*, and the gap between the two is where the sampler zoo earns its keep. The check `Theorem 7.4.1 constants` pins all five numbers.

11.3.8 Theorem 7.4.2 — expansive spaces

The second analysis drops the known path and characterizes the *space*. Three definitions, each about how much of $\mathcal{Q}_{\text{free}}$ a point can see.

DEFINITIONS 7.4.2–7.4.5 — ε-GOOD, β-LOOKOUT, (ε, α, β)-EXPANSIVE, LINKING SEQUENCE

Let $\text{reach}(x) = \{\, y \in \mathcal{Q}_{\text{free}} : \Delta(x, y) \neq \text{NIL} \,\}$ be what x sees, and $\text{reach}(S) = \bigcup_{x \in S} \text{reach}(x)$.

- $\mathcal{Q}_{\text{free}}$ is **ϵ-good** if every x sees at least an ϵ fraction of its component: $\mu(\text{reach}(x)) \geq \epsilon\, \mu(\mathcal{Q}_{\text{free}})$.
- The **β-lookout** of a connected S is the part of S that sees at least a β fraction of what S does not: $\text{lookout}_\beta(S) = \{\, x \in S : \mu(\text{reach}(x) \setminus S) \geq \beta\, \mu(\mathcal{Q}_{\text{free}} \setminus S) \,\}$.
- $\mathcal{Q}_{\text{free}}$ is **$(\epsilon, \alpha, \beta)$-expansive** if it is ϵ-good and every connected S has $\mu(\text{lookout}_\beta(S)) \geq \alpha\, \mu(S)$: a fixed fraction of any region is lookout.
- A **linking sequence** from x is $x_1 = x$, $x_i \in \text{lookout}_\beta(X_{i-1})$ with $X_i = X_{i-1} \cup \text{reach}(x_i)$: each new point was chosen from the part of the current reach set that sees a lot of new space.

Choset's figure 7.19 gives the intuition: two squares of side W joined by a corridor of width w and length W. A point in mid-corridor sees a sliver of each room, a point in a room sees most of that room and almost nothing beyond the corridor, so $\epsilon \approx \alpha \approx \beta \approx w/W$. Exercise 2 asks for the computation; the widget below measures what the estimate predicts.

INTERACTIVE · w11.2

Narrow Passage

Uniform sampling does not see everything at practical sample counts — and biased sampling is not always better: the Gaussian sampler loses to uniform once the corridor is wide.

Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/`

ch11-prm#w11.2

THEOREM 7.4.2 (HSU, LATOMBE AND MOTWANI; CHOSET §7.4.2)

If $\mathcal{Q}_{\text{free}}$ is $(\epsilon, \alpha, \beta)$-expansive and the planner draws $2n+2$ uniform samples with

$$n = \left\lceil \frac{8 \ln(8/\epsilon\alpha\delta)}{\epsilon\alpha} + \frac{3}{\beta} \right\rceil,$$

then with probability at least $1-\delta$, every connected component of $\mathcal{Q}_{\text{free}}$ is captured by one connected component of the roadmap.

DERIVATION Theorem 7.4.2 — linking sequences and overlapping reach sets

\Pr[\text{some component split}] \le \delta \text{ when } n \ge \lceil 8\ln(8/\epsilon\alpha\delta)/\epsilon\alpha + 3/\beta \rceil

Step 1 — Lemma 7.4.6: a sample has a long linking sequence. Let $s = 1/\alpha\epsilon$. For a sample $x \in V$, the probability that V contains no linking sequence of length t starting at x is at most $s\, e^{-(n-t-1)/s}$. Sketch: at each stage the lookout of the current reach set has measure at least $\alpha\mu(X_{i-1}) \ge \alpha\epsilon\, \mu(\mathcal{Q}_{\text{free}})$ by expansiveness and ϵ-goodness, so each of the remaining samples lands in it with probability at least $1/s$; the chance that $n-t-1$ independent tries all miss is $(1-1/s)^{n-t-1} \le e^{-(n-t-1)/s}$, and a union over the t stages contributes the factor s after some bookkeeping.
Step 2 — Lemma 7.4.7: a long linking sequence sees most of the component. With $t \ge \beta^{-1} \ln 4$, the reach set X_t of a linking sequence covers at least three quarters of the component: each step adds at least a β fraction of what is still uncovered, so the uncovered measure shrinks by $(1-\beta)$ per step and $(1-\beta)^t \le e^{-\beta t} \le 1/4$.
Step 3 — two such sets overlap. If x and y in the same component both have linking sequences with $\mu(X_t), \mu(Y_t) \ge \frac{3}{4}\mu$, then $\mu(X_t \cap Y_t) \ge \frac{1}{2}\mu$ of the component.
Step 4 — a third batch hits the overlap. Draw a further n samples. One lands in the overlap and is therefore connected to both x's and y's roadmap components with probability at least $1-(1-1/2)^n \ge 1-e^{-n/2}$ — Choset carries the sharper $1-e^{-n\epsilon/2}$ through the ϵ-goodness of the overlap's points.
Step 5 — union bound over pairs. There are $\binom{n}{2}$ pairs of first-batch samples; each fails to be joined with probability at most the sum of the two Lemma

7.4.6 terms and the Step 4 term. Requiring the total to be at most δ and **Step 6 — solving for** n gives the stated expression, with $3/\beta$ accounting for the linking-sequence length t of Step 2 and the logarithm absorbing the $\binom{n}{2}$ and the constant s. ■

Put in the corridor's numbers. At $\epsilon = \alpha = \beta = 0.1$ and $\delta = 0.05$ the theorem asks for $n = 7775$, that is $2n + 2 = 15{,}552$ samples. The narrow-passage widget connects a corridor with $w/W = 0.125$ on every seed with a hundred and fifty. **Right shape, hopeless constants** — the theorem's value is not the number but the *dependence*: $1/\epsilon\alpha$ in front of a logarithm, so halving the corridor roughly quadruples the samples the bound demands, and the widget's uniform curve falls off in just that way as w shrinks.

11.3.9 Theorem 7.4.3 — abstract path tiling

The third analysis is the most general and, once seen, the simplest. It does not care what Δ is or how samples are drawn.

THEOREM 7.4.3 (LADD AND KAVRAKI; CHOSET §7.4.3)

Let $R \subset Q_{\text{free}} \times Q_{\text{free}}$ be any measurable relation used as the local planner ($x\ R\ y$ means $\Delta(x, y) \neq \text{NIL}$) and μ any sampling measure. PRM with (R, μ) is probabilistically complete iff Algorithm 15 — guess $x_{\ell+1} \sim \mu$, require $x_\ell\ R\ x_{\ell+1}$, stop when $x_\ell\ R\ y$ — succeeds with nonzero probability on every solvable query. When it does, there exist ℓ and $p > 0$ with $\Pr[\text{FAILURE after } n \text{ samples}] \leq \ell\, e^{-pn}$.

DERIVATION **Theorem 7.4.3 — both directions, and the exponential**

$\Pr[\text{FAILURE}] \leq \ell\,(1-p)^n \leq \ell\, e^{-pn}$

Step 1 — forward. A run of Algorithm 15 that succeeds in ℓ guesses produces $x_1\ R\ x_2\ R\ \cdots\ R\ x_\ell\ R\ y$. The x_i are draws from μ and every consecutive pair is R-related, so the same draws, handed to PRM, produce a roadmap containing that path. PRM succeeds whenever Algorithm 15 does.
Step 2 — converse. Suppose PRM succeeds with probability $P > 0$ using n nodes, and take a minimal successful roadmap. Its success does not depend on the order in which the nodes were drawn, so one of the $n!$ orderings lists the solution path's nodes first and in path order — and that ordering is a successful run of Algorithm 15 with probability at least $P/n!$.

Nonzero is nonzero.
Step 3 — rectangles. The set $S \subset R^\ell$ of successful ℓ-sequences has positive μ^ℓ-measure. Any measurable set of positive product measure contains a product $A_1 \times \cdots \times A_\ell$ of positive measure — so pick one, with every $A_i \; R \; A_{i+1}$ pairwise.
Step 4 — the rate. Let $p = \min_i \mu(A_i)$. Each of the ℓ rectangles receives one of the n samples with probability $1 - (1 - \mu(A_i))^n \geq 1 - (1 - p)^n$.
Step 5 — union bound. FAILURE requires some A_i to be empty, which happens with probability at most $\ell(1 - p)^n \leq \ell \, e^{-pn}$. ■
What was never used. Symmetry of R and reflexivity were not needed: an **asymmetric** local planner — Chapter 21's Dubins paths, which cannot be driven backward — gives a directed roadmap and the same theorem. Nor was it required that μ be uniform or that Q be a configuration space: time as an extra coordinate, which Chapter 14 previews for kinodynamic planning, is covered as written.

11.4 The algorithm

ALGORITHM BASIC PRM -- roadmap construction (Choset Algorithm 6) COST
n samples (plus rejections) + n k-nearest-neighbor queries + at most nk calls to Δ, each ⌈dist/step⌉ collision checks

in n, the number of nodes; k, the number of neighbors; a sampler; a distance function dist; a local planner Δ
out a roadmap G = (V, E)

1: $V \leftarrow \emptyset$; $E \leftarrow \emptyset$
2: **while** $|V| < n$ **do**
3: **repeat** $q \leftarrow$ a random configuration **until** q is collision-free
4: $V \leftarrow V \cup \{q\}$
5: **for all** $q \in V$ **do**
6: $N_q \leftarrow$ the k closest neighbors of q in V under `dist`
7: **for all** $q' \in N_q$ **do**
8: **if** $(q, q') \notin E$ **and** $\Delta(q, q') \neq$ NIL **then** $E \leftarrow E \cup \{(q, q')\}$
9: **return** $G = (V, E)$

Two orders are both in Choset and both in the implementation. `build(n)` is Algorithm 6 literally: all nodes first, then every node's k nearest among *all* nodes — the order the worked example below is traced under. `grow(n)` is the incremental form of §7.1.1's "adding to the roadmap": each new node connects

to its k nearest among the nodes already present, which is what lets Roadmap Rain scrub the build by time, and what makes the planner single-query when q_{start} and q_{goal} are the first two nodes (§7.2).

> **ALGORITHM** SOLVE QUERY (Choset Algorithm 7) COST at most 2k calls to Δ plus one Dijkstra or A* (Chapter 6) over G
>
> **in** q_start, q_goal, k, the roadmap G
>
> **out** a path from q_start to q_goal, or FAILURE
>
> ---
>
> 1: $N_{q_{\text{start}}} \leftarrow$ the k closest neighbors of q_{start} in V; likewise $N_{q_{\text{goal}}}$
> 2: $V' \leftarrow V \cup \{q_{\text{start}}, q_{\text{goal}}\}$; $E' \leftarrow E$
> 3: **for** q' in $N_{q_{\text{start}}}$ in order of increasing `dist` **do if** $\Delta(q_{\text{start}}, q') \neq$ NIL **then** add (q_{start}, q') to E' and **break**
> 4: **for** q' in $N_{q_{\text{goal}}}$ in order of increasing `dist` **do if** $\Delta(q_{\text{goal}}, q') \neq$ NIL **then** add (q_{goal}, q') to E' and **break**
> 5: $P \leftarrow$ shortest path from q_{start} to q_{goal} in $G' = (V', E')$
> 6: **if** P is not empty **then return** P **else return** FAILURE

The implementation prefers, among q_{goal}'s candidates, those already in q_{start}'s component — a cheap way to avoid attaching the two endpoints to two components that will never meet. And it can report FAILURE falsely: the roadmap may simply not have seen the passage yet. That is the honest reading of "probabilistically complete", and the planner's answer to it is to grow and ask again.

11.4.1 The six-sample roadmap, by hand

The example the implementation is tested against is small enough to trace. A point robot lives in $\mathcal{W} = [0, 10]^2$ with one wall $\mathcal{WO} = [4.5, 5.5] \times [0, 7]$, so $\mathcal{QO} = \mathcal{WO}$ and the gap is above $y = 7$. Six free samples arrive in order: $s_1 = (1, 2)$, $s_2 = (3, 5)$, $s_3 = (2, 8)$, $s_4 = (7, 2)$, $s_5 = (8, 6)$, $s_6 = (6, 9)$. The metric is Euclidean, Δ is the straight line with subdivision at step 0.25. The distances that matter:

pair	distance	pair	distance
s_1s_2	$\sqrt{13} = 3.606$	s_1s_4	6.000
s_2s_3	$\sqrt{10} = 3.162$	s_2s_4	5.000
s_3s_6	$\sqrt{17} = 4.123$	s_2s_6	5.000
s_4s_5	$\sqrt{17} = 4.123$	s_2s_5	5.099
s_5s_6	$\sqrt{13} = 3.606$		

$k = 1$. Each sample's single nearest neighbor: $s_1 \to s_2$, $s_2 \to s_3$, $s_3 \to s_2$, $s_4 \to s_5$, $s_5 \to s_6$, $s_6 \to s_5$. The candidate edges are $\{s_1s_2, s_2s_3, s_4s_5, s_5s_6\}$, every

one of them lies on one side of the wall, and Δ accepts all four. The roadmap has **two components**, $\{s_1, s_2, s_3\}$ and $\{s_4, s_5, s_6\}$. It has not seen the gap.

$k = 2$**.** The second neighbors add s_1s_4, s_2s_4 and s_3s_6 (the others repeat). The segment s_1s_4 runs along $y = 2$ straight through the wall: NIL. The segment s_2s_4 crosses the wall's x-band at $y \in [3.125, 3.875]$, well below the gap: NIL. The segment s_3s_6 crosses the band at $y \in [8.625, 8.875]$, above $y = 7$: free. Five edges, $\{s_1s_2, s_2s_3, s_3s_6, s_5s_6, s_4s_5\}$, **one component**, and the roadmap path from s_1 to s_4 goes the long way round the top:

$$s_1 \longrightarrow s_2 \longrightarrow s_3 \longrightarrow s_6 \longrightarrow s_5 \longrightarrow s_4, \qquad 3.606 + 3.162 + 4.123 + 3.606 + 4.123 = 18.620.$$

Two rejections and one extra edge are the whole difference between a roadmap that has found the gap and one that has not — which is the k-nearest gap of Theorem 7.4.1 in miniature.

11.5 Implementation in Rust

The `sampling` crate is introduced here and imported by Chapters 12, 13, 14, 21 and 22. Its design rule is the chapter's thesis: the planner may ask the world one question, and it may ask the configuration space three. Everything is generic over Chapter 5's `Manifold`, which is *not* redefined.

RUST **crates/sampling/src/lib.rs**

```rust
use manifold::Manifold;                // Ch. 5: type Point; dim, dist, interpolate, sample
use rand::rngs::SmallRng;

/// What this chapter asks of the world -- and nothing else. Blanket-implemented for any
/// (robot, world) pair with a Ch. 2 `Collision` impl, so planners never see parry2d.
pub trait FreeSpace<M: Manifold> {
    fn is_free(&self, q: &M::Point) -> bool;
    /// Collision-check counter: the book's cost unit, shown separately from Delta calls.
    fn checks(&self) -> u64;
}

/// Node sampler. `None` = attempt rejected -- the Gaussian, bridge and OBPRM samplers reject
/// most attempts *by design*, so callers loop on attempts and count them.
pub trait Sampler<M: Manifold> {
    fn sample(&mut self, space: &M, free: &dyn FreeSpace<M>, rng: &mut SmallRng) -> Option<M::Point>;
}
pub struct Uniform;
pub struct Gaussian { pub sigma: f64 }
pub struct Bridge { pub length: f64 }
pub struct Obprm { pub step: f64 }
pub struct MedialAxis { pub step: f64 }
pub struct Hybrid<M: Manifold> { pub parts: Vec<(Box<dyn Sampler<M>>, f64)> }
/// Replays a fixed list -- the device that makes the six-sample roadmap independent of any
/// RNG's bit stream. Ch. 12's hand-traced RRT replays its targets through it too.
```

```rust
pub struct Scripted<P> { list: Vec<P>, i: usize, require_free: bool }

/// Coordinates for kd-tree splitting; `period(axis)` is Some(2pi) on S1 factors.
/// The metric stays Manifold::dist -- the chart only decides how to split and prune.
/// `DVector`, not `SVector<f64, { Self::D }>`: stable Rust cannot size an array with an
/// associated const inside generic code (that needs the unstable `generic_const_exprs`).
pub trait Chart: Manifold {
    const D: usize;
    fn coords(&self, q: &Self::Point) -> nalgebra::DVector<f64>;
    fn period(&self, axis: usize) -> Option<f64>;
}
pub trait NearestNeighbors<M: Manifold> {
    fn insert(&mut self, idx: usize, q: M::Point);
    fn k_nearest(&self, q: &M::Point, k: usize) -> Vec<(usize, f64)>;   // sorted by dist
    fn within(&self, q: &M::Point, r: f64) -> Vec<(usize, f64)>;        // Ch. 13's
   Q_near
}

pub enum EdgeCheck { Incremental { step: f64 }, Subdivision { step: f64 } }   // figure
   7.6
pub struct LocalPath<M: Manifold> { pub waypoints: Vec<M::Point>, pub length: f64 }

/// Delta(q, q'): the local planner. `StraightLine` is deterministic and symmetric; Ch.
   21's
/// Dubins and Reeds-Shepp impls are asymmetric and the roadmap becomes directed (sec
   .7.1.1).
pub trait Steer<M: Manifold> {
    fn steer(&self, space: &M, free: &dyn FreeSpace<M>, from: &M::Point, to: &M::Point)
        -> Option<LocalPath<M>>;
    fn symmetric(&self) -> bool { true }
}
pub struct StraightLine { pub check: EdgeCheck }
```

The kd-tree is hand-rolled, not because a crate could not do flat $\mathbb{R}^d$ — `kiddo` is the candidate cross-check in the tests — but because T^2 and SE(2) need periodic axes, and the pruning bound is the one place the periodicity must be exact.

RUST **crates/sampling/src/nn.rs**

```rust
/// A kd-tree that splits on chart coordinates and measures with the manifold's metric.
/// Insertion is incremental (depth-cycled split axis): every planner in Part III grows
/// its point set one node at a time.
pub struct KdTree<M: Chart> {
    root: Option<Box<Node<M>>>,
    weights: DVector<f64>,              // per-axis scale so the bound never exceeds
   dist
    space: M,
}
struct Node<M: Chart> { idx: usize, c: DVector<f64>, axis: usize,
                        left: Option<Box<Node<M>>>, right: Option<Box<Node<M>>> }

impl<M: Chart> KdTree<M> {
    pub fn insert(&mut self, idx: usize, q: M::Point) {
        let c = self.space.coords(&q);
        let mut depth = 0;
        let mut slot = &mut self.root;
        while let Some(node) = slot {
            let a = node.axis;
```

```
            slot = if c[a] < node.c[a] { &mut node.left } else { &mut node.right };
            depth += 1;
        }
        *slot = Some(Box::new(Node { idx, c, axis: depth % M::D, left: None, right: None
    }));
    }

    /// sqrtSigma (w?.gap?)2: the distance from a query to a box, with the circular gap
    on a
    /// periodic axis -- min(|c - s|, T - |c - s|) -- so a subtree touching the seam is
    /// never pruned away from a query on the other side of it.
    fn lower_bound(&self, qc: &DVector<f64>, lo: &[f64], hi: &[f64]) -> f64 {
        let mut s = 0.0;
        for i in 0..M::D {
            let c = qc[i];
            let gap = if c < lo[i] || c > hi[i] {
                match self.space.period(i) {
                    Some(t) => {
                        let (dl, dh) = ((c - lo[i]).abs(), (c - hi[i]).abs());
                        dl.min(t - dl).min(dh).min(t - dh)
                    }
                    None => if c < lo[i] { lo[i] - c } else { c - hi[i] },
                }
            } else { 0.0 };
            s += (gap * self.weights[i]).powi(2);
        }
        s.sqrt()
    }
    // k_nearest / within: depth-first, nearer child first, far child only if
    // lower_bound(...) <= current k-th best (or r). See `kd_knn`, `kd_range`.
}
```

The planner itself is short, because every hard decision was pushed into a trait.

RUST **crates/sampling/src/prm.rs**

```
use petgraph::{graph::UnGraph, unionfind::UnionFind};

pub enum Connect { KNearest { k: usize }, Radius { r: f64 }, Sparse { k: usize, redundant:
    usize }, Lazy { k: usize } }

pub struct Roadmap<M: Manifold> {
    pub graph: UnGraph<M::Point, f64>,   // edge weight = local path length
    pub components: UnionFind<usize>,    // O(1) "how many components?" after every edge
}

pub struct Prm<M: Chart, S: Sampler<M>, N: NearestNeighbors<M>> {
    pub space: M, pub sampler: S, pub nn: N, pub steer: Box<dyn Steer<M>>, pub connect:
   Connect,
    pub roadmap: Roadmap<M>,
    pub rng: SmallRng,                   // seeded by the caller; the widget shows the
   seed
    pub delta_calls: u64,                // the coarse rain's cost
}

impl<M: Chart, S: Sampler<M>, N: NearestNeighbors<M>> Prm<M, S, N> {
    /// Choset Algorithm 6. Resumable: a larger `n` extends the roadmap (sec.7.1.1 "
```

```
    Adding").
     pub fn build(&mut self, free: &dyn FreeSpace<M>, n: usize) {
         let first_new = self.roadmap.graph.node_count();
         while self.roadmap.graph.node_count() < n {
             if let Some(q) = self.sampler.sample(&self.space, free, &mut self.rng) {
                 let i = self.roadmap.graph.add_node(q.clone()).index();
                 self.nn.insert(i, q);
             }
         }
         for i in 0..self.roadmap.graph.node_count() {
             let q = &self.roadmap.graph[i.into()];
             for (j, d) in self.candidates(i, q) {
                 // Why skip old-old pairs: they were tried when the old nodes were
    connected.
                 if i < first_new && j < first_new { continue; }
                 if self.roadmap.graph.find_edge(i.into(), j.into()).is_some() { continue;
     }
                 self.delta_calls += 1;
                 if let Some(lp) = self.steer.steer(&self.space, free, q, &self.roadmap.
    graph[j.into()]) {
                     self.roadmap.graph.add_edge(i.into(), j.into(), lp.length);
                     self.roadmap.components.union(i, j);
                 }
             }
         }
     }

     /// Choset Algorithm 7: attach both endpoints, then Ch. 6 Dijkstra on the roadmap.
     pub fn query(&mut self, free: &dyn FreeSpace<M>, q_start: &M::Point, q_goal: &M::
    Point, k: usize)
         -> Option<Vec<M::Point>>
     {
         let a = self.attach(free, q_start, k, None)?;
         // Prefer goal candidates already in the start's component: cheap, and it avoids
         // attaching the two endpoints to components that will never meet.
         let comp = self.roadmap.components.find(a);
         let b = self.attach(free, q_goal, k, Some(comp))?;
         let path = search::dijkstra(&self.roadmap.graph, a.into(), b.into())?;
         Some(std::iter::once(q_start.clone())
             .chain(path.iter().map(|&i| self.roadmap.graph[i].clone()))
             .chain(std::iter::once(q_goal.clone()))
             .collect())
     }
 }
```

The worked example is an executable, and its output is what the test asserts.

RUST **crates/sampling/examples/six_sample_prm.rs**

```
use cspace::R2;
use sampling::{analysis::{prm_failure_bound, samples_for_failure}, *};

fn main() {
    let world = BoxWorld::new([0.0, 10.0], [0.0, 10.0], &[[4.5, 5.5, 0.0, 7.0]]);   //
    the wall
    let samples = vec![[1., 2.], [3., 5.], [2., 8.], [7., 2.], [8., 6.], [6., 9.]];
    for k in [1usize, 2] {
        let mut prm = Prm::new(R2::unit_box(10.0), Scripted::new(samples.clone(), true),
```

```
                                KdTree::new(), Box::new(StraightLine { check: EdgeCheck::
    Subdivision { step: 0.25 } }),
                                Connect::KNearest { k }, SmallRng::seed_from_u64(1));
        prm.build(&world, 6);
        println!("k = {k}: edges {:?}, {} component(s)", prm.roadmap.edge_labels(), prm.
    roadmap.component_count());
        if let Some(len) = prm.roadmap.path_length(0, 3) { println!("  path s1 -> s4: {
    len:.3}"); }
    }
    for n in [500, 1000] { println!("Pr[FAILURE] <= {:.4} at n = {n}", prm_failure_bound
    (2, 0.9, 1.2, 0.1, n)); }
    println!("smallest n with bound <= 1%: {}", samples_for_failure(2, 0.9, 1.2, 0.1,
    0.01));
}
```

```
k = 1: edges ["12", "23", "45", "56"], 2 component(s)
k = 2: Delta(s1, s4) = NIL (crosses the wall at y = 2); Delta(s2, s4) = NIL (crosses at y
    in [3.125, 3.875])
k = 2: edges ["12", "23", "36", "45", "56"], 1 component(s)
  path s1 -> s4: 18.620
Pr[FAILURE] <= 0.3060 at n = 500
Pr[FAILURE] <= 0.0039 at n = 1000
smallest n with bound <= 1%: 892
```

RUST `crates/sampling/tests/prm.rs`

```rust
#[test]
fn reproduces_six_sample_roadmap() {
    let (edges1, comps1) = six_sample(1);
    assert_eq!(edges1, ["12", "23", "45", "56"]);
    assert_eq!(comps1, 2);
    let (edges2, comps2, len) = six_sample_with_path(2);
    assert_eq!(edges2, ["12", "23", "36", "45", "56"]);
    assert_eq!(comps2, 1);
    assert!((len - 18.620).abs() < 1e-3);
}

/// 2 000 points on T2, 200 queries, k = 8: the periodic kd-tree agrees with a linear
   scan,
/// including the neighbors that live across the seam.
#[test]
fn kd_tree_matches_linear_on_torus() {
    let mut rng = SmallRng::seed_from_u64(5);
    let (mut kd, mut lin) = (KdTree::new(T2), Linear::new(T2));
    for i in 0..2000 { let q = T2.sample(&mut rng); kd.insert(i, q); lin.insert(i, q); }
    for _ in 0..200 {
        let q = T2.sample(&mut rng);
        assert_eq!(kd.k_nearest(&q, 8), lin.k_nearest(&q, 8));
    }
}
```

The TypeScript port that runs the widgets, `web/lib/sampling/{prm,sampler,kdtree,steer}.ts`, is pinned to the same roadmap: the check `six-sample roadmap, k = 2` asserts the edge set $\{12, 23, 36, 45, 56\}$, one component and

the path length 18.620 to 10^{-3}, and the constants check asserts $\sigma = 0.8727$, $m = 24$, 0.306, 0.0039 and 892.

11.6 Putting it together: one planner, two robots

The integration lab is the point of the generic design. The *same* `Prm` — the same file, the same code path, no planner line changed — runs on Rusty as a disc in the Apartment, where the space is $\mathbb{R}^2$ with Euclidean distance, and on Reach on the Workbench, where the space is T^2 and the straight line between two configurations may pass through the seam. The only two things that differ are the `Manifold` and the `FreeSpace`, and both come from earlier chapters. Roadmap Rain's two tabs are that lab; the checks `PRM path is collision-free by an independent dense check` and `the same Prm on T² (Reach, Workbench)` re-verify every returned path at five millimetres, independently of the planner's own checks, on both robots.

Three lessons to take from the lab before the chapter closes.

The cost unit is the collision check. Build a six-hundred-node roadmap with $k = 6$ and watch the two counters: a few thousand calls to Δ, and an order of magnitude more collision checks, because every edge is a row of them. When Chapter 12 claims that a tree planner is cheaper for one query, or Chapter 13 that lazy collision checking saves most of FMT*'s budget, those claims are in this unit, and they are only meaningful because this chapter counted it separately.

The roadmap is the expensive thing; the query is cheap. Once built, a query costs two attachments and a Dijkstra — milliseconds. That is the economics of a *multi-query* planner: pay once, ask often. It is the right economics for a robot that lives in one room and is asked for a thousand paths.

It is the wrong economics for a robot that is asked for one path, now. Most robots have one query — from where they are to where they want to be — and then the world changes. Move one obstacle on the Workbench and the roadmap is stale: every edge through the moved object's new C-obstacle is wrong, and nothing in the roadmap knows which. A planner that builds only the part of $\mathcal{Q}_{\text{free}}$ the current query needs, and stops the instant start and goal connect, would spend its budget on the right problem. That planner is a *tree*, and it is Chapter 12.

11.7 Exercises

1 FOUNDATION •• **The k-nearest gap in Theorem 7.4.1**

Step 2 of the ball-tiling proof assumes every pair of samples within ρ of each other is tried by Δ. Exhibit a configuration of samples, in the plane, where every ball $B_{\rho/2}(x_i)$ of the tiling contains a sample and yet the $k = 1$ roadmap is disconnected. Then state the smallest change to the connection rule that restores the proof, and say what it costs

in calls to Δ per node as n grows.

HINTS

<>Put three samples in one ball, close together, and one sample in the next ball along the path. Whose nearest neighbor is the lonely one?</> <>A radius rule — connect every pair within distance ρ — restores Step 2 verbatim. How many samples fall within ρ of a given one when n is large?</> <>Linear in n. Chapter 13's r_n = γ (log n / n)^(1/d) is the radius that shrinks just fast enough to make that count logarithmic while keeping the balls of this proof hit forever.</>

2 FOUNDATION •• **Figure 7.19 by measure**

Two squares of side W are joined by a corridor of width w and length W. Compute $\mu(\text{reach}(x))$ for a point x at mid-corridor to leading order in w/W, and for a point in the middle of a room. Deduce the ϵ of Definition 7.4.2, then argue — by taking S to be one room plus half the corridor — that α and β are also of order w/W. Where does the straight-line local planner enter?

WORK IT OUT

For w / W = 0.1, what is ε to one significant figure (as a fraction of μ(Q_free) $\approx 2W^2$)?

Answer. 0.05 From mid-corridor, straight lines reach a wedge of each room whose area is of order wW, so $\mu(\text{reach}(x)) \approx 2\,c\,wW$ for a constant c near $\frac{1}{2}$, against $\mu(\mathcal{Q}_{\text{free}}) \approx 2W^2$: $\epsilon \approx c\,w/W$. Choset writes the estimate as $\epsilon \approx w/W$ with the constant dropped; the point is the dependence, which the narrow-passage widget's uniform curve follows.

3 CONCEPTUAL • **Predict the narrow passage, then verify**

PREDICT FIRST

In Narrow Passage at n = 150 the uniform sampler connects every one of 24 seeds at w = 0.50 m (w/W = 0.125). Before touching the slider: what fraction of seeds connect at w = 0.15 m (w/W = 0.0375)?

- Still nearly all — the corridor is only three times narrower
  The probability that one of 150 samples lands in the corridor scales with its area, which is three times smaller, and the roadmap needs several samples in it (or lucky alignment) to chain through. Theorem 7.4.2 says the required n scales like $1/(\varepsilon\alpha) \propto (W/w)^2$, a factor of nine.
- ✓ Roughly half
  Right. The measured rate at w = 0.15 m is about 40 percent of seeds. The exponent of Theorem 7.4.2 scales as $1/(\varepsilon\alpha) \propto (W/w)^2$: nine times the samples for the same confidence, and at fixed n the success probability falls off accordingly.
- Essentially none
  Too pessimistic. The corridor is straight, so two samples at its mouths that happen to see each other through it suffice; with 150 samples in 48 m^2 that still happens on a good fraction of seeds.

Now enable the Gaussian sampler. Its curve stays at the top at every width, because Derivation 4 puts its mass within $O(\sigma)$ of *every* boundary — the corridor's two walls included — so the corridor is sampled on every seed. Read what it pays: the attempts counter runs several times the node count, and the cloud shows samples hugging the rooms' outer walls where they do nothing for the query. Then enable the bridge test: at w = 0.25 m it recovers most of uniform's loss, and at w = 0.15 m it does not, because the bridge of length one metre now has both piers in collision in only a thin set of directions.

4 CONCEPTUAL •• **Interleaved components on the torus**

In Roadmap Rain's torus tab, find a seed and a sample count n at which two roadmap components interleave on the unwrapped square — their nodes alternate along a strip — separated only by the amber band of the block's C-obstacle. Which connection strategy would join them soonest: more neighbors k, connection sampling around low-degree nodes, or the bridge test? Explain using where each puts its next sample.

5 PRACTICAL •• **Lazy PRM**

Implement `Connect::Lazy` (Bohlin and Kavraki, Choset ref. 53): build the roadmap with no edge checks at all, search, then check the nodes and then the edges along the

candidate path coarse-to-fine — the midpoints of every edge first, then the quarter points — deleting the first blocked edge and searching again until a path survives. Record, per edge, the resolution at which it was last checked so that a later path does not repeat work. Report collision checks against `KNearest` on the Workbench over 50 seeds, at the same n and k, and explain the ratio.

6 PRACTICAL • • • **Halton points and dispersion**

Add a `Halton` sampler (bases 2 and 3 mapped onto the torus) and an estimator of dispersion $\delta(P, \rho)$ by probing with a fine grid. Show that two builds at the same n produce identical roadmaps, that dispersion falls like $n^{-1/2}$ on T^2, and that the planner is now resolution complete rather than probabilistically complete. Then build the adversarial layout Choset promises exists: a free space where the Halton roadmap at some n misses a passage that uniform sampling finds on most seeds.

11.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

Chapter 7, §7.1 and §7.4, is this chapter's source: Algorithms 6, 7 and 15, the sampler zoo of §7.1.3, and the three completeness analyses with the definitions quoted here. The bridge test (ref. 193), the Gaussian sampler (ref. 59), OBPRM (ref. 18), MAPRM (ref. 427), Lazy PRM (refs. 53–54) and the kd-tree (ref. 124) are cited through its bibliography. `link`

Kavraki, L. E., Švestka, P., Latombe, J.-C., and Overmars, M. H. (1996). Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces. *IEEE Transactions on Robotics and Automation 12(4), 566–580*.

The planner. Two phases, coarse and fine sampling, the k-nearest connection rule and the first experiments on many-degree-of-freedom arms. `doi:10.1109/70.508439`

Hsu, D., Latombe, J.-C., and Motwani, R. (1999). Path Planning in Expansive Configuration Spaces. *International Journal of Computational Geometry and Applications 9(4–5), 495–512*.

(ε, α, β)-expansiveness and Theorem 7.4.2; the paper's linking sequences are the proof device Derivation 2 follows. `doi:10.1142/S0218195999000285`

Karaman, S. and Frazzoli, E. (2011). Sampling-based Algorithms for Optimal Motion Planning. *International Journal of Robotics Research 30(7), 846–894*.

Where the k-nearest gap of Theorem 7.4.1 is closed: the r_n and k_n connection rules of PRM*, which Chapter 13 derives. `doi:10.1177/0278364911406761`

de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. (2008). Computational Geometry: Algorithms and Applications. *Springer, 3rd edition*.
Chapter 5 is the kd-tree Choset recommends: O(n log n) to build, O(n^{1−1/d} + m) per range query. The periodic pruning bound is this book's addition. doi:10.1007/978-3-540-77974-2

LaValle, S. M. (2006). Planning Algorithms. *Cambridge University Press*.
Chapter 5 treats sampling-based planning with the fullest discussion of metrics on SE(2) and SE(3), dispersion and discrepancy, and Sukharev grids. link

Şucan, I. A., Moll, M., and Kavraki, L. E. (2012). The Open Motion Planning Library. *IEEE Robotics and Automation Magazine 19(4), 72–82*.
The reference implementations of PRM and of every sampler in this chapter, with the state-space abstraction this book's Manifold trait mirrors. doi:10.1109/MRA.2012.2205651

CHAPTER 12

Tree Planners: EST, RRT, and Friends

SAMPLING-BASED PLANNING · Intermediate · 65 min

> *The introduced sampling methods are fundamentally conditional: the generation of a new configuration depends on the initial and goal configuration and any previously generated configurations.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), §7.2*

Chapter 11 spent its budget learning *all* of $\mathcal{Q}_{\text{free}}$ so that any query is cheap. Most robots do not have many queries. They have one, now — from where they are to where they want to be — and then the world changes and the roadmap is stale.

This chapter inverts the economics. Grow a tree from the configuration you are in toward the one you want, sample *conditionally* on what the tree already knows, and stop the instant the two meet. Two families do it with opposite instincts. The Expansive-Spaces Tree picks a lonely node and pushes outward from it. The Rapidly-exploring Random Tree picks a random target and pulls the nearest node toward it. The surprise is a single geometric fact: **RRT's pull is a Voronoi bias.** The probability that a node is chosen to grow equals the volume of its Voronoi region, so the frontier grows fastest exactly where the tree has not been — and that one sentence explains why an algorithm you can write in thirty lines covers configuration spaces nobody can draw.

Along the way the chapter adds `extend` to Chapter 11's `Steer`, imports its samplers and kd-tree unchanged, and shows through Choset's roadmap of trees that PRM, RRT and EST are three settings of one dial. The sister book's Chapter 20 has the RRT preview; this is the reference it points to.

AFTER THIS CHAPTER YOU CAN

- Invert Chapter 11's economics: grow a tree from the configuration you are in toward the one you want, sample conditionally on what the tree already knows, and stop the instant the two meet
- State and prove the Voronoi-bias lemma — RRT extends a node with probability equal to the volume of its Voronoi cell — and use it to explain why a thirty-line algorithm covers spaces nobody can visualize
- Trace three RRT extensions by hand, including the one that is Trapped, and read the three-valued extend contract that RRT-Connect's greedy loop depends on
- Explain EST's push (π_T $\propto 1/$w_T), SBL's lazy edges, and why goal bias p = 1 is a potential-field planner with Chapter 7's local minima
- Show, with Choset's SRT, that PRM, RRT and EST are three settings of one dial — and name the modern descendant of EST's grid, KPIECE
- Write sampling::tree — one Tree<M>, Est, Rrt, RrtConnect, Sbl, Srt — on top of Chapter 11's Sampler, KdTree and Steer with nothing re-introduced, and benchmark five planners on three worlds

ASSUMED

- Everything in Chapter 11 is imported: Sampler and Uniform, NearestNeighbors and KdTree, Steer and StraightLine, FreeSpace, the (ε, α, β) vocabulary and Theorem 7.4.1's tiling
- Manifold::interpolate (Chapter 5) — what extend moves along
- Parent-following path extraction and the lattice A* lane (Chapter 6); why pure goal-seeking is a potential planner (Chapter 7)
- The three worlds; Hitch's kinematics exist but are deliberately not used by this chapter's steer (Chapter 2)

12.1 The problem: the roadmap is stale

Build Chapter 11's roadmap for Reach on the Workbench. Then slide the block half a metre toward the base.

INTERACTIVE · `w12.0`

Stale roadmap, fresh tree

A roadmap answers many queries fast and one changed obstacle for nothing: a tree spends its budget on the one query it was asked, from scratch, and is done when start

and goal meet.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch12-tree-planners#w12.0

Every edge of the roadmap was collision-checked against a world that no longer exists. The red ones now pass through the block's new C-obstacle, and the purple path the roadmap would still confidently return runs straight through them. The roadmap does not know which edges died. To find out, it must re-check all of them — which costs about what it cost to build — and a roadmap that must be rebuilt after every change has lost the one thing it was for.

The tree on the right never learned the old world. Rooted at the start, it grows toward the goal on the world as it is now, spends its collision checks only on the part of Q_{free} this query passes through, and stops the moment a node reaches the goal. For one query it is cheaper; for one query in a world that moves it is the only sensible thing to do.

That is the single-query planner, and the chapter is about how a tree decides where to grow.

12.2 Building intuition: push from the lonely node, or pull toward empty space

Two instincts. **EST pushes**: pick a node that has few neighbors — it is on the frontier — and sample a new configuration near it. **RRT pulls**: pick a random *target* anywhere in Q, find the tree node nearest to it, and step from that node toward the target. Both ask the tree where it has not been; they answer in different geometry.

INTERACTIVE · w12.1

Tree Grower

RRT does not explore randomly: it explores where the Voronoi cells are large, a deterministic function of the tree — and the random target only decides which large cell goes next.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch12-tree-planners#w12.1

Leave the Voronoi overlay on and watch a dozen iterations. The shaded cells are the Voronoi regions of the tree's nodes — every point of the square is colored by the node nearest to it, and the shade is the cell's area. The random target lands somewhere; the node that owns that somewhere is the one pulled. Big cells get pulled often, small ones rarely, and a cell is big exactly when its node faces unexplored space. Nothing random decides *that*; the tree does. The randomness

only decides which large cell goes next.

Now the misconception the widget exists to kill: *"RRT explores randomly."* It does not. It explores where the Voronoi cells are large, which is a deterministic function of the current tree, and the proof is one line in the Foundation section. Three more things to try. Slide η down and the tree becomes dense and short-edged, every step small, many iterations to cross the square. Slide it up and *Trapped* flashes at the amber band — a long step from a node near the C-obstacle lands inside it. Set the goal bias to one and the planner stops being a tree planner at all: every iteration pulls the goal-nearest node toward the goal, straight into the first wall, and the "stuck" badge lights. That is Chapter 7's gradient descent wearing a tree's clothes.

> **THE ONE SENTENCE VERSION**
>
> A tree planner samples conditionally on the tree it has already built; RRT's rule — pull the nearest node toward a uniform random target — chooses node v with probability equal to the measure of its Voronoi cell, so the tree reaches into unexplored space by construction, not by luck.

12.3 The mathematics

$T = (V, E)$, T_{init}, T_{goal}, $\text{par}(v)$	a tree rooted at q_start or q_goal; the parent map — every node has a path to its root by construction
q_{rand}, q_{near}, q_{new}	the sampled target; the tree node nearest to it; the node produced by extend Choset Alg. 11
η	step_size: how far extend moves from q_near toward q_rand
p	goal bias: the probability that q_rand := q_goal Choset §7.2.2; experimental 0.05
$\pi_T(q)$, $w_T(q)$	EST's selection distribution over V; the density weight (nodes in a neighborhood of q) Choset §7.2.1
$\text{Vor}_T(v)$	the Voronoi region of v with respect to V under dist
$G_T = (\mathcal{V}_T, \mathcal{E}_T)$, ℓ	SRT's roadmap of trees; the number of merge attempts Choset Alg. 13–14

12.3.1 Definitions

DEFINITIONS — SINGLE-QUERY PLANNER, CONDITIONAL SAMPLING, THE EXTEND CONTRACT

A **single-query planner** solves one $(q_{\text{start}}, q_{\text{goal}})$ pair, explores only the part of Q_{free} relevant to it, and keeps every node reachable from a root by construction.
Its sampling is **conditional** (Choset §7.2): the distribution of the next configuration depends on the tree or trees built so far and on the query — unlike PRM's unconditional rain.
The **extend status** is three-valued. `Reached`: $q_{new} = q_{rand}$, the target was within η and the local path to it is free. `Advanced`: q_{new} was added η short of q_{rand}. `Trapped`: the node or the local path to it is blocked, and nothing was added. The third value is the one RRT-Connect's greedy loop needs.

DEFINITIONS — VORONOI BIAS, LAZY COLLISION CHECKING, ROADMAP OF TREES

Under uniform q_{rand}, the **Voronoi bias** is $\Pr[q_{near} = v] = \mu(\text{Vor}_T(v) \cap Q)/\mu(Q)$.
Lazy collision checking (SBL): nodes are checked on creation; an edge is checked once, and only when it lies on a candidate path that would join the two trees.
A **roadmap of trees** (SRT) is a PRM roadmap with every tree contracted to a vertex: $(T_i, T_j) \in \mathcal{E}_T$ iff some $q_i \in T_i$ and $q_j \in T_j$ were joined by a local path.

12.3.2 The Voronoi-bias lemma

DERIVATION **RRT's pull is a Voronoi bias**

$\Pr[q_{near} = v] = \mu(\text{Vor}_T(v) \cap Q)/\mu(Q)$

Step 1 — q_{near} is the Voronoi owner. By definition, q_{near} is the node of V nearest to q_{rand} under `dist`. The set of points whose nearest node is v is, by definition, v's Voronoi region $\text{Vor}_T(v)$. So $q_{near} = v$ iff $q_{rand} \in \text{Vor}_T(v)$, up to ties of measure zero.
Step 2 — uniform sampling turns volume into probability. With q_{rand} uniform on Q, $\Pr[q_{rand} \in A] = \mu(A \cap Q)/\mu(Q)$ for any measurable A. Substitute $A = \text{Vor}_T(v)$.
Step 3 — interior cells shrink, frontier cells do not. As a region of Q fills with nodes, the cells of the nodes inside it shrink to the inter-node spacing

— each is bounded by its neighbors on all sides. A node on the frontier has no neighbors on the far side, so its cell is bounded there only by the edge of Q and stays large.

Step 4 — the fraction of effort spent inside decays. Summing Step 2 over interior nodes gives the probability that an iteration is spent densifying what is already explored; by Step 3 that sum shrinks as the tree grows, and the complement — iterations spent on frontier nodes — does not. Expected growth points outward. ■

Metric sensitivity. The cells are Voronoi cells *under* `dist`. On SE(2) with a badly scaled exchange rate between metres and radians, the cells stretch along the cheap axis and the tree grows the wrong way — Choset's §7.2.2 caution and Cheng and LaValle's [C ref. 103]. Exercise 1 asks what RRT does to Hitch's heading as the rotation weight goes to zero. The check `Voronoi bias` freezes a 40-node tree, samples twenty thousand targets, and finds the selection frequencies within one percent of the Voronoi shares.

12.3.3 Step size and metric: two knobs with no theory behind their defaults

Two parameters sit in Algorithm 11 with nothing in the proofs to set them, and Choset's §7.2.2 is honest about both.

Step size η. Small η makes every extension short and almost always free: the tree is dense, the frontier advances slowly, and iterations are spent densifying. Large η crosses the space in a few steps and is `Trapped` whenever a node near an obstacle owns a large cell on the far side of it — which, by the Voronoi lemma, is often. The widget's iterations-to-solution curve has a minimum in between, and its location depends on the clearance of the problem, not on anything the algorithm knows. A common practical rule sets η near the local planner's check `step` times a small integer, so that an extension is a few collision checks; the book's defaults are 0.6 m for Rusty and 0.4 rad for Reach, chosen by looking at the widget.

The metric. Nearest-neighbor under a different `dist` is a different planner, because the cells are different. On T^2 the circular metric is forced by the topology. On SE(2) there is no canonical choice, and the exchange rate between a metre and a radian decides whether the tree explores positions or headings first. Chapter 5 introduced the weight as a constructor argument for exactly this reason; Exercise 1 follows it to the limit. Cheng and LaValle's observation [C ref. 103] is the practical one: a metric that approximates the *cost-to-go under the robot's own motion* makes the Voronoi cells meaningful, and a straight-line metric for a car does not — which is the Hitch row of the lab.

Goal bias p is the third knob, and it does have a theory, in Derivation 4: any $p < 1$ keeps the planner complete, and p near 0.05 is what the experiments favor.

12.3.4 Single-query PRM and Lazy PRM, for the record

Choset points out that PRM itself can be run single-query: insert q_{start} and q_{goal} as the first two nodes, grow, and stop when they share a component — which is exactly how Chapter 11's Roadmap Rain works. Bohlin and Kavraki's Lazy PRM goes further and creates a roadmap whose nodes and edges have *not* been checked at all, searches it, and checks only along candidate paths, coarse-to-fine, recording per edge the resolution reached so that a later path does not repeat work. It was shown experimentally to find a free path well before the roadmap was fully checked. The tree planners of this chapter are the other answer to the same question, and SBL borrows Lazy PRM's idea wholesale.

12.3.5 Probabilistic completeness, honestly

Choset's text says RRT "has been shown to be probabilistically complete under certain assumptions [271]" and leaves it there. The 2005 book contains no argument, and the chapter should say why the argument is harder than it looks.

THEOREM — PROBABILISTIC COMPLETENESS OF RRT

Suppose a query admits a path of length L and clearance $\rho > 0$, and $\eta \le \rho$. Then the probability that the tree has no node in $B_{\rho/2}(q_{\text{goal}})$ after n iterations decays exponentially in n, with constants depending on ρ, L and $\mu(\mathcal{Q}_{\text{free}})$.

DERIVATION **Why the RRT completeness proof is delicate**

$\Pr[\text{no node in } B_{\rho/2}(q_{\text{goal}}) \text{ after } n] \le Ce^{-cn}$

Step 1 — tile the path. Cover the known path with $m = \lceil 2L/\rho \rceil$ balls of radius $\rho/2$, as in Chapter 11's Theorem 7.4.1. Consecutive balls lie inside a common free ball of radius ρ.

Step 2 — the delicate step. Suppose the tree has a node v in ball i. One wants to say: a sample in ball $i+1$ makes v extend into ball $i+1$. But the nearest node to that sample need *not* be v — it may be some other node, elsewhere in the tree, that happens to be closer. The honest statement is weaker: a sample in a suitable sub-ball of ball $i+1$ makes *some* node's extension land in ball $i+1$. One shows that whichever node is nearest, its extension by $\eta \le \rho$ toward a target in that sub-ball ends inside the free ball of Step 1, hence is `Advanced` or `Reached`, hence adds a node at least as deep along the path.

Step 3 — a constant per iteration. The sub-ball has positive measure, so the event of Step 2 has probability at least a constant $c > 0$ per iteration, independent of the tree.

Step 4 — geometric domination. The number of iterations to advance from ball i to ball $i+1$ is dominated by a geometric random variable with success probability c.
Step 5 — Chernoff. The time to traverse all m balls is a sum of m such variables; a Chernoff bound gives the exponential tail. ■
The gap in the sketch is Step 2, and it is real: making "some node's extension lands in the next ball" rigorous for every tree shape is what the 2005 text does not do. The clean proof for geometric RRT — and for the kinodynamic version with forward propagation, where it is harder still — is Kleinbort, Solovey, Littlefield, Bekris and Halperin (2019), and that is the paper the chapter cites for the theorem rather than the textbook sentence.

12.3.6 EST covers an expansive space

THEOREM — EST IS PROBABILISTICALLY COMPLETE ON EXPANSIVE SPACES (HSU, LATOMBE AND MOTWANI)

With $\pi_T \propto 1/w_T$ in an $(\epsilon, \alpha, \beta)$-expansive space, the reach set of the tree's nodes covers at least three quarters of its component after $O(\beta^{-1} \ln 4)$ lookout hits, each with per-attempt probability bounded below in terms of $\alpha\epsilon$. Hence EST is probabilistically complete.

DERIVATION **EST's push is a linking sequence**

```
\mu(\mathrm{reach}(V)) \ge \tfrac34\,\mu(\Qfree) \text{ after } O(\beta^{-1}\ln 4) \text{ lookout hits}
```

Step 1 — a low-w_T node is on the frontier. $w_T(q)$ counts nodes in a neighborhood of q. A node with few neighbors has empty space around it in the tree's own geometry; $\pi_T \propto 1/w_T$ makes it the likely one to push from.
Step 2 — a fraction of the reach set is lookout. By expansiveness, $\mu(\text{lookout}_\beta(S)) \ge \alpha\,\mu(S)$ for the current reach set $S = \text{reach}(V)$, and by ϵ-goodness $\mu(S) \ge \epsilon\,\mu(\mathcal{Q}_{\text{free}})$. A sample near a frontier node lands in the lookout with probability bounded below by a constant times $\alpha\epsilon$.
Step 3 — a lookout hit grows the reach set. A new node in $\text{lookout}_\beta(S)$ sees at least a β fraction of $\mathcal{Q}_{\text{free}} \setminus S$, so $\text{reach}(V \cup \{q_{new}\})$ covers at least a β fraction more of what was uncovered. This is one step of Chapter 11's linking sequence.

Step 4 — Lemma 7.4.7 bounds the step count. After $t \geq \beta^{-1} \ln 4$ such steps the uncovered measure is at most $(1-\beta)^t \leq 1/4$ of the component. ■
Why uniform π_T fails. With π_T uniform over V, the dense interior — where most nodes are — is pushed most often, and the frontier starves. The grid-cell π_T of §7.2.3 (choose a non-empty cell uniformly, then a node in it) is an $O(1)$ substitute for the $O(n)$ neighbor count, with the same bias toward sparse regions; it is the device SBL uses and the one KPIECE generalizes.

12.3.7 Goal bias: why $p = 1$ is a potential-field planner

DERIVATION **Pure goal bias is gradient descent with a tree's bookkeeping**

p = 1 \Rightarrow \text{greedy descent of } \mathrm{dist}(\cdot, \qgoal);\; p < 1 \Rightarrow \text{complete}

Step 1 — at $p = 1$ the goal-nearest node is always extended. With $q_{rand} \equiv q_{\text{goal}}$, q_{near} is the node minimizing $\text{dist}(\cdot, q_{\text{goal}})$ — the same node every iteration until an extension succeeds, when its child is nearer still. The sequence of new nodes is greedy descent of $\text{dist}(\cdot, q_{\text{goal}})$ over the frontier, with step η.
Step 2 — once Trapped, forever Trapped. When the goal-nearest node's extension toward the goal is blocked, no node is added, so the goal-nearest node does not change, so the next iteration repeats the same blocked extension. The planner is wedged at a local minimum of $\text{dist}(\cdot, q_{\text{goal}})$ restricted to the tree — exactly Chapter 7's failure mode, and the check `goal bias p = 1 traps the tree` measures a trapped streak of several hundred iterations after seven nodes.
Step 3 — any $p < 1$ restores completeness. With probability $1 - p$ per iteration the target is uniform, and the Voronoi-bias argument of the completeness theorem applies to those iterations with its constants scaled by $1 - p$. Small p — Choset's experimental 0.05 — buys faster convergence when the straight line toward the goal happens to be free, and costs almost nothing when it is not. ■

12.3.8 SRT specializes to PRM, RRT and EST

DERIVATION **One dial, three planners (Choset §7.3)**

(1\text{ node}, 1\text{ pair}, 0\text{ merges}) = \text{Alg. 6};\quad (0\text{ roadmap trees}, 0\text{ pairs}) = \text{Alg. 13}

Step 1 — a one-node tree's representative is the node. SRT connects trees by their representatives — roots, in this implementation. With one node per tree, "the k nearest trees" are "the k nearest configurations": PRM's N_q.
Step 2 — one close pair is PRM's edge test. Algorithm 14 tries `close_pairs` configuration pairs between two neighboring trees with Δ. With one pair and one-node trees, the pair *is* the two nodes, and the test is Algorithm 6's line 8. With zero merge iterations, nothing else happens. The check `SRT with (1 node, 1 close pair, 0 merges)` replays Chapter 11's six samples through this path and recovers the edge set $\{12, 23, 36, 45, 56\}$.
Step 3 — no roadmap trees is a bidirectional tree planner. With no sampled trees, only T_{init} and T_{goal} exist, and the connection phase between them with zero close pairs is Algorithm 13's merge — a bidirectional RRT (or EST, with the other grower). The check `SRT with no roadmap trees and no close pairs is a bidirectional RRT` finds two trees and one merge edge. ■
Why SRT parallelizes. Trees are grown independently — embarrassingly parallel — and only the connection phase couples them; Akinc et al. and Bekris et al. [C refs. 14, 43] distribute the trees and gather the edges.

12.4 The algorithms

ALGORITHM `BUILD RRT / EXTEND RRT (Choset Algorithms 10-11)` COST

`per iteration: one nearest-neighbor query (O(log n) expected with the kd-tree) plus` $\lceil \eta/\text{step} \rceil$ `collision checks`

in a root q_0, a sampler for q_rand, step size η, a local planner Δ, an iteration budget n
out a tree T rooted at q_0, or a path once a node reaches q_goal

1: $T \leftarrow$ the tree with the single node q_0
2: **for** $i = 1 \ldots n$ **do**
3: $\quad q_{rand} \leftarrow$ RANDOM_CONFIG — with probability p, q_{goal} instead
4: $\quad$ EXTEND(T, q_{rand})
5: **return** T

```
6:  procedure EXTEND(T, q):
7:      q_near ← NEAREST_NEIGHBOR(q, T)
8:      q_new ← the point η along the geodesic from q_near toward q — or q itself if dist(q_near, q) ≤ η
9:      if Δ(q_near, q_new) ≠ NIL then add q_new to T with parent q_near; return Reached if q_new = q else Advanced
10:     return Trapped
```

ALGORITHM CONNECT RRT / MERGE RRT (Choset Algorithms 12-13), and RRT-Connect COST per attempt: one or two EXTEND calls (Extend) or up to ⌈dist/η⌉ (Connect)

in two trees T_init, T_goal; a merge budget ℓ; a greediness setting per side

out a path from q_start to q_goal when the trees meet, or failure after ℓ attempts

```
1:  procedure CONNECT(T, q): repeat s ← EXTEND(T, q) until s ≠ Advanced; return s
2:  procedure MERGE(T_1, T_2, ℓ):
3:      for i = 1 . . . ℓ do
4:          q_rand ← RANDOM_CONFIG
5:          if EXTEND(T_1, q_rand) ≠ Trapped then
6:              if GROW(T_2, q_new) = Reached then return the path through q_new
7:          SWAP(T_1, T_2)
8:      return failure
9:  where GROW is EXTEND in Algorithm 13 as written, CONNECT in RRT-Connect [Kuffner and LaValle] — and both sides CONNECT in the greediest setting
```

Line 7 matters more than it looks: the trees swap only after a successful first extension, so the tree that just grew hands the initiative to the other, and neither starves (Exercise 2).

INTERACTIVE · w12.2

Bidirectional Connect

Greedier is not always faster: Connect/Connect wins in open rooms and loses in clutter, where long greedy runs end Trapped and every one of them was paid for in collision checks.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch12-tree-planners#w12.2

Four planners, one seed, in lockstep. Choset's Algorithm 13 extends both trees by η; RRT-Connect extends one and lets the other *Connect* — keep stepping toward the new node until it is reached or trapped; the greediest setting lets both Connect. The misconception to watch die is *"greedier is always faster."* In the open rooms a Connect run crosses metres in one attempt and the greedy planners merge early. Near the doorways a greedy run is a long sequence of steps that ends `Trapped` against the wall, and every step of it was paid for in collision checks — the table's third column. Which setting wins depends on the clutter, and the chapter has no theory for the crossover (Exercise 4 measures it).

The fourth lane is **SBL** — Sá nchez and Latombe's Single-query, Bi-directional, Lazy planner: a bidirectional EST with the grid-cell π_T of §7.2.3, whose edges are *not checked when created*. Only when a new node lands within reach of the other tree does SBL check the candidate path joining the two roots — bridge first, then the longest unchecked edges — and if an edge fails it cuts the tree there and hands the severed piece to the other side, so no sample is wasted. The dashed edges on the canvas have never been collision-checked. The check counter shows what that saves: the check `SBL: lazy edges` records more than a thousand edges created and a few dozen ever verified.

ALGORITHM BUILD EST / EXTEND EST (Choset Algorithms 8-9) COST per iteration: one weighted draw from π_T (O(1) with grid cells, O(n) naive), one sample in a ball, one call to Δ

in a root q_0, a neighborhood radius, a density weight w_T, a local planner Δ
out a tree T

1: $T \leftarrow$ the tree with the single node q_0
2: **for** $i = 1 \ldots n$ **do**
3: $q_{rand} \leftarrow$ a node of T drawn with probability $\pi_T(q) \propto 1/w_T(q)$
4: $q_{new} \leftarrow$ a uniform free configuration in the neighborhood ball of q_{rand}
5: **if** $\Delta(q_{rand}, q_{new}) \neq$ NIL **then** add q_{new} to T with parent q_{rand} and update w_T
6: **return** T

ALGORITHM CONNECT SRT (Choset Algorithm 14) COST per tree: k + r candidate edges × (close_pairs calls to Δ + at most merge_iters extension pairs)

in a set of trees V_T with representatives; k nearest and r random neighbor trees; close_pairs; merge_iters
out the roadmap of trees G_T = (V_T, E_T)

$\mathcal{E}_T \leftarrow \emptyset$
for all $T_i \in \mathcal{V}_T$ **do**
  $N_{T_i} \leftarrow$ the k nearest trees by representative distance, plus r random others
  for all $T_j \in N_{T_i}$ not already in T_i's component **do**
    for $c = 1 \ldots$ close_pairs: pick $q_i \in T_i$ at random, $q_j \leftarrow$ nearest in T_j; **if** $\Delta(q_i, q_j) \neq$ NIL **then** add (T_i, T_j) and **break**
    if not joined and merge_iters > 0 **then** MERGE(T_i, T_j, merge_iters) and add (T_i, T_j) on success
return G_T

INTERACTIVE · w12.3

Roadmap-of-Trees Dial

PRM and RRT are not different algorithms but two corners of one box: SRT with one node per tree is PRM, SRT with no roadmap trees is a bidirectional RRT, and everything between is a roadmap whose nodes are trees.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch12-tree-planners#w12.3

Turn the dial. At one node per tree, one close pair and no merges, the picture is Chapter 11's roadmap: dots joined by straight edges. At zero roadmap trees it is this chapter's bidirectional RRT. Between, Choset's figure 7.16 — blobs of tree joined by solid close-pair edges and dashed merges. The misconception, *"PRM and RRT are different algorithms,"* does not survive the slider.

12.4.1 KPIECE: EST's grid, generalized

The grid-cell π_T of SBL bins tree nodes by their chart coordinates and picks a cell before a node. Şucan and Kavraki's **KPIECE** (2008) makes the bin a *projection* — any low-dimensional function of the configuration, joint angles for Reach, position for Hitch — and chooses the cell by an *importance* that rewards cells recently discovered, productive when expanded, rarely selected, isolated from neighbors and sparsely covered:

$$\mathrm{I}(c) \;=\; \frac{\mathrm{score}(c)}{\mathrm{selections}(c)\,\big(1+\mathrm{neighbors}(c)\big)\,\mathrm{coverage}(c)},$$

with a cell's score starting at 1 + log(iteration created) and decaying by a factor after every expansion — more after a failed one. Exterior cells (missing a neighbor) are chosen with high probability; the expansion itself is EST's push, a sample in the η-ball of a motion in the cell joined by Δ. It is the planner OMPL reaches for by default, it is labelled *modern* here because it postdates Choset, and

the implementation is single-level where the paper uses several.

12.4.2 The three extensions, by hand

The example the implementation is tested against. A point robot in $\mathcal{Q} = [0, 10]^2$ with the obstacle $\mathcal{QO} = [3, 4] \times [0, 3]$, root $q_0 = (1, 1)$, $\eta = 1$, Euclidean `dist`, straight-line `extend` with subdivision at 0.25. The targets are fixed and replayed through Chapter 11's `Scripted` sampler — as *targets*, not nodes, because Choset's RANDOM_CONFIG draws from all of $\mathcal{Q}$ and the third target sits on the obstacle's edge on purpose: $q_{rand}^{(1)} = (5, 4)$, $q_{rand}^{(2)} = (2, 6)$, $q_{rand}^{(3)} = (4, 1.5)$.

Extension 1. The only node is q_0, at distance 5 from $(5, 4)$. The direction is $(4, 3)/5 = (0.8, 0.6)$, so $q_{new} = (1.8, 1.6)$. The edge never enters $x \geq 3$: free. `Advanced`.

Extension 2. Distances to $(2, 6)$: $\sqrt{26} = 5.099$ from q_0, $\sqrt{0.04 + 19.36} = 4.4045$ from $(1.8, 1.6)$. So $q_{near} = (1.8, 1.6)$; the direction is $(0.2, 4.4)/4.4045$ and $q_{new} = (1.8454, 2.5990)$. `Advanced`.

Extension 3. Distances to $(4, 1.5)$: 3.0414 from q_0, 2.2023 from $(1.8, 1.6)$, 2.4187 from $(1.8454, 2.5990)$. The nearest is $(1.8, 1.6)$ *again* — it becomes the branch point. Direction $(2.2, -0.1)/2.2023$, $q_{new} = (2.7990, 1.5546)$, with $x < 3$, so node and edge are free. `Advanced`.

Final tree: four nodes; parents $(1.8, 1.6) \mapsto q_0$, $(1.8454, 2.5990) \mapsto (1.8, 1.6)$, $(2.7990, 1.5546) \mapsto (1.8, 1.6)$; total edge length 3.000 — three steps of exactly η.

The same third extension with $\eta = 1.5$. From $(1.8, 1.6)$ a step of 1.5 toward $(4, 1.5)$ lands at $(3.2985, 1.5319)$, inside $\mathcal{QO}$. The local planner checks the far endpoint first and rejects it: `Trapped`, tree unchanged at three nodes. Nothing in the tree records the attempt except the counter — which is why the widget's iterations-versus-nodes gap is the trapped count.

12.5 Implementation in Rust

Everything from Chapter 11 is imported; nothing is re-introduced. The chapter adds one tree type, one trait with a default method, and the planners.

RUST **crates/sampling/src/tree/mod.rs**

```rust
use manifold::Manifold;                                                  // Ch. 5
use crate::{Chart, FreeSpace, Sampler, Steer, Uniform, nn::KdTree};      // Ch. 11
use rand::rngs::SmallRng;

pub type NodeId = usize;
pub struct Node<M: Manifold> { pub q: M::Point, pub parent: Option<NodeId>, pub depth:
   u32 }

/// One tree type for every planner in Part III. Ch. 13 wraps it with costs; it does not
   fork it,
/// so RRT and RRT* can be fed identical samples side by side.
pub struct Tree<M: Chart> { pub nodes: Vec<Node<M>>, nn: KdTree<M>, pub space: M }

```

```rust
impl<M: Chart> Tree<M> {
    pub fn rooted(space: M, q0: M::Point) -> Self {
        let mut nn = KdTree::new(space.clone());
        nn.insert(0, q0.clone());
        Self { nodes: vec![Node { q: q0, parent: None, depth: 0 }], nn, space }
    }
    /// O(log n) expected -- the kd-tree with periodic axes from Chapter 11.
    pub fn nearest(&self, q: &M::Point) -> (NodeId, f64) { self.nn.k_nearest(q, 1)[0] }
    pub fn add(&mut self, q: M::Point, parent: NodeId) -> NodeId {
        let id = self.nodes.len();
        self.nn.insert(id, q.clone());
        self.nodes.push(Node { q, parent: Some(parent), depth: self.nodes[parent].depth +
    1 });
        id
    }
    /// Root ... v, by following parents. Every node has one: that is what "tree" buys.
    pub fn path_to_root(&self, mut v: NodeId) -> Vec<M::Point> {
        let mut out = vec![self.nodes[v].q.clone()];
        while let Some(p) = self.nodes[v].parent { out.push(self.nodes[p].q.clone()); v =
    p; }
        out.reverse();
        out
    }
}
```

The chapter's one addition to Chapter 11's `Steer` is a trait with a default method, blanket- implemented for every steer — so Chapter 21's Dubins steer gets an RRT for free.

RUST **crates/sampling/src/tree/extend.rs**

```rust
/// The three-valued answer RRT-Connect's greedy loop needs (Choset Algs. 11-12).
pub enum ExtendStatus<P> { Reached(P), Advanced(P), Trapped }

/// Ch. 12's addition to Ch. 11's `Steer`: move at most `eta` toward `toward` along the
/// manifold's interpolant, then run the local planner on that short piece.
pub trait Extend<M: Manifold>: Steer<M> {
    fn extend(&self, space: &M, free: &dyn FreeSpace<M>, from: &M::Point, toward: &M::
   Point, eta: f64)
        -> ExtendStatus<M::Point>
    {
        let d = space.dist(from, toward);
        if d <= eta {                                   // close enough: try to reach
   exactly
            return match self.steer(space, free, from, toward) {
                Some(_) => ExtendStatus::Reached(toward.clone()),
                None => ExtendStatus::Trapped,
            };
        }
        // Why a fixed stride: the node that owns q_rand's Voronoi cell moves eta toward
   it.
        // That is the whole Voronoi bias; a variable stride would be a different planner.

        let q_new = space.interpolate(from, toward, eta / d);
        match self.steer(space, free, from, &q_new) {
            Some(_) => ExtendStatus::Advanced(q_new),
            None => ExtendStatus::Trapped,
        }
```

```
    }
}
impl<M: Manifold, S: Steer<M>> Extend<M> for S {}       // every Steer, including Ch. 21's Dubins
```

RUST `crates/sampling/src/tree/rrt.rs`

```rust
pub struct GoalBias { pub p: f64 }                      // default 0.05 (Choset sec .7.2.2)

pub struct Rrt<M: Chart, S: Extend<M>, Smp: Sampler<M>> {
    pub tree: Tree<M>, pub steer: S, pub sampler: Smp, pub eta: f64, pub bias: GoalBias,
    pub rng: SmallRng,
    pub trapped: u64,                                   // the counter the widgets display
    pub goal_node: Option<NodeId>,
}

impl<M: Chart, S: Extend<M>, Smp: Sampler<M>> Rrt<M, S, Smp> {
    /// Choset Alg. 11 driven by the sampler; returns the status so callers can count Trapped.
    pub fn extend_once(&mut self, free: &dyn FreeSpace<M>, goal: &M::Point) -> ExtendStatus<M::Point> {
        let q_rand = if self.rng.random::<f64>() < self.bias.p { goal.clone() }
                     else { match self.sampler.sample(&self.tree.space, free, &mut self.rng) {
                         Some(q) => q, None => return ExtendStatus::Trapped } };
        let (near, _) = self.tree.nearest(&q_rand);
        let st = self.steer.extend(&self.tree.space, free, &self.tree.nodes[near].q, &q_rand, self.eta);
        match &st {
            ExtendStatus::Trapped => self.trapped += 1,
            ExtendStatus::Reached(q) | ExtendStatus::Advanced(q) => {
                let id = self.tree.add(q.clone(), near);
                // Goal test: within eta of the goal, and the last short segment is free.
                if self.goal_node.is_none() && self.tree.space.dist(q, goal) <= self.eta
                    && self.steer.steer(&self.tree.space, free, q, goal).is_some() { self.goal_node = Some(id); }
            }
        }
        st
    }

    /// Choset Alg. 12: repeat extend toward q until Reached or Trapped.
    pub fn connect(&mut self, free: &dyn FreeSpace<M>, q: &M::Point) -> bool {
        loop {
            let (near, _) = self.tree.nearest(q);
            match self.steer.extend(&self.tree.space, free, &self.tree.nodes[near].q, q, self.eta) {
                ExtendStatus::Advanced(qn) => { self.tree.add(qn, near); }
                ExtendStatus::Reached(qn) => { self.tree.add(qn, near); return true; }
                ExtendStatus::Trapped => { self.trapped += 1; return false; }
            }
        }
    }

    /// Choset Alg. 10 plus the goal test, single tree.
    pub fn plan(&mut self, free: &dyn FreeSpace<M>, goal: &M::Point, n: usize) -> Option<
```

```
    Vec<M::Point>> {
         for _ in 0..n { if self.goal_node.is_some() { break; } self.extend_once(free,
    goal); }
         self.goal_node.map(|v| { let mut p = self.tree.path_to_root(v); p.push(goal.clone
    ()); p })
     }
}

/// Choset Alg. 13 with a greediness switch per side; RRT-Connect = (Extend, Connect).
pub enum Greed { Extend, Connect }
pub struct RrtConnect<M: Chart, S: Extend<M>> { pub a: Rrt<M, S, Uniform>, pub b: Rrt<M,
    S, Uniform>, pub greed: (Greed, Greed) }
impl<M: Chart, S: Extend<M>> RrtConnect<M, S> {
    pub fn merge(&mut self, free: &dyn FreeSpace<M>, attempts: usize) -> Option<Vec<M::
    Point>> {
         let (mut t1, mut t2) = (&mut self.a, &mut self.b);
         for _ in 0..attempts {
             let Some(q_rand) = t1.sampler.sample(&t1.tree.space, free, &mut t1.rng) else
    { continue };
             if let Some(q_new) = t1.grow(free, &q_rand, &self.greed.0) {
                 if t2.grow_to(free, &q_new, &self.greed.1) { return Some(join(&self.a, &
    self.b, &q_new)); }
             }
             std::mem::swap(&mut t1, &mut t2);            // Alg. 13 line 9: neither tree
    starves
         }
         None
     }
}
```

EST and SBL share the push and differ in bookkeeping: EST keeps a density weight per node, SBL keeps a grid of cells and a lazy flag per edge.

RUST **crates/sampling/src/tree/est.rs**

```
pub enum Density { Naive { radius: f64 }, GridCells { cell: f64 } }

pub struct Est<M: Chart, S: Steer<M>> {
    pub tree: Tree<M>, pub steer: S, pub radius: f64, pub density: Density,
    weight: Vec<f64>,                        // w_T per node under Density::Naive
    cells: HashMap<Vec<i64>, Vec<NodeId>>,  // occupied cells under Density::GridCells
    pub rng: SmallRng,
}

impl<M: Chart, S: Steer<M>> Est<M, S> {
    /// pi_T: the node to push from. Sparse neighborhoods -- the frontier -- are favored.
    fn select(&mut self) -> NodeId {
        match self.density {
            Density::Naive { .. } => {
                // Why 1/w_T and not uniform: uniform over V pushes the dense interior
    most often.
                let dist = WeightedIndex::new(self.weight.iter().map(|w| 1.0 / w)).unwrap
    ();
                dist.sample(&mut self.rng)
            }
            Density::GridCells { .. } => {
                // sec.7.2.3: a random non-empty cell, then a random node in it -- O(1),
    same bias.
```

```
                let cell = self.cells.values().choose(&mut self.rng).unwrap();
                *cell.choose(&mut self.rng).unwrap()
            }
        }
    }

    /// Choset Alg. 9 from a chosen node: a free sample in its ball, joined by Delta.
    pub fn extend_once(&mut self, free: &dyn FreeSpace<M>) -> Option<NodeId> {
        let from = self.select();
        let q_new = sample_ball(&self.tree.space, &self.tree.nodes[from].q, self.radius,
   &mut self.rng);
        if !free.is_free(&q_new) { return None; }
        self.steer.steer(&self.tree.space, free, &self.tree.nodes[from].q, &q_new)?;
        let id = self.tree.add(q_new, from);
        self.register(id);                  // bump w_T of the neighbors, or file the
   cell
        Some(id)
    }
}
```

RUST **crates/sampling/src/tree/sbl.rs**

```
struct LazyNode<P> { q: P, parent: Option<NodeId>, tree: Side, checked: bool }

impl<M: Chart, S: Steer<M>> Sbl<M, S> {
    /// One iteration: expand the tree whose turn it is (grid-cell pi_T), then try to
   bridge.
    pub fn step(&mut self, free: &dyn FreeSpace<M>) -> Option<Vec<M::Point>> {
        let side = self.turn(); let from = self.select(side);
        let q_new = self.sample_near(from, free)?;            // nodes ARE checked on
   creation
        let id = self.push(LazyNode { q: q_new.clone(), parent: Some(from), tree: side,
   checked: false });
        let (other, d) = self.nearest_in(side.other(), &q_new)?;
        if d > self.radius { return None; }
        self.try_bridge(free, id, other)                       // the only place edges
   get checked
    }

    /// Check the candidate path lazily: the bridge first, then unchecked edges longest-
   first.
    /// The first failure cuts that edge and re-roots the severed piece onto the other
   tree
    /// through the (verified) bridge -- no sample is wasted.
    fn try_bridge(&mut self, free: &dyn FreeSpace<M>, a: NodeId, b: NodeId) -> Option<Vec
   <M::Point>> {
        self.edge_checks += 1;
        self.steer.steer(&self.tree.space, free, &self.q(a), &self.q(b))?;
        for e in self.unchecked_edges_on(a, b).sorted_by_length_desc() {
            self.edge_checks += 1;
            if self.steer.steer(&self.tree.space, free, &self.q(e.parent), &self.q(e.
   child)).is_some() {
                self.nodes[e.child].checked = true;
            } else {
                self.cut_and_transfer(e.child, a, b);
                return None;
            }
        }
```

```
            Some(self.join(a, b))
        }
    }
```

The worked example is an executable whose output the test asserts, and the TypeScript port is pinned to the same four numbers.

RUST **crates/sampling/examples/three_extensions.rs**

```
fn main() {
    let world = BoxWorld::new([0.0, 10.0], [0.0, 10.0], &[[3.0, 4.0, 0.0, 3.0]]);
    let targets = vec![[5.0, 4.0], [2.0, 6.0], [4.0, 1.5]];
    for eta in [1.0, 1.5] {
        let mut rrt = Rrt::new(R2::unit_box(10.0), [1.0, 1.0],
            StraightLine { check: EdgeCheck::Subdivision { step: 0.25 } },
            Scripted::targets(targets.clone()),          // targets need not be free:
   RANDOM_CONFIG draws from Q
            eta, GoalBias { p: 0.0 }, SmallRng::seed_from_u64(12));
        println!("eta = {eta}");
        for i in 1..=3 {
            let st = rrt.extend_once(&world, &[9.0, 9.0]);
            println!("  extension {i}: {st:?}");
        }
        println!("  parents: {:?}, total length {:.3}", rrt.tree.parent_map(), rrt.tree.
   total_length());
    }
}
```

```
eta = 1
  extension 1: q_near = (1, 1), dist 5.0000, q_new = (1.8000, 1.6000), Advanced
  extension 2: q_near = (1.8, 1.6), dist 4.4045, q_new = (1.8454, 2.5990), Advanced
  extension 3: q_near = (1.8, 1.6), dist 2.2023, q_new = (2.7990, 1.5546), Advanced
  parents: {1: 0, 2: 1, 3: 1}, total length 3.000
eta = 1.5
  extension 1: q_new = (2.2000, 1.9000), Advanced
  extension 2: q_new = (2.1335, 3.3985), Advanced
  extension 3: q_near = (2.2, 1.9), q_new = (3.6837, 1.5704) in Q0, Trapped
  parents: {1: 0, 2: 1}, total length 3.000
```

The test pins the $\eta = 1$ trace to 10^{-4} and the statuses exactly; a second test pins the first twenty nodes of a `SmallRng::seed_from_u64(12)` run so the TypeScript port can be compared node by node; a third checks the SRT degeneration against Chapter 11's six-sample edge set. The TypeScript checks `three extensions (η = 1)` and `third extension with η = 1.5 ... Trapped` replay the same trace — the second one computing the blocked configuration $(3.2985, 1.5319)$ from $(1.8, 1.6)$, which is where the port's third extension starts because its first two were the $\eta = 1$ ones.

RUST **crates/sampling/tests/tree.rs**

```
#[test]
fn reproduces_three_extensions() {
```

```
    let (nodes, statuses, length) = trace(1.0);
    assert!(statuses.iter().all(|s| matches!(s, ExtendStatus::Advanced(_))));
    for (got, want) in nodes.iter().zip([[1.8, 1.6], [1.8454, 2.5990], [2.7990, 1.5546]])
    {
        assert!((got[0] - want[0]).abs() < 1e-4 && (got[1] - want[1]).abs() < 1e-4);
    }
    assert!((length - 3.0).abs() < 1e-9);
    let (_, statuses15, _) = trace(1.5);
    assert!(matches!(statuses15[2], ExtendStatus::Trapped));
}

#[test]
fn srt_degenerates_to_prm() {
    let srt = Srt::prm_like(R2::unit_box(10.0), k = 2, Scripted::new(SIX_SAMPLES.to_vec(),
    true));
    srt.add_trees(6); srt.connect_all();
    assert_eq!(srt.edge_labels(), ["12", "23", "36", "45", "56"]);
}
```

12.6 Putting it together: five planners, three worlds

The integration lab runs RRT, EST, RRT-Connect, SBL and KPIECE on the Apartment (Rusty as a disc, $\mathbb{R}^2$), the Workbench (Reach, T^2) and the Lot (Hitch, SE(2)), twenty seeds each, and reports medians of iterations, collision checks and path length. The bars are the data the `tree_bench` example writes; hover for the numbers.

INTERACTIVE · f12.4

Five planners × three worlds

Single- and multi-query planners have different economics, lazy checking buys checks with path quality — and a feasible path for Hitch under a straight-line steer is a path the car cannot drive.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch12-tree-planners#f12.4

Four things to read off it.

Single-query planners beat the roadmap at their own game. On the Workbench, RRT-Connect merges in a handful of attempts and a few hundred collision checks — less than a hundredth of what the Chapter 11 roadmap spent to learn the whole torus. That is the economics the hook promised: pay for the query you have.

The lazy planner buys checks with path quality. SBL's collision-check bill is the smallest of the five on the Apartment, and its path is the longest: it verified only the edges on one bridging path and never looked at the rest. Shortcutting repairs some of that, as w12.2's last column shows.

Push is more expensive than pull in these worlds. EST spends an order of magnitude more checks than RRT on the Apartment — its ball samples near frontier nodes fail often in the rooms' corners, and each failure is a check — while its coverage guarantee is no better. Where EST and KPIECE earn their keep is in spaces with no useful metric for the pull, which is Chapter 14's forward-propagation preview.

The Hitch row is wrong, on purpose. Every planner finds a path for Hitch in the Lot on most seeds with a straight-line steer in SE(2) — one that interpolates heading as freely as position, so the car slides sideways and spins in place along it. It is drawn in orange with the sentence it deserves: *a steer must respect the robot.* Hitch's kinematics exist in Chapter 2 and this chapter deliberately does not use them. Chapter 14 previews the alternative — replace `extend` with a simulator, so that EST's push returns as forward propagation — and Chapter 21 does it properly with a Dubins `Steer` that drops into this chapter's `Rrt` through the blanket `Extend`.

And one thing none of the planners can answer. Look at any of the paths: jagged, a random accident of sample order, and feasible. How *good* is it — and does more sampling make it better? For RRT the answer, proved in 2011, is *no, almost surely*. That is Chapter 13.

12.7 Exercises

1 FOUNDATION •• **Voronoi bias under a weighted metric**

Prove the Voronoi-bias lemma for the weighted SE(2) metric $w_t\|X' - X''\| + w_r|\theta' - \theta''|_{S^1}$, and describe how the Voronoi cells deform as $w_r \to 0$. What does RRT do to Hitch's heading in that limit, and why does the resulting tree look two-dimensional?

HINTS

<>Steps 1 and 2 of the lemma used only that nearest-neighbor is defined by dist and that q_rand is uniform. Neither depends on which metric.</> <>With w_r = 0, two configurations at the same position but different headings are at distance zero. Every cell becomes a cylinder over a planar cell.</> <>The nearest node is then chosen by position alone, and the heading of q_new is whatever interpolate produces — a free spin the car cannot execute. The tree's projection to the plane is an ordinary planar RRT.</>

2 FOUNDATION •• **Why Algorithm 13 swaps**

Choset's MERGE swaps T_1 and T_2 only after a successful first extension. Show by example that without the swap one tree can starve — receive no new nodes for arbitrarily many iterations — and state the invariant the swap maintains about the two trees' expected node counts. Then explain why RRT-Connect's Connect on the second tree makes the swap *more* important, not less.

3 CONCEPTUAL • **Predict the step-size trade-off, then verify**

PREDICT FIRST

In Tree Grower at $\eta = 0.15$ rad the first solution takes many hundred iterations with a modest fraction Trapped. Before moving the slider: at $\eta = 0.6$ rad, what happens to the iteration count and to the Trapped fraction?

- Both fall — bigger steps cover the square faster and fail no more often

  The first half is right and the second is not. A step four times longer is four times more likely to run into the amber C-obstacle from a node near it, so the Trapped fraction rises; what falls is the number of successful steps needed.
- ✓ Iterations fall, the Trapped fraction rises

  Right. Fewer steps cross the square, but each long step is more likely to be blocked, so a larger share of iterations add nothing. The minimizing η is where those two effects balance — well below the largest step the slider allows.
- Both rise — long steps keep missing the goal

  The goal test accepts any node within η of the goal with a free segment to it, so a large η makes the goal easier, not harder, to hit.

Verify, then sweep η and find the value that minimizes iterations to the first solution on a fixed seed. It is not the largest: past it, the Trapped fraction grows faster than the step saves iterations. Use the Voronoi overlay to see why — a node near the amber band owns a large cell on its far side, is pulled often, and fails every time its step is longer than its clearance.

4 CONCEPTUAL •• **Where greedy loses**

In Bidirectional Connect, compare Connect/Connect with Extend/Extend on median collision checks over at least twenty seeds. Then raise the merge budget and look for the regime in which the greedy planner's advantage disappears or reverses. Relate the

crossover to the expected length of a Connect run before it ends Trapped — a geometric random variable whose parameter is the probability that one step of η is blocked.

WORK IT OUT

If each step of a Connect run is blocked independently with probability 0.3, what is the expected number of steps (and collision-check rows) in a run that ends Trapped?

Answer. 3.33 A geometric variable with failure probability 0.3 has mean $1/0.3 = 3.33$ steps, each a row of $\lceil \eta/\text{step} \rceil$ collision checks — all spent even though the run added nodes that may be useless. Extend spends one row and asks a fresh random target.

5 PRACTICAL •• **KPIECE with a projection**

Implement KPIECE's cell-importance selection over a *projection* of the configuration — the end-effector position for Reach, rather than the joint angles — in `kpiece.rs`. Keep the importance formula of this chapter (score, selections, neighbors, coverage), initialize a cell's score from the iteration of its discovery, and decay it after every expansion. Add it to `tree_bench` and report where it beats RRT and where it does not, on all three worlds.

6 PRACTICAL ••• **Resolution-aware lazy edges**

Make `Sbl`'s lazy edge store resolution-aware, as Choset's description of lazy evaluation suggests: record the coarsest resolution at which an edge has been checked so far, and when an edge recurs on a candidate path, resume from that resolution instead of restarting. Then move the goal slightly, keep both trees, and measure over fifty seeds how many collision checks reuse saves against planning from scratch.

12.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

§7.2 and §7.3 are this chapter's source: Algorithms 8–14, the two-family framing, the step-size and metric-sensitivity discussion, SBL (ref. 367, Sá nchez and Latombe), and SRT (refs. 14, 43). The epigraph is from §7.2. `link`

LaValle, S. M. and Kuffner, J. J. (2001). Randomized Kinodynamic Planning. *International Journal of Robotics Research 20(5), 378–400*.
RRTs, with the Voronoi-bias intuition and the kinodynamic setting this chapter defers to Chapters 14 and 21. doi:10.1177/02783640122067453

Kuffner, J. J. and LaValle, S. M. (2000). RRT-Connect: An Efficient Approach to Single-Query Path Planning. *IEEE International Conference on Robotics and Automation, 995–1001*.
The (Extend, Connect) setting of w12.2's dial, and the greedy heuristic the chapter labels classical. doi:10.1109/ROBOT.2000.844730

Hsu, D., Latombe, J.-C., and Motwani, R. (1999). Path Planning in Expansive Configuration Spaces. *International Journal of Computational Geometry and Applications 9(4–5), 495–512*.
EST and its coverage analysis — the push, the 1/w_T weighting, and the linking-sequence argument Derivation 3 follows. doi:10.1142/S0218195999000285

Kleinbort, M., Solovey, K., Littlefield, Z., Bekris, K. E., and Halperin, D. (2019). Probabilistic Completeness of RRT for Geometric and Kinodynamic Planning with Forward Propagation. *IEEE Robotics and Automation Letters 4(2)*.
The rigorous completeness proof this chapter cites in place of the 2005 textbook's sentence — including the delicate step that the nearest node need not be the one in the current ball. link

Şucan, I. A., Moll, M., and Kavraki, L. E. (2012). The Open Motion Planning Library. *IEEE Robotics and Automation Magazine 19(4), 72–82*.
Reference implementations of EST, RRT, RRT-Connect, SBL and KPIECE; KPIECE itself is Şucan and Kavraki's WAFR 2008 paper, Kinodynamic Motion Planning by Interior-Exterior Cell Exploration, whose importance formula this chapter's implementation follows. doi:10.1109/MRA.2012.2205651

LaValle, S. M. (2006). Planning Algorithms. *Cambridge University Press*.
§5.5 is the fullest textbook treatment of RRTs, bidirectional variants and their failure modes. link

CHAPTER 13

Optimal Sampling-Based Planning

SAMPLING-BASED PLANNING · Advanced · 80 min

> *In general, if paths with certain optimality criteria are desired, it is worth trying to build these paths during the roadmap construction phase of PRM. For example, a large dense roadmap will probably yield shorter paths than a smaller and sparser roadmap.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), §7.1.2*

Chapters 11 (Chapter 11) and 12 (Chapter 12) bought feasibility with randomness and proved the price: the failure probability decays exponentially in the number of samples. They said nothing about the *quality* of the path, and anyone who has watched Tree Grower knows why — the first path is a jagged accident of sample order.

This chapter asks the question Choset's 2005 text could not. Does more sampling make the path better? For RRT the answer, proved by Karaman and Frazzoli in 2011, is *no, almost surely*: the tree is anchored to its root through edges it never reconsiders. The repair is two local operations, choose-parent and rewire, inside a neighborhood of radius $r_n = \gamma(\log n/n)^{1/d}$, and the "aha" is *why that exact rate*. The ball must shrink so that the per-iteration cost stays $O(\log n)$, but no faster than the scale at which a random geometric graph stays connected — Chapter 11's tiling balls, now required to keep receiving samples forever.

From RRT* and PRM* the chapter builds the modern batch family — Informed RRT* samples only where improvement is possible, FMT* and BIT* search an implicit random geometric graph with a heuristic — and ends with what "anytime" means for a robot that must move before the planner is done. Nothing in this chapter is in Choset; its scaffolding, the ball tiling and the expansiveness constants,

is. The sister book's Chapter 20 states the central theorem in a paragraph and defers here.

AFTER THIS CHAPTER YOU CAN

- State what asymptotic optimality means for a sampling-based planner and prove that RRT does not have it — its best path converges, almost surely, to the wrong value
- Derive the two local operations, choose-parent and rewire, that repair it, and trace one RRT* iteration on a five-node tree by hand
- Explain why the neighborhood radius must be exactly r_n = γ(log n/n)^{1/d}: shrink faster and the tiling balls go empty, slower and the per-iteration work stops being O(log n) — and compute both the 2011 and the 2020 constants
- Derive the informed set as the ellipse with foci at start and goal and string length c_best, and its measure as the share of the space still worth sampling
- Read FMT *as lazy dynamic programming on one batch and BIT* as A* over an implicit random geometric graph, and say what each has at a given deadline
- Write sampling::optimal — CostTree, RrtStar, PrmStar, InformedSampler, Fmt, Bit, Anytime — and run the anytime contract: stop at a deadline, execute, keep refining

ASSUMED

- Everything in Chapter 11 is imported: Sampler, KdTree::within for the r_n-ball, Steer, FreeSpace, Theorem 7.4.1's tiling constant and the (ε, α, β) vocabulary
- Chapter 12's Tree, Extend and RRT — the baseline whose Voronoi bias is unchanged by rewiring
- A* on the inflated lattice as a cost floor; g + h as an edge key (Chapter 6)
- Manifold::dist as a cost; the geodesic through the seam on T^2 (Chapter 5)

13.1 The problem: the path does not get better

Let Reach execute the path Chapter 12's RRT found on the Workbench.

INTERACTIVE · w13.0

Reach wobbles, then doesn't

Feasibility and quality are separate contracts: a tree planner's first path is an accident of sample order, and more sampling leaves it alone unless the planner is allowed to rewire.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch13-optimal-sampling#w13.0

Three needless reversals of the fingertip, a loop that goes the long way round the block. Give the planner ten thousand more iterations and the path does not change: the tree gets denser everywhere, and the path to the goal is the same jagged chain of edges it was when it first arrived. On the right, the *same* random samples, in the same order, with one flag flipped: the tree is allowed to reconsider which node each node hangs from. The cost drops within the first few hundred iterations after the first solution and keeps dropping, and the arm goes the short way.

Feasibility and quality are separate contracts. Part III has so far delivered only the first.

13.2 Building intuition: a tree allowed to change its mind

The ordinary RRT adds a node by attaching it to the nearest tree node, and that attachment is final. RRT* adds the same node — the Voronoi bias of Chapter 12 is untouched — and then does two things in a ball around it. **Choose-parent:** among the tree nodes in the ball, attach to the one through which the new node's cost-to-come is smallest, not the nearest. **Rewire:** for every other node in the ball, if going through the new node would be cheaper than its current cost, make the new node its parent and propagate the saving down its subtree.

INTERACTIVE · w13.1

Rewire Watch

More samples do not make RRT's path better — its cost curve is flat forever. RRT* on the very same samples descends, because choose-parent and rewire let the tree change its mind inside a ball that shrinks just slowly enough.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch13-optimal-sampling#w13.1

Left and right consume the same seeded stream of random targets, so they have the same nodes; only the edges differ. Watch the two cost curves. The left one is flat — it changes only on the rare occasion that a *new* node happens to enter the goal region more cheaply than the old one, and it never approaches the

dashed lattice floor. The right one descends, and every drop is a flash of rewired edges somewhere in the tree. The misconception this widget exists to kill is *"more samples make RRT's path better."* They do not. They make RRT*'s path better, and the whole chapter is about what the asterisk costs.

Two more things to try. Slide γ/γ^* below one. The right curve stalls — not immediately, but it stops approaching the floor — because the ball now shrinks faster than samples arrive to fill it, and the theorem's hypothesis has failed. Slide it to three and the descent is faster per iteration and slower per second: every insertion now considers dozens of neighbors, and the histogram of $|Q_{near}|$ tells you how many. The constant γ^* is where those two pressures balance, and it is a *computed* number, not a tuning knob.

THE ONE SENTENCE VERSION

RRT's path is frozen by its parent pointers; RRT* re-chooses the parent of every new node and of every node near it inside a ball of radius $\gamma(\log n/n)^{1/d}$, which is the smallest ball that keeps the random geometric graph connected along the optimal path and the largest that keeps the per-iteration work logarithmic — and with it the best path's cost converges to the optimum almost surely.

13.3 The mathematics

$c(\sigma)$, c^*, c_n^{ALG}	path cost (dist-length); the optimal cost; the best cost ALG holds after n samples
Cost(v)	cost-to-come of tree node v along its parent chain
ζ_d	volume of the unit ball in $\mathbb{R}^d$ ($\zeta_2 = \pi$)
γ, γ^*, k_n	the radius constant; its critical value; k_n = k* log n book-wide with n, r_n
Q_{near}	{ v : dist(v, q_new) ≤ r_n } — or the k_n nearest
δ-robust	a δ-clear path exists (feasibility) / c* is the limit of such paths' costs (optimality)
$X_{\hat{f}}$, c_{best}, c_{min}	the informed set; the incumbent cost; dist(q_start, q_goal)
$\hat{g}$, $\hat{h}$, $\hat{c}$	admissible estimates of cost-to-come, cost-to-go, edge cost
$G_n = (V_n, E_n)$	the random geometric graph on n samples with connection radius r_n

13.3.1 Definitions

DEFINITIONS — ASYMPTOTIC OPTIMALITY, ALMOST-SURE SUBOPTIMALITY

A planner is **asymptotically optimal** if, for every robustly feasible problem with a robustly optimal solution,

$$\Pr\left[\lim_{n\to\infty} c_n^{\mathrm{ALG}} = c^*\right] = 1.$$

It is **almost surely suboptimal** if $\Pr[\lim_n c_n^{\mathrm{ALG}} = c^*] = 0$: the limit exists — c_n is monotone — and is wrong with probability one. Robustness is the regularity the theorems need: a path that can be perturbed by δ without leaving Q_{free}, and an optimum that is the limit of such paths' costs, so that a shrinking tube around it stays free.

DEFINITIONS — CHOOSE-PARENT, REWIRE, THE INFORMED SET, ANYTIME

Choose-parent. Attach q_{new} to the $v \in Q_{near}$ minimizing $\mathrm{Cost}(v)+c(v, q_{new})$ subject to $\Delta(v, q_{new}) \neq \mathrm{NIL}$.
Rewire. For each $v \in Q_{near}$ with $\mathrm{Cost}(q_{new}) + c(q_{new}, v) < \mathrm{Cost}(v)$ and $\Delta(q_{new}, v) \neq \mathrm{NIL}$, set $\mathrm{par}(v) := q_{new}$ and propagate the decrease to v's subtree.
Informed set. $X_{\hat{f}} = \{x : \hat{g}(x) + \hat{h}(x) \leq c_{best}\}$. For `dist` cost in $\mathbb{R}^d$ it is the prolate hyperspheroid with foci $q_{\mathrm{start}}, q_{\mathrm{goal}}$, transverse diameter c_{best} and conjugate diameter $\sqrt{c_{best}^2 - c_{min}^2}$.
Anytime planner. Returns a feasible solution early and improves it monotonically until interrupted — "deadline in, best-so-far out".

13.3.2 RRT is almost surely suboptimal

THEOREM (KARAMAN AND FRAZZOLI 2011)

For $d \geq 2$, `dist` cost, and the paper's regularity assumptions — robust feasibility, a robustly optimal path, finite $\mu(Q_{\mathrm{free}})$ — the RRT's best cost converges to a suboptimal value: $\Pr[\lim_n c_n^{\mathrm{RRT}} = c^*] = 0$.

DERIVATION **Why the tree's first children decide its fate**

$\Pr[\lim_n c_n^{\mathrm{RRT}} = c^*] = 0$

Step 1 — every path passes through a child of the root. Any tree path from q_{start} to the goal leaves the root along one of the root's edges. The best path's cost is at least the best over the root's children u of $c(q_{\text{start}}, u)$ plus the optimal cost from u to the goal.
Step 2 — the first children leave in the wrong directions. With positive probability, the root's first few children all leave at angles bounded away from the optimal path's initial direction — the first few random targets simply landed elsewhere. Call the cone of good initial directions C.
Step 3 — later good children are summably unlikely. For the root to acquire a new child in C at iteration n, a random target must fall in the root's Voronoi cell intersected with C. That cell shrinks as the tree densifies around the root, so the probability of acquiring a good child at iteration n decays with n fast enough that its sum over n is finite.
Step 4 — Borel–Cantelli. A sequence of events whose probabilities sum to a finite value happens only finitely often, almost surely. So the root almost surely acquires only finitely many good children — and with positive probability, by Step 2, none at all.
Step 5 — a fixed loss. A path that leaves the root outside C costs at least $c^* + \epsilon$ for an $\epsilon > 0$ depending on the cone, by the regularity of the optimum. With positive probability the limit of c_n is therefore at least $c^* + \epsilon$; the paper sharpens this to probability one. ■
Why shortcutting does not rescue it. Chapter 11's greedy shortcutting replaces pieces of the path by geodesics between its own points. It cannot change which side of an obstacle the path passes — its homotopy class — and RRT's commitment through a bad first child is often exactly that. Exercise 6 asks for the two-class map where shortcutting can never win.

13.3.3 PRM*, RRG and RRT* are asymptotically optimal

THEOREM (KARAMAN AND FRAZZOLI 2011)

With the connection radius

$$r_n = \gamma\left(\frac{\log n}{n}\right)^{1/d}, \qquad \gamma > \gamma^*_{2011} = 2\left(1 + \frac{1}{d}\right)^{1/d}\left(\frac{\mu(Q_{\text{free}})}{\zeta_d}\right)^{1/d},$$

or the k-nearest rule $k_n = k^* \log n$ with $k^* > e(1 + 1/d)$, PRM* and RRT* satisfy $\Pr[\lim_n c_n = c^*] = 1$, and the expected per-iteration cost stays $O(\log n)$ because $\mathbb{E}|Q_{near}| = \Theta(\log n)$.

DERIVATION **The shrinking tiling, and where the constant comes from**

$\sum_n \Pr[\text{some ball empty at } n] < \infty \iff c\gamma^d > 1 + 1/d$

Step 1 — fix a robustly optimal path. Let σ^* have clearance δ, so a tube of radius δ around it is free.
Step 2 — tile it with shrinking balls. As in Theorem 7.4.1, cover σ^* with balls — but of radius proportional to r_n, say $r_n/4$, with centers $r_n/4$ apart, so that any two consecutive balls lie inside a common ball of radius $r_n/2 \le \delta$ once n is large. The number of balls is $M_n \approx 4L/r_n$, growing like $(n/\log n)^{1/d}$.
Step 3 — one empty ball. A ball of radius $r_n/4$ has measure $\zeta_d(r_n/4)^d = c\,\gamma^d \log n/n$ for the constant $c = \zeta_d/(4^d \mu(Q_{\text{free}}))$, measuring relative to Q_{free}. Being missed by all n samples has probability

$$\left(1 - \frac{c\,\gamma^d \log n}{n}\right)^n \le e^{-c\gamma^d \log n} = n^{-c\gamma^d}.$$

Step 4 — the union bound, and the constant. The probability that *some* ball is empty at stage n is at most $M_n n^{-c\gamma^d} \approx 4L\,(n/\log n)^{1/d}\, n^{-c\gamma^d}$. Summing over n converges iff the exponent satisfies $c\gamma^d - 1/d > 1$, that is $c\gamma^d > 1 + 1/d$. Solving for γ gives a threshold of the form $(1+1/d)^{1/d}(\mu(Q_{\text{free}})/\zeta_d)^{1/d}$ times a geometric factor from the ball radii — the 2 in γ^*_{2011}. *This step is where* $(1 + 1/d)$ *enters*, and Exercise 1 asks for it with the constants carried.
Step 5 — Borel–Cantelli. Summable, so almost surely every ball is hit for all large n.
Step 6 — the graph shadows the path. Samples in consecutive balls are within r_n of each other and connected by a free segment (Step 2), so G_n contains a path shadowing σ^* whose cost tends to c^* as the balls shrink. PRM*'s Dijkstra finds something at least that good.
Step 7 — RRT* follows. Choose-parent and rewire make the tree's cost-to-come at each sample match the shortest path in G_n along the shadow, in the limit — this is the step the 2020 correction is about. ■
The other direction. Why not take γ huge and be done? Because $\mathbb{E}|Q_{near}| = n \cdot \mu(B_{r_n})/\mu(Q_{\text{free}}) \propto \gamma^d \log n$: the per-iteration work is logarithmic for *any* fixed γ, but its constant is γ^d, and the η cap on edge length means a large ball also wastes steer calls on neighbors the planner cannot reach in one step. The rate $(\log n/n)^{1/d}$ is forced by Step 4; the constant is a budget.

INTERACTIVE · `f13.4`

The shrinking tiling

Why r_n ∝ (log n / n)^{1/d}: the balls must shrink so that each node has only O(log n) neighbors, but no faster than the rate at which every ball along the optimal path keeps receiving samples forever.

Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch13-optimal-sampling#f13.4`

13.3.4 The 2020 correction

THEOREM (SOLOVEY, JANSON, SCHMERLING, FRAZZOLI AND PAVONE 2020)

The 2011 proof of RRT*'s optimality has a gap at Step 7: rewiring reconsiders only the nodes inside the *current* ball, so the tree need not reproduce the RRG's shortest path along the shadow. A repaired argument establishes asymptotic optimality of RRT* — with a *smaller* constant,

$$\gamma > \gamma^*_{2020} = \big(2(1+1/d)\big)^{1/d} \left(\frac{\mu(Q_{\text{free}})}{\zeta_d}\right)^{1/d},$$

and correspondingly for the k-nearest variant.

DERIVATION **Repairing Step 7**

asymptotic optimality of RRT* with $\gamma^*_{2020} < \gamma^*_{2011}$

Step 1 — locate the gap. In Step 7 above, the shadow path exists in G_n, but RRT* is a tree: a node's parent is chosen among the nodes present *when it was inserted*, and rewiring later touches only nodes within r_n of each new node. Nothing guarantees that the tree's chain of parents ever aligns with the graph's shortest path.
Step 2 — track a chain in the tree directly. Instead of comparing with the graph, follow a sequence of tree nodes, one per tiling ball along σ^*, and bound the tree path's cost through them.
Step 3 — samples must arrive in a usable order. For the chain to form, each ball must contain a sample *and* the samples must arrive so that when the sample in ball $i+1$ is inserted, a node in ball i is already present and within r_n — then choose-parent can pick it, or a later rewire can fix it. That is a per-ball probability, smaller than mere occupancy.
Step 4 — close the union bound. The modified events are still summable for γ above a threshold, and the threshold that comes out is γ^*_{2020} — smaller than the 2011 constant, because the repaired argument needs one sample per ball at the right time rather than a geometric factor of slack. ■
The implementation names both constants, defaults to the corrected one,

and the widget lets you pick either. The chapter says plainly: the 2011 proof had a gap that took nine years to notice, and the result survived.

13.3.5 The $(\epsilon, \alpha, \beta)$ lens, and the k-nearest variant

Two remarks tie the theorem back to Chapter 11. First, Step 6 needs consecutive balls to *see* each other through the local planner, which is ϵ-goodness in miniature: the shadowing balls are mutually visible because the tube around σ^* is free. A space that is not ϵ-good — a corridor thinner than every r_n for the n you can afford — defeats the asymptotic argument in practice exactly as it defeated Theorem 7.4.2's constants. Second, the k-nearest rule is the same theorem in a different coordinate. In a region of density ρ the k_n nearest neighbors occupy a ball of radius $(k_n/\rho\zeta_d)^{1/d} \propto (\log n/n)^{1/d}$ — the radius rule again, with the density made implicit — and the constant $k^* = e(1 + 1/d)$ comes out of the same union bound with a Chernoff bound on a Poisson count in place of the emptiness probability. The k-nearest rule adapts to local density, which is why some implementations prefer it; the radius rule is the one whose constant is tied to $\mu(\mathcal{Q}_{\text{free}})$ and therefore to the world. The check `k-nearest RRT*` runs both on seed 13 and finds their costs within one percent of each other.

13.3.6 The constants, for $n = 1000$ in the unit square

$(\log n/n)^{1/2} = (6.9078/1000)^{1/2} = 0.08311$. With $d = 2$, $\mu(\mathcal{Q}_{\text{free}}) = 1$, $\zeta_2 = \pi$:

constant	value	r_{1000}
$\gamma^*_{2011} = 2{\cdot}1.5^{1/2}\pi^{-1/2}$	1.3820	0.1149
$\gamma^*_{2020} = 3^{1/2}\pi^{-1/2}$	0.9772	0.0812
$k^* = e(1 + 1/2)$	4.077	$k_{1000} = \lceil 4.077 \cdot 6.9078 \rceil = 29$

A ball of radius 0.08 in a unit square after a thousand samples: about twenty neighbors to consider per insertion, each a steer call. That is the price of the asterisk, and the check `r_1000 in the unit square` pins every number in the table.

13.3.7 Informed sampling is exactly where improvement is possible

THEOREM (GAMMELL, SRINIVASA AND BARFOOT 2014)

With `dist` cost and admissible $\hat{g}, \hat{h}$, no configuration outside $X_{\hat{f}}$ lies on a path cheaper than c_{best}. Sampling uniformly in $X_{\hat{f}}$ preserves RRT*'s guarantees and multiplies the probability of a useful sample by $\mu(\mathcal{Q}_{\text{free}})/\mu(X_{\hat{f}})$.

DERIVATION **The ellipse, and how to sample it**

X_{\hat f} = \text{ellipse with foci } \qstart, \qgoal,\ \text{string length } c_{best}

Step 1 — triangle inequality. Any path through x costs at least $\text{dist}(q_{\text{start}}, x) + \text{dist}(x, q_{\text{goal}}) = \hat{g}(x) + \hat{h}(x)$.

Step 2 — outside is useless. If $\hat{g}(x) + \hat{h}(x) > c_{best}$, no path through x can beat the incumbent. The set of useful points is $\{x : \hat{g}(x) + \hat{h}(x) \le c_{best}\}$ — the set of points whose string from one focus to the other has length at most c_{best}: an ellipse in the plane, a prolate hyperspheroid in $\mathbb{R}^d$, with transverse semi-axis $c_{best}/2$ and conjugate semi-axes $\sqrt{c_{best}^2 - c_{min}^2}/2$.

Step 3 — sample it directly. Draw a uniform point of the unit ball, scale by $\text{diag}(c_{best}/2, \sqrt{c_{best}^2 - c_{min}^2}/2, \dots)$, rotate so the first axis points along $(q_{\text{goal}} - q_{\text{start}})/c_{min}$ — a Householder reflection does it in one line — and translate to the midpoint. Uniform in the ball maps to uniform in the ellipsoid because the map is affine.

Step 4 — reject what leaves Q. Early on the ellipse spills far outside the room, and the draws that leave the chart's box are rejected. On T^2 the formula is wrong once c_{best} exceeds the torus circumference (Exercise 2); the implementation falls back to rejection against the manifold metric there, exact and slower. ■

Necessary, not sufficient. Every improving path lies in the ellipse; most of the ellipse lies in no improving path, because obstacles cut it. The readout in the widget is $\mu(X_{\hat{f}} \cap Q)/\mu(Q)$ — the part of the room worth sampling at all — and the check `informed set measure` pins it against a Monte Carlo count and the unclipped πab.

INTERACTIVE · w13.2

Ellipse Shrinker

After the first solution, not every sample is equally useful: only configurations inside the ellipse with foci at start and goal and string length c_best can lie on a shorter path, and the ellipse shrinks with every improvement.

Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch13-optimal-sampling#w13.2`

13.3.8 FMT* and BIT*: search the graph you never build

THEOREM (JANSON, SCHMERLING, CLARK AND PAVONE 2015)

On one batch of n samples with $r_n = \gamma(\log n/n)^{1/d}$ and $\gamma > 2(1/d)^{1/d}(\mu(\mathcal{Q}_{\text{free}})/\zeta_d)^{1/d}$, growing the tree outward in cost-to-come order and connecting each frontier node to the locally optimal parent *chosen ignoring obstacles, then checking only that edge* is asymptotically optimal with $O(n \log n)$ expected collision checks.

DERIVATION **Lazy dynamic programming is wrong only where it does not matter**

$\Pr[\lim c_n^{\text{FMT}^*} = c^*] = 1$ with $O(n \log n)$ checks

Step 1 — the lazy choice. FMT* expands the open node z of least cost-to-come and, for each unvisited x within r_n, picks the parent y minimizing $\text{Cost}(y) + c(y, x)$ over the open nodes in x's ball *without* collision checking, then checks the one edge (y, x). If it is blocked, x is left for later — it is not given a second-best parent.

Step 2 — when the lazy choice is wrong. The choice is wrong only when the optimal in-ball parent is blocked: x gets no parent now and may get a worse one later. Inside a δ-clear corridor the ball of radius r_n is eventually inside the corridor, and that event stops happening.

Step 3 — bounded error. Each lazily chosen edge is within $O(r_n)$ of the local optimum, there are $O(1/r_n)$ edges along the shadow path, and the total excess stays bounded and vanishes along a subsequence.

Step 4 — the bill. One nearest-neighbor structure, $O(\log n)$ expected neighbors per node, one collision check per *accepted* edge: $O(n \log n)$ in all, and no rewiring. ■

BIT.* Gammell, Srinivasa and Barfoot's Batch Informed Trees runs the same idea in batches. Each batch adds informed samples to an implicit random geometric graph of radius r_n and searches it exactly like Chapter 6's A*: an edge queue keyed $\hat{g}(v) + \hat{c}(v, x) + \hat{h}(x)$, a vertex queue keyed $\hat{g}(v) + \hat{h}(v)$, collision checks only when an edge is popped. After a solution, samples and vertices that cannot improve it are pruned, and the tree is kept across batches so the next batch repairs rather than rebuilds — an LPA*-style reuse.

13.4 The algorithm

ALGORITHM `RRT* -- one iteration: sample, extend, choose-parent, rewire` `COST |Q_near| = O(log n) steer calls per iteration, O(n log n) total; cost propagation O(|subtree|) per rewire`

in the cost tree T, the sampler, η, the neighborhood rule (r_n or k_n), the local planner Δ

out T with q_new inserted at its cheapest reachable parent and its neighbors rewired; a new incumbent if a goal-connected node got cheaper

1: $q_{rand} \leftarrow$ SAMPLE; $q_{near} \leftarrow$ NEAREST(T, q_{rand}); $q_{new} \leftarrow$ EXTEND(q_{near}, q_{rand}, η) — `Trapped` ends the iteration
2: $Q_{near} \leftarrow \{ v \in T : \text{dist}(v, q_{new}) \le \min(r_n, \eta) \}$, or the k_n nearest; always including q_{near}
3: **choose-parent:** $v^* \leftarrow \arg\min_{v \in Q_{near}} \text{Cost}(v) + c(v, q_{new})$ subject to $\Delta(v, q_{new}) \neq$ NIL — try candidates in order of that sum and stop at the first free one
4: add q_{new} with parent v^* and $\text{Cost}(q_{new}) = \text{Cost}(v^*) + c(v^*, q_{new})$
5: **rewire: for all** $v \in Q_{near} \setminus \{v^*\}$ **do**
6: **if** $\text{Cost}(q_{new}) + c(q_{new}, v) < \text{Cost}(v)$ **and** $\Delta(q_{new}, v) \neq$ NIL **then** $\text{par}(v) \leftarrow q_{new}$; propagate the decrease through v's subtree
7: **goal test:** if $\text{dist}(q_{new}, q_{\text{goal}}) \le \eta$ and $\Delta(q_{new}, q_{\text{goal}}) \neq$ NIL, record q_{new} as goal-connected
8: incumbent $\leftarrow$ min over goal-connected v of $\text{Cost}(v) + c(v, q_{\text{goal}})$ — a rewire deep in the tree lowers it without touching the goal

Line 3 is sorted so that the first free candidate is the answer; line 2's η cap is what keeps the steer calls short. With `rewire = false` and $Q_{near} = \{q_{near}\}$ the same code is Chapter 12's RRT, which is how the widgets feed one seed to both.

ALGORITHM `PRM* -- the radius rule for a roadmap` `COST O(n log n) expected edges, each one steer call`

in n, γ (or k*), the sampler, Δ

out a roadmap with O(n log n) expected edges whose shortest paths converge to the optimum

1: **for** $i = 1 \ldots n$ **do** $q_i \leftarrow$ SAMPLE; $r \leftarrow \gamma(\log i / i)^{1/d}$ (or $k \leftarrow \lceil k^* \log i \rceil$)
2: **for all** q_j within r of q_i (or its k nearest) **do if** $\Delta(q_i, q_j) \neq$ NIL **then** add the edge with cost $c(q_i, q_j)$
3: query by Chapter 11's Algorithm 7

ALGORITHM `FMT* -- lazy dynamic programming on one batch` COST `O(n log n)`
`expected: one kd-tree, O(log n) neighbors per node, one check per accepted edge`
in n samples including q_start, q_goal; r_n; Δ
out a tree over the samples and the path to q_goal, or failure if the open set empties first

1: Cost(q_{start}) $\leftarrow$ 0; OPEN $\leftarrow$ $\{q_{\text{start}}\}$; all other samples UNVISITED
2: **while** OPEN is not empty **do**
3: $z \leftarrow$ the OPEN node of least Cost; **if** $z = q_{\text{goal}}$ **then return** the path
4: **for all** UNVISITED x within r_n of z **do**
5: $y \leftarrow$ arg min over OPEN nodes within r_n of x of Cost(y) + $c(y, x)$ — obstacles ignored
6: **if** $\Delta(y, x) \neq$ NIL **then** par(x) $\leftarrow$ y; Cost(x) $\leftarrow$ Cost(y) + $c(y, x)$; move x to OPEN
7: move z to CLOSED
8: **return** failure

ALGORITHM `BIT* -- one batch, searched like A*` COST per batch: `an implicit RGG of radius r_n over tree vertices and samples, searched by an edge queue keyed` $\hat{g}$`(v) +` $\hat{c}$`(v, x) +` $\hat{h}$`(x) with collision checks only on pop`
in the current tree, the unconnected samples, a batch size, γ, the incumbent c_best
out an improved incumbent, or an exhausted batch

1: **prune:** drop every sample and vertex x with $\hat{g}(x) + \hat{h}(x) \geq c_{best}$; detach children of dropped vertices and return them to the sample set if they could still help
2: **sample:** draw `batch` new configurations from the informed set; $r_n \leftarrow \gamma(\log n/n)^{1/d}$ over vertices plus samples
3: **re-key:** every tree vertex v enters the vertex queue keyed $g(v) + \hat{h}(v)$
4: **repeat** while a queue is non-empty:
5: while the best vertex key $\leq$ the best edge key, pop a vertex and push its edges (v, x) to samples (and to vertices it could rewire) within r_n, keyed $\hat{g}(v) + \hat{c}(v, x) + \hat{h}(x)$
6: pop the best edge (v, x); **if** $g(v) + \hat{c}(v, x) + \hat{h}(x) \geq c_{best}$ **then** the batch is exhausted — clear both queues
7: **if** $g(v) + c(v, x) < g(x)$ **and** $\Delta(v, x) \neq$ NIL **then** attach or rewire x under v, propagate, and push x to the vertex queue; update c_{best} if q_{goal} got cheaper

13.4.1 The five-node rewire, by hand

Euclidean plane, no obstacles, a tree of four nodes before one RRT* iteration: the root $A = (0, 0)$ with Cost = 0; $B = (3, 0)$ under A, Cost = 3; $C = (3, 3)$ under B, Cost = 6; $D = (0, 3)$ under A, Cost = 3. The new sample is $E = (1.5, 1.5)$ and the neighborhood radius for this step is $r = 2.5$.

Near set. $\text{dist}(E, \cdot) = 1.5\sqrt{2} = 2.1213$ to each of A, B, C, D: all four are in Q_{near}.

Choose-parent. Via A: $0 + 2.1213 = 2.1213$. Via B or D: $3 + 2.1213 = 5.1213$. Via C: $6 + 2.1213 = 8.1213$. Parent A; $\text{Cost}(E) = 2.1213$.

Rewire. The cost through E to any neighbor is $2.1213 + 2.1213 = 4.2426 = 3\sqrt{2}$. B: $4.2426 > 3$, keep. D: keep. C: $4.2426 < 6$ — **rewire** $\text{par}(C) := E$ and $\text{Cost}(C)$ drops from 6 to 4.2426, a 29.3 percent saving; C has no children, so the propagation is a no-op. Plain RRT would have attached E under its nearest node — a tie here, broken toward A — and left C at 6 forever.

13.5 Implementation in Rust

`sampling::optimal` wraps Chapter 12's tree; it does not fork it. Nothing from Chapters 11 or 12 is redefined, and the radius functions are plain, so that a widget can violate the bound on purpose.

RUST **crates/sampling/src/optimal/radius.rs**

```rust
/// r_n = gamma (log n / n)^{1/d}. gamma is a parameter, not hidden: the widget must be able to break it.
pub fn r_n(n: usize, d: usize, gamma: f64) -> f64 {
    gamma * ((n as f64).ln() / n as f64).powf(1.0 / d as f64)
}
/// gamma* = 2 (1 + 1/d)^{1/d} (mu_free / zeta_d)^{1/d} -- Karaman & Frazzoli 2011, Theorem 38.
pub fn gamma_star_kf2011(d: usize, mu_free: f64) -> f64 {
    2.0 * (1.0 + 1.0 / d as f64).powf(1.0 / d as f64) * (mu_free / unit_ball_volume(d)).powf(1.0 / d as f64)
}
/// gamma* = (2 (1 + 1/d))^{1/d} (mu_free / zeta_d)^{1/d} -- Solovey, Janson, Schmerling, Frazzoli & Pavone 2020.
pub fn gamma_star_solovey2020(d: usize, mu_free: f64) -> f64 {
    (2.0 * (1.0 + 1.0 / d as f64)).powf(1.0 / d as f64) * (mu_free / unit_ball_volume(d)).powf(1.0 / d as f64)
}
/// k_n = ?e (1 + 1/d) log n?.
pub fn k_n(n: usize, d: usize) -> usize { (std::f64::consts::E * (1.0 + 1.0 / d as f64) * (n as f64).ln()).ceil() as usize }

pub enum Neighborhood { Radius { gamma: f64, eta: f64 }, KNearest { k_star: f64 } }
```

RUST **crates/sampling/src/optimal/rrt_star.rs**

```rust
/// Ch. 12's tree plus cost-to-come and child lists. Wrapping, not forking: `inner` is the
/// same arena the Ch. 12 widgets draw, so RRT and RRT* can be fed identical samples.
pub struct CostTree<M: Chart> { pub inner: Tree<M>, pub cost: Vec<f64>, children: Vec<SmallVec<[NodeId; 4]>> }

impl<M: Chart> CostTree<M> {
    pub fn near(&self, q: &M::Point, r: f64) -> Vec<(NodeId, f64)> { self.inner.nn.within(q, r) }   // Ch. 11 KdTree::within
    pub fn add(&mut self, q: M::Point, parent: NodeId, edge: f64) -> NodeId {
        let id = self.inner.add(q, parent);
        self.cost.push(self.cost[parent] + edge);
        self.children.push(SmallVec::new());
        self.children[parent].push(id);
        id
    }
    /// Change v's parent and push the cost change through its subtree (iterative DFS).
    pub fn reparent(&mut self, v: NodeId, new_parent: NodeId, edge: f64) -> usize {
        let old = self.inner.nodes[v].parent.unwrap();
        self.children[old].retain(|&c| c != v);
        self.inner.nodes[v].parent = Some(new_parent);
        self.children[new_parent].push(v);
        let delta = self.cost[new_parent] + edge - self.cost[v];
        let (mut stack, mut touched) = (vec![v], 0);
        while let Some(u) = stack.pop() { self.cost[u] += delta; touched += 1; stack.extend(self.children[u].iter().copied()); }
        touched - 1
    }
}

impl<M: Chart, S: Extend<M>, Smp: Sampler<M>> RrtStar<M, S, Smp> {
    /// (parent, edge cost) minimizing Cost(v) + c(v, q_new) over `near` with Delta succeeding.
    /// Sorted by that sum, so the first candidate whose segment is free is the answer.
    fn choose_parent(&self, free: &dyn FreeSpace<M>, q_new: &M::Point, near: &[(NodeId, f64)]) -> Option<(NodeId, f64)> {
        let mut cands: Vec<_> = near.iter().map(|&(v, d)| (self.tree.cost[v] + d, v, d)).collect();
        cands.sort_by(|a, b| a.0.partial_cmp(&b.0).unwrap());
        cands.into_iter().find(|&(_, v, _)| self.steer.steer(&self.space, free, &self.tree.q(v), q_new).is_some())
             .map(|(_, v, d)| (v, d))
    }

    /// For each v in `near`: if Cost(new) + c(new, v) < Cost(v) and Delta(new, v), reparent.
    fn rewire(&mut self, free: &dyn FreeSpace<M>, new: NodeId, near: &[(NodeId, f64)]) -> usize {
        let c_new = self.tree.cost[new];
        let mut count = 0;
        for &(v, d) in near {
            if v == self.tree.parent(new) { continue; }
            if c_new + d < self.tree.cost[v] && self.steer.steer(&self.space, free, &self.tree.q(new), &self.tree.q(v)).is_some() {
                self.propagated += self.tree.reparent(v, new, d);
                count += 1;
            }
        }
        count
    }

```

```
    /// One iteration: sample -> nearest -> extend -> near set -> choose_parent -> rewire -> goal test.
    pub fn iterate(&mut self, free: &dyn FreeSpace<M>) -> Option<f64> {
        let q_rand = self.sample(free)?;
        let (near_id, _) = self.tree.inner.nearest(&q_rand);
        let q_new = match self.steer.extend(&self.space, free, &self.tree.q(near_id), &q_rand, self.eta) {
            ExtendStatus::Trapped => return None, ExtendStatus::Reached(q) | ExtendStatus::Advanced(q) => q };
        let n = self.tree.size() + 1;
        let mut near = match self.nbhd {
            Neighborhood::Radius { gamma, eta } => self.tree.near(&q_new, r_n(n, M::D, gamma).min(eta)),
            Neighborhood::KNearest { k_star } => self.tree.inner.nn.k_nearest(&q_new, k_n(n, M::D)),
        };
        if !near.iter().any(|&(v, _)| v == near_id) { near.push((near_id, self.space.dist(&self.tree.q(near_id), &q_new))); }
        // The nearest node's segment is already known free: it is always a valid parent.
        let (parent, edge) = self.choose_parent(free, &q_new, &near).unwrap_or((near_id, self.space.dist(&self.tree.q(near_id), &q_new)));
        let id = self.tree.add(q_new.clone(), parent, edge);
        self.rewires += self.rewire(free, id, &near);
        if self.space.dist(&q_new, &self.goal) <= self.goal_radius && self.steer.steer(&self.space, free, &q_new, &self.goal).is_some() { self.goal_nodes.push(id); }
        self.refresh_best()                                  // Some(cost) only if the incumbent dropped
    }
}
```

PRM* is Chapter 11's `Prm` with the connection rule tied to n, and the anytime contract is a trait with three methods.

RUST **crates/sampling/src/optimal/prm_star.rs**

```
/// Ch. 11's PRM with the connection rule tied to n: a ball of radius r_n, or the k_n nearest.
/// Each new node connects under the *current* rule (the incremental form); the expected number
/// of edges is O(n log n), which is what makes the roadmap both connected and affordable.

pub struct PrmStar<M: Chart, S: Sampler<M>> { pub prm: Prm<M, S, KdTree<M>>, nbhd: Neighborhood }

impl<M: Chart, S: Sampler<M>> PrmStar<M, S> {
    pub fn add_sample(&mut self, free: &dyn FreeSpace<M>) -> Option<usize> {
        let n = self.prm.roadmap.node_count() + 1;
        self.prm.connect = match self.nbhd {
            Neighborhood::Radius { gamma, .. } => Connect::Radius { r: r_n(n, M::D, gamma) },
            Neighborhood::KNearest { .. } => Connect::KNearest { k: k_n(n, M::D) },
        };
        self.prm.add_sample(free)
    }
}
```

RUST **crates/sampling/src/optimal/anytime.rs**

```rust
pub struct Solution<P> { pub path: Vec<P>, pub cost: f64 }

/// "Deadline in, best-so-far out." One unit of work per `step`, `Some(cost)` when the incumbent
/// improved; `best()` is always available. Chapter 23's planning stack consumes exactly this.
pub trait Anytime<P> {
    fn step(&mut self) -> Option<f64>;
    fn best(&self) -> Option<Solution<P>>;
    fn run_until(&mut self, deadline: std::time::Instant) -> Option<Solution<P>> {
        while std::time::Instant::now() < deadline { self.step(); }
        self.best()
    }
}
```

FMT* is a batch and a heap; BIT* is the same heap with a heuristic and a second queue.

RUST **crates/sampling/src/optimal/fmt.rs**

```rust
impl<M: Chart, S: Steer<M>> Anytime<M::Point> for Fmt<M, S> {
    /// Expand one frontier node -- lazy dynamic programming, one collision check per accepted edge.
    fn step(&mut self) -> Option<f64> {
        let z = self.open.pop()?;                                   // least cost-to-come
        for (x, _) in self.nn.within(&self.samples[z], self.r) {
            if self.state[x] != Unvisited { continue; }
            // Locally optimal parent among OPEN nodes in x's ball, obstacles ignored ...
            let Some((y, c)) = self.nn.within(&self.samples[x], self.r).into_iter()
                .filter(|&(y, _)| self.state[y] == Open)
                .map(|(y, d)| (y, self.cost[y] + d))
                .min_by(|a, b| a.1.partial_cmp(&b.1).unwrap()) else { continue };
            // ... then check only that edge. Wrong exactly when the best parent is blocked.
            if self.steer.steer(&self.space, &*self.free, &self.samples[y], &self.samples[x]).is_some() {
                self.cost[x] = c; self.parent[x] = Some(y); self.state[x] = Open; self.open.push(x, c);
            }
        }
        self.state[z] = Closed;
        (z == self.goal_idx).then(|| { self.done = true; self.cost[z] })
    }
    fn best(&self) -> Option<Solution<M::Point>> { self.done.then(|| self.path_to(self.goal_idx)) }
}
```

RUST **crates/sampling/src/optimal/bit.rs**

```rust
impl<M: Chart, S: Steer<M>> Anytime<M::Point> for Bit<M, S> {
    /// One unit of work: pop the best edge (expanding vertices while a vertex could beat it), or
```

```
    /// open a new batch when both queues are empty. Keys are Ch. 6's f = g + h on the implicit RGG.
    fn step(&mut self) -> Option<f64> {
        if self.q_v.is_empty() && self.q_e.is_empty() { self.new_batch(); return None; }
        // prune, sample, re-key
        while self.q_v.peek_key() <= self.q_e.peek_key() { let v = self.q_v.pop().unwrap(); self.expand(v); }
        let (v, x) = self.q_e.pop()?;
        let c = self.space.dist(&self.pts[v], &self.pts[x]);
        if self.g[v] + c + self.h_hat(x) >= self.c_best { self.q_v.clear(); self.q_e.clear(); return None; }
        if self.g[v] + c >= self.g[x] { return None; }
        self.edge_checks += 1;
        self.steer.steer(&self.space, &*self.free, &self.pts[v], &self.pts[x])?;    // lazy: checked on pop
        self.attach(x, v, c);                                                        // or rewire + propagate
        (self.in_tree[GOAL] && self.g[GOAL] < self.c_best).then(|| { self.c_best = self.g[GOAL]; self.c_best })
    }
    fn best(&self) -> Option<Solution<M::Point>> { self.c_best.is_finite().then(|| self.path_to(GOAL)) }
}
```

The informed sampler is a Chapter 11 `Sampler`, so Informed RRT* is RRT* with a different sampler and no planner code changed.

RUST **crates/sampling/src/optimal/informed.rs**

```
pub struct InformedSampler<M: Chart> {
    start: M::Point, goal: M::Point, c_min: f64, pub c_best: f64,
    center: DVector<f64>, axis: DVector<f64>, fallback: Uniform,
}

impl<M: Chart> Sampler<M> for InformedSampler<M> {
    fn sample(&mut self, space: &M, free: &dyn FreeSpace<M>, rng: &mut SmallRng) -> Option<M::Point> {
        if !self.c_best.is_finite() { return self.fallback.sample(space, free, rng); }
    // no incumbent yet
        let a = self.c_best / 2.0;
        let b = (self.c_best.powi(2) - self.c_min.powi(2)).max(0.0).sqrt() / 2.0;
        // Unit ball -> scale -> rotate (Householder: e_1 ? axis) -> translate -> reject outside ?.
        let u = unit_ball(rng);
        let s = SVector::from_fn(|i, _| if i == 0 { a * u[i] } else { b * u[i] });
        let v = (SVector::ith(0, 1.0) - self.axis).normalize();
        let r = s - 2.0 * v.dot(&s) * v;
        let c = r + self.center;
        let q = space.from_coords(&c)?;                        // None if outside the chart's box
        free.is_free(&q).then_some(q)
    }
}

impl<M: Chart> InformedSampler<M> {
    pub fn update(&mut self, c_best: f64) { self.c_best = c_best; }
    /// mu(X_f ? ?)/mu(?) -- w13.2's readout; in the plane the ellipse is clipped against the box by quadrature.
```

```
    pub fn measure_ratio(&self, space: &M) -> f64 { ellipse_box_area(self, space.bounds()) / space.measure() }
}
```

The worked example is an executable whose output is the test's expectation.

RUST **crates/sampling/examples/rewire_five_nodes.rs**

```
fn main() {
    let plane = R2::unit_box(20.0).centered();                                  // no obstacles
    let mut star = RrtStar::new(plane, [0.0, 0.0], StraightLine::subdivision(0.25), Uniform,
                                Neighborhood::Radius { gamma: 1.0, eta: 10.0 }, SmallRng::seed_from_u64(0));
    let b = star.tree.add([3.0, 0.0], 0, 3.0);
    let c = star.tree.add([3.0, 3.0], b, 3.0);
    let _d = star.tree.add([0.0, 3.0], 0, 3.0);
    let e = [1.5, 1.5];
    let near = star.tree.near(&e, 2.5);
    for &(v, d) in &near { println!("{}: dist {d:.4}, via {:.4}", NAMES[v], star.tree.cost[v] + d); }
    let (parent, edge) = star.choose_parent(&Free, &e, &near).unwrap();
    let id = star.tree.add(e, parent, edge);
    println!("parent {}, Cost(E) = {:.4}", NAMES[parent], star.tree.cost[id]);
    let before = star.tree.cost[c];
    star.rewire(&Free, id, &near);
    println!("C: {before:.4} -> {:.4}, parent now {}", star.tree.cost[c], NAMES[star.tree.parent(c)]);
    for (name, g) in [("2011", gamma_star_kf2011(2, 1.0)), ("2020", gamma_star_solovey2020(2, 1.0))] {
        println!("gamma*_{name} = {g:.4}, r_1000 = {:.4}", r_n(1000, 2, g));
    }
    println!("k_1000 = {}", k_n(1000, 2));
}
```

```
A: dist 2.1213, via 2.1213
B: dist 2.1213, via 5.1213
C: dist 2.1213, via 8.1213
D: dist 2.1213, via 5.1213
parent A, Cost(E) = 2.1213
B keep, D keep, C rewire
C: 6.0000 -> 4.2426, parent now E
gamma*_2011 = 1.3820, r_1000 = 0.1149
gamma*_2020 = 0.9772, r_1000 = 0.0812
k_1000 = 29
```

RUST **crates/sampling/tests/optimal.rs**

```
#[test]
fn reproduces_five_node_rewire() {
    let r = five_node_rewire();
    assert_eq!(r.near.len(), 4);
    assert!((r.cost_e - 2.1213).abs() < 1e-4);
    assert_eq!(r.parent_c, r.e);                  // rewired under E
```

```
    assert!((r.cost_c - 4.2426).abs() < 1e-4 && (r.cost_b - 3.0).abs() < 1e-9);
}

#[test]
fn radius_constants() {
    assert!((r_n(1000, 2, gamma_star_kf2011(2, 1.0)) - 0.1149).abs() < 5e-5);
    assert!((r_n(1000, 2, gamma_star_solovey2020(2, 1.0)) - 0.0812).abs() < 5e-5);
    assert_eq!(k_n(1000, 2), 29);
}

/// Seed 13 on the Apartment, 6 000 iterations: RRT* non-increasing and within 5% of the
   lattice
/// floor; RRT on the same samples never rewires and ends well above it.
#[test]
fn rrt_star_cost_converges_under_fixed_seed() {
    let (plain, star, floor) = seed_13_scoreboard(6000);
    assert!(star.cost_history.windows(2).all(|w| w[1].cost <= w[0].cost + 1e-9));
    assert!(star.best_cost < floor * 1.05);
    assert!(plain.best_cost > star.best_cost * 1.2);
}
```

The TypeScript port's checks replay all of it: the five-node rewire and the radii exactly, and the seed-13 scoreboard — RRT at 13.680 m, RRT* at 10.885 m against a lattice floor of 11.508 m after six thousand iterations, the same node set in both trees, and every returned path re-checked at five millimetres.

13.6 Putting it together: the anytime contract

The lab is the Batch Bench. Four asymptotically optimal planners, one seed, one world, and one deadline each: stop at the deadline, execute what you have, keep planning, and switch to the better path when it arrives. That loop is the one Chapter 23's planning stack runs for Reach, and it is what `trait Anytime` is for — one unit of work per `step`, `Some(cost)` when the incumbent improved, `best()` always available.

> **INTERACTIVE · w13.3**
>
> **Batch Bench**
>
> Asymptotically optimal does not mean better at every budget: FMT *has nothing until its batch is processed*, BIT*'s first batch often beats RRT* early, and the curves cross — the deadline decides the winner.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch13-optimal-sampling#w13.3

Read the curves at three deadlines. At a hundred milliseconds, RRT* and Informed RRT* have usually found a first path and begun to improve it; BIT* has processed its first batch and often holds the best path of the four; FMT* has nothing — it is still drawing its batch or marching through it. At half a second

the curves have crossed at least once. At two seconds FMT* has returned its single answer, often the best, and will never improve it; the incremental planners are still descending. The misconception the lab kills is *"asymptotically optimal means better at every budget."* The theorems are all about $n \to \infty$; at a deadline the winner is decided by constants, and the constants depend on the world.

Three honesty items the chapter keeps. Every theorem here is asymptotic — at finite n the cost is a random variable, and the widgets show its spread across seeds. The hypotheses are necessary: robust optimality, `dist` as the cost, and the γ bound, which Rewire Watch lets you violate. And nothing in this chapter makes Hitch drivable: the straight-line steer is still wrong for a car, and the cost it optimizes is the length of a path the car cannot follow. Kinodynamic RRT* replaces the steer with a two-point boundary-value solver in Chapter 21.

One last contrast before Part III closes. Asymptotic optimality is *global* and *slow*: the planner converges to the best path in any homotopy class, eventually. Chapter 19's trajectory optimization is the complementary route — *local* and *fast*, descending a cost from an initial guess and never leaving its homotopy class. The two meet in practice as a pipeline: an RRT* path to pick the class, an optimizer to make it smooth.

13.7 Exercises

1 FOUNDATION • • • **Where (1 + 1/d) comes from**

Carry out Step 4 of the optimality derivation with the constants kept. Tile a path of length L with balls of radius $r_n/4$ whose centers are $r_n/4$ apart; count them; write the probability that a given ball is empty after n uniform samples in $\mathcal{Q}_{\text{free}}$; apply the union bound; and find the smallest exponent of γ for which $\sum_n \Pr[\text{some ball empty at } n]$ converges. Identify the step at which $(1 + 1/d)$ enters.

HINTS

<>The number of balls is about 4L / r_n ∝ (n / log n)^(1/d). The emptiness probability is at most n^(−cγ^d) with c = ζ_d / (4^d μ(Q_free)).</> <>Σ_n n^(1/d − cγ^d) (log n)^(−1/d) converges iff cγ^d − 1/d > 1.</> <>So cγ^d > 1 + 1/d, i.e. γ^d > (1 + 1/d) · 4^d μ(Q_free) / ζ_d. The 4^d is the geometric slack of this particular tiling; a tighter tiling gives the paper's 2^d.</>

WORK IT OUT

With d = 2 and μ(Q_free)/ζ_d = $1/\pi$, what is the threshold on γ^2 from this tiling (balls of radius r_n/4)?

Answer. 7.64 $\gamma^2 > (1 + 1/2) \cdot 16/\pi = 24/\pi = 7.64$, so $\gamma > 2.76$ — looser than the paper's $\gamma^*_{2011} = 1.38$ by exactly the factor 2 of the tiling radius. The dependence on $(1 + 1/d)$, $\mu(Q_{\text{free}})$ and ζ_d is the same.

2 **FOUNDATION** •• **The informed set on the torus**

Show that on T^2 the planar ellipse formula undercounts $\mu(X_{\hat{f}})$ once c_{best} exceeds the torus circumference 2π: the geodesic from a focus to a point may go through the seam, so points the planar formula excludes satisfy $\hat{g} + \hat{h} \leq c_{best}$. Derive the correct sampler — a union of ellipses over the seam images of the foci, or rejection on the unwrapped cover — and say which the implementation uses and why.

3 **CONCEPTUAL** • **Predict the half-radius, then verify**

PREDICT FIRST

In Rewire Watch on the Apartment at $\gamma/\gamma = 1$ *and seed 13, RRT* ends within a few percent of the lattice floor by 6 000 iterations. Before touching the slider: what does $\gamma/\gamma^* = 0.5$ do to its cost curve?

- Nothing — the ball is still large enough for choose-parent and rewire to work

  Early on, yes: r_n is large for small n at any γ. But the ball shrinks as n grows, and at half the critical γ it shrinks faster than samples arrive to fill it. The curve stalls above the floor rather than converging.

- ✓ A slower descent that stalls above the floor

  Right. Rewiring still happens while the ball is large, so the early descent survives, but the expected size of Q_near falls like $\gamma^2 \log n$ — a quarter of the critical value — and eventually a new node finds no neighbor but q_near. The tree reverts to RRT behavior and the curve flattens short of the optimum.

- A flat curve from the start, identical to RRT

  Too strong. For n in the hundreds, r_n at $\gamma/\gamma^* = 0.5$ is still a few multiples of the inter-node spacing, and the first rewires do happen. The failure is asymptotic, which is exactly the theorem's point.

Verify, then read the $|Q_{near}|$ histogram at both settings. Its mean scales with $\gamma^d \log n$: at the critical γ it hovers around a few times $\log n$; at half, around a quarter of that, and the fraction of insertions with $|Q_{near}| = 1$ — no choice to make — climbs as n grows.

4 CONCEPTUAL •• **Where BIT *and RRT* cross**

In Batch Bench on the Workbench, find the deadline at which BIT*'s and RRT*'s costs are equal on a fixed seed, then vary the batch size and find the one that moves the crossing earliest. Relate it to how many batches fit before the deadline: a batch too small is searched before it has enough samples to improve the incumbent, a batch too large is still being searched when the clock stops.

5 PRACTICAL •• **The k-nearest neighborhood**

Implement `Neighborhood::KNearest` with $k_n = \lceil k^* \log n \rceil$, $k^* = e(1 + 1/d)$, and a test that the radius and k-nearest variants reach costs within 3 percent of each other after 6 000 iterations under seed 13 on the Apartment. The TypeScript check `k-nearest RRT*` asks for 10 percent; tighten it and report how many seeds survive.

6 PRACTICAL • • • **Shortcutting cannot win the homotopy class**

Run Chapter 11's `shortcut_greedy` on RRT's path and plot, over 100 seeds, its cost against RRT*'s at equal wall-clock time on the Apartment. Then build the two-homotopy-class map where it cannot win: a single obstacle between start and goal, with the shorter way round narrower than the longer. Show that RRT commits to the wide side on a fixed fraction of seeds and that no amount of shortcutting moves it to the narrow one, while RRT*'s incumbent eventually does.

13.8 References

Karaman, S. and Frazzoli, E. (2011). Sampling-based Algorithms for Optimal Motion Planning. *International Journal of Robotics Research 30(7), 846–894*.
RRT's almost-sure suboptimality, PRM, *RRG and RRT*, and the radius r_n = γ(log n/n)^{1/d} with the 2011 constant this chapter calls γ^*_{2011}. doi:10.1177/0278364911406761

Solovey, K., Janson, L., Schmerling, E., Frazzoli, E., and Pavone, M. (2020). Revisiting the Asymptotic Optimality of RRT*. *IEEE International Conference on Robotics and Automation*.
The gap in the 2011 RRT *proof and its repair, with the smaller constant* γ_{2020} the implementation defaults to. link

Gammell, J. D., Srinivasa, S. S., and Barfoot, T. D. (2014). Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic. *IEEE/RSJ International Conference on Intelligent Robots and Systems*.
The informed set, its direct sampling by scaling and rotating the unit ball, and the measure ratio w13.2 reads out. doi:10.1109/IROS.2014.6942976

Gammell, J. D., Barfoot, T. D., and Srinivasa, S. S. (2020). Batch Informed Trees (BIT*): Informed Asymptotically Optimal Anytime Search. *International Journal of Robotics Research 39(5)*.
BIT* in its journal form: batches on an implicit random geometric graph, the edge queue keyed $\hat{g} + \hat{c} + \hat{h}$, pruning and tree reuse. The ICRA 2015 paper introduced it. link

Janson, L., Schmerling, E., Clark, A., and Pavone, M. (2015). Fast Marching Tree: a Fast Marching Sampling-Based Method for Optimal Motion Planning in Many Dimensions. *International Journal of Robotics Research 34(7), 883–921*.
FMT*: lazy dynamic programming on one batch, its radius constant, and the O(n log n) collision-check bound. doi:10.1177/0278364915577958

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

The scaffolding: Theorem 7.4.1's ball tiling and the (ε, α, β) constants reappear inside the optimality proofs; §7.1.2's shortcutting is the pre-2011 answer to path quality, and the epigraph's 'large dense roadmap' is the intuition PRM* turns into a theorem. link

Şucan, I. A., Moll, M., and Kavraki, L. E. (2012). The Open Motion Planning Library. *IEEE Robotics and Automation Magazine 19(4), 72–82*.

Reference implementations of RRT, *PRM*, Informed RRT, *FMT* and BIT*, against which this chapter's constants and defaults were compared. doi:10.1109/MRA.2012.2205651

CHAPTER 14

Planning for Many Bodies: Multi-Robot, Manipulation, and Kinodynamic Previews

SAMPLING-BASED PLANNING · Advanced · 85 min

> *Decoupled planning does not increase the dimensionality of the configuration space. It is incomplete, however, even when the algorithms used in both of its stages are complete.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), §7.5.2*

Three chapters of sampling-based planning assumed one robot, one query, and one rigid notion of "free." This chapter removes those assumptions one at a time and finds that the machinery does not care. Two robots' configuration space is the product of theirs. A manipulated object is one more body, sometimes free and sometimes rigidly tied to a hand. A car that cannot be steered along a straight line can still grow a tree by *simulating* instead of steering. Every planner from Chapter 11, 12 (Chapter 12) and 13 (Chapter 13) runs unchanged on a composite configuration space.

What changes is the bill. The dimension of a composite space is the sum of the dimensions, and the sample count of Theorem 7.4.1 grows like $(1/\rho)^d$; the number of pairwise collision checks grows like m^2. Planning for many bodies is therefore mostly about *when to refuse to pay*: decouple the robots and coordinate their velocities, give them priorities, decompose the motion into modes, or — the modern answer — search over conflicts instead of over the joint space. Each of these refusals costs completeness or optimality somewhere, and this chapter is honest about where.

Choset's §7.5 surveys these extensions in ten pages. Here multi-robot and

manipulation planning get full treatment with running code; conflict-based search (2015) and the task-and-motion view (2021) are added because they changed practice; forward propagation gets exactly one page as Hitch's first planner — Chapter 21 owns the rest; and OMPL and MoveIt are named as the systems in which all of Part III is infrastructure. The sister book's Chapter 23 makes the same "roll the simulator forward" move as this chapter's preview, in a different costume.

AFTER THIS CHAPTER YOU CAN

- Build the composite configuration space of several bodies as a product manifold, and the composite free space as the lifted individual free spaces minus every pairwise robot–robot C-obstacle
- Run Chapter 11–13's planners on the product unchanged, and account for the price: dimension adds, Theorem 7.4.1's sample count multiplies, and the pairwise checks grow like m^2
- State decoupled, prioritized and centralized multi-robot planning, and construct the swap instance on which the first two fail for every priority order while the third succeeds
- Prove conflict-based search complete and optimal for sum-of-costs on a grid, trace it on the swap instance, and compare its constraint tree with a coupled A* over the joint state
- Model manipulation as motion on a union of transit and transfer manifolds, build the manipulation graph of grasps at placements, and plan on it with a lazy two-level PRM
- Preview planning with a simulator instead of a steer: a forward-propagated RRT for Hitch whose edges are drivable by construction — and say precisely what Chapter 21 adds

ASSUMED

- Chapter 2's Collision contract (a robot–robot query is the same geometric primitive as a robot–obstacle one), Hitch's kinematic update as a black box, Reach's forward kinematics and closed-form IK
- Chapter 4: a disc–disc C-obstacle is a disc of radius $r_1 + r_2$
- Chapter 5: Manifold and Chart; the Product construction and its ℓ^2 metric
- Chapter 6: A* on a grid, with its tie-breaks — CBS's low level and the coordination search
- Chapter 11: FreeSpace, Steer, PRM, Theorem 7.4.1's (1/ρ)^d and Theorem 7.4.3's asymmetric R; Chapter 12: Tree, extend, RRT and RRT-Connect

14.1 The problem: two free paths are not a free plan

Two Rustys in the Apartment. Rusty 1 wants to go from room A to room C; Rusty 2 wants the reverse. Each is handed to the single-robot planner of Chapter 6 — A* on the inflated lattice — and each gets back a path that touches no wall. Execute both at once.

> INTERACTIVE · w14.0
>
> **Two Rustys collide**
>
> Two individually collision-free paths are not a collision-free plan: the second body is an obstacle that moves.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch14-many-bodies#w14.0

They meet head-on in the corridor. Nothing in either planner was wrong; nothing in either planner could have been right. Each one checked its robot against the walls and never against the other robot, because the other robot was not a configuration it ever saw. The collision lives in a space neither planner searched: the set of *pairs* of configurations, where the second body is an obstacle that moves.

That space is the product $Q = Q_1 \times Q_2$, four-dimensional for two discs, and its obstacle has two parts. The lifted walls, which each robot already knew about, and a new set — the configurations where the two bodies overlap — which neither did. The hook, in numbers: at the composite configuration $q = (1.0, 1.0, 1.8, 1.4)$ in a corridor of height 2, each disc of radius 0.5 is 0.5 clear of both walls, and their centers are $\sqrt{0.8^2 + 0.4^2} = 0.8944$ apart. Each robot's own check passes. The composite check does not: $0.8944 < 1.0$, a penetration of 0.1056.

14.2 Building intuition: one space, many bodies

14.2.1 The amber hole that moves

> INTERACTIVE · w14.1
>
> **Two-Robot Composite C-space**
>
> "Plan each robot separately and add a wait" fails on the swap problem: no wait works, and the alcove detour is only expressible in the composite space.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch14-many-bodies#w14.1

The top panel is two discs of radius 0.5 in a corridor with one alcove, swapping ends. The bottom-left panel is robot 2's slice of the four-dimensional composite

space: the two coordinates of robot 2, with robot 1 frozen where it is. Slate is where robot 2 alone would hit a wall — the Chapter 4 picture, a disc robot inflated against the corridor. Amber is new: the disc of radius $r_1 + r_2 = 1$ around robot 1, the robot–robot C-obstacle QO_{12} cut at robot 1's current position. Drag robot 1 and the amber hole follows. That is the whole difference between one robot and two: a C-obstacle that is a *function of the other robot's configuration*.

At the default width of 3.2 radii the amber disc spans the corridor's full height. Below four radii of width the two discs cannot pass each other anywhere in the corridor, so whichever robot is in the corridor blocks it entirely, and the only resolution is the alcove: someone steps aside, the other passes, the first steps back out. Try the four planner chips.

Decoupled. Plan each robot alone, then coordinate speeds along the two fixed paths. The bottom-right panel is the coordination diagram $[0, 1]^2$ of path parameters: amber where the two bodies, each at its own parameter, overlap. A velocity schedule is a monotone path from $(0, 0)$ to $(1, 1)$ — neither robot drives backward along its path. Both paths planned alone are the corridor axis, so the obstacle is a band along the anti-diagonal $s_1 + s_2 \approx 1$, covering 39.0 percent of the square, and no monotone path crosses it. No wait of any length fixes this.

Prioritized 1→2 and 2→1. Robot 1 plans alone in configuration–time space; robot 2 plans with robot 1 as a moving obstacle. Robot 1's trajectory is the axis, the alcove is of no use to a robot planning alone, and robot 2 finds no trajectory within its budget. Reverse the order and the same thing happens with the roles swapped. The prioritized planner fails in *both* orders.

Coupled. RRT-Connect from Chapter 12 on the product $(\mathbb{R}^2)^2$, with a free-space predicate that checks each disc against the walls and then the pair against each other. It finds the detour: one robot into the alcove, the other past, the first out. On seed 14 the trees merge at attempt 2562 with 60 nodes between them, after 3278 composite checks of which 199 died on the pair term alone. The sample count is the price of four dimensions; the pair term is the price of two bodies. Both are reported, not hidden.

The misconception this widget exists to kill: *"plan each robot separately and add a wait."* The swap problem has no wait that works. "Robot 1 waits in the alcove" is a different *path* for robot 1, and only the composite space can express it.

14.2.2 Modes, not a bigger robot

INTERACTIVE · w14.2

Pick-and-Place Modes

Manipulation planning is not path planning with a bigger robot: motion lives in a union of lower-dimensional manifolds, and switching happens only at their intersections — the grasps.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/

```
ch14-many-bodies#w14.2
```

Reach moves a puck from placement A to placement B on the Workbench. The puck is slate when it rests on the table and orange when the arm holds it, and the two colors are two different robots. With the puck at rest it is an obstacle, and the arm moves on a *transit* manifold: the amber region on the left torus is Reach's C-obstacle with one more body in the scene. With the puck held it is part of the robot, rigidly attached to the hand, and the arm moves on a *transfer* manifold: the amber region on the right torus is a different shape, because the robot has a different shape.

Both manifolds have the dimension of the arm alone — two — inside a composite space of dimension four. The system can leave one only at a configuration that lies on both: the arm holding the puck *at* a placement, which is a grasp. The graph on the right has a node for every feasible (placement, grasp) pair and an edge for every transit (same placement, different grasp: a regrasp) or transfer (same grasp, different placement). A manipulation plan is a path in this graph.

The headline slider adds grasps. With the front grasp alone, placement B — behind the bar — admits no grasp configuration at all, and the graph is disconnected: regrasping is impossible because there is only one grasp. With two grasps the graph is still disconnected if there is nowhere to set the puck down in between; add placement C and the plan is transit, transfer, transit, transfer — carry the puck from A to C with the front grasp, release, regrasp from the side, carry it to B. Grey dashed edges were never planned for real; only the four on the chosen path were, out of seven. That is the lazy two-level PRM of §7.5.3, and the counter is the number Exercise 4 asks about.

The misconception killed: *"manipulation planning is path planning with a bigger robot."* It is a union of lower-dimensional manifolds, and switching happens only at their intersections.

14.2.3 Conflicts, not the joint space

INTERACTIVE · w14.4

Constraint Tree

Optimal multi-agent paths do not require searching the joint space: CBS searches a tree of constraints whose size tracks the number of conflicts, not 5 to the number of agents.

Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch14-many-bodies#w14.4`

A room with two pillars, three Rustys crossing from the west wall to the east wall in reversed order. The root of the tree plans every agent alone with Chapter 6's A* over (cell, time). Expanding a node means finding the first conflict in its

paths — two agents in one cell at one time, or swapping an arc — and spawning two children, each forbidding that cell to one of the two agents at that time, each re-planning only that agent. Nodes are popped in order of the sum of individual costs, so the first conflict-free node popped has the smallest sum any conflict-free plan can have.

Watch the two numbers in the readout. On the three-agent room the constraint tree has three nodes and two expansions; a coupled A* over the joint state, which must consider up to $5^3 = 125$ joint moves per expansion, generates 1412 successors to find the same cost. Slide to five agents: the tree grows to eleven nodes, while the joint search's branching factor is 3125 and it is not run. The tree's size tracks *conflicts*, which in an apartment are few; the joint search's size tracks *agents*, exponentially.

The misconception killed: *"optimal multi-agent paths require searching the joint space."*

> **THE ONE SENTENCE VERSION**
>
> Every planner in Part III runs unchanged on a composite configuration space — only the price changes, and it is exponential in the number of bodies, so many-body planning is about when to refuse to pay it: decoupling, prioritization, modes, and conflict-based search.

14.3 The mathematics

$\mathcal{Q} = \mathcal{Q}_1 \times \cdots \times \mathcal{Q}_m$, $q = (q_1, \ldots, q_m)$	composite configuration space of m bodies; dim $\mathcal{Q} = \Sigma_i$ dim $\mathcal{Q}_i$
$\mathcal{QO}_i^{\mathcal{W}}$, $\mathcal{QO}_{ij}$	robot–obstacle C-obstacle of body i, lifted to $\mathcal{Q}$; robot–robot C-obstacle $\{q : R_i(q_i) \cap R_j(q_j) \neq \emptyset\}$
$c_i : [0,1] \to \mathcal{Q}_{free,i}$, $(s_1, \ldots, s_m) \in [0,1]^m$	individual paths; the coordination space of path parameters
$\mathcal{Q}_i \times [0,T]$	configuration–time space for prioritized planning
(a_i, a_j, v, t), (a_i, v, t), N, SIC(N)	CBS conflict; constraint; constraint-tree node; its sum of individual costs
$\mathcal{G}$, P, $\mathcal{M}_{transit}(p)$, $\mathcal{M}_{transfer}(g)$	grasp set; stable placements; transit manifold (object fixed at p); transfer manifold (object rigidly attached by g)
MG	manipulation graph: nodes (p, g), edges transit/transfer paths
$f : \mathcal{Q} \times U \to \mathcal{Q}$, U, Δt	Choset's control function (state propagation); control set; propagation interval

14.3.1 Definitions

DEFINITIONS — COMPOSITE FREE SPACE, DECOUPLING, COORDINATION

Composite free space. With $\pi_i : \mathcal{Q} \to \mathcal{Q}_i$ the projections,

$$\mathcal{Q}_{\text{free}} = \bigcap_{i=1}^{m} \pi_i^{-1}(\mathcal{Q}_{free,i}) \setminus \bigcup_{i<j} \mathcal{QO}_{ij}, \qquad \mathcal{QO}_{ij} = \{\, q : R_i(q_i) \cap R_j(q_j) \neq \emptyset \,\}.$$

The individual free spaces, lifted, minus every pairwise interference.
Decoupled planning (Kant and Zucker, [C ref. 219]). Plan each c_i in $\mathcal{Q}_{free,i}$ alone; then choose velocities along the fixed paths so that no two robots collide. **Prioritized planning.** Plan the robots in a fixed order; each treats the robots before it as moving obstacles. **Centralized planning.** Plan once in $\mathcal{Q}$.
Coordination diagram. For two robots, the subset of $[0,1]^2$ where $R_1(c_1(s_1))$ and $R_2(c_2(s_2))$ overlap. A monotone free path from $\mathbf{0}$ to $\mathbf{1}$ is a valid velocity schedule; monotone because a robot does not drive backward along its own path.

DEFINITIONS — CONFLICT, CONSTRAINT, TRANSIT, TRANSFER, PROPAGATION

Conflict and constraint (CBS). A *vertex conflict* (a_i, a_j, v, t): agents a_i and a_j occupy cell v at time t. An *edge conflict*: they traverse one arc in opposite directions between $t-1$ and t. A *constraint* (a_i, v, t) forbids one agent one cell (or one arc) at one time.
Transit and transfer paths (§7.5.3). A transit path is robot motion with the object at rest at a stable placement p; a transfer path is motion with the object grasped by g. Modes switch only at $\mathcal{M}_{transit}(p) \cap \mathcal{M}_{transfer}(g)$: a grasp configuration at a placement.
Forward propagation. Producing $q' = f(q,u)$ by applying a control u for Δt, instead of steering to a chosen q'. The relation $q\,R\,q'$ — "some u drives q to q', collision-free" — is asymmetric and irreflexive.

14.3.2 Structure and price of the composite C-obstacle

THEOREM (THE PAIRWISE C-OBSTACLE OF TWO DISCS)

For discs of radii r_i, r_j with centers p_i, p_j, $\mathcal{QO}_{ij} = \{q : \|p_i - p_j\| < r_i + r_j\}$ is a cylinder over a ball in the difference coordinate $p_i - p_j$: convex in that coordinate, of full dimension in $\mathcal{Q}$. In Theorem 7.4.1 with $d = \sum_i d_i$, the

number of samples needed to tile a path of clearance ρ grows like $(1/\rho)^d$.

DERIVATION **Why two bodies cost a dimension, not a check**

$\mathcal{QO}_{ij}$ is a cylinder; $\sigma\rho^d$ shrinks geometrically in d

Step 1 — Minkowski with both bodies moving. Chapter 4 showed that a disc robot of radius r collides with a point obstacle iff its center is within r. Replace the point by a second disc of radius r_j: $R_i(q_i) \cap R_j(q_j) \neq \emptyset$ iff some point is within r_i of p_i and within r_j of p_j, iff $\|p_i - p_j\| \leq r_i + r_j$. The closed-set convention of Chapter 2 is kept: touching counts.

Step 2 — the cylinder. The condition depends on q only through $p_i - p_j$. In coordinates $(p_i - p_j,\ p_i + p_j,\ \text{everything else})$, $\mathcal{QO}_{ij}$ is a ball of radius $r_i + r_j$ in the first factor times the whole of the others. It is convex in the difference coordinate and has full dimension d in $\mathcal{Q}$ — a four-dimensional region for two discs, not a surface.

Step 3 — the lifted walls. $\pi_i^{-1}(\mathcal{QO}_i^{\mathcal{W}})$ is a cylinder in the complementary direction: robot i's Chapter 4 C-obstacle times the other robots' entire spaces. $\mathcal{Q}_{\text{free}}$ is the complement of the union of both kinds of cylinder. Neither kind alone is the obstacle; their union is.

Step 4 — the exponential price. Theorem 7.4.1 bounds PRM's failure probability by $\ell\,(1 - \sigma\rho^d)^n$ where $\sigma\rho^d$ is the measure of a ball of radius ρ relative to $\mu(\mathcal{Q}_{\text{free}})$ and $\ell \sim L/\rho$ balls tile the path. At fixed clearance $\rho < 1$, $\sigma\rho^d$ shrinks geometrically in d, so the samples needed for a fixed failure bound grow like $(1/\rho)^d$. For two Rustys $d = 4$; for three, 6. The constant did not change; the exponent did.

Why a coupled path has less clearance. In $\mathcal{Q}$ the clearance of a path is the minimum over all bodies and all pairs. A coupled path that threads two robots past each other has, at the passing moment, clearance about $\|p_1 - p_2\| - (r_1 + r_2)$, typically smaller than either robot's clearance to the walls. Smaller ρ, larger $(1/\rho)^d$: the two effects compound.

Polygons. For polygonal bodies, $\mathcal{QO}_{ij}$ at fixed orientations is the Minkowski sum $R_i \oplus (-R_j)$ in the difference coordinate, computable with Chapter 4's star algorithm; the cylinder structure is unchanged.

14.3.3 Decoupled and prioritized planning are incomplete

THEOREM (CHOSET §7.5.2, MADE CONCRETE)

There are instances with a coupled solution on which no velocity coordination of independently planned paths succeeds, and on which prioritized planning fails for every priority order.

DERIVATION **The swap instance**

\text{no monotone path in } [0,1]^2;\ \text{no order works}

Step 1 — the instance. A corridor narrower than $2(r_1 + r_2)$, with one alcove in its north wall, and two robots that must swap ends. On the grid: a corridor one cell wide, seven cells long, with one alcove cell above the middle; robot 1 from cell $(0, 0)$ to $(6, 0)$, robot 2 the reverse.
Step 2 — the coordination diagram. Each robot's individually shortest path is the corridor axis. Robot 1 at parameter s_1 is at $x = 0.8 + 4.4s_1$; robot 2 at $x = 5.2 - 4.4s_2$. They overlap when $|4.4(s_1 + s_2) - 4.4| < 1$, i.e. $|s_1 + s_2 - 1| < 0.227$: a band along the anti-diagonal from $(0, 1)$ to $(1, 0)$, which every monotone path from $(0, 0)$ to $(1, 1)$ must cross. The widget's raster finds the band occupies 39.0 percent of the square, every occupied cell within $|s_1 + s_2 - 1| < 0.25$, and A* over the three monotone moves fails.
Step 3 — prioritized, both orders. Plan robot 1 first: its path alone is the axis, and it occupies cell x at time x and then the far end forever. Robot 2 must traverse every corridor cell robot 1 traverses, in the opposite direction, and the alcove is reachable only from the middle cell, which robot 1 occupies at $t = 3$ — exactly when robot 2, having left at $t = 0$, would need it. On the grid the time-expanded A* explores every (cell, time) pair to the horizon and finds nothing. Swap the priorities and the instance is its own mirror image.
Step 4 — what the coupled planner says. One robot drives to the middle, steps into the alcove, waits while the other passes, steps out, continues. In the composite space this is one path; CBS finds it with a sum of costs of 15 — one robot's seven steps plus the other's eight — and the brute-force joint A* agrees. Neither decoupled stage can express it: the velocity stage cannot change a path, and the prioritized stage never revisits robot 1.
Why decoupling stays useful. Its dimension never grows — m problems of dimension d_i, then a search in $[0, 1]^m$ — and in most apartments robots do not need to swap in a corridor. Choset's remark is the honest summary: incomplete *"even when the algorithms used in both of its stages are complete."* CBS's contribution is to re-plan *paths*, not just velocities, and to do so only

where a conflict demands it.

14.3.4 Conflict-based search is complete and optimal

THEOREM (SHARON, STERN, FELNER AND STURTEVANT 2015)

On a finite graph with unit-time moves and waits, conflict-based search returns a conflict-free joint plan minimizing the sum of individual costs whenever one exists.

DERIVATION Why the first conflict-free node popped is optimal

\mathrm{SIC}(\text{child}) \ge \mathrm{SIC}(\text{parent});\ \text{no solution is pruned}

Step 1 — the low level. For one agent under a set of constraints, Chapter 6's A* over nodes (v, t) with the five moves (four neighbors and a wait) to $(v', t+1)$, minus the constrained nodes and arcs, with the Manhattan heuristic — admissible on a 4-connected grid. Cost convention: a move or a wait away from the goal costs one; a wait at the goal costs nothing, so an agent that arrives and stays is charged exactly its arrival time. The goal is reached when the agent is at its goal cell at a time no later than which no constraint ever touches that cell again.
Step 2 — the high level. Best-first search over a binary tree of constraint sets, keyed on $\mathrm{SIC}(N) = \sum_i \mathrm{cost}(\mathrm{path}_i)$. The root has no constraints and every agent's individually optimal path.
Step 3 — expansion. Pop the cheapest node N; scan its paths in time order for the first conflict. If there is none, return N's paths.
Step 4 — splitting. For a vertex conflict (a_i, a_j, v, t) create two children, adding (a_i, v, t) to one and (a_j, v, t) to the other; for an edge conflict, the two directed arcs. In each child re-plan *only* the constrained agent under its own constraints; every other path is inherited. A child whose agent has no path is discarded.
Step 5 — nothing is pruned. Any conflict-free solution has a_i and a_j not both at v at t, so at least one of them satisfies the new constraint: every valid solution consistent with N is consistent with at least one child. By induction from the root, every valid solution is consistent with some leaf frontier at all times.
Step 6 — the first pop is optimal and the search ends. A child's constraint set contains its parent's, so the constrained agent's optimal cost cannot decrease: SIC is non-decreasing down the tree. Costs are integers. When a

conflict-free node is popped with cost c, every node still in the open list has cost $\geq c$, and by Step 5 every valid solution is consistent with some open node, so none costs less than c. For completeness: the low level has a finite horizon, each constraint deletes one (cell, time) node of a finite graph, so the tree is finite and a solution consistent with the frontier is eventually popped.
Edge conflicts are needed: two agents swapping an arc never share a cell. **Worst case** the tree is exponential in the number of conflicts. **Why it wins** in the apartment regime: conflicts there are few and local, and each one costs two low-level searches rather than a factor of 5 in the branching of a joint search.

14.3.5 A manipulation plan is a path in a union of manifolds

THEOREM (THE STRUCTURE OF MANIPULATION PLANNING)

In the composite space of robot and object, feasible motions lie in $\bigcup_p \mathcal{M}_{transit}(p) \cup \bigcup_{g \in \mathcal{G}} \mathcal{M}_{transfer}(g)$, each of dimension $\dim \mathcal{Q}_{robot}$, and the problem is connectivity of the manipulation graph MG whose nodes are the intersections (p, g).

DERIVATION **Transit, transfer, and the intersections between them**

\text{dim} = \dim \Q_{robot};\ \text{switches only at } (p, g)

Step 1 — transit. An unheld object at a stable placement cannot move: its coordinates are frozen at p. The reachable set is $\mathcal{Q}_{robot} \times \{p\}$, of dimension $\dim \mathcal{Q}_{robot}$, with the object an obstacle in the robot's collision check.
Step 2 — transfer. A held object's configuration is a fixed rigid transform of the hand's: $c_g(q) = \phi(q) \cdot g$ where ϕ is the robot's forward kinematics and g the grasp. The reachable set is the graph $\{(q, c_g(q))\}$, again of dimension $\dim \mathcal{Q}_{robot}$, with the object part of the robot in the check.
Step 3 — switching. Leaving a transit manifold for a transfer manifold requires the object to be at a placement *and* in the hand: $c_g(q) = p$. For Reach and a puck this is two equations in the two joint angles — solved in closed form by Chapter 2's `Reach::ik` on a virtual arm whose last link is the forearm extended to the puck's center. Up to two configurations per (p, g).
Step 4 — the graph. A sequence transit–grasp–transfer–release–transit is a path through MG; a regrasp is a transit edge between two grasps

at one placement. Connectivity of MG is necessary and, if every edge's lower-level planner is complete, sufficient.

Two-level FuzzyPRM (Nielsen and Kavraki, [C ref. 335]) builds MG first with every edge assigned a probability of being free from a few interpolated configurations, searches it, and plans for real only the edges on the candidate path — the lazy verification of Chapter 11, one level up.

aSyMov [C ref. 170] used several specialized roadmaps to connect task-level AI planning to motion planning: proto-TAMP. The **task-and-motion view** (Garrett et al. 2021) is the generalization: symbolic actions are modes, their continuous parameters are grasps and placements, and the planner interleaves the search over the discrete structure with the search in the continuous space — which is exactly what the two levels of MG do, for one object.

14.3.6 Steer-free planners inherit Theorem 7.4.3

THEOREM (PROBABILISTIC COMPLETENESS WITHOUT A STEER)

With $R = \{(q, q') : \exists u \in U,\ q' = f(q, u)$ and the propagated segment is free$\}$, Choset's abstract path-tiling theorem applies verbatim: the random incremental algorithm is probabilistically complete iff it succeeds with nonzero probability, and then its failure probability decays exponentially in the number of samples.

DERIVATION **The proof never used symmetry**

$\Pr[\text{failure}] \le \ell e^{-pn}$

Step 1. R is measurable for a continuous f and a closed obstacle set, which is all Theorem 7.4.3 asks of the relation.

Step 2. Choset, after the proof: *"the symmetry and reflexivity properties of the local planner were never used in the proof. In particular, the proof will still hold for an asymmetric and irreflexive local planner."* A car that can reach q' from q in Δt cannot in general reach q from q'; the relation is asymmetric, and the bound $\ell(1-p)^n \le \ell e^{-pn}$ does not care.

Step 3. Sampling controls instead of configurations changes the measure μ over which the probability p of hitting the next set is computed. The theorem allows any sampling distribution: "the sampling distribution is not necessarily uniform."

The preview stops here. What f *is* for Hitch, why it cannot be inverted

into a `Steer`, and whether the car can reach every configuration at all — controllability — are Chapters 20 and 21.

14.4 The algorithm

Three of this chapter's algorithms are Chapters 11–12's with a different space; the rest are new coordination schemes on top of them.

ALGORITHM `COMPOSITE PRM / RRT -- Chapters 11-12 on the product` COST as Chapters 11-12 with d = Σ d_i; plus $O(m^2)$ pairwise checks per sample

in m bodies with charts Q_i and Chapter 2 checkers; Δ, the sampler, η

out a roadmap or tree in $Q = Q_1 \times \cdots \times Q_m$

1: $Q \leftarrow$ `CompositeSpace`$(Q_1, \ldots, Q_m)$: dist is the ℓ^2 combination, interpolate is componentwise
2: $Q_{\text{free}} \leftarrow$ `MultiBody`: **for each** i check $R_i(q_i)$ against the world; **then for each** $i < j$ check the pair
3: run BASIC PRM (C Alg. 6–7), RRT (C Alg. 10–12) or RRT-Connect (C Alg. 13) with no other change
4: execute: all bodies move together along the composite path at a composite speed v_{max}

ALGORITHM `PRIORITIZED PLANNING -- Choset §7.5.2` COST m single-robot plans in $Q_i \times$ [0, T]; incomplete

in robots in a fixed order, starts, goals, horizon T, v_max

out one trajectory per robot, or failure at the first robot that finds none

1: $O \leftarrow \emptyset$ (moving obstacles)
2: **for** robot i **in** order **do**
3: grow an RRT in $Q_i \times [0, T]$: q_{rand} with t_{rand}; nearest among nodes with $t < t_{rand}$; advance $\le \eta_t$ in time and $\le v_{max}\,\Delta t$ in space; a segment is free iff clear of the world and of every $o \in O$ at every sampled time
4: near the goal: drive to it, then **rest until** T **— that wait must also be free**
5: **if** no trajectory within the budget **then return** failure at i
6: $O \leftarrow O \cup \{$robot i's trajectory$\}$ — never revisited

ALGORITHM `ONE-ROBOT-AT-A-TIME EXTEND -- C §7.5.2, after [C ref. 14]` COST `m sequential extensions, cumulative collision checks`

in a tree node q_near in the composite space, q_rand

out q_new with as many robots advanced as could be, or NIL if none could

1: $q_{new} \leftarrow q_{near}$
2: **for** robot $i = 1 \ldots m$ **do**
3: move robot i incrementally toward $q_{rand,i}$; check against the obstacles **and against robots** $1 \ldots i-1$ **at their new positions**
4: **if** collision **then** sample a new random target for robot i alone and retry; keep the furthest free position
5: **return** q_{new} if any robot moved, else NIL — more expensive than one check of all robots at once, "considerably more effective in covering the space"

ALGORITHM `CBS -- Sharon et al. 2015, Algorithm 1` COST `low level O(|V|·T·log) per re-plan; high level exponential in conflicts in the worst case`

in grid, starts, goals

out conflict-free paths minimizing SIC, or failure

1: root: **for each** agent plan alone by A* over (cell, time); SIC ← sum of costs; OPEN ← {root}
2: **while** OPEN not empty **do**
3: $N \leftarrow$ pop the node of least SIC (ties toward fewer conflicts)
4: $C \leftarrow$ the first conflict in N's paths; **if** none **then return** N's paths
5: **for each** agent a of C **do**
6: $N' \leftarrow N$ plus the constraint (a, v, t) (or the arc); re-plan agent a under its constraints
7: **if** a has a path **then** push N' with its SIC
8: **return** failure

ALGORITHM `TWO-LEVEL MANIPULATION PRM -- after Nielsen & Kavraki [C ref. 335]` COST `upper level over |P|·|𝒢| nodes; lower level a Chapter 12 planner per edge, lazily`

in placements P, grasps $\mathcal{G}$, the robot's start, the object's start and goal placements

out a sequence of transit and transfer paths, or 'MG disconnected'

1: **nodes:** for each (p, g) solve $c_g(q) = p$; keep configurations free in both $\mathcal{M}_{transit}(p)$ and $\mathcal{M}_{transfer}(g)$; add the robot's start as a free-hand node at p_{start}
2: **edges:** transit between nodes sharing p; transfer between nodes sharing g; each with a belief = share of a few interpolated configurations that are free
3: **repeat:** Dijkstra on MG with weight $1 - \log(\text{belief})$ to any node at p_{goal}
4: **for each** edge on the path not yet verified: RRT-Connect in that edge's mode; **if** it fails, delete the edge and go to 3
5: **return** the verified path's segments, each tagged transit or transfer

ALGORITHM FORWARD-PROPAGATED RRT -- classical [C ref. 270-272]; PREVIEW of Chapter 21 COST per iteration one nearest-neighbor query and k propagations; no steer, no extend

in q_init, the simulator f, control sampler over U, Δt, k controls per extension, goal region
out a tree whose every edge is a simulated motion; the controls along the path to the goal

1: $q_{rand} \leftarrow$ SAMPLE (goal with probability p); $q_{near} \leftarrow$ NEAREST(T, q_{rand})
2: **for** $j = 1 \ldots k$ **do** $u_j \leftarrow$ sample U; integrate f from q_{near} under u_j for Δt in substeps, checking each state
3: among the free results keep the one closest to q_{rand} — LaValle and Kuffner's rule; **if** none, the iteration adds nothing
4: add it to T with its control and its simulated segment; **if** within the goal radius, done
5: the path is a list of controls; **executing it means re-running** f, which reproduces every node exactly

14.4.1 The two-disc test, by hand

Corridor $y \in [0, 2]$, two discs of radius 0.5, composite $q = (x_1, y_1, x_2, y_2)$.

At $q = (1.0, 1.0, 1.8, 1.4)$**.** Disc 1's clearance to the walls is $\min(1.0 - 0.5,\ 2 - 1.0 - 0.5) = 0.5$; disc 2 spans $y \in [0.9, 1.9] \subset (0, 2)$. Both are free alone. The centers are $\sqrt{0.8^2 + 0.4^2} = \sqrt{0.80} = 0.8944$ apart, less than $r_1 + r_2 = 1$: $q \in QO_{12}$, collision, penetration 0.1056.

At $q' = (1.0, 1.0, 2.0, 1.4)$**.** $\sqrt{1.0^2 + 0.4^2} = \sqrt{1.16} = 1.0770 > 1$: free.

Along the edge $q' \to q$, only x_2 moves, from 2.0 to 1.8. The edge enters QO_{12} where $(x_2 - 1)^2 + 0.16 = 1$, i.e. $x_2 = 1 + \sqrt{0.84} = 1.9165$. Chapter 11's subdivision schedule at step 0.1 on an edge of length 0.2 has one interior check, the midpoint

$x_2 = 1.9$, where $\|p_1 - p_2\| = \sqrt{0.81 + 0.16} = 0.9849 < 1$: rejected at the first interior check. (`steer` tests the far endpoint q first and would already have said no.)

Prioritized slice. Freeze robot 1 at $(1.0, 1.0)$. Robot 2's slice C-obstacle is the disc of radius 1 about $(1, 1)$, spanning $y \in [0, 2]$ — the full corridor height at $x_2 = 1$. Scanning robot 2's own free band $y \in [0.5, 1.5]$ at $x_2 = 1$, every position collides: blocked fraction 1.00. Robot 2 must wait or fail: Derivation 2, in numbers.

14.4.2 CBS on the swap corridor, by hand

The grid is a corridor of seven cells, $(0, 0)$ to $(6, 0)$, with one alcove cell $(3, 1)$ above the middle; agent 1 goes from $(0, 0)$ to $(6, 0)$, agent 2 the reverse. Every number below is produced by `Cbs` on this instance and pinned by a check.

Root. Each agent planned alone takes the axis in six moves: SIC $= 6 + 6 = 12$. The first conflict is a vertex conflict at cell $(3, 0)$, $t = 3$ — the middle, where two robots driving at the same speed from opposite ends must meet.

Depth 1. Two children, each forbidding $(3, 0)$ at $t = 3$ to one agent. The constrained agent waits one step before the middle: SIC $= 13$ in both. Each now has an *edge* conflict at $t = 4$ — the agents try to swap the arc into the middle cell, which no vertex constraint forbids. This is why edge conflicts are not optional.

Depths 2 and 3. Each edge conflict splits into two arc constraints; the re-planned agent waits once more, SIC $= 14$, and the conflict moves one cell and one time step along the corridor. At depth 3 the costs reach 15 and the tree has grown to 23 nodes, of which CBS expands 12 in order of cost.

Depth 4. The first conflict-free node popped: agent 1 drives $0, 1, 2$, waits, then $3, 4, 5, 6$ — seven steps; agent 2 drives $6, 5, 4, 3$, steps *up* into the alcove $(3, 1)$, steps back down after agent 1 has passed beneath, and continues $2, 1, 0$ — eight steps. SIC $= 15$, which the brute-force joint A* confirms is the minimum: no plan that keeps both robots in the corridor can work, and every plan that uses the alcove costs at least two extra steps for the robot that detours plus one wait for the other. The joint search needed 19 expansions and 138 joint successors to say the same thing; here it is cheap, because there are two agents. With five it is not run.

14.5 Implementation in Rust

The `multi` crate adds no planner. It adds a space (`Composite`), a free-space predicate (`MultiBody`), three coordination schemes (`prioritized`, `coordination`, `cbs`), a mode graph (`manip`) and a propagation trait (`propagate`), and imports everything else from `manifold`, `collide`, `robots`, `search` and `sampling`. The TypeScript port the widgets run is `web/lib/multi/` with the same module names.

RUST **crates/multi/src/composite.rs**

```rust
use manifold::{Chart, Manifold};
use sampling::FreeSpace;

```

```rust
/// ? = A * B. dist is the l2 combination so Theorem 7.4.1's balls are balls; interpolate is
/// componentwise, which is why Chapter 11's straight-line Steer works unchanged on the product.
pub struct Composite<A: Manifold, B: Manifold> { pub a: A, pub b: B }

impl<A: Manifold, B: Manifold> Manifold for Composite<A, B> {
    type Point = (A::Point, B::Point);
    fn dim(&self) -> usize { self.a.dim() + self.b.dim() }
    fn dist(&self, p: &Self::Point, q: &Self::Point) -> f64 {
        self.a.dist(&p.0, &q.0).hypot(self.b.dist(&p.1, &q.1))
    }
    fn interpolate(&self, p: &Self::Point, q: &Self::Point, t: f64) -> Self::Point {
        (self.a.interpolate(&p.0, &q.0, t), self.b.interpolate(&p.1, &q.1, t))
    }
    fn sample(&self, rng: &mut impl rand::Rng) -> Self::Point { (self.a.sample(rng), self.b.sample(rng)) }
}

/// m copies of one chart -- three Rustys. `Product<M, N>` is the const-generic form of the same.
pub struct Product<M: Chart, const N: usize> { pub factor: M }

/// Two discs overlap iff the center distance is below r_1 + r_2: Chapter 4's disc C-obstacle with
/// both bodies moving. In Rust this is one parry2d distance query, the same one the world check makes.
pub fn discs_collide(p1: [f64; 2], r1: f64, p2: [f64; 2], r2: f64) -> (bool, f64) {
    let d = ((p1[0] - p2[0]).powi(2) + (p1[1] - p2[1]).powi(2)).sqrt();
    (d <= r1 + r2, d)
}

/// Free space of m bodies: each against the world (Chapter 2), then every pair. Two counters,
/// because the lab reports how much of the bill is pairwise.
pub struct MultiBody<'w, M: Manifold> {
    pub bodies: Vec<Body<'w>>, pub space: M,
    pub world_checks: u64, pub pair_checks: u64, pub pair_only_rejections: u64,
}

impl<'w, M: Manifold<Point = Vec<[f64; 2]>>> FreeSpace<M> for MultiBody<'w, M> {
    fn is_free(&mut self, q: &M::Point) -> bool {
        // World first: cheaper, and where most random composite samples die.
        for (b, qi) in self.bodies.iter().zip(q) { self.world_checks += 1; if !b.world.is_free(qi) { return false; } }
        for i in 0..q.len() { for j in i + 1..q.len() {
            self.pair_checks += 1;
            if discs_collide(q[i], self.bodies[i].r, q[j], self.bodies[j].r).0 { self.pair_only_rejections += 1; return false; }
        } }
        true
    }
}
```

The two-disc test is the unit test and the worked example at once:

RUST **crates/multi/examples/two_discs.rs**

```rust
use multi::composite::{discs_collide, MultiBody};
```

```rust
use sampling::steer::{check_schedule, EdgeCheck};

fn main() {
    let r = 0.5;
    let (q1, q2) = ([1.0, 1.0], [1.8, 1.4]);
    let (hit, d) = discs_collide(q1, r, q2, r);
    println!("q : dist {d:.4}, {}, penetration {:.4}", if hit { "collide" } else { "free"
    }, 2.0 * r - d);
    let (hit, d) = discs_collide(q1, r, [2.0, 1.4], r);
    println!("q': dist {d:.4}, {}", if hit { "collide" } else { "free" });
    // Boundary along the edge q' -> q: (x_2 - 1)2 + 0.42 = 1.
    println!("boundary x2 = {:.4}", 1.0 + (1.0 - 0.16f64).sqrt());
    // Subdivision at step 0.1 on an edge of length 0.2: one interior check, the midpoint.

    let ts = check_schedule(EdgeCheck::Subdivision { step: 0.1 }, 0.2);
    let x2 = 2.0 - 0.2 * ts[0];
    let (hit, d) = discs_collide(q1, r, [x2, 1.4], r);
    println!("first interior check x2 = {x2:.1}: dist {d:.4} -> {}", if hit { "rejected" }
    else { "accepted" });
    // Robot 2's slice at x_2 = 1 with robot 1 pinned: scan its own free band.
    let blocked = (0..400).filter(|k| { let y = 0.5 + (*k as f64 + 0.5) / 400.0;
   discs_collide(q1, r, [1.0, y], r).0 }).count();
    println!("slice at x2 = 1.0: {:.2} blocked", blocked as f64 / 400.0);
}

#[test]
fn reproduces_two_disc_test() {
    let (hit, d) = discs_collide([1.0, 1.0], 0.5, [1.8, 1.4], 0.5);
    assert!(hit && (d - 0.8944).abs() < 1e-4 && (1.0 - d - 0.1056).abs() < 1e-4);
    let (hit, d) = discs_collide([1.0, 1.0], 0.5, [2.0, 1.4], 0.5);
    assert!(!hit && (d - 1.0770).abs() < 1e-4);
    assert!((1.0 + 0.84f64.sqrt() - 1.9165).abs() < 1e-4);
    let (hit, d) = discs_collide([1.0, 1.0], 0.5, [1.9, 1.4], 0.5);
    assert!(hit && (d - 0.9849).abs() < 1e-4);
    // MultiBody agrees with the closed form on 1 000 seeded composite configurations (
   parry2d vs. hypot).
    let mut rng = rand::rngs::SmallRng::seed_from_u64(14);
    let mut mb = MultiBody::two_discs(&CORRIDOR, 0.5);
    for _ in 0..1000 { let q = mb.space.sample(&mut rng); assert_eq!(mb.is_free(&q),
   MultiBody::closed_form(&CORRIDOR, &q, 0.5)); }
}
```

Expected output:

```
q : dist 0.8944, collide, penetration 0.1056
q': dist 1.0770, free
boundary x2 = 1.9165
first interior check x2 = 1.9: dist 0.9849 -> rejected
slice at x2 = 1.0: 1.00 blocked
```

Conflict-based search needs the time-expanded grid as a Chapter 6 `Graph`, and nothing else new:

RUST **crates/multi/src/cbs.rs**

```rust
use search::{astar, Graph, Edge};
use search::grid::{Grid, GridIdx};

```

```rust
pub struct Conflict { pub a: usize, pub b: usize, pub cell: GridIdx, pub t: u32, pub edge: Option<(GridIdx, GridIdx)> }
pub struct Constraint { pub agent: usize, pub cell: GridIdx, pub t: u32, pub from: Option<GridIdx> }

/// One agent's search space: (cell, time), five moves, minus what its constraints delete.
struct TimedGrid<'g> { grid: &'g Grid, goal: GridIdx, vertex_ban: HashSet<(GridIdx, u32)>, edge_ban: HashSet<(GridIdx, GridIdx, u32)>, last_goal_ban: i64, horizon: u32 }

impl<'g> Graph for TimedGrid<'g> {
    type Node = (GridIdx, u32);          // t = u32::MAX marks "resting at the goal from here on"
    fn neighbors(&self, &(cell, t): &Self::Node) -> Vec<Edge<Self::Node>> {
        let mut out = Vec::new();
        if cell == self.goal && t as i64 >= self.last_goal_ban { out.push(Edge { to: (cell, u32::MAX), cost: 0.0 }); }
        if t >= self.horizon { return out; }
        for nb in self.grid.moves_with_wait(cell) {
            if self.vertex_ban.contains(&(nb, t + 1)) { continue; }
            if nb != cell && self.edge_ban.contains(&(cell, nb, t + 1)) { continue; }
            // A wait at the goal is free; everything else costs one time step.
            let cost = if nb == self.goal && cell == self.goal { 0.0 } else { 1.0 };
            out.push(Edge { to: (nb, t + 1), cost });
        }
        out
    }
}

pub struct CtNode { pub constraints: Vec<Constraint>, pub paths: Vec<Vec<GridIdx>>, pub cost: u32 }

pub struct Cbs<'g> { grid: &'g Grid, pub expansions: u64, pub low_level_expansions: u64 }

impl<'g> Cbs<'g> {
    /// Sharon et al., Algorithm 1: best-first on SIC over a binary constraint tree.
    pub fn solve(&mut self, starts: &[GridIdx], goals: &[GridIdx]) -> Option<Vec<Vec<GridIdx>>> {
        let mut open = BinaryHeap::new();
        open.push(self.root(starts, goals)?);
        while let Some(n) = open.pop() {
            self.expansions += 1;
            let Some(c) = first_conflict(&n.paths) else { return Some(n.paths) };
            for (agent, constraint) in split(&c) {
                let mut cs = n.constraints.clone(); cs.push(constraint);
                // Re-plan only the constrained agent; inherit every other path verbatim.
                let own: Vec<_> = cs.iter().filter(|k| k.agent == agent).collect();
                if let Some(path) = self.low_level(starts[agent], goals[agent], &own) {
                    let mut paths = n.paths.clone(); paths[agent] = path;
                    let cost = sic(&paths, goals);
                    open.push(CtNode { constraints: cs, paths, cost });
                }
            }
        }
        None
    }
}
```

The manipulation graph and the propagation trait are the two remaining

types; the first is a `petgraph::UnGraph` over `(placement, grasp)`, the second is as small as the preview allows:

RUST **crates/multi/src/manip.rs · src/propagate.rs**

```rust
pub enum Mode { Transit { placement: usize }, Transfer { grasp: usize } }

pub struct Grasp { pub alpha: f64 }      // the puck's bearing in the hand frame
pub struct Placement { pub x: f64, pub y: f64 }

impl PuckArm {
    /// c_g(q): the held puck is a rigid transform of the hand.
    pub fn held_center(&self, q: &T2, g: &Grasp) -> Point2 {
        let tip = self.arm.tip(q); let h = self.arm.tip_heading(q) + g.alpha;
        Point2::new(tip.x + self.hold * h.cos(), tip.y + self.hold * h.sin())
    }
    /// Grasp configurations at (p, g): Chapter 2's IK on a virtual arm whose last link is
    /// link 2 extended by the hold offset, so that the virtual tip is the puck's center.
    pub fn grasp_configs(&self, p: &Placement, g: &Grasp) -> Vec<T2> {
        let w = Vector2::new(self.arm.l[1] + self.hold * g.alpha.cos(), self.hold * g.alpha.sin());
        let virt = self.arm.with_lengths([self.arm.l[0], w.norm()]);
        virt.ik(p.into()).into_iter().flatten().map(|(t1, t2)| T2::new(t1, t2 - w.y.atan2(w.x))).collect()
    }
}

/// MG: nodes (p, g, ik branch); edges transit (same p) or transfer (same g), each with a FuzzyPRM belief.
pub struct ManipulationGraph { pub g: UnGraph<MgNode, MgEdge>, pub start: NodeIndex, pub goals: Vec<NodeIndex> }

/// PREVIEW (Chapter 21 owns the full version). A simulator step; Hitch's Chapter 2 kinematic
/// update implements it as a black box. Nothing about the dynamics is derived or inspected here.
pub trait Propagate<M: Manifold> {
    type Control;
    fn propagate(&self, q: &M::Point, u: &Self::Control, dt: f64) -> M::Point;
}

impl Propagate<SE2> for Hitch {
    type Control = CarCmd;
    fn propagate(&self, q: &Pose2, u: &CarCmd, dt: f64) -> Pose2 { self.step(&HitchCfg { body: *q, trailer: None }, u, dt).body }
}

pub struct KinoRrtPreview<M: Chart, P: Propagate<M>> {
    pub tree: Tree<M>, pub sim: P, pub controls: Vec<Option<P::Control>>, pub k: usize, pub dt: f64, pub rng: SmallRng,
}

impl<M: Chart, P: Propagate<M>> KinoRrtPreview<M, P> {
    /// Sample q_rand; nearest; sample k controls; propagate each in substeps, checking every state;
    /// keep the free result closest to q_rand (LaValle & Kuffner's rule). No Steer, no Extend.
    pub fn extend_once(&mut self, free: &mut dyn FreeSpace<M>, sample_u: &mut dyn FnMut(&mut SmallRng) -> P::Control) -> Option<NodeId> {
        let q_rand = self.tree.space.sample(&mut self.rng);
```

```
        let (near, _) = self.tree.nearest(&q_rand);
        let mut best: Option<(P::Control, M::Point, f64)> = None;
        for _ in 0..self.k {
            let u = sample_u(&mut self.rng);
            let Some(q) = self.propagate_checked(free, &self.tree.q(near), &u) else {
    continue };
            let d = self.tree.space.dist(&q, &q_rand);
            if best.as_ref().map_or(true, |b| d < b.2) { best = Some((u, q, d)); }
        }
        let (u, q, _) = best?;
        let id = self.tree.add(q, near);
        self.controls.push(Some(u));
        Some(id)
    }
}
```

`cargo run --example three_rustys -p multi` runs prioritized planning in both orders, coupled PRM on `Product<R2, 3>` and CBS on the inflated Apartment grid, printing success, makespan, SIC and check counts as a snapshot test at seed 14; `hitch_no_steer` grows `KinoRrtPreview` in the Lot and asserts the goal region within 4 000 iterations. The TypeScript checks behind this page do the same: on seed 14 the forward-propagated tree reaches the Lot goal at iteration 63 with 63 nodes after 504 propagations, 140 of them wasted on a collision; the path has 14 nodes, and re-running its 13 stored controls through `Hitch.step` reproduces every one of them to the last bit, all 66 simulator states are free, and the chord-versus-mean-heading mismatch of every edge is zero to machine precision — the signature of an exact arc.

14.6 Putting it together: three ways to pay

INTERACTIVE · w14.3

Hitch Without a Steer

An RRT does not need a steering function: sample controls, integrate the simulator, keep the free result closest to q_rand — and every edge is drivable by construction.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch14-many-bodies#w14.3

The integration lab is the three widgets above read side by side, with the bills totted up.

Three Rustys. In the swap corridor, prioritized planning fails in both orders at a cost of about 3 000 configuration–time RRT iterations each; the decoupled coordinator fails in one grid search over a 48×48 diagram; CBS on the grid version succeeds with a sum of costs of 15 after 12 constraint-tree expansions and 23 nodes, each expansion two time-expanded A* searches; the brute-force joint A* agrees on 15 after 19 expansions and 138 joint successors — cheap for two

agents, and on the three-agent room already 1412 successors against CBS's three nodes. The coupled RRT-Connect in $(\mathbb{R}^2)^2$ succeeds at 2562 attempts and 3278 composite checks. Every one of those numbers is produced by the code behind the page and pinned by a check.

Reach and the puck. Seven edges in the manipulation graph, four verified, 218 collision checks at the lower level, one upper-level round. Switch the eager toggle in w14.2 and every edge is planned whether or not it is needed — Exercise 4 asks you to count.

Hitch without a steer. The preview above. One control per extension and the tree wanders; twenty and most propagations are thrown away; eight is the default because that is where, on the seeds tried, the time to the goal region stopped improving. The grey ghost is Chapter 12's RRT on SE(2) pretending θ is a coordinate like any other: it reaches the goal too, along a path that slides sideways by several radians in total. A car cannot execute it. The blue tree's path it can, because every edge *was* an execution. This is as far as the chapter goes: the simulator is a black box, and Chapter 21 opens it.

14.6.1 OMPL and MoveIt

Everything in Part III is infrastructure in two systems you will meet in practice. The Open Motion Planning Library (Şucan, Moll and Kavraki 2012) is the sampling-based core; MoveIt (Coleman, Şucan, Chitta and Correll 2014) wraps it with a planning scene, kinematics plugins and request adapters for manipulators. The mapping onto this book's traits is interface-level and worth having in one place:

OMPL	this book	chapter
`StateSpace`, `CompoundStateSpace`	`Manifold`, `Product` / `Composite Space`	5, 14
`StateSampler`	`Sampler`	11
`StateValidity Checker`	`FreeSpace` over a `Collision`	2, 11
`MotionValidator`	`Steer` + `EdgeCheck`	11
`Planner`: `PRM`, `RRT`, `RRTConnect`, `EST`, `SBL`, `KPIECE1`, `RRTstar`, `PRMstar`, `Informed RRTstar`, `BITstar`, `FMT`	`Prm`, `Rrt`, `RrtConnect`, `Est`, `Sbl`, `Kpiece`, `RrtStar`, `PrmStar`, `InformedRrtStar`, `Bit`, `Fmt`	11–13
`control::` `SpaceInformation`, `StatePropagator`	`Propagate` (preview)	14, 21
MoveIt `Planning Scene`, `RobotModel`, `CollisionDetection`	`Scene`, `Reach/Hitch`, `Collision`	2

OMPL	this book	chapter
MoveIt `Planning RequestAdapter` chain (fix start state, shortcut, time-parametrize)	`shortcutGreedy`; Chapter 18's time scaling	11, 18

The point of the table is not that the names line up; it is that nothing in those systems is a different *kind* of thing from what this book has built, body by body. The capstone's stack is mapped onto OMPL, MoveIt and Nav2 through this table in Chapter 23.

14.6.2 Where §7.5.4–7.5.6 went

> INTERACTIVE · `f14.5`
>
> **Pointer box: assembly, flexible objects, biology**
>
> Three of Choset's §7.5 extensions are condensed here, not taught: each is Part III's machinery on a configuration space with a different shape.
>
> Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch14-many-bodies#f14.5`

Choset's remaining three subsections — assembly planning, flexible objects, biological applications — are each a configuration space with a different shape under the same machinery: one subassembly as the robot and the rest as the workspace; a deformable body with an energy threshold standing in for a free-space predicate; a molecule as a tree-like articulated robot with dihedral angles as joints. The pointer box above condenses each to a paragraph. Nonprehensile manipulation — parts feeding, squeeze grasps, programmable vector fields, the $2n - 1$ squeezes that orient any polygonal part — and the NonDirectional Blocking Graph are mentioned in those paragraphs and taught nowhere in this book. Parallel SRT, which Choset notes plans for many robots by prioritizing each roadmap edge and centralizing its endpoints, is a sentence here and an exercise in Chapter 12.

Part IV begins with the assumption every chapter so far has made in silence: that the robot knows where it is.

14.7 Exercises

1 FOUNDATION •• **Which C-obstacle dominates**

Two unit-radius discs translate in $[0, 10]^2$ each, so $Q = [0, 10]^4$. Compute the four-dimensional measure of $QO_{12} \cap [0, 10]^4$ exactly (hint: it is the measure of the set of pairs of centers within distance 2, which is $\iint$ of the area of a disc of radius 2 clipped to the square — bound it above and below first), and compare it with the measure of $QO_1^{\mathcal{W}}$ for a 1×1 pillar, lifted to Q. Then let the number of discs grow to m and say which family of obstacles dominates the measure of $Q \setminus Q_{\text{free}}$, and why the answer is the same statement as "pair checks grow like m^2."

WORK IT OUT

Ignoring the clipping at the square's edges, what fraction of [0,10]^4 does QO_{12} occupy? (Area of a disc of radius 2 over the area of the square.)

Answer. 0.1257 $\pi \cdot 2^2/100 = 0.1257$: for every position of disc 1, disc 2's center must avoid a disc of radius 2, which is 4π of the 100 available. The lifted pillar, inflated by the unit radius to a 3×3 square, removes $9/100 = 0.09$ for *each* disc. With m discs there are m lifted pillars and $\binom{m}{2}$ pair cylinders; the pairs dominate from $m = 3$ on.

2 FOUNDATION ••• **CBS never prunes a solution**

Prove Step 5 of the CBS derivation with edge conflicts included: for an edge conflict in which a_i traverses $u \to v$ and a_j traverses $v \to u$ between $t - 1$ and t, show that every conflict-free solution consistent with the parent is consistent with at least one of the two children. Then construct a two-agent instance whose solution lies at depth ≥ 3 in the constraint tree — three constraints before the first conflict-free node — and verify your depth with the `Cbs` class's `solve().depth`.

HINTS

<>A conflict-free solution cannot have both agents traverse the same arc in opposite directions at the same time, so at least one of the two arc constraints holds for it.</> <>For depth 3: a corridor where two agents must swap and the only passing place is two cells away from where they first meet, so the first constraint merely shifts the conflict by one step.</> <>The swap corridor of this chapter solves at depth 4 on seed-independent CBS (it is deterministic); shorten the corridor on one side and watch the depth change.</>

3 CONCEPTUAL • **Predict the width at which both orders fail**

PREDICT FIRST

In w14.1 at 3.2 radii of corridor width both prioritized orders fail. Widen the corridor. Before dragging: at what width do both orders start to succeed, and what happens to the coordination diagram's band there?

- At exactly 4 radii; the band disappears entirely
  At 4 radii the discs can just pass each other — but only along one line, and a time-expanded RRT must find that line. The band thins but does not vanish until the robots have room to spare: the diagram is built from the two axis paths, which still overlap at $s_1 + s_2 = 1$ whenever the robots are on the same line.
- ✓ Somewhat above 4 radii; the band stays as long as both paths are the axis
  Right. The prioritized planner needs room to sidestep in configuration–time space, which it finds reliably a little above 4 radii. The coordination diagram, however, is computed from the two paths planned alone — both the corridor axis — so its band is unchanged by width; decoupling fails at every width until the alone-paths themselves differ.
- Never: the alcove is the only passing place
  The alcove is the only passing place below 4 radii. Above it the corridor itself is wide enough for two discs side by side.

Verify, then note the asymmetry: a planner that can change *paths* (prioritized, coupled) is helped by width; one that can only change *velocities* (decoupled) is not, as long as its first stage keeps returning the same two paths.

4 CONCEPTUAL •• **Minimum grasps, and the price of eagerness**

PREDICT FIRST

In w14.2 with $|\mathcal{G}| = 1$ the manipulation graph is disconnected. What is the minimum $|\mathcal{G}|$ that connects it with placement C enabled, and through which node does the path pass?

- ✓ $|\mathcal{G}| = 2$, regrasping at C from the front grasp to the left grasp
  A, 2.1 m from the base, is reachable only by grasps that hold the puck ahead of the forearm (of the first two, only the front grasp); B, behind the bar, only by a side grasp. No single grasp works at both, so one transfer cannot do it; with front and left the plan carries the puck to C, releases, and regrasps from the left.
- $|\mathcal{G}| = 2$, a single transfer from A to B
  No single grasp is feasible at both A and B: the grasps that reach A hold the puck ahead of the forearm, and exactly those are blocked by the bar at B.
- $|\mathcal{G}| = 3$ — the right grasp is needed too
  The right grasp also works at B, but the left one already does; the third grasp adds nodes and edges without changing connectivity.

Now switch the eager toggle on and read the verified-edges counter: every edge of MG is planned, including the ones no path uses. Record the ratio of collision checks eager/lazy at $|\mathcal{G}| = 2$ and again at $|\mathcal{G}| = 6$, where MG has 45 edges.

5 PRACTICAL •• **One robot at a time**

Implement Choset's one-robot-at-a-time extension (§7.5.2, after [C ref. 14]) as an `Extend` implementation on `Product<R2, N>`: move robot 1 toward its target and check it against the obstacles; then robot 2, checked against the obstacles *and* robot 1's new position; and so on, re-sampling a robot's target when it collides. Compare its coupled-RRT success rate at a fixed budget with the componentwise `extend` of Chapter 12 over 50 seeds in a four-Rusty Apartment, and report the mean number of configurations "close to obstacles" that Choset says the plain extension produces.

6 PRACTICAL ••• **Makespan CBS against coupled A***

Add a makespan objective to `Cbs` — the high level keyed on $\max_i \text{cost}_i$ rather than the sum — keeping edge conflicts, and prove or refute that the first conflict-free node popped is still optimal (the sum argument used integrality and monotonicity; check both). Benchmark CBS under both objectives against `jointAStar` on the inflated Apartment grid with 3 to 6 agents, plotting constraint-tree expansions and joint successors generated against the number of agents, and mark the agent count at which the joint search stops finishing.

14.8 References

Sharon, G., Stern, R., Felner, A., and Sturtevant, N. R. (2015). Conflict-Based Search for Optimal Multi-Agent Pathfinding. *Artificial Intelligence 219, 40–66*.

CBS as this chapter states and proves it: the two-level search, vertex and edge conflicts, the constraint tree, and the optimality argument of Derivation 3. doi:10.1016/j.artint.2014.11.006

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

§7.5: control-based planning, centralized/decoupled/prioritized multi-robot planning with its incompleteness remark and the one-robot-at-a-time extension [ref. 14], transit/transfer manipulation with the two-level FuzzyPRM of Nielsen and Kavraki [ref. 335] and aSyMov [ref. 170], and the velocity coordination of Kant and Zucker [ref. 219]; the epigraph is from §7.5.2. link

LaValle, S. M. and Kuffner, J. J. (2001). Randomized Kinodynamic Planning. *International Journal of Robotics Research 20(5), 378–400*.

The forward-propagated RRT of the preview — sample controls, integrate, keep the result closest to the random state — in its original kinodynamic setting; Chapter 21 returns to it in full. doi:10.1177/02783640122067453

Kuffner, J. J. and LaValle, S. M. (2000). RRT-Connect: An Efficient Approach to Single-Query Path Planning. *IEEE International Conference on Robotics and Automation*.

The bidirectional planner the coupled lab and the manipulation planner's lower level run unchanged on the composite space and on each mode's torus. doi:10.1109/ROBOT.2000.844730

Kavraki, L. E., Švestka, P., Latombe, J.-C., and Overmars, M. H. (1996). Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces. *IEEE Transactions on Robotics and Automation 12(4), 566–580*.

The roadmap whose two-level, lazily verified descendant plans the manipulation graph's edges, and whose sample-count bound Derivation 1 reads with $d = \Sigma\, d_i$. doi:10.1109/70.508439

Garrett, C. R., Chitnis, R., Holladay, R., Kim, B., Silver, T., Kaelbling, L. P., and Lozano-Pérez, T. (2021). Integrated Task and Motion Planning. *Annual Review of Control, Robotics, and Autonomous Systems 4, 265–293*.

The task-and-motion view that generalizes the manipulation graph: symbolic actions as modes, continuous parameters as grasps and placements, interleaved discrete and continuous search. link

Şucan, I. A., Moll, M., and Kavraki, L. E. (2012). The Open Motion Planning Library. *IEEE Robotics and Automation Magazine 19(4), 72–82*.

The system in which Part III is infrastructure; the interface mapping StateSpace ↔ Manifold, StateSampler ↔ Sampler, MotionValidator ↔ Steer, Planner ↔ Prm/Rrt/RrtStar is drawn from its design. doi:10.1109/MRA.2012.2205651

Coleman, D., Şucan, I. A., Chitta, S., and Correll, N. (2014). Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study. *Journal of Software Engineering for Robotics 5(1)*.

MoveIt as the planning scene, kinematics plugins and request adapters around OMPL — the manipulation-side half of the systems table. `link`

Part IV

Planning with Uncertainty

CHAPTER 15

Kalman Filtering: The Geometric Observer

PLANNING WITH UNCERTAINTY · Intermediate · 55 min

> *The Kalman filter is both an observer and a filter.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), §8.2.1*

Fourteen chapters of planners assumed the planner knew q. This chapter admits that Rusty does not. Encoders drift, landmarks are noisy, and the configuration a planner imagines is only a *belief* about the state the robot occupies. Choset's Chapter 8 teaches the Kalman filter as no other robotics text does — not as Bayes' rule with Gaussians, which the sister book does completely, but as an **observer**: a copy of the system corrected toward the set of states consistent with the latest measurement.

The whole chapter is one picture. An ellipse of plausible states touches a line of measurement-consistent states, and the point of contact is the update. Once that picture is in place, the Kalman gain, the innovation covariance, data association by χ^2, and Kalman SLAM all fall out of it — and so does the warning that an optimal observer still cannot learn a direction the sensors never see. Everything deeper is a link into the sister volume. What stays here is exactly the estimation vocabulary Chapters 16 and 23 consume.

◎ AFTER THIS CHAPTER YOU CAN

- See the Kalman filter as a Luenberger observer whose gain is recomputed every step, and say precisely what that gain optimizes
- Derive the update as a projection onto the measurement hyperplane:

first Euclidean, then Mahalanobis, then onto the most likely output
- Run Choset's dead-reckoning example by hand and read off why one variance shrinks while the other grows under an optimal filter
- State and apply the observability rank test, for linear systems and for the linearized pair of a landmark localizer
- Wrap the sister book's vendored Kalman code in a thin estimate crate: observer, rank test, Choset-notation adapters, landmark models, Kalman SLAM

ASSUMED

- Rusty, the Apartment, and the range sensor (Chapter 2); SE(2) as a manifold and why heading needs care (Chapter 5)
- Linear algebra: null spaces, column spaces, positive-definite matrices as inner products
- Multivariate Gaussians — the sister book's Chapter 2 is the primer this chapter assumes

15.1 The problem: a planner that does not know where it is

Put Rusty in the Apartment corridor and ask it to drive a loop on encoders alone. Each tick the wheels report how far they turned; integrate that and you have a position. The orange track below is that integral. The gray dashed track is where the robot actually went.

INTERACTIVE · f15.0

Dead-Reckoning Drift

Integrating odometry only ever loses information. Without an observer, the error grows without bound — and nothing in the integration can tell you so.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch15-kalman-observer#f15.0

Two things go wrong, and they are different in kind. The right wheel is a fraction of a percent larger than the odometry believes, so every straight run bends into an arc — a *bias*, invisible to any amount of averaging. On top of it the wheels slip by a random amount each tick — *noise*, which random-walks the heading and, through the heading, the position. Choset's Figure 8.1 shows the same thing on a B21: a rotational error approaching 45° after one lap. The command-integration estimate "is continuously losing information about its

location," and nothing in the integration can tell you how much.

The cure is also in the widget. Tick *landmark fixes*. Whenever a mapped corner comes within range, one range-and-bearing sighting pulls the estimate back and collapses the ellipse. That pull is the subject of this chapter: how hard to tug, in which direction, and what the tug can and cannot fix.

One sentence of bookkeeping, said once for all of Part IV. For Rusty the planner's configuration $q \in \mathrm{SE}(2)$ and the filter's state x_t are the same object; the sister book writes x_t, u_t, z_t and this book keeps Choset's $x(k), u(k), y(k)$ because his geometry reads better in it. The notation table below sits the two side by side so nobody translates twice.

15.2 Building intuition

15.2.1 The knob the filter turns

Strip the problem to Choset's §8.2.6 toy: a robot on a rail with state $x = [x_r, v_r]^{\mathsf{T}}$, a force input, Newton's law sampled at $T = 0.5$ s, and a noisy velocity sensor. An **observer** for it is a copy of the dynamics plus a correction proportional to the innovation — the difference between what the sensor said and what it would have said had the prediction been exact:

$$\hat{x}(k{+}1 \mid k) = F\,\hat{x}(k \mid k) + G\,u(k), \qquad \hat{x}(k{+}1 \mid k{+}1) = \hat{x}(k{+}1 \mid k) + K\left[\, y(k{+}1) - H\,\hat{x}(k{+}1 \mid k) \,\right].$$

K is a knob. Turn it to zero and you have dead reckoning; turn it up and the estimate chases the sensor, noise and all; turn it far enough and the estimate oscillates and diverges. The Observer Bench lets you turn it. Then it lets the filter turn it for you.

> **INTERACTIVE · w15.1**
>
> **Observer Bench**
>
> The Kalman gain is not a tuning constant. It is a trajectory R(k) — and the knob you were turning is what the filter computes.
>
> Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch15-kalman-observer#w15.1`

Watch three things. The ellipse in the phase plane grows orange on every predict and squishes purple on every update, always toward the green line of states consistent with the reading. The dots on the unit disc are the eigenvalues of the error dynamics; with the velocity sensor one of them sits at exactly 1 no matter what you do, and the position variance climbs forever. And when you tick *let the filter choose*, the gain stops being a constant: it becomes a sequence $R(k)$ that starts large, settles, and the "equivalent knob" readout drifts to 1 on its own.

The Kalman gain is not a tuning constant. It is a trajectory, and the knob you were turning is what the filter computes.

$x(k) \in \mathbb{R}^n$, $u(k) \in \mathbb{R}^m$, $y(k) \in \mathbb{R}^p$	State, input, output at sample k (period T). Sister book: x_t, u_t, z_t
$F(k)$, $G(k)$, $H(k)$	Dynamics, input, and output matrices. Sister book: A_t, B_t, C_t
$v(k) \sim \mathcal{N}(0, V)$, $w(k) \sim \mathcal{N}(0, W)$	Process and measurement noise and their covariances. Sister book: N(0, R_t), N(0, Q_t)
$\hat{x}(k_1 \mid k_2)$, $P(k_1 \mid k_2)$	Estimate and error covariance at k_1 given outputs through k_2. Sister book: μ_t, Σ_t (bars when predicted)
$\Omega = \{x : Hx = y\}$, Ω^*	States consistent with the output y; with the most likely output y*.
$\nu = y - H\hat{x}(k+1 \mid k)$, $S = HPH^\mathsf{T} + W$	Innovation and its covariance. Sister book: z_t − C_t $\bar{\mu}$_t, S_t
$R = PH^\mathsf{T}S^{-1}$	The Kalman gain — Choset's letter. Not a covariance. Sister book: K_t. Thrun's R is the process covariance.
$\|x\|_M^2 = x^\mathsf{T}P^{-1}x$	Mahalanobis norm under the predicted covariance P(k+1 | k).
$Q = [H;\ HF;\ \ldots;\ HF^{n-1}]$	Observability matrix (Thm. 8.2.2).
$a(\cdot)$, $\chi^2_{ij} = \nu_{ij}^\mathsf{T} S_{ij}^{-1} \nu_{ij}$	Association map; Mahalanobis innovation of measurement i against landmark j. Sister book: c_t^i, d_ij

THE SYMBOL HAZARD, STATED ONCE

Choset's R is the **gain** and his V, W are **covariances**. The sister book, in Thrun's convention, uses K for the gain and R, Q for the covariances. The vendored code speaks Thrun; the `estimate` crate translates at its boundary and nowhere else, and every widget labels both.

15.2.2 The picture

Here is the one picture the chapter is built on, drawn from the numbers of Choset's example. The orange ellipse is the predicted belief. The green line is Ω: every state that would produce the reading exactly. Where should the estimate go?

INTERACTIVE · f15.3

Hyperplane and Ellipse

The update is perpendicular to the measurement line — in the covariance metric. That is why uncertain directions move more.

> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch15-kalman-observer#f15.3

The naive answer is the gray foot: drop perpendicularly onto Ω, correct the velocity the sensor measured and leave the position alone. The right answer is the purple foot: perpendicular *in the metric of the covariance*. Position and velocity are correlated in P — a robot that was going faster than we thought has also gone farther — so correcting the velocity should drag the position with it, and the correlated, more uncertain direction moves more. Add sensor noise and the line itself moves to Ω^* through the most likely output y^*; the Kalman update is the Mahalanobis foot on *that* line. Three derivations, one picture.

15.3 The mathematics

15.3.1 Definitions

An **observer** for the system $x(k{+}1) = Fx(k) + Gu(k)$, $y(k) = Hx(k)$ is a recursion that reconstructs $x(k)$ from the known signals $u(\cdot)$, $y(\cdot)$ and the model (F, G, H) when H is not invertible — which it usually is not, being $p \times n$ with $p < n$ (§8.2.1; App. J.4). The Kalman filter is both an observer and a filter: it reconstructs the state *and* suppresses the noise.

The **innovation** $\nu = y(k{+}1) - H\hat{x}(k{+}1 \mid k)$ is what the sensor said minus what it would have said had the prediction been exact. The **consistent set** $\Omega = \{x \in \mathbb{R}^n : Hx = y(k{+}1)\}$ is the affine subspace of states that reproduce the output; its dimension is $n - p$ when H has full row rank, which Choset assumes throughout §8.2.

The **Mahalanobis distance** $d_M(x, \hat{x}) = \|x - \hat{x}\|_M$ with $\|x\|_M^2 = x^{\mathsf{T}} P(k{+}1 \mid k)^{-1} x$ measures distance in standard deviations under the predicted covariance (§8.2.3, footnote 5). It comes from an inner product $\langle x_1, x_2 \rangle_M = x_1^{\mathsf{T}} P^{-1} x_2$, so "orthogonal" has a meaning under it.

A linear system is **observable** when the initial state is determined by u, y over a finite window; for LTI systems, iff $\operatorname{rank} Q = n$ (App. J.4, Thm. J.4.1). The **Luenberger observer** (eq. J.9) is $\dot{\hat{x}} = A\hat{x} + Bu + K(y - C\hat{x})$, with error dynamics $\dot{e} = (A - KC)e$; its discrete cousin is what the Bench runs.

Data association is the choice of $a(i)$, the landmark that explains measurement i (§8.3.2).

15.3.2 The simple observer is an orthogonal projection

STATEMENT (§8.2.2)

With no noise, the update closest to the prediction in the Euclidean sense is $\hat{x}^+ = \hat{x}^- + H^\mathsf{T}(HH^\mathsf{T})^{-1}\,\nu$.

DERIVATION Projecting the prediction onto Ω

$\Delta x = H^\mathsf{T}(HH^\mathsf{T})^{-1}\nu$

Write $\Delta x = \hat{x}(k{+}1 \mid k{+}1) - \hat{x}(k{+}1 \mid k)$ and ask for the shortest Δx that lands in Ω.

Step 1 — the true state lies in Ω. With no measurement noise, $Hx(k{+}1) = y(k{+}1)$ exactly. So the update should land in Ω, and since the prediction is our best guess, it should be the point of Ω nearest the prediction.

Step 2 — shortest means orthogonal. The shortest vector from a point to an affine subspace is orthogonal to it: $a^\mathsf{T}\Delta x = 0$ for every a parallel to Ω. A vector is parallel to Ω iff $Ha = 0$, i.e. $a \in \mathrm{null}(H)$ (Choset's problems 4–5).

Step 3 — orthogonal to the null space means in the column space of H^T. This is the fundamental theorem of linear algebra (problem 6): $\mathrm{null}(H)^\perp = \mathrm{column}(H^\mathsf{T})$. So $\Delta x = H^\mathsf{T}\gamma$ for some $\gamma \in \mathbb{R}^p$.

Step 4 — guess γ linear in the innovation. Try $\gamma = K\nu$ with $K \in \mathbb{R}^{p\times p}$, so $\Delta x = H^\mathsf{T}K\nu$. Enforce membership in Ω: $H(\hat{x}^- + \Delta x) = y$, i.e. $H\Delta x = \nu$.

Step 5 — solve. $HH^\mathsf{T}K\nu = \nu$ for every ν forces $K = (HH^\mathsf{T})^{-1}$, which exists because H has full row rank. The guess is verified and the observer is complete.

Why it is not enough. The update is always perpendicular to Ω, so only the component of the state that directly affects the current reading is corrected. Errors parallel to Ω — position, for a velocity sensor — are never touched, and the estimate does not in general converge. The covariance is what fixes this.

15.3.3 Prediction grows the ellipse; the update is the Mahalanobis-closest point

STATEMENT (§8.2.3)

$P(k{+}1 \mid k) = FP(k \mid k)F^\mathsf{T} + V$. With noiseless outputs, the most likely point of Ω under the predicted Gaussian is reached by $\Delta x = PH^\mathsf{T}(HPH^\mathsf{T})^{-1}\nu$, and $P(k{+}1 \mid k{+}1) = P - RHP$ with $R = PH^\mathsf{T}(HPH^\mathsf{T})^{-1}$.

DERIVATION **Covariance prediction and the skewed projection**

$\Delta x = PH^\mathsf{T}(HPH^\mathsf{T})^{-1}\nu$

Step 1 — expand the definition. $P(k{+}1 \mid k) = \mathbb{E}[(x(k{+}1) - \hat{x}(k{+}1 \mid k))(\cdot)^\mathsf{T}]$. Substitute $x(k{+}1) = Fx + Gu + v$ and $\hat{x}(k{+}1 \mid k) = F\hat{x} + Gu$; the input cancels and the error is $F(x - \hat{x}) + v$.
Step 2 — the cross terms vanish. $v(k)$ is zero-mean and independent of both $x(k)$ and $\hat{x}(k \mid k)$, so $\mathbb{E}[F(x - \hat{x})v^\mathsf{T}] = F\,\mathbb{E}[x - \hat{x}]\,\mathbb{E}[v]^\mathsf{T} = 0$. What remains is $F\,\mathbb{E}[(x - \hat{x})(x - \hat{x})^\mathsf{T}]F^\mathsf{T} + \mathbb{E}[vv^\mathsf{T}] = FPF^\mathsf{T} + V$. This is eq. (8.12). Note that FPF^T can shrink if F is contractive; prediction only ever *adds* uncertainty because of $+V$.
Step 3 — most likely point on Ω. The Gaussian's exponent is $-\frac{1}{2}(x - \hat{x}^-)^\mathsf{T}P^{-1}(x - \hat{x}^-)$, so maximizing it over Ω means minimizing $\|\Delta x\|_M$ subject to $\hat{x}^- + \Delta x \in \Omega$. Replace Euclidean orthogonality by $\langle a, \Delta x\rangle_M = a^\mathsf{T}P^{-1}\Delta x = 0$ for all $a \in \text{null}(H)$.
Step 4 — the column space moves. $P^{-1}\Delta x \in \text{column}(H^\mathsf{T})$, hence $\Delta x \in \text{column}(PH^\mathsf{T})$: $\Delta x = PH^\mathsf{T}\gamma$. Figure 8.3 draws this: the Mahalanobis-equidistant ellipses around the prediction grow until one touches Ω, and the touch point is not under the prediction unless P is a multiple of the identity.
Step 5 — solve as before. Guess $\gamma = K\nu$, enforce $H\Delta x = \nu$: $HPH^\mathsf{T}K\nu = \nu$, so $K = (HPH^\mathsf{T})^{-1}$ and $R = PH^\mathsf{T}(HPH^\mathsf{T})^{-1}$ — eq. (8.21). The covariance update $P^+ = P - RHP$, eq. (8.15), is Choset's problem 7 and this chapter's Exercise 1.
Why it is still not enough. Noiseless outputs make P^+ singular in the measured directions — exactly zero uncertainty along H — and a singular P makes the Mahalanobis norm meaningless on the next step. The last ingredient is sensor noise.

15.3.4 The Kalman filter: most likely output, then project

STATEMENT (§8.2.4)

With $w \sim \mathcal{N}(0, W)$: $R = P^-H^\mathsf{T}(HP^-H^\mathsf{T} + W)^{-1}$, $\hat{x}^+ = \hat{x}^- + R\nu$, $P^+ = P^- - RHP^-$.

DERIVATION **From Ω to Ω^*: Theorem 8.2.1 and the final gain**

$R = P^-H^\mathsf{T}(HP^-H^\mathsf{T} + W)^{-1}$

Step 1 — project the prediction into output space. The state Gaussian $(\hat{x}^-, P^-)$ maps under H to an output-space Gaussian with mean $\hat{y} = H\hat{x}^-$ and covariance $\hat{W} = HP^-H^\mathsf{T}$.
Step 2 — the measurement is a second Gaussian there. The reading $y(k{+}1)$ carries noise of covariance W, so the sensor's claim about the output is $\mathcal{N}(y, W)$. The two are independent.
Step 3 — Theorem 8.2.1 (product of Gaussians). The product of Gaussians with pairs (z_1, C_1) and (z_2, C_2) is proportional to a Gaussian with $z_3 = z_1 + Q(z_2 - z_1)$, $C_3 = C_1 - QC_1$, $Q = C_1(C_1 + C_2)^{-1}$. The proof (Choset's problem 8, Exercise 2 here) multiplies the exponentials and matches the terms quadratic and linear in z.
Step 4 — the most likely output. Apply it with $(z_1, C_1) = (\hat{y}, \hat{W})$ and $(z_2, C_2) = (y, W)$: the peak is $y^* = \hat{y} + \hat{W}(\hat{W} + W)^{-1}(y - \hat{y})$. Define $\Omega^* = \{x : Hx = y^*\}$ — Figure 8.4's second line.
Step 5 — project onto Ω^* as before. The noiseless result of the previous derivation gives $\Delta x = P^-H^\mathsf{T}(HP^-H^\mathsf{T})^{-1}(y^* - H\hat{x}^-)$. Substitute $y^* - \hat{y} = \hat{W}(\hat{W} + W)^{-1}\nu$ with $\hat{W} = HP^-H^\mathsf{T}$: the two inverses collapse,

$$\Delta x = P^-H^\mathsf{T}\,(HP^-H^\mathsf{T})^{-1}\,HP^-H^\mathsf{T}\,(HP^-H^\mathsf{T} + W)^{-1}\,\nu = P^-H^\mathsf{T}\,(HP^-H^\mathsf{T} + W)^{-1}\,\nu.$$

That is eq. (8.24) with R as stated, and $P^+ = P^- - RHP^-$ follows as in problem 7.
Two honesty notes Choset insists on. For Gaussian noise these are not merely the "best" estimates but the *correct* posterior: if the belief at k is Gaussian, so is the belief at $k{+}1$, and this is it. For non-Gaussian noise the same equations give the best *linear* estimator, and a nonlinear one may do better. The sister book reaches the same formulas by completing the square — see its Chapter 6 for that route, the information form, and the proof the two coincide.

15.3.5 The Kalman filter is a Luenberger observer with a scheduled gain

This is the chapter's lens, and Choset implies it across §8.2 and Appendix J without stating it.

STATEMENT

Substituting the prediction into the update,

$$\hat{x}(k{+}1 \mid k{+}1) = F\hat{x}(k \mid k) + Gu(k) + R(k{+}1)\big[\, y(k{+}1) - H\big(F\hat{x}(k \mid k) + Gu(k)\big)\big],$$

a Luenberger observer whose gain $R(k{+}1)$ is recomputed from P each step. With any *fixed* gain K the error obeys $e(k{+}1) = (I - KH)F\,e(k) + (I - KH)v(k) - Kw(k)$, stable iff every eigenvalue of $(I - KH)F$ lies inside the unit disc (Thm. J.5.1). Among all gains, the Kalman gain is the one minimizing $\operatorname{tr} P^+$ at each step.

DERIVATION **Error dynamics of the fixed-gain observer**

$e(k+1) = (I - KH)Fe(k) + noise$

Step 1 — write the observer. $\hat{x}^+ = F\hat{x} + Gu + K(y - H(F\hat{x} + Gu))$.
Step 2 — subtract from the true dynamics. $x^+ = Fx + Gu + v$ and $y = Hx^+ + w = H(Fx + Gu + v) + w$. The inputs cancel.
Step 3 — collect the error $e = x - \hat{x}$: $e^+ = Fe + v - K\big(HFe + Hv + w\big) = (I - KH)Fe + (I - KH)v - Kw$.
Step 4 — read off stability. The noise-free part is the unforced linear system $e^+ = (I - KH)Fe$; by Thm. J.5.1 it decays iff $|\lambda_i| < 1$ for every eigenvalue of $(I - KH)F$. Choosing K is choosing where those eigenvalues sit — Thm. J.3.2 transposed. For a single output this is Ackermann's formula on the pair (F, HF), which `placeObserverGain` implements, and which *fails* exactly when (F, H) is unobservable: for the velocity-sensor rail robot the position eigenvalue is pinned at 1 and no K moves it.
Step 5 — the filter's choice. Take the error covariance of the fixed-gain observer under the noise model, in the form valid for *any* K (the Joseph form): $P^+(K) = (I - KH)P^-(I - KH)^\mathsf{T} + KWK^\mathsf{T}$. Differentiating $\operatorname{tr} P^+(K)$ in K and setting it to zero gives $K = P^-H^\mathsf{T}(HP^-H^\mathsf{T} + W)^{-1}$ — the Kalman gain. For a scalar state with $P^- = 1.7$, $H = 1$, $W = 0.5$ a brute-force search over K lands on $0.7727 = 1.7/2.2$ with $P^+ = 0.3864$; the check file does this search so you need not trust the calculus.
From knob to schedule. Iterate predict and update on P alone and, for an observable pair, P^- converges to the fixed point of the Riccati recursion; the gain converges with it to a constant R_∞. That constant is a perfectly good Luenberger gain — the Bench's knob is scaled so that $g = 1$ is R_∞. For the position sensor, $R_\infty = [0.589, 0.287]^\mathsf{T}$ and the error eigenvalues are $0.634 \pm 0.097i$, modulus 0.641. For the velocity sensor P never converges but R still does, to $[0.500, 0.358]^\mathsf{T}$, with eigenvalues $\{1, 0.642\}$ — the 1 is the position direction nobody measures.

15.3.6 Observability: the rank test

THEOREM 8.2.2 (OBSERVABILITY TEST)

The LTI system $x(k{+}1) = Fx(k) + Gu(k) + v(k)$, $y(k) = Hx(k) + w(k)$ is observable iff

$$Q = \begin{bmatrix} H \\ HF \\ HF^2 \\ \vdots \\ HF^{n-1} \end{bmatrix} \quad \text{has rank } n.$$

For any observable linear system the Kalman estimate converges in the sense that the expected error between the actual and estimated state is bounded for all time.

DERIVATION Why stacking powers of F decides everything

$rankQ = n \Leftrightarrow x(0) recoverable$

Step 1 — stack the outputs. With zero noise, $y(0) = Hx(0)$, $y(1) = HFx(0) + HGu(0)$, $y(2) = HF^2x(0) +$ known input terms, and so on. Move the known input terms to the left: the stacked vector $\tilde{y} = Q\,x(0)$.
Step 2 — recoverability is injectivity. $x(0)$ is determined by $\tilde{y}$ iff Q has a trivial null space, i.e. rank n. Knowing $x(0)$ and the inputs, every later state follows.
Step 3 — higher powers add nothing. By Cayley–Hamilton F^n is a linear combination of $I, F, \ldots, F^{n-1}$, so rows HF^k for $k \geq n$ lie in the span of those already stacked.
Step 4 — duality. (F, H) is observable iff $(F^\mathsf{T}, H^\mathsf{T})$ is controllable (App. J.5.2), which is why the same rank test, transposed, appears in Chapter 20 as the controllability matrix.
The dead-reckoning case. Velocity sensor: $Q = [0\ 1;\ 0\ 1]$, rank 1. The null direction is $[1, 0]^\mathsf{T}$ — position — and the position variance grows without bound through optimal updates. Choset's verdict: "This failure is not the fault of the Kalman filter ... The problem instead lies with the system itself." Position sensor: $Q = [1\ 0;\ 1\ 0.5]$, rank 2, and both variances settle.
For nonlinear systems the test is applied to the linearized pair $(\partial f/\partial x, \partial h/\partial x)$ at the current estimate and is only *local* — true at this pose, this input, these landmarks. The Observability Probe recomputes it every frame for that

reason.

15.3.7 Two hand-checked steps

Choset's §8.2.6 robot, with his numbers: $T = 0.5$, $m = 1$, $F = \left[\begin{smallmatrix} 1 & 0.5 \\ 0 & 1 \end{smallmatrix}\right]$, $G = [0,\ 0.5]^\mathsf{T}$, $u = 0$, $V = \left[\begin{smallmatrix} 0.2 & 0.05 \\ 0.05 & 0.1 \end{smallmatrix}\right]$, $H = [0\ \ 1]$, $W = 0.5$, $\hat{x}(k \mid k) = [2, 4]^\mathsf{T}$, $P(k \mid k) = \mathrm{diag}(1, 2)$. He reports no measurement value; this chapter fixes $y(k{+}1) = 2.7$ and $y(k{+}2) = 2.0$. These two readings are ours, not his.

Step 1. Predict: $\hat{x}^- = F[2, 4]^\mathsf{T} = [4, 4]^\mathsf{T}$ and $P^- = FPF^\mathsf{T} + V = \left[\begin{smallmatrix} 1.7 & 1.05 \\ 1.05 & 2.1 \end{smallmatrix}\right]$. Update: $\nu = 2.7 - 4 = -1.3$, $S = 2.1 + 0.5 = 2.6$, $R = [1.05,\ 2.1]^\mathsf{T}/2.6 = [0.4038,\ 0.8077]^\mathsf{T}$, so $\nu/S = -0.5$ exactly and

$$\hat{x}^+ = [4,\ 4]^\mathsf{T} + R\,(-1.3) = [3.475,\ 2.95]^\mathsf{T}, \qquad P^+ = P^- - R\,[1.05\ \ 2.1] = \begin{bmatrix} 1.2760 & 0.2019 \\ 0.2019 & 0.4038 \end{bmatrix}.$$

Step 2. Predict: $\hat{x}^- = [4.95,\ 2.95]^\mathsf{T}$, $P^- = \left[\begin{smallmatrix} 1.7788 & 0.4538 \\ 0.4538 & 0.5038 \end{smallmatrix}\right]$. Update: $\nu = -0.95$, $S = 1.0038$, $R = [0.4521,\ 0.5019]^\mathsf{T}$, $\hat{x}^+ = [4.520,\ 2.473]^\mathsf{T}$, $P^+ = \left[\begin{smallmatrix} 1.5737 & 0.2261 \\ 0.2261 & 0.2510 \end{smallmatrix}\right]$.

Read the diagonal. The velocity variance goes $2 \to 0.404 \to 0.251$: the squish of Figure 8.7. The position variance goes $1 \to 1.276 \to 1.574$: growing, through two *optimal* updates, because the sensor never sees position. That is Theorem 8.2.2 in two numbers.

WORK IT OUT

Carry the example one more step with y(k+3) = 2.3. What is the position variance P_{11}(k+3 | k+3)?

Answer. 1.873 Predict: $P^-_{11} = 1.5737 + 2(0.5)(0.2261) + 0.25(0.2510) + 0.2 = 2.0625$, $P^-_{12} = 0.2261 + 0.5(0.2510) + 0.05 = 0.4015$, $P^-_{22} = 0.2510 + 0.1 = 0.3510$. Update: $S = 0.3510 + 0.5 = 0.8510$, $R_1 = 0.4015/0.8510 = 0.4719$, and $P^+_{11} = 2.0625 - 0.4719 \times 0.4015 = 1.8730$. Larger again than last step's 1.5737: the optimal update recovers some of what prediction added, never all of it, and never enough to turn the trend. The velocity variance, meanwhile, falls to 0.2062.

15.3.8 EKF for range-and-bearing localization

STATEMENT (§8.3, §8.3.1)

For $x = [x_r, y_r, \theta_r]^\mathsf{T}$ with $f(x, u) = [x_r + u_1 \cos\theta_r,\ y_r + u_1 \sin\theta_r,\ \theta_r + u_2]^\mathsf{T}$ and $h_j(x) = [\rho_j,\ \mathrm{atan2}(y_r - y_{\ell j},\ x_r - x_{\ell j}) - \theta_r]^\mathsf{T}$, the EKF is the Kalman filter with $F = \partial f/\partial x$ at $\hat{x}(k \mid k)$ and $H_j = \partial h_j/\partial x$ at $\hat{x}(k{+}1 \mid k)$ (eqs. 8.37–8.45).

DERIVATION **Linearizing the unicycle and the landmark sensor**

FandH_jbyfirst − orderTaylor

Step 1 — Taylor about the estimate. Expand f about $\hat{x}(k \mid k)$ and h about $\hat{x}(k{+}1 \mid k)$ to first order; the Kalman equations then apply to the deviation. Nothing else changes: the EKF is eqs. (8.26)–(8.32) with f, h for the means and their Jacobians for the covariances.
Step 2 — the motion Jacobian.

$$F = \begin{bmatrix} 1 & 0 & -u_1 \sin\theta_r \\ 0 & 1 & u_1 \cos\theta_r \\ 0 & 0 & 1 \end{bmatrix}.$$

The third column is the whole story of heading error: a small $\delta\theta$ becomes a position error $u_1\,\delta\theta$ sideways to the motion, every step.
Step 3 — the sensor Jacobian. With $\Delta = \hat{x} - x_{\ell_j}$ and $\rho = \|\Delta\|$,

$$H_j = \begin{bmatrix} \Delta x/\rho & \Delta y/\rho & 0 \\ -\Delta y/\rho^2 & \Delta x/\rho^2 & -1 \end{bmatrix}.$$

Choset prints the bearing row before simplifying, as the raw derivative of atan2 through $1/(1 + (\Delta y/\Delta x)^2)$; the check file keeps both forms and confirms they agree to 10^{-15}, and both agree with a central-difference Jacobian to 10^{-9}. Read the second row: bearing sensitivity to position falls off as $1/\rho$ while its sensitivity to heading is exactly -1 at any range. A distant landmark is nearly a pure heading sensor.
Step 4 — stack the visible landmarks. $p(k)$ sightings give a $2p \times 3$ matrix H by stacking $H_{a(1)}, \ldots, H_{a(p)}$ and a block-diagonal W. For block-diagonal W this equals applying the sightings one at a time.
Step 5 — range-only is a deletion. Drop the bearing rows from h and H (§8.3.3). The observability consequences are the subject of the Probe below.
Wrap the bearing innovation. The 2005 text subtracts angles; near $\pm\pi$ that produces an innovation of 2π and a catastrophic update. The implementation here takes the residual through `angleDiff` and the heading through ⊞. The sister book's Chapter 7 does this properly — the UKF and the error-state EKF on SE(2) — and its Chapter 11 covers gating and multi-hypothesis tracking when association is genuinely ambiguous.

15.4 The algorithm

Choset numbers none of Chapter 8's algorithms; the names below follow his section titles.

ALGORITHM `kalman_filter(x̂(k|k), P(k|k), u(k), y(k+1); F, G, H, V, W)` COST `O(n³) in general; O(n²p + p³) when p ≪ n`

in the current belief, the input applied, the output observed, the model

out $\hat{x}(k+1 \mid k+1)$, $P(k+1 \mid k+1)$, and (ν, S, R) for association and diagnostics

1: $\hat{x}(k+1 \mid k) = F\hat{x}(k \mid k) + Gu(k)$ — eq. 8.26
2: $P(k+1 \mid k) = FP(k \mid k)F^{\mathsf{T}} + V$ — eq. 8.27
3: $\nu = y(k+1) - H\hat{x}(k+1 \mid k)$ — eq. 8.30
4: $S = HP(k+1 \mid k)H^{\mathsf{T}} + W$ — eq. 8.31
5: $R = P(k+1 \mid k)H^{\mathsf{T}}S^{-1}$ — eq. 8.32
6: $\hat{x}(k+1 \mid k+1) = \hat{x}(k+1 \mid k) + R\nu$ — eq. 8.28
7: $P(k+1 \mid k+1) = P(k+1 \mid k) - RHP(k+1 \mid k)$ — eq. 8.29; in code, the Joseph form, which equals this at the optimal gain and stays positive-definite under round-off
8: return $\hat{x}(k+1 \mid k+1)$, $P(k+1 \mid k+1)$, (ν, S, R)

Choset's reading of R: if R is "large" the sensor is more believable than the prediction and the update leans on it; if "small", the reading barely moves the estimate. Line 5 says exactly how large — the ratio of predicted uncertainty to total innovation uncertainty, in matrix form.

ALGORITHM `observability_rank(F, H)` COST `one rank computation on an np × n matrix`

in the LTI pair, or the Jacobian pair of a nonlinear system at the current estimate

out rank Q, and a basis of null(Q): the directions no output can correct

1: $Q \leftarrow H$
2: for $k = 1, \ldots, n-1$: append HF^k to Q
3: return rank Q under a relative tolerance; observable iff it equals n — Thm. 8.2.2

ALGORITHM `ekf_localize(x̂, P, u, {y_i}; f, h, ∂f, ∂h, V, W)` COST as `kalman_filter plus the Jacobians; association O(n_ℓ p³) per sighting`

in belief, input, the sightings this step, the nonlinear models and their Jacobians

out updated belief and the association map a(·)

1: $\hat{x}^- = f(\hat{x}, u)$; $F = \partial f/\partial x|_{\hat{x}}$; $P^- = FPF^\mathsf{T} + V$ — eqs. 8.37–8.39
2: for each sighting y_i and each landmark j: $\nu_{ij} = y_i - h_j(\hat{x}^-)$, $S_{ij} = H_j P^- H_j^\mathsf{T} + W_i$, $\chi^2_{ij} = \nu_{ij}^\mathsf{T} S_{ij}^{-1} \nu_{ij}$ — §8.3.2
3: $a(i) = \arg\min_j \chi^2_{ij}$; reject the sighting if the minimum exceeds a gate, found a new landmark if it exceeds a high threshold (§8.4.2)
4: stack $h_{a(i)}$, $H_{a(i)}$ over accepted sightings; wrap bearing residuals
5: one `kalman_filter` update with the stacked (H, W) — eqs. 8.40–8.45
6: return $\hat{x}^+, P^+, a$

Choset minimizes χ^2_{ij} alone. When the candidates' gates differ in size this is not quite maximum likelihood: the $\log\det S_{ij}$ term matters, and the sister's `mlAssociate`, which the range–bearing path here delegates to, keeps it. The range-only path uses Choset's rule verbatim because the sister's routine assumes a bearing row.

15.4.1 The ellipse that never shrinks

Now the question the Probe was built to answer. Rusty drives a square in room A of the Apartment while the EKF above localizes it against known landmarks. How many does it need?

INTERACTIVE · w15.2

Observability Probe

More iterations do not mean more certainty. Along a direction the sensors cannot see, the optimal filter is optimally uncertain — forever.

Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch15-kalman-observer#w15.2`

With one range-only landmark the rank of the linearized pair is 2. The single range row and its products with F all share the same (x, y) direction — radial from the landmark — so the direction *tangent* to the constant-range circle never appears in Q. The posterior ellipse stretches along that circle and stays stretched: after 240 steps its trace is 0.482 against 0.034 with two landmarks, an order of magnitude, and no number of further iterations closes the gap. Switch to two non-collinear landmarks and the rank is 3; put both landmarks and the robot on one line and it falls to 2 again (Exercise 4). Range *and* bearing from one landmark is observable while the robot moves — the third column of HF carries u_1 — and loses a rank the moment it stops: two numbers cannot pin three. The NEES tile

reads near 3 for the observable cases, which is what a consistent 3-DOF filter should report (sister Ch. 6 on NEES and NIS).

15.4.2 Kalman SLAM

If the landmarks are unknown, put them in the state (§8.4; Smith, Self and Cheeseman). For an omnidirectional vehicle measuring the *relative displacement* to n_ℓ uniquely identified landmarks, the system is linear: $x = [x_r, y_r, x_{\ell 1}, y_{\ell 1}, \dots]^\mathsf{T}$, $F = I$, $G = [I_2; 0]$, $V = \text{blkdiag}(V_r, 0, \dots)$, and each H_i has -1 in the robot columns and $+1$ in landmark i's. The sister's plain `Kf` runs it unchanged. Its observability matrix has rank 4 of 6 for two landmarks: the two null directions shift robot and map together. The filter learns the map's *shape* — in the check, the covariance of $x_{\ell 1} - x_r$ falls monotonically from trace 0.079 to 0.020 — while the map's absolute covariance stalls at 0.059 and goes no lower. No sighting can say where the world is.

For range and bearing on the unicycle (§8.4.2) the state is $[x_r, y_r, \theta_r, x_{\ell 1}, \dots]$ and H_i is eq. (8.47): dense in the first three columns, two nonzero entries at landmark i, zero elsewhere. Association is eq. (8.48) against every landmark in the state, and a minimum χ^2 above a high threshold founds a new one. The *forget the map* toggle in the Probe runs this: three landmarks founded from their first sightings, 588 of the remaining 597 sightings matched, the worst map error 0.12 m at 1.6σ. Honest caveats: the covariance update is $O(n_\ell^2)$ per step, and on long loops the linearization makes the filter inconsistent. The sister book's Chapter 14 shows the cost and the inconsistency, and its Chapter 15 shows what replaced it.

15.5 Implementation in Rust

The `estimate` crate is thin by design: every file is either an adapter over the sister book's vendored filters or a piece Choset has and the sister does not. No filter is re-implemented.

RUST **crates/estimate/src/lti.rs**

```rust
use nalgebra::{DMatrix, SMatrix};

/// An LTI system in Choset's letters. Every field names the Thrun symbol the
/// sister book uses, because the two conventions collide in the worst place:
/// Choset's `R` is the Kalman *gain*; Thrun's `R` is the process *covariance*.
pub struct Lti<const N: usize, const U: usize, const M: usize> {
    pub f: SMatrix<f64, N, N>, // A_t
    pub g: SMatrix<f64, N, U>, // B_t
    pub h: SMatrix<f64, M, N>, // C_t, assumed full row rank
    pub v: SMatrix<f64, N, N>, // R_t  (process covariance)
    pub w: SMatrix<f64, M, M>, // Q_t  (measurement covariance)
}

impl<const N: usize, const U: usize, const M: usize> Lti<N, U, M> {
    /// App. J.5: F = I + T A, G = T B, H = C.
    pub fn discretize(a: SMatrix<f64, N, N>, b: SMatrix<f64, N, U>, c: SMatrix<f64, M, N>,
```

```
    t: f64,
                        v: SMatrix<f64, N, N>, w: SMatrix<f64, M, M>) -> Self {
        Self { f: SMatrix::identity() + t * a, g: t * b, h: c, v, w }
    }

    /// Thm. 8.2.2: Q = [H; HF; ...; HF^{N-1}]. Dynamic because N.M is not a
    /// const expression -- the one place this crate leaves the stack.
    pub fn observability_matrix(&self) -> DMatrix<f64> {
        let mut q = DMatrix::zeros(N * M, N);
        let mut hf = self.h;
        for k in 0..N {
            q.view_mut((k * M, 0), (M, N)).copy_from(&hf);
            hf = hf * self.f; // Cayley-Hamilton says stopping at N-1 loses nothing
        }
        q
    }

    /// rank Q == N under a *relative* SVD tolerance -- a parameter, not a magic
    /// number: a 0/1 matrix needs none, a linearized Jacobian mixing metres
    /// and radians needs a generous one.
    pub fn is_observable(&self, tol: f64) -> bool {
        self.observability_matrix().rank(tol) == N
    }
}
```

The observer is the knob. Its `step` is three lines; `error_eigs` is the unit-disc inset, and `place_gain` is Theorem J.3.2 transposed — Ackermann's formula on the pair (F, HF), which is why it can only succeed when (F, H) is observable.

RUST `crates/estimate/src/observer.rs`

```
use nalgebra::{Complex, SMatrix, SVector};
use crate::lti::Lti;

/// Fixed-gain Luenberger observer, App. J.4 discretized by J.5:
///   x? = F x + G u + K (y - H(F x + G u)).
pub struct Observer<const N: usize, const U: usize, const M: usize> {
    pub sys: Lti<N, U, M>,
    pub k: SMatrix<f64, N, M>,
    pub xhat: SVector<f64, N>,
}

impl<const N: usize, const U: usize, const M: usize> Observer<N, U, M> {
    pub fn step(&mut self, u: &SVector<f64, U>, y: &SVector<f64, M>) {
        let predicted = self.sys.f * self.xhat + self.sys.g * u;
        let nu = y - self.sys.h * predicted; // the innovation: what the sensor saw minus
    what it should have
        self.xhat = predicted + self.k * nu;
    }

    /// e(k+1) = (I - K H) F e(k): inside the unit disc ? the error decays (Thm. J.5.1).
    pub fn error_matrix(&self) -> SMatrix<f64, N, N> {
        (SMatrix::identity() - self.k * self.sys.h) * self.sys.f
    }

    pub fn error_eigs(&self) -> Vec<Complex<f64>> {
        self.error_matrix().complex_eigenvalues().iter().copied().collect()
    }
```

```rust

    /// The error covariance a fixed gain *actually* produces (Joseph form) --
    /// valid for any K, not only the optimal one. The Kalman gain is the
    /// argmin of its trace; see `kf::kalman_step`, which never calls this.
    pub fn covariance_update(&self, p: &SMatrix<f64, N, N>) -> SMatrix<f64, N, N> {
        let ikh = SMatrix::<f64, N, N>::identity() - self.k * self.sys.h;
        ikh * p * ikh.transpose() + self.k * self.sys.w * self.k.transpose()
    }
}
```

The Kalman step itself is an adapter. `filters::kalman::Kf` is the sister book's filter, vendored unchanged; this function only renames its inputs into Choset's letters and hands back (ν, S, R) because data association and the widgets need them.

RUST **crates/estimate/src/kf.rs**

```rust
pub use filters::kalman::{Gaussian, Kf}; // sister Ch. 6 -- not re-derived, not re-implemented
use nalgebra::{SMatrix, SVector};
use crate::lti::Lti;

/// One Choset step, eqs. 8.26-8.32, through the sister's `Kf`.
pub fn kalman_step<const N: usize, const U: usize, const M: usize>(
    sys: &Lti<N, U, M>, bel: &mut Gaussian<N>, u: &SVector<f64, U>, y: &SVector<f64, M>,
) -> (SVector<f64, M>, SMatrix<f64, M, M>, SMatrix<f64, N, M>) {
    let mut kf = Kf::from(bel.clone());
    kf.predict(&sys.f, &sys.v, Some((&sys.g, u)));            // Thrun: A, R, (B, u)
    let info = kf.update(y, &sys.h, &sys.w);                   // Thrun: C, Q
    *bel = kf.belief();
    (info.innovation, info.s, info.k)                          // Choset: nu, S, R
}

/// sec.8.2.6 constants: T = 0.5, m = 1, V, W = 0.5, x = [2, 4], P = diag(1, 2).
pub fn dead_reckoning() -> (Lti<2, 1, 1>, Gaussian<2>) {
    let (t, m) = (0.5, 1.0);
    let sys = Lti {
        f: SMatrix::<f64, 2, 2>::new(1.0, t, 0.0, 1.0),
        g: SMatrix::<f64, 2, 1>::new(0.0, t / m),
        h: SMatrix::<f64, 1, 2>::new(0.0, 1.0),                // velocity sensor
        v: SMatrix::<f64, 2, 2>::new(0.2, 0.05, 0.05, 0.1),
        w: SMatrix::<f64, 1, 1>::new(0.5),
    };
    (sys, Gaussian { x: SVector::<f64, 2>::new(2.0, 4.0), p: SMatrix::from_diagonal(&[1.0, 2.0].into()) })
}
```

RUST **crates/estimate/examples/dead_reckoning.rs**

```rust
use estimate::{kf::{dead_reckoning, kalman_step}, lti::Lti};
use nalgebra::{SMatrix, SVector};

fn main() {
    let (sys, mut bel) = dead_reckoning();
```

```
    for (k, y) in [2.7, 2.0].into_iter().enumerate() {
        let (nu, s, r) = kalman_step(&sys, &mut bel, &SVector::zeros(), &SVector::<f64,
   1>::new(y));
        println!("step {}: nu = {:+.3}  S = {:.4}  R = [{:.4}, {:.4}]", k + 1, nu[0], s
   [(0, 0)], r[(0, 0)], r[(1, 0)]);
        println!("        x? = [{:.3}, {:.3}]   P? = [[{:.4}, {:.4}], [{:.4}, {:.4}]]",
            bel.x[0], bel.x[1], bel.p[(0, 0)], bel.p[(0, 1)], bel.p[(1, 0)], bel.p[(1, 1)
   ]);
    }
    let pos = Lti { h: SMatrix::<f64, 1, 2>::new(1.0, 0.0), ..sys };
    println!("rank Q = {} (velocity sensor), {} (position sensor)",
        sys.observability_matrix().rank(1e-9), pos.observability_matrix().rank(1e-9));
}
```

```
step 1: nu = -1.300  S = 2.6000  R = [0.4038, 0.8077]
        x? = [3.475, 2.950]   P? = [[1.2760, 0.2019], [0.2019, 0.4038]]
step 2: nu = -0.950  S = 1.0038  R = [0.4521, 0.5019]
        x? = [4.520, 2.473]   P? = [[1.5737, 0.2261], [0.2261, 0.2510]]
rank Q = 1 (velocity sensor), 2 (position sensor)
```

RUST `crates/estimate/src/kf.rs (tests)`

```rust
#[test]
fn reproduces_choset_8_2_6_two_steps() {
    let (sys, mut bel) = dead_reckoning();
    kalman_step(&sys, &mut bel, &SVector::zeros(), &SVector::<f64, 1>::new(2.7));
    assert_relative_eq!(bel.x, SVector::<f64, 2>::new(3.475, 2.95), epsilon = 1e-3);
    kalman_step(&sys, &mut bel, &SVector::zeros(), &SVector::<f64, 1>::new(2.0));
    assert_relative_eq!(bel.p[(0, 0)], 1.5737, epsilon = 1e-3);
    assert_relative_eq!(bel.p[(1, 1)], 0.2510, epsilon = 1e-3);
    assert_eq!(sys.observability_matrix().rank(1e-9), 1);
}

/// Derivation 4 made executable: an `Observer` whose gain is set to R(k) each
/// step reproduces the Kalman mean bit-for-bit.
#[test]
fn kf_is_observer_with_scheduled_gain() {
    let (sys, mut bel) = dead_reckoning();
    let mut obs = Observer { sys: sys.clone(), k: SMatrix::zeros(), xhat: bel.x };
    for y in [2.7, 2.0, 2.4, 1.9] {
        let (_, _, r) = kalman_step(&sys, &mut bel, &SVector::zeros(), &SVector::<f64,
   1>::new(y));
        obs.k = r;
        obs.step(&SVector::zeros(), &SVector::<f64, 1>::new(y));
        assert_eq!(obs.xhat, bel.x);
    }
}
```

The landmark models carry Choset's conventions — the bearing is measured *from* the landmark, a quirk the translation into the sister's `Feature` type absorbs with one rotation by π.

RUST `crates/estimate/src/landmarks.rs`

```rust
use filters::nonlinear::MeasurementModel; // sister Ch. 7
use geom::SE2;
use nalgebra::{SMatrix, SVector};
use sim::Landmark;

/// h_j = [rho, atan2(y_r - y_l, x_r - x_l) - theta_r]; rows of ?h/?x: [Deltax/rho,
   Deltay/rho, 0], [-Deltay/rho2, Deltax/rho2, -1].
pub struct RangeBearing<'m> { pub map: &'m [Landmark] }
/// sec.8.3.3: the same model with the bearing row removed.
pub struct RangeOnly<'m> { pub map: &'m [Landmark] }

impl MeasurementModel<SE2, 3, 2> for RangeBearing<'_> {
    fn h(&self, x: &SE2, j: usize) -> SVector<f64, 2> {
        let (dx, dy) = (x.x() - self.map[j].x, x.y() - self.map[j].y);
        SVector::new((dx * dx + dy * dy).sqrt(), geom::wrap(dy.atan2(dx) - x.theta()))
    }
    fn jacobian(&self, x: &SE2, j: usize) -> SMatrix<f64, 2, 3> {
        let (dx, dy) = (x.x() - self.map[j].x, x.y() - self.map[j].y);
        let q = (dx * dx + dy * dy).max(1e-12);
        let rho = q.sqrt();
        SMatrix::new(dx / rho, dy / rho, 0.0, -dy / q, dx / q, -1.0)
    }
    /// The innovation wraps: the one place where z - h(x) is wrong.
    fn residual(&self, z: &SVector<f64, 2>, zhat: &SVector<f64, 2>) -> SVector<f64, 2> {
        SVector::new(z[0] - zhat[0], geom::wrap(z[1] - zhat[1]))
    }
}

/// sec.8.3.2: a(i) = argmin_j nu_ij^T S_ij-1 nu_ij. `None` above `chi2_new` => found a
   new landmark (sec.8.4.2).
pub fn associate_ml(y_i: &SVector<f64, 2>, bel: &Gaussian<3>, map: &[Landmark],
                    w: &SMatrix<f64, 2, 2>, chi2_new: f64) -> Option<usize> {
    let model = RangeBearing { map };
    let pose = SE2::from_vector(&bel.x);
    let best = (0..map.len()).map(|j| {
        let h = model.jacobian(&pose, j);
        let s = h * bel.p * h.transpose() + w;
        let nu = model.residual(y_i, &model.h(&pose, j));
        (j, (nu.transpose() * s.try_inverse().expect("S is PD") * nu)[(0, 0)])
    }).min_by(|a, b| a.1.total_cmp(&b.1))?;
    (best.1 <= chi2_new).then_some(best.0)
}
```

> **NOTE**
>
> The widgets on this page run the TypeScript port of these files, `web/lib/estimate/`, over the vendored `web/lib/filters/kf.ts` and `ekf.ts`. Thirteen checks in `__checks_ch15__.ts` pin every number printed above: the two §8.2.6 steps to 10^{-3}, both ranks, the placed eigenvalues $\{0.5, 0.2\}$ and the error ratio that converges to 0.5, the brute-force gain 0.7727, the Riccati limit, the Jacobian agreement, the linearized ranks 2/3/3/2/2, the one-versus-two-landmark traces, both SLAM instances, and Problem 9.

15.6 Putting it together

The integration lab is the two widgets above, run in sequence on the Apartment, and it ends with the object Chapter 16 inherits.

Dead reckoning. Rusty's encoders through `odometryDelta` give Choset's input $u = [\Delta s, \Delta\theta]$ each tick. `ChosetEkf::predict` alone is Figure 8.1: the orange track in the hook bends off the gray one and the ellipse grows without bound. Its claimed σ tracks the actual error only as long as the noise model is honest — the wheel-radius bias is *not* in the model, and when you raise it the error outruns the ellipse. That gap is what the sister book's NEES instrument measures, and the reason Chapter 16 will care about beliefs that are not Gaussian.

One landmark. In the Probe with one range-only landmark the rank reads 2 and the ellipse's major axis runs along the constant-range circle. After 240 steps the pose trace is 0.482. The filter is doing nothing wrong; "the problem lies with the system." A planner handed this belief should read the long axis and plan a motion that *changes the geometry* — toward a second landmark, or around the first — because no amount of sitting still will shrink it.

Two landmarks. Rank 3, trace 0.034, mean NEES 3.8 against the expected 3. This is the belief a planner wants: a Gaussian bel(x_t) with a covariance it can read, in the same SE(2) coordinates its configuration lives in.

Forget the map. Kalman SLAM founds the three landmarks from their first sightings and matches 588 of 597 later ones, with the worst landmark 0.12 m off at 1.6σ. The covariance is now 9×9 and grows by two rows per landmark; the correlations between landmarks are what let a re-sighting of one improve all the others. The state is also what makes the filter's eventual inconsistency possible, which is the sister book's story, not this one.

What a planner takes from here is small and exact: a mean in SE(2), a 3×3 covariance, an innovation covariance S for each candidate sighting, and the rank test to tell whether the next action can even reduce the uncertainty. Chapter 16 asks what happens when the belief is not Gaussian, and how to plan on it.

15.7 Exercises

1 FOUNDATION •• **The covariance update (Choset's problem 7)**

Starting from $P^+ = \mathbb{E}[(x - \hat{x}^+)(x - \hat{x}^+)^\mathsf{T}]$ and $\hat{x}^+ = \hat{x}^- + Rv$, derive $P^+ = P^- - RHP^-$. Follow Choset's three steps: expand into $P^- - 2\,\mathbb{E}[Rv(x - \hat{x}^-)^\mathsf{T}] + \mathbb{E}[Rvv^\mathsf{T}R^\mathsf{T}]$; show the middle term is $2RHP^-$; show $\mathbb{E}[vv^\mathsf{T}] = HP^-H^\mathsf{T}$ *when the output is noiseless* and mark that step — it is where §8.2.3's assumption enters, and where $+W$ appears in the §8.2.4 version. Then check that with $R = P^-H^\mathsf{T}(HP^-H^\mathsf{T} + W)^{-1}$ the Joseph form $(I - RH)P^-(I - RH)^\mathsf{T} + RWR^\mathsf{T}$ collapses to the same expression.

2 FOUNDATION •• **Product of Gaussians, and two gains that are one**

Prove Theorem 8.2.1 via Choset's problem 8: write the product of the exponents, collect the terms quadratic in z to get $C_3^{-1} = C_1^{-1} + C_2^{-1}$, the terms linear in z to get $C_3^{-1}z_3 = C_1^{-1}z_1 + C_2^{-1}z_2$, and use the matrix identity $(C_1^{-1}+C_2^{-1})^{-1} = C_1 - C_1(C_1+C_2)^{-1}C_1$ to reach $Q = C_1(C_1 + C_2)^{-1}$. Then show the sister book's gain $K = \bar{\Sigma}C^{\mathsf{T}}(C\bar{\Sigma}C^{\mathsf{T}} + Q)^{-1}$ and Choset's R coincide under the notation table's map — and say in one sentence why the sister's Q is Choset's W, not his Q.

3 CONCEPTUAL •• **Predict the schedule's limit**

PREDICT FIRST

In the Observer Bench with the default V and W and the velocity sensor, tick 'let the filter choose' and wait. Where do the two eigenvalues of (I − R(k)H)F settle?

- Both inside the disc, near 0.6
  Only the velocity eigenvalue can move. With H = [0 1], (I − KH)F is upper triangular with 1 and $1 - K_2$ on the diagonal; no gain changes the 1.
- ✓ One at exactly 1, the other near 0.64
  Position is unobservable (rank Q = 1), so its eigenvalue is pinned at 1 for every gain. The velocity eigenvalue is $1 - R_2(\infty) = 1 - 0.358 = 0.642$, read off the Riccati limit $R\infty = [0.500, 0.358]^{\mathsf{T}}$.
- Both at 1: the filter stops correcting
  The velocity sensor keeps informing velocity; its gain converges to a positive constant, not to zero.

Now switch to the position sensor. Predict which component of P shrinks and verify: both do, the eigenvalues become a complex pair of modulus 0.641, and the gain converges to $[0.589, 0.287]^{\mathsf{T}}$. Reconcile the two outcomes with the two rank computations in the text.

WORK IT OUT

With the velocity sensor and Choset's V, W, what is the steady-state velocity gain $R_2(\infty)$?

Answer. 0.358 Iterate $P^- = FPF^{\mathsf{T}} + V$, $R = P^-H^{\mathsf{T}}/(HP^-H^{\mathsf{T}} + W)$, $P = (I - RH)P^-(I - RH)^{\mathsf{T}} + RWR^{\mathsf{T}}$ from $P = \mathrm{diag}(1, 2)$. P_{22} converges even though P_{11} does not, and $R_2 = P_{22}^-/(P_{22}^- + 0.5) \to 0.358$.

4 CONCEPTUAL •• The collinear trap

PREDICT FIRST

In the Observability Probe, switch to range-only, show two landmarks, and drag them so that both landmarks and the robot's path lie on one line. What does the rank panel read while the robot is on that line?

- 3 — two landmarks are always enough
  Two range rows $[\Delta x_1/\rho_1, \Delta y_1/\rho_1, 0]$ and $[\Delta x_2/\rho_2, \Delta y_2/\rho_2, 0]$ are parallel when the landmarks are collinear with the robot; they span one direction, not two.
- ✓ 2 — the direction across the line is blind
  Both range rows point along the line. The HF rows add a heading column proportional to u_1, which buys one more rank but not two: motion across the line is never measured until the robot leaves it.
- 1 — nothing is observable
  Along the line both ranges still change with position, and the HF rows still carry heading information when $u_1 \neq 0$. Rank 1 needs the robot to also be driving along the line with no heading change.

Explain the reading with the linearized rank test: write the two range rows of H and the two rows of HF for a robot at $(2, 0.35, 0.7)$ with landmarks at $(0.35, 0.35)$ and $(3.65, 0.35)$, and show the four rows span only two directions. The check file computes exactly this case.

5 PRACTICAL •• The steady-state gain

Implement `Lti::steady_state_gain(&self, p0, max_iters, tol) -> (SMatrix<f64, N, M>, bool)` by iterating the Riccati recursion in Joseph form until $P(k{+}1 \mid k)$ stops moving, returning the gain and whether it converged. Test that an `Observer` with that gain and the full Kalman filter agree in their means to 10^{-6} after 50 steps on the §8.2.6 system with the *position* sensor, and that with the velocity sensor the recursion reports non-convergence while the gain itself still settles. The TypeScript `riccatiSteadyState` and `gainSchedule` in `web/lib/estimate/kf.ts` are the port.

6 PRACTICAL ••• Choset's problem 9 as a test (stretch)

Implement the unicycle with a position-only sensor $y = x_1 + w$, $\mathrm{Var}\, w = 0.5$, no process noise, and run one EKF step from $\hat{x}(1 \mid 1) = [1, 0.5, \pi/4]^\mathsf{T}$ with $u = [3, \pi]$, $T = 0.25$, $P(1 \mid 1) = I$, $y(2) = 1.7$. Print $\hat{x}(2 \mid 2)$ and $P(2 \mid 2)$, then run the same step through the sister's UKF and report the gap.

WORK IT OUT

What is $\hat{x}_1(2 \mid 2)$, the corrected x-position?

Answer. 1.652 Prediction: $\hat{x}^- = [1 + 0.75\cos\frac{\pi}{4},\ 0.5 + 0.75\sin\frac{\pi}{4},\ \frac{\pi}{2}] = [1.5303, 1.0303, 1.5708]$ and $P^- = FF^\mathsf{T}$ with $F_{13} = -F_{23} = -0.5303$, so $P^-_{11} = 1.28125$. Update: $S = 1.28125 + 0.5 = 1.78125$, $\nu = 0.1697$, $R_1 = 0.7193$, and $\hat{x}^+_1 = 1.5303 + 0.1221 = 1.6524$; the full posterior is $[1.6524, 1.0035, 1.5203]^\mathsf{T}$ with $P^+_{11} = 0.3596$.

HINTS

<>The input enters as a displacement: Choset's f uses Tu_1 and Tu_2, so $u_1T = 0.75$ m and $u_2T = \pi/4$.</> <>With $P = I$ and no process noise, $P^- = FF^\mathsf{T}$; only the third column of F is off the identity, so P^- has $1 + F^2_{13}$ on the first two diagonal entries and $F_{13}F_{23}$ off them.</> <>$H = [1\ 0\ 0]$ picks out the first row of P^-: $R = P^-_{\cdot 1}/S$.</>

15.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

Chapter 8 is this chapter's source: the observer-first derivation (§8.2.2–8.2.4), the dead-reckoning example (§8.2.6), Theorem 8.2.2, the EKF Jacobians (§8.3), and Kalman SLAM (§8.4). Appendix J supplies the Luenberger observer and the rank tests. `link`

Kalman, R. E. (1960). A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering 82(1), 35–45*.

The original. Kalman's derivation is by orthogonal projection in the Hilbert space of random variables — the same geometry Choset draws in state space. `doi:10.1115/1.3662552`

Luenberger, D. G. (1971). An Introduction to Observers. *IEEE Transactions on Automatic Control 16(6), 596–602*.

The observer as a copy of the plant with a correcting term, and the error dynamics (A − KC)e that this chapter's Bench puts on the unit disc. `doi:10.1109/TAC.1971.1099826`

Thrun, S., Burgard, W., and Fox, D. (2005). Probabilistic Robotics. *MIT Press*.
The Bayes-filter derivation of the same equations, and the convention (K for the gain, R and Q for the covariances) the vendored code speaks. Chapters 3, 7 and 10 of this book are the sister volume's Chapters 6, 11 and 14. link

Bar-Shalom, Y., Li, X. R., and Kirubarajan, T. (2001). Estimation with Applications to Tracking and Navigation. *Wiley*.
NEES, NIS and the χ^2 consistency tests the Probe's tiles report; §5.4 is the standard reference for reading them. doi:10.1002/0471221279

Huang, G. P., Mourikis, A. I., and Roumeliotis, S. I. (2010). Observability-based Rules for Designing Consistent EKF SLAM Estimators. *International Journal of Robotics Research 29(5), 502–528*.
Why Kalman SLAM goes inconsistent: the linearized system acquires an observable direction the true system lacks, and the filter gains information it should not. The modern coda to §8.4. doi:10.1177/0278364909353640

Shannon Dynamics (2026). Probabilistic Robotics via Rust — Chapters 5–7, 11, 14. *Sister volume, web edition*.
Completing the square, the information form and RTS smoothing (Ch. 6); UKF and the error-state EKF on SE(2) (Ch. 7); gating and multi-hypothesis tracking (Ch. 11); EKF-SLAM's cost and inconsistency (Ch. 14). The filters this chapter wraps are vendored from there. link

CHAPTER 16

Bayesian Localization and Mapping

PLANNING WITH UNCERTAINTY · Intermediate · 60 min

> *Most important, the resulting estimate may be an arbitrary distribution instead of a Gaussian.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), Chapter 9*

Chapter 15 gave the planner a Gaussian. This chapter takes it away. Rusty is switched on in the Apartment without being told where; three doors look alike; the belief has three peaks and no mean worth driving toward. Choset's Chapter 9 answers the estimation half — recursive Bayesian filtering over *any* posterior, on a grid or by particles, a physical sensor model, occupancy-grid mapping, Rao-Blackwellized SLAM — and the sister book treats every piece in depth, so what stays here is compressed and linked.

What neither book writes down is the half a *planner* needs: what do Parts II–III do when q_{start} is a distribution? Not "plan from the mean" — the mean can be inside a wall. Not "solve a POMDP" — that is the sister's Chapter 22. The answer every fielded robot runs is a loop: **sample the belief, score each first action against a cost-to-go field computed once by backward Dijkstra, take the cheapest in expectation, sense, update, repeat.** The Chapter 6 universal plan is exactly what makes planning on a belief cheap. This chapter owns one artifact, `replan_on_belief`, and the capstone's navigation panel is that artifact running.

◎ AFTER THIS CHAPTER YOU CAN

- Derive recursive Bayesian localization once (Choset 9.7–9.15), name its

two independence assumptions, and say when the second one lies
- Run the same recursion as a grid (Algorithm 16) and as a particle set (Algorithms 17–18), and read mean, mode, entropy and $\tilde{p}$ off either
- Build Choset's four-component proximity sensor model, cache it as 256 histograms, and know why its σ is wider than the laser's
- State the log-odds occupancy recursion (Algorithm 19) and the Rao-Blackwellized SLAM factorization (Algorithm 20), with the sister book carrying the machinery
- Plan on a belief: sample it, score every first action against a cost-to-go field computed once by backward Dijkstra, take the expected minimum, sense, repeat — and name that loop honestly as the QMDP approximation

ASSUMED

- Rusty, the Apartment, and the ray-cast range sensor $\rho(x, s)$ (Chapter 2) — the sensor model is a likelihood over its readings
- Search, D*, and the universal plan as backward Dijkstra (Chapter 6); the distance transform as a clearance cost (Chapter 7)
- The Gaussian belief of Chapter 15 as one representation among three; the estimate crate this chapter extends

16.1 The problem: switched on, told nothing

Rusty was last seen in room A. Someone carried it to room B and switched it on. The belief Chapter 15 would hand a planner — a tight Gaussian around the last known pose — is confident and wrong, and a confident, wrong belief is worse than no belief at all.

INTERACTIVE · f16.0

The Kidnapped Robot

A confident Gaussian is worse than no belief at all when it is wrong. A belief that admits it knows nothing can still be planned on.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch16-bayesian-localization#f16.0

On the left the planner does exactly what Part II taught it: from the believed cell it follows the cost-to-go field Up, through room A's doorway, toward the goal. In room B that doorway is a wall. The wheels slip, the odometry reports the move, the predict-only belief sails into the corridor and the chassis stays where it was. On the right a uniform particle cloud over all free space knows nothing,

which is the honest state. It moves where *every* hypothesis stays cheap, each scan kills the rooms that do not match what it sees, and it arrives. The check file pins both outcomes on the default seed: the Gaussian bumps and never reaches the goal; the cloud has its mode within half a metre of the truth inside twenty scans and arrives with at most three bumps.

Choset calls the three situations *position tracking* (initial configuration approximately known), *global localization* (unknown), and the *kidnapped robot problem* (believed wrongly, to be unlearned). The Kalman filter handles the first. This chapter is about the other two, and about what a planner can do while the belief is still a cloud.

One line of bookkeeping, carried over from Chapter 15: for Rusty the planner's configuration $q \in \mathrm{SE}(2)$ and the filter's state $x(k)$ are the same object, and a grid belief lives on the Chapter 6 lattice, so the cost-to-go J and the belief share cells. The sister book writes x_t, u_t, z_t and $\mathrm{bel}(x_t)$; this chapter keeps Choset's $P(x(k) \mid u(0{:}k{-}1), y(1{:}k))$ because his derivation reads better in it.

16.2 Building intuition

16.2.1 One recursion, two tables

Here is one recorded Apartment run — seeded odometry increments and 36-beam scans from the Chapter 2 simulator — replayed into two localizers that start uniform over free space. Left: Choset's Algorithm 16 on a grid over SE(2). Right: Algorithms 17–18 as a particle cloud. Same data, same sensor model, same recursion; only the way the posterior is written down differs.

> **INTERACTIVE · w16.1**
>
> **Grid vs Particles**
>
> The particle filter is not a cheaper grid. At equal cost the two fail differently: the grid blurs, the cloud starves, and only one of them recovers from a kidnap without being told.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch16-bayesian-localization#w16.1

Watch three things. First, the
mode and the
mean are different markers for a reason: during the bimodal phase, when rooms A and C look identical from inside, the mean (Choset's eq. 9.17, a weighted average of positions and a circular average of headings) sits in the corridor wall between them. "The mean of a bimodal distribution might lie within an obstacle so that no meaningful commands can be generated." The mode is always a possible location, but it can jump between peaks from one step to the next. Neither

is what a planner should drive toward; that is the subject of the last section.

Second, the cost meters. At the default budget the grid is $30 \times 23 \times 18 = 12\,420$ cells and its one-byte expected-distance table cost 10 782 ray casts to build, once; the cloud is 600 particles at four doubles each, and every step costs it 600 ray casts per beam it uses. Slide the budget down and the two fail *differently*: the grid blurs — a 0.6 m cell cannot say which side of a doorway you are on — while the cloud starves — 150 particles over 90 m^2 of free space leave whole rooms unrepresented, and a hypothesis with no particle near it is gone for good. That is the misconception the widget kills: the particle filter is not a cheaper grid. It is a different approximation with a different failure mode.

Third, press *kidnap*. The $\tilde{p}$ strip under each panel is Choset's monitor from §9.1.4: the average observation likelihood $\tilde{p} = N^{-1} \sum_j P(y(i) \mid x_j)$, here on a per-beam scale so one threshold serves every budget. When the robot is carried to room C the grid's $\tilde{p}$ dives and, with auto-restart on, it does what Burgard et al. did — begins global localization again from uniform. The plain cloud has no particles in room C and nothing to resample toward. The remedies Choset lists — random particles in the motion model (Fox et al.), restart on a $\tilde{p}$ threshold (Burgard et al.), a random fraction scaled by $\tilde{p}$ (Lenser and Veloso), a smoothed $\tilde{p}$ (Gutmann and Fox) — are all switchable in the library, and the hook's cloud uses two of them.

16.2.2 Five cells, by hand

Everything the widget does is this figure done many times. Choset's Figure 9.1 is a corridor with doors; shrink it to five cells on a loop with doors at 1 and 3, and use the numbers of his problems 6 and 7: $P(\text{door} \mid \text{door cell}) = 0.8$, $P(\text{door} \mid \text{wall cell}) = 0.4$, and "move right one" succeeds with probability 0.8 and stays put with 0.2.

> **INTERACTIVE · f16.3**
>
> **The Five-Cell Corridor**
>
> Sensing sharpens, moving smears — and both are the same equation applied to a table of five numbers.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch16-bayesian-localization#f16.3

Start uniform,

0.2 everywhere , entropy $\log_2 5 = 2.3219$ bits. The robot sees a door. Multiply each cell by the

likelihood — 0.8 at cells 1 and 3, 0.4 elsewhere — to get the unnormalized $[0.08, 0.16, 0.08, 0.16, 0.08]$; the sum is $\eta^{-1} = 0.56$; divide and the

posterior is $[\frac{1}{7}, \frac{2}{7}, \frac{1}{7}, \frac{2}{7}, \frac{1}{7}] \approx [0.1429, 0.2857, 0.1429, 0.2857, 0.1429]$, entropy 2.2359 bits. Sensing sharpened the belief, and it is bimodal: two doors look alike.

Now move right. Each cell receives 0.8 of its left neighbour and keeps 0.2 of itself: $bel'(c) = 0.8\,bel(c{-}1) + 0.2\,bel(c)$, which gives $\frac{1}{7}[1.0, 1.2, 1.8, 1.2, 1.8] \approx [0.1429, 0.1714, 0.2571, 0.1714, 0.2571]$ and entropy 2.2811 bits. Moving smeared it. Every number here is produced by `fiveCellSenseMove()` in the library and asserted to 10^{-4} by the check file; the figure prints what the code returns.

The two steps are the whole chapter. Sensing is a multiplication by a likelihood and a renormalization; moving is a convolution with a stochastic kernel. A convolution with a kernel that has any spread can only increase entropy, so the belief sharpens only when the robot looks.

$P(x(k) \mid u(0{:}k{-}1), y(1{:}k))$	The posterior over configurations after k steps. Sister book: bel(x_t)
$P(x(k) \mid u(k{-}1), x(k{-}1))$, $P(y(k) \mid x(k))$	Motion model and sensor model; the map m is background knowledge in both. Sister: p(x_t | u_t, x_{t−1}), p(z_t | x_t)
$\eta(k)$; $\mathcal{M} = \{(x_j, \omega_j)\}_{j=1}^{N}$	Normalizer, computed on the fly (9.6); particle set with importance weights.
(α, β, d)	Choset's odometry increment: initial rotation, translation, final rotation (Fig. 9.5). Sister: (δ_rot1, δ_trans, δ_rot2)
$d(x)$, $h_{i,j}$	Ray-cast expected distance; sensor-model histogram bin for expected d_i and reading d_j.
m_l, Odds$(\cdot)$	Occupancy of cell l; odds P/(1 − P).
$\tilde{p}$	Average observation likelihood over the cloud — the kidnap monitor of §9.1.4.
$J(c)$	Cost-to-go of lattice cell c to q_goal: the Chapter 6 universal plan, backward Dijkstra.
$\mathcal{J}_{bel}(a) = \sum_j \omega_j[1 + J(f(x_j, a))]$	Expected one-step-plus-cost-to-go of action a under the sampled belief.
λ, β_{info}	Collision / off-grid penalty (what J reads outside the free set); information-bonus weight.

16.2.3 What a planner does with a belief

The metaphor for the whole chapter is one line: **don't drive from where you probably are — drive so that wherever you are, the rest of the trip stays cheap.**

Take a 3×3 grid, rows numbered top to bottom, four-connected, unit step cost, an obstacle at $(1, 0)$ and the goal at $(0, 1)$. Backward Dijkstra from the goal labels every free cell with its cost-to-go J: 0 at the goal, 1 at its three neighbours, 2 at $(1, 2)$ and $(2, 1)$, 3 at the bottom corners, and λ at the obstacle. Suppose the belief says the robot is at $(2, 1)$ with weight 0.6 — the mode — or at $(2, 0)$ with weight 0.4. From the mode, Up is obviously best: $1 + J(1, 1) = 2$. But Up from $(2, 0)$ drives into the obstacle. Score every action by its *expected* cost under the

belief:

$$\mathcal{J}_{bel}(\text{Up}) = 0.6 \cdot 2 + 0.4 \cdot \lambda, \qquad \mathcal{J}_{bel}(\text{Right}) = 0.6 \cdot 4 + 0.4 \cdot 3 = 3.6, \qquad \mathcal{J}_{bel}(\text{Left}) = 0.6 \cdot 4 + 0.4 \cdot \lambda.$$

At $\lambda = 10$ these are 5.2, 3.6, 6.4, and Down is off-grid for both hypotheses at 10. **Right wins.** The planner first moves *sideways*, into the column that is safe under both hypotheses, and only then Up. Set the collision penalty with

20 and the inset table in the last widget recomputes: Up and Right tie when $0.6 \cdot 2 + 0.4\lambda = 3.6$, at $\lambda^* = 6$; below it the planner gambles on the mode, and at $\lambda = 4$ Up costs 2.8 and wins. The break-even is printed, not hidden, because the answer depends on it.

16.3 The mathematics

16.3.1 Definitions

Position tracking, global localization, kidnapped robot (§9.1). The initial configuration is known approximately, unknown, or believed wrongly and to be unlearned. **Posterior representation.** Mathematically $P : X \to \mathbb{R}$ lives in an infinite-dimensional space; a filter stores a finite approximation — a Gaussian $(\hat{x}, P)$, a grid over SE(2), or weighted samples. §9.1.4 treats all three; this chapter's `Belief` trait is what a planner needs from any of them: weighted samples on the lattice, the mode, the entropy.

Sensor model (§9.1.5). $P(y \mid x) = P(y \mid d(x))$ (9.22): the likelihood depends on the pose only through the ray-cast expected distance $d(x)$, and $P(y \mid d)$ is a mixture of four densities — hit, short, random, max-range (9.23) — binned at resolution Δ so each histogram h_i sums to one (9.24).

Occupancy grid (§9.2.1). A map m of binary cells m_l assumed independent, $P(m) = \prod_l P(m_l)$ (9.40), each updated by the inverse model $P(m_l \mid x(k), y(k))$.

Belief-replanning policy (this book). $a^* = \arg\min_a \mathcal{J}_{bel}(a)$, recomputed after every sense–update. An expectation-of-cost rule with replanning — *not* a POMDP policy.

16.3.2 Recursive Bayesian localization, derived once

STATEMENT (EQS. 9.1, 9.15)

$$P(x(k) \mid u(0{:}k{-}1), y(1{:}k)) = \eta(k)\, P(y(k) \mid x(k)) \sum_{x(k-1) \in X} P(x(k) \mid u(k{-}1), x(k{-}1))\, P(x(k{-}1) \mid u(0{:}k{-}2), y(1{:}k{-}1)).$$

The continuous form (9.2) replaces the sum by an integral, and the Kalman filter of Chapter 15 is its Gaussian special case (Choset's problem 2).

DERIVATION **Seven steps and two assumptions (§9.1.3)**

(9.7) → (9.15)

Step 1 — Bayes rule with the past as background. Treat $u(0{:}k{-}1), y(1{:}k{-}1)$ as background knowledge E and apply $P(A \mid B, E) = P(B \mid A, E)P(A \mid E)/P(B \mid E)$ (problem 1) with $A = x(k)$, $B = y(k)$:

$$P(x(k) \mid u(0{:}k{-}1), y(1{:}k)) = \frac{P(y(k) \mid u(0{:}k{-}1), y(1{:}k{-}1), x(k))\, P(x(k) \mid u(0{:}k{-}1), y(1{:}k{-}1))}{P(y(k) \mid u(0{:}k{-}1), y(1{:}k{-}1))}. \tag{9.7}$$

Step 2 — Assumption A: the measurement is conditionally independent of the past. Once $x(k)$ is known, $y(k)$ does not depend on earlier measurements or controls, so the first factor collapses to $P(y(k) \mid x(k))$ (9.8). This is where the map enters: it holds given $x(k)$ *and* m, and fails when m is unknown — Choset's problem 8 and Exercise 2.
Step 3 — the denominator is a constant. It does not depend on $x(k)$; call it $\eta(k)^{-1}$ (9.9).
Step 4 — total probability over the previous state. Expand the remaining factor: $P(x(k) \mid u(0{:}k{-}1), y(1{:}k{-}1)) = \sum_{x(k-1)} P(x(k) \mid u(0{:}k{-}1), y(1{:}k{-}1), x(k{-}1))\, P(x(k{-}1) \mid u(0{:}k{-}1), y(1{:}k{-}1))$ (9.10).
Step 5 — Assumption B: the motion is Markov. Given $x(k{-}1)$ and $u(k{-}1)$, $x(k)$ is independent of everything older: $P(x(k) \mid u(0{:}k{-}1), y(1{:}k{-}1), x(k{-}1)) = P(x(k) \mid u(k{-}1), x(k{-}1))$ (9.11).
Step 6 — drop $u(k{-}1)$ from the previous belief. The second factor conditions $x(k{-}1)$ on $u(k{-}1)$, a motion carried out *after* the robot was at $x(k{-}1)$. Choset argues that for small motions the fact that the robot moved a few inches tells you nothing about where it was, so $P(x(k{-}1) \mid u(0{:}k{-}1), y(1{:}k{-}1)) = P(x(k{-}1) \mid u(0{:}k{-}2), y(1{:}k{-}1))$ (9.13). *This is the step that lies.* His counterexample: two rooms, one small and one large, no door between them; if the robot moved farther than the small room's diameter, it must now be in the large room, and the motion *did* carry information about $x(k{-}1)$. Small time steps make the lie harmless.
Step 7 — substitute. (9.13) into (9.10) gives the prediction (9.14); (9.14) into (9.9) gives (9.15), which is (9.1).
Collapsibles. The sister book proves the same recursion by induction on t in its Chapter 5, with the Markov assumption named once and tested by its Markov Breaker widget. Problem 2 — rederive the Kalman filter from (9.2) by assuming Gaussian motion and sensor models — is the bridge back to Chapter 15.

16.3.3 The normalizer is free

STATEMENT (EQS. 9.5–9.6)

$$\eta(k) = \Big[\sum_{x(k)\in X} P(y(k) \mid x(k))\, P(x(k) \mid u(0{:}k{-}1), y(1{:}k{-}1)) \Big]^{-1}.$$

DERIVATION One running sum

$\eta^{-1} = \Sigma_x P(y|x)P(x)$

Step 1. $\eta^{-1} = P(y(k) \mid u(0{:}k{-}1), y(1{:}k{-}1))$ is the marginal likelihood of the reading, "which generally is hard to compute" because consecutive measurements are dependent when the location is unknown (9.5).
Step 2. Apply total probability over $x(k)$: $\eta^{-1} = \sum_x P(y(k) \mid x)\, P(x \mid u(0{:}k{-}1), y(1{:}k{-}1))$ (9.6).
Step 3. Every term in that sum is a product the update (9.4) already forms. Algorithm 16 lines 6–10 accumulate η while multiplying; lines 11–13 divide. In the five-cell example the sum is 0.56, and the check file confirms over 240 random updates that the normalizer the filter returns equals $\sum_x P(y \mid x)P(x)$ computed independently, to 10^{-15}.
Collapsible. Divide η^{-1} by the number of beams and you have $\tilde{p}$ on a per-beam scale — the quantity the kidnapped-robot monitors of §9.1.4 threshold. It costs nothing extra.

16.3.4 Three representations, two of them here

Kalman filters keep a Gaussian: efficient, floating-point resolution, high dimensions — and unable to say "one of three doors." Jensfeld and Christensen's mixture of Gaussians stretches it; this book stops at Chapter 15.

Grids (Burgard, Fox, and coworkers) store P on a three-dimensional lattice over (x_r, y_r, θ_r); Choset reports 10–30 cm and 2–10° as sufficient. The prediction step is $O(N^2)$ if done as written on line 4 of Algorithm 16. Fox, Burgard, and Thrun's remedy: shift every (x, y)-plane by the offsets $\Delta x, \Delta y$ its heading implies, then convolve with a bounded separable kernel for the motion noise,

$$P'(x, y, \theta) = 0.5\, P(x, y, \theta) + 0.25\, \big(P(x{-}1, y, \theta) + P(x{+}1, y, \theta)\big), \tag{9.16}$$

once per axis, $\frac{2}{3}/\frac{1}{3}$ at the borders — $O(NW)$ for kernel width W. The mean is a weighted average of cell centres, with the heading by the circular average $\hat{\varphi} = \operatorname{atan2}(\sum P(i) \sin\varphi(i), \sum P(i) \cos\varphi(i))$ (9.17); the mode is one max, sharpened to sub-cell accuracy by averaging around it.

Particles store $\mathcal{M} = \{(x_j, \omega_j)\}$ and run sequential importance sampling with resampling — "a survival of the fittest scheme." Motion is sampled, not convolved: Choset encodes an increment as (α, β, d) (Fig. 9.5) and perturbs each with Gaussians whose widths grow with both rotations and the translation,

$$\alpha' = \alpha + \alpha\,\mathcal{N}(0,\sigma_1) + d\,\mathcal{N}(0,\sigma_2), \quad \beta' = \beta + \beta\,\mathcal{N}(0,\sigma_3) + d\,\mathcal{N}(0,\sigma_4), \quad d' = d + d\,\mathcal{N}(0,\sigma_5) + (\alpha+\beta)\,\mathcal{N}(0,\sigma_6), \tag{9.19–9.21}$$

then applies (9.18): $x' = x + d'\cos(\theta+\alpha')$, $y' = y + d'\sin(\theta+\alpha')$, $\theta' = \theta + \alpha' + \beta'$. With all $\sigma_i = 0$ this is the exact increment, and the check confirms the sampler's mean displacement is unbiased to a millimetre over 20 000 draws. Samples that would cross a wall are rejected, as Choset suggests. The sister's Chapter 8 owns resampling variance, KLD-adaptive sample sizes and the histogram filter's bookkeeping; its Chapter 12 owns Augmented MCL and the grid-versus-MCL comparison table. Both are vendored unchanged under `web/lib/localize/`, and nothing in this chapter re-implements a filter.

16.3.5 The sensor model, and why its σ is not the laser's

STATEMENT (EQS. 9.22–9.25)

$P(y \mid x) = P(y \mid d(x))$, with $d(x)$ by ray casting. For expected distance d_i the histogram h_i has bins

$$h_{i,j} = \alpha\,\mathcal{N}(d_j; d_i, \sigma) + \beta\,\lambda e^{-\lambda d_j} + \gamma \quad (j < n-1), \qquad h_{i,n-1} = \delta, \qquad \sum_j h_{i,j} = 1, \tag{9.23–9.24}$$

each continuous term integrated over its bin of width Δ. A scan is scored as a product over beams, $P(y_1, \ldots, y_m \mid x) = \prod_i P(y_i \mid x)$ (9.25).

The four situations come straight from Choset's Figure 9.9, one ultrasound scan from a B21: most beams hit the nearest object (Gaussian); some are shorter — crosstalk, a person, a refrigerator not in the map (exponential); some pass through a bookshelf and return nonsense (uniform); some never return (max range). Fit the weights by log-likelihood maximization: the library's EM over $(\alpha, \beta, \gamma, \delta)$ recovers $(0.6, 0.15, 0.15, 0.1)$ from 6000 synthetic readings to within 0.03. Two honesty items Choset insists on, and the code keeps:

- **Beams are not independent.** (9.25) is false and useful; neighbouring beams see the same unmodeled chair. Fox et al. used 60 of the 181 beams of a SICK scanner; the `every` parameter in `scan_likelihood` is that remedy, and the widgets default to every third beam.
- **The variance encodes the map, not just the sensor.** "The variances of the Gaussians do not depend solely on the accuracy of the sensor. They also

encode the uncertainty of the map." A SICK laser's σ is a few centimetres; the localizer uses $\sigma = 0.25$ m. The check file runs the same 600 particles both ways on the same log: at 0.25 m the mode ends within 0.6 m of the truth; at the raw 0.08 m no particle ever lands within one σ of the right answer and the cloud starves, six metres off. That is Choset's problem 9, and the sister's Chapter 10 is where beam intrinsics and the likelihood-field alternative are fitted properly.

The cost trick is also his: discretize distances to 256 values and $P(y \mid d)$ is "only 256 histograms with 256 bins each," 88 kB at double precision; precompute $d(x)$ for every grid cell and heading bin as one byte each, and the grid localizer's correction is a table lookup per beam. The widget's 0.4 m grid built that table from 10 782 ray casts and reads it for the rest of the run.

16.3.6 Log-odds occupancy mapping

STATEMENT (EQS. 9.35–9.36)

With poses known, each cell's posterior odds update by

$$\log \mathrm{Odds}(m \mid x(1{:}k), y(1{:}k)) = \log \mathrm{Odds}(m \mid x(k), y(k)) - \log \mathrm{Odds}(m) + \log \mathrm{Odds}(m \mid x(1{:}k{-}1), y(1{:}k{-}1)),$$

and with prior $P(m) = 0.5$ the middle term vanishes.

DERIVATION Why the hard denominator cancels (§9.2.1)

(9.26) → (9.39)

Step 1 — Bayes on m with the pose history as background. $P(m \mid x(1{:}k), y(1{:}k)) = P(y(k) \mid m, x(1{:}k), y(1{:}k{-}1))\, P(m \mid x(1{:}k), y(1{:}k{-}1))/P(y(k) \mid x(1{:}k), y(1{:}k{-}1))$ (9.26).
Step 2 — the reading depends only on the map and the current pose. $P(y(k) \mid m, x(1{:}k), y(1{:}k{-}1)) = P(y(k) \mid m, x(k))$ (9.27).
Step 3 — Bayes again, to swap the forward model for the inverse one. $P(y(k) \mid m, x(k)) = P(m \mid x(k), y(k))\, P(y(k) \mid x(k))/P(m \mid x(k))$, and $P(m \mid x(k)) = P(m)$ because a pose alone says nothing about occupancy (9.28–9.29). The posterior now contains the *inverse sensor model* $P(m \mid x(k), y(k))$ — the thing a designer can actually write down — and two denominators nobody wants to compute.
Step 4 — do the same for $\neg m$ and divide. $P(y(k) \mid x(k))$ and $P(y(k) \mid x(1{:}k), y(1{:}k{-}1))$ appear identically in both and cancel (9.30–9.31). What remains is a product of three odds ratios.

Step 5 — take logs. Products become sums: the log-odds recursion above (9.35); prior 0.5 kills the prior term (9.36). Algorithm 19 line 3 is the same statement in probabilities,

$$P_m \leftarrow \left[1 + \frac{1-P(m|x(i),y(i))}{P(m|x(i),y(i))}\,\frac{P(m)}{1-P(m)}\,\frac{1-P_m}{P_m}\right]^{-1}, \tag{9.39}$$

and the check file confirms the two forms agree to 10^{-15} on 500 random triples.

Collapsibles. The cell-independence assumption (9.40) is false and useful — a detected door should update a door-shaped set of cells together — and the per-beam independence of §9.2.1's last paragraph is false again. Choset's inverse model for sonar is the piecewise-linear cone (9.41) with deviation $s(y,\theta) = g(y)\,\mathcal{N}(\theta;0,\sigma_\theta)$ (9.42), $\sigma_\theta = 0.05$ rad; at $y = 2$ m on the optical axis he states $s \approx 0.16$, and the library's `chosetConeModel` gives 0.1596. The lasers in this book use the sister's thin-beam inverse model instead; its Chapter 13 has the independence trap and the MAP-mapping repair. Mapping the Apartment from 120 known poses marks 88.7 % of observed wall cells occupied and 99.8 % of observed free cells free, and the map's total entropy falls to under 30 % of its initial value — not monotonically, because a confident cell contradicted by a later beam climbs back toward one bit.

16.3.7 Rao-Blackwellized SLAM, as a statement

STATEMENT (EQS. 9.45–9.47)

$$P(x(1{:}k), m \mid u(0{:}k{-}1), y(1{:}k)) = P(m \mid x(1{:}k), y(1{:}k))\; P(x(1{:}k) \mid y(1{:}k), u(0{:}k{-}1)).$$

Given the path, the map is the closed-form mapping-with-known-poses posterior of the previous derivation; the path is what a particle filter samples. Each particle is a path *with its own map*.

DERIVATION **Marginalize what you can, sample the rest (§9.2.2)**

Algorithm 20

Step 1 — chain rule. Factor the joint over path and map as $P(m \mid x(1{:}k), y(1{:}k), u(0{:}k{-}1))\, P(x(1{:}k) \mid y(1{:}k), u(0{:}k{-}1))$ (9.45).

Step 2 — the map does not depend on the controls once the path is

known (9.46), giving (9.47).
Step 3 — each particle carries a path $h^{(j)}(1{:}k)$ and the maximum-likelihood map for it, $m^{(j)} = \arg\max_m P(m \mid h^{(j)}(1{:}k), y(1{:}k{-}1))$ (9.48), built incrementally by Algorithm 19.
Step 4 — the weight is the scan's likelihood under that particle's own map, Algorithm 20 line 11: $\omega_j = P(y(i) \mid x_j, m^{(j)}(1{:}i{-}1))$. A particle whose map disagrees with what it now sees dies at the next resampling; this is what closes loops.
Step 5 — resample (Algorithm 18). Choset's two engineering warnings: recomputing each map from scratch is quadratic in k, and copying a full map per particle on every resample is ruinous — hence tree-shared maps (Montemerlo et al.; Parr and Eliazar) or a bounded window of scans per particle (Hähnel, Burgard, Fox, and Thrun), and scan-matched odometry to cut the particle count.
Collapsibles. Scan matching alone (9.44) — greedy maximization of $P(y(k) \mid x(k), \hat{m})\, P(x(k) \mid u(k{-}1), \hat{x}(k{-}1))$ — produces locally crisp, globally inconsistent maps (Figs. 9.19–9.20); the particle posterior over paths is what repairs them. On a 45-step Apartment log with slipping odometry, six particles each carrying a 0.2 m occupancy grid end 0.12 m from the truth where dead reckoning ends 0.20 m off. The sister's Chapter 17 is the full treatment, landmark and grid versions both.

16.3.8 Replanning on a belief as expected cost-to-go

This derivation is the book's addition; neither Choset nor the sister states it as a planning primitive.

STATEMENT

Let J be the exact cost-to-go on the free lattice from backward Dijkstra (Chapter 6), with $J(\text{obstacle}) = J(\text{off-grid}) = \lambda$. For a belief given by weighted samples $\{(x_j, \omega_j)\}$,

$$a^* = \arg\min_a \sum_j \omega_j \left[\, c(a) + J(f(x_j, a)) \,\right]$$

minimizes the expected cost of "take a, then follow the optimal plan from wherever you actually are." The certainty-equivalent choice $\arg\min_a [c(a) + J(f(x_{MAP}, a))]$ coincides with it iff all samples agree on the ranking.

DERIVATION **Bellman, then an expectation in the wrong place, then replanning to fix it**

a decision costs O(K· | A |) after one Dijkstra

Step 1 — known state. With x known, Bellman's principle says the optimal first action is $\arg\min_a [c(a) + J(f(x,a))]$, where J is the optimal cost-to-go. Backward Dijkstra from the goal computes J for *every* cell at once — the universal plan of Chapter 6 — so following it from any start reproduces A*'s path and cost. The check file does this on fifty seeded Apartment queries: one sample of weight one, and the belief planner's path cost equals $J(\text{start})$ to 10^{-14}.

Step 2 — unknown state, no more observations. If the robot will never look again, the best open-loop first action minimizes $\mathbb{E}_x[c(a) + J(f(x,a))]$: the expectation of the known-state objective. Moving the expectation outside the min over *future* actions — assuming the state will be revealed after this one step — is exactly the QMDP approximation of Littman, Cassandra, and Kaelbling. It is optimistic: it never plans to gather information, because it assumes information will arrive for free.

Step 3 — the price of a decision. J depends on the goal and the map, not on the belief. Compute it once, $O(|V| \log |V|)$; thereafter each decision is K samples times $|A|$ actions of array reads. This is what makes planning on a belief cheap enough to do every tick, and it is why the Chapter 6 universal plan was worth more than its footnote.

Step 4 — replanning restores the observations. The QMDP rule ignores the scan that follows the action. Taking the action, scanning, updating, and *re-deciding* puts the observation back — not optimally, but at the only point where it exists. The three-by-three example shows the shape of the result: the planner's first move is sideways, into cells that are safe under both hypotheses, because the expected cost charges λ for every hypothesis that would hit a wall. Written out, every $\mathcal{J}_{bel}(a)$ is affine in λ — a feasible part A_a plus the collision mass B_a times λ — so the break-even between two actions is exact, $\lambda^* = (A_2 - A_1)/(B_1 - B_2)$; for Up against Right it is $(3.6 - 1.2)/0.4 = 6$.

Step 5 — the information bonus, named as a heuristic. Subtract $\beta_{info}\,\Delta H_a$, the expected one-step entropy reduction a scan would deliver after a. The library estimates it as the mutual information between "which sample is true" and the noise-free scan each sample predicts: positions whose predicted scans differ are told apart by one reading, mirror rooms are not. It makes the path hug distinctive geometry. It is a heuristic for the POMDP of the sister's Chapter 22, and it is switched off by default.

Collapsibles. The belief roadmap of Prentice and Roy plans in the space of Gaussians with covariance propagated along roadmap edges; the sister's Chapter 21 is the value function our J specializes. When only the map changes, Chapter 6's D* repairs J instead of rerunning Dijkstra (Exercise 6). Nav2's fielded stack is this loop with AMCL as the belief and a costmap-based planner as J.

WORK IT OUT

In the 3×3 example, at what λ do Up and Right cost the same under the belief?

Answer. 6 $\mathcal{J}_{bel}(\text{Up}) = 0.6 \cdot 2 + 0.4\lambda$ and $\mathcal{J}_{bel}(\text{Right}) = 3.6$ for every λ, so $1.2 + 0.4\lambda = 3.6$ gives $\lambda^* = 6$. Below it — at $\lambda = 4$, Up costs 2.8 — the planner takes the mode's gamble.

16.4 The algorithm

Choset numbers these 16 through 20; the last two are this book's.

ALGORITHM `Algorithm 16 -- grid_localization(bel, u(0:k−1), y(1:k))` COST `O(N²) naive per step; O(N W) with shift + separable kernel (9.16)`

in an initial belief P(x(0)) on the grid, movements and measurements

out the posterior P(x(k) | u(0:k−1), y(1:k)) on the grid

1: $P(x) \leftarrow P(x(0))$
2: for $i = 1, \ldots, k$:
3: for all x: $P'(x) \leftarrow \sum_{x'} P(x \mid u(i{-}1), x')\, P(x')$ — in practice shift each θ-plane by $(\Delta x, \Delta y)$, then convolve with (9.16) per axis
4: $\eta \leftarrow 0$; for all x: $P(x) \leftarrow P(y(i) \mid x)\, P'(x)$, $\eta \leftarrow \eta + P(x)$ — the likelihood from the one-byte $d(x)$ table and the cached histograms
5: for all x: $P(x) \leftarrow P(x)/\eta$
6: return P; $\tilde{p} = \eta^{-1}$ per beam is the restart monitor

ALGORITHM `Algorithm 17 -- particle_localization(M, u(0:k−1), y(1:k))` COST `O(N) plus one sensor-model evaluation per particle per beam`

in a set M of N samples (x_j, ω_j), movements and measurements

out M representing the posterior

1: for $i = 1, \ldots, k$:
2: for $j = 1, \ldots, N$: draw $x \sim P(x \mid u(i{-}1), x_j)$ by (9.18)–(9.21); $x_j \leftarrow x$; reject draws that cross a wall
3: $\eta \leftarrow 0$; for all j: $\omega_j = P(y(i) \mid x_j)$ by ray casting $d(x_j)$ per beam, $\eta \leftarrow \eta + \omega_j$
4: for all j: $\omega_j \leftarrow \omega_j / \eta$
5: $M \leftarrow$ resample(M) — Algorithm 18, in this book only when N_{eff} falls below $N/2$, with Augmented-MCL injection or a sensor-resetting floor when $\tilde{p}$ asks for it
6: return M

ALGORITHM `Algorithm 18 -- resample(M)` COST `O(N), one uniform draw`

in N weighted samples
out N equally weighted samples drawn with probability proportional to ω

1: $M' \leftarrow \emptyset$; $\Delta \leftarrow \text{rand}((0, N^{-1}])$; $c \leftarrow \omega_0$; $i \leftarrow 0$
2: for $j = 0, \ldots, N{-}1$: $u \leftarrow \Delta + jN^{-1}$; while $u > c$: $i \leftarrow i + 1$, $c \leftarrow c + \omega_i$; $M' \leftarrow M' \cup \{(x_i, N^{-1})\}$
3: return M' — the low-variance comb; Isard and Blake's binary search is the $O(N \log N)$ alternative

ALGORITHM `Algorithm 19 -- occupancy_grid_mapping(y(1:k), x(1:k), P0(m))`
COST `O(cells in the beam or cone) per scan`

in measurements, the known poses they were taken from, an initial occupancy prior
out P(m | x(1:k), y(1:k)) per cell

1: $P_m \leftarrow P_0(m)$ for every cell
2: for $i = 1, \ldots, k$, for every cell the scan touches: $P_m \leftarrow \left[1 + \frac{1-P(m|x(i),y(i))}{P(m|x(i),y(i))} \frac{P(m)}{1-P(m)} \frac{1-P_m}{P_m}\right]^{-1}$ — (9.39); in code, the log-odds sum (9.35)
3: return P_m; threshold at 0.5 for the maximum-likelihood map

ALGORITHM `Algorithm 20 -- rbpf_slam(M, u(0:k−1), y(1:k))` COST `O(N · scan) per step with incremental maps`

in N particles, each a path with its own map
out M representing P(x(1:k), m | u(0:k−1), y(1:k))

1: for all j: $x_j \leftarrow (0,0,0)$, $m^{(j)} \leftarrow P_0(m)$
2: for $i = 1, \ldots, k$:
3: for all j: draw $x \sim P(x \mid u(i{-}1), x_j)$; $x_j \leftarrow x$
4: $\eta \leftarrow 0$; for all j: $\omega_j = P(y(i) \mid x_j, m^{(j)}(1{:}i{-}1))$, $\eta \leftarrow \eta + \omega_j$ — the scan scored against *this particle's* map
5: normalize; $M \leftarrow$ resample(M); integrate $y(i)$ at x_j into $m^{(j)}$ by Algorithm 19
6: return M

ALGORITHM `cost_to_go(lattice, goal, λ) and replan_on_belief(J, samples, actions, λ, β_info)` COST O(|V| log |V|) once per goal; O(K · |A|) per decision

in the Chapter 6 lattice and goal; a sampled belief {(x_j, ω_j)}; the action set

out J on every cell; the argmin action and the full per-action table

1: $J \leftarrow$ backward Dijkstra from the goal (Chapter 6 `universal_plan`); $J \leftarrow \lambda$ on obstacles, off-grid, and cells the goal cannot reach
2: for each action a: $\mathcal{J}_{bel}(a) \leftarrow \sum_j \omega_j \, [\, c(a) + J(f(x_j, a)) \,]$, with $J = \lambda$ whenever a is infeasible from x_j; record the collision mass $B_a = \sum_{j:a \text{ infeasible}} \omega_j$
3: if $\beta_{info} > 0$: $\mathcal{J}_{bel}(a) \leftarrow \mathcal{J}_{bel}(a) - \beta_{info} \, \Delta H_a$
4: return $\arg\min_a \mathcal{J}_{bel}(a)$ and the table — the widget's inset *is* this table
5: the loop: act, scan, fold the bumper reading in, update the belief, go to 2; rerun 1 only when the goal or the map changes (or let D* repair it)

16.5 Implementation in Rust

The `estimate` crate of Chapter 15 gains a `bayes` module. Nothing in it re-implements a filter: the sister book's `HistogramFilter`, `ParticleFilter`, `low_variance_resample`, `GridLocalizer`, `Mcl`, `AugmentedMcl`, `OccGrid`, and `GridRbpf` are vendored unchanged. What the module adds is Choset's vocabulary around them, the `Belief` trait a planner consumes, and the one owned artifact.

RUST **crates/estimate/src/bayes/mod.rs**

```rust
pub use filters::nonparametric::{HistogramFilter, ParticleFilter, ParticleSet,
    low_variance_resample};
pub use filters::localize::{GridLocalizer, Mcl, AugmentedMcl};
pub use filters::occgrid::OccGrid;
use search::{Lattice, CellIdx};   // the Chapter 6 grid and its moves
use rand_pcg::Pcg64;

```

```rust
/// What a planner needs from any posterior: weighted samples on the planning
/// lattice plus the two statistics Choset discusses (sec.9.1.4). Implemented for the
/// grid belief, the particle set, and the Chapter 15 Gaussian, so
/// `replan_on_belief` never knows which representation it was handed.
pub trait Belief {
    /// K cells with weights summing to one. The grid returns its K heaviest
    /// marginal cells; the cloud draws K by the low-variance comb (Alg. 18).
    fn samples(&self, k: usize, lattice: &Lattice, rng: &mut Pcg64) -> Vec<(CellIdx, f64)>;
    /// Choset: "the mode has the advantage that it generally corresponds to a
    /// possible location of the vehicle" -- unlike the mean (9.17).
    fn mode(&self, lattice: &Lattice) -> CellIdx;
    fn entropy_bits(&self) -> f64;
}

/// Choset's increment (Fig. 9.5): rotate alpha toward the target, drive d, rotate beta.
/// The sister's (delta_rot1, delta_trans, delta_rot2) under other letters; `From` both ways.
#[derive(Clone, Copy, Debug)]
pub struct Increment { pub alpha: f64, pub d: f64, pub beta: f64 }

/// sigma_1...sigma? of (9.19)-(9.21). Widths grow with the rotations *and* the translation,
/// which is why a long straight run still fans out in heading.
pub type Sigmas = [f64; 6];

pub fn sample_motion(x: &SE2, u: Increment, s: &Sigmas, rng: &mut Pcg64) -> SE2 {
    let n = |sigma: f64| if sigma > 0.0 { Normal::new(0.0, sigma).unwrap().sample(rng) } else { 0.0 };
    let alpha = u.alpha + u.alpha * n(s[0]) + u.d * n(s[1]);
    let beta  = u.beta  + u.beta  * n(s[2]) + u.d * n(s[3]);
    let d     = u.d     + u.d     * n(s[4]) + (u.alpha + u.beta) * n(s[5]);
    SE2::new(x.x() + d * (x.theta() + alpha).cos(),       // (9.18)
             x.y() + d * (x.theta() + alpha).sin(),
             wrap(x.theta() + alpha + beta))
}
```

The five-cell corridor is the sister's one-dimensional histogram filter with Choset's problem 6–7 parameters, and nothing more. It exists so the §3 numbers have one source.

RUST **crates/estimate/src/bayes/corridor.rs**

```rust
use filters::nonparametric::HistogramFilter;

/// Choset's Fig. 9.1 corridor on a loop of `N` unit cells. `sense` is the
/// filter's `correct`, `step` its `predict` with the two-point kernel of
/// problem 7 -- no new filter code lives here.
pub struct Corridor<const N: usize> {
    pub filter: HistogramFilter<N>,
    pub doors: Vec<usize>,
    pub p_door_at_door: f64,   // 0.8
    pub p_door_at_wall: f64,   // 0.4
    pub p_move: f64,           // 0.8 succeeds, 0.2 stays
}

pub struct Sense { pub likelihood: [f64; 5], pub unnormalized: [f64; 5], pub evidence: f64, pub posterior: [f64; 5] }
```

```rust

impl<const N: usize> Corridor<N> {
    /// Update (9.4)-(9.6): multiply, sum, divide -- and hand back the sum, because
    /// eta-1 is free and the kidnap monitors of sec.9.1.4 want it.
    pub fn sense(&mut self, saw_door: bool) -> Sense {
        let lik = |c: usize| {
            let p = if self.doors.contains(&c) { self.p_door_at_door } else { self.
   p_door_at_wall };
            if saw_door { p } else { 1.0 - p }
        };
        let prior = self.filter.belief();
        let unnormalized: Vec<f64> = (0..N).map(|c| prior[c] * lik(c)).collect();
        let evidence: f64 = unnormalized.iter().sum();                // Alg. 16 lines
   6-10
        self.filter.correct(|c| lik(c));                              // lines 11-13
   inside
        Sense { likelihood: array(lik), unnormalized: array(|c| unnormalized[c]),
                evidence, posterior: self.filter.belief() }
    }

    /// Prediction (9.3) for "move `steps` places": land with p_move, stay otherwise,
    /// written on the displacement error the filter hands us, with the loop's wrap.
    pub fn step(&mut self, steps: i64) -> [f64; N] {
        let (n, p) = (N as i64, self.p_move);
        let wrap = |d: i64| d - n * ((d as f64 / n as f64).round() as i64);
        self.filter.predict(steps, |d| if wrap(d) == 0 { p } else if wrap(d + steps) == 0
    { 1.0 - p } else { 0.0 });
        self.filter.belief()
    }
}
```

`cargo run --example five_cells -p estimate` builds the instance with doors $\{1, 3\}$, runs sense then move, and prints the table the figure above draws:

```
prior       0.2000 0.2000 0.2000 0.2000 0.2000   H = 2.3219 bits
sense door  0.1429 0.2857 0.1429 0.2857 0.1429   eta-1 = 0.56, H = 2.2359 bits
move right  0.1429 0.1714 0.2571 0.1714 0.2571   H = 2.2811 bits
```

`#[test] fn reproduces_five_cell_sense_move()` asserts every entry to 10^{-4}; the TypeScript check does the same. Choset's problem 6 — ten places, landmarks at $\{0, 3, 6\}$, detect, move 3, detect, move 4, detect nothing — runs on the same type with $N = 10$ and gives $P(7) = 0.30$, $P(4) = 0.15$, $P(0) = 0.10$; with problem 7's 0.8/0.2 motion the mode stays at 7 but drops to 0.2578 (Exercise 1).

The sensor model is a cache first and a formula second.

RUST `crates/estimate/src/bayes/sensor.rs`

```rust
/// Choset (9.23): hit N(y; d, sigma) + short lambdae^{-lambday} + random gamma + max-
   range delta, binned
/// at Delta. The expected distance d(x) comes from the Chapter 2 ray cast -- never
/// re-implemented here. sigma is *not* the laser's: it also encodes the map's
   uncertainty.
pub struct ProximityModel {
    pub sigma: f64, pub lambda: f64,
    pub alpha: f64, pub beta: f64, pub gamma: f64, pub delta: f64,
    pub delta_bin: f64, pub max_range: f64,
```

```
    rows: Vec<OnceCell<Box<[f64]>>>,              // the 256 histograms of sec.9.1.5, lazily
}

impl ProximityModel {
    pub fn n_bins(&self) -> usize { (self.max_range / self.delta_bin).ceil() as usize + 1 }

    /// h_i: P(bin j | expected distance in bin i). The three continuous terms are
    /// each normalized on [0, d_max) and the row is scaled so Sigma_j h_{i,j} = 1 (9.24),
    /// which is what Choset's problem 3(b) asks for by "appropriate values for delta_i".
    pub fn row(&self, i: usize) -> &[f64] {
        self.rows[i].get_or_init(|| {
            let d = self.bin_center(i.min(self.n_bins() - 2));
            let m = self.n_bins() - 1;
            let mut h = vec![0.0; m + 1];
            let hit_norm = normal_cdf(self.max_range, d, self.sigma) - normal_cdf(0.0, d, self.sigma);
            let short_norm = 1.0 - (-self.lambda * self.max_range).exp();
            for j in 0..m {
                let (lo, hi) = (j as f64 * self.delta_bin, ((j + 1) as f64 * self.delta_bin).min(self.max_range));
                let hit   = (normal_cdf(hi, d, self.sigma) - normal_cdf(lo, d, self.sigma)) / hit_norm;
                let short = ((-self.lambda * lo).exp() - (-self.lambda * hi).exp()) / short_norm;
                let rand  = (hi - lo) / self.max_range;
                h[j] = self.alpha * hit + self.beta * short + self.gamma * rand;
            }
            h[m] = self.delta;
            let s: f64 = h[..m].iter().sum();
            for v in &mut h[..m] { *v *= (1.0 - self.delta) / s; }
            h.into_boxed_slice()
        })
    }

    /// P(y | d) = h_{bin(d), bin(y)}, eq. (9.22) with the table doing the work.
    pub fn likelihood(&self, y: f64, d_expected: f64) -> f64 { self.row(self.bin(d_expected))[self.bin(y)] }

    /// log ?_k P(y_k | d_k) over every `every`-th beam -- (9.25) with Fox et al.'s
    /// 60-of-181 remedy for its false independence assumption.
    pub fn log_scan_likelihood(&self, scan: &[f64], expected: &[f64], every: usize) -> f64 {
        scan.iter().zip(expected).step_by(every.max(1))
            .map(|(&y, &d)| self.likelihood(y, d).max(1e-300).ln()).sum()
    }
}
```

And the owned artifact. `J` is the Chapter 6 `UniversalPlan` with λ outside the free set; the decision is a loop over samples and actions.

RUST **crates/estimate/src/bayes/replan.rs**

```
use search::{Lattice, CellIdx, Action, GridGraph, universal_plan, UniversalPlan};

/// K draws per decision; collision / off-grid penalty lambda; information bonus weight
```

```rust
   (0 disables).
pub struct ReplanConfig { pub k_samples: usize, pub lambda: f64, pub beta_info: f64 }

/// Ch. 6 universal plan: exact cost-to-go to `goal`, with lambda wherever the plan has
/// nothing to say. Computed once per goal; D* (Ch. 6) repairs it when the map changes.
pub struct CostToGo { plan: UniversalPlan<CellIdx>, pub lambda: f64 }

pub fn cost_to_go(graph: &GridGraph, goal: CellIdx, lambda: f64) -> CostToGo {
    CostToGo { plan: universal_plan(graph, goal), lambda }
}
impl CostToGo {
    pub fn j(&self, c: CellIdx) -> f64 { self.plan.cost_to_go.get(&c).copied().unwrap_or(
   self.lambda) }
    /// c(a) + J(f(x, a)), or lambda when `a` leaves the free set from `x`.
    pub fn action_cost(&self, lattice: &Lattice, x: CellIdx, a: Action) -> f64 {
        match lattice.successor(x, a) {
            Some(next) if lattice.is_free(next) && !lattice.cuts_corner(x, a) => a.
   step_cost(lattice) + self.j(next),
            _ => self.lambda,
        }
    }
}

/// Expected one-step entropy reduction after `a` -- a heuristic, named so.
pub trait InfoBonus { fn expected_entropy_drop(&self, after: &[(Option<CellIdx>, f64)], a:
    Action) -> f64; }

/// a* = argmin_a Sigma_j omega_j [ c(a) + J(f(x_j, a)) ] - beta_info . DeltaH_a. Returns
    the per-action
/// table too: the widget's inset and the sec.3 micro-example test both need it.
pub fn replan_on_belief(
    j: &CostToGo, samples: &[(CellIdx, f64)], lattice: &Lattice,
    cfg: &ReplanConfig, info: Option<&dyn InfoBonus>,
) -> (Action, Vec<(Action, f64)>) {
    let table: Vec<(Action, f64)> = lattice.actions().iter().map(|&a| {
        let expected: f64 = samples.iter().map(|&(x, w)| w * j.action_cost(lattice, x, a)
   ).sum();
        let bonus = match info {
            Some(b) if cfg.beta_info > 0.0 => {
                let after: Vec<_> = samples.iter().map(|&(x, w)| (lattice.successor(x, a).
   filter(|&n| lattice.is_free(n)), w)).collect();
                cfg.beta_info * b.expected_entropy_drop(&after, a)
            }
            _ => 0.0,
        };
        (a, expected - bonus)
    }).collect();
    let best = table.iter().min_by(|p, q| p.1.total_cmp(&q.1)).expect("at least one
   action").0;
    (best, table)
}

/// The loop every fielded robot runs: sense -> update -> replan -> act. One loop for a
/// grid, a cloud, or the Chapter 15 Gaussian.
pub struct BeliefPlanner<B: Belief> { pub belief: B, pub j: CostToGo, pub cfg:
   ReplanConfig, pub rng: Pcg64 }

impl<B: Belief + Localizer> BeliefPlanner<B> {
    pub fn decide(&mut self, lattice: &Lattice) -> Action {
        let samples = self.belief.samples(self.cfg.k_samples, lattice, &mut self.rng);
        replan_on_belief(&self.j, &samples, lattice, &self.cfg, None).0
```

```
    }
    pub fn step(&mut self, lattice: &Lattice, rusty: &mut sim::Rusty, world: &sim::World)
    -> Action {
        let a = self.decide(lattice);
        let (u, bumped) = rusty.drive_lattice_step(a, world, &mut self.rng); // odometry
    reports the move either way
        self.belief.correct_contact(&u, bumped);          // the bumper is a sensor too (
    Ch. 3)
        self.belief.predict(&u, &mut self.rng);           // Alg. 17 lines 2-5
        self.belief.correct(&rusty.scan(world, &mut self.rng)); // lines 6-13, then Alg.
    18 when N_eff is low
        a
    }
}
```

`cargo run --example replan_3x3` builds the lattice with the obstacle at $(1, 0)$, prints J, then the four expected costs at $\lambda \in \{10, 6, 4\}$ with the chosen action:

```
J =  [ 1  0  1 ]
     [ lambda  1  2 ]
     [ 3  2  3 ]
belief: (2,1) omega=0.6  (2,0) omega=0.4
lambda = 10   Up 5.20   Right 3.60   Down 10.00   Left 6.40   -> Right   (mode says Up)
lambda =  6   Up 3.60   Right 3.60   Down  6.00   Left 4.00   -> tie, Up by action order
lambda =  4   Up 2.80   Right 3.60   Down  4.00   Left 3.20   -> Up
break-even lambda* = 6.0
```

`#[test] fn replan_disagrees_with_mode_at_lambda_10()` asserts $\{5.2, 3.6, 6.4, 10\}$, the tie at 6, and the flip at 4; `#[test] fn point_mass_belief_equals_a_star()` checks that one sample of weight one reproduces the Chapter 6 path step by step on fifty seeded Apartment queries.

> **NOTE**
>
> The widgets on this page run the TypeScript port of these files, `web/lib/estimate/bayes.ts` and `replan.ts`, over the vendored `web/lib/filters`, `localize`, and `mapping/occgrid`. Fourteen checks in `__checks_ch16__.ts` pin every number printed above: the five-cell table to 10^{-4}, problem 6's posterior, the normalizer identity, the histogram rows and the ML fit, the unbiased motion sampler, the recorded run on both localizers (12 420 cells, 10 782 ray casts, problem 9's starving cloud), mapping with known poses (88.7 % / 99.8 %), the cone model's 0.16, the 3×3 decision and its break-even, the point-mass agreement with A*, the headless integration lab, the hook's two outcomes, and the Rao-Blackwellized filter's 0.12 m against dead reckoning's 0.20 m.

16.6 Putting it together

The integration lab is the owned artifact running in the Apartment, with the belief that opened the chapter.

> **INTERACTIVE · w16.2**
>
> **Plan-on-Belief**
>
> Planning on a belief does not mean planning from its mean or its mode. The mean can be in a wall and the mode can be wrong; expected cost-to-go over samples is the honest minimum short of a POMDP.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch16-bayesian-localization#w16.2

Rusty is in room B, flush under the corridor wall at $(5.0, 3.4)$. The cloud puts 0.6 of its mass in room A under A's open doorway and 0.4 in room B — the 3×3 example with the Apartment's walls. The goal is the west end of the corridor. The

graph field is J on the 0.4 m lattice inflated by Rusty's radius, computed once; every tick the planner draws $K = 24$ cells from the cloud, scores the eight lattice actions, executes the argmin, scans, folds the bumper reading in, and updates. The

posterior belief is drawn as dots and filled cells; the route the mode alone would take is the thick haloed

path stroke — two purple families on one canvas, kept apart by shape rather than hue, which is the book's rule for this chapter.

What happens, and what the check pins. With $\lambda = 20$ the first action is sideways: Up carries about 0.4 of collision mass — pinned above 0.3 — because it is a door under one hypothesis and a wall under the other. Within a few steps the corridor walls come into view, the cloud collapses onto room B, the entropy falls to under half its starting value, and the path straightens toward the goal. The belief planner arrives without a bump. Tick *plan from the mode* and the first action is Up; Rusty hits room B's wall at least once before the scans sort it out. Both runs are in the check file on the widget's seed, and the same scene holds on six seeds.

The λ slider. Every row of the inset table is affine in λ, so the break-even with the current winner is computed exactly and drawn under the slider. Drag below it and the belief planner gambles on the mode too; the 3×3 inset moves with the same number, and its $\lambda^* = 6$ never changes because its geometry never does.

The information bonus. Raise β_{info} and the path detours toward geometry that distinguishes the hypotheses — the corridor's asymmetric north doorways — and the entropy sparkline drops sooner. It is a heuristic and the widget says so; it is also a preview of what a POMDP planner buys for its price.

Freeze the belief and the planner replans on stale information: it is still the expected-cost rule, still better than the mode when the two hypotheses disagree,

but nothing it does can shrink the cloud. Replanning without sensing is the QMDP approximation with no observations to put back.

Three honesty items the chapter keeps, because the capstone inherits this loop. Assumption B of the derivation fails for large moves, and the loop takes small ones for that reason as much as for control. The belief planner is a one-step expectation with replanning, not an optimal POMDP policy, and its answer depends on λ — a price the user sets, printed next to the decision it changes. And the five-cell numbers use Choset's problem parameters; they are not a figure from his text.

What Chapter 23 takes from here is exactly `BeliefPlanner::step`: `AugmentedMcl` fed by the proximity model and the odometry adapter, `cost_to_go` over the Chapter 6 lattice inflated by Chapter 7's brushfire clearance, and `replan_on_belief` every tick. Chapter 19's MPC consumes the same belief's mean *and* covariance; Chapter 22 learns heuristics on the same sampled loop.

16.7 Exercises

1 FOUNDATION •• **Choset's problem 6, then problem 7**

Ten places on a circle, landmarks at $\{0, 3, 6\}$ that all look alike, $P(\text{landmark} \mid \text{landmark place}) = 0.8$ and 0.4 elsewhere. The robot detects a landmark, moves 3 places counterclockwise and detects a landmark, moves 4 and detects none. Compute the posterior by hand with deterministic moves (problem 6). Then redo it with problem 7's motion — each unit step succeeds with 0.8 and stays with 0.2 — and explain why the mode's probability drops while the mode itself does not move.

WORK IT OUT

Problem 6: what is the posterior probability of place 7?

Answer. 0.3 After the first detection the belief is $\propto [0.8, 0.4, 0.4, 0.8, 0.4, 0.4, 0.8, 0.4, 0.4, 0.4]$. Shift by 3, multiply by the same likelihood, shift by 4, multiply by $[0.2, 0.6, \dots]$ (no landmark seen), normalize: place 7 carries 0.30, place 4 carries 0.15, place 0 carries 0.10. With the noisy motion the library gives $P(7) = 0.2578$.

2 FOUNDATION •• **Problem 8: where the map hides in Assumption A**

Argue that once $x(k)$ and the map m are known, $y(k)$ is independent of $y(1{:}k{-}1)$ — this is step 2 of the derivation. Then exhibit the dependence when m is *not* given: construct two maps that agree at $x(k{-}1)$ and differ at $x(k)$, and show that $y(k{-}1)$ changes the predictive distribution of $y(k)$ through the posterior over maps. Relate the result to

why Bayesian SLAM carries one map per particle rather than one map for all.

3 CONCEPTUAL •• **Predict the first move and the flip**

PREDICT FIRST

In Plan-on-Belief with the default belief and $\lambda = 20$, what is the first action, and roughly where does the break-even marker sit?

- Up; break-even near $\lambda = 2$, since the mode is in room A
  Up from room B is a wall for 0.4 of the mass. Its expected cost carries $0.4 \cdot \lambda$, which at $\lambda = 20$ is already 8 — far more than any sideways step.
- ✓ A sideways step; break-even where $0.4 \cdot \lambda$ equals the extra path length of going around
  Every row is affine in λ with slope equal to its collision mass. Up has slope ≈ 0.4; the sideways actions have slope ≈ 0. They cross where $0.4 \cdot \lambda$ pays for the detour, which the widget computes exactly and draws under the slider.
- Down, away from the wall under both hypotheses
  Down is safe under both hypotheses but increases J for both; the expected cost includes the cost-to-go, not only the collision risk.

Verify both on the widget. Then lower K until the decision turns noisy: at what K does the marked break-even move by more than one slider step between consecutive ticks? Explain with the variance of a K-sample estimate of the collision mass.

4 CONCEPTUAL •• **Kidnap during the bimodal phase**

PREDICT FIRST

In Grid vs Particles, kidnap Rusty while the posterior is split between rooms A and C. Before the next frame is drawn, where does the mean marker land?

- On the true position in room C
  The mean of a bimodal distribution is between the modes, not at either. The filter has not yet seen the scan that will move mass toward C.
- ✓ In the wall between the rooms, roughly midway
  Eq. 9.17 averages positions by weight; with mass in A and C the average lies in the corridor wall between them, a configuration the robot cannot occupy. The heading mean is a circular average and can be equally meaningless.
- On the mode, since the filter snaps the mean to the nearest peak
  Nothing in (9.17) snaps anything. The mode is a separate statistic, computed by a maximum, and it is the marker that stays in free space.

Then explain, from the circular average in (9.17), why the heading mean of two particles facing +90° and −90° is 0° with zero resultant length — a direction neither hypothesis supports.

5 PRACTICAL • • • **A Belief for the Chapter 15 Gaussian**

Implement `Belief` for `Gaussian<3>` by sigma points — seven for $n = 3$, weights from the unscented transform — instead of seeded draws, mapping each sigma point to its lattice cell and merging duplicates. Property-test that for a tight Gaussian ($\sigma \leq 0.1$ m) `replan_on_belief` and A* from the mean agree on 100 seeded Apartment queries, and that for a wide one ($\sigma \geq 0.8$ m) they disagree at least once. The TypeScript `consolidateCells` is the merge step.

6 PRACTICAL • • • **D* Lite under the belief planner (stretch)**

Replace the full backward Dijkstra in `cost_to_go` with Chapter 6's D* Lite so that an occupancy change repairs only the affected cells. Measure `BeliefPlanner::step` time, fixed seed, before and after on a run with twenty door events (a door closes, the lattice cell becomes occupied, J must change). Report the speedup and the number of cells touched per repair.

HINTS

<>D *Lite searches from the goal toward the robot — which is exactly the direction backward Dijkstra already runs, so the cost-to-go labels it maintains are* J.</> <>*The belief planner reads* J *at successors of* sampled* cells, not only at the robot's cell; make sure the repaired region covers every cell a sample might query, or fall back to λ for stale ones.</> <>Twenty door events in 90 steps means Dijkstra from scratch 20 times on ~700 cells versus 20 local repairs; the speedup is the ratio of cells expanded.</>

16.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

Chapter 9 is this chapter's source: the derivation with its two assumptions and the two-rooms caveat (§9.1.3), grids and particles with Algorithms 16–18 (§9.1.4), the sensor model (§9.1.5), occupancy grids and Algorithm 19 (§9.2.1), Rao-Blackwellized SLAM and Algorithm 20 (§9.2.2), and problems 6–9. link

Fox, D., Burgard, W., and Thrun, S. (1999). Markov Localization for Mobile Robots in Dynamic Environments. *Journal of Artificial Intelligence Research 11, 391–427*.

The grid localizer of Algorithm 16 as fielded on Rhino: shift-and-convolve prediction, the sensor model's treatment of unmodeled obstacles, and the 60-of-181 beam subsampling this chapter adopts. doi:10.1613/jair.616

Fox, D. (2003). Adapting the Sample Size in Particle Filters Through KLD-Sampling. *International Journal of Robotics Research 22(12), 985–1003*.

How many particles a cloud needs, and why the answer changes as it converges — the budget slider's missing axis, treated fully in the sister's Chapter 8. doi:10.1177/0278364903022012001

Moravec, H. P. and Elfes, A. (1985). High Resolution Maps from Wide Angle Sonar. *IEEE International Conference on Robotics and Automation*.

The occupancy grid. Choset's cone model (9.41)–(9.42) is an explicit approximation of Elfes's mixture of Gaussians and linear functions. doi:10.1109/ROBOT.1985.1087316

Grisetti, G., Stachniss, C., and Burgard, W. (2007). Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters. *IEEE Transactions on Robotics 23(1), 34–46*.
Algorithm 20 made practical: scan-matched proposals and adaptive resampling, the recipe the vendored GridRbpf follows. doi:10.1109/TRO.2006.889486

Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence 101(1–2), 99–134*.
The POMDP, and the QMDP approximation (first proposed by Littman, Cassandra and Kaelbling in 1995) that this chapter's expected-cost-to-go rule is an instance of — with the honest warning that it never plans to gather information. doi:10.1016/S0004-3702(98)00023-X

Prentice, S. and Roy, N. (2009). The Belief Roadmap: Efficient Planning in Belief Space by Factoring the Covariance. *International Journal of Robotics Research 28(11–12), 1448–1465*.
Planning in the space of Gaussian beliefs by propagating covariance along roadmap edges: the step above the one-step expectation taken here, and below the full POMDP. doi:10.1177/0278364909341659

Macenski, S., Martí n, F., White, R., and Giné s Clavero, J. (2020). The Marathon 2: A Navigation System. *IEEE/RSJ International Conference on Intelligent Robots and Systems*.
Nav2: the fielded instance of this chapter's loop — AMCL as the belief, costmaps and a planner server as the cost-to-go, replanned continuously. link

Thrun, S., Burgard, W., and Fox, D. (2005). Probabilistic Robotics. *MIT Press*.
The depth this chapter links to rather than repeats: histogram and particle filters, beam models, grid localization and MCL, occupancy grids, FastSLAM, MDPs and POMDPs. Its chapters 4, 6, 8, 9, 13, 15 and 16 are the sister volume's 8, 10, 12, 13, 17, 21 and 22. link

Shannon Dynamics (2026). Probabilistic Robotics via Rust — Chapters 5, 8, 10, 12, 13, 17, 21, 22. *Sister volume, web edition*.
The induction proof of the recursion (Ch. 5), histogram and particle filters with KLD sizing (Ch. 8), sensor models (Ch. 10), grid localization and Augmented MCL with the comparison table (Ch. 12), occupancy grids and MAP mapping (Ch. 13), FastSLAM (Ch. 17), and the value function and POMDP this chapter's planner approximates (Chs. 21–22). The filters this chapter wraps are vendored from there. link

Part V

Dynamics, Trajectories, and Constraints

CHAPTER 17

Robot Dynamics

DYNAMICS, TRAJECTORIES, AND CONSTRAINTS · Intermediate · 55 min

> *A path is not a complete description of the motion of a robot system, however, as the timing of the motion is not specified.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), Chapter 10*

Parts I through IV planned *paths*: curves in Q_{free} with no clock attached. A real motor does not execute a curve. It applies a torque, and the configuration that results depends on masses, lever arms, and how fast everything else is already moving. This chapter builds the model that turns geometry into motion for the rest of Part V.

The idea the chapter unpacks fits in one sentence. **The entire dynamics of an arm is encoded in one configuration-dependent symmetric matrix $M(q)$.** Kinetic energy is $\frac{1}{2}\dot{q}^{\mathsf{T}} M(q)\dot{q}$; the Coriolis and centrifugal forces are nothing but derivatives of M — the Christoffel symbols; gravity is the gradient of a scalar. The reader who watched Reach's Jacobian ellipse collapse in Chapter 4 now watches a second ellipse — the inertia ellipse — breathe as the elbow folds, and learns why "force equals mass times acceleration" is false, term by term, in joint coordinates.

Everything here is Choset's Chapter 10, with one addition the 2005 text did not need because it assumed a Mathematica pipeline and never integrated anything: the skew-symmetry of $\dot{M} - 2C$ is turned into a *test oracle*, and the integrator that powers every widget on this page is required to conserve energy to a stated tolerance. If the code and the prose ever disagree, the energy bar settles it.

◎ AFTER THIS CHAPTER YOU CAN

- Derive the equations of motion of Reach from its kinetic and potential energy, and put them in the standard form M(q) $\ddot{q}$ + C(q, $\dot{q}$) $\dot{q}$ + g(q) = τ

- Compute the Christoffel symbols of an inertia matrix, say why C is not unique but C $\dot{q}$ is, and prove that $\dot{M} - 2C$ is skew-symmetric so that energy is conserved exactly
- Read the inertia matrix as a configuration-dependent metric on joint-velocity space, and reproduce the micro-example M = [[2, 1/3], [1/3, 1/3]] at Choset's worked pose
- Impose a Pfaffian constraint A(q) $\dot{q}$ = 0 with Lagrange multipliers, eliminate them with the projection P, and recover Choset's knife-edge equations
- Integrate Euler's equation for a spinning rigid body in the body frame, and explain why ω changes with no torque applied
- Implement the dynamics crate in Rust: a Dynamics trait, the hand-derived 2R model, a Jacobian-assembled 3R model, finite-difference Christoffel symbols, and an RK4 integrator whose energy drift is a unit test

ASSUMED

- Reach's forward kinematics and the three-robot simulator (Chapter 2)
- Reach's Jacobian and its singularities (Chapter 4); SO(3) as a matrix group (Chapter 5)
- The virtual-work identity u = J^T f from the potential-function chapter (Chapter 7)
- Vector calculus: partial derivatives of a quadratic form, the chain rule, the gradient of a scalar

17.1 The problem: a path says nothing about time

Take a path for Reach — a smooth curve on the torus, the kind every planner in Part III hands back — and replay it. Then replay it twice as fast.

The same planned path (purple) replayed on Reach at two speeds, with a torque gauge for each joint. At 1× both gauges stay inside their limits of 20 and 12 N·m. At 2× the accelerations quadruple, the velocity-product terms quadruple, and both motors peg red. The planner that produced this curve never asked what the motors can do.

Nothing about the curve changed. What changed is $\dot{q}$ and $\ddot{q}$, and the torque a joint must supply depends on both — quadratically on $\dot{q}$, which is why doubling the speed does not double the torque but quadruples the part of it that comes from motion. A path is a set of configurations; a trajectory is that set *with a clock*,

and only the clocked version can be feasible or infeasible. Chapter 18 chooses the clock. This chapter builds the thing the clock is measured against: the map from $(q, \dot{q}, \ddot{q})$ to the torques τ that produce them.

17.2 Building intuition

17.2.1 The arm is a different machine in every configuration

Before any formalism, watch Reach fall. The Lagrangian Lab hangs the arm on a vertical Workbench, releases it from a configuration you choose, and keeps the books.

> INTERACTIVE · w17.1
>
> **Lagrangian Lab**
>
> The velocity-product terms are not a small correction you can drop — zero them and energy appears from nowhere. And M $\ddot{q}$ is not a force on its own.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch17-robot-dynamics#w17.1

Three things are worth noticing, and each becomes a theorem in the next section.

Energy is conserved, exactly, by construction. The total $E = K + V$ sits on its dashed reference line while kinetic and potential energy trade places violently. There is no friction in the default model and no torque, so this is what the physics must do — and the integrator reproduces it to about one part in 10^8 over ten seconds. That number is not an accident; it is a unit test, and the end of the chapter prints it.

Switch Coriolis off and the physics lies. The toggle hands the same integrator a model in which the velocity-product vector $C(q, \dot{q})\dot{q}$ has been zeroed — "a small correction," the folklore says. Within two swings the energy bar leaves its line. The arm gains or loses energy from nowhere, by more than twenty percent of what it started with. Those terms are not a correction. They are the only thing that makes joint-space Newton's law consistent with workspace Newton's law.

Neither $M\ddot{q}$ nor $C\dot{q}$ is a force. Watch the torque gauge. Under zero applied torque the three stacks satisfy $M\ddot{q} + C\dot{q} + g = 0$ at every instant, and as the arm swings through different configurations the share carried by $M\ddot{q}$ and by $C\dot{q}$ changes while their sum does not. Choset says this precisely: "neither $M(q)\ddot{q}$ nor $C(q, \dot{q})\dot{q}$ individually should be thought of as a generalized force; only their sum is a force."

17.2.2 Inertia is a metric

The second widget throws the arm away and keeps the matrix.

INTERACTIVE · w17.2

Inertia Ellipsoid

Inertia is not a number. M(q) is a configuration-dependent metric on joint-velocity space, and it is anisotropic.

Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch17-robot-dynamics#w17.2`

The orange curve is the set of joint velocities that carry exactly one joule of kinetic energy, $\{\dot{q} : \dot{q}^{\mathsf{T}} M(q)\dot{q} = 1\}$. If inertia were a number the curve would be a circle. It is an ellipse, its semi-axes are $1/\sqrt{\lambda_i}$ for the eigenvalues of $M(q)$, and as q_2 sweeps from 0 to π the entry M_{11} reads 3, then 2, then 1: the stretched arm is three times harder to swing about the shoulder than the folded one. Move the shoulder instead and nothing happens — there is no q_1 anywhere in M, and the text will ask you to say why.

THE ONE SENTENCE VERSION

Kinetic energy is a quadratic form in $\dot{q}$, and the matrix of that form is $M(q)$. Everything else — the velocity-product forces, the shape of unforced motion, what "orthogonal" means for a constrained mechanism — is a statement about how M changes with q.

$q \in \mathbb{R}^n,\ u \in \mathbb{R}^n$	Generalized coordinates and generalized forces, paired so that $u^{\mathsf{T}}\dot{q}$ is power. For Reach u = τ, the joint torques. Chs. 18–19 keep u for controls.
$L(q,\dot{q}) = K(q,\dot{q}) - V(q)$	The Lagrangian: kinetic minus potential energy.
$M(q),\ C(q,\dot{q}),\ g(q),\ b(q,\dot{q})$	Inertia matrix, a Coriolis matrix, gravity vector, dissipation — the standard form (10.7). Book-wide (TOC §2).
$\Gamma^i_{jk}(q),\ \Gamma^i(q),\ \dot{q}^{\mathsf{T}}\Gamma(q)\dot{q}$	Christoffel symbols of M; the i-th symmetric n×n slice; the velocity-product vector (10.10).
$T(q)f = u$	Actuation matrix and raw actuator forces; T = J^{T} for forces applied at a body point (10.12).
$A(q)\dot{q} = 0,\ \lambda \in \mathbb{R}^k$	k Pfaffian constraints (rows ω_j(q)) and their Lagrange multipliers.
$P,\ P_u$	Projections onto motions that satisfy the constraints / forces that do work (10.18–10.19).
$r_{cm},\ I_{zz},\ \mathcal{I},\ \mathcal{I}_s$	Center of mass; planar inertia about z; body-frame and spatial inertia tensors.
$\omega,\ \hat{\omega},\ R \in \mathrm{SO}(3)$	Body angular velocity, its skew matrix, the orientation.
$E = K + V,\ h$	Total mechanical energy; the integrator step.

17.3 The mathematics

17.3.1 Definitions

Generalized coordinates and forces. A configuration is $q \in \mathbb{R}^n$ locally — joint angles for Reach, (x, y, θ) for a planar body — and a generalized force u is anything that pairs with $\dot{q}$ to give power: $u^{\mathsf{T}}\dot{q}$ is the rate at which work is done on the system. Torques pair with joint rates; a force through the center of mass pairs with its velocity.

The Lagrangian is $L(q, \dot{q}) = K(q, \dot{q}) - V(q)$ with kinetic energy quadratic in the velocities and potential energy a function of configuration alone. The Euler–Lagrange equations (Choset eq. 10.1)

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}} - \frac{\partial L}{\partial q} = u$$

are the equations of motion. Their derivation from the principle of least action is in any mechanics text; this chapter takes them as the recipe.

The inertia matrix $M(q) = \partial^2 K / \partial \dot{q}\, \partial \dot{q}$, so that $K = \tfrac{1}{2}\dot{q}^{\mathsf{T}} M(q)\dot{q}$ (eqs. 10.13–10.14). It is symmetric by construction and positive definite because kinetic energy is positive for every nonzero velocity.

Christoffel symbols. For each i, the $n \times n$ matrix with entries

$$\Gamma^i_{jk}(q) = \tfrac{1}{2}\left(\frac{\partial M_{ij}}{\partial q_k} + \frac{\partial M_{ik}}{\partial q_j} - \frac{\partial M_{jk}}{\partial q_i}\right)$$

(eq. 10.9). Terms with $j = k$ are *centrifugal*; terms with $j \neq k$ are *Coriolis*. There are n^3 of them, symmetric in the lower pair.

A Pfaffian constraint is $A(q)\dot{q} = 0$ for a $k \times n$ matrix of full row rank. It is *holonomic* if each row is $\partial c / \partial q$ for some $c(q) = 0$ — the constraint is really a constraint on configurations in disguise — and *nonholonomic* otherwise. Chapter 20 decides which; this chapter only imposes them.

The inertia tensor of a rigid body with density ρ occupying V, in a frame at its center of mass, is $\mathcal{I} = \int_V \rho(r)\,(\|r\|^2 \mathbb{1} - rr^{\mathsf{T}})\,dV$ (eq. 10.42). Its eigenvectors are the *principal axes*; its eigenvalues the principal moments.

17.3.2 Euler–Lagrange gives the standard form

STATEMENT

For $L = \tfrac{1}{2}\dot{q}^{\mathsf{T}} M(q)\dot{q} - V(q)$, the Euler–Lagrange equations are

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = u, \qquad C_{ij} = \sum_k \Gamma^i_{jk}\,\dot{q}_k, \qquad g = \frac{\partial V}{\partial q}.$$

DERIVATION **From the Lagrangian to M $\ddot{q}$ + C $\dot{q}$ + g = u**

\text{eq. (10.7), with } C \text{ from the Christoffel symbols}

Step 1 — the momentum. $\partial L/\partial \dot{q} = M(q)\dot{q}$, because K is a quadratic form and V has no $\dot{q}$ in it.
Step 2 — its time derivative. $\frac{d}{dt}(M\dot{q}) = M\ddot{q} + \dot{M}\dot{q}$, and since M depends on t only through q, $\dot{M}_{ij} = \sum_k (\partial M_{ij}/\partial q_k)\,\dot{q}_k$.
Step 3 — the configuration derivative. $\partial L/\partial q_i = \frac{1}{2}\dot{q}^{\mathsf{T}}(\partial M/\partial q_i)\dot{q} - \partial V/\partial q_i$.
Step 4 — subtract and symmetrize. The i-th equation reads

$$\sum_j M_{ij}\ddot{q}_j + \sum_{j,k} \frac{\partial M_{ij}}{\partial q_k}\dot{q}_j\dot{q}_k - \tfrac{1}{2}\sum_{j,k} \frac{\partial M_{jk}}{\partial q_i}\dot{q}_j\dot{q}_k + \frac{\partial V}{\partial q_i} = u_i.$$

The middle sum is a quadratic form in $\dot{q}$ whose coefficient matrix $\partial M_{ij}/\partial q_k$ is *not* symmetric in (j,k). A quadratic form only sees the symmetric part, so replace it by $\frac{1}{2}(\partial M_{ij}/\partial q_k + \partial M_{ik}/\partial q_j)$. Together with the third sum that is exactly $\sum_{jk} \Gamma^i_{jk}\dot{q}_j\dot{q}_k$ — the symmetrization *is* where the three-term Christoffel formula comes from.
Step 5 — read off the pieces. Define $c_i = \sum_{jk} \Gamma^i_{jk}\dot{q}_j\dot{q}_k$ (eq. 10.8), $g_i = \partial V/\partial q_i$, and any matrix C with $C\dot{q} = c$; the choice $C_{ij} = \sum_k \Gamma^i_{jk}\dot{q}_k$ is the one the rest of the chapter uses. ■
The RP arm as a check (Choset Examples 10.1.2, 10.2.1). With $M = \mathrm{diag}(I_1 + I_2 + m_1 r_1^2 + m_2 q_2^2,\ m_2)$, the only nonzero symbols are $\Gamma^1_{12} = \Gamma^1_{21} = m_2 q_2$ and $\Gamma^2_{11} = -m_2 q_2$, and the velocity-product vector is $(2m_2 q_2 \dot{q}_1 \dot{q}_2,\ -m_2 q_2 \dot{q}_1^2)$ — eqs. (10.5)–(10.6) exactly. The cast of this book is revolute, so the RP arm survives only here.
C is not unique; $C\dot{q}$ is. Add to C any matrix S with $S\dot{q} = 0$ — for $n = 2$, $S = s\,[\dot{q}_2, -\dot{q}_1]^{\mathsf{T}}[\dot{q}_2, -\dot{q}_1]$ for any scalar s — and the equations of motion are unchanged. What *is* changed is the property the next derivation proves, which is why the Christoffel choice is the one worth naming.

17.3.3 $\dot{M} - 2C$ is skew-symmetric, so energy is conserved exactly

STATEMENT

With $C_{ij} = \sum_k \Gamma^i_{jk}\dot{q}_k$, the matrix $\dot{M} - 2C$ is skew-symmetric. Hence along any motion $\dot{E} = \dot{q}^{\mathsf{T}}(u - b)$: with no applied torque and no dissipation, $E = K + V$ is constant.

DERIVATION **Why the Christoffel choice conserves energy**

$\dot{q}^\top(\dot{M} - 2C)\dot{q} = 0, \quad \dot{E} = \dot{q}^\top(u - b)$

Step 1 — write both pieces with the same partials. From Step 2 above, $\dot{M}_{ij} = \sum_k \frac{\partial M_{ij}}{\partial q_k}\dot{q}_k$. From the definition, $2C_{ij} = \sum_k \left(\frac{\partial M_{ij}}{\partial q_k} + \frac{\partial M_{ik}}{\partial q_j} - \frac{\partial M_{jk}}{\partial q_i}\right)\dot{q}_k$.
Step 2 — the first terms cancel. Subtracting leaves $(\dot{M} - 2C)_{ij} = \sum_k \left(\frac{\partial M_{jk}}{\partial q_i} - \frac{\partial M_{ik}}{\partial q_j}\right)\dot{q}_k$.
Step 3 — what remains is antisymmetric. Swap i and j and the bracket changes sign: $(\dot{M} - 2C)_{ji} = -(\dot{M} - 2C)_{ij}$. So $\dot{q}^\top(\dot{M} - 2C)\dot{q} = 0$ for every $\dot{q}$, since a quadratic form of a skew matrix vanishes identically.
Step 4 — differentiate the energy. $E = \frac{1}{2}\dot{q}^\top M\dot{q} + V$, so $\dot{E} = \dot{q}^\top M\ddot{q} + \frac{1}{2}\dot{q}^\top\dot{M}\dot{q} + \dot{q}^\top g$. Substitute $M\ddot{q} = u - C\dot{q} - g - b$:

$$\dot{E} = \dot{q}^\top(u - b) + \tfrac{1}{2}\dot{q}^\top(\dot{M} - 2C)\dot{q} = \dot{q}^\top(u - b). \qquad \blacksquare$$

Why this is the test the `dynamics` crate runs. The derivation used only (i) the correct M, (ii) the correct Γ obtained from it, and (iii) $g = \partial V/\partial q$. An error in any of the three breaks the identity, and an integrator run from rest with $u = b = 0$ will show it as energy drift. That is one assertion covering the whole derivation. The library check `dynamics: released from rest ... E stays within 1e-6 relative under RK4` is exactly this.
Why dropping C breaks it. With $C \equiv 0$ the Step 4 remainder is $\frac{1}{2}\dot{q}^\top\dot{M}\dot{q}$, which is not zero — it is the rate at which the configuration-dependence of M pumps energy in or out. That is the Lagrangian Lab's lie, and the sign of the drift depends on whether $\dot{M}$ is positive or negative along the motion (Exercise 4).

17.3.4 The 2R arm, worked in full

This is Choset's Problem 10.2 — Reach in a vertical plane — and the model every later chapter in Part V runs.

STATEMENT

For links of length L_i, mass m_i, inertia I_i about a center of mass at distance r_i from joint i, with gravity a_g along $-x_2$ and $h(q) = m_2 L_1 r_2 \sin q_2$:

$$M(q) = \begin{pmatrix} I_1 + I_2 + m_1 r_1^2 + m_2(L_1^2 + r_2^2 + 2L_1 r_2 \cos q_2) & I_2 + m_2 r_2^2 + m_2 L_1 r_2 \cos q_2 \\ I_2 + m_2 r_2^2 + m_2 L_1 r_2 \cos q_2 & I_2 + m_2 r_2^2 \end{pmatrix}, \qquad C\dot{q} = \begin{pmatrix} -h\dot{q}_2^2 - 2h\dot{q}_1\dot{q}_2 \\ h\dot{q}_1^2 \end{pmatrix},$$

$$g(q) = \begin{pmatrix} (m_1 r_1 + m_2 L_1)\, a_g \cos q_1 + m_2 r_2\, a_g \cos(q_1 + q_2) \\ m_2 r_2\, a_g \cos(q_1 + q_2) \end{pmatrix},$$

and the only nonzero Christoffel symbols are $\Gamma^1_{12} = \Gamma^1_{21} = \Gamma^1_{22} = -h$ and $\Gamma^2_{11} = h$.

DERIVATION **The Lagrange recipe applied to Reach**

M, Γ, g of the statement

Step 1 — centers of mass by forward kinematics. Chapter 2's φ evaluated at r_i instead of L_i: $c_1 = r_1(\cos q_1, \sin q_1)$ and $c_2 = (L_1 \cos q_1 + r_2 \cos(q_1 + q_2), L_1 \sin q_1 + r_2 \sin(q_1 + q_2))$.

Step 2 — center-of-mass Jacobians. Differentiating, $J_{c_1} = r_1 \begin{pmatrix} -\sin q_1 & 0 \\ \cos q_1 & 0 \end{pmatrix}$ and

$$J_{c_2} = \begin{pmatrix} -L_1 \sin q_1 - r_2 \sin(q_1 + q_2) & -r_2 \sin(q_1 + q_2) \\ L_1 \cos q_1 + r_2 \cos(q_1 + q_2) & r_2 \cos(q_1 + q_2) \end{pmatrix}.$$

These are Chapter 4's Jacobian construction applied to interior points of the links.

Step 3 — kinetic energy. Each link contributes translation of its center of mass plus rotation about it: $K = \sum_i \frac{1}{2} \left(m_i \|J_{c_i} \dot{q}\|^2 + I_i \omega_i^2 \right)$ with $\omega_1 = \dot{q}_1$ and $\omega_2 = \dot{q}_1 + \dot{q}_2$, i.e. $J_{\omega_1} = (1, 0)$, $J_{\omega_2} = (1, 1)$. Hence

$$M = \sum_i \left(m_i J_{c_i}^{\mathsf{T}} J_{c_i} + I_i J_{\omega_i}^{\mathsf{T}} J_{\omega_i} \right).$$

Multiplying out: $\|J_{c_2} \dot{q}\|^2 = (L_1^2 + r_2^2 + 2 L_1 r_2 \cos q_2)\dot{q}_1^2 + 2(r_2^2 + L_1 r_2 \cos q_2)\dot{q}_1 \dot{q}_2 + r_2^2 \dot{q}_2^2$, using $\cos q_1 \cos(q_1 + q_2) + \sin q_1 \sin(q_1 + q_2) = \cos q_2$. Collect by $\dot{q}_i \dot{q}_j$ and the three entries of M in the statement appear.

Step 4 — differentiate for Γ. Only q_2 appears in M, and only through $\cos q_2$: $\partial M_{11}/\partial q_2 = -2h$, $\partial M_{12}/\partial q_2 = -h$, every other partial is zero. Feeding these into the three-term formula: $\Gamma^1_{12} = \frac{1}{2}(\partial M_{11}/\partial q_2) = -h$; $\Gamma^1_{22} = \frac{1}{2}(2\,\partial M_{12}/\partial q_2 - \partial M_{22}/\partial q_1) = -h$; $\Gamma^2_{11} = \frac{1}{2}(2\,\partial M_{21}/\partial q_1 - \partial M_{11}/\partial q_2) = h$; $\Gamma^2_{12} = \frac{1}{2}(\partial M_{21}/\partial q_2 + \partial M_{22}/\partial q_1 - \partial M_{12}/\partial q_2) = 0$. Contracting, $c_1 = -2h\dot{q}_1\dot{q}_2 - h\dot{q}_2^2$ and $c_2 = h\dot{q}_1^2$ (Problem 10.3).

Step 5 — gravity. $V = m_1 a_g r_1 \sin q_1 + m_2 a_g \left(L_1 \sin q_1 + r_2 \sin(q_1 + q_2) \right)$, and $g = \partial V / \partial q$ is the statement. ■

The same recipe for 3R. Nothing in Steps 1–5 used $n = 2$ except the trigonometric collapse in Step 3. For Reach's three-link variant there are three center-of-mass Jacobians, nine entries of M and 27 symbols — too many to differentiate by hand without error, so the library assembles M from the Jacobians and takes the Christoffel symbols by central differences. The 2R case is the check: the finite-difference symbols agree with the closed form above to 10^{-7} at fifty random states.

The numbers

Fix the micro-example that every later chapter in Part V reuses: $L_1 = L_2 = 1$ m, uniform rods so $r_i = L_i/2$, $m_1 = 2$ kg, $m_2 = 1$ kg, hence $I_1 = m_1 L_1^2/12 = 1/6$ and $I_2 = 1/12$ kg·m^2, $a_g = 9.81$ m/s^2, at Choset's worked pose $q = (\pi/4, \pi/2)$ from Example 3.8.1 — the configuration the reader already knows from Chapter 2. Then $\cos q_2 = 0$ and

$$M(q) = \begin{pmatrix} 2 & 1/3 \\ 1/3 & 1/3 \end{pmatrix} \quad \text{exactly:} \quad M_{11} = \tfrac{1}{6} + \tfrac{1}{12} + 2 \cdot \tfrac{1}{4} + 1 \cdot (1 + \tfrac{1}{4}) = 2,\ M_{22} = \tfrac{1}{12} + \tfrac{1}{4} = \tfrac{1}{3},\ M_{12} = M_{22} + 0.$$

$\det M = 5/9 > 0$ with eigenvalues 2.0642 and 0.2691. Gravity: $g_1 = (1 + 1)(9.81)\cos\frac{\pi}{4} + 0.5 \cdot 9.81 \cos\frac{3\pi}{4} = 1.5 \cdot 9.81 \cdot 0.70711 = 10.405$ N·m and $g_2 = 0.5 \cdot 9.81 \cdot (-0.70711) = -3.468$ N·m. With $\dot{q} = (1, 0)$ rad/s the kinetic energy is $K = \frac{1}{2} M_{11} = 1.000$ J, $h = m_2 L_1 r_2 \sin q_2 = 0.5$, and $C\dot{q} = (0,\ h\dot{q}_1^2) = (0, 0.5)$ N·m: joint 2 must push outward at half a newton-meter just to keep the elbow angle fixed while the shoulder turns. Released from rest at this pose with $\tau = 0$, $E = V = 17.342$ J and must stay there.

WORK IT OUT

Sweep q_2 through $0, \pi/2, \pi$ at the same masses. What is M_{11} at $q_2 = 0$ and at $q_2 = \pi$?

Answer. 3 At $q_2 = 0$ the arm is straight and $M_{11} = 1/4 + 2 \cdot 1/4 + 1 \cdot (1 + 1/4 + 1) = 3$; at $q_2 = \pi$ it is folded back on itself and $M_{11} = 1/4 + 1/2 + 1/4 = 1$. The library check `M11 sweeps 3 → 2 → 1` pins all three.

17.3.5 Constrained dynamics and the projection P

Rusty's axle, Hitch's wheels, a knife-edge on ice: each may move along its heading and spin but not slide sideways. That is a constraint on *velocities*, and it enters the equations of motion through a Lagrange multiplier.

STATEMENT (CHOSET EQS. 10.16–10.20)

With $A(q)\dot{q} = 0$, the constrained equations are $M\ddot{q} + C\dot{q} + g = u + A^{\mathsf{T}}\lambda$ and $A\ddot{q} + \dot{A}\dot{q} = 0$: $n + k$ equations for the $n + k$ unknowns $(\ddot{q}, \lambda)$. Eliminating λ gives $P\ddot{q} = PM^{-1}(u - C\dot{q} - g)$ with $P = \mathbb{1} - M^{-1}A^{\mathsf{T}}(AM^{-1}A^{\mathsf{T}})^{-1}A$ of rank $n - k$.

DERIVATION Eliminating the multipliers

$P\ddot{q} = PM^{-1}(u - C\dot{q} - g)$

Step 1 — differentiate the constraint. $A\dot{q} = 0$ holds for all time, so $\frac{d}{dt}(A\dot{q}) = A\ddot{q} + \dot{A}\dot{q} = 0$ (eq. 10.17).
Step 2 — solve the force balance for $\ddot{q}$. From (10.16), $\ddot{q} = M^{-1}(u + A^{\mathsf{T}}\lambda - C\dot{q} - g)$.
Step 3 — substitute into the differentiated constraint. $\dot{A}\dot{q} + AM^{-1}(u + A^{\mathsf{T}}\lambda - C\dot{q} - g) = 0$.
Step 4 — solve for λ. The $k \times k$ matrix $AM^{-1}A^{\mathsf{T}}$ is invertible because A has full row rank and $M \succ 0$: $\lambda = (AM^{-1}A^{\mathsf{T}})^{-1}\left(-\dot{A}\dot{q} + AM^{-1}(C\dot{q} + g - u)\right)$.
Step 5 — substitute back and factor. Using $-\dot{A}\dot{q} = A\ddot{q}$ and collecting, $(\mathbb{1} - A^{\mathsf{T}}(AM^{-1}A^{\mathsf{T}})^{-1}AM^{-1})(M\ddot{q} + C\dot{q} + g - u) = 0$. Call the bracket P_u (eq. 10.18); then $P = M^{-1}P_uM$ (eq. 10.19) and rearranging gives the statement (eq. 10.20). ■
Orthogonality with respect to M. P and $\mathbb{1} - P$ split a velocity into a part that satisfies the constraints and a part in the constrained directions, and $(P\dot{q})^{\mathsf{T}}M(\mathbb{1} - P)\dot{q} = 0$ for every $\dot{q}$. The inner product is M, not the identity: Choset is explicit that this is "the appropriate one when discussing dynamics," because M carries the metric of coordinates that mix lengths and angles. Also $P = P_u^{\mathsf{T}}$, which the library checks numerically alongside the rank.
The knife-edge (Example 10.3.1). $q = (q_1, q_2, q_3)$ is contact point and heading, $M = \mathrm{diag}(m, m, I)$, $A = [\sin q_3, -\cos q_3, 0]$, $\dot{A} = \dot{q}_3[\cos q_3, \sin q_3, 0]$. Then $P = \begin{pmatrix} \cos^2 q_3 & \sin q_3 \cos q_3 & 0 \\ \sin q_3 \cos q_3 & \sin^2 q_3 & 0 \\ 0 & 0 & 1 \end{pmatrix}$, rank 2, and the closed forms (10.25)–(10.28) follow — including the constraint force $\lambda_1 = (u_2 - m\dot{q}_1\dot{q}_3)\cos q_3 - (u_1 + m\dot{q}_2\dot{q}_3)\sin q_3$. The library solves the $(n + k) \times (n + k)$ system directly and reproduces those four formulas to 10^{-9} at thirty random states. This is Rusty's no-side-slip constraint, and Chapter 20 will show it is nonholonomic.

17.3.6 Euler's equation in the body frame

For a spinning rigid body no choice of three angles gives a smooth global coordinate system — the same fact Chapter 5 made about SO(3) — so Choset does not pick one. He takes $R \in \mathrm{SO}(3)$ as the orientation and the body-frame angular velocity ω as the velocity, and derives the dynamics directly.

STATEMENT (CHOSET EQS. 10.43–10.44)

With $\mathcal{I}$ the constant body-frame inertia tensor,

$$\dot{R} = R\hat{\omega}, \qquad \tau = \hat{\omega}\,\mathcal{I}\,\omega + \mathcal{I}\,\dot{\omega}.$$

In particular $\dot{\omega} \neq 0$ when $\tau = 0$ unless ω is along a principal axis.

DERIVATION From the spatial frame to the body frame

$\tau = \hat{\omega}\mathcal{I}\,\omega + \mathcal{I}\,\dot{\omega}$

Step 1 — kinetic energy in the spatial frame. A point at r_s moves at $\omega_s \times r_s$, so $K = \frac{1}{2}\int_V \rho\,\|\omega_s \times r_s\|^2\,dV = \frac{1}{2}\omega_s^{\mathsf{T}}\mathcal{I}_s\omega_s$ with $\mathcal{I}_s = \int_V \rho(\|r_s\|^2 \mathbb{1} - r_s r_s^{\mathsf{T}})\,dV$ (eqs. 10.34–10.36). Because it is written in a fixed frame, $\mathcal{I}_s$ changes as the body turns.
Step 2 — angular momentum and its rate. $P = \mathcal{I}_s\omega_s$ and $\tau_s = \dot{P}$. The density does not change, so $\dot{\mathcal{I}}_s = \omega_s \times \mathcal{I}_s$ comes only from the rotation, giving $\tau_s = \omega_s \times \mathcal{I}_s\omega_s + \mathcal{I}_s\dot{\omega}_s$ — Euler's equation in the inertial frame (10.37), or $\tau_s = \hat{\omega}_s\mathcal{I}_s\omega_s + \mathcal{I}_s\dot{\omega}_s$ in matrix form (10.38).
Step 3 — change frames. $\omega_s = R\omega$, $\tau_s = R\tau$, $\mathcal{I}_s = R\mathcal{I}R^{\mathsf{T}}$, $\hat{\omega}_s = \dot{R}R^{\mathsf{T}}$, and $\dot{R} = R\hat{\omega}$ (10.39, 10.41).
Step 4 — substitute. $R\tau = \dot{R}R^{\mathsf{T}}R\mathcal{I}R^{\mathsf{T}}R\omega + R\mathcal{I}R^{\mathsf{T}}(\dot{R}\omega + R\dot{\omega})$.
Step 5 — kill one term. $\dot{R}\omega = R\hat{\omega}\omega = R(\omega \times \omega) = 0$.
Step 6 — premultiply by R^{T}. $\tau = R^{\mathsf{T}}\dot{R}\,\mathcal{I}\,\omega + \mathcal{I}\,\dot{\omega} = \hat{\omega}\mathcal{I}\,\omega + \mathcal{I}\,\dot{\omega}$. ■
The principal-axis form (10.45). Align the body frame with the eigenvectors of $\mathcal{I}$ and $\tau_x = I_{xx}\dot{\omega}_x + (I_{zz} - I_{yy})\omega_y\omega_z$, with the two cyclic permutations. Set $\tau = 0$: $\dot{\omega}_x = (I_{yy} - I_{zz})\omega_y\omega_z/I_{xx}$ is zero only if two of the components vanish. Spin about the axis of *intermediate* inertia and a small perturbation in the other two components grows exponentially — the flip the next widget shows. The parallel-axis theorem (10.33 planar, 10.46 spatial) and the composite-body rule $\mathcal{I} = \sum_i R_i(\mathcal{I}_i + m_i(\|r_i\|^2\mathbb{1} - r_i r_i^{\mathsf{T}}))R_i^{\mathsf{T}}$ are functions in the library, not sections here.

INTERACTIVE · w17.3

Tumbling Body

No torque does not mean constant angular velocity. Angular momentum and energy are conserved; ω is not — and about the intermediate axis the brick flips.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch17-robot-dynamics#w17.3

Angular momentum and kinetic energy are conserved — the readout holds both to five decimals over the whole run — while the angular velocity is not. That is not a numerical artifact; it is what the equation says. Choset: "$\dot{\omega}$ may not be zero even if τ is zero. Although the angular momentum and kinetic energy of a rotating body are constant when no external torques are applied, the angular velocity of the body may not be constant."

17.3.7 RK4 on the first-order state, and what its energy drift means

Write $x = (q, \dot{q})$ and $\dot{x} = (\dot{q},\ M^{-1}(u - C\dot{q} - g - b))$. Classical RK4 has local error $O(h^5)$ and global error $O(h^4)$, and the energy error inherits that order. Three honesty items the prose must keep:

1. **RK4 is not symplectic.** Its energy error is bounded by $c\,h^4$ over the horizons we use because the global error is, not because of any structural guarantee. Run it for an hour and it will drift.
2. **The bound is empirical.** The test asserts $|E(t) - E(0)|/E(0) < 10^{-6}$ for $h = 1$ ms over 10 s on the micro-example released from rest; the measured value is about 10^{-8}.
3. **Explicit Euler drifts linearly** and is kept only as the bad example: on the same run it loses track of the energy by more than ten percent. Semi-implicit (symplectic) Euler keeps the error bounded, but at first order the bound is also around ten percent on this violently swinging arm — cheap, not accurate.

17.4 The algorithm

Choset numbers none of these; the names follow his text.

ALGORITHM `lagrange_recipe(links, ϕ, a_g)` COST $O(n^2)$ Jacobian products for M; $O(n^3)$ Christoffel symbols

in link parameters (L_i, r_i, m_i, I_i), forward kinematics, gravitational acceleration

out M(q), Γ(q), g(q) as callables

1: for each link i: $c_i(q) \leftarrow$ position of its center of mass by φ at r_i; $J_{c_i}(q) \leftarrow \partial c_i/\partial q$; $J_{\omega_i} \leftarrow (1,\dots,1,0,\dots,0)$ with i ones
2: $M(q) \leftarrow \sum_i \left(m_i J_{c_i}^{\mathsf{T}} J_{c_i} + I_i J_{\omega_i}^{\mathsf{T}} J_{\omega_i}\right)$
3: $V(q) \leftarrow \sum_i m_i a_g\,[c_i(q)]_y$; $g(q) \leftarrow \sum_i m_i a_g\,[J_{c_i}(q)]_{y,\cdot}$
4: $\Gamma^i_{jk}(q) \leftarrow \frac{1}{2}(\partial_k M_{ij} + \partial_j M_{ik} - \partial_i M_{jk})$, by closed form or central differences
5: return (M, Γ, g)

ALGORITHM `forward_dynamics(q, q̇, u)` COST one n×n solve, O(n³)

in state and generalized force
out $\ddot{q}$

1: $c \leftarrow \dot{q}^{\mathsf{T}}\Gamma(q)\dot{q}$; $r \leftarrow u - c - g(q) - b(q,\dot{q})$
2: return $M(q)^{-1}r$ by Cholesky or pivoted elimination — Choset eq. (11.46)

ALGORITHM `constrained_forward_dynamics(q, q̇, u, A, Ȧ)` COST one (n+k)×(n+k) KKT solve

in state, force, constraint matrix and its rate
out $(\ddot{q}, \lambda)$

1: assemble $\begin{pmatrix} M & -A^{\mathsf{T}} \\ A & 0 \end{pmatrix}\begin{pmatrix} \ddot{q} \\ \lambda \end{pmatrix} = \begin{pmatrix} u - C\dot{q} - g - b \\ -\dot{A}\dot{q} \end{pmatrix}$ — eqs. (10.16)–(10.17)
2: solve; return $(\ddot{q}, \lambda)$. The constraint force on the system is $A^{\mathsf{T}}\lambda$.

ALGORITHM `rk4_step(f, x, h)` COST 4 dynamics evaluations

in $\dot{x} = \mathrm{f(x)}$, state, step
out x(t + h)

1: $k_1 \leftarrow f(x)$; $k_2 \leftarrow f(x + \frac{h}{2}k_1)$; $k_3 \leftarrow f(x + \frac{h}{2}k_2)$; $k_4 \leftarrow f(x + hk_3)$
2: return $x + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$

ALGORITHM `euler_body_step(R, ω, τ, 𝓘, h)` COST O(1)

in orientation as a unit quaternion, body angular velocity, body torque, body inertia tensor, step

out (R, ω) after h

1: state $x \leftarrow (R, \omega)$ with $\dot{R} = \frac{1}{2} R \otimes (0, \omega)$ and $\dot{\omega} = \mathcal{I}^{-1}(\tau - \omega \times \mathcal{I}\,\omega)$ — eqs. (10.43)–(10.44)
2: one `rk4_step` on x; renormalize R

17.5 Implementation in Rust

The `dynamics` crate is introduced here and imported by Chapters 18–21 and 23. Its surface is a trait with four physical callbacks and two derived operations.

RUST **crates/dynamics/src/lib.rs**

```rust
use nalgebra::{SMatrix, SVector};

/// A fully actuated second-order mechanical system in generalized coordinates.
/// Everything downstream (time scaling, iLQR, kinodynamic RRT) talks to this trait.
pub trait Dynamics<const N: usize> {
    /// M(q): symmetric positive definite. We return the full matrix -- the
    /// Cholesky factorization is cheap for N <= 3 and keeps the code readable.
    fn mass(&self, q: &SVector<f64, N>) -> SMatrix<f64, N, N>;

    /// The velocity-product vector C(q, q_dot) q_dot = q_dot^T Gamma(q) q_dot --
    returned as a vector,
    /// because the matrix C is not unique and only the product is a generalized
    /// force (Choset sec.10.2).
    fn velocity_product(&self, q: &SVector<f64, N>, qd: &SVector<f64, N>) -> SVector<f64,
    N>;

    /// g(q) = ?V/?q.
    fn gravity(&self, q: &SVector<f64, N>) -> SVector<f64, N>;

    /// V(q), so E = K + V can be tracked. K = ? q_dot^T M q_dot is provided below.
    fn potential_energy(&self, q: &SVector<f64, N>) -> f64;

    /// Dissipation b(q, q_dot); default: none.
    fn dissipation(&self, _q: &SVector<f64, N>, _qd: &SVector<f64, N>) -> SVector<f64, N>
    {
        SVector::zeros()
    }

    /// q = M-1(u - C q_dot - g - b). Default impl; override only for a faster recursive
    form.
    fn forward(&self, q: &SVector<f64, N>, qd: &SVector<f64, N>, u: &SVector<f64, N>)
        -> SVector<f64, N>
    {
        let rhs = u - self.velocity_product(q, qd) - self.gravity(q) - self.dissipation(q,
    qd);
        self.mass(q).cholesky().expect("M(q) must be positive definite").solve(&rhs)
    }

    /// u = M q + C q_dot + g + b -- the torque that produces a given acceleration.
    fn inverse(&self, q: &SVector<f64, N>, qd: &SVector<f64, N>, qdd: &SVector<f64, N>)
        -> SVector<f64, N>
```

```
    {
        self.mass(q) * qdd + self.velocity_product(q, qd) + self.gravity(q) + self.
    dissipation(q, qd)
    }

    fn kinetic_energy(&self, q: &SVector<f64, N>, qd: &SVector<f64, N>) -> f64 {
        0.5 * qd.dot(&(self.mass(q) * qd))
    }

    fn energy(&self, q: &SVector<f64, N>, qd: &SVector<f64, N>) -> f64 {
        self.kinetic_energy(q, qd) + self.potential_energy(q)
    }
}
```

The 2R model is Derivation 3 typed in. Every expression is a line of the derivation, which is the point: a reader who has done the algebra should recognize the file.

RUST **crates/dynamics/src/reach2r.rs**

```rust
use nalgebra::{Matrix2, Vector2};
use crate::Dynamics;

/// Choset Problem 10.2: planar 2R arm in a vertical plane. Units: kg, m, s.
pub struct Reach2R {
    pub l1: f64, pub r1: f64, pub m1: f64, pub i1: f64,
    pub l2: f64, pub r2: f64, pub m2: f64, pub i2: f64,
    pub a_g: f64, // 0.0 lays the Workbench flat
}

impl Reach2R {
    /// Uniform rods: r_i = L_i/2, I_i = m_i L_i2/12.
    pub fn uniform(l1: f64, m1: f64, l2: f64, m2: f64, a_g: f64) -> Self {
        Self {
            l1, r1: l1 / 2.0, m1, i1: m1 * l1 * l1 / 12.0,
            l2, r2: l2 / 2.0, m2, i2: m2 * l2 * l2 / 12.0,
            a_g,
        }
    }

    /// h(q) = m_2 L_1 r_2 sin q_2 -- every velocity-product term is built from it.
    fn h(&self, q: &Vector2<f64>) -> f64 {
        self.m2 * self.l1 * self.r2 * q[1].sin()
    }

    /// Closed form: the four nonzero symbols Gamma1_1_2 = Gamma1_2_1 = Gamma1_2_2 = -h,
    Gamma2_1_1 = h.
    pub fn christoffel(&self, q: &Vector2<f64>) -> [Matrix2<f64>; 2] {
        let h = self.h(q);
        [Matrix2::new(0.0, -h, -h, -h), Matrix2::new(h, 0.0, 0.0, 0.0)]
    }
}

impl Dynamics<2> for Reach2R {
    fn mass(&self, q: &Vector2<f64>) -> Matrix2<f64> {
        let c2 = q[1].cos();
        let m22 = self.i2 + self.m2 * self.r2 * self.r2;
        let m12 = m22 + self.m2 * self.l1 * self.r2 * c2;
```

```
        let m11 = self.i1 + self.i2 + self.m1 * self.r1 * self.r1
            + self.m2 * (self.l1 * self.l1 + self.r2 * self.r2 + 2.0 * self.l1 * self.r2 * c2);
        Matrix2::new(m11, m12, m12, m22)
    }

    fn velocity_product(&self, q: &Vector2<f64>, qd: &Vector2<f64>) -> Vector2<f64> {
        let h = self.h(q);
        Vector2::new(-h * qd[1] * qd[1] - 2.0 * h * qd[0] * qd[1], h * qd[0] * qd[0])
    }

    fn gravity(&self, q: &Vector2<f64>) -> Vector2<f64> {
        let (c1, c12) = (q[0].cos(), (q[0] + q[1]).cos());
        Vector2::new(
            (self.m1 * self.r1 + self.m2 * self.l1) * self.a_g * c1 + self.m2 * self.r2 * self.a_g * c12,
            self.m2 * self.r2 * self.a_g * c12,
        )
    }

    fn potential_energy(&self, q: &Vector2<f64>) -> f64 {
        self.m1 * self.a_g * self.r1 * q[0].sin()
            + self.m2 * self.a_g * (self.l1 * q[0].sin() + self.r2 * (q[0] + q[1]).sin())
    }
}
```

For arms longer than two links the symbols are differenced rather than derived. The function is generic over anything that can produce $M(q)$, and the 2R closed form is what it is tested against.

RUST **crates/dynamics/src/lagrange.rs**

```
use nalgebra::{SMatrix, SVector};

/// Gamma^i_jk = ?(?M_ij/?q_k + ?M_ik/?q_j - ?M_jk/?q_i) (Choset eq. 10.9), with the
/// partials by central differences. O(n3) symbols from 2n evaluations of M.
/// The 3R arm has no other path; the 2R arm's closed form checks this one.
pub fn christoffel<const N: usize>(
    mass: impl Fn(&SVector<f64, N>) -> SMatrix<f64, N, N>,
    q: &SVector<f64, N>,
    step: f64,
) -> [SMatrix<f64, N, N>; N] {
    let mut d_m = [SMatrix::<f64, N, N>::zeros(); N]; // d_m[k] = ?M/?q_k
    for k in 0..N {
        let (mut qp, mut qm) = (*q, *q);
        qp[k] += step;
        qm[k] -= step;
        d_m[k] = (mass(&qp) - mass(&qm)) / (2.0 * step);
    }
    let mut gamma = [SMatrix::<f64, N, N>::zeros(); N];
    for i in 0..N {
        for j in 0..N {
            for k in 0..N {
                gamma[i][(j, k)] = 0.5 * (d_m[k][(i, j)] + d_m[j][(i, k)] - d_m[i][(j, k)]);
            }
        }
    }
```

```
    gamma
}

/// max |(? - 2C) + (? - 2C)^T| -- zero when C is the Christoffel choice. A
/// diagnostic for a model, not a step of any algorithm.
pub fn skew_symmetry_defect<const N: usize>(
    mass: impl Fn(&SVector<f64, N>) -> SMatrix<f64, N, N> + Copy,
    q: &SVector<f64, N>,
    qd: &SVector<f64, N>,
) -> f64 {
    let gamma = christoffel(mass, q, 1e-6);
    let mut c = SMatrix::<f64, N, N>::zeros();
    let mut m_dot = SMatrix::<f64, N, N>::zeros();
    for i in 0..N {
        for j in 0..N {
            for k in 0..N {
                c[(i, j)] += gamma[i][(j, k)] * qd[k];
            }
        }
    }
    // ? = Sigma_k ?M/?q_k q_dot_k, recomputed here so the two sides share no code path.
    for k in 0..N {
        let (mut qp, mut qm) = (*q, *q);
        qp[k] += 1e-6;
        qm[k] -= 1e-6;
        m_dot += (mass(&qp) - mass(&qm)) / 2e-6 * qd[k];
    }
    let s = m_dot - 2.0 * c;
    (s + s.transpose()).abs().max()
}
```

The integrator is thirty lines, and the test next to it is the chapter's contract.

RUST **crates/dynamics/src/integrate.rs**

```
use nalgebra::SVector;
use crate::Dynamics;

/// One classical RK4 step on x = (q, q_dot) with u held over the step.
pub fn rk4_step<const N: usize>(
    dyn_: &impl Dynamics<N>,
    q: &SVector<f64, N>, qd: &SVector<f64, N>, u: &SVector<f64, N>, h: f64,
) -> (SVector<f64, N>, SVector<f64, N>) {
    let f = |q: &SVector<f64, N>, qd: &SVector<f64, N>| (*qd, dyn_.forward(q, qd, u));
    let (k1q, k1v) = f(q, qd);
    let (k2q, k2v) = f(&(q + k1q * (h / 2.0)), &(qd + k1v * (h / 2.0)));
    let (k3q, k3v) = f(&(q + k2q * (h / 2.0)), &(qd + k2v * (h / 2.0)));
    let (k4q, k4v) = f(&(q + k3q * h), &(qd + k3v * h));
    (
        q + (k1q + k2q * 2.0 + k3q * 2.0 + k4q) * (h / 6.0),
        qd + (k1v + k2v * 2.0 + k3v * 2.0 + k4v) * (h / 6.0),
    )
}

#[cfg(test)]
mod tests {
    use super::*;
    use crate::reach2r::Reach2R;
    use nalgebra::Vector2;
```

```rust
    use std::f64::consts::PI;

    /// Zero torque, zero friction => E constant. The tolerance is empirical
    /// (RK4 is not symplectic), calibrated once and never loosened.
    #[test]
    fn energy_drift_rk4() {
        let arm = Reach2R::uniform(1.0, 2.0, 1.0, 1.0, 9.81);
        let (mut q, mut qd) = (Vector2::new(PI / 4.0, PI / 2.0), Vector2::zeros());
        let e0 = arm.energy(&q, &qd);
        let mut worst = 0.0_f64;
        for _ in 0..10_000 {
            (q, qd) = rk4_step(&arm, &q, &qd, &Vector2::zeros(), 1e-3);
            worst = worst.max(((arm.energy(&q, &qd) - e0) / e0).abs());
        }
        assert!(worst < 1e-6, "RK4 energy drift {worst:e}");
    }
}
```

17.5.1 The worked example, printed

`cargo run --example two_r_numbers -p dynamics` builds `Reach2R::uniform(1.0, 2.0, 1.0, 1.0, 9.81)`, evaluates it at $q = (\pi/4, \pi/2)$ and prints

```
M   = [[2.0000, 0.3333], [0.3333, 0.3333]]      det 0.5556   eig 2.0642 0.2691
g   = [10.4051, -3.4684]
Cq_dot  = [0.0000, 0.5000]        for q_dot = (1, 0)
K   = 1.0000 J    V = 17.3418 J    E = 18.3418 J
M11 sweep over q2 in {0, pi/2, pi}:  3.0000  2.0000  1.0000
energy drift, tau = 0, h = 1 ms, 10 s:  rk4 9.4e-9   semi-implicit Euler 1.1e-1
   explicit Euler 1.4e-1
```

`#[test] fn reproduces_micro_example()` asserts each line to 10^{-4}. Three more tests pin the chapter's claims: `christoffel_closed_form_matches_finite_differences` (2R closed form against central differences with step 10^{-6}, agreement to 10^{-7}), `energy_drift_rk4` above (with the explicit Euler run asserted to be *worse* than 10^{-3}, so the comparison in the widget is honest), and `rapier_agrees`, which assembles Reach in `rapier2d` as two rigid bodies joined by revolute joints and checks that one engine step at $\Delta t = 10^{-4}$ agrees with `forward` to 10^{-3} relative, and 2 s trajectories to 10^{-2} — the engine's own integrator sets that tolerance, and the text says so.

> **NOTE**
>
> The widgets on this page run the TypeScript port in `lib/dynamics/`, a line-for-line translation of the Rust above. Its self-checks reproduce every number in the printout: `M = [[2, 1/3], [1/3, 1/3]]`, `g = (10.4051, −3.4684)`, `C` $\dot{q}$ `= (0, 0.5)`, the $3 \to 2 \to 1$ sweep, the eigenvalues, the energy drift, the knife-edge closed forms, and the brick that flips.

17.6 Putting it together

The integration lab drops the same `Reach2R` into Chapter 2's Workbench with gravity on and a PD controller holding a planned waypoint:

$$\tau = g(q^*) + K_p(q^* - q) - K_d\,\dot{q}.$$

Hold the micro-example pose. The gravity compensation term alone is $g(q^*) = (10.405, -3.468)$ N·m — more than half the shoulder motor's 20 N·m budget just to stand still. Raise K_p and the arm overshoots on release; the overshoot is governed by $M(q)$, which is why a gain that is crisp with the arm folded ($M_{11} = 1$) rings with it stretched ($M_{11} = 3$). The torque trace this lab produces, peak and all, is the limit Chapter 18's time scaler inherits.

Here is where the chapter closes and the next one opens. Take any planned path $q(s)$, substitute $\dot{q} = q'\dot{s}$ and $\ddot{q} = q''\dot{s}^2 + q'\ddot{s}$ into the standard form, and the n equations collapse to

$$a(s)\,\ddot{s} + b(s)\,\dot{s}^2 + c(s) = \tau, \qquad a = Mq',\ b = Mq'' + q'^{\mathsf{T}}\Gamma q',\ c = g.$$

A path plus these equations is a *one-dimensional* dynamics in the path parameter. The whole of Chapter 18 lives in the $(s, \dot{s})$ plane that equation defines. Chapter 19 linearizes `forward` along a trajectory for iLQR. Chapter 20 reuses $A(q)\dot{q} = 0$ and asks whether the knife-edge constraint can be integrated. All three import this crate and none of them re-derives a line of it.

17.7 Exercises

1 FOUNDATION •• **The three-term formula, and what breaks without it**

Starting from $K = \frac{1}{2}\dot{q}^{\mathsf{T}}M(q)\dot{q}$, carry out Step 4 of the first derivation and show that the symmetrization produces exactly $\Gamma^i_{jk} = \frac{1}{2}(\partial_k M_{ij} + \partial_j M_{ik} - \partial_i M_{jk})$. Then take $C' = C + S$ with $S\dot{q} = 0$ — for $n = 2$, $S = s\,ww^{\mathsf{T}}$ with $w = (\dot{q}_2, -\dot{q}_1)$ — and show the equations of motion are unchanged while $\dot{M} - 2C'$ is skew only if $s = 0$.

> **HINTS**
>
> <>A quadratic form $\sum_{jk} B_{jk}\dot{q}_j\dot{q}_k$ only depends on the symmetric part $\frac{1}{2}(B + B^{\mathsf{T}})$.</> <>For the second part, $\dot{M} - 2C' = (\dot{M} - 2C) - 2S$, and $S = s\,ww^{\mathsf{T}}$ is symmetric. A symmetric matrix is skew only if it is zero.</>

2 FOUNDATION • **Choset Problem 10.1: a point mass in polar coordinates**

With $q = (r, \theta)$ and $K = \frac{1}{2}m(\dot{r}^2 + r^2\dot{\theta}^2)$, write $M(q)$, compute the Christoffel symbols, and verify $\Gamma^1_{22} = -mr$ and $\Gamma^2_{12} = \Gamma^2_{21} = mr$. Then show that the unforced motions $u = 0$ are straight lines in the Cartesian plane — so the symbols are not describing a force but the bending of straight lines in these coordinates.

WORK IT OUT

For m = 2 kg at r = 1.5 m, what is $\Gamma^2{}_{12}$?

Answer. 3 kg·m $M = \mathrm{diag}(m, mr^2)$, so $\partial M_{22}/\partial r = 2mr$ and $\Gamma^2_{12} = \frac{1}{2}\,\partial M_{22}/\partial q_1 = mr = 3$. The centrifugal $\Gamma^1_{22} = -\frac{1}{2}\,\partial M_{22}/\partial r = -mr$ is the familiar $-mr\dot{\theta}^2$.

3 CONCEPTUAL •• **Predict the ellipse**

PREDICT FIRST

In the Inertia Ellipsoid the semi-axes at $q_2 = \pi/2$ are 1.928 and 0.696. At $q_2 = \pi$, where $M_{11} = 1$ and $M_{12} = 1/3 - 1/2 = -1/6$, what happens to the two semi-axes?

- Both grow — the folded arm is lighter in every direction
  M_{11} drops from 2 to 1, so the shoulder axis does get easier; but M_{22} is unchanged and the coupling M_{12} changes sign. The eigenvalues move toward each other (1.039 and 0.294), and $1/\sqrt{\lambda}$ gives 0.981 and 1.844: the short axis grows, the long axis shrinks.
- ✓ The short axis grows, the long axis shrinks, and the long axis rotates
  Eigenvalues 1.039 and 0.294 give semi-axes 0.981 and 1.844. With M_{12} negative the eigenvector of the small eigenvalue tilts into the quadrant where $\dot{q}_1$ and $\dot{q}_2$ have the same sign — the cheap direction is now to swing both joints together.
- Nothing: the ellipse only depends on the masses
  M depends on q_2 through cos q_2 in M_{11} and M_{12}. The ellipse is a function of configuration — that is the whole point of the widget.

Now move the slider and read the printed semi-axes. The long axis rotates because the off-diagonal entry M_{12} changed sign: at $q_2 = \pi/2$ the two joints are inertially coupled positively, at $q_2 = \pi$ negatively, and the eigenvector of the small eigenvalue — the direction a joule of energy buys the most speed — swings across the $\dot{q}_1$ axis.

4 CONCEPTUAL •• **Which way does the lie go?**

In the Lagrangian Lab switch Coriolis off and release from $(\pi/4, \pi/2)$. Does E increase or decrease first, and why does the sign depend on the release configuration? Confirm with the torque-gauge stacks.

> **HINTS**
>
> <>With C zeroed, Derivation 2 leaves $\dot{E} = \frac{1}{2}\dot{q}^{\mathsf{T}}\dot{M}\dot{q}$. The sign of $\dot{E}$ is the sign of $\dot{M}_{11}\dot{q}_1^2 + \ldots$, dominated by $\partial M_{11}/\partial q_2 \cdot \dot{q}_2 = -2h\dot{q}_2$.</> <>So the lie adds energy while the elbow is closing ($\dot{q}_2 < 0$, $h > 0$) and removes it while the elbow opens. From $(\pi/4, \pi/2)$ gravity first swings the elbow one way; the release pose decides which.</> <>The amber stack in the gauge is $C\dot{q}$ of the honest model, i.e. the torque the lying model is silently failing to account for. Its sign on joint 2 is the sign of $h\dot{q}_1^2$.</>

5 PRACTICAL •• **Reach3R from three Jacobians**

Implement `Reach3R` by assembling $M(q) = \sum_i (m_i J_{c_i}^{\mathsf{T}} J_{c_i} + I_i J_{\omega_i}^{\mathsf{T}} J_{\omega_i})$ from three center-of-mass Jacobians, and g from the y-rows of the same Jacobians. Add a property test that $M(q)$ is positive definite on 1000 seeded configurations (`SmallRng`), that `skew_symmetry_defect` is below 10^{-6} with finite-difference symbols, and that `inverse(forward(q, q̇, τ))` returns τ to 10^{-9}. The TypeScript `PlanarArm.reach3R()` does the same with links of 0.6, 0.5, 0.4 m and 1.2, 1.0, 0.8 kg; its check `M(q) is positive definite at 200 seeded configurations of each arm` is the one to match.

6 PRACTICAL ••• **Rusty's chassis as a knife-edge (stretch)**

Implement Rusty's chassis as a planar rigid body with the no-side-slip Pfaffian constraint of Example 10.3.1 using `constrained_forward_dynamics`. Verify λ against eq. (10.28) and show that the reduced two-equation form of Choset p. 361, $\ddot{q}_1 \cos q_3 + \ddot{q}_2 \sin q_3 = (u_1 \cos q_3 + u_2 \sin q_3)/m$ and $\ddot{q}_3 = u_3/I$, emerges from $P\ddot{q} = PM^{-1}u$. Then drive it: with $u_3 \neq 0$ and $u_1 = u_2 = 0$ the body turns in place, and the constraint force $\lambda_1 = -m\dot{q}_1\dot{q}_3 \cos q_3 - m\dot{q}_2\dot{q}_3 \sin q_3$ is the centripetal force the floor supplies. Chapter 20 will show this constraint cannot be integrated to a constraint on q.

17.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

Chapter 10 is the source of everything here: the Lagrangian recipe, the standard form and Christoffel symbols (§10.2), Pfaffian constraints and the projections P and P_u (§10.3), and Euler's equation in both frames (§10.4). Our 2R arm is Problems 10.2–10.3 worked in full. link

Murray, R. M., Li, Z., and Sastry, S. S. (1994). A Mathematical Introduction to Robotic Manipulation. *CRC Press*.

Chapter 4 derives the manipulator equations with the Christoffel symbols in the index convention used here, and proves the skew-symmetry of $\dot{M} - 2C$ that this chapter turns into a test oracle. link

Lynch, K. M. and Park, F. C. (2017). Modern Robotics: Mechanics, Planning, and Control. *Cambridge University Press*.

Chapter 8 covers the same dynamics with the Newton–Euler recursion as the computational alternative to the Lagrangian route taken here; written by one of Choset's co-authors. link

Featherstone, R. (2008). Rigid Body Dynamics Algorithms. *Springer*.

The composite-rigid-body and articulated-body algorithms that build M(q) and solve forward dynamics in O(n) — what replaces our $O(n^3)$ Jacobian assembly once an arm has more than a handful of joints. doi:10.1007/978-1-4899-7560-7

Carpentier, J., Saurel, G., Buondonno, G., Mirabel, J., Lamiraux, F., Stasse, O., and Mansard, N. (2019). The Pinocchio C++ library: A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives. *IEEE/SICE International Symposium on System Integration (SII)*.

A production rigid-body library that assembles the same quantities this chapter hand-rolls, with analytic derivatives — the modern form of the 'build M from link Jacobians' pattern behind Reach3R. doi:10.1109/SII.2019.8700380

Hairer, E., Lubich, C., and Wanner, G. (2006). Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations. *Springer Series in Computational Mathematics 31, 2nd edition*.

Why RK4 is not symplectic and what symplectic integrators preserve instead — the theory behind the honesty item that the energy-drift bound is empirical. doi:10.1007/3-540-30666-8

CHAPTER 18

Trajectory Planning

DYNAMICS, TRAJECTORIES, AND CONSTRAINTS · Advanced · 70 min

> *This kind of trajectory is called a "bang-bang" trajectory, and at least one of the actuators is always saturated. The heart of the time-scaling problem is to find the switching points between maximum and minimum acceleration.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), §11.2*

Every planner in Parts II and III handed Reach a curve and walked away. The curve had no clock. This chapter attaches one, and the first thing it discovers is that the best clock is almost never uniform. Run a path at constant speed and some motor pegs where the configuration is heavy and idles where it is light; run it on the clock this chapter computes and exactly one motor is at its limit at every instant, in turn.

The idea that makes this tractable fits in one sentence. **Once the path is fixed, an n-joint arm with n torque limits is a one-dimensional system in the path parameter** s, and its time-optimal execution is a curve in the $(s, \dot{s})$ phase plane that hugs a velocity-limit curve and switches between maximum and minimum acceleration a finite number of times. A reader who can read that phase plane can read every time-scaling result since 1985, including the 2018 reformulation that replaced switch-point hunting with interval propagation.

The second half of the chapter asks what happens when the path is *not* fixed, and meets the honest ancestor of every state-lattice planner: Choset's grid search over $(q, \dot{q})$, run here on Chapter 6's A*. It is time-optimal for its own discretization, complete in a precise ϵ-sense, and exponential in the number of joints — which is exactly why Chapter 19 lets the path move by optimization instead.

◎ AFTER THIS CHAPTER YOU CAN

- Substitute a path q(s) into the Chapter 17 equations and reduce an n-DOF arm to a(s) $\ddot{s}$ + b(s) $\dot{s}^2$ + c(s) = u — one degree of freedom in the path parameter
- Read the (s, $\dot{s}$) phase plane: motion cones, the velocity-limit curve where the cone collapses, the inadmissible region above it, and why the time-optimal clock is bang-bang
- Run the Time-Scaling Algorithm of Bobrow–Dubowsky–Gibson and Shin–McKay as Choset states it, including the zero-inertia modification and the footnote-3 bisection for tangent points
- Reformulate the same problem as TOPP-RA: reachable intervals in x = $\dot{s}^2$ propagated backward by two-variable linear programs, then a greedy forward pass
- Plan directly in (q, $\dot{q}$) with Choset's GRID SEARCH, state Theorem 11.3.2 precisely, and see A* with a bang-bang heuristic return the same lattice time with far fewer expansions
- Implement the trajectory crate in Rust: Path, PathDynamics, TorqueLimits, time_scale, toppra, lattice_search, and a Trajectory type that no controller can confuse with a Path

ASSUMED

- The standard form M(q) $\ddot{q}$ + C(q, $\dot{q}$) $\dot{q}$ + g(q) = u, Christoffel symbols, and the RK4 integrator (Chapter 17)
- Best-first search with a priority knob: breadth-first, Dijkstra, A* (Chapter 6)
- Paths on the torus and the chart lift across the seam (Chapter 5); the planners that produce paths (Chapters 11–13)
- Ordinary differential equations in one variable; the chain rule for q(s(t))

18.1 The problem: one path, two clocks

Take the spline below — three waypoints on Reach's torus, the arm hanging below its shoulder on a vertical Workbench — and execute it twice in the same 0.62 s. The left arm runs a *uniform* clock: accelerate briefly, cruise at constant $\dot{s}$, brake briefly. The right arm runs the clock this chapter computes.

The same path (purple) on the same arm with the same motors, both finishing in 0.618 s. The uniform clock spends its acceleration where the clock says and pegs a motor; the time-scaled clock spends it where the configuration can afford it, and at every instant exactly one gauge is full and neither exceeds its rail.

Nothing about the curve differs between the two lanes. What differs is $s(t)$: the map from time to position along the path. Choset's §11.1 makes the distinction a definition. A **path** is a twice-differentiable curve $q : [0, 1] \to \mathcal{Q}$. A **time scaling** is a twice-differentiable, monotone map $s : [0, t_f] \to [0, 1]$ with $\dot{s} > 0$ on $(0, t_f)$. A **trajectory** is their composition $q(s(t))$. Uniform time scalings $s(t) = kt$ are a thin subset of the ones allowed, and the question of this chapter is: of all admissible clocks, which is fastest?

18.2 Building intuition

18.2.1 The phase plane is a speed-limit sign that changes every meter

The whole problem lives in one picture. Put s on the horizontal axis and $\dot{s}$ on the vertical. A time scaling is a curve from $(0, \dot{s}_0)$ to $(1, \dot{s}_f)$ that never dips below $\dot{s} = 0$. At every state $(s, \dot{s})$ the torque limits allow a range of path accelerations $L \le \ddot{s} \le U$ — a *cone* of directions the curve may take. Above a certain speed the cone closes: there is no torque vector that keeps the arm on the path. That locus is the **velocity-limit curve**, and the region above it is **inadmissible**, hatched amber in the widget exactly as a configuration-space obstacle would be — it is an obstacle in $(s, \dot{s})$.

INTERACTIVE · w18.1

Phase-Plane Racer

The speed limit along a path is a coupled function of configuration and speed, not the slowest joint's limit — and a greedy controller cannot pass a narrow waist.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch18-trajectory-planning#w18.1

Three things to notice, each of which becomes a theorem in the next section.

The limit is coupled, not per-joint. Drag a waypoint so the elbow folds sharply mid-path and the amber region dips to a waist. No single joint is near its own speed limit there; what has happened is that the centrifugal torque one joint needs, $h\dot{q}_1^2$, is a torque the other joint's motor must supply, and above some $\dot{s}$ it cannot while the first joint is also braking at full torque. The dashed grey ghost is a constant-speed clock; wherever it turns amber, that clock is infeasible, and no controller that only looks at the current joint speeds will know why.

The optimal curve is made of two kinds of arc. Solid blue curves are integrated forward at the maximum acceleration U; the dashed blue curve F is integrated backward from the goal at the minimum acceleration L. The orange profile is a concatenation of pieces of them, switching at the ticked points. The fastest trajectory is the one that is *highest* in the plane while staying out of the amber: $t_f = \int_0^1 ds/\dot{s}$, so area under the curve is time saved.

Zero-inertia points are where the picture changes shape. The vertical dashed amber lines mark s values where some $a_i(s) = (Mq')_i$ changes sign: that actuator momentarily cannot change $\ddot{s}$ at all, and instead bounds $\dot{s}$ directly. The limit curve is continuous through such a point but its slope is not, and the optimal profile may have to *slide* along it rather than bounce off. Raise both torque limits by a factor of 1.5 on the default spline and a slide appears near $s = 0.57$.

18.2.2 One motor at a time

The design question most readers bring to this chapter is: "time-optimal means every motor at full power, right?" The Saturation Scope answers it with the torque histories of the solution above.

> **INTERACTIVE · w18.3**
>
> **Saturation Scope**
>
> Time-optimal does not mean every motor at full power: exactly one actuator is saturated at a time, the switch is where the saturated joint changes, and on a singular arc neither is.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch18-trajectory-planning#w18.3

At every instant one strip rides its rail and the other does whatever the coupling demands; the switch is the instant the saturated joint changes. Scale the limits to 1.5× and the amber stretch is a **singular arc**: the profile slides along the velocity-limit curve with an acceleration strictly between L and U, and *neither* motor is saturated. Choset's §11.2.1 derives exactly when that happens and what acceleration to use.

> **THE ONE-SENTENCE VERSION**
>
> Fix the path and the n torque limits become $2n$ bounds on a single scalar acceleration $\ddot{s}$. The time-optimal clock maximizes $\dot{s}$ pointwise subject to those bounds, which makes it bang-bang in $\ddot{s}$ — and finding it is a boundary-value problem in a two-dimensional plane that can be drawn, read, and solved by integration.

$s \in [0,1]$, $s(t)$, $\dot{s}$, $\ddot{s}$	Path parameter; the time scaling s : [0, t_f] → [0, 1] and its rates. Choset §11.1
$q(s)$, $q' = dq/ds$, q''	The path and its derivatives in s. q″ must be continuous — b(s) contains it.
$a(s)$, $b(s)$, $c(s)$	Inertial, velocity-product and gravity vectors of the path-constrained dynamics a s̈ + b ṡ² + c = u (eq. 11.6).
$u_i^{\min} \le u_i \le u_i^{\max}$	Actuator limits, possibly state dependent (eq. 11.1). Symmetric constant bounds |u_i| ≤ u_i^max are the default here.
α_i, β_i; L_i, U_i; $L(s,\dot{s})$, $U(s,\dot{s})$	Per-actuator acceleration bounds (eq. 11.8), their assignment by the sign of a_i, and the max/min over i (eq. 11.9).
$v(s)$, $\dot{s}^{\max}(s)$, $\dot{s}^{\max}_{zip}(s)$	Velocity-limit curve where L = U (eq. 11.10); its generalization through zero-inertia points; the direct bound from a zero-inertia row (eq. 11.11).
F, A_i, $\mathcal{S} = \{s_1, s_2, \dots\}$	The backward minimum-acceleration curve; the forward curves of the construction; the switch list.
$(s_{lim}, \dot{s}_{lim})$, $(s_{tan}, \dot{s}_{tan})$, $\ddot{s}^{\pm}_{tangent}$	Where A_i penetrates the limit curve; the tangent point found from it; the left and right tangent accelerations at a singular point.
$x = \dot{s}^2$	Squared path speed — the variable in which every actuator limit is linear (TOPP-RA). Pham & Pham 2018
h, $\mathcal{A}$, $\mathcal{A}(q,\dot{q})$, $\hat{\mathcal{A}}$	Lattice timestep; the discretized control set $\{-a_{max}, 0, a_{max}\}^n$; the feasible-acceleration parallelepiped of a manipulator and its one-step conservative subset. Choset §11.3.3
$\delta_v(c_0, c_1)$, ϵ, T_{opt}	Speed-dependent safety margin $c_0 + c_1\|\dot{q}\|$; the approximation parameter of Theorem 11.3.2; the optimal time.

18.3 The mathematics

18.3.1 Definitions

Path, time scaling, trajectory are as in the hook: $q : [0,1] \to \mathcal{Q}$ twice differentiable; $s : [0, t_f] \to [0,1]$ twice differentiable and monotone with $\dot{s} > 0$ on the open interval; the trajectory is $q(s(t))$. Twice differentiability of s is what makes $\ddot{q}$ exist and be bounded, which is what makes torque finite.

The motion cone at $(s, \dot{s})$ is the set of tangent directions of curves through that state whose path acceleration satisfies $L(s,\dot{s}) \le \ddot{s} \le U(s,\dot{s})$. The state is **inadmissible** when $L > U$: the cone is empty and the robot is doomed to leave the path immediately. At an admissible state the robot may still be doomed *eventually*, if every curve inside the cones from there reaches the inadmissible region (Choset's figure 11.2).

The velocity-limit curve $v(s)$ is the locus where $L = U$ — the cone collapses to one vector. It is computed by equating $L_i = U_j$ for every pair of actuators, solving each for $\dot{s}$, and keeping the minimum.

A bang-bang time scaling has $\ddot{s} \in \{L, U\}$ at every instant except possibly on singular arcs. At least one actuator is saturated at all times.

Zero-inertia, critical, singular. A point where $a_i(s) = 0$ for some i is a *zero-inertia point*: actuator i cannot affect $\ddot{s}$ and instead bounds $\dot{s}$ directly. A point of the limit curve $\dot{s}^{\max}$ set by such a bound is *critical* ($L < U$ there, unlike on v). A critical point from which integrating U forward or L backward penetrates the limit curve immediately is *singular*.

A $\delta_v(c_0, c_1)$-safe trajectory keeps clearance at least $c_0 + c_1\|\dot{q}\|$ from every obstacle at every instant — the faster, the wider the berth.

18.3.2 Path-constrained dynamics

STATEMENT (CHOSET EQS. 11.2–11.6)

Along $q(s)$, the standard form $M\ddot{q} + \dot{q}^{\mathsf{T}}\Gamma\dot{q} + g = u$ becomes

$$a(s)\,\ddot{s} + b(s)\,\dot{s}^2 + c(s) = u, \qquad a = M(q)\,q', \quad b = M(q)\,q'' + q'^{\mathsf{T}}\,\Gamma(q)\,q', \quad c = g(q).$$

DERIVATION Substituting the path into the Chapter 17 equations

$a(s)\ddot{s} + b(s)\dot{s}^2 + c(s) = u$

Step 1 — the chain rule. With $q = q(s(t))$, $\dot{q} = q'\dot{s}$ and $\ddot{q} = q''\dot{s}^2 + q'\ddot{s}$ (eqs. 11.2–11.3). The first is linear in $\dot{s}$; the second has a term in $\dot{s}^2$ and a term in $\ddot{s}$, and nothing else.

Step 2 — substitute. Into $M(q)\ddot{q} + \dot{q}^{\mathsf{T}}\Gamma(q)\dot{q} + g(q) = u$:

$$M\,(q''\dot{s}^2 + q'\ddot{s}) + (q'\dot{s})^{\mathsf{T}}\,\Gamma\,(q'\dot{s}) + g = u.$$

Step 3 — group by powers of the clock. The velocity-product term is bilinear in $\dot{q}$, so the two factors of $\dot{s}$ come out as $\dot{s}^2$:

$$\underbrace{Mq'}_{a(s)}\ \ddot{s} + \underbrace{\left(Mq'' + q'^{\mathsf{T}}\Gamma q'\right)}_{b(s)}\ \dot{s}^2 + \underbrace{g}_{c(s)} = u.$$

Every coefficient depends on s alone. The n equations now constrain one scalar unknown $\ddot{s}$ at each state $(s, \dot{s})$ — that is the whole content of path-velocity decomposition.

The velocity-product piece of b is exactly Chapter 17's `velocityProduct( q, q')`, the vector $C(q, \dot{q})\dot{q}$ evaluated at $\dot{q} = q'$. No new physics is computed here; the library function `PathDynamics::abc` calls the Chapter 17 model and does the two matrix–vector products.

Why b_i can vanish when a_i does not, and vice versa (Choset Problem 11.5). $a_i = (Mq')_i$ is zero when the path's tangent is orthogonal, in the M-metric's i-th row, to joint i — the RP arm's prismatic joint at the midpoint of a straight Cartesian line, where the radial velocity is momentarily zero. b_i collects Mq'' and the centrifugal/Coriolis terms, which can vanish on a straight segment ($q'' = 0$) with no Coriolis coupling even where $a_i \neq 0$. The micro-example below has $b_1 = 0$ and $a_1 = \pi$ at every s; Choset's RP arm has $a_2(\frac{1}{2}) = 0$ and $b_2 \equiv 0$.

The micro-example. Reach from the Chapter 17 micro-example ($L_i = 1$, $r_i = \frac{1}{2}$, $m_1 = 2$, $m_2 = 1$, $I_1 = \frac{1}{6}$, $I_2 = \frac{1}{12}$) on a *horizontal* Workbench ($a_g = 0$), joint 2 held at $\pi/2$ by its own motor while joint 1 sweeps a quarter turn: $q(s) = (\pi s/2,\ \pi/2)$, so $q' = (\pi/2, 0)$, $q'' = 0$, and M is constant along the path with $M = \left[\begin{smallmatrix} 2 & 1/3 \\ 1/3 & 1/3 \end{smallmatrix}\right]$ from Chapter 17. Then

$$a(s) = Mq' = \left(\pi,\ \tfrac{\pi}{6}\right) = (3.1416,\ 0.5236), \qquad b(s) = \left(\Gamma^1_{11},\ \Gamma^2_{11}\right)\tfrac{\pi^2}{4} = (0,\ 0.5 \cdot 2.4674) = (0,\ 1.2337), \qquad c(s) = 0.$$

Joint 2's $b_2 = h\,(q_1')^2$ is the centrifugal torque the elbow motor must supply to keep the elbow at $\pi/2$ while the shoulder swings — the coupling that will set the speed limit.

18.3.3 Acceleration bounds and the velocity-limit curve

STATEMENT (CHOSET EQS. 11.8–11.10)

With $\alpha_i = (u_i^{\max} - b_i\dot{s}^2 - c_i)/a_i$ and β_i the same with $u_i^{\min}$: $U_i = \alpha_i, L_i = \beta_i$ if $a_i > 0$ and swapped if $a_i < 0$; $L = \max_i L_i$, $U = \min_i U_i$; the feasible set is $L \le \ddot{s} \le U$, and it is empty above $v(s) = \min_{i \neq j} \dot{s}_{ij}(s)$ where $\dot{s}_{ij}$ solves $L_i = U_j$.

DERIVATION From torque limits to a cone, and from the cone to the curve

$L(s,\dot{s}) \le \ddot{s} \le U(s,\dot{s}); \quad v(s) = \min_{i,j} \dot{s}_{ij}(s)$

Step 1 — isolate $\ddot{s}$ in each row. Row i of $u^{\min} \le a\ddot{s} + b\dot{s}^2 + c \le u^{\max}$ reads $u_i^{\min} - b_i\dot{s}^2 - c_i \le a_i\ddot{s} \le u_i^{\max} - b_i\dot{s}^2 - c_i$.
Step 2 — divide, and flip if negative. Dividing by $a_i > 0$ gives $\beta_i \le \ddot{s} \le \alpha_i$. Dividing by $a_i < 0$ reverses the inequality: $\alpha_i \le \ddot{s} \le \beta_i$. That swap is the whole content of eq. 11.8, and it is why the sign of a_i matters: a joint whose coordinate decreases along the path has its "maximum torque" bound on the *low* side of $\ddot{s}$.

Step 3 — intersect the intervals. $\ddot{s}$ must satisfy all n rows, so it lies in $[\max_i L_i, \min_i U_i] = [L, U]$. Each L_i and U_i is an affine function of $x = \dot{s}^2$ at fixed s: $L_i = p_i + m_i x$ with $m_i = -b_i/a_i$. The max of affine functions is convex and piecewise affine; the min is concave. Neither is smooth, so neither is v.

Step 4 — the boundary of admissibility. The cone is empty when $L > U$, i.e. when some $L_i > U_j$. At fixed s, as $\dot{s}$ grows from zero, the first value at which some pair crosses, $L_i(\dot{s}) = U_j(\dot{s})$, is where admissibility is lost. Because each side is affine in x, the crossing is in closed form, $x_{ij} = (p_j^U - p_i^L)/(m_i^L - m_j^U)$, and $v(s) = \min$ over pairs of $\sqrt{x_{ij}}$ taken over positive roots. The minimum rule is exact rather than heuristic: if any $L_i = U_j$ then $L \geq L_i = U_j \geq U$, so the state is already on or past the boundary.

What the closed form assumes (Choset footnote 2). That $\dot{s} = 0$ is admissible for every s — the robot can hold any configuration statically — and that admissibility is lost at most once as $\dot{s}$ grows. Friction, weak actuators, or velocity-dependent limits can violate this and create inadmissible *islands* in the plane. The algorithm below ignores islands; the library reports the first assumption's failure as an error rather than guessing.

A check that is not a frozen number. `velocityLimit` is compared against a brute-force scan of $L \leq U$ with bisection on 150 seeded random (a, b, c) rows in two and three dimensions; the worst disagreement is 4×10^{-14}.

Back to the micro-example, limits $|\tau_1| \leq 20$, $|\tau_2| \leq 12$ N·m. Joint 1: $U_1 = 20/\pi = 6.366$, $L_1 = -6.366$, independent of $\dot{s}$ because $b_1 = c_1 = 0$. Joint 2: $U_2 = (12 - 1.2337\dot{s}^2)/0.5236 = 22.918 - 2.3562\dot{s}^2$ and $L_2 = -22.918 - 2.3562\dot{s}^2$. The only pair that can collapse is $L_1 = U_2$: $-6.366 = 22.918 - 2.3562\dot{s}^2$, so $\dot{s}^2 = 12.429$ and

$$v(s) = 3.5254 \quad \text{for every } s$$

— a joint-1 speed of 5.54 rad/s. Faster than this, the elbow motor cannot supply the centrifugal torque $h\dot{q}_1^2$ while the shoulder brakes at full torque. Lower $\tau_2^{\max}$ to 10 N·m and v drops to 3.2875; the library pins both.

18.3.4 Time optimality is bang-bang

STATEMENT

$t_f = \int_0^1 ds/\dot{s}$, so the time-optimal scaling maximizes $\dot{s}$ pointwise subject to the cones; its phase-plane curve follows a cone boundary ($\ddot{s} = U$ or $\ddot{s} = L$) except possibly at zero-inertia points, and at least one actuator is saturated

at all times.

DERIVATION **Why the optimum rides the cone boundary**

$\ddot{s} \in \{L, U\}$ except on singular arcs

Step 1 — higher is faster. For two admissible curves $\dot{s}_a(s) \geq \dot{s}_b(s)$ on $[0, 1]$, $\int ds/\dot{s}_a \leq \int ds/\dot{s}_b$. Time is monotone in the curve's height, so the optimum is the pointwise-highest admissible curve.

Step 2 — a tangent strictly inside the cone is wasteful. If at some state the curve's slope $d\dot{s}/ds = \ddot{s}/\dot{s}$ lies strictly between $L/\dot{s}$ and $U/\dot{s}$, one can raise the curve locally — accelerate harder before, brake harder after — without leaving admissibility, and the new curve is higher. So on an optimal curve the tangent is on the boundary almost everywhere.

Step 3 — the reachable set is bounded by two integral curves. From any state, the set of all curves with tangents inside the cones is bounded above by the U-integral curve and below by the L-integral curve (Choset's figure 11.2). The optimum is therefore a concatenation of pieces of U- and L-curves.

Step 4 — where the switches are. The pieces meet where a U-curve crosses an L-curve, or where the limit curve forces a change: a U-curve that would penetrate the amber must have been abandoned earlier for an L-curve that grazes the limit tangentially. Finding those points is the algorithm.

The Pontryagin reading. Chapter 19 derives the same result from the minimum principle: for a minimum-time problem the Hamiltonian is affine in the control, $\partial H/\partial u$ contains no u, and H is minimized at a bound of the control set. Choset's §11.3.1 makes the connection explicit for the double integrator, where the switch is at $t_s = \sqrt{d/u_{\max}}$ — the same number the lattice reproduces below.

The micro-example never touches its limit curve. With constant $U_1 = 6.366$ and $L_1 = -6.366$ the A_0 curve is $x = \dot{s}^2 = 2 \cdot 6.366\, s$ and F is $x = 2 \cdot 6.366\,(1 - s)$; they cross at $s = \frac{1}{2}$ with $\dot{s} = \sqrt{6.366} = 2.5231 < 3.5254$. One switch, at

$$t_s = \sqrt{2 \cdot 0.5/6.366} = 0.3963 \text{ s}, \qquad t_f = 2t_s = 0.7927 \text{ s}.$$

Joint 2 is honest at the peak: during acceleration $\tau_2 = a_2 U_1 + b_2 \dot{s}^2 = 0.5236 \cdot 6.366 + 1.2337 \cdot 6.366 = 11.19 < 12$ N·m. That is Choset's Example 11.2.1 situation — the limit curve never reached, one switch — on this book's arm. Turn gravity on ($a_g = 9.81$) and at $s = \frac{1}{2}$ the gravity vector is $c = (10.405, -3.468)$, giving $U_1 = 3.054$, $L_1 = -9.678$ and $v(\frac{1}{2}) = 4.080$; at $s = 0$ the shoulder's gravity torque is $c_1 = 19.62$ N·m against a 20 N·m limit, so $U_1(0) = 0.1210$ — the arm can barely

lift, and the whole motion takes 2.3276 s with its single switch pushed out to $s = 0.7342$.

18.3.5 Zero-inertia points and singular arcs

STATEMENT (CHOSET §11.2.1)

Where $a_i(s) = 0$, row i becomes the direct velocity bound $u_i^{\min} \le b_i \dot{s}^2 + c_i \le u_i^{\max}$, giving $\dot{s}_{zip}^{\max}(s)$; the true limit is $\dot{s}^{\max} = \min(v, \dot{s}_{zip}^{\max})$. At a singular point the feasible accelerations are $\ddot{s}_{\max} = \min(\ddot{s}^{+}_{tangent}, U)$ and $\ddot{s}_{\min} = \max(\ddot{s}^{-}_{tangent}, L)$ with $\ddot{s}_{tangent} = \dot{s}^{\max}\, d\dot{s}^{\max}/ds$.

DERIVATION **Sliding along the limit curve**

$\ddot{s}_{\max} = \min(\ddot{s}^{+}_{tangent}, U), \quad \ddot{s}_{\min} = \max(\ddot{s}^{-}_{tangent}, L)$

Step 1 — the row loses its $\ddot{s}$. With $a_i = 0$, row i of eq. 11.7 reads $u_i^{\min} \le b_i \dot{s}^2 + c_i \le u_i^{\max}$. Nothing the actuator does changes $\ddot{s}$; the constraint is on $\dot{s}$ alone. Solving for $\dot{s}$ gives $\dot{s}_{zip}^{\max}$ (eq. 11.11). If $b_i = 0$ too, the row constrains nothing — provided the robot can hold the configuration, which is Choset's Problem 11.6 and the standing assumption.

Step 2 — a critical point is not a collapsed cone. On $v(s)$ the cone is a single vector, $L = U$. At a point of $\dot{s}^{\max}$ set by a zero-inertia bound, $L < U$: the cone is open, but it may point out of the admissible region.

Step 3 — clip to the tangent. Following U forward from a singular point penetrates the limit curve; the fastest admissible acceleration is the one that keeps the state *on* the curve, which is the curve's tangent in the $(s, \dot{s})$ plane: $d\dot{s}/dt = \ddot{s} = \dot{s}\, d\dot{s}^{\max}/ds$. Hence $\ddot{s}_{\max} = \min(\ddot{s}^{+}_{tangent}, U)$, with the right-hand derivative where the curve has a corner, and symmetrically $\ddot{s}_{\min} = \max(\ddot{s}^{-}_{tangent}, L)$ for the backward integration.

Step 4 — singular arcs. Critical points lie where $M(q)$ loses rank along the path's tangent, a lower-dimensional set; a path crossing it transversally has isolated critical points, a path moving along it has a *critical arc*. Integrating $\ddot{s}_{\max}$ and $\ddot{s}_{\min}$ instead of U and L lets the algorithm slide along such an arc with an acceleration strictly between L and U, rather than chattering between them — and on a slide neither actuator is saturated.

In $x = \dot{s}^2$ the rule is a one-liner. The library integrates in x, where a stage from s_k to s_{k+1} with constant $\ddot{s}$ moves x by $2\ddot{s}\,\Delta$. When the free step would land above $x_{\max}(s_{k+1})$, the stage acceleration that lands exactly on the curve is $u^* = (x_{\max} - x_k)/(2\Delta)$; if u^* is inside the cone the stage slides, otherwise it penetrates. At a critical point that is Choset's rule; at an ordinary point

of v the collapsed cone makes the test pass only when the integral curve is tangent to the limit — which is the tangent-point condition of step 4 of the algorithm, found for free.

Choset's RP arm, honestly. Example 11.2.1 runs the RP arm of Chapter 17's Example 10.1.2 ($m_1 = 5$, $I_1 = 0.1$, $r_1 = 0.2$, $m_2 = 3$, $I_2 = 0.05$, $a_g = 9.8$) along the straight Cartesian line $x(s) = (2s - 1, 1)$ with limits ±20 N·m and ±40 N. The path has $a_2(\frac{1}{2}) = 0$ with $b_2 \equiv 0$ — a zero-inertia point that imposes no velocity bound, the case of Problem 11.5. Choset reports a minimum time of approximately 0.888 s with one switch. With the printed parameters, however, the gravity torque on joint 1 at either end of the path is $a_g(m_1 r_1 + m_2\sqrt{2})\cos(\pi/4) = 36.33$ N·m against a 20 N·m limit, so $U(0,0) = -2.572$ and $L(1,0) = +2.572$: the arm can neither start forward from rest nor come to rest at the end, and both algorithms in this chapter report that the rest-to-rest problem has no solution — Choset's Problems 11.3 and 11.4 in the flesh. The 0.888 s figure is not reproducible from the printed parameters. The library's cross-check therefore asserts the diagnosis, and reports the gravity-free execution: one switch at $s = \frac{1}{2}$ and $t_f = 1.1445$ s, with TOPP-RA agreeing to the grid.

18.3.6 GRID SEARCH is optimal for its discretization

STATEMENT (CHOSET ALGORITHM 21, THEOREM 11.3.2)

For $\ddot{q}_i = a_i$ with $a_i \in \{-a_{\max}, 0, a_{\max}\}$ held for time h, every reachable state lies on a lattice with spacing $a_{\max}h$ in $\dot{q}$ and $\frac{1}{2}a_{\max}h^2$ in q; breadth-first expansion by level returns the fewest-steps safe trajectory. For a point robot in $n \in \{2, 3\}$ dimensions among polyhedral obstacles, if a $\delta_v(c_0, c_1)$-safe trajectory of duration T_{opt} exists, then with $h \le v_{\max}/a_{\max}$ and $h \le c_0\epsilon/(2a_{\max}c_1(1-\epsilon) + 5v_{\max})$ the search returns a $(1-\epsilon)\delta_v$-safe trajectory of duration at most T_{opt} from an $O(\epsilon)$-perturbed start to an $O(\epsilon)$ goal neighborhood, in time $O((1/\epsilon)^{3n})$.

DERIVATION **Why the tree is a grid, and what the theorem buys**

$O((1/\epsilon)^{3n})$, time $\le T_{opt}$, $(1-\epsilon)\delta_v$-safe

Step 1 — closure under the control set. Integrating $\ddot{q}_i = c\,a_{\max}$, $c \in \{-1, 0, 1\}$, for time h from $(q_i, \dot{q}_i)$ gives $\dot{q}_i + c\,a_{\max}h$ and $q_i + \dot{q}_i h + \frac{1}{2}c\,a_{\max}h^2$. Write $\dot{q}_i = m\,a_{\max}h$ and $q_i = p \cdot \frac{1}{2}a_{\max}h^2$ with integers m, p; then the new indices are $m + c$ and $p + 2m + c$ — integers again. Every node of Choset's

figure 11.8 lies on the grid of figure 11.9, and the tree is really a graph on that grid.

Step 2 — level is time. Each edge takes exactly h, so breadth-first expansion is uniform-cost search with unit edges, and the first level at which a node enters the goal region is the minimum number of steps: time-optimal for the chosen discretization of time and controls. A* with an admissible heuristic pops the same level with fewer expansions. The per-axis continuous bang-bang time $\max_i T_i$ is admissible because it ignores the other axes, the obstacles, the velocity limit and the discretization — every one of which can only make the lattice slower.

Step 3 — safety absorbs the discretization. Any continuous δ_v-safe trajectory can be tracked by lattice moves within an error that shrinks with h; the margin $c_0 + c_1\|\dot{q}\|$ is spent on that error, leaving $(1-\epsilon)\delta_v$. The two bounds on h are what make the bookkeeping close — the first keeps $v_{\max}$ an integer multiple of $a_{\max}h$, the second ties the tracking error to the margin.

Step 4 — the grid is finite, so the time is bounded. The configuration-space diameter, the velocity bound and h fix the number of lattice points, which is $O((1/\epsilon)^{3n})$ once h is expressed through ϵ. Polynomial in ϵ, exponential in n: nobody has run this beyond a few degrees of freedom, and Choset says so.

What the theorem does not say. The $(1-\epsilon)\delta_v$-safe trajectory the algorithm finds may look nothing like the true time-optimal δ_v-safe one; the only promise is about its duration. And the proof — Donald and Xavier's — is beyond this chapter as it is beyond Choset's.

The manipulator generalization (Choset pp. 396–399). For $M(q)\ddot{q} + C\dot{q} + g = u$ with box torque limits, the feasible accelerations at a state form a parallelepiped $\mathcal{A}(q, \dot{q})$ that changes during the step and collapses where M^{-1} loses rank. A bound on the largest eigenvalue of M keeps it from collapsing; a bound on $\dot{M}$ gives a conservative subset $\hat{\mathcal{A}} \subset \mathcal{A}$ feasible over the whole step; the controls are the $\ddot{q}$ grid points inside $\hat{\mathcal{A}}$. The completeness result survives with $(1+\epsilon)T_{opt}$ in place of T_{opt}; the running time stays exponential in n.

INTERACTIVE · w18.2

Lattice Hopper

Time-optimality lives in the state space (q, $\dot{q}$): a grid in q alone cannot represent momentum, and the lattice is optimal only for its own discretization.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch18-trajectory-planning#w18.2

The hand trace Choset's figure 11.9 invites: $a_{\max} = 1$, $h = \frac{1}{2}$, from rest at 0 to rest at 1. The lattice has $\Delta\dot{q} = 0.5$ and $\Delta q = 0.125$; the goal is $p = 8$. The search returns the four-level sequence $(+1, +1, -1, -1)$ — velocities $0.5, 1.0, 0.5, 0$ and positions $0.125, 0.5, 0.875, 1.0$ — in $t_f = 2$ s, which is the continuous $2\sqrt{d/a_{\max}}$ exactly, and the 81 control sequences of length four collapse to 49 distinct lattice states. For the widget's default $d = 2$ the continuous bound is $2\sqrt{2} = 2.828$ s; the lattice returns 3 s at $h \in \{1, \frac{1}{2}, \frac{1}{4}\}$ and 2.875 s at $h = \frac{1}{8}$ (23 levels). Finer is never slower, and never faster than the bound.

18.3.7 TOPP-RA: reachability on squared speed

STATEMENT (PHAM & PHAM 2018)

With $x = \dot{s}^2$ and $u = \ddot{s}$, $dx/ds = 2u$, and each actuator limit $u_i^{\min} \le a_i u + b_i x + c_i \le u_i^{\max}$ is *linear* in (x, u) at fixed s. On a grid $s_0 < \cdots < s_N$ with $x_{k+1} = x_k + 2u_k\Delta_k$, the set of x_k from which some feasible u_k lands x_{k+1} in a target interval is itself an interval, computable by two two-variable linear programs. A backward pass yields the controllable intervals K_k; a greedy forward pass picking the largest feasible $x_{k+1} \in K_{k+1}$ is time-optimal for the discretization.

DERIVATION **Interval propagation instead of switch hunting**

$K_k = \{x \ge 0 : \exists u \text{ feasible at } k,\ x + 2u\Delta_k \in K_{k+1}\}; \quad x_{k+1} = \max$

Step 1 — the substitution. $\frac{d}{ds}\dot{s}^2 = 2\dot{s}\,\frac{d\dot{s}}{ds} = 2\ddot{s}$. In x the kinematics are linear and a constant-$\ddot{s}$ stage is a straight line; the micro-example's switch at $s = \frac{1}{2}$ lands on the grid exactly, and $\dot{s} = 0$ is no longer a singularity of the integration.

Step 2 — convexity of the stage constraint set. At fixed s_k the $2n$ inequalities in (x_k, u_k) are half-planes, and $x_{k+1} = x_k + 2u_k\Delta$ is affine. The feasible set is a convex polygon; its projection onto the x_k axis is an interval.

Step 3 — propagate backward. $K_N = [x_f, x_f]$. Given $K_{k+1} = [\ell, h]$, add the two half-planes $\ell \le x_k + 2u_k\Delta \le h$ and take the minimum and maximum of x_k over the polygon — two two-variable LPs, solved exactly by vertex enumeration since there are at most $2n + 6$ half-planes. The result is K_k, the set of states at s_k from which the goal is still reachable: the complement of Choset's "doomed" states, computed without ever drawing a curve.

Step 4 — greedy forward is optimal. A larger x_k never shrinks the reachable set downstream (controllability is monotone in x under these constraints), so choosing the largest admissible $x_{k+1} \in K_{k+1}$ at each stage dominates every other choice pointwise, and pointwise-highest is fastest

by the first derivation.
What it sidesteps. No tangent points, no root finding, no footnote-3 bisection, no special case for zero-inertia rows — a row with $a_i = 0$ is just a half-plane with no u in it. The price is a grid: the answer is optimal for the discretization, and the stage constraints are enforced at s_k only. On the chapter's seeded gravity-loaded splines at $N = 800$ the two constructions agree on t_f to 0.38% at worst, and they agree on which problems have no solution at all.

18.4 The algorithm

Choset's five steps, with the §11.2.1 modification and the footnote-3 bisection that this chapter's implementation uses in step 4. The construction is drawn live in w18.1: F dashed, each A_i solid, the hit points in amber.

ALGORITHM `time_scale(a, b, c, u_min, u_max, ṡ0, ṡ_f)` COST `O(N)` stages per curve; `O(N log(1/tol))` per switch for the bisection

in path-constrained dynamics on a grid s_k = k/N, actuator limits, boundary speeds
out switch list $\mathcal{S}$, the profile (s_k, ṡ_k), t_f — or StallsAtZero / NoSolution / LimitIsland

1: $\mathcal{S} \leftarrow \{\}$, $i \leftarrow 0$, $(s_i, \dot{s}_i) \leftarrow (0, \dot{s}_0)$. Tabulate $\dot{s}^{\max}(s_k) = \min(v, \dot{s}^{\max}_{zip}, \dot{s}^{\max}_{rate})$ and fail with NoSolution if any $\dot{s}^{\max}(s_k) \le 0$ (the robot cannot hold that configuration).
2: **Backward curve** F**.** Integrate $\ddot{s} = L$ backward from $(1, \dot{s}_f)$ until the limit curve is penetrated (record the hit), $s = 0$, or $\dot{s} = 0$ at $s < 1$ — in which case return StallsAtZero (Problem 11.3). A stage that would penetrate but can land on the curve with $\ddot{s}_{\min} = \max(\ddot{s}^{-}_{tangent}, L)$ slides instead.
3: **Forward curve** A_i**.** Integrate $\ddot{s} = U$ forward from $(s_i, \dot{s}_i)$, sliding with $\ddot{s}_{\max} = \min(\ddot{s}^{+}_{tangent}, U)$ where allowed, until it crosses F — append the crossing s to $\mathcal{S}$ and **return** — or penetrates the limit curve at $(s_{lim}, \dot{s}_{lim})$. If it reaches $s = 1$ below $\dot{s}_f$ or reaches $\dot{s} = 0$, return NoSolution (Problem 11.4).
4: **Tangent point (footnote 3).** On the line $s = s_{lim}$ bisect on $\dot{s}' \in [0, \dot{s}_{lim}]$ for the highest L-curve from $(s_{lim}, \dot{s}')$ that does not penetrate the limit curve; its point of closest approach is $(s_{tan}, \dot{s}_{tan})$ — a tangent point, or a critical point. Integrate $\ddot{s} = L$ backward from $(s_{lim}, \dot{s}')$ until it rises above A_i; the crossing s_{i+1} is a max→min switch — append it. If it reaches $\dot{s} = 0$ or passes below the start, return NoSolution: the start

state is doomed (figure 11.2).

5: $(s_{i+2}, \dot{s}_{i+2}) \leftarrow (s_{tan}, \dot{s}_{tan})$ is a min→max switch — append it, set $i \leftarrow i+2$, go to 3. A min→max switch immediately followed by a max→min at the same node is a slide, not a switch, and is removed from $\mathcal{S}$.

Two implementation choices matter. The curves are integrated in $x = \dot{s}^2$ by RK4 on the uniform grid, with the stage states clamped to the admissible region at the half-grid nodes — next to a zero-inertia point a stage that lands a hair above the limit would otherwise read a bound n_i/a_i with $a_i \approx 0$ and drive x negative. And the bisection in step 4 replaces the slope test $dv/ds = U/v$ of Choset's main text: it never selects a tangent point whose own deceleration curve is doomed, which the slope test can, and it costs about 48 extra L-curve integrations per switch.

ALGORITHM `toppra(a, b, c, u_min, u_max, grid)` COST `O(N) two-variable LPs backward, O(N) forward`

in path-constrained dynamics tabulated at s_0 < ... < s_N, limits, $\dot{s}_0$, $\dot{s}$_f

out controllable sets K_k, the greedy profile x_k, t_f — or the first k at which K_k is empty

1: $K_N \leftarrow [\dot{s}_f^2, \dot{s}_f^2]$.

2: for $k = N-1$ down to 0: half-planes $u_i^{\min} \le a_i u + b_i x + c_i \le u_i^{\max}$ at s_k, plus $x \ge 0$ and $K_{k+1}^{lo} \le x + 2u\Delta_k \le K_{k+1}^{hi}$; $K_k \leftarrow [\min x, \max x]$ over the polygon's vertices. If empty, report no solution from s_k.

3: if $\dot{s}_0^2 \notin K_0$: no solution (the start is doomed).

4: for $k = 0$ to $N-1$: $u_k \leftarrow$ the largest u feasible at (s_k, x_k) with $x_k + 2u\Delta_k \in K_{k+1}$; $x_{k+1} \leftarrow x_k + 2u_k\Delta_k$.

5: $t_f \leftarrow \sum_k 2\Delta_k/(\sqrt{x_k} + \sqrt{x_{k+1}})$.

ALGORITHM `GRID SEARCH((q*_start, q̇*_start), G, 𝒜, h) -- Choset Algorithm 21` COST `O((1/ε)^{3n}) -- Theorem 11.3.2`

in lattice start state, goal region, control set {−a_max, 0, a_max}^{n}, timestep

out a piecewise-constant-acceleration trajectory to G, or FAILURE

1: place the start at the root of T (level 0); $level \leftarrow 0$, $solved \leftarrow$ false

2: while not $solved$: if level $level$ is empty return FAILURE

3: for each node in level $level$, for each control in $\mathcal{A}$: integrate the control for time h, getting $newnode$

4: if $newnode$ is new, and the arc neither passes within $c_0 + c_1\|\dot{q}\|$ of an obstacle nor exceeds $v_{\max}$: add it as a child
5: if the arc enters G: $solved \leftarrow$ true, store $newnode$
6: $level \leftarrow level + 1$
7: return the stored trajectory that reaches G first

ALGORITHM `lattice_search(start, goal, a_max, v_max, h, safe) -- ours`
COST `same worst case; in practice 10-700× fewer expansions than breadth-first on the chapter's instances`
in the same, as integer lattice coordinates (p, m) with q = p·$\frac{1}{2}$a_max h^2, $\dot{q}$ = m·a_max h
out the same trajectory, with a count of expansions

1: graph: node (p, m); neighbors $(p + 2m + c,\ m + c)$ for $c \in \{-1, 0, 1\}^n$, kept if the quadratic arc is safe and $|(m + c)\, a_{\max} h| \le v_{\max}$; every node inside G shares one key
2: heuristic $h(p, m) \leftarrow \max_i T_i$, the per-axis continuous bang-bang time to the goal — admissible and consistent
3: Chapter 6's `BestFirst` with $f = g + h$; breadth-first is the same engine with $f =$ link length
4: return the back-pointer path; controls are Δm between consecutive states

18.5 Implementation in Rust

The `trajectory` crate is introduced here and imported by Chapters 19, 21 and 23. Its surface is a `Path` trait, the path-constrained dynamics, the limits, two time scalers, the lattice, and one type — `Trajectory` — whose whole job is to be a different type from `Path`.

RUST `crates/trajectory/src/path.rs`

```rust
use nalgebra::SVector;

/// A twice-differentiable curve q : [0, 1] -> Q. `ddq` must be *continuous*: the
/// velocity-product term b(s) contains M q'', and a C1 spline would make it jump
/// at every knot -- the phase plane would grow a fence of fake zero-inertia points.
pub trait Path<const N: usize> {
    fn q(&self, s: f64) -> SVector<f64, N>;
    fn dq(&self, s: f64) -> SVector<f64, N>;
    fn ddq(&self, s: f64) -> SVector<f64, N>;
}

/// A straight line in the chart; ddq ? 0. The micro-example's path.
pub struct Segment<const N: usize> { pub a: SVector<f64, N>, pub b: SVector<f64, N> }
```

```rust

impl<const N: usize> Path<N> for Segment<N> {
    fn q(&self, s: f64) -> SVector<f64, N> { self.a + s * (self.b - self.a) }
    fn dq(&self, _s: f64) -> SVector<f64, N> { self.b - self.a }
    fn ddq(&self, _s: f64) -> SVector<f64, N> { SVector::zeros() }
}

/// Natural cubic spline through m >= 2 waypoints at uniform knots s_k = k/(m - 1).
/// Natural ends (q'' = 0) are the right choice for a rest-to-rest motion: b(s)
/// then vanishes at both ends and the plane's edges are governed by a and c alone.
pub struct CubicSpline<const N: usize> {
    knots: Vec<f64>,
    pts: Vec<SVector<f64, N>>,
    /// Second derivatives at the knots, from one tridiagonal solve per coordinate.
    m2: Vec<SVector<f64, N>>,
}

impl<const N: usize> CubicSpline<N> {
    /// Waypoints on T^n are unwrapped first: consecutive angles differ by the
    /// shortest arc, so a path through the seam is fitted in the chart lift
    /// rather than interpolated across a 2pi jump (Chapter 5).
    pub fn on_torus(waypoints: &[SVector<f64, N>]) -> Self {
        let mut pts = vec![waypoints[0]];
        for w in &waypoints[1..] {
            let prev = *pts.last().unwrap();
            pts.push(prev + (w - prev).map(|d| (d + PI).rem_euclid(2.0 * PI) - PI));
        }
        Self::fit(pts)
    }

    fn fit(pts: Vec<SVector<f64, N>>) -> Self {
        let m = pts.len();
        let knots: Vec<f64> = (0..m).map(|k| k as f64 / (m - 1) as f64).collect();
        // Interior rows: h M_{k-1} + 4h M_k + h M_{k+1} = 6 (d_k - d_{k-1}); M_0 = M_{m
   -1} = 0.
        let m2 = natural_second_derivatives(&knots, &pts);
        Self { knots, pts, m2 }
    }
}
```

RUST **crates/trajectory/src/constrained.rs + limits.rs**

```rust
use dynamics::Dynamics;

/// a(s) s + b(s) ?2 + c(s) = u -- Chapter 17's model restricted to a path (C eq. 11.6).
pub struct PathDynamics<'a, D: Dynamics<N>, P: Path<N>, const N: usize> {
    pub dynamics: &'a D,
    pub path: &'a P,
}

impl<D: Dynamics<N>, P: Path<N>, const N: usize> PathDynamics<'_, D, P, N> {
    /// b uses the velocity-product *vector* at q_dot = q'(s): q'^T Gamma q' = C(q, q') q
   ' exactly.
    pub fn abc(&self, s: f64) -> (SVector<f64, N>, SVector<f64, N>, SVector<f64, N>) {
        let (q, dq, ddq) = (self.path.q(s), self.path.dq(s), self.path.ddq(s));
        let m = self.dynamics.mass(&q);
        (m * dq, m * ddq + self.dynamics.velocity_product(&q, &dq), self.dynamics.gravity
   (&q))
```

```rust
    }

    /// The torque that executes the path at phase state (s, ?) with path acceleration s.
    pub fn torque(&self, s: f64, sdot: f64, sddot: f64) -> SVector<f64, N> {
        let (a, b, c) = self.abc(s);
        a * sddot + b * sdot * sdot + c
    }
}

pub struct TorqueLimits<const N: usize> { pub min: SVector<f64, N>, pub max: SVector<f64, N> }

/// Below this |a_i| a row is zero-inertia: it bounds ?, not s.
pub const ZERO_INERTIA_EPS: f64 = 1e-9;

/// (L, U) at a phase-plane state -- eqs. 11.8-11.9. Dividing by a negative a_i
/// flips the inequality; that swap is the whole content of 11.8.
pub fn accel_bounds<const N: usize>(abc: &Abc<N>, lim: &TorqueLimits<N>, sdot: f64) -> (f64, f64) {
    let x = sdot * sdot;
    let (mut lo, mut hi) = (f64::NEG_INFINITY, f64::INFINITY);
    for i in 0..N {
        let ai = abc.0[i];
        if ai.abs() < ZERO_INERTIA_EPS { continue; }
        let alpha = (lim.max[i] - abc.1[i] * x - abc.2[i]) / ai;
        let beta  = (lim.min[i] - abc.1[i] * x - abc.2[i]) / ai;
        let (li, ui) = if ai > 0.0 { (beta, alpha) } else { (alpha, beta) };
        lo = lo.max(li);
        hi = hi.min(ui);
    }
    (lo, hi)
}

/// v(s) from the pairwise crossings L_i = U_j (eq. 11.10). Each bound is affine
/// in x = ?2, p + m x, so the crossing is x = (p_U - p_L)/(m_L - m_U); keep the
/// smallest positive root over all pairs. +inf when no pair can collapse the cone.
pub fn velocity_limit<const N: usize>(abc: &Abc<N>, lim: &TorqueLimits<N>) -> f64 {
    let rows: Vec<Row> = (0..N).filter(|&i| abc.0[i].abs() >= ZERO_INERTIA_EPS).map(|i| Row::new(abc, lim, i)).collect();
    let mut best = f64::INFINITY;
    for li in &rows { for uj in &rows {
        let den = li.m_l - uj.m_u;
        if den.abs() < 1e-14 { continue; }
        let x = (uj.p_u - li.p_l) / den;
        if x > 0.0 && x < best { best = x; }
    } }
    best.sqrt()
}

/// ?_max(s) = min(v, ?_zip, ?_rate): the true limit curve of sec.11.2.1, with
/// joint-rate limits |q_dot_i| <= v_i folded in as rows ? <= v_i / |q'_i| (Exercise 5).
pub fn sdot_max<const N: usize>(abc: &Abc<N>, dq: &SVector<f64, N>, lim: &TorqueLimits<N>,
    rate: Option<&[f64; N]>) -> LimitPoint {
    let v = velocity_limit(abc, lim);
    let zip = zero_inertia_limit(abc, lim);        // u_min <= b_i ?2 + c_i <= u_max on rows with a_i = 0
    let rate = joint_rate_limit(dq, rate);
    LimitPoint::min_of(v, zip, rate)               // remembers which bound is active: cone, zip or rate
}
```

RUST **crates/trajectory/src/time_scale.rs**

```rust
pub enum TimeScaleError {
    /// F reached ? = 0 before s = 0: full braking cannot bring the arm to rest (Problem 11.3).
    StallsAtZero { s: f64 },
    /// A_i reached ? = 0 or s = 1, or the start is doomed (Problem 11.4, figure 11.2).
    NoSolution { s: f64, detail: String },
    /// More switches than the admissible region can honestly have (footnote 2).
    LimitIsland { s: f64 },
}

pub struct TimeScaling {
    pub switches: Vec<f64>,          // Choset's list ?
    pub sdot: Vec<f64>,              // the profile on s_k = k/N
    pub stage: Vec<f64>,             // s on each stage: x(s) is piecewise linear, so a replay applies exactly this
    pub mode: Vec<PhaseMode>,        // Max | Min | Slide -- Slide is a singular arc
    pub t: Vec<f64>,                 // t[N] = t_f
}

/// One grid stage in x = ?2. When the free step leaves the admissible region,
/// try the stage acceleration that lands exactly on the limit curve; it is
/// feasible iff it lies in the cone -- Choset's s_max = min(s?_tangent, U) at a
/// critical point, and the tangency test for free on an ordinary point of v(s).
fn step_clipped(t: &Tables, k: usize, x: f64, mode: Mode, dir: i32) -> StepOut {
    let raw = rk4_in_x(t, k, x, mode, dir);              // stage states clamped to x <= x_max at the half grid
    let kn = (k as i32 + dir) as usize;
    if !above(raw, t.xmax[kn]) { return StepOut::free(raw); }
    let u_star = dir as f64 * (t.xmax[kn] - x) / (2.0 * t.ds);
    let (l0, u0) = accel_bounds(&t.abc[k], &t.lim, x.sqrt());
    let (l1, u1) = accel_bounds(&t.abc[kn], &t.lim, t.xmax[kn].sqrt());
    if u_star >= l0.min(l1) && u_star <= u0.max(u1) { StepOut::slide(t.xmax[kn]) } else { StepOut::penetrated(raw) }
}

pub fn time_scale<const N: usize>(pd: &PathDynamics<'_, impl Dynamics<N>, impl Path<N>, N>, lim: &TorqueLimits<N>,
                                  sdot0: f64, sdot_f: f64, grid: usize) -> Result<TimeScaling, TimeScaleError> {
    let t = Tables::tabulate(pd, lim, grid);
    // Step 2: F backward from (1, ?_f).
    let f = integrate_backward(&t, grid, sdot_f * sdot_f, Mode::Min)?;    // StallsAtZero if x <= 0 before s = 0
    let (mut ki, mut xi, mut switches, mut profile) = (0, sdot0 * sdot0, vec![], Profile::new(grid));
    loop {
        // Step 3: A_i forward with U; crossing F is tested on the free curve before penetration.
        match integrate_forward(&t, ki, xi, Mode::Max, &f)? {
            Forward::CrossedF { node, frac, a } => { profile.write(ki, node, &a, Mode::Max); profile.write_tail(&f);
                                                     switches.push_crossing(node, frac, grid); break; }
            Forward::Penetrated { k_lim, a, .. } => {
                // Step 4, footnote 3: bisect on ?' at s_lim for the highest admissible L-curve.
```

```
                let (x_prime, lc) = bisect_highest_admissible_l_curve(&t, k_lim, a[k_lim], &f);
                let k_tan = lc.closest_approach_to_limit();                          // tangent or critical point
                let meet = meet_backward(&t, &a, ki, k_lim, x_prime)?;               // where L rises above A_i
                profile.write(ki, meet.node, &a, Mode::Max);
                profile.write(meet.node + 1, k_tan, &lc, Mode::Min);
                switches.push_max_to_min(meet); switches.push_min_to_max(k_tan);  // Step 5
                (ki, xi) = (k_tan, lc[k_tan]);
            }
        }
    }
    Ok(profile.finish(switches))     // t_f = Sigma 2Delta/(?_k + ?_{k+1}); slides reported as Mode::Slide
}
```

RUST `crates/trajectory/src/toppra.rs`

```
/// TOPP-RA (Pham & Pham 2018): controllable intervals in x = ?2 backward, greedy forward.

pub fn toppra<const N: usize>(pd: &PathDynamics<'_, impl Dynamics<N>, impl Path<N>, N>, lim: &TorqueLimits<N>,
                              sdot0: f64, sdot_f: f64, grid: usize) -> Result<TimeScaling, TimeScaleError> {
    let ds = 1.0 / grid as f64;
    let abc: Vec<Abc<N>> = (0..=grid).map(|k| pd.abc(k as f64 * ds)).collect();
    let mut k_set = vec![(sdot_f * sdot_f, sdot_f * sdot_f); grid + 1];
    for k in (0..grid).rev() {
        // Half-planes in (x, u): the 2N actuator rows at s_k, x >= 0, and l <= x + 2uDelta <= h for K_{k+1}.
        let mut hp = actuator_half_planes(&abc[k], lim);     // a row with a_i = 0 is a half-plane with no u in it
        hp.push(HalfPlane { p: 1.0, r: 2.0 * ds, q: k_set[k + 1].1 });
        hp.push(HalfPlane { p: -1.0, r: -2.0 * ds, q: -k_set[k + 1].0 });
        k_set[k] = extreme_x(&hp).ok_or(TimeScaleError::NoSolution { s: k as f64 * ds, detail: "controllable set empty".into() })?;
    }
    if sdot0 * sdot0 < k_set[0].0 || sdot0 * sdot0 > k_set[0].1 { return Err(TimeScaleError::NoSolution { s: 0.0, detail: "?_0 not controllable".into() }); }
    let mut x = vec![sdot0 * sdot0; grid + 1];
    for k in 0..grid {
        let (lo, hi) = feasible_u(&abc[k], lim, x[k]);                       // the cone at (s_k, x_k)
        let u = hi.min((k_set[k + 1].1 - x[k]) / (2.0 * ds)).max(lo.max((k_set[k + 1].0 - x[k]) / (2.0 * ds)));
        x[k + 1] = (x[k] + 2.0 * u * ds).max(0.0);                           // greedy: largest x_{k+1} in K_{k+1}
    }
    Ok(TimeScaling::from_x(x, ds))
}
```

RUST `crates/trajectory/src/lattice.rs`

```rust
use search::{astar, Graph};

/// Integer lattice coordinates: q_i = p_i . ? a_max h2, q_dot_i = m_i . a_max h.
#[derive(Clone, PartialEq, Eq, Hash)]
pub struct LatticeState<const N: usize> { pub p: [i64; N], pub m: [i64; N] }

pub struct LatticeGraph<'a, const N: usize> {
    pub a_max: f64, pub h: f64, pub v_max: f64,
    pub c0: f64, pub c1: f64,
    /// Clearance at a configuration; the lattice samples it along every arc.
    pub clearance: &'a dyn Fn(&SVector<f64, N>) -> f64,
    pub goal: LatticeState<N>,
}

impl<const N: usize> Graph<LatticeState<N>> for LatticeGraph<'_, N> {
    /// Neighbors are (p + 2m + c, m + c) for c in {-1, 0, 1}?, kept when the
    /// quadratic arc is delta_v(c_0, c_1)-safe and the end velocity is within v_max.
    fn neighbors(&self, s: &LatticeState<N>) -> Vec<(LatticeState<N>, f64)> {
        controls::<N>().filter_map(|c| {
            let next = LatticeState { p: array_fn(|i| s.p[i] + 2 * s.m[i] + c[i]), m:
   array_fn(|i| s.m[i] + c[i]) };
            self.arc_safe(s, &c).then_some((next, self.h))
        }).collect()
    }
}

/// Minimum time of a 1-D double integrator from (q_0, v_0) to (q_1, v_1): one switch,
/// via the peak velocity v_p with (v_p2 - v_02)/2a + (v_p2 - v_12)/2a = Deltaq. This is
/// also Chapter 19's min-time double integrator read off the minimum principle.
pub fn bang_bang_time_1d(q0: f64, v0: f64, q1: f64, v1: f64, a: f64) -> f64 { /* two
   candidate families */ }

/// Algorithm 21 with A* (h = max_i per-axis bang-bang time, admissible) or Choset's
   literal breadth-first.
pub fn lattice_search<const N: usize>(start: LatticeState<N>, goal: LatticeState<N>, g: &
   LatticeGraph<'_, N>) -> Option<Vec<[i64; N]>> {
    let h = |n: &LatticeState<N>| (0..N).map(|i| bang_bang_time_1d(q_of(n.p[i], g), v_of(
   n.m[i], g), q_of(goal.p[i], g), v_of(goal.m[i], g), g.a_max))
                                        .fold(0.0, f64::max);
    let path = astar(g, start, goal, h)?;
    Some(path.windows(2).map(|w| array_fn(|i| w[1].m[i] - w[0].m[i])).collect())   //
   controls are Deltam
}

/// A time-parameterized curve -- Choset sec.1.3's distinction made a type. Trackers
/// accept a `Trajectory`, never a `Path`: handing a geometric curve to a
/// controller is a compile error, not a runtime surprise.
pub struct Trajectory<M: Manifold> { pub samples: Vec<(f64, M)> }
```

18.5.1 The worked example, printed

`cargo run --example reach_sweep -p trajectory` builds the micro-example and prints

```
a(0.5) = [3.1416, 0.5236]    b(0.5) = [0.0000, 1.2337]    c(0.5) = [0.0000, 0.0000]
v(s)   = 3.5254  (constant; the pair L_1 = U_2)
switches = [0.5000]    t_s = 0.3963 s    t_f = 0.7927 s    peak ? = 2.5231
```

```
tau_2 at the switch = 11.19 N.m  (< 12: honest)
TOPP-RA, 200-point grid: t_f = 0.7927 s
gravity on:  c(0.5) = [10.4051, -3.4684]   U_1(0.5) = 3.0542   L_1(0.5) = -9.6782   v
   (0.5) = 4.0799
            U_1(0) = 0.1210   t_f = 2.3276 s   switches = [0.7342]
tau_2 = 10 N.m: v = 3.2875   switches = [0.5071]   t_f = 0.7928 s
RP arm (Choset Ex. 11.2.1, printed parameters): |c_1(0)| = 36.33 > 20  ->  U(0,0) =
   -2.572, L(1,0) = +2.572
            time_scale: StallsAtZero   toppra: controllable set empty at s = 0.999
            a_g = 0:  one switch at 0.5000, t_f = 1.1445 s  (toppra 1.1445)
lattice 1-DOF, a = 1, h = 0.5, 0 -> 1:  controls [+1, +1, -1, -1]  t_f = 2.000 = 2sqrt(d/
   a)
            81 sequences -> 49 states at level 4;  expansions A* 5 vs breadth-first 43
lattice 2-DOF around a disc, h = 0.5:  6 levels;  expansions A* 17 / Alg. 21 4769 / heap
   BFS 12866
            c_1 = 0.5: 8 levels;  h = 0.25: 12 -> 14 levels (3.0 -> 3.5 s), A* 494
   expansions
```

`#[test] fn reproduces_micro_example()` asserts each line to 10^{-3} (the integrator's grid sets the tolerance) and $|t_f^{\text{TOPP-RA}} - t_f| < 5 \times 10^{-3}$ on a 200-point grid. `#[test] fn lattice_finds_bang_bang()` is the hand trace above. `#[test] fn replay_tracks_path()` closes the loop with Chapter 17: it time-scales the widgets' default spline ($t_f = 0.6176$ s, switches at 0.398, 0.571, 0.668), computes $u(t) = a\ddot{s} + b\dot{s}^2 + c$ from the profile, integrates `Reach2R` with Chapter 17's RK4 at $h = 0.25$ ms, and asserts that the integrated $q(t)$ stays within 10^{-3} rad of $q(s(t))$ — the measured worst error is 4.5×10^{-4} rad — with every sampled torque inside its limit to within the one-percent slop of the stage-constant $\ddot{s}$. The arm hangs below the shoulder in that test on purpose: an open-loop replay through the inverted configuration diverges exponentially whatever the planner did, which is Chapter 19's cue for feedback.

> **NOTE**
>
> The widgets on this page run the TypeScript port in `lib/trajectory/`, a line-for-line translation of the Rust above. Its thirteen self-checks reproduce every number in the printout and add the invariants the mathematics guarantees: the closed-form $v(s)$ against a brute-force scan, an actuator saturated at every node of every solved profile, the profile never above $\dot{s}^{\max}$ and touching it only at switches or slides, $x(s)$ continuous across every switch (the bug that check was written to catch jumped the profile by 1.2 in x), TOPP-RA within 0.38% of the switching construction on seeded gravity-loaded splines and agreeing with it on which problems are unsolvable, and t_f non-increasing as the motors get stronger.

18.6 Putting it together

The integration lab is the pipeline Chapter 23 ships: a Chapter 12 RRT path for Reach, shortcut by Chapter 13 into a polyline, smoothed into a `CubicSpline` on T^2,

time-scaled here, and replayed by the Chapter 17 integrator with the computed torques. On the Workbench with gravity the gauge grazes its rail at every instant and never crosses it — that is what bang-bang looks like from the motor's side.

Two honesty items the lab makes visible. First, the shortcutting step matters more than it looks: a polyline has corners, a spline through a polyline's vertices has large q'' near sharp ones, and large q'' is large $b(s)$, which lowers the velocity-limit curve exactly where the path turns. The time scaler is only as fast as the path is smooth. Second, decoupled planning is honest about time only *locally*: it finds the fastest clock for the path it was handed, and a slightly different path through the same homotopy class may be much faster. Choset's §11.2.2 sketches Shiller and Dubowsky's answer — enumerate grid paths, bound their times from below with a speed cap, time-scale the best, prune the rest, and polish locally with the time scaler as the objective. Modern practice replaces that search with the optimizers of the next chapter, which move the path and the clock together. The lattice of §11.3.3 is the other direct route, and its exponential grid is the reason Chapter 21's state lattices and hybrid A* prune it with motion primitives and a heuristic rather than enumerate it.

What survives into Chapter 19 unchanged: the time-scaled trajectory is the warm start CHOMP and iLQR polish; the bang-bang structure reappears from Pontryagin's minimum principle; and the `Trajectory` type is what every controller there accepts.

18.7 Exercises

1 FOUNDATION • **When is there no limit curve at all?**

Show that a single actuator with constant symmetric bounds $|u| \le u_{\max}$ and $a(s) \neq 0$ has $L < U$ at every state, so no velocity-limit curve exists and the time-optimal clock is pure bang-bang with one switch. Then replace the bound by the DC-motor law $u^{\max}(s, \dot{s}) = u_0 - k\dot{s}$ (torque falling with speed) and show that a limit curve appears. Where is it?

> **HINTS**
>
> <>With one row, $L = \beta$ and $U = \alpha$ (for $a > 0$) and $\alpha - \beta = 2u_{\max}/a > 0$ for every $\dot{s}$.</> <>With the motor law, $\alpha = (u_0 - k\dot{s} - b\dot{s}^2 - c)/a$ while $\beta = (-u_0 - b\dot{s}^2 - c)/a$; the cone collapses when $\alpha = \beta$, i.e. $k\dot{s} = 2u_0$. The limit is $\dot{s} = 2u_0/k$, independent of the path — a speed above which the motor cannot produce positive torque at all.</>

2 FOUNDATION •• **Gravity on the micro-example**

For the micro-example path with gravity on, derive $c(s)$ in closed form from Chapter 17's $g(q)$ and show that $U_1(0) = (20 - 19.62)/\pi$. What torque limit on joint 1 would make $s = 0$ inadmissible at rest, and what does that mean physically?

WORK IT OUT

U_1 at s = 0 with gravity on, in units of $1/\text{s}^2$

Answer. 0.121 $c_1(s) = (m_1 r_1 + m_2 L_1)\, a_g \cos q_1 + m_2 r_2\, a_g \cos(q_1 + q_2)$ with $q_1 = \pi s/2$, $q_2 = \pi/2$: at $s = 0$ the second cosine vanishes and $c_1(0) = 2 \cdot 9.81 = 19.62$ N·m. Then $U_1(0) = (20 - 19.62)/\pi = 0.1210$. Any $\tau_1^{\max}$ below 19.62 N·m makes $\alpha_1 < 0 < \beta_1$ impossible to reconcile with $\dot{s} = 0$: $L > 0$ at rest, the shoulder cannot hold the horizontal arm, and Choset's standing assumption fails — the algorithm reports `NoSolution` rather than a trajectory.

3 CONCEPTUAL •• **Predict the switch**

PREDICT FIRST

In the Phase-Plane Racer switch on the micro-example preset (gravity off) and lower τ_2 from 12 to 10 N·m. Where does the single switch go, and does the trajectory now touch v(s)?

- Before s = $\frac{1}{2}$, and it touches v(s)
  Lowering τ_2 lowers v from 3.5254 to 3.2875, but the peak speed 2.52 is still well below it, so the limit curve is not reached. And the switch cannot move earlier: F is unchanged ($L = L_1 = -6.366$ regardless of τ_2) while A_0 is lower.
- ✓ After s = $\frac{1}{2}$, and it does not touch v(s)
  $U_2 = 19.1 - 2.356\,\dot{s}^2$ drops below $U_1 = 6.366$ once $\dot{s} > 2.325$, so A_0 accelerates less near its peak and is lower than before. F is unchanged. A lower A_0 meets F later: the switch moves to s = 0.5071, the peak drops to 2.504 < v = 3.2875, and t_f barely changes, 0.7928 s.
- It stays at s = $\frac{1}{2}$ by symmetry
  The symmetry that put it at $\frac{1}{2}$ was $L_1 = -U_1$ with joint 1 binding on both sides. With $\tau_2 = 10$ joint 2 binds the acceleration side near the peak but not the braking side, so the two halves are no longer mirror images.

Now lower τ_2 to 8 N·m: the switch moves to 0.554 and t_f to 0.7962 s — the elbow motor is throttling the shoulder's acceleration without either of them being "the slow joint".

4 **CONCEPTUAL** •• **Levels and detours on the lattice**

In the Lattice Hopper (1-DOF, $a_{\max} = 1$, $d = 2$) predict the level count and the lattice t_f at $h = \frac{1}{4}$ before running it, then at $h = \frac{1}{8}$. Then switch to the 2-DOF room, set $c_1 = 0.5$ and explain the longer, slower trajectory in terms of Choset's figure 11.11.

WORK IT OUT

Lattice t_f at h = $\frac{1}{4}$ for d = 2, a_max = 1, in seconds

Answer. 3 s The continuous bound is $2\sqrt{2} = 2.828$ s. Levels are multiples of h, and the rest-to-rest sequence must have as many +1 as −1 controls with the position landing on the goal cell, so 12 levels at $h = \frac{1}{4}$ (3.0 s) and 23 at $h = \frac{1}{8}$ (2.875 s). With $c_1 > 0$ the required clearance grows with speed; near the disc the fast straight trajectories are pruned and the search returns 14 levels at $h = \frac{1}{4}$ — a detour that is slower *and* longer, because the safe speed past the obstacle is the binding constraint, exactly as in figure 11.11.

5 PRACTICAL •• **Joint-rate limits as rows of the limit curve**

Add joint-velocity limits $|\dot{q}_i| \le v_{\max,i}$ to `limits.rs` as additional rows of the velocity-limit computation. Since $\dot{q}_i = q_i'\dot{s}$, each is the bound $\dot{s} \le v_{\max,i}/|q_i'(s)|$, a critical-type point ($L < U$ there). Show in a test that $\dot{s}^{\max}(s)$ becomes $\min(v(s), \dot{s}_{zip}, \min_i v_{\max,i}/|q_i'(s)|)$, and that the time scaler slides along the rate limit where it is active rather than bouncing. The TypeScript port's `jointRateLimit` and the `'rate'` kind of `LimitPoint` are the reference.

6 PRACTICAL ••• **TOPP-RA versus the switching construction (stretch)**

Run `toppra` and `time_scale` on 50 seeded natural cubic splines for Chapter 17's `Reach3R` with gravity, limits drawn from a fixed range, $N = 800$. Report, honestly, the fraction on which both report no solution, the fraction on which exactly one does, and the distribution of $|t_f^{\text{TOPP-RA}} - t_f|/t_f$ on the rest. Explain every disagreement larger than the grid resolution: a zero-inertia point the slope test mishandles, a stage constraint TOPP-RA enforces only at s_k, or an inadmissible island (footnote 2) that neither algorithm is built for. The 2-DOF version of this experiment is the chapter's check: six solvable splines of eight, no disagreement on solvability, worst relative difference 3.8×10^{-3}.

18.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

Chapter 11 is the source: the path-constrained dynamics and the phase plane (§11.2), the Time-Scaling Algorithm and its footnote-3 bisection, zero-inertia points (§11.2.1), Example 11.2.1 (whose printed parameters this chapter re-examines), the Shiller–Dubowsky sketch (§11.2.2), and GRID SEARCH with Theorem 11.3.2 (§11.3.3). Choset's bibliography is where Shin & McKay [385], Pfeiffer & Johanni [348], Slotine & Yang [388] and Donald & Xavier [135] are cited. `link`

Bobrow, J. E., Dubowsky, S., and Gibson, J. S. (1985). Time-Optimal Control of Robotic Manipulators Along Specified Paths. *International Journal of Robotics Research 4(3)*.

One of the two 1985 papers that solved the decoupled problem; the phase-plane construction with its switching curves is theirs (Shin & McKay's companion paper in IEEE Transactions on Automatic Control 30(6) reached the same result independently). `doi:10.1177/027836498500400301`

Pham, H. and Pham, Q.-C. (2018). A New Approach to Time-Optimal Path Parameterization Based on Reachability Analysis. *IEEE Transactions on Robotics 34(3)*.
TOPP-RA: controllable sets in squared speed propagated by small linear programs, then a greedy forward pass. The reformulation that replaces switch-point hunting, implemented here with exact two-variable LPs and checked against the switching construction. doi:10.1109/TRO.2018.2819195

Pivtoraiko, M., Knepper, R. A., and Kelly, A. (2009). Differentially Constrained Mobile Robot Motion Planning in State Lattices. *Journal of Field Robotics 26(3)*.
The state lattice as the descendant of GRID SEARCH: motion primitives on a regular lattice of states, searched with a heuristic. Chapter 21 develops the lineage. doi:10.1002/rob.20285

Dolgov, D., Thrun, S., Montemerlo, M., and Diebel, J. (2010). Path Planning for Autonomous Vehicles in Unknown Semi-Structured Environments. *International Journal of Robotics Research 29(5)*.
Hybrid A*: the practical end of the GRID SEARCH → state lattice line, with continuous states attached to discrete cells. Cited here for the lineage; Chapter 21 implements it. doi:10.1177/0278364909359210

LaValle, S. M. and Kuffner, J. J. (2001). Randomized Kinodynamic Planning. *International Journal of Robotics Research 20(5)*.
Choset's 'third approach' to direct trajectory planning: trade the lattice's optimality for probabilistic completeness in (q, $\dot{q}$). Chapter 21's kinodynamic trees use the same state. doi:10.1177/02783640122067453

Lynch, K. M. and Park, F. C. (2017). Modern Robotics: Mechanics, Planning, and Control. *Cambridge University Press*.
Section 9.4 presents the same time-scaling algorithm, by one of Choset's co-authors, with the (s, $\dot{s}$) phase plane drawn the way w18.1 draws it. link

CHAPTER 19

Trajectory Optimization and MPC

DYNAMICS, TRAJECTORIES, AND CONSTRAINTS · Advanced · 75 min

> *Since nonlinear optimization uses local gradient information, it will converge to a locally optimal solution. For some problems, the control parameter space will be littered with many local optima. Therefore, the solution achieved will depend heavily on the initial guess.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), §11.3.2*

Chapter 18 fixed the path and optimized the clock. This chapter lets the path move too, and discovers that once a trajectory is a vector of numbers, "planning" becomes "descend a cost". Choset gave this two pages of optimal control and three of nonlinear programming, with the warning quoted above. Twenty years later that warning is the whole discipline: CHOMP, STOMP, TrajOpt, iLQR and model predictive control are five answers to *which* descent, in *which* metric, with obstacles as *cost* or as *constraint*.

The idea the chapter unpacks is this. **A trajectory optimizer is a local planner with a global vocabulary.** It polishes the homotopy class it is handed — brilliantly: smoother, shorter, dynamically feasible, closed-loop — and it never changes its mind. Hand it the other side of the obstacle and it will polish that instead. So the pipeline Chapter 23 ships keeps Part III's sampling planners for the global decision and uses this chapter's machinery to make that decision fast, smooth and safe.

Every method here is *local*, and the prose never says "optimal" without "locally". CHOMP's obstacle term is a cost, so a collision can survive convergence, and the widget shows it when it happens. Our `sco` is a recipe in TrajOpt's shape, not TrajOpt's QP-based solver, and says so. iLQR needs a regularization and

a line search, and the defaults are stated. MPC's stability through a terminal Lyapunov cost is a pointer, not a theorem proved.

AFTER THIS CHAPTER YOU CAN

- State Pontryagin's minimum principle, derive the min-effort double integrator's u(t) = 6d/t_f^2 − 12d t/t_f^3 and recover Chapter 18's bang-bang result from the same Hamiltonian
- Transcribe an optimal-control problem to a nonlinear program the three ways Choset lists, and name their modern descendants: collocation (CHOMP), shooting (iLQR), full collocation (TrajOpt)
- Derive CHOMP's covariant functional gradient, say why A^{-1} spreads a push along the whole path, find the step bound $\eta > \frac{1}{2}(1 + (\lambda/\varepsilon)/\lambda\_\min(A))$ that keeps it from oscillating, and compute the micro-example's equilibrium clearance $(\lambda - 1)/(2(1 + \lambda))$
- Run the iLQR backward pass as a quadratic value recursion with a feedback policy, verify that on a linear-quadratic problem it is the Riccati recursion, and add DDP's second-order terms
- Close the loop as MPC, and show why a terminal cost-to-go from Chapter 6 — not a longer horizon — is what cures receding-horizon myopia
- Implement the trajopt crate: a shared Gauss–Newton/Levenberg–Marquardt optimizer that Chapter 21 imports, chomp, stomp, sco, ilqr and mpc, each checked against a closed form or an invariant

ASSUMED

- Distance fields and the J^T lift of workspace forces to joint torques (Chapter 7); body-point Jacobians (Chapter 4)
- Best-first search and the wave-front cost-to-go (Chapter 6); RRT paths and shortcutting (Chapters 12–13)
- The standard form of dynamics and its RK4 integrator (Chapter 17); time scaling and the Trajectory type (Chapter 18)
- Multivariable calculus: gradients, Hessians, Lagrange multipliers; one linear-algebra fact — a tridiagonal system solves in O(N)

19.1 The problem: a jagged path, polished

Take a Chapter 12 RRT path for Reach on the Workbench, seeded, fourteen nodes from the home pose to the far side of the block. Shortcut it with Chapter 13's greedy shortcutter — three waypoints survive — resample to twenty-four, and run forty CHOMP steps against the Chapter 7 distance field of the block. Then

hand both the original and the polished path to Chapter 18's time scaler on a horizontal Workbench with the same motors.

> Left: the RRT path, time-scaled, 1.803 s. Right: after shortcutting and forty CHOMP iterations, time-scaled, 0.992 s. Same arm, same start and goal, same $|\tau| \le (20, 12)$ N·m; the smoothed path keeps 0.146 m of clearance from the block and executes in 55% of the time, because the time scaler is only as fast as the path's curvature lets it be.

Nothing about the planner's global decision changed: both paths go around the block the same way. What changed is everything the sampling planner never cared about — curvature, clearance, and therefore the clock. The question of this chapter is how to make that change deliberately, and what "deliberately" cannot do.

19.2 Building intuition

19.2.1 A rubber band in a force field

Think of the discretized trajectory $\xi = (q_1, \dots, q_{N-1})$ as a chain of beads on a rubber band with its ends pinned at q_0 and q_N. Smoothness pulls each bead toward the midpoint of its neighbours. The obstacle term is a force field: zero beyond a margin ϵ from the nearest obstacle, growing as a bead approaches, pushing straight out along ∇D. CHOMP is what the band does when you let go — with one twist that makes it work on robots: a push on one bead is spread along the whole band by A^{-1}, so the band translates and bends instead of growing a kink where it was touched.

> INTERACTIVE · w19.1
>
> **CHOMP Smoother**
>
> Optimization does not find the best path — it finds the nearest stationary point of the homotopy class it started in, and can converge in collision; switch the initial path and the answer changes.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch19-trajectory-optimization#w19.1

Three things to notice, each of which becomes a theorem below.

The step size has a stability bound, and it grows with the path's length. Slide η down and the band oscillates — on the micro-example preset it flips between the disc's centre and the edge of the margin. The bound is $\eta > \frac{1}{2}(1 + (\lambda/\epsilon)/\lambda_{\min}(A))$: for one free waypoint and $\epsilon = \frac{1}{2}$ it is $(1 + \lambda)/2$; for twenty waypoints it is about 90, because A^{-1} amplifies a low-frequency push by $(m + 1)^2/\pi^2$. Above the bound, the step that is right for a short path crawls on a long one.

The answer depends on the initial path. The two initial paths on the Workbench start in different homotopy classes around the block. CHOMP lowers the cost of both and they end 1.98 rad apart at mid-path. From the straight chart line it converges — gradient at machine precision — with a body point still inside the block, at $D = -0.183$ m, and the badge says **converged in collision**. That is not a bug in the optimizer. The obstacle term is a cost; the smoothness term is a cost; the sum has a stationary point there, and a gradient method finds stationary points.

Clearance is bought with λ and ϵ, not asked for. On the micro-example preset the single waypoint settles where $2y = \lambda(2 - 2y)$: at $y^* = \lambda/(1+\lambda)$, clearance $D^* = (\lambda - 1)/(2(1+\lambda))$ — zero at $\lambda = 1$, 1/6 at 2, 0.3 at 4. Raise λ and the band stands further off; it never stands off by exactly what you need. The sequential-convex recipe later in the chapter is the method that does.

THE ONE-SENTENCE VERSION

Every optimizer in this chapter minimizes a cost over a finite vector of trajectory parameters by local steps — first-order in some metric, second-order in a local model, or by sampling — and so every one of them converges to the nearest stationary point of the basin it started in. What they differ in is the metric, the model, and whether obstacles are a cost or a constraint.

$x = (q, \dot{q})$, $\dot{x} = f(x, u)$	State and state equation (Choset eq. 11.24).
$J = \int_0^{t_f} \ell(x, u, t)\, dt$	Objective; ℓ is the running cost — Choset's optimal-control Lagrangian, renamed to avoid the dynamics L.
$H = \ell + \lambda^{\mathsf{T}} f$, $\lambda(t)$	Hamiltonian and costate (Lagrange multipliers for the state equation), eq. 11.26.
X	Finite parameter vector of the transcribed program (eq. 11.40).
$\xi = (q_1, \ldots, q_{N-1})$	Discretized trajectory with fixed endpoints q_0, q_N.
E, $A = \mathsf{E}^{\mathsf{T}}\mathsf{E}$, $\mathcal{F}_{smooth} = \frac{1}{2}\|\mathsf{E}\xi + e\|^2$	Finite-difference operator; the smoothness metric A = tridiag(−1, 2, −1); the smoothness functional.
$c(d)$, ϵ, $\mathcal{F}_{obs}$, λ_{obs}	Obstacle cost of a distance, its margin, the obstacle functional, its weight.
$\bar{\nabla}\mathcal{F} = A^{-1}\nabla\mathcal{F}$, η	Covariant gradient; the step is $1/\eta$.
x_k, u_k; f_x, f_u; $\ell_x, \ell_{xx}, \ldots$	Discrete-time trajectory and its local derivatives.
$V_k(x)$, V_x, V_{xx}; Q_x, Q_u, Q_{xx}, Q_{uu}, Q_{ux}	Value function and its quadratic model; the action-value expansion at one stage.
k_k, $\mathbf{K}_k$, μ, α	Feedforward and feedback gains; regularization on Q_uu; line-search step.
$sd(\cdot)$, $\hat{n}$, Δ, μ_{pen}; N_h	Signed distance, contact normal, trust radius, penalty weight (TrajOpt); the MPC horizon.

19.3 The mathematics

19.3.1 Definitions

Direct trajectory planning (Choset §11.3) searches the full state space for $(x(t), u(t))$ jointly, rather than a path first and a clock second.

An extremal control satisfies $\partial H/\partial u = 0$ along the trajectory — necessary, not sufficient; it is **locally optimal** if also $\partial^2 H/\partial u^2 > 0$ (the Legendre–Clebsch condition, eq. 11.29). A **singular optimal control** is extremal with $\partial^2 H/\partial u^2$ only semidefinite: the min-time double integrator, and Chapter 18's zero-inertia arcs.

Transcription replaces $(x(t), u(t))$ by a finite X and the continuous constraints by finitely many checks at $j\Delta t$ (eqs. 11.40–11.45).

The covariant gradient is steepest descent in the metric A: $\bar{\nabla}\mathcal{F} = A^{-1}\nabla\mathcal{F}$, the direction that most decreases $\mathcal{F}$ per unit of $\frac{1}{2}\,\delta\xi^{\mathsf{T}} A\,\delta\xi$ — a perturbation measured by how much it changes the *whole* path's smoothness, not by how far one bead moves.

Receding horizon (MPC). Each tick, solve a horizon-N_h problem from the measured state, apply the first control, discard the rest, repeat.

19.3.2 Pontryagin and the min-effort double integrator

STATEMENT (CHOSET EXAMPLE 11.3.1)

For $\ddot{q} = u$, $\ell = u^2$, rest to rest over distance d in fixed t_f, the extremal control is $u(t) = 6d/t_f^2 - 12d\,t/t_f^3$, and it is a local minimum.

DERIVATION **The minimum principle on the simplest system**

$u(t) = 6d/t_f^2 - 12d\,t/t_f^3, \quad \partial^2 H/\partial u^2 = 2 > 0$

Step 1 — the Hamiltonian. With $x = (q, \dot{q})$, $f = (x_2, u)$ and $\ell = u^2$: $H = u^2 + \lambda_1 x_2 + \lambda_2 u$ (eq. 11.26).
Step 2 — extremality. $\partial H/\partial u = 2u + \lambda_2 = 0$, so $u = -\lambda_2/2$ (eq. 11.31).
Step 3 — the adjoint equation. $\dot{\lambda} = -\partial H/\partial x$ gives $\dot{\lambda}_1 = 0$ and $\dot{\lambda}_2 = -\lambda_1$ (eq. 11.32): λ_1 is constant, λ_2 is affine in t, and therefore so is $u(t) = c_0 + c_1 t$.
Step 4 — the terminal conditions. $x_2(t_f) = \int_0^{t_f} u\,dt = 0$ gives $c_0 t_f + c_1 t_f^2/2 = 0$; $x_1(t_f) = \int_0^{t_f}\int_0^{t} u = d$ gives $c_0 t_f^2/2 + c_1 t_f^3/6 = d$. Solving: $c_0 = 6d/t_f^2$, $c_1 = -12d/t_f^3$.
Step 5 — convexity. $\partial^2 H/\partial u^2 = 2 > 0$: the extremal is a local minimum. The position is the cubic $q(t) = 3d(t/t_f)^2 - 2d(t/t_f)^3$ — the minimum-acceleration-energy profile every spline library rediscovers.

Discretized and solved numerically. With $N = 100$ stages of $\Delta t = 0.01$, $\ell_k = u_k^2 \Delta t$ and a terminal weight of 10^6 on $\|x_N - (1,0)\|^2$, iLQR's first and only iteration returns $u_0 = 5.9405$ against Pontryagin's $6 - 12 \cdot 0.005 = 5.94$, with $\max_k |u_k - (6 - 12\, t_{k+1/2})| = 5.1 \times 10^{-4}$ and $x_N = (1,0)$ to 10^{-4}. One iteration, because the problem is linear-quadratic — the next derivation but one says why.

19.3.3 Minimum time is bang-bang, by the minimum principle

STATEMENT (CHOSET §11.3.1)

With $\ell = 1$, $|u| \le u_{\max}$ and t_f free, $u^* = -u_{\max} \operatorname{sign}(\lambda_2)$, with at most one switch.

DERIVATION Where the control set's boundary does the work

$u^* = -u_{\max} \operatorname{sign} \lambda_2$, one switch at $t_s = \sqrt{d/u_{\max}}$

Step 1 — no u in $\partial H/\partial u$. $H = 1 + \lambda_1 x_2 + \lambda_2 u$ is affine in u; $\partial H/\partial u = \lambda_2$ says nothing about u, and $\partial^2 H/\partial u^2 = 0$, so Legendre–Clebsch is silent. The extremality condition 11.28 is useless here.
Step 2 — use the inequality instead. The minimum principle 11.27 says $H^* \le H$ for every feasible u: $\lambda_2 u^* \le \lambda_2 u$ for all $|u| \le u_{\max}$.
Step 3 — so u^* is at a bound. If $\lambda_2 < 0$ the inequality forces $u^* = u_{\max}$; if $\lambda_2 > 0$, $u^* = -u_{\max}$. The control lives on the boundary of its set — bang-bang.
Step 4 — at most one switch. λ_2 is affine in t (same adjoint equation), so it changes sign at most once. Rest to rest over d: accelerate for t_s, brake for t_s, with $u_{\max} t_s^2 = d$, so $t_s = \sqrt{d/u_{\max}}$ — the 0.3963 s of Chapter 18's micro-example read off a Hamiltonian instead of a phase plane.
Shooting. When the extremality conditions cannot be solved by hand, guess $\lambda(0)$, integrate state and adjoint with u from $\partial H/\partial u = 0$, compare $x(t_f)$ with the target, and correct the guess — typically by a Gauss–Newton step on the terminal miss. This is where the "initial guess" problem is first explicit, and `optim::lm` below is the corrector.

19.3.4 Transcription

STATEMENT (CHOSET §11.3.2)

The program 11.33–11.39 becomes the finite program 11.40–11.45 once (a) $q(t)$, or (b) $u(t)$, or (c) both are parameterized and the inequality constraints are enforced at $k+1$ points; (a) computes u from the dynamics, (b) integrates them, (c) adds the dynamics as equality constraints.

DERIVATION Three ways to make a trajectory a vector

$X \in \mathbb{R}^p$; $\min J(X)$ s.t. $g(X) \le 0$, $h(X) = 0$

Step 1 — choose a basis. Choset's list: polynomial coefficients, a truncated Fourier series, spline coefficients, wavelets, piecewise-constant accelerations or forces. The basis decides whether one parameter moves the whole trajectory or a finite support, which decides the conditioning of everything downstream.
Step 2 — sample the constraints. Torque limits 11.42 and obstacle constraints 11.43 are checked at $j\Delta t$, $j = 0 \dots k$; endpoint conditions 11.44–11.45 are equalities on X.
Step 3 — hand it to a solver. Sequential quadratic programming needs J, g, h and their gradients, usually a Hessian approximation too, and all of it at least C^1 in X. Choset's two warnings are exactly the two the modern methods still trip on: smoothness in X, and local optima.
The modern names. (a) is *state collocation*: CHOMP's ξ is a sampled $q(t)$, and its smoothness functional is the acceleration energy the dynamics would charge. (b) is *direct shooting*: iLQR parameterizes u_k and integrates f to get x_k. (c) is *full collocation*: TrajOpt carries both and enforces the dynamics as equalities. Choset's penalty variant — constraints folded into the objective — is CHOMP's obstacle term and TrajOpt's outer loop.

19.3.5 CHOMP's covariant functional gradient

STATEMENT (RATLIFF ET AL. 2009; ZUCKER ET AL. 2013)

With $\mathcal{F} = \mathcal{F}_{smooth} + \lambda_{obs}\mathcal{F}_{obs}$, $\nabla\mathcal{F}_{smooth} = A\xi + b$, and for a body point $x(q,\beta)$ moving at $\|x'\|$ along the path

$$\nabla_q \mathcal{F}_{obs} = \sum_{\beta} J_\beta^{\mathsf{T}} \|x'\| \left[(\mathbb{1} - \hat{x}'\hat{x}'^{\mathsf{T}}) \nabla c - c\,\kappa \right], \qquad \xi \leftarrow \xi - \tfrac{1}{\eta} A^{-1} \nabla\mathcal{F}.$$

DERIVATION **Steepest descent in the smoothness metric**

$\xi \leftarrow \xi - \frac{1}{\eta} A^{-1}(A\xi + b + \lambda \nabla \mathcal{F}_{obs})$

Step 1 — the smoothness functional is a quadratic. $\mathcal{F}_{smooth} = \frac{1}{2}\sum_k \|q_{k+1} - q_k\|^2 = \frac{1}{2}\|\mathsf{E}\xi + e\|^2$ with E the finite-difference operator on the free waypoints and e carrying the fixed ends. Its gradient is $A\xi + b$ with $A = \mathsf{E}^\mathsf{T}\mathsf{E} = \text{tridiag}(-1, 2, -1) \otimes \mathbb{1}_3$, i.e. $2q_k - q_{k-1} - q_{k+1}$ at each free waypoint.

Step 2 — the obstacle functional is an arc-length integral. $\mathcal{F}_{obs} = \sum_\beta \int_0^1 c(D(x_\beta(q(s)))) \, \|x'_\beta\| \, ds$: cost per metre travelled by each body point, so retiming the path does not change it, and a longer path through cost costs more.

Step 3 — differentiate under the integral. The variation of $c(D(x))\|x'\|$ with respect to the body point's path gives the Euler–Lagrange expression $\|x'\|\left[(\mathbb{1}_3 - \hat{x}'\hat{x}'^\mathsf{T})\nabla c - c\,\kappa\right]$, with $\kappa = (\mathbb{1}_3 - \hat{x}'\hat{x}'^\mathsf{T})x''/\|x'\|^2$ the path's curvature. The projection $(\mathbb{1}_3 - \hat{x}'\hat{x}'^\mathsf{T})$ discards the component of ∇c along the path — pushing a bead *along* the band does nothing but retime it — and the curvature term says that where the cost is positive, straightening the path shortens the exposure.

Step 4 — pull back to joint space. A workspace force f at body point β is the joint torque $J_\beta^\mathsf{T} f$ — Chapter 7's §4.7 lift, Chapter 17's eq. 10.12, the same virtual-work identity every time. Summing over body points gives $\nabla_q \mathcal{F}_{obs}$ at each waypoint. The library uses Chapter 7's `controlPointJacobian` for J_β unchanged.

Step 5 — the covariant step. Minimize the first-order model $\mathcal{F}(\xi) + \nabla\mathcal{F}^\mathsf{T}\delta\xi$ subject to $\frac{1}{2}\,\delta\xi^\mathsf{T} A\,\delta\xi \le \text{const}$. The Lagrange condition gives $\delta\xi \propto -A^{-1}\nabla\mathcal{F}$: steepest descent in the metric that measures a perturbation by the smoothness it costs. A is block-tridiagonal, so $A^{-1}\nabla\mathcal{F}$ is one Thomas sweep per coordinate, $O(N)$; the library's sweep agrees with the dense inverse to 4×10^{-15}.

The obstacle cost and its slope. $c(d) = -d + \epsilon/2$ for $d < 0$, $(\epsilon - d)^2/(2\epsilon)$ on $[0, \epsilon]$, 0 above: C^1 at both seams, with $c'(0) = -1$ on both sides. The *pointwise* form $\nabla_q \mathcal{F}_{obs} = \sum_\beta J_\beta^\mathsf{T} c'(D)\nabla D$ drops the arc-length weighting; it is what the micro-example pins exactly, and the library checks it against finite differences of its cost to 4×10^{-10}.

Why A^{-1} spreads a push. A^{-1} is dense and positive: its columns are tent functions peaked at the pushed waypoint and decaying linearly to the pinned ends. A force on one bead moves every bead, most where it was pushed — the band bends rather than kinks.

The step bound. Linearize the obstacle term as a Hessian of at most λ/ϵ per waypoint ($c'' = 1/\epsilon$ in the margin, $\|\nabla D\| \le 1$). The iteration matrix is

$\mathbb{1} - \frac{1}{\eta}(\mathbb{1} + A^{-1}H)$ with eigenvalues $1-(1+\mu_i)/\eta$, $\mu_i \in [0, (\lambda/\epsilon)/\lambda_{\min}(A)]$ and $\lambda_{\min}(A) = 2-2\cos(\pi/(m+1))$. It is a contraction iff $\eta > \frac{1}{2}(1+(\lambda/\epsilon)/\lambda_{\min}(A))$. For $m = 1$, $\epsilon = \frac{1}{2}$: $\eta > (1+\lambda)/2$. For $m = 20$, $\lambda = 2$, $\epsilon = \frac{1}{2}$: $\eta^* = 90.0$; the library runs at $2\eta^*$ without a single cost increase in 400 steps and at $\eta^*/4$ records 36 increases in 100. The bound is worst-case — a path with few waypoints inside the margin tolerates a smaller η — and it is the mechanism behind the widget's headline.

STOMP's gradient-free replacement. Sample $\delta\xi_k \sim \mathcal{N}(0, A^{-1})$ for K rollouts — smooth noise, because its covariance is the smoothness metric — evaluate $S_k = \mathcal{F}(\xi + \delta\xi_k)$, weight $w_k \propto \exp(-h(S_k - S_{\min})/(S_{\max} - S_{\min}))$, update by $\sum_k w_k \delta\xi_k$ smoothed once more through A^{-1}. No gradient is formed, so any cost that can be *evaluated* can be optimized. On the micro-example at $\lambda = 2$, seed 7, $K = 24$, 120 iterations end at $y = 0.6566$ against CHOMP's $y^* = 0.6667$, with the accepted cost never rising.

The micro-example, step by step. A point robot, three waypoints with the ends fixed: $q_0 = (0,0)$, $q_2 = (2,0)$, free $q_1 = (1, 0.2)$; one disc at $(1,0)$ of radius $\frac{1}{2}$, so $D(q_1) = 0.2 - 0.5 = -0.3$ (inside) and $\nabla D = (0,1)$; $\epsilon = \frac{1}{2}$, $\lambda = 1$, $\eta = 1$, unit spacing, pointwise gradient. Smoothness: $\nabla_{q_1}\mathcal{F}_{smooth} = 2q_1 - q_0 - q_2 = (0, 0.4)$, $A = 2$. Obstacle: $D < 0$, so $c = 0.55$ and $c' = -1$, giving $\nabla_{q_1}\mathcal{F}_{obs} = (0, -1)$. Total $(0, -0.6)$; covariant step $q_1 \leftarrow (1, 0.2) - \frac{1}{1} \cdot \frac{1}{2}(0, -0.6) = (1, 0.5)$: one step lands the waypoint exactly on the disc, $D = 0$. Second step: $\nabla\mathcal{F}_{smooth} = (0, 1.0)$, $c'(0) = -1$ so $\nabla\mathcal{F}_{obs} = (0, -1)$, total 0 — a fixed point *in contact*. In general, inside the margin $c' = -(\epsilon - D)/\epsilon$ and the equilibrium is $y^* = \lambda/(1+\lambda)$, clearance $D^* = (\lambda - 1)/(2(1+\lambda))$: $0, 1/6, 0.3$ for $\lambda = 1, 2, 4$, which the library reaches to 10^{-4} in 200 iterations at $\eta = 4$. The lesson: clearance is bought with λ and ϵ; it is never guaranteed.

19.3.6 The iLQR backward pass

STATEMENT

With $Q_x = \ell_x + f_x^\top V'_x$, $Q_u = \ell_u + f_u^\top V'_x$, $Q_{xx} = \ell_{xx} + f_x^\top V'_{xx} f_x$, $Q_{uu} = \ell_{uu} + f_u^\top V'_{xx} f_u$, $Q_{ux} = \ell_{ux} + f_u^\top V'_{xx} f_x$, the optimal local policy is $\delta u = k + \mathbf{K}\,\delta x$ with $k = -Q_{uu}^{-1} Q_u$, $\mathbf{K} = -Q_{uu}^{-1} Q_{ux}$, and the value model recurses as $V_x = Q_x - \mathbf{K}^\top Q_{uu} k$, $V_{xx} = Q_{xx} - \mathbf{K}^\top Q_{uu} \mathbf{K}$.

DERIVATION **Bellman, expanded to second order about a nominal rollout**

$k = -Q_{uu}^{-1} Q_u$, $\mathbf{K} = -Q_{uu}^{-1} Q_{ux}$, $V_{xx} = Q_{xx} - \mathbf{K}^\top Q_{uu} \mathbf{K}$

Step 1 — Bellman. $V_k(x) = \min_u \left[\ell(x,u) + V_{k+1}(f(x,u))\right]$, with $V_N = \ell_f$.
Step 2 — expand about the nominal. Write $x = \bar{x}_k + \delta x$, $u = \bar{u}_k + \delta u$, and expand the bracket to second order: the coefficients are the Q terms of the statement. Dropping the second derivatives of f — the tensor terms $V'_x \cdot f_{xx}$, $V'_x \cdot f_{uu}$, $V'_x \cdot f_{ux}$ — is iLQR; keeping them is DDP. The library computes them by central differences of the Jacobians when `ddp: true`.
Step 3 — minimize the quadratic in δu. $\partial/\partial\delta u = 0$ gives $Q_u + Q_{uu}\delta u + Q_{ux}\delta x = 0$, i.e. $\delta u = -Q_{uu}^{-1}(Q_u + Q_{ux}\delta x) = k + \mathbf{K}\delta x$. This is a *policy*: a feedforward correction plus a linear feedback on the deviation from the nominal.
Step 4 — substitute back. The minimized quadratic is the new value model: $V_x = Q_x + \mathbf{K}^\mathsf{T}Q_{uu}k + \mathbf{K}^\mathsf{T}Q_u + Q_{ux}^\mathsf{T}k$ and $V_{xx} = Q_{xx} + \mathbf{K}^\mathsf{T}Q_{uu}\mathbf{K} + \mathbf{K}^\mathsf{T}Q_{ux} + Q_{ux}^\mathsf{T}\mathbf{K}$, which collapse to the statement's forms once k and $\mathbf{K}$ are substituted (Tassa et al. keep the cross terms for numerical symmetry; so does the library).
Step 5 — recurse, then roll out. From $V_N = \ell_f$ down to $k = 0$. The forward pass is closed-loop: $u_k = \bar{u}_k + \alpha k_k + \mathbf{K}_k(x_k - \bar{x}_k)$, with backtracking on α against the expected decrease $\Delta V(\alpha) = \alpha \sum_k k_k^\mathsf{T}Q_u + \frac{1}{2}\alpha^2 \sum_k k_k^\mathsf{T}Q_{uu}k_k$. A step is accepted when the realized decrease is a reasonable fraction of the promise.
Regularization. Q_{uu} need not be positive definite far from a minimum; the library adds $\mu\mathbb{1}$, raises μ tenfold when a Cholesky pivot fails or no α descends, and lowers it on success. Defaults: $\mu_0 = 10^{-6}$ (zero for linear-quadratic checks), $\alpha \in \{1, \frac{1}{2}, \ldots, 1/64\}$.
On a linear-quadratic problem one pass is Riccati. If $f = Ax + Bu$ and ℓ is quadratic about the origin, $f_x = A$, $f_u = B$ and the recursion for V_{xx} is exactly the discrete Riccati recursion $P_k = Q + A^\mathsf{T}P_{k+1}A + A^\mathsf{T}P_{k+1}B\,\mathbf{K}_k$ with $\mathbf{K}_k = -(R + B^\mathsf{T}P_{k+1}B)^{-1}B^\mathsf{T}P_{k+1}A$. The library computes that recursion independently; on a 40-stage double-integrator problem the gains agree to 1.1×10^{-15} and the $\alpha = 1$ forward pass lands on the optimal cost $\frac{1}{2}x_0^\mathsf{T}P_0x_0 = 10.88611263$ in one iteration.
The first nonlinear term that breaks this. For the unicycle, f_x depends on θ through $\cos\theta$, $\sin\theta$; the rollout is no longer the linear prediction, $\alpha = 1$ may overshoot, and the pass has to be repeated. Hitch's parking problem below takes 79 iLQR iterations, or 30 with DDP's second-order terms, and the two reach *different* local optima — DDP's cheaper on this instance, 56.24 against 60.62, both parking within 5 cm and 0.05 rad.

INTERACTIVE · w19.2

iLQR Backward Pass

iLQR is not gradient descent on the control sequence: it builds a local value function

and a feedback policy at every stage, and the forward rollout is closed-loop.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch19-trajectory-optimization#w19.2

19.3.7 TrajOpt's sequential convex optimization

STATEMENT (SCHULMAN ET AL. 2013, 2014)

Minimize $J(X) + \mu_{pen} \sum_i |g_i(X)|^+$ by repeatedly convexifying J and g about the current X, solving the convex subproblem inside a trust region $\|\delta X\|_\infty \le \Delta$, accepting if the true merit improves, and increasing μ_{pen} in an outer loop until the constraints hold. Collision constraints are $sd(R(q), \mathcal{WO}_i) \ge d_{safe}$, linearized at the witness point as $sd(q) + \hat{n}^\mathsf{T} J_p(q)\,\delta q$.

DERIVATION An exact penalty, convexified

\text{violation} = 0 \text{ at convergence (for finite } \mu_{pen})

Step 1 — the ℓ_1 penalty is exact. For a finite μ_{pen} larger than the constraints' multipliers, the minimizer of $J + \mu_{pen} \sum |g_i|^+$ satisfies $g_i \le 0$ exactly — not approximately, as a quadratic penalty would. The hinge is non-smooth, which is why the subproblem is solved as a convex program rather than by a gradient step.

Step 2 — convexify. Replace J by its quadratic model and each g_i by its linearization; the hinge of an affine function is convex. For collision, $g = d_{safe} - sd(q)$ and the linearization at the witness pair is $d_{safe} - sd(q_0) - \hat{n}^\mathsf{T} J_p(q_0)\,\delta q$ — exactly the contact normal times the body-point Jacobian, Chapter 7's lift once more.

Step 3 — the trust region guarantees descent on the merit. The convex model agrees with the true merit to first order; inside a small enough box the true merit decreases when the model's does. Accept and expand Δ on success; reject and shrink on failure.

Step 4 — the penalty loop. When the inner loop converges with violations left, multiply μ_{pen} and repeat. Swept volumes between consecutive footprints are handled as convex hulls of the two shapes — a recipe this book states and does not implement.

What our sco simplifies, and says so. The convex subproblem is a QP in TrajOpt. Here it is the book's own Levenberg–Marquardt on the residual vector $[\mathsf{E}\xi + e;\ \sqrt{\mu}\,|g_{lin}|^+]$, boxed by the same Δ — a recipe in TrajOpt's shape, not TrajOpt. On the twelve-waypoint disc problem it ends after 27 convexifications at $\mu_{pen} = 10^5$ with every waypoint at least $d_{safe} = 0.1$ from

the disc, the tightest at 0.1000 and a violation of 6×10^{-7}; CHOMP on the same scene with $\lambda = 1$, $\epsilon = 0.5$ settles at a clearance of 0.471 — far more than asked, because it never saw the constraint. Cost versus constraint, in one number each.

19.3.8 MPC: the loop around any of them

STATEMENT

Each tick: measure x; solve the horizon-N_h problem from x with the obstacles where the sensor says they are; apply u_0; shift the solution one stage as the next warm start. The horizon trades compute for foresight, and past the point where the plan is corrected anyway it buys compute only. A terminal cost that approximates the cost-to-go is what lets a short horizon see around a corner.

INTERACTIVE · w19.3

MPC Horizon

A longer horizon is not always better: past the point where the plan is corrected anyway, horizon buys compute, not safety — and too short a horizon is myopic.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch19-trajectory-optimization#w19.3

The widget's numbers are the chapter's check. Rusty crosses the Apartment from room A to room E, $\Delta t = 0.1$ s, $v_{\max} = 1$ m/s, with a disc sliding in and out of the corridor through room B's door, frozen at its measured position each tick. With the Chapter 6 wave-front cost-to-go as terminal cost, horizons $N_h = 6, 10, 24$ all arrive — in 125, 127 and 139 ticks — without touching the disc or a wall, at 1.9, 3.9 and 10.0 ms of iLQR per tick. With a plain quadratic pull to the goal as terminal cost, neither $N_h = 10$ nor 24 ever arrives: the pull parks Rusty against the corridor wall it cannot see around. And the open-loop ghost at $N_h = 16$, executing each plan to its end before replanning, collides for 10 ticks where MPC waits. Horizon is not what cured myopia; the terminal cost was. Stability of the closed loop via a terminal cost that is a Lyapunov function is the standard theorem (Rawlings, Mayne and Diehl); this chapter points at it and does not prove it.

19.4 The algorithm

ALGORITHM `chomp_step(ξ, q_0, q_N, D, {J_β}, ε, λ, η)` COST `O(N·β) distance queries + one O(N) Thomas sweep per coordinate`

in free waypoints, fixed ends, distance field with gradient, body-point Jacobians, margin, weight, step

out ξ updated in place; F_smooth, F_obs, min clearance

1: $g_k \leftarrow 2q_k - q_{k-1} - q_{k+1}$ for each free waypoint (the smoothness gradient $A\xi + b$)
2: for each waypoint k and body point β: $x \leftarrow x_\beta(q_k)$, $d \leftarrow D(x)$, $f \leftarrow c'(d)\,\nabla D(x)$; in the arclength form, $f \leftarrow \|x'\|\,[(\mathbb{1} - \hat{x}'\hat{x}'^{\mathsf{T}})f - c(d)\,\kappa]$ with x', x'' by central differences along ξ
3: $g_k \leftarrow g_k + \lambda\, J_\beta(q_k)^{\mathsf{T}} f$
4: solve $A\,\delta = g$ by the Thomas sweep on tridiag$(-1, 2, -1)$, one coordinate at a time
5: $\xi \leftarrow \xi - \delta/\eta$; report F_{smooth}, F_{obs}, $\min d$. Stable iff $\eta > \frac{1}{2}(1 + (\lambda/\epsilon)/\lambda_{\min}(A))$.

ALGORITHM `stomp_step(ξ, K, rng, noise, h)` COST `K full cost evaluations; no gradients`

in the same problem, a seeded generator, rollout count, noise scale, temperature

out ξ updated if the cost did not rise

1: $LL^{\mathsf{T}} = A^{-1}$ once; for $k = 1 \ldots K$: $\delta\xi_k \leftarrow \text{noise} \cdot L\,z_k$, $z_k \sim \mathcal{N}(0, \mathbb{1})$ per coordinate
2: $S_k \leftarrow \mathcal{F}(\xi + \delta\xi_k)$; $w_k \leftarrow \exp(-h(S_k - S_{\min})/(S_{\max} - S_{\min}))$, normalized
3: $\delta\xi \leftarrow M \sum_k w_k \delta\xi_k$ with $M = A^{-1}$ column-scaled so its largest entry is $1/m$
4: if $\mathcal{F}(\xi + \delta\xi) \le \mathcal{F}(\xi)$: $\xi \leftarrow \xi + \delta\xi$

ALGORITHM `sco(ξ_0, sd, d_safe, μ_0, Δ_0) -- a recipe in TrajOpt's shape`

COST `outer penalty loop × inner convexify-solve-accept loop; each solve a few LM iterations`

in initial waypoints, signed-distance field, clearance, initial penalty and trust radius

out ξ with violation below tolerance, the merit history

1: repeat (penalty loop): repeat (inner loop):
2: convexify: at each waypoint record $sd(q_k)$ and $\hat{n}_k = \nabla sd(q_k)$; model $g_k(\delta) = d_{safe} - sd(q_k) - \hat{n}_k^\mathsf{T}\delta q_k$
3: solve $\min_\delta \frac{1}{2}\|\mathsf{E}(\xi+\delta)+e\|^2 + \mu\sum_k |g_k(\delta)|^+$ with $\|\delta\|_\infty \le \Delta$ — here by LM on the residuals $[\mathsf{E}(\xi+\delta)+e;\ \sqrt{\mu}\,|g_k(\delta)|^+]$ with the box as trust region (TrajOpt: a QP)
4: if the *true* merit $\frac{1}{2}\|\mathsf{E}\xi+e\|^2 + \mu\sum_k |d_{safe} - sd(q_k)|^+$ decreased: accept, $\Delta \leftarrow 1.5\Delta$; else $\Delta \leftarrow \Delta/2$
5: until four rejections; if the worst violation $\ge$ tol: $\mu \leftarrow 10\mu$ and repeat the penalty loop

ALGORITHM `ilqr(f, ℓ, ℓ_f, x₀, ū, μ₀)` COST `O(N (n_x + n_u)³) per iteration`

in discrete dynamics with Jacobians, running and terminal cost with derivatives, start, initial controls, regularization
out ($\bar{x}$, $\bar{u}$, {k_k}, {**K**_k}), the cost per iteration

1: roll $\bar{u}$ out from x_0; $J \leftarrow \sum \ell + \ell_f$
2: **backward pass:** $V_x, V_{xx} \leftarrow \nabla\ell_f, \nabla^2\ell_f$; for $k = N-1 \dots 0$: form $Q_x, Q_u, Q_{xx}, Q_{uu}, Q_{ux}$ (add $V_x \cdot f_{xx}$ etc. for DDP); if $Q_{uu} + \mu\mathbb{1}$ is not positive definite, $\mu \leftarrow 10\mu$ and restart; $k_k \leftarrow -(Q_{uu}+\mu\mathbb{1})^{-1}Q_u$, $\mathbf{K}_k \leftarrow -(Q_{uu}+\mu\mathbb{1})^{-1}Q_{ux}$; update V_x, V_{xx}; accumulate $\Delta V = \alpha a + \alpha^2 b$
3: **forward pass:** for $\alpha \in \{1, \frac{1}{2}, \dots\}$: $u_k \leftarrow \bar{u}_k + \alpha k_k + \mathbf{K}_k(x_k - \bar{x}_k)$, $x_{k+1} \leftarrow f(x_k, u_k)$; accept the first α whose realized decrease is a fair fraction of $-\Delta V(\alpha)$; $\mu \leftarrow \mu/10$
4: if no α descends: $\mu \leftarrow 10\mu$; repeat from 2 until the relative decrease is below tol

ALGORITHM `mpc_tick(x_meas, t)` COST one warm-started iLQR solve (a few iterations) per tick

in measured state, tick; a horizon N_h, a cost builder seeing the obstacles at t, a warm start
out u_0 to apply; the plan

1: $\bar{u} \leftarrow$ last plan shifted by one stage (its last control repeated), or zeros
2: $(\bar{x}, \bar{u}) \leftarrow$ `ilqr`$(f, \ell_t, \ell_f, x_{meas}, \bar{u})$ with the obstacle frozen where it is at t and ℓ_f the cost-to-go when one is available
3: apply $\bar{u}_0$; keep $\bar{u}$ for the next tick

ALGORITHM `optim::lm(r, J_r, x₀, Δ) -- shared with Chapter 21` COST one m×n Jacobian and one n×n solve per trial step

in residual function (and Jacobian, else central differences), start, optional box trust radius
out x minimizing $\frac{1}{2}\|r(x)\|^2$, cost history

1: $\lambda \leftarrow 10^{-3}$; repeat:
2: $J \leftarrow J_r(x)$; stop if $\|J^{\mathsf{T}} r\|_\infty <$ tol
3: solve $(J^{\mathsf{T}}J + \lambda\,\mathrm{diag}(J^{\mathsf{T}}J))\,\delta = -J^{\mathsf{T}}r$; clip $\|\delta\|_\infty \le \Delta$
4: if $\frac{1}{2}\|r(x+\delta)\|^2 < \frac{1}{2}\|r(x)\|^2$: accept, $\lambda \leftarrow 0.3\lambda$; else $\lambda \leftarrow 10\lambda$ and retry

19.5 Implementation in Rust

The `trajopt` crate is introduced here and imported by Chapters 21 and 23. Its one shared piece is the optimizer; the rest are the five methods, each small enough to read beside its derivation.

RUST **crates/trajopt/src/optim.rs**

```rust
use nalgebra::{DMatrix, DVector};

/// A nonlinear least-squares problem ?||r(x)||2. `jacobian` defaults to central differences.
pub trait LeastSquares {
    fn residual(&self, x: &DVector<f64>) -> DVector<f64>;
    fn jacobian(&self, x: &DVector<f64>) -> DMatrix<f64> { finite_difference_jacobian(|y| self.residual(y), x, 1e-6) }
}

pub struct LmOptions { pub max_iter: usize, pub tol_grad: f64, pub lambda0: f64, pub trust: f64 }

/// Levenberg-Marquardt with Marquardt scaling and a box trust region:
///     (J^TJ + lambda diag(J^TJ)) delta = -J^Tr,   ||delta||inf <= trust,   accept iff the true cost drops.
/// lambda -> 0 is Gauss-Newton, lambda -> inf a short gradient step. The box is the crude trust
/// region `sco` leans on, and Chapter 21's nonlinear steering reuses the whole thing.
pub fn levenberg_marquardt(p: &impl LeastSquares, x0: DVector<f64>, o: &LmOptions) -> LmResult {
    let (mut x, mut lambda) = (x0, o.lambda0);
    let mut r = p.residual(&x);
    let mut cost = 0.5 * r.norm_squared();
    for _ in 0..o.max_iter {
        let j = p.jacobian(&x);
        let g = j.transpose() * &r;
        if g.amax() < o.tol_grad { break; }
        let jtj = j.transpose() * &j;
        loop {
            let mut a = jtj.clone();
            for i in 0..a.nrows() { a[(i, i)] += lambda * jtj[(i, i)].max(1e-12); }
            let mut delta = a.lu().solve(&(-&g)).expect("J^TJ + lambdaD is positive
```

```
    definite");
            let step = delta.amax();
            if step > o.trust { delta *= o.trust / step; }
            let x_trial = &x + &delta;
            let r_trial = p.residual(&x_trial);
            let c_trial = 0.5 * r_trial.norm_squared();
            if c_trial < cost { x = x_trial; r = r_trial; cost = c_trial; lambda *= 0.3;
    break; }
            lambda *= 10.0;
            if lambda > 1e15 { return LmResult::stationary(x, cost); }
        }
     }
     LmResult::converged(x, cost)
}
```

RUST `crates/trajopt/src/chomp.rs`

```rust
use nalgebra::{Point2, SMatrix, SVector};

/// A discretized trajectory with fixed endpoints; xi holds only the N-1 free waypoints.
pub struct ChompProblem<'a, const N: usize> {
    pub q0: SVector<f64, N>, pub q_end: SVector<f64, N>,
    pub field: &'a dyn DistanceField,            // Chapter 7: dist(x), grad(x), signed
    /// Body points: workspace position and 2*N Jacobian at q (Reach: samples along both
    links).
    pub body: Vec<Box<dyn Fn(&SVector<f64, N>) -> (Point2<f64>, SMatrix<f64, 2, N>) + 'a
    >>,
    pub eps: f64, pub lambda_obs: f64, pub eta: f64,
    pub obstacle_form: ObstacleGradient,        // Pointwise (the micro-example) |
    ArcLength (full CHOMP)
}

/// c(d): -d + epsilon/2 below zero, (epsilon - d)2/(2epsilon) on [0, epsilon], 0 above
    -- C1 at both seams.
pub fn obstacle_cost(d: f64, eps: f64) -> f64 { if d < 0.0 { -d + eps / 2.0 } else if d <
    eps { (eps - d).powi(2) / (2.0 * eps) } else { 0.0 } }
pub fn obstacle_slope(d: f64, eps: f64) -> f64 { if d < 0.0 { -1.0 } else if d < eps { -(
    eps - d) / eps } else { 0.0 } }

/// Solve A y = r for A = tridiag(-1, 2, -1) ? I by one Thomas sweep per coordinate: the
/// covariant preconditioner. A-1 is dense and positive -- a push on one waypoint moves
    them all.
pub fn solve_smoothness_metric<const N: usize>(r: &[SVector<f64, N>]) -> Vec<SVector<f64,
    N>> {
    let m = r.len();
    let mut bp = vec![2.0; m];
    let mut dp: Vec<SVector<f64, N>> = r.to_vec();
    for k in 1..m { let w = -1.0 / bp[k - 1]; bp[k] = 2.0 + w; dp[k] = dp[k] - w * dp[k -
    1]; }
    let mut y = vec![SVector::zeros(); m];
    y[m - 1] = dp[m - 1] / bp[m - 1];
    for k in (0..m - 1).rev() { y[k] = (dp[k] + y[k + 1]) / bp[k]; }
    y
}

/// One covariant step xi <- xi - (1/eta) A-1 ?F. Returns the statistics *before* the
    step.
pub fn chomp_step<const N: usize>(p: &ChompProblem<'_, N>, xi: &mut [SVector<f64, N>]) ->
```

```rust
    ChompStats {
    let full: Vec<_> = std::iter::once(p.q0).chain(xi.iter().copied()).chain(std::iter::once(p.q_end)).collect();
    let mut grad: Vec<SVector<f64, N>> = (1..full.len() - 1).map(|k| 2.0 * full[k] - full[k - 1] - full[k + 1]).collect();
    let (mut f_obs, mut min_clear) = (0.0, f64::INFINITY);
    for k in 1..full.len() - 1 {
        for b in &p.body {
            let (x, j) = b(&full[k]);
            let d = p.field.dist(&x);
            min_clear = min_clear.min(d);
            let c = obstacle_cost(d, p.eps);
            let mut f = obstacle_slope(d, p.eps) * p.field.grad(&x);          // ?c = c'(D) ?D
            let mut w = 1.0;
            if p.obstacle_form == ObstacleGradient::ArcLength {
                let (xm, _) = b(&full[k - 1]); let (xp, _) = b(&full[k + 1]);
                let v = 0.5 * (xp - xm); let acc = xp - 2.0 * x + xm;        // x', x'' along the path
                if let Some(t) = v.try_normalize(1e-9) {
                    let proj = |z: nalgebra::Vector2<f64>| z - t * t.dot(&z);
                    let kappa = proj(acc.coords) / v.norm_squared();
                    f = v.norm() * (proj(f) - c * kappa);                   // ||x'||[(I - x'x'^T)?c - c kappa]
                    w = v.norm();
                }
            }
            f_obs += c * w;
            grad[k - 1] += p.lambda_obs * (j.transpose() * f);              // J^T f: Chapter 7's lift
        }
    }
    let dir = solve_smoothness_metric(&grad);
    for (q, d) in xi.iter_mut().zip(dir) { *q -= d / p.eta; }
    ChompStats { f_smooth: smoothness(&full), f_obs, min_clearance: min_clear }
}

/// Sufficient step bound: eta > ?(1 + (lambda/epsilon)/lambda_min(A)), lambda_min(A) = 2 - 2cos(pi/(m+1)).
pub fn stable_eta(m: usize, lambda_obs: f64, eps: f64) -> f64 {
    0.5 * (1.0 + lambda_obs / eps / (2.0 - 2.0 * (std::f64::consts::PI / (m as f64 + 1.0)).cos()))
}
```

RUST `crates/trajopt/src/ilqr.rs`

```rust
/// Discrete-time dynamics x_{k+1} = f(x_k, u_k) with Jacobians; implemented by `models::*`.
pub trait DiscreteDynamics<const NX: usize, const NU: usize> {
    fn step(&self, x: &SVector<f64, NX>, u: &SVector<f64, NU>) -> SVector<f64, NX>;
    /// (f_x, f_u). Default: central differences -- override with analytic forms for speed.
    fn jacobians(&self, x: &SVector<f64, NX>, u: &SVector<f64, NU>) -> (SMatrix<f64, NX, NX>, SMatrix<f64, NX, NU>) {
        numeric_jacobians(|a, b| self.step(a, b), x, u, 1e-6)
    }
}

```

```rust
pub struct BackwardPass<const NX: usize, const NU: usize> {
    pub k: Vec<SVector<f64, NU>>,          // feedforward
    pub big_k: Vec<SMatrix<f64, NU, NX>>,  // feedback gains ?_k -- the policy, not just a plan
    pub v_xx: Vec<SMatrix<f64, NX, NX>>,   // the value model, for the widget's ellipses
    pub expected_decrease: (f64, f64),     // DeltaV(alpha) = alpha.a + alpha2.b, for the line search
    pub ok: bool,                          // every regularized Q_uu was positive definite
}

pub fn backward_pass<const NX: usize, const NU: usize>(
    f: &impl DiscreteDynamics<NX, NU>, cost: &impl StageCost<NX, NU>,
    xs: &[SVector<f64, NX>], us: &[SVector<f64, NU>], mu: f64, ddp: bool,
) -> BackwardPass<NX, NU> {
    let n = us.len();
    let (mut v_x, mut v_xx) = cost.terminal_derivs(&xs[n]);
    let mut bp = BackwardPass::with_capacity(n);
    let (mut a, mut b) = (0.0, 0.0);
    for k in (0..n).rev() {
        let (fx, fu) = f.jacobians(&xs[k], &us[k]);
        let l = cost.running_derivs(&xs[k], &us[k], k);
        let q_x  = l.lx + fx.transpose() * v_x;
        let q_u  = l.lu + fu.transpose() * v_x;
        let mut q_xx = l.lxx + fx.transpose() * v_xx * fx;
        let mut q_uu = l.luu + fu.transpose() * v_xx * fu;
        let mut q_ux = l.lux + fu.transpose() * v_xx * fx;
        if ddp {                                   // the tensor terms V_x . f_xx etc., by differencing the Jacobians
            let so = second_order_terms(f, &xs[k], &us[k], &v_x);
            q_xx += so.xx; q_uu += so.uu; q_ux += so.ux;
        }
        let q_uu_reg = q_uu + mu * SMatrix::identity();
        let Some(chol) = q_uu_reg.cholesky() else { bp.ok = false; return bp; };
        let kk = -chol.solve(&q_u);
        let big_k = -chol.solve(&q_ux);
        a += kk.dot(&q_u);
        b += 0.5 * kk.dot(&(q_uu * kk));
        // Value update with the cross terms (Tassa, Erez & Todorov 2012, eq. 11).
        v_x  = q_x + big_k.transpose() * q_uu * kk + big_k.transpose() * q_u + q_ux.transpose() * kk;
        v_xx = q_xx + big_k.transpose() * q_uu * big_k + big_k.transpose() * q_ux + q_ux.transpose() * big_k;
        v_xx = 0.5 * (v_xx + v_xx.transpose());
        bp.push(k, kk, big_k, v_xx);
    }
    bp.expected_decrease = (a, b);
    bp
}

/// Closed-loop rollout u_k = ?_k + alpha k_k + ?_k (x_k - x_k): the policy, not the plan.

pub fn forward_pass<const NX: usize, const NU: usize>(f: &impl DiscreteDynamics<NX, NU>,
    xs: &[SVector<f64, NX>], us: &[SVector<f64, NU>], bp: &BackwardPass<NX, NU>, alpha: f64)
    -> (Vec<SVector<f64, NX>>, Vec<SVector<f64, NU>>) {
    let mut xs_new = vec![xs[0]];
    let mut us_new = Vec::with_capacity(us.len());
    for k in 0..us.len() {
        let u = us[k] + alpha * bp.k[k] + bp.big_k[k] * (xs_new[k] - xs[k]);
        xs_new.push(f.step(&xs_new[k], &u));
```

```
        us_new.push(u);
    }
    (xs_new, us_new)
}

pub fn ilqr<const NX: usize, const NU: usize>(f: &impl DiscreteDynamics<NX, NU>, cost: &impl StageCost<NX, NU>, x0: SVector<f64, NX>, u_init: &[SVector<f64, NU>], opts: IlqrOptions) -> IlqrResult<NX, NU> {
    let mut us = u_init.to_vec();
    let mut xs = rollout(f, x0, &us);
    let mut j = total_cost(cost, &xs, &us);
    let mut mu = opts.mu0;
    for _ in 0..opts.max_iter {
        let mut bp = backward_pass(f, cost, &xs, &us, mu, opts.ddp);
        while !bp.ok { mu = (mu * 10.0).max(1e-6); bp = backward_pass(f, cost, &xs, &us, mu, opts.ddp); }
        let mut accepted = false;
        for &alpha in &[1.0, 0.5, 0.25, 0.125, 0.0625, 0.03125, 0.015625] {
            let (xs_t, us_t) = forward_pass(f, &xs, &us, &bp, alpha);
            let j_t = total_cost(cost, &xs_t, &us_t);
            let expected = -(alpha * bp.expected_decrease.0 + alpha * alpha * bp.expected_decrease.1);
            if j_t < j && (expected <= 1e-12 || (j - j_t) / expected > 1e-4) {
                let rel = (j - j_t) / j.abs().max(1e-300);
                xs = xs_t; us = us_t; j = j_t; mu = (mu * 0.1).max(opts.mu_min); accepted = true;
                if rel < opts.tol { return IlqrResult::converged(xs, us, bp); }
                break;
            }
        }
        if !accepted { mu *= 10.0; if mu > opts.mu_max { break; } }
    }
    IlqrResult::stopped(xs, us)
}
```

RUST `crates/trajopt/src/mpc.rs`

```
/// Receding horizon around any solver that accepts a warm start.
pub struct Mpc<S: HorizonSolver> { pub solver: S, pub horizon: usize, warm: Option<Vec<S::Control>> }

impl<S: HorizonSolver> Mpc<S> {
    /// One tick: solve from the *measured* state with the obstacles where the sensor says they
    /// are, apply the first control, keep the rest shifted by one stage as the next warm start.
    pub fn tick(&mut self, x_meas: &S::State, t: usize) -> S::Control {
        let init = match &self.warm {
            Some(w) => { let mut u = w[1..].to_vec(); u.push(*w.last().unwrap()); u }
            None => vec![S::Control::zeros(); self.horizon],
        };
        let plan = self.solver.solve(x_meas, &init, t);        // iLQR, a few iterations
        let u0 = plan.controls[0];
        self.warm = Some(plan.controls);
        u0
    }
}

```

```rust
/// Chapter 6's wave-front distance to the goal on an occupancy grid, bilinearly interpolated:
/// the terminal cost that lets a short horizon see around a corner.
pub struct GridCostToGo { grid: OccupancyGrid, value: Vec<f64> }
impl GridCostToGo {
    pub fn new(world: &World, goal: Point2<f64>, cell: f64, inflation: f64) -> Self {
        let grid = grid_from_world(world, cell, inflation);
        let wf = wavefront(&grid, grid.cell_at(goal), Connectivity::Eight);   // Chapter 7's wave-front on Chapter 6's grid
        let value = wf.label.iter().map(|&l| if l >= 2 { (l - 2) as f64 * cell } else { f64::INFINITY }).collect();
        Self { grid, value }
    }
    pub fn at(&self, x: f64, y: f64) -> f64 { self.grid.bilinear(&self.value, x, y) }
}
```

19.5.1 The worked example, printed

`cargo run --example chomp_three -p trajopt` builds the micro-example with an analytic `DiscField` so the numbers are exact (the widget uses the Chapter 7 grid field) and prints

```
grad_smooth = (0.0000, 0.4000)    grad_obs = (0.0000, -1.0000)    c(D) = 0.5500
q1 <- (1.0000, 0.5000)            clearance = 0.0000
step 2: |grad| = 0.0000 (fixed point, in contact)
lambda_obs in {1, 2, 4}, eta = 4, 200 iterations:  D* = 0.0000  0.1667  0.3000
stability: eta* = (1 + lambda)/2;  lambda = 2: eta = 1.4 -> y = 0.4615 (oscillates), eta = 1.6 -> y = 0.6667
20 waypoints, lambda = 2, eps = 0.5:  eta* = 90.0;  2 eta*: 0 cost rises in 400 steps, clearance 0.494;  eta*/4: 36 rises in 100
STOMP seed 7, K = 24, 120 iterations, lambda = 2:  y = 0.6566  (y* = 0.6667)
sco, 12 waypoints, d_safe = 0.1:  27 convexifications, mu = 1e5, tightest clearance 0.1000, violation 5.6e-7
```

`cargo run --example ilqr_min_effort -p trajopt` discretizes $\ddot{q} = u$ at $N = 100$, $t_f = 1$, $d = 1$, cost $\sum u_k^2 \Delta t$ plus $10^6 \|x_N - (1,0)\|^2$, and prints

```
u_0 = 5.9405 (Pontryagin 5.9400)   u_50 = -0.0600 (-0.0600)   max |u_k - (6 - 12 t_{k+1/2})| = 5.1e-4
alpha = 1 accepted; converged in one iteration (linear-quadratic: one pass is Riccati)
LQR check, N = 40 double integrator:  max |K - K_riccati| = 1.1e-15;  cost after one pass 10.88611263 = 1/2 x0' P0 x0
Hitch parking, N = 30:  iLQR 79 iterations 559.76 -> 60.62;  DDP 30 iterations -> 56.24 (different local optima; both park within 5 cm, 0.05 rad)
Reach on the Workbench, 24 waypoints, 8 body points, arclength, eta = 8:
  straight chart line:  F 1.1895 -> 1.0612, |grad| 5e-15, clearance -0.183  (CONVERGED IN COLLISION)
  detour via folded elbow:  F 0.9236 -> 0.3724, clearance 0.099;  mid-path separation 1.98 rad
MPC, Apartment, dt = 0.1:  cost-to-go terminal  N_h = 6: 125 ticks 1.9 ms | 10: 127 ticks 3.9 ms | 24: 139 ticks 10.0 ms, no contact
                           quadratic terminal   N_h = 10, 24: never arrives (myopic);
  open-loop ghost N_h = 16: 10 collision ticks
hook pipeline:  RRT (seed 19) 14 nodes -> shortcut 3 -> 40 CHOMP steps, clearance 0.146;  t_f 1.803 s -> 0.992 s
```

`#[test] fn reproduces_micro_example()` asserts the first block to 10^{-4} and `#[test] fn ilqr_matches_pontryagin()` the second: $\max_k |u_k - (6 - 12\, t_{k+1/2})| < 5 \times 10^{-2}$ and convergence in one iteration. `#[test] fn argmin_agrees()` runs `argmin`'s L-BFGS on the micro-example's objective and asserts the same fixed point to 10^{-6} — the one place a library optimizer appears, as a cross-check and never as the implementation.

> **NOTE**
>
> The widgets on this page run the TypeScript port in `lib/trajopt/`, a line-for-line translation of the Rust above. Its fifteen self-checks reproduce every number in the two printouts and add the invariants: the Thomas sweep against the dense inverse, the pointwise gradient against finite differences, the iLQR gains against an independent Riccati recursion, the car model against Chapter 2's `Hitch.step`, the brushfire field bracketed by the exact Euclidean distance ($d/\sqrt{2} - 1.5$ cells $\le D \le d + 1.5$ cells at all 379 sampled points of the Workbench), and the receding-horizon loop against its open-loop ghost.

19.6 Putting it together

The Reach pipeline the capstone ships is the hook: a Chapter 12 RRT → Chapter 13 shortcut → `chomp` → Chapter 18 `CubicSpline` + `time_scale` → Chapter 17 replay. On the seeded instance above it takes the time-optimal execution from 1.803 s to 0.992 s without changing the planner's global decision. Three honesty items the pipeline makes visible.

Where optimization beats search. Smoothness, clearance and dynamics are what a sampling planner ignores and an optimizer measures. Forty covariant steps cost less than the RRT that found the path, and the time scaler rewards every bit of curvature removed. iLQR goes further than CHOMP can: it respects the dynamics by construction, and it returns a feedback policy — the gains $\mathbf{K}_k$ — which is what makes the receding-horizon loop closed-loop rather than a replanner.

Where it sticks. Homotopy and narrow passages. From the straight chart line, CHOMP on the Workbench converges with a body point inside the block, at machine-precision stationarity. No η, λ or ϵ fixes that; only a different initial path does, which is a sampling planner's job. Choset's "the solution achieved will depend heavily on the initial guess" is the chapter's epigraph for a reason.

Where the loop earns its keep. MPC's value is not foresight — a frozen-obstacle planner has none past its sensor — but correction: the obstacle moves, the plan moves with it, every tick. The Apartment check shows the horizon buying compute (1.9 → 10.0 ms) while the terminal cost-to-go, not the horizon, decides whether Rusty arrives at all. The sister book's MPPI runs the same loop

with sampling in place of iLQR; a side-by-side on the same model, horizon and seeds is the honest way to compare them, and anything less is not shown.

On to Chapter 20, where the question becomes whether the car can reach a configuration at all — and to Chapter 21, where `trajopt::optim` steers it there.

19.7 Exercises

1 FOUNDATION •• **The equilibrium and its stability**

For the three-waypoint micro-example derive $y^* = \lambda/(1+\lambda)$ from $\nabla_{q_1}\mathcal{F} = 0$ inside the margin, then write the covariant update as $y \leftarrow y(1-(1+\lambda)/\eta) + \lambda/\eta$ and show it is a contraction iff $\eta > (1+\lambda)/2$. The design brief for this chapter claimed stability "for every $\eta \geq 1$"; find the smallest λ for which $\eta = 1$ oscillates.

WORK IT OUT

For $\lambda = 2$, the smallest stable η

Answer. 1.5 Inside the margin $c'(D) = -(\epsilon - D)/\epsilon = 2y - 2$ (with $\epsilon = \frac{1}{2}$, $D = y - \frac{1}{2}$), so $\nabla_{q_1}\mathcal{F} = (0,\ 2y + \lambda(2y-2))$ and the update is $y \leftarrow y - \frac{1}{2\eta}\big((2+2\lambda)y - 2\lambda\big)$. Fixed point $y^* = \lambda/(1+\lambda)$; contraction factor $|1-(1+\lambda)/\eta| < 1$ iff $\eta > (1+\lambda)/2$. At $\eta = 1$ the factor is $-\lambda$: marginal at $\lambda = 1$, oscillating for every $\lambda > 1$. The library pins $\eta = 1.4$ oscillating and $\eta = 1.6$ converging at $\lambda = 2$.

2 FOUNDATION •• **One pass is Riccati**

Show that for $x_{k+1} = Ax_k + Bu_k$ and $\ell = \frac{1}{2}(x^\mathsf{T}Qx + u^\mathsf{T}Ru)$, $\ell_f = \frac{1}{2}x^\mathsf{T}Q_f x$, the iLQR backward pass with $\mu = 0$ produces $V_{xx,k} = P_k$ of the discrete Riccati recursion and $\mathbf{K}_k = -(R + B^\mathsf{T}P_{k+1}B)^{-1}B^\mathsf{T}P_{k+1}A$, and that the $\alpha = 1$ forward pass is the optimal trajectory in one iteration. Then exhibit the first term that breaks this for the unicycle $x_{k+1} = x_k + \Delta t\,(v\cos\theta, v\sin\theta, \omega)$.

HINTS

<>With a quadratic cost about the origin and a linear model, $Q_{xx} = Q + A^\mathsf{T}PA$, $Q_{uu} = R + B^\mathsf{T}PB$, $Q_{ux} = B^\mathsf{T}PA$, and the value update $V_{xx} = Q_{xx} - \mathbf{K}^\mathsf{T}Q_{uu}\mathbf{K}$ is the Joseph form of the Riccati step.</> <>Along the nominal, $f(x,u) = f(\bar{x},\bar{u}) + f_x\delta x + f_u\delta u$ exactly for a linear model, so the closed-loop rollout coincides with the linear prediction the backward pass assumed; the realized decrease equals $\Delta V(1)$.</> <>For the unicycle f_x contains $-v\Delta t \sin\theta$ and $v\Delta t\cos\theta$: the prediction is first-order in $\delta\theta$ and the rollout is not, so $\alpha = 1$ can overshoot and the pass must repeat — 79 times on Hitch's parking problem.</>

3 CONCEPTUAL •• **Predict the clearance**

PREDICT FIRST

In the CHOMP Smoother load the micro-example preset and set λ_obs = 4 (η stays above the bound). What clearance does the waypoint converge to?

- Zero — it ends in contact as at $\lambda = 1$
  At $\lambda = 1$ the obstacle push balances the smoothness pull exactly at D = 0. Quadrupling λ makes the push stronger at every D inside the margin, so the balance point moves outward.
- ✓ 0.3 m
  D = $(\lambda - 1)/(2(1+\lambda)) = 3/10$. *The waypoint sits at* $y = 4/5$, inside the margin ε = 0.5 (D* < ε), where the balance is 2y = λ(2 − 2y).
- 0.5 m — right at the edge of the margin
  At D = ε the obstacle gradient is zero, so nothing holds the waypoint there against the smoothness pull toward y = 0; the equilibrium must be strictly inside the margin for every finite λ.

Now set $\epsilon = 1.0$ and predict again before pressing run. With $c'(D) = -(\epsilon - D)/\epsilon$ and $D = y - \frac{1}{2}$ the balance is $2y = \lambda(1.5 - y)$, so $y^* = 1.5\lambda/(2+\lambda)$: at $\lambda = 4$, $y^* = 1.0$ and the clearance rises from 0.3 to 0.5 because the margin's push now reaches further out.

4 CONCEPTUAL •• **Horizon versus cost-to-go**

In the MPC Horizon widget, switch the terminal cost-to-go off. Find the longest N_h at which Rusty still fails to round the corridor into room E, and explain the failure with the Chapter 6 backward-Dijkstra cost-to-go: what does the quadratic pull believe about the wall? Switch the cost-to-go back on and verify that $N_h = 6$ now succeeds. Which parameter decided the outcome, and what did the horizon buy instead?

> **HINTS**
>
> <>A quadratic pull to the goal is the cost-to-go of a world without walls. From the corridor the goal is north-east through a wall; every horizon that ends before the door sees only that wall and a gradient pointing into it.</> <>The wave-front distance differs from the Euclidean one exactly where walls intervene; as a terminal cost it tells the short horizon that the door is "closer" to the goal than the wall is. The check records $N_h = 6, 10, 24$ all arriving in 125–139 ticks with it, and neither 10 nor 24 arriving without it.</> <>What grows with N_h is the solve time, 1.9 → 10.0 ms per tick in the check; the driven path barely changes once the terminal cost is right.</>

5 PRACTICAL •• **DDP's second-order terms by finite differences**

`ilqr.rs` adds $V_x \cdot f_{xx}$, $V_x \cdot f_{uu}$ and $V_x \cdot f_{ux}$ by central differences of the Jacobians when `ddp` is set. Verify the implementation against the analytic second derivatives of the unicycle, then compare iteration counts and wall-clock time of iLQR and DDP on Hitch's parking problem over 20 seeded starts. Report, honestly, how often the two reach different local optima — the chapter's single instance has DDP finding a cheaper one in fewer iterations, and one instance is not a trend.

6 PRACTICAL ••• **STOMP against CHOMP on the two-homotopy Workbench (stretch)**

Run `stomp` from 20 seeds and `chomp` from both initial paths on the Workbench scene of the smoother widget. Report the success rate (final clearance positive), the median final cost, and the seeds that never escape the straight-line class's collision minimum. STOMP's noise can jump a waypoint across the block where a gradient cannot; it can also wander. Include the seeds that fail.

19.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

§11.3.1 (the Hamiltonian, the minimum principle, Example 11.3.1 and the bang-bang recovery, shooting) and §11.3.2 (transcription, the three parameterizations, the two warnings) are the root of everything here; §12.5.3's remark that underactuated systems must parameterize u(t) is Chapter 21's hook. Bryson & Ho, Kirk and Pontryagin et al. are cited there as the classical texts. link

Zucker, M., Ratliff, N., Dragan, A. D., Pivtoraiko, M., Klingensmith, M., Dellin, C. M., Bagnell, J. A., and Srinivasa, S. S. (2013). CHOMP: Covariant Hamiltonian Optimization for Motion Planning. *International Journal of Robotics Research 32(9–10)*.

The journal version of CHOMP: the smoothness metric, the arc-length obstacle functional and its gradient with the velocity projection and curvature term, the C^1 obstacle cost used here, and the Hamiltonian Monte Carlo variant this chapter does not implement. The 2009 ICRA paper by Ratliff, Zucker, Bagnell and Srinivasa introduced the method. doi:10.1177/0278364913488805

Kalakrishnan, M., Chitta, S., Theodorou, E., Pastor, P., and Schaal, S. (2011). STOMP: Stochastic Trajectory Optimization for Motion Planning. *IEEE International Conference on Robotics and Automation*.

The gradient-free sibling: noise with covariance A^{-1}, exponentiated cost weights, and the smoothing matrix M. Implemented here with a seeded generator and an accept-if-not-worse guard. doi:10.1109/ICRA.2011.5980280

Schulman, J., Duan, Y., Ho, J., Lee, A., Awwal, I., Bradlow, H., Pan, J., Patil, S., Goldberg, K., and Abbeel, P. (2014). Motion Planning with Sequential Convex Optimization and Convex Collision Checking. *International Journal of Robotics Research 33(9)*.

TrajOpt: the ℓ_1 penalty, the trust-region sequential convex loop, signed-distance constraints linearized at the witness, and swept-volume collision checking. Our sco keeps the loop's shape with LM in place of the QP and says so. doi:10.1177/0278364914528132

Li, W. and Todorov, E. (2004). Iterative Linear Quadratic Regulator Design for Nonlinear Biological Movement Systems. *International Conference on Informatics in Control, Automation and Robotics (ICINCO)*.

iLQR: DDP without the second-order dynamics terms. Jacobson and Mayne's Differential Dynamic Programming (1970) is the ancestor with the terms kept; Tassa, Erez and Todorov (IROS 2012) supply the regularization and line search used here. link

Tassa, Y., Erez, T., and Todorov, E. (2012). Synthesis and Stabilization of Complex Behaviors through Online Trajectory Optimization. *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

The practical iLQR/DDP: Q_uu regularization, the expected-decrease line search $\Delta V(\alpha) = \alpha a + \alpha^2 b$, and the value update with cross terms — the exact forms backwardPass and forwardPass implement — run inside an MPC loop. link

Rawlings, J. B., Mayne, D. Q., and Diehl, M. M. (2017). Model Predictive Control: Theory, Computation, and Design (2nd ed.) *Nob Hill Publishing*.

The stability theory this chapter points at and does not prove: a terminal cost that is a control Lyapunov function makes the receding-horizon loop stable. Our cost-to-go terminal is the practical cousin. link

CHAPTER 20

Nonholonomic Systems I — Controllability

DYNAMICS, TRAJECTORIES, AND CONSTRAINTS · Advanced · 70 min

> *We know by experience, however, that this velocity constraint does not imply a constraint on configurations; the car can reach any position and orientation in the obstacle-free plane. In fact, the prevented sideways translation can be approximated by parallel-parking maneuvers.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), Chapter 12*

Hitch cannot slide sideways. Every reader also knows Hitch can end up half a metre to the left of where it started, parallel-parked, by wiggling. This chapter turns that folk knowledge into a theorem, and the theorem has a precise shape: **a velocity constraint is not a configuration constraint, and the Lie bracket is the exact bookkeeping of what wiggling buys.** Follow *drive* for a time ϵ, then *steer* for ϵ, then undo both in order. You do not return home. You are displaced by ϵ^2 in a direction neither input produces directly — the bracket $[g_1, g_2]$ — plus a remainder of order ϵ^3 that this chapter measures rather than waves away.

Stack brackets until they span the tangent space and Chow's theorem says the system reaches everything nearby; stop short and Frobenius says it is trapped on a lower-dimensional leaf forever. That is the first half. The second half asks the same question of second-order systems — Reach with its third motor dead, Rusty with mass — where drift complicates the bookkeeping and the symmetric product of Lewis and Murray halves the work.

Everything here is *analysis*. Nothing in this chapter produces a plan; the catalogue of steering methods that cash these verdicts in is Chapter 21. Three honesty items the prose keeps throughout: the Lie algebra rank condition is necessary only for analytic fields; Sussmann's condition is sufficient, not necessary;

and STLC for a second-order system is always a statement *at rest*.

AFTER THIS CHAPTER YOU CAN

- Write a robot's motion constraints as vector fields and a distribution, in both the positive (span) and negative (annihilated by a covector) forms, and say what makes a constraint nonholonomic
- Derive the Lie bracket as the ε^2 residue of the four-flow loop, compute it by hand for the unicycle and Hitch, and verify it numerically to the $O(\varepsilon^3)$ remainder
- Run the filtration $D_1 \subseteq D_2 \subseteq \cdots$ to its involutive closure, read Frobenius' theorem off it, and know when a rank at one point says nothing about another
- State the four controllability notions and their ladder, apply Chow's theorem (the LARC) and Sussmann's good-and-bad-bracket test, and name the system each counterexample lives on
- For a simple mechanical control system, compute covariant derivatives and symmetric products from Chapter 17's Christoffel symbols and run the Lewis–Murray tests at rest, with and without Pfaffian constraints
- Implement the nonholonomic module: VectorField, lieBracket, filtration, larc, sussmannStlc, symmetricProduct, lewisMurray, constrainedInputs — each pinned to a hand computation

ASSUMED

- Hitch's kinematic model and the Lot (Chapter 2); Reach's Jacobian (Chapter 4)
- Manifolds, charts and tangent spaces, and why this chapter computes in local coordinates on purpose (Chapter 5)
- The standard form $M\,\ddot{q} + C\,\dot{q} + g = u$, Christoffel symbols and Pfaffian constraints with the projection P (Chapter 17)
- Multivariable calculus: Jacobians, Taylor expansions to second order, rank of a matrix

20.1 The problem: sideways is forbidden, sideways is reachable

Tell Hitch "half a metre to the left, same heading". A holonomic ghost — the free-flying rectangle of Chapter 4 — slides there in a second. Hitch has two inputs, a speed and a steering rate, and no combination of them has a sideways component. Yet Hitch gets there, by spinning a little, driving a little, unspinning,

undriving, fourteen times over.

> Hitch is asked for 0.5 m of pure sideways displacement. The purple ghost slides there in one second. Hitch repeats the loop spin → drive → unspin → undrive with $\varepsilon = 0.19$ rad: each loop buys $\varepsilon \sin \varepsilon \approx \varepsilon^2 = 0.036$ m sideways, so fourteen loops are needed — and each leaks $\varepsilon(1 - \cos \varepsilon) \approx \varepsilon^3/2$ backward, the $O(\varepsilon^3)$ term the chapter keeps.

Two things in that picture are the whole chapter. The sideways gain per loop is *quadratic* in the wiggle amplitude, which is why parallel parking is slow and why a planner that only has the four-flow recipe would be a terrible planner. And the gain has a *direction* that neither input has — the Lie bracket of the two input fields, which is the one new mathematical object you need.

20.2 Building intuition

20.2.1 Wiggling has a direction

The loop below is Choset's equation (12.3): from x_0, follow g_1 for ϵ, then g_2 for ϵ, then $-g_1$ for ϵ, then $-g_2$ for ϵ. For the unicycle's drive and spin fields the four legs are exact — two arcs of length ϵ and two spins of angle ϵ — so the residue can be written in closed form and compared to the prediction $\epsilon^2[g_1, g_2]$.

> **INTERACTIVE · w20.1**
>
> **Lie Bracket Parallel-Parker**
>
> The net motion of drive–steer–undrive–unsteer is not zero: it is $\varepsilon^2[g_1, g_2] + O(\varepsilon^3)$, and for a car that direction is sideways.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch20-nonholonomic-i#w20.1

Three things to notice, each a theorem below.

There is no first-order term. Halve ϵ and the residue drops by four, not two: the blue dot in the inset slides down a line of slope two. At $\epsilon = 0.1$ the loop from heading $\pi/6$ moves Hitch by $[0.005424, -0.008396, 0]$ against the prediction $\epsilon^2[g_1, g_2] = [0.005, -0.00866, 0]$; the gap is 5.0×10^{-4}, the $O(\epsilon^3)$ remainder.

The direction is new. $[g_1, g_2] = [\sin x_3, -\cos x_3, 0]^\top$ is perpendicular to the heading — exactly the direction the no-slip constraint forbids. Two fields that cannot move you sideways produce a loop that does.

Brackets stack. Switch to the four-state car. Its inputs are speed and *steering rate*, and the first bracket $g_3 = [g_1, g_2]$ only buys heading change — a car that can set its steering angle but has not yet driven has turned nothing. Loop g_1 with g_3 and the residue is sideways: the second-degree bracket $g_4 = [g_1, g_3]$. Hitch needs two rungs of brackets for the same direction the unicycle needed one, and

that number — the degree at which the brackets fill the tangent space — is what the filtration counts.

20.2.2 Four words for four sets

"Controllable" is one word in linear systems theory and four in this chapter, because for a nonlinear system the *shape* of the reachable set matters: does it have interior, does it have interior without leaving a small ball, does it contain a neighbourhood of the start inside that ball?

> **INTERACTIVE · w20.2**
>
> **Reachable Set Grower**
>
> Accessible, STLA, STLC and controllable are four different sets — a curve, a patch without x inside, a blob, and a blob you must leave V to fill.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch20-nonholonomic-i#w20.2

With drive alone the reachable set is a curve — the integral curve of g_1 — and no controllability property holds. With drive and spin, controls in $[-1, 1]^2$, the sampled set fills a blob with the start in its interior: small-time locally controllable. Flip to the forward-only, left-only control set $U \in \mathcal{U}^+$ and the set keeps its interior but hugs the start's boundary — nothing is behind Hitch, because in small time the heading cannot swing past $\pi/2$. That system is *accessible*, *STLA* and (in the open plane) *controllable*, yet not STLC. The macro adds $[g_1, g_2]$ as if it were a third input, and sideways becomes first-order: the widget is pretending to be a system whose Lie algebra is spanned at degree one, which is what every steering method of Chapter 21 constructs by other means.

20.2.3 Rank is a property of the point

Choset's Example 12.1.7 is two innocent-looking fields on $\mathbb{R}^3$, $g_1 = [x_1 \cos x_3,\ x_2 \sin x_3,\ 0]^\top$ and $g_2 = \partial/\partial x_3$, whose bracket has determinant $x_1 x_2$ against them.

> **INTERACTIVE · w20.3**
>
> **Foliation Leaves**
>
> A rank test at one configuration settles controllability only there: the same two fields give a line, a half-plane or a full quadrant depending on where you start.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch20-nonholonomic-i#w20.3

Start anywhere off the coordinate planes and the system reaches a full three-dimensional neighbourhood. Start on the plane $x_1 = 0$ and g_1 loses its first component for good — the reachable set is a half-plane, and no sequence of

controls ever leaves it. Start on the x_3-axis and it is a line. Nine leaves in all. A rank test at one configuration settles controllability *there*; the trailer's jackknife in Chapter 23 is the same lesson in a less innocent system.

THE ONE-SENTENCE VERSION

Velocity constraints restrict the *directions* a system can move in at each instant; they restrict the *configurations* it can reach only if the constrained directions close under the Lie bracket. Count the dimension of the bracket closure at a point: if it equals the dimension of the state space the system is small-time locally accessible there (Chow); if the system is also symmetric it is small-time locally controllable; if the closure stops short, Frobenius confines the system to a leaf of that dimension.

M, $x \in M$, n	state manifold, state, its dimension M = Q for a kinematic system; M = TQ, x = (q, $\dot{q}$) for a second-order one
T_xM, TM, $\omega(x)$, $\phi_t^g(x)$	tangent space, tangent bundle, a covector (one-form), the flow of g for time t $\omega(x)\dot{x} = 0$ is the negative form of a constraint
$\mathcal{D} = \text{span}(\mathcal{G})$, $\mathcal{D}_k$, $\overline{\mathcal{D}}$, Lie($\mathcal{G}$)	distribution, k-th step of the filtration, involutive closure, Lie algebra generated Choset §12.1.2–12.1.3
$[g_1, g_2] = \frac{\partial g_2}{\partial x} g_1 - \frac{\partial g_1}{\partial x} g_2$	the Lie bracket (eq. 12.5) the sign under which the four-flow loop moves by $+\varepsilon^2[g_1, g_2]$
g_0, g_i, $u \in U \subset \mathbb{R}^m$	drift field, control fields, control vector and set $\dot{x} = g_0 + \Sigma\, g_i u_i$ (eq. 12.6)
$\mathcal{U}^{\pm}$, $\mathcal{U}^{+}$	control sets with the origin interior to the convex hull / merely spanning $\mathbb{R}^m$ $[-1, 1]^m$ and its vertices / $[0, 1]^m$
$R^V(x, \leq T)$	states reachable within time T without leaving the neighbourhood V the set the four notions are about
$\langle v_1, v_2 \rangle = v_1^{\mathsf{T}} M(q) v_2$, $\nabla_{Y_1} Y_2$, $\langle Y_1 : Y_2 \rangle$	kinetic-energy metric, covariant derivative, symmetric product Chapter 17's Christoffel symbols, Choset's convention (no M^{-1} inside Γ)
$Y_i(q)$, Sym($\mathcal{Y}$)	columns of M^{-1}(q)T(q); the symmetric closure input fields on Q, not on TQ
$P = I - M^{-1}A^{\mathsf{T}}(AM^{-1}A^{\mathsf{T}})^{-1}A$, $\tilde{Y}_i = PY_i$, $\tilde{\nabla}$	constraint projection, constrained inputs and connection Chapter 17's projectionP
$q = (x, y, \phi, \theta)$, L, γ; ψ, $\theta_1 = \theta - \psi$, d	Hitch: rear-axle position, steering angle, heading (Choset's ordering), wheelbase, steering limit; trailer heading, hitch angle, hitch length Chapter 2 integrates ψ; Choset writes θ_1

20.3 The mathematics

Following Choset, the state space is a vector space $M = \mathbb{R}^n$ throughout: every manifold in this book is locally diffeomorphic to $\mathbb{R}^n$, and the price — angles wrap, so a heading coordinate is only a chart — is one this chapter pays knowingly. The global structure returns in Chapter 5's `Manifold`, which is where the planners of Part III live; here we differentiate, and differentiation is local.

20.3.1 Vector fields, flows, distributions

A **vector field** is a smooth map $g : M \to TM$ with $g(x) \in T_xM$ — in coordinates, a column vector that depends on x. Its **flow** $\phi_t^g(x)$ solves $\dot{x} = g(x)$ from x for time t, and the **integral curve** through x is $\{\phi_t^g(x) : t \in \mathbb{R}\}$. A set $\mathcal{G}$ of fields generates the **distribution** $\mathcal{D} = \operatorname{span}(\mathcal{G})$, a linear subspace of T_xM at each x; it is **regular** if that subspace has the same dimension everywhere.

The unicycle's two fields, drive and spin, are the positive form of its distribution,

$$g_1(x) = \begin{bmatrix} \cos x_3 \\ \sin x_3 \\ 0 \end{bmatrix}, \qquad g_2(x) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \qquad \mathcal{D}(x) = \{u_1 g_1(x) + u_2 g_2(x)\},$$

and the no-slip constraint is its negative form: $\mathcal{D}(x) = \{\dot{x} : \omega(x)\dot{x} = 0\}$ with the covector $\omega(x) = [-\sin x_3,\ \cos x_3,\ 0]$ (Choset eq. 12.1). A velocity constraint $f(q, \dot{q}) = 0$ that cannot be integrated to a configuration constraint is **nonholonomic**; the Pfaffian ones, $a(q)\dot{q} = 0$, are the kind Chapter 17 handled with the projection P, and the whole question of this chapter is whether they integrate.

20.3.2 The Lie bracket from the four-flow loop

STATEMENT (CHOSET EQ. 12.3–12.5)

From x_0, follow g_1 for ϵ, g_2 for ϵ, $-g_1$ for ϵ, $-g_2$ for ϵ. Then

$$x(4\epsilon) = x_0 + \epsilon^2 \left[g_1, g_2\right](x_0) + O(\epsilon^3), \qquad \left[g_1, g_2\right] = \frac{\partial g_2}{\partial x} g_1 - \frac{\partial g_1}{\partial x} g_2,$$

with no $O(\epsilon)$ term. If the bracket is not in $\operatorname{span}(\mathcal{G})$ it is a new direction, followable approximately, at the price of $O(\epsilon^2)$ motion for $O(\epsilon)$ time. If $[g_1, g_2] = 0$ the fields commute.

DERIVATION **Expanding the four flows to second order**

$x(4\epsilon) - x_0 = \epsilon^2\,[g_1, g_2](x_0) + O(\epsilon^3)$

Step 1 — one flow. Taylor-expand $\phi_\epsilon^g(x)$ in ϵ: $\dot{x} = g(x)$ and $\ddot{x} = \frac{\partial g}{\partial x}\dot{x} = \frac{\partial g}{\partial x} g$, so $\phi_\epsilon^g(x) = x + \epsilon g(x) + \frac{\epsilon^2}{2}\frac{\partial g}{\partial x}(x)\, g(x) + O(\epsilon^3)$.
Step 2 — compose g_2 after g_1. Write $x_1 = \phi_\epsilon^{g_1}(x_0)$. Then $\phi_\epsilon^{g_2}(x_1) = x_1 + \epsilon g_2(x_1) + \frac{\epsilon^2}{2}\frac{\partial g_2}{\partial x} g_2 + O(\epsilon^3)$, and re-expanding $g_2(x_1)$ at x_0 gives $g_2(x_1) = g_2(x_0) + \epsilon\frac{\partial g_2}{\partial x} g_1(x_0) + O(\epsilon^2)$. The cross term $\epsilon^2\frac{\partial g_2}{\partial x} g_1$ appears; the self terms $\frac{\epsilon^2}{2}\frac{\partial g_i}{\partial x} g_i$ are carried along.
Step 3 — compose $-g_1$. The ϵg_1 term cancels the one from step 1. Re-expanding $-g_1$ at the current point produces $-\epsilon^2\frac{\partial g_1}{\partial x}(g_1 + g_2)$ from the first-order displacement $\epsilon(g_1 + g_2)$ accumulated so far, and the self term $+\frac{\epsilon^2}{2}\frac{\partial g_1}{\partial x} g_1$ from the flow of $-g_1$ itself (the sign of g enters twice). The two g_1-self contributions cancel: $\frac{\epsilon^2}{2} - \epsilon^2 + \frac{\epsilon^2}{2} = 0$. What survives from this leg is $-\epsilon^2\frac{\partial g_1}{\partial x} g_2$.
Step 4 — compose $-g_2$. Likewise the ϵg_2 term cancels, the g_2-self terms cancel, and the re-expansion of $-g_2$ at a point displaced by ϵg_2 (the g_1 displacement is gone) gives $-\epsilon^2\frac{\partial g_2}{\partial x} g_2 + \frac{\epsilon^2}{2}\frac{\partial g_2}{\partial x} g_2$, which together with step 2's $\frac{\epsilon^2}{2}\frac{\partial g_2}{\partial x} g_2$ sums to zero.
Step 5 — collect. Every first-order term and every self term is gone; the survivors are the two cross terms, $\epsilon^2\big(\frac{\partial g_2}{\partial x} g_1 - \frac{\partial g_1}{\partial x} g_2\big)$ — the bracket.
The unicycle in closed form. Drive by ϵ at heading x_3, spin by ϵ, drive back by ϵ at heading $x_3 + \epsilon$, spin back:

$$x(4\epsilon) - x_0 = \epsilon \begin{bmatrix} \cos x_3 - \cos(x_3 + \epsilon) \\ \sin x_3 - \sin(x_3 + \epsilon) \\ 0 \end{bmatrix} = \epsilon^2 \begin{bmatrix} \sin x_3 \\ -\cos x_3 \\ 0 \end{bmatrix} + \frac{\epsilon^3}{2} \begin{bmatrix} \cos x_3 \\ \sin x_3 \\ 0 \end{bmatrix} + O(\epsilon^4).$$

The ϵ^2 term is $[g_1, g_2]$; the $\epsilon^3/2$ term points *along the heading*, so the loop also drifts forward by half a cube — the leak the hook measures and the number w20.1 plots.
The linear case and the sign (Choset Problem 8). For $g_1 = Ax$, $g_2 = Bx$ equation (12.5) gives $[g_1, g_2] = BAx - ABx = (BA - AB)x$, the *negative* of the matrix commutator $AB - BA$ that the sister book's Lie-group chapter uses. The sign is immaterial for spans and determinants' zero sets, and material for the direction of the loop residue; the library keeps Choset's and pins it in a check.

The library computes the bracket by formula (12.5) from each field's Jacobian — analytic for every system of this chapter, a fourth-order central difference otherwise — and the check compares the finite-difference bracket with the hand formula to 10^{-9} at the micro-example's configuration. Two properties hold for every pair and triple of fields and are property-tested on seeded random polynomial fields: **skew-symmetry** $[g_1, g_2] = -[g_2, g_1]$ and the **Jacobi identity** $[g_1, [g_2, g_3]] + [g_3, [g_1, g_2]] + [g_2, [g_3, g_1]] = 0$.

20.3.3 Filtrations, the involutive closure, Frobenius

A **Lie product of degree** k is a bracket expression in which the original fields appear k times. The **Lie algebra** $\mathrm{Lie}(\mathcal{G})$ is the span of all Lie products of all degrees, built by Choset's recursion

$$\mathcal{G}_1 = \mathcal{G}, \qquad \mathcal{G}_{i+1} = \mathcal{G}_i \cup \{[g_j, g_k] : g_j \in \mathcal{G}_1,\ g_k \in \mathcal{G}_i\}, \qquad \mathcal{D}_i = \mathrm{span}(\mathcal{G}_i),$$

whose span sequence $\mathcal{D}_1 \subseteq \mathcal{D}_2 \subseteq \cdots$ is the **filtration** of $\mathcal{D}_1$. If the filtration is regular, each step either raises the dimension or stops, so it terminates at a finite k with $\mathcal{D}_k = \mathcal{D}_{k+1} = \cdots = \overline{\mathcal{D}}$, the **involutive closure**; $\mathcal{D}$ is **involutive** if $\mathcal{D} = \overline{\mathcal{D}}$. If all products vanish beyond some degree the algebra is **nilpotent**, which Chapter 21's chained forms exploit.

STATEMENT (FROBENIUS; CHOSET §12.1.3)

A regular m-dimensional distribution has an m-dimensional **integral manifold** through every point — the set reachable along fields in $\mathcal{D}$, with $\mathcal{D}(x)$ its tangent space — if and only if it is involutive. When the integral manifolds are smaller than M they are the **leaves** of a **foliation**, and a system confined to a leaf of dimension $k < n$ satisfies no controllability property at all.

DERIVATION Why involutive is exactly what integrates

```
\dim \overline{\mathcal{D}}(x) = \text{dimension of the leaf through } x
```

Step 1 — necessity. If the fields of $\mathcal{D}$ are tangent to a submanifold N, their flows stay in N, so the four-flow loop stays in N, so its residue $\epsilon^2[g_1, g_2] + O(\epsilon^3)$ is tangent to N in the limit: the bracket lies in $TN = \mathcal{D}$. Involutive is forced.

Step 2 — sufficiency, in one picture. Choose m independent fields of an involutive $\mathcal{D}$ near x. Because their brackets stay in $\mathcal{D}$, the fields can be re-combined (pointwise, by an invertible $m \times m$ matrix) into m *commuting*

fields, and commuting flows can be straightened into coordinate flows $\partial/\partial y_1, \dots, \partial/\partial y_m$ by the inverse function theorem. The leaf through x is then $\{y_{m+1} = \text{const}, \dots, y_n = \text{const}\}$.

Step 3 — regularity and termination. If every $\mathcal{D}_i$ is regular, $\dim \mathcal{D}_i$ is constant in x and non-decreasing in i, bounded by n; so it strictly increases or the recursion stops, after at most $n - \dim \mathcal{D}_1$ steps. Without regularity nothing bounds the degree a priori — the library runs to a stated maximum and reports a stall separately from a closure.

Step 4 — the planning consequence. A system whose closure has dimension $k < n$ at x never leaves a k-dimensional leaf: it is neither controllable nor accessible, and no amount of cleverness with the inputs changes that. The unicycle with drive only ($k = 1$) is the canonical case.

Example 12.1.7, by the numbers. $g_1 = [x_1 \cos x_3, x_2 \sin x_3, 0]^\top$, $g_2 = \partial_3$, $[g_1, g_2] = [x_1 \sin x_3, -x_2 \cos x_3, 0]^\top$ and $\det[g_1\ g_2\ [g_1, g_2]] = x_1 x_2$. The library's `larc` returns STLC at $(0.7, -1.1, 0.4)$, a two-dimensional leaf at $(0, 1.3, 0.4)$ — the ladder stays at $(2, 2, 2, 2, 2)$ through degree five — and a one-dimensional leaf on the x_3-axis. Nine leaves: a line, four half-planes, four quadrants (Choset Figure 12.9).

The Philip Hall basis. Skew-symmetry and Jacobi mean most products in $\mathcal{G}_i$ are redundant; the Hall basis chooses the smallest set at each degree. The library prunes only the trivial redundancies ($[g, g] = 0$, $[g_j, g_k] = -[g_k, g_j]$ at degree two) and products that vanish identically near the test point, and never prunes a product merely because it is dependent *at* the point — Example 12.1.7 on a coordinate plane is exactly a field that is zero at x with nonzero brackets. Exercise 6 asks for the Hall basis proper.

20.3.4 Control-affine systems and the four notions

Choset's class is the **control-affine** system

$$\dot{x} = g_0(x) + \sum_{i=1}^{m} g_i(x)\, u_i, \qquad u \in U \subset \mathbb{R}^m,$$

with **drift** g_0 and independent control fields g_i; **driftless** if $g_0 = 0$. Kinematic systems may be driftless; second-order ones never are, because the state carries the velocity and the velocity carries the configuration whether or not any input acts. Two classes of control set matter: $\mathcal{U}^{\pm}$, whose convex hull contains the origin in its interior (positive combinations reach every direction — the cube $[-1, 1]^m$, or just its vertices), and the larger $\mathcal{U}^{+}$, which merely spans (the non-negative cube). A driftless system with $U \in \mathcal{U}^{\pm}$ is **symmetric**: every motion can be run backward.

Let $R^V(x, \le T)$ be the states reachable within time T by trajectories that never leave the neighbourhood V of x. The system is, from x:

- **controllable** if every $x_{goal} \in M$ is in $R^M(x, \leq T)$ for some finite T;
- **accessible** if $R^M(x, \leq T)$ contains a full n-dimensional set for some T;
- **small-time locally accessible (STLA)** if $R^V(x, \leq T)$ contains a full n-dimensional set for *every* V and *every* $T > 0$;
- **small-time locally controllable (STLC)** if $R^V(x, \leq T)$ contains a *neighbourhood of* x for every V and T.

STLC is the one planners want: a system STLC everywhere can follow any curve in M arbitrarily closely, so it can thread any clutter a free-flying body can thread. The implications among the four, and the conditions on the ones that are only sometimes true, are Choset's Figure 12.12:

The linear double integrator is the warning. $\ddot{q} = u$ with $|u| \leq 1$ is "controllable" in the Kalman sense, yet it is STLC only from rest: at $\dot{q} \neq 0$ reaching a point behind you requires the velocity to change sign, and the velocity coordinate must leave any small V first. For second-order systems **STLC always means STLC at zero velocity**, and the linearizations of the nonholonomic systems of this chapter fail the Kalman rank test outright — their controllability is inherently nonlinear.

20.3.5 Chow's theorem and Sussmann's refinement

STATEMENT (CHOSET THEOREMS 12.3.1 AND 12.3.3)

Chow / LARC. The control-affine system is STLA from x if — and, for analytic fields, only if — it satisfies the **Lie algebra rank condition** $\mathrm{Lie}\{g_0, \ldots, g_m\}(x) = T_xM$, for any $U \in \mathcal{U}^+$. If the system is symmetric the LARC also gives STLC.
Sussmann. With drift, the system is STLC at x if (1) $g_0(x) = 0$, (2) $U \in \mathcal{U}^{\pm}$, (3) the LARC is satisfied by **good** brackets up to degree k, and (4) every **bad** bracket of degree $j \leq k$ is a linear combination of good brackets of degree less than j. A bracket is bad if g_0 appears an odd number of times and every g_i an even number (zero counts); good otherwise.

DERIVATION **Brackets as motions, and why bad ones are one-way**

\text{LARC} \Rightarrow \text{STLA};\quad \text{LARC} + \text{symmetry} \Rightarrow \text{STLC};\quad \text{neutralized bad brackets} \Rightarrow \text{STLC at equilibria}

Step 1 — every Lie product is a motion. A degree-k product is realized by nesting four-flow loops: $[g_1, [g_1, g_2]]$ is a loop whose second leg is itself a loop. The net motion is $O(\epsilon^k)$ in time $O(\epsilon)$ — slower by one power per degree, which is why parking into a space ϵ longer than the car costs $\Theta(1/\epsilon^2)$ reversals (Chapter 21).

Step 2 — full rank gives interior. If the products span T_xM, there are n independent motion directions available in arbitrarily small time inside arbitrarily small V, and small combinations of them sweep an open set: STLA. For analytic fields the converse holds because an analytic system's reachable set is determined by all derivatives at x, which is the Lie algebra.
Step 3 — symmetry gives both signs. Reverse every leg of a loop and the residue flips sign; for a symmetric system the reversed controls are available, so each bracket direction comes with both signs and the open set of step 2 can be centred on x: STLC.
Step 4 — bad brackets. Write the loop for a bracket with its control amplitudes u_i explicit. A control appearing an even number of times contributes u_i^2 to the net motion, so flipping u_i changes nothing: the direction can be followed one way only, like a drift. A bad bracket is a one-way direction. It is harmless if a *faster* motion (a good bracket of lower degree) can cancel it — the bad bracket is **neutralized** — and fatal otherwise.
Step 5 — the degree-one bad bracket. g_0 itself is bad (drift once, controls zero times) and there is nothing of lower degree to neutralize it, hence the hypothesis $g_0(x) = 0$: Sussmann's theorem speaks only at equilibria.
Step 6 — from local to global. On a connected M, STLC everywhere gives controllability: chain neighbourhoods along any curve. The converse fails — the forward-only unicycle is controllable in the open plane and STLC nowhere.
Choset's four counterexamples, each in the library. The one-way watch knob (accessible, controllable, STLA, not STLC). $g_0 = [x_2^2, 0]^\top$, $g_1 = \partial_2$: at the origin $[g_0, g_1] = 0$ and $[g_1, [g_0, g_1]] = [2, 0]^\top$ is bad — `sussmannStlc` reports the LARC satisfied at degree three and a neutralization residual of exactly 1 (nothing of lower degree points along x_1), hence STLA only; indeed x_1 never decreases. The double integrator: STLC at $(0.5, 0)$, undecided at $(0.5, 0.3)$ because $g_0 \neq 0$. The unicycle with $U \in \mathcal{U}^+$: STLA by Chow, STLC unavailable because symmetry is unavailable.

For Hitch the verdict is the one Choset promises in §12.5.6 and the micro-example below computes: the car is STLC at every configuration with $|\phi| < \pi/2$, whatever the steering limit γ. A car that can only turn left reaches every pose in the Lot that a normal car can — slowly.

20.3.6 Global controllability from recurrent drift

STATEMENT (CHOSET THEOREM 12.3.5, AFTER LIAN, WANG AND FU)

If the drift g_0 is **weakly positively Poisson stable** — every point is nonwandering: for every T and neighbourhood V of every x, $\phi_t^{g_0}(V) \cap V \neq \emptyset$ for some $t > T$ — and $U \in \mathcal{U}^{\pm}$, then the LARC implies controllability on M.

DERIVATION **Recurrence supplies the backward motions symmetry would have**

WPPS drift + LARC $\Rightarrow$ controllable

Step 1. The LARC gives forward local reachability: an open set of states is reached from any x in short time (STLA).
Step 2. To move *against* the drift, wait: recurrence returns the unforced flow arbitrarily close to where it started, so a neighbourhood of any state is reached again later without controls, and forward-only motions plus waiting generate backward displacements too.
Step 3. Chain neighbourhoods along any path, as in the connected-and-STLC argument.
The example. $g_0 = \frac{1}{2}[-x_2, x_1]^\top$ (circular orbits — WPPS), $g_1 = \partial_1$: $[g_0, g_1] = [0, -\frac{1}{2}]^\top$ and $\det[g_1\ [g_0, g_1]] = -\frac{1}{2}$ everywhere, so the system is controllable, though STLC only at the origin where g_0 vanishes. The library returns STLA by the LARC and $[0, -0.5]$ for the bracket. **The robotics case:** energy-conserving drift on a compact configuration space is WPPS by Poincaré recurrence — a frictionless Reach on the horizontal Workbench (Exercise 1), a satellite on $SO(3)$ with a single body-fixed torque.

20.3.7 Second-order systems: the symmetric product

A **simple mechanical control system** is Chapter 17's standard form without potential terms, $M(q)\ddot{q} + C(q, \dot{q})\dot{q} = T(q)u$, so that with $Y_i(q)$ the columns of $M^{-1}(q)T(q)$,

$$\ddot{q} = -M^{-1}(q)C(q, \dot{q})\dot{q} + \sum_{i=1}^{m} Y_i(q)\, u_i.$$

One can lift this to $M = T\boldsymbol{Q}$ with $x = (q, \dot{q})$, $g_0 = [\dot{q}; -M^{-1}C\dot{q}]$, $g_i = [0; Y_i]$ and bracket away — correct, and twice the dimension it needs to be. The acceleration $\ddot{q}$ is not intrinsic (a free particle in polar coordinates has $\ddot{r} \neq 0$); the intrinsic acceleration is $\nabla_{\dot{q}}\dot{q} = \ddot{q} + M^{-1}\dot{q}^\top \Gamma \dot{q} \in T_q\boldsymbol{Q}$, the coordinate acceleration with the centripetal term projected out, and the object that does the projecting is the **covariant derivative** of the kinetic-energy metric,

$$\nabla_{Y_1} Y_2 = \frac{\partial Y_2}{\partial q}\, Y_1 + M^{-1}(q)\, Y_1^{\mathsf{T}}\, \Gamma(q)\, Y_2, \qquad \langle Y_1 : Y_2 \rangle = \nabla_{Y_1} Y_2 + \nabla_{Y_2} Y_1.$$

Γ are Chapter 17's Christoffel symbols in Choset's convention (no M^{-1} folded in; the amber term is the velocity-product form $(Y_1^{\mathsf{T}}\Gamma Y_2)_i = \sum_{jk} \Gamma^i_{jk} Y_{1,j} Y_{2,k}$). The equations of motion become $\nabla_{\dot q}\dot q = \sum Y_i u_i$ — Newton's $a = M^{-1} f$ with the right a — and unforced motions are geodesics.

STATEMENT (LEWIS AND MURRAY; CHOSET THEOREMS 12.4.1–12.4.2)

From an equilibrium $(q, 0)$, with $\mathrm{Sym}(\mathcal{Y})$ the closure of $\mathcal{Y}$ under iterated symmetric products and a product **bad** if every Y_i appears an even number of times:

1. STLA iff $\mathrm{Sym}(\mathcal{Y})(q) = T_q\mathcal{Q}$; STLC if also every bad symmetric product is a combination of good ones of lower degree.
2. **STLCA** (configurations locally fill a neighbourhood, velocities ignored) iff $\mathrm{Lie}(\mathrm{Sym}(\mathcal{Y}))(q) = T_q\mathcal{Q}$; **STLCC** and **STLEC** (zero-velocity configurations fill a neighbourhood of q) under the same neutralization; equilibrium controllable if that holds at every q.

DERIVATION **Why the symmetric product halves the computation**

[g_i, g_j] = 0,\quad [g_0, g_i] = \begin{bmatrix}-Y_i\\ 0\end{bmatrix},\quad [g_i,[g_0, g_j]] = \begin{bmatrix}0\\ \langle Y_i : Y_j\rangle\end{bmatrix}\ \text{at rest}

Step 1 — the intrinsic acceleration. On $S^1 \subset \mathbb{R}^2$ a point mass at angle θ has $\ddot q = \ddot\theta[-\sin\theta, \cos\theta]^{\mathsf{T}} + \dot\theta^2[-\cos\theta, -\sin\theta]^{\mathsf{T}}$. Only the $\ddot\theta$ term is visible to an observer confined to the circle; the $\dot\theta^2$ term keeps the mass on it. Subtracting the centripetal term is what $M^{-1}\dot q^{\mathsf{T}}\Gamma\dot q$ does in general coordinates.

Step 2 — the connection in coordinates. For a metric $\langle v_1, v_2\rangle = v_1^{\mathsf{T}} M v_2$ the Levi-Civita connection is $\nabla_{Y_1} Y_2 = \frac{\partial Y_2}{\partial q} Y_1 + M^{-1} Y_1^{\mathsf{T}}\Gamma Y_2$, with the Christoffel symbols of Chapter 17's equation (10.9). Euclidean metric: $\Gamma = 0$ and $\nabla_{Y_1} Y_2$ is the directional derivative.

Step 3 — block-differentiate on $T\mathcal{Q}$. With $g_0 = [\dot q;\ -M^{-1}\dot q^{\mathsf{T}}\Gamma\dot q]$ and $g_i = [0;\ Y_i(q)]$, the Jacobian of g_i has only a lower-left block $\partial Y_i/\partial q$, acting on the upper half of its argument — which is zero for every g_j, so $[g_i, g_j] = 0$. For $[g_0, g_i]$ the upper-right block of $\partial g_0/\partial x$ (the identity) hits Y_i and gives the $-Y_i$ in the configuration slot; the velocity slot is linear in $\dot q$ and vanishes at rest. For $[g_i, [g_0, g_j]]$ the velocity-product's second

derivative in $\dot{q}$ survives at rest and is exactly $M^{-1}(Y_i^\mathsf{T}\Gamma Y_j + Y_j^\mathsf{T}\Gamma Y_i)$, while the two $\partial Y/\partial q$ terms assemble the rest of $\langle Y_i : Y_j\rangle$.

Step 4 — the reading. $\mathrm{Sym}(\mathcal{Y})(q)$ is the set of velocity directions reachable from rest; $\mathrm{Lie}(\mathrm{Sym}(\mathcal{Y}))(q)$ the set of configuration directions. Sussmann's good/bad bookkeeping carries over verbatim because the lift makes g_0 appear once in every symmetric product.

Step 5 — in the library. `liftToStateSpace` builds the six-dimensional fields for the planar body with thrusters, and a check confirms the three identities numerically to 10^{-12} against `symmetricProduct` on Q.

The planar body with thrusters (Choset Example 12.4.3) — and Reach's free third link. Unit mass and inertia, $\Gamma = 0$, $Y_1 = [\cos q_3, \sin q_3, 0]^\mathsf{T}$ (thrust through the centre of mass), $Y_2 = [-\sin q_3, \cos q_3, -d]^\mathsf{T}$ (the offset thruster). Then $\langle Y_1 : Y_2\rangle = [d\sin q_3, -d\cos q_3, 0]^\mathsf{T}$, rank 3: STLA from rest. The bad products are $\langle Y_1 : Y_1\rangle = 0$ and $\langle Y_2 : Y_2\rangle = 2\nabla_{Y_2}Y_2 = 2d\,Y_1$ — Choset prints dY_1; the factor is immaterial — neutralized by Y_1: STLC at rest, confirming the Lie-bracket computation of Example 12.3.4, where $\det[g_1 \cdots g_6] = d^4$. A single offset thruster gives $\mathrm{rank}(Y_2\ \langle Y_2 : Y_2\rangle\ \langle Y_2 : \langle Y_2 : Y_2\rangle\rangle) = 3$, STLA, with nothing to neutralize the bad products: Sussmann is silent, and the system is in fact controllable but not STLC (Choset Figure 12.14). Section "Putting it together" asks whether this body *is* Reach's third link when the third motor is dead; the honest answer is "kinematically, yes; dynamically, not quite".

The hopper (Example 12.4.4). $M = \mathrm{diag}(I, mq_3^2, m)$, $\Gamma^2_{23} = \Gamma^2_{32} = mq_3$, $\Gamma^3_{22} = -mq_3$, $Y_1 = [1/I, -1/(mq_3^2), 0]^\mathsf{T}$, $Y_2 = [0, 0, 1/m]^\mathsf{T}$. Angular momentum is conserved, so velocities fill at most two dimensions: $\langle Y_1 : Y_1\rangle = [0, 0, -2/(m^2q_3^3)]^\mathsf{T} \in \mathrm{span}(Y_2)$, the other products vanish, and Sym is two-dimensional (not STLA) while $[Y_1, Y_2] = [0, -2/(m^2q_3^3), 0]^\mathsf{T}$ completes Lie(Sym): STLEC and equilibrium controllable. The library reports exactly this, and that $\mathrm{Sym}(\mathcal{Y}) = \mathrm{span}(\mathcal{Y})$ — the hopper is *maximally reducible* to a kinematic system, Chapter 21's Theorem 12.4.5.

20.3.8 Nonholonomic constraints through the projection

STATEMENT (CHOSET §12.4.3)

For $M\ddot{q} + \dot{q}^\mathsf{T}\Gamma\dot{q} = Tu + A^\mathsf{T}\lambda$ with $A\dot{q} = 0$ (k Pfaffian constraints), eliminate λ with Chapter 17's $P = I - M^{-1}A^\mathsf{T}(AM^{-1}A^\mathsf{T})^{-1}A$ to get $\tilde{\nabla}_{\dot{q}}\dot{q} = \sum \tilde{Y}_i u_i$, $\tilde{Y}_i = PY_i$, with the **constrained connection** $\tilde{\nabla}_{\tilde{Y}_1}\tilde{Y}_2 = P\nabla_{\tilde{Y}_1}\tilde{Y}_2$. Theorems 12.4.1–12.4.2 apply unchanged, but velocities live in the $(n_Q - k)$-dimensional distribution $\mathcal{D} = \ker A$, so the most one can hope for is $\mathrm{Sym}(\tilde{\mathcal{Y}}) = \mathcal{D}$ and STLEC.

DERIVATION **The knife-edge: Rusty with mass, force and torque**

$\langle \tilde{Y}_1 : \tilde{Y}_2 \rangle = -\frac{\tan q_3}{I}\tilde{Y}_1, \quad \det_2 = \frac{\cos^2 q_3}{m^2 I^2}, \quad \det_3 = \frac{2}{m^2 I^4}$

Step 1 — the projection. $M = \mathrm{diag}(m, m, I)$, $A = [\sin q_3, -\cos q_3, 0]$ (no sideways velocity), so P has the upper block $\begin{bmatrix} \cos^2 q_3 & \sin q_3 \cos q_3 \\ \sin q_3 \cos q_3 & \sin^2 q_3 \end{bmatrix}$ and a 1 in the corner: it projects planar velocity onto the heading.
Step 2 — the inputs. A force along q_1 and a torque about q_3: $Y_1 = e_1/m$, $Y_2 = e_3/I$, hence $\tilde{Y}_1 = [\cos^2 q_3, \sin q_3 \cos q_3, 0]^\top/m$ — the forward force projected onto the heading — and $\tilde{Y}_2 = e_3/I$.
Step 3 — the connection. $\Gamma = 0$, so $\nabla_{\tilde{Y}_2}\tilde{Y}_1 = \frac{\partial \tilde{Y}_1}{\partial q_3}\cdot\frac{1}{I} = \frac{1}{mI}[-\sin 2q_3, \cos 2q_3, 0]^\top$ and $\nabla_{\tilde{Y}_1}\tilde{Y}_2 = 0$; projecting with P gives $\langle \tilde{Y}_1 : \tilde{Y}_2 \rangle = \frac{1}{mI}[-\sin q_3 \cos q_3, -\sin^2 q_3, 0]^\top = -\frac{\tan q_3}{I}\tilde{Y}_1$. Both self products vanish. So $\mathrm{Sym}(\tilde{\mathcal{Y}}) = \mathrm{span}(\tilde{\mathcal{Y}}) = \mathcal{D}$: the knife-edge is maximally reducible — it is "secretly kinematic", the unicycle with velocity inputs, which is why Chapter 21 can steer it with Chapter 20's kinematic tools.
Step 4 — configurations. $[\tilde{Y}_1, \tilde{Y}_2] = \frac{1}{mI}[\sin 2q_3, -\cos 2q_3, 0]^\top$ and $\det[\tilde{Y}_1\ \tilde{Y}_2\ [\tilde{Y}_1, \tilde{Y}_2]] = \cos^2 q_3/(m^2 I^2)$, which vanishes at $q_3 = \pm\pi/2$ — there the projected force $\tilde{Y}_1$ itself vanishes, and the distribution is not regular. One more bracket, $[\tilde{Y}_2, [\tilde{Y}_1, \tilde{Y}_2]] = \frac{2}{mI^2}[\cos 2q_3, \sin 2q_3, 0]^\top$, gives $\det[\tilde{Y}_2\ [\tilde{Y}_1, \tilde{Y}_2]\ [\tilde{Y}_2, [\tilde{Y}_1, \tilde{Y}_2]]] = 2/(m^2 I^4)$: STLCA and STLEC everywhere by Theorem 12.4.2.
In the library (with $m = 2$, $I = 0.5$ so the numbers are not all ones): every field above matches Choset's closed form to 10^{-9}, $\det_2 = 0.584984$ at $q_3 = 0.7$ against the formula's 0.584984, $\det_3 = 8.000000$; `larc` on the reduction $\{\tilde{Y}_1, \tilde{Y}_2\}$ at $q_3 = \pi/2$ returns rank 2 and "undecided" at degree two and STLC at degree three — the LARC refusing to certify the constrained system until it has the bracket that actually moves it.

20.3.9 The micro-example, by hand

Unicycle at $x_3 = \pi/6$. $g_1 = [\cos x_3, \sin x_3, 0]^\top = [0.8660, 0.5, 0]^\top$, $g_2 = e_3$. $\partial g_2/\partial x = 0$, and $\partial g_1/\partial x$ has one nonzero column, $\partial g_1/\partial x_3 = [-\sin x_3, \cos x_3, 0]^\top$, so

$$[g_1, g_2] = -\frac{\partial g_1}{\partial x} g_2 = \begin{bmatrix} \sin x_3 \\ -\cos x_3 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ -0.866025 \\ 0 \end{bmatrix}, \qquad \det[g_1\ g_2\ [g_1, g_2]] = \det \begin{bmatrix} \cos x_3 & 0 & \sin x_3 \\ \sin x_3 & 0 & -\cos x_3 \\ 0 & 1 & 0 \end{bmatrix} = 1.$$

The exact loop with $\epsilon = 0.1$ moves $[0.005424, -0.008396, 0]$ against $\epsilon^2[g_1,$

$g_2] = [0.005, -0.008660, 0]$; the gap, 5.0×10^{-4}, is $O(\epsilon^3)$ — half a cube is 5×10^{-4} exactly.

Hitch's car (Choset eq. 12.34), $q = (x, y, \phi, \theta)$, $L = 1$. $g_1 = [\cos\theta, \sin\theta, 0, \frac{1}{L}\tan\phi]^\top$, $g_2 = e_3$. Then $g_3 = [g_1, g_2] = -\partial g_1/\partial\phi = [0, 0, 0, -\sec^2\phi/L]^\top$, and $g_4 = [g_1, g_3] = -\frac{\partial g_1}{\partial\theta}\, g_{3,\theta} = \frac{\sec^2\phi}{L}[-\sin\theta, \cos\theta, 0, 0]^\top$, with $\det[g_1\ g_2\ g_3\ g_4] = -\sec^4\phi/L^2$. At $q = 0$ the columns are $e_1, e_3, -e_4, e_2$ and the determinant is -1; the filtration climbs $(2, 3, 4)$ and the car is STLC by symmetry. At a generic configuration with $L = 2.6$ the finite-difference g_3, g_4 agree with these formulas to 10^{-14}.

With the trailer, state $(x, y, \phi, \theta, \psi)$ and $\dot\psi = (v/d)\sin(\theta - \psi)$, the ladder is $(2, 3, 4, 5)$: one more degree for one more coordinate, STLC again. The library finds the same ladder at hitch angle $\theta - \psi = \pi/2$ — the trailer's Lie algebra is full rank everywhere, as Laumond proved; the jackknife of Chapter 23 is not a loss of controllability, and the chart that *does* go singular at $\pm\pi/2$ is Chapter 21's chained form.

20.4 The algorithm

Choset numbers none of these; the names match the Rust module.

ALGORITHM `lie_bracket(g1, g2, x)` COST $O(n^2)$ with analytic Jacobians; $O(n^2)$ field evaluations by differences

in two vector fields with Jacobians (analytic, or fourth-order central differences at $h = 10^{-3}$), a point x

out $g_1, g_2 = (\partial g_2/\partial x)\, g_1 - (\partial g_1/\partial x)\, g_2$

1: $J_1 \leftarrow \partial g_1/\partial x\,(x)$, $J_2 \leftarrow \partial g_2/\partial x\,(x)$ — analytic if the field has one, else $\frac{1}{12h}[f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)]$ per column
2: return $J_2\, g_1(x) - J_1\, g_2(x)$
3: as a *field*: the closure $x \mapsto$ step 2, whose own Jacobian falls back to differences of an exactly evaluated bracket — one difference level per nesting depth

ALGORITHM `filtration(G, x, max_deg, tol) -- Chow's test` COST `(m+1)^k` products at degree k before pruning; rank by modified Gram-Schmidt

in generators g_0 (if drift), $g_1 \ldots$ g_m; the test point; the highest degree; a relative rank tolerance

out dims = (dim $\mathcal{D}_1$, dim $\mathcal{D}_2$, …), every product with its generator counts and good/bad tag, a greedy basis, terminated / stalled flags

1: $\mathcal{G}_1 \leftarrow \{g_0, g_1, \ldots, g_m\}$ with counts e_i; dims $\leftarrow$ [rank$\{g(x) : g \in \mathcal{G}_1\}$]
2: for deg = 2, ..., max_deg: for each $g_j \in \mathcal{G}_1$ and each h new at the previous degree, form $[g_j, h]$ (at degree 2 only $j < k$); drop it if it vanishes at x *and* at four probe points nearby; record counts = counts$(h) + e_j$ and bad $\iff$ counts$_0$ odd and all other counts even
3: append rank of all values so far to dims; stop early if the rank is n or nothing new was formed (closure certain); otherwise continue — a stall is *not* a stop for a non-regular filtration
4: return dims, products, a greedy independent subset, and stalled = trailing degrees without growth

ALGORITHM `larc(sys, x, max_deg) / sussmann_stlc(sys, x, max_deg)` COST one filtration, plus one least-squares projection per bad bracket

in a control-affine system (drift or null, controls, control-set class), a point, a degree budget
out Verdict $\in$ { stlc-symmetric, stlc-sussmann, stla, confined(dim), undecided } with the filtration and, for Sussmann, every bad bracket's neutralization residual

1: $F \leftarrow$ filtration$(\{g_0, g_1, \ldots\}, x)$
2: **larc:** if rank $F = n$: driftless and $U \in \mathcal{U}^{\pm} \Rightarrow$ stlc-symmetric, else stla; if F terminated below n or stalled two degrees $\Rightarrow$ confined(rank); else undecided
3: **sussmann:** require $g_0(x) = 0$ and $U \in \mathcal{U}^{\pm}$, else undecided
4: $k \leftarrow$ the least degree at which the *good* products alone have rank n (none $\Rightarrow$ stla if the LARC holds)
5: for every bad product of degree $j \le k$: residual $\leftarrow$ relative least-squares residual against the good products of degree $< j$; neutralized iff residual $< 10^{-6}$
6: all neutralized $\Rightarrow$ stlc-sussmann at degree k; else stla with the offending brackets named

ALGORITHM `lewis_murray(∇, 𝒴, q, max_deg, lie_deg, target)` COST each symmetric product is one Jacobian-vector product and one Christoffel form, O(n_Q^3); then one Lie filtration of a basis of Sym

in a connection (kinetic, or constrained P∇), input fields Y$_1$ ... Y_m on Q, a configuration, degree budgets, the velocity dimension to fill (n_Q, or n_Q − k under constraints)
out MechVerdict $\in$ { stlc, stla, stlec, stlca, confined, undecided }, dim Sym, dim Lie(Sym), maximally-reducible flag, bad products with residuals

1: Sym $\leftarrow$ the recursion $S_1 = \mathcal{Y}$, $S_{i+1} = S_i \cup \{\langle Y_j : h\rangle : Y_j \in S_1,$ h new in $S_i\}$ with $j \le k$ at degree 2, counts added, bad iff all counts even; dims recorded at q
2: maximally reducible $\iff$ $\dim \mathrm{Sym}(q) = \dim \mathrm{span}(\mathcal{Y})(q)$ (Theorem 12.4.5, for Chapter 21)
3: neutralized $\iff$ every bad product's residual against lower-degree good ones is $< 10^{-6}$
4: if target $= n_Q$ and $\dim \mathrm{Sym} = n_Q$: neutralized $\Rightarrow$ stlc, else stla (Theorem 12.4.1)
5: else $L \leftarrow$ filtration of a basis of Sym (as fields) to lie_deg; $\dim L = n_Q$ and neutralized $\Rightarrow$ stlc (and STLCC); $\dim L = n_Q$ alone $\Rightarrow$ stlca; closed below $\Rightarrow$ confined; else undecided (Theorem 12.4.2)

20.5 Implementation in Rust

The `nonholonomic` crate is introduced here; Chapter 21 adds `steer/`, `lattice/`, `reduction/` and `car_grid.rs` to it. Brackets, filtrations and symmetric products are hand-rolled per the book's policy; `nalgebra` supplies the fixed-size vectors and the rank. The TypeScript port in `web/lib/nonholonomic/` is the code behind every widget on this page.

RUST `crates/nonholonomic/src/field.rs`

```rust
use nalgebra::{SMatrix, SVector};

/// A smooth vector field on a chart of an N-dimensional state manifold.
///
/// Brackets need Jacobians, so the trait carries one. The default is a
/// fourth-order central difference with a deliberately *large* step: a nested
/// bracket differentiates a bracket, and a difference of a difference loses
/// half its digits per level if h is tiny. Concrete systems override it with
/// the analytic Jacobian, which is what makes degree-4 products trustworthy.
pub trait VectorField<const N: usize> {
    fn eval(&self, x: &SVector<f64, N>) -> SVector<f64, N>;

    fn jacobian(&self, x: &SVector<f64, N>) -> SMatrix<f64, N, N> {
        central_difference(|y| self.eval(y), x, 1e-3)
    }
}

/// A borrowed or boxed field is still a field, so `Bracket(&g1, &g2)` needs no clones
/// and the filtration can bracket `Box<dyn VectorField<N>>` generators.
impl<const N: usize, F: VectorField<N> + ?Sized> VectorField<N> for &F {
    fn eval(&self, x: &SVector<f64, N>) -> SVector<f64, N> { (**self).eval(x) }
    fn jacobian(&self, x: &SVector<f64, N>) -> SMatrix<f64, N, N> { (**self).jacobian(x) }

}
impl<const N: usize, F: VectorField<N> + ?Sized> VectorField<N> for Box<F> {
    fn eval(&self, x: &SVector<f64, N>) -> SVector<f64, N> { (**self).eval(x) }
    fn jacobian(&self, x: &SVector<f64, N>) -> SMatrix<f64, N, N> { (**self).jacobian(x) }
```

```rust
}

/// ?f/?x by the 5-point stencil (f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)) / 12h, per column.
pub fn central_difference<const N: usize>(
    f: impl Fn(&SVector<f64, N>) -> SVector<f64, N>,
    x: &SVector<f64, N>,
    h: f64,
) -> SMatrix<f64, N, N> {
    let mut j = SMatrix::<f64, N, N>::zeros();
    for k in 0..N {
        let mut e = SVector::<f64, N>::zeros();
        e[k] = h;
        let col = (f(&(x - 2.0 * e)) - 8.0 * f(&(x - e)) + 8.0 * f(&(x + e)) - f(&(x + 2.0 * e))) / (12.0 * h);
        j.set_column(k, &col);
    }
    j
}

/// [g1, g2] = (?g2/?x) g1 - (?g1/?x) g2 -- Choset eq. (12.5), same sign convention,
/// so the loop g1 -> g2 -> -g1 -> -g2 moves by +epsilon2[g1, g2]. A `Bracket` is itself a
/// `VectorField`, so brackets nest: `Bracket(g1, Bracket(g1, g2))` is the car's g4.
pub struct Bracket<F, G>(pub F, pub G);

impl<const N: usize, F: VectorField<N>, G: VectorField<N>> VectorField<N> for Bracket<F, G> {
    fn eval(&self, x: &SVector<f64, N>) -> SVector<f64, N> {
        self.1.jacobian(x) * self.0.eval(x) - self.0.jacobian(x) * self.1.eval(x)
    }
}

/// Choset's unicycle (Example 12.1.3): drive and spin, with analytic Jacobians.
pub struct Drive;
pub struct Spin;

impl VectorField<3> for Drive {
    fn eval(&self, x: &SVector<f64, 3>) -> SVector<f64, 3> {
        SVector::<f64, 3>::new(x[2].cos(), x[2].sin(), 0.0)
    }
    fn jacobian(&self, x: &SVector<f64, 3>) -> SMatrix<f64, 3, 3> {
        let mut j = SMatrix::<f64, 3, 3>::zeros();
        j[(0, 2)] = -x[2].sin();
        j[(1, 2)] = x[2].cos();
        j
    }
}

impl VectorField<3> for Spin {
    fn eval(&self, _x: &SVector<f64, 3>) -> SVector<f64, 3> {
        SVector::<f64, 3>::new(0.0, 0.0, 1.0)
    }
    fn jacobian(&self, _x: &SVector<f64, 3>) -> SMatrix<f64, 3, 3> {
        SMatrix::<f64, 3, 3>::zeros()
    }
}

/// The four-flow loop of eq. (12.3) by RK4, returning x(4epsilon) - x_0 and the
/// prediction epsilon2[g1, g2](x_0); their difference is the O(epsilon3) remainder.
pub fn four_flow_loop<const N: usize>(
```

```
    g1: &impl VectorField<N>,
    g2: &impl VectorField<N>,
    x0: &SVector<f64, N>,
    eps: f64,
) -> (SVector<f64, N>, SVector<f64, N>) {
    let mut x = *x0;
    x = flow(g1, &x, eps);
    x = flow(g2, &x, eps);
    x = flow(g1, &x, -eps);
    x = flow(g2, &x, -eps);
    (x - x0, eps * eps * Bracket(g1, g2).eval(x0))
}
```

The filtration follows Chosetʼs recursion literally, with the generator counts that make Sussmannʼs tagging a counting exercise.

RUST **crates/nonholonomic/src/filtration.rs**

```
use nalgebra::SVector;

/// A Lie product with its provenance: which generators, how many times.
pub struct LieProduct<const N: usize> {
    pub field: Box<dyn VectorField<N>>,
    pub degree: usize,
    /// counts[i] = occurrences of generator i; index 0 is the drift when there is one.
    pub counts: Vec<usize>,
    pub value: SVector<f64, N>,
    /// Sussmann: bad iff the drift appears an odd number of times and every control an even number.
    pub bad: bool,
}

pub struct Filtration<const N: usize> {
    /// dim D_1, dim D_2, ... -- (2, 3, 4) for the car, (2, 3, 4, 5) with the trailer.
    pub dims: Vec<usize>,
    pub products: Vec<LieProduct<N>>,
    /// Full rank, or nothing left to bracket: the closure is certain.
    pub terminated: bool,
    /// Trailing degrees that added no dimension. A stall is reported, never used to stop:
    /// a non-regular filtration may pause and grow again (the knife-edge at q_3 = +/-pi/2).
    pub stalled: usize,
}

pub fn filtration<const N: usize>(
    fields: &[Box<dyn VectorField<N>>],
    x: &SVector<f64, N>,
    max_deg: usize,
    drift_index: Option<usize>,
    tol: f64,
) -> Filtration<N> {
    let m = fields.len();
    let mut products: Vec<LieProduct<N>> = (0..m)
        .map(|i| {
            let mut counts = vec![0; m];
            counts[i] = 1;
            LieProduct { field: fields[i].clone_box(), degree: 1, counts: counts.clone(),
```

```
                            value: fields[i].eval(x), bad: tag_bad(&counts, drift_index) }
        })
        .collect();
    let mut dims = vec![rank(products.iter().map(|p| p.value), tol)];
    let mut frontier: Vec<usize> = (0..m).collect();
    let mut terminated = false;

    for deg in 2..=max_deg {
        let mut fresh = Vec::new();
        for j in 0..m {
            for &h in &frontier {
                // Degree 2: [g_j, g_k] with j < k only -- [g, g] = 0 and antisymmetry.
                if deg == 2 && j >= products[h].counts.iter().position(|&c| c == 1).
   unwrap() { continue; }
                let g = Bracket(fields[j].clone_box(), products[h].field.clone_box());
                if vanishes_near(&g, x) { continue; }          // identically zero, not
   merely at x
                let mut counts = products[h].counts.clone();
                counts[j] += 1;
                let bad = tag_bad(&counts, drift_index);
                fresh.push(LieProduct { value: g.eval(x), field: Box::new(g), degree: deg,
    counts, bad });
            }
        }
        let start = products.len();
        products.extend(fresh);
        frontier = (start..products.len()).collect();
        dims.push(rank(products.iter().map(|p| p.value), tol));
        if *dims.last().unwrap() == N || frontier.is_empty() { terminated = true; break; }

    }
    let stalled = dims.windows(2).rev().take_while(|w| w[0] == w[1]).count();
    Filtration { dims, products, terminated, stalled }
}

pub enum Verdict { StlcSymmetric { degree: usize }, StlcSussmann { degree: usize,
   neutralized: Vec<String> },
                   Stla { degree: usize }, Confined { dim: usize }, Undecided(String) }

/// Chow / LARC (Theorem 12.3.1) with the symmetry shortcut.
pub fn larc<const N: usize>(sys: &ControlAffine<N>, x: &SVector<f64, N>, max_deg: usize)
   -> Verdict {
    let f = filtration(&sys.generators(), x, max_deg, sys.drift_index(), 1e-7);
    let rank = *f.dims.last().unwrap();
    let degree = f.dims.iter().position(|&d| d == N).map_or(0, |i| i + 1);
    match (rank == N, sys.drift.is_none() && sys.control_set == ControlSet::PlusMinus) {
        (true, true) => Verdict::StlcSymmetric { degree },
        (true, false) => Verdict::Stla { degree },
        (false, _) if f.terminated || f.stalled >= 2 => Verdict::Confined { dim: rank },
        _ => Verdict::Undecided(format!("rank {rank} < {N} after degree {max_deg}")),
    }
}

/// Sussmann (Theorem 12.3.3) at an equilibrium: good brackets span, bad ones are
/// neutralized by lower-degree good ones (relative least-squares residual < 1e-6).
pub fn sussmann_stlc<const N: usize>(sys: &ControlAffine<N>, x: &SVector<f64, N>, max_deg:
    usize) -> Verdict {
    let Some(g0) = &sys.drift else { return larc(sys, x, max_deg) };
    if g0.eval(x).norm() > 1e-8 { return Verdict::Undecided("g0(x) != 0: only equilibria".
   into()); }
    let f = filtration(&sys.generators(), x, max_deg, Some(0), 1e-7);
```

```
    let good_rank_at = |deg| rank(f.products.iter().filter(|p| !p.bad && p.degree <= deg).map(|p| p.value), 1e-7);
    let Some(k) = (1..=f.dims.len()).find(|&d| good_rank_at(d) == N) else {
        return Verdict::Stla { degree: f.dims.len() };
    };
    let mut neutralized = Vec::new();
    for p in f.products.iter().filter(|p| p.bad && p.degree <= k) {
        let lower: Vec<_> = f.products.iter().filter(|q| !q.bad && q.degree < p.degree).map(|q| q.value).collect();
        if relative_residual(&p.value, &lower) > 1e-6 { return Verdict::Stla { degree: k }; }
        neutralized.push(p.label());
    }
    Verdict::StlcSussmann { degree: k, neutralized }
}
```

The mechanical half imports Chapter 17's `Model` for M, Γ and the projection P, and differentiates only the input fields.

RUST `crates/nonholonomic/src/mechanical.rs`

```
use dynamics::{Model, Pfaffian};
use nalgebra::{SMatrix, SVector};

/// ?_{Y1} Y2 = (?Y2/?q) Y1 + M-1 Y1^T Gamma Y2, Choset's convention for Gamma (no M-1 inside).
pub fn covariant<const Q: usize>(
    y1: &dyn VectorField<Q>, y2: &dyn VectorField<Q>, model: &Model<Q>, q: &SVector<f64, Q>,
) -> SVector<f64, Q> {
    let gamma = model.christoffel(q);                       // Q slices of Q*Q
    let (v1, v2) = (y1.eval(q), y2.eval(q));
    let mut form = SVector::<f64, Q>::zeros();
    for i in 0..Q { form[i] = (v1.transpose() * gamma[i] * v2)[(0, 0)]; }
    y2.jacobian(q) * v1 + model.inertia(q).try_inverse().unwrap() * form
}

/// ?Y1 : Y2? = ?_{Y1}Y2 + ?_{Y2}Y1; with a Pfaffian constraint, P ? in place of ? (eq. 12.22).
pub fn symmetric_product<const Q: usize>(
    y1: &dyn VectorField<Q>, y2: &dyn VectorField<Q>, model: &Model<Q>,
    constraint: Option<&Pfaffian<Q>>, q: &SVector<f64, Q>,
) -> SVector<f64, Q> {
    let s = covariant(y1, y2, model, q) + covariant(y2, y1, model, q);
    match constraint {
        None => s,
        Some(c) => dynamics::projection_p(&model.inertia(q), &c.a(q)) * s,
    }
}

/// ?_i = P Y_i -- the constrained inputs of sec.12.4.3.
pub fn constrained_inputs<const Q: usize>(
    inputs: &[Box<dyn VectorField<Q>>], constraint: &Pfaffian<Q>, model: &Model<Q>,
) -> Vec<Box<dyn VectorField<Q>>> {
    inputs.iter().map(|y| Box::new(Projected { y: y.clone_box(), constraint: constraint.clone(), model: model.clone() })
                      as Box<dyn VectorField<Q>>).collect()
}
```

```rust

pub enum MechVerdict { Stlc, Stla, Stlec, Stlca, Confined { dim: usize }, Undecided }

/// Theorems 12.4.1-12.4.2 from rest. `target` is n_Q, or n_Q - k under k constraints.
pub fn lewis_murray<const Q: usize>(
    inputs: &[Box<dyn VectorField<Q>>], model: &Model<Q>, constraint: Option<&Pfaffian<Q>>,
    q: &SVector<f64, Q>, max_deg: usize, lie_deg: usize, target: usize,
) -> MechVerdict {
    let sym = sym_closure(inputs, model, constraint, q, max_deg);  // same recursion, ?:? for [,], j <= k at degree 2
    let neutralized = sym.products.iter().filter(|p| p.bad).all(|p| {
        let lower: Vec<_> = sym.products.iter().filter(|s| !s.bad && s.degree < p.degree).map(|s| s.value).collect();
        relative_residual(&p.value, &lower) < 1e-6
    });
    if target == Q && sym.rank == Q {
        return if neutralized { MechVerdict::Stlc } else { MechVerdict::Stla };
    }
    let lie = filtration(&sym.basis_fields(), q, lie_deg, None, 1e-7);
    match (*lie.dims.last().unwrap() == Q, neutralized, lie.terminated) {
        (true, true, _) => MechVerdict::Stlec,
        (true, false, _) => MechVerdict::Stlca,
        (false, _, true) => MechVerdict::Confined { dim: *lie.dims.last().unwrap() },
        _ => MechVerdict::Undecided,
    }
}
```

20.5.1 The worked example, printed

`cargo run --example chow_car -p nonholonomic` prints the micro-example and the lab's verdicts. Every line is reproduced by the TypeScript checks (`npm run check`, the `nonholonomic:` block) to the digits shown.

```
unicycle at x3 = pi/6:
  [g1, g2]              = [0.500000, -0.866025, 0.000000]      (hand formula: identical)
  loop residue, eps = 0.1 = [0.005424, -0.008396, 0.000000]
  eps^2 [g1, g2]          = [0.005000, -0.008660, 0.000000]      gap 5.00e-4 = O(eps^3)
  det[g1 g2 [g1,g2]]      = 1.000000
  log-log slope of |residue| over eps = 0.4 ... 0.025:  1.993  1.998  2.000  2.000

Hitch (eq. 12.34), L = 1, q = 0:
  columns  e1, e3, -e4, e2      det = -1.000000      dims = [2, 3, 4]      Verdict::
   StlcSymmetric
  generic q, L = 2.6:  |g3 - hand| = 0.0e+0   |g4 - hand| = 2.3e-15   det = -0.189977 = -
   sec^4(phi)/L^2
  with trailer (L = d = 1):  dims = [2, 3, 4, 5] at q = 0 and at hitch angle pi/2 --
   regular everywhere

Example 12.1.7:  interior StlcSymmetric;  half-plane Confined { dim: 2 } dims
   [2,2,2,2,2];  axis Confined { dim: 1 }

PBWT, d = 1.3, a_g = 0:  det[g1..g6] = 2.856100 = d^4;  at rest StlcSussmann, [g2,[g0,g2
   ]] = 2d g1 neutralized (1.4e-15)
  u1 only: rank 2 (Confined);  u2 only: rank 6 at degree 6, det of Choset's six columns =
    -130.5169 = -16 d^8, Stla
```

```
   symmetric product: <Y1:Y2> = [d sin q3, -d cos q3, 0];  <Y2:Y2> = 2d Y1;  Lewis-Murray
    Stlc

knife-edge, m = 2, I = 0.5:  <Y~1:Y~2> = -(tan q3 / I) Y~1;  Sym = span (maximally
   reducible)
  det2 = 0.584984 = cos^2(q3)/(m^2 I^2) at q3 = 0.7;  det3 = 8.000000 = 2/(m^2 I^4);
   Stlec
  reduction LARC at q3 = pi/2:  degree 2 rank 2 (undecided),  degree 3 StlcSymmetric dims
    [1,2,3]

Reach 3R, u3 = 0, a_g = 0, q = (0.3, 1.1, -0.5):  Sym dims 2 -> 3 (Stla);  bad <Y1:Y1>
   residual 0.105, <Y2:Y2> 0.048
  -> Lewis-Murray cannot certify STLC; STLKC does the planner's job in Chapter 21
```

20.6 Putting it together

The Integration lab drops the chapter's tests on the lab's three systems.

Hitch, with and without the trailer. `larc(carSystem(2.6), q)` at every configuration with $|\phi| < \gamma = 0.6$ returns STLC with the ladder $(2, 3, 4)$ — including $\phi = 0.58$ rad, hard against the steering stop, where Choset's remark that a car which can only turn one way reaches every pose becomes a number. With the trailer the ladder is $(2, 3, 4, 5)$ at hitch angle 0, at 0.8, and at $\pi/2$. **This is a deviation from the design brief**, which expected the ladder to stall at $\pm\pi/2$; the computation does not stall and the literature agrees (the one-trailer system is STLC everywhere). What *is* singular at $\theta_1 = \pm\pi/2$ is the trailer's velocity $v \cos\theta_1$ and with it the chained-form coordinate change and the flat-output lift of Chapter 21 — the jackknife is a planning and dynamics failure, not a controllability one, and Chapter 23 will say so.

Reach's 3R with the third motor dead, horizontal Workbench. Choset's §12.5.7 says the free third link "is equivalent to the PBWT in zero gravity, except the thruster forces at the third joint are generated by the actuators at the first two joints". Kinematically that is right, and Chapter 21 drives the link like a car along its two decoupling fields. Dynamically the arm is not the free body: its two input fields $Y_i = (M^{-1}[e_1\ e_2])_{\cdot i}$ carry the arm's inertia, and the Lewis–Murray test at $q = (0.3, 1.1, -0.5)$ returns $\dim \mathrm{Sym} = 3$ (STLA from rest) with the bad products $\langle Y_1 : Y_1 \rangle$ and $\langle Y_2 : Y_2 \rangle$ *outside* $\mathrm{span}(\mathcal{Y})$ — relative residuals 0.105 and 0.048. Sussmann's condition is sufficient, not necessary, so the arm may well be STLC; the test cannot say, and the chapter does not claim it. The design brief's expectation of a printed `Stlc` is therefore not met, and the honest line in its place is: *STLA, with STLKC — hence STLEC — supplied by the kinematic reduction of the next chapter*, which is exactly the property the switch-minimizing planner needs.

Rusty with mass, the knife-edge: maximally reducible, so everything Part III did with Rusty's kinematic model was legitimate for the dynamic one too, and STLEC everywhere once the degree-three bracket is admitted.

Which leaves the question this chapter cannot answer and the next one is about: *controllable — with what controls?* The four-flow loop is a proof, not a plan.

Chapter 21 spends the controllability established here six different ways.

20.7 Exercises

1 FOUNDATION •• **Reach's 2R with one motor, on the horizontal Workbench**

Write Reach's 2R dynamics (Chapter 17, $a_g = 0$) in the control-affine form on $T(T^2)$ with $u_2 = 0$: $g_0 = [\dot{q};\ -M^{-1}C\dot{q}]$, $g_1 = [0;\ M^{-1}e_1]$. Argue that g_0 is WPPS from energy conservation on the compact T^2, attempt the LARC for Theorem 12.3.5 (the library's `liftToStateSpace` and `larc` will do the brackets), and explain why the arm is *not* controllable with $u_2 \in \mathbb{R}$ but $u_1 = 0$.

> **HINTS**
>
> <>Energy-conserving drift on a compact space is WPPS by Poincaré recurrence (Choset, after Theorem 12.3.5); the arm's state space $T(T^2)$ is not compact, but the energy level sets are.</> <>With $u_1 = 0$ the shoulder is free: the total angular momentum about the shoulder is conserved, a velocity constraint that integrates the acceleration constraint away — the arm lives on a 3-dimensional leaf of the 4-dimensional state space. Compare the hopper.</> <>With $u_2 = 0$ no such conservation law exists (the elbow torque is internal), and $[g_0, g_1]$, $[g_1, [g_0, g_1]]$, $[g_0, [g_0, g_1]]$ generically complete the rank: the LARC holds, and Theorem 12.3.5 gives controllability.</>

2 FOUNDATION •• **The car's ladder, symbolically, and the trailer's extra rung**

Prove Choset's Problem 13 — the car (12.34) is STLC — by exhibiting the ladder $(2, 3, 4)$: compute $g_3 = [g_1, g_2]$ and $g_4 = [g_1, g_3]$ by hand and show $\det[g_1\ g_2\ g_3\ g_4] = -\sec^4\phi/L^2$. Then add the trailer with $\dot{\theta}_1 = \frac{u_1}{L}\tan\phi - \frac{u_1}{d_1}\sin\theta_1$ for the hitch angle $\theta_1 = \theta - \psi$, show one more degree is needed, and locate the configurations where $g_5 = [g_1, g_4]$ loses its θ_1-component.

WORK IT OUT

$\det[g_1\ g_2\ g_3\ g_4]$ for L = 1 at $\phi = \pi/4$

Answer. -4 $\sec^2(\pi/4) = 2$, so $-\sec^4\phi/L^2 = -4$. At $\phi = 0$ the fifth field is $[g_1, g_4] = e_5/(Ld_1^2)$ (in the chart with the trailer's absolute heading): pure trailer motion, and the ladder reads $(2, 3, 4, 5)$. Nowhere does it stall — the one-trailer system is STLC at every configuration — but the θ_1-component of g_4, $(\sec^2\phi/L)\cos\theta_1/d_1$, vanishes at $\theta_1 = \pm\pi/2$: the hitch angle stops responding to steering at first order there, which is the first symptom of the jackknife.

3 CONCEPTUAL •• **Predict the residue, then its error**

PREDICT FIRST

In the Parallel-Parker at $\varepsilon = 0.2$ with the unicycle pair, from heading $\pi/6$, the sideways (second) component of the exact residue is compared with $\varepsilon^2[g_1, g_2]_2 = -0.0346$. Which is it?

- Exactly -0.0346: the prediction is exact for the unicycle
  The ε^2 term is only the leading term. The closed form $\varepsilon[\cos x_3 - \cos(x_3 + \varepsilon), \sin x_3 - \sin(x_3 + \varepsilon), 0]$ has an $\varepsilon^3/2$ term along the heading, which has a nonzero second component at $x_3 = \pi/6$.
- ✓ About -0.0324: the loop under-delivers sideways by $\approx +0.0022$
  The exact residue is $[0.02332, -0.03242, 0]$; the $O(\varepsilon^3)$ term is $(\varepsilon^3/2)[\cos x_3, \sin x_3, 0] = 0.004 \times [0.866, 0.5, 0]$, whose second component $+0.002$ pushes the residue toward zero. Finite differences of the loop confirm the sign.
- About -0.0368: the loop over-delivers
  That would need the ε^3 correction to point against the bracket. It points along the heading, g_1, and at $x_3 = \pi/6$ the heading has a positive y-component — the correction is +, the bracket component is −.

Now switch to the car pair (g_1, g_3). The residue is $\epsilon^2 g_4$, sideways — but the *original* inputs are speed and steering rate, and g_3 is itself a bracket. Realized with speed and steering-rate loops, the sideways motion needs a loop whose one leg is a loop: $O(\epsilon^3)$ in the original inputs for $O(\epsilon)$ time. That extra power is the price of putting the steering angle in the state, and the reason the four-state car's filtration is one rung taller than the reduced car's.

4 CONCEPTUAL •• **How far behind, how soon**

In the Reachable Set Grower switch to the forward-only control set $\mathcal{U}^+$ with both inputs. Find the smallest horizon T for which a sampled state lies *behind* the start ($x < 0$), and relate it to the turning radius ($\rho = 1$ at unit speed and unit turn rate) and to the "controllable, not STLC" rung of the ladder.

HINTS

<>With $v \le 1$ and $0 \le \omega \le 1$ the heading can grow at most at rate 1, so $\dot{x} = v\cos\theta \ge 0$ for all $t \le \pi/2$: nothing is behind the start before $T = \pi/2$ whatever the controls.</> <>To actually get behind, the trajectory must also stay inside V — a half-turn of radius 1 needs a ball of radius about 1. Below that radius no T suffices: the property fails for *small* neighbourhoods, which is the definition of "not STLC".</> <>In the open plane with $V = M$ a full loop returns Hitch to any point behind: controllable. The ladder's dashed arrow STLA → STLC is what the forward-only set breaks; the solid STLA → accessible and controllable → accessible both still hold.</>

5 PRACTICAL •• **Reach's free third link as VectorField<6>**

Implement `impl VectorField<6>` for the planar body with thrusters of mass m and inertia I (Choset Problems 14–16) with the analytic Jacobians, build Choset's $g_3 \dots g_6$ with `Bracket`, and reproduce $\det[g_1 \cdots g_6] = d^4$ for $m = I = 1$ in a test. Then show that `sussmann_stlc` returns `StlcSussmann` at rest only when $a_g = 0$.

HINTS

<>Only g_1, g_2 depend on the state, through x_3; their Jacobians have two nonzero entries each. g_0's Jacobian is the constant shift matrix with the identity in the upper-right block.</> <>With $a_g \ne 0$, $g_0(x) = [0,0,0,0,-a_g,0]^\top \ne 0$ at every state: the degree-one bad bracket cannot be neutralized and the theorem does not apply. The TypeScript `pbwt(d, 9.81)` check returns `undecided` for exactly this reason.</> <>For general m, I the thrust fields scale by $1/m$ and the torque entry by $1/I$; the determinant becomes $d^4/(m^4 I^2)$ — check it against the library's `pbwt(d, 0, m, I)`.</>

6 PRACTICAL • • • **A Philip Hall basis, and a symbolic determinant (stretch)**

Replace Choset's recursion in `filtration` by a Philip Hall basis so that only the independent products of each degree are formed; benchmark bracket evaluations for the car-with-trailer to degree five against the current implementation. Then add a small symbolic backend (polynomials in cos, sin, tan, sec of the state) so that the car's $\det[g_1\ g_2\ g_3\ g_4] = -\sec^4\phi/L^2$ is produced as an expression rather than a number.

> **HINTS**
>
> <>Hall words: generators are Hall elements; $[u, v]$ is a Hall element iff u, v are, $u > v$ in the ordering, and if $u = [u', u'']$ then $u'' \le v$. The count at degree k is Witt's formula $\frac{1}{k}\sum_{e|k}\mu(e)m^{k/e}$: for $m = 2$ that is $2, 1, 2, 3, 6$ at degrees $1 \ldots 5$, against $2, 1, 2, 4, 8$ for the current recursion.</> <>A symbolic field is a vector of expressions; the bracket needs symbolic partial derivatives of cos, sin, tan only — the chain rule on four function kinds.</>

20.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

Chapter 12, §12.1–12.4: the vector-field preliminaries, the four-flow derivation (eq. 12.3–12.5), Examples 12.1.1–12.1.8 and 12.4.3–12.4.4, 12.4.9, Theorems 12.3.1, 12.3.3, 12.3.5, 12.4.1–12.4.2, and Problems 1–29, which this chapter follows section by section. Chow's 1939 paper and the Lian–Wang–Fu recurrence theorem are cited there as refs. 112 and 289. `link`

Sussmann, H. J. (1987). A General Theorem on Local Controllability. *SIAM Journal on Control and Optimization 25(1)*.

The good-and-bad-bracket sufficient condition for small-time local controllability at an equilibrium, stated here as Theorem 12.3.3 and implemented as sussmannStlc with explicit neutralization residuals. `doi:10.1137/0325011`

Lewis, A. D. and Murray, R. M. (1997). Configuration Controllability of Simple Mechanical Control Systems. *SIAM Journal on Control and Optimization 35(3)*.

The symmetric product, the symmetric closure and the tests for STLA, STLC, STLCA, STLCC and equilibrium controllability from rest (Theorems 12.4.1–12.4.2), implemented as lewisMurray. `doi:10.1137/S0363012995287155`

Murray, R. M., Li, Z., and Sastry, S. S. (1994). A Mathematical Introduction to Robotic Manipulation. *CRC Press*.

Chapter 7's treatment of nonholonomic systems — distributions, Frobenius, Chow, the Philip Hall basis and chained forms — is the standard long-form companion to Choset's §12.1–12.3 and the source of Exercise 6. link

Bullo, F. and Lewis, A. D. (2005). Geometric Control of Mechanical Systems: Modeling, Analysis, and Design for Simple Mechanical Control Systems. *Springer, Texts in Applied Mathematics 49*.

The affine-connection formulation of mechanical control systems in full: covariant derivatives, the symmetric product, constrained connections, and the kinematic reductions Chapter 21 uses. doi:10.1007/978-1-4899-7276-7

Solà, J., Deray, J., and Atchuthan, D. (2018). A micro Lie theory for state estimation in robotics. *arXiv:1812.01537*.

The bridge from this chapter's vector-field brackets to the sister book's matrix Lie groups: the Lie bracket of left-invariant fields on SE(2) is the matrix commutator (up to Choset's sign), which is why the four-flow residue of Hitch's drive and spin is the se(2) bracket of their generators. link

CHAPTER 21

Nonholonomic Systems II — Steering Cars, Trailers, and Underactuated Systems

DYNAMICS, TRAJECTORIES, AND CONSTRAINTS · Advanced · 85 min

> *This implies that it is possible to parallel-park your car into any parking space $\epsilon > 0$ longer than your car.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), Chapter 12*

Chapter 20 proved that Hitch can reach every pose in the Lot. It did not say *how*, and the proof's own recipe — nested four-flow loops — is a terrible plan: each loop of amplitude ϵ buys ϵ^2 of sideways motion, so a small sideways correction costs a great many reversals. This chapter is the catalog of better answers, and its one idea is that **every steering method is a different way of spending the controllability you already own**. You can solve an optimal-control problem exactly (Dubins, Reeds–Shepp). You can change coordinates until the brackets become a ladder you climb with sinusoids (chained forms). You can push on the Jacobian of the end-state map until the error vanishes (gradient steering). You can find two coordinates from which the whole state is read off by differentiation (flatness). Or you can let a search integrate the dynamics forward and never invert anything at all — Barraquand and Latombe's CAR GRID SEARCH, and the kinodynamic trees that descend from it.

None of these methods is "the" way to park a car, and the chapter's widgets are built to make that visible: one query, six fares. Exact methods ignore obstacles; search methods stop in a goal *region*; flat polynomials ignore the steering limit; sinusoids ignore everything except the chained coordinates and swing the trailer

past jackknife. The second half of the chapter is about how today's autonomous vehicles recombine these pieces — state lattices and hybrid A* are Algorithm 22 with lattice-consistent primitives, one continuous state per cell, Reeds–Shepp shots and a dual heuristic; Kinodynamic RRT* is RRT* with an exact optimal-control steer. It ends with Reach's third motor switched off and the dead link driven like a car.

Honesty items kept throughout: Algorithm 22 is resolution complete, not exact; Reeds–Shepp and Algorithm 22 paths have curvature jumps a real steering actuator cannot follow; flatness handles neither control bounds nor obstacles without numerical optimization; forward propagation is metric-sensitive and offers no optimality without an exact steer; the decoupled planner stops at every switch and so forfeits global time-optimality; and the $\Theta(1/\epsilon^2)$ cusp count for tight parking is a theorem, not a bug.

AFTER THIS CHAPTER YOU CAN

- Steer the driftless systems of Chapter 20 exactly in the absence of obstacles: Brockett's energy-optimal sinusoids, chained forms climbed by the $(a/4\pi)^k b/k!$ ladder, and the coordinate changes that put the unicycle and Hitch's car into chained form
- Derive the LSL closed form of a Dubins path, generate the remaining words by reflection and reversal, and implement the Reeds–Shepp family with the time-flip and reflect symmetries — every word checked by integration
- Plan by differential flatness: read the state and controls of the car, and of the car with a trailer, off a polynomial in the flat outputs, and say exactly where the lift is singular and where control bounds break it
- Implement Choset's Algorithm 22 CAR GRID SEARCH with the bucketed OPEN list, and explain state lattices and hybrid A* as Algorithm 22 plus three named deltas
- Build the forward-propagation RRT of §7.5.1 on Hitch's dynamic model, explain its metric sensitivity, and contrast it with Kinodynamic RRT*, which needs an exact optimal steer
- Test decoupling vector fields with Theorem 12.4.7, plan Reach's 3R arm with an unactuated third joint as a car with four primitives, and time-scale the plan with Chapter 18 so that one actuator is always saturated

ASSUMED

- Chapter 20: vector fields, Lie brackets, the LARC and STLC verdicts for the unicycle, the car and the car with trailer; the symmetric product and

the kinetic-energy connection
- Chapter 11 (the Steer interface, samplers, kd-tree), Chapter 12 (RRT and its Voronoi bias), Chapter 13 (RRT *rewiring), Chapter 6 (A* and its heap)
- Chapter 17 (the standard form, PlanarArm) and Chapter 18 (path dynamics and time-optimal time scaling); Chapter 19 (Gauss–Newton and Levenberg–Marquardt)
- Hitch and the Lot from Chapter 2; SE(2) from Chapter 5

21.1 The problem: same destination, six fares

Hitch is at the south curb of the Lot, heading east. Bay 3 is up and to the left. Three copies of Hitch leave at once.

One query, three planners from this chapter, replayed side by side (seed 21). Dotted: the Reeds–Shepp shortest path — four crisp segments, one cusp, exact, blind to obstacles. Solid: a quintic in the flat output — smooth, no cusp, a different length, blind to the steering limit. Dashed: a forward-propagated tree on Hitch's dynamic model — no steering function at all, collision-aware, ending somewhere inside a goal region. The scoreboard is computed live by planPark.

The three paths are all correct, and all different, because each one answers a different question. The Reeds–Shepp path is the *shortest* path for a car that may reverse; it has a cusp because stopping and backing up is free in its cost. The flat-output path is the *simplest polynomial* consistent with the two poses; it never stops, but nothing in its derivation stops it from asking for more steering than Hitch has. The tree path is the first thing a randomized search found by *simulating* Hitch; it is jittery and does not end exactly on the goal, but it is the only one of the three that looked at the pillars while planning. Chapter 20's controllability theorem guaranteed all three exist. It said nothing about which one you want, and the rest of this chapter is about making that choice deliberately.

21.2 Building intuition

21.2.1 One query, several optimality criteria

Start with the flagship. Hitch now pulls its trailer, and the scoreboard grows a column the planners were never asked about: how far the trailer swings.

INTERACTIVE · w21.1

Trailer Parker

There is no single 'path for a car': each steering method optimizes something different, and the trailer's swing is a cost none of them was asked about.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch21-nonholonomic-ii#w21.1

Three observations, each a section below.

The trailer is a hidden cost. With the trailer on, the chained-form sinusoids reach the bay exactly — in the car's chained coordinates — while the trailer, simply towed along, swings far past $|\theta_1| = \pi/2$. The sinusoid method steers the four-state car; it does not know the trailer exists. The flat-output method with the trailer is different in kind: its flat output *is* the trailer's axle, so the trailer is parked by construction and the car's motion is derived from it.

Exactness and obstacle-awareness trade off. The Reeds–Shepp word and the sinusoids land on the goal pose to round-off and do not look at the pillars. Turn "obstacles" on and the third method becomes Algorithm 22, which looks at everything and stops anywhere within ±0.5 m and ±0.2 rad of the goal. Exact steering is an obstacle-free subroutine; search is how obstacles enter.

Control bounds are where each method is weakest. The flat planner's red $\max|\phi|$ is the curvature its polynomial asks for, not what Hitch can do. The sinusoids ask for 1.22 rad of steering on the default query — twice Hitch's limit. Only the methods built from full-lock arcs (Reeds–Shepp, Algorithm 22) respect $|\phi| \le 0.6$ by construction.

21.2.2 Shortest paths have words

The shortest path between two poses of a car with a minimum turning radius ρ is a short *word* in three letters: L and R for full-lock arcs, S for straight. Forward-only (Dubins), six words suffice; allowed to reverse (Reeds–Shepp), a handful of families with cusps join them.

INTERACTIVE · w21.4

Reeds–Shepp Atlas

Reversing is not a Dubins word run backward: allowing cusps changes which word is shortest almost everywhere near the start, and the shortest length still varies continuously across the word boundaries.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch21-nonholonomic-ii#w21.4

The tint is the *synthesis*: for the goal heading on the slider, every point of the plane is colored by the family of its shortest path. Drag the goal along a

boundary and the word flips — but the length in the readout moves continuously, because the optimal length is a continuous function of the goal even though the optimal word is not. Toggle "Dubins only" and compare: near the start, behind it and beside it, the forward-only car pays enormously (a loop of nearly $2\pi\rho$) for what the Reeds–Shepp car does with one cusp. The Reeds–Shepp length is never longer — the chapter's check confirms it on 400 random pairs — and reversing is *not* a Dubins word run backward: the reversing words are new words.

21.2.3 State from a curve

Here is a planner with no differential equation in it.

> INTERACTIVE · w21.2
>
> **Flatness Follower**
>
> Planning a car does not need its equations integrated: pick a curve for the flat output and the heading, speed, turn rate and steering angle are read off its derivatives.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch21-nonholonomic-ii#w21.2

Drag the curve. Hitch is placed on it by differentiation alone: its heading is the curve's tangent, its steering angle is arctan of the wheelbase times the curve's curvature. Nothing was integrated. That is *differential flatness*: the car has two outputs — the rear-axle midpoint — from which state and inputs are algebraic functions of finitely many derivatives. Turn on the trailer and the flat output moves to the trailer's axle; now the *car's* position is a derivative away (the trailer tangent tells you where the hitch is) and its steering is two more. Make a corner tight and part of the curve turns red: the curve is still a perfectly good flat output, but it asks for more curvature than $1/\rho$, and flatness alone has no way to say no.

21.2.4 A tree without a steering function

Every planner so far needed to *invert* the car — find controls that go from here to there. Choset's §7.5.1 points out that a tree planner needs much less: a way to propagate a state forward under a control.

> INTERACTIVE · w21.5
>
> **Forward-Propagation Tree**
>
> An RRT does not need a steering function: a simulator and a metric are enough for completeness — but optimality (RRT*) needs the exact steer.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch21-nonholonomic-ii#w21.5

With one sampled control per extension the tree drifts; with eight it reaches.

Push the heading weight w_θ up and the tree spends its effort matching headings of random samples instead of covering the Lot — the planner is only as good as its notion of "near", and the metric is a tuning knob, not a derived quantity. Turn on Kinodynamic RRT* and the tree changes model: rewiring compares the cost of reaching a node through two different parents, which needs the *exact* optimal cost between two states. For the kinematic car that is the Reeds–Shepp length; for Hitch's dynamic model nobody has a closed form, which is why the toggle switches systems rather than pretending.

21.2.5 Grid search, then and now

Barraquand and Latombe's 1990s answer to car planning among obstacles was a best-first search over six motion primitives — full left, straight, full right, forward and reverse — with an occupancy grid on (x, y, θ) to stop it revisiting. Today's planners in autonomous vehicles are recognizably the same algorithm.

> **INTERACTIVE · w21.3**
>
> **Lattice vs Hybrid A***
>
> Hybrid A* is not a new algorithm: it is CAR GRID SEARCH with one continuous state per cell, an analytic Reeds–Shepp shot, and a better heuristic.
>
> Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch21-nonholonomic-ii#w21.3`

The left panel is a *state lattice*: the primitives were built so that every motion ends exactly on a lattice state, so the search is plain A* on a graph. The right panel is *hybrid A**: the states float freely as in Algorithm 22, each grid cell keeps the best one, and every few expansions the planner tries an analytic Reeds–Shepp "shot" to the goal (the dashed purple fans are shots that hit something). Coarsen the grid: the lattice, whose primitives must land on coarse lattice points, takes long detours; hybrid A* barely notices because its states are continuous. Neither is new; both are Algorithm 22 plus a named change.

21.2.6 Driving a dead joint

The last widget leaves the Lot. Reach's three-link arm lies flat on the Workbench with its third motor switched off.

> **INTERACTIVE · w21.6**
>
> **Decoupling Drive**
>
> An unactuated joint does not make an arm unplannable: along decoupling vector fields the free link moves like a car with two actions, and any speed profile along them needs zero torque at the dead joint.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch21-nonholonomic-ii#w21.6

The free third link cannot be torqued. Chapter 20 showed the arm is nevertheless accessible from rest, and Choset's §12.5.7 shows how to *use* that: there are two motions the dead link can perform at any speed — sliding along its own length, and rotating about its center of percussion, the one point on the link that a force at the joint perpendicular to the link leaves momentarily unaccelerated. Drive with those two (forward and reverse each) and the link behaves like a car with four actions. The planner is Algorithm 22 with cost "number of switches"; the execution is Chapter 18's time-optimal scaling, segment by segment, coming to rest at every switch. In the inset one torque bar is always full.

THE ONE-SENTENCE VERSION

Chapter 20 established *that* Hitch can reach every pose; steering is the choice of *how* to spend that controllability — by solving the optimal control exactly (Dubins, Reeds–Shepp), by climbing the bracket ladder with sinusoids in chained coordinates, by descending the end-state error, by reading the state off flat outputs, or by searching over forward-integrated primitives — and each choice buys exactness, smoothness or obstacle-awareness with some other property the others keep.

$\rho = r_{\min} = L/\tan\gamma$; v, ω	minimum turning radius; inputs of the reduced car (Choset eq. 12.35) $\lvert\omega\rvert \le \lvert v\rvert/\rho$; Hitch: L = 2.6 m, γ = 0.6 rad, ρ = 3.80 m
$L_\pm$, $R_\pm$, $S_\pm$	CAR GRID SEARCH actions: full left / right / straight, forward / reverse Choset Alg. 22
C, S, $\lvert$, C_a, $C_{\pi/2}$; (t, p, q)	Reeds–Shepp alphabet: arc, straight, cusp, arc of angle a, quarter arc; normalized segment lengths arcs in radians, straights in units of ρ
$z = A(x)x$, $v = B(x)u$	chained-form coordinates and inputs Choset §12.5.1–12.5.2
u^p, $f(x_{start}, u^p)$, e	finite control parameterization, end-state map, end-state error §12.5.4
y, φ, ψ_u	flat outputs and the maps x = ϕ(y, $\dot{y}$, . . .), u = ψ_u(y, $\dot{y}$, . . .) Choset writes ψ for the input map; here ψ is the trailer heading, so the input map is ψ_u
ψ, $\theta_1 = \theta - \psi$, d	trailer heading, hitch angle, hitch length Chapter 2 integrates ψ; d = 2.5 m
$V_i(q)$, $\mathcal{V}$, w	decoupling vector fields, a set of them, kinematic inputs §12.4.2

$G(q_{goal})$, prop$(x, u, \Delta t)$	goal region of Alg. 22; forward propagation of $\dot{x}$ = f(x, u) §12.5.6, §7.5.1

21.3 The mathematics

Throughout, the *steering problem* is: given x_{start} and x_{goal}, find controls driving the system from one to the other, *ignoring obstacles*. It is exactly what PRM and RRT call a local planner. Choset splits the state of many systems into *shape* variables, driven directly by the inputs, and *fiber* variables, which change only when the shape variables move around closed loops — the astronaut who reorients by cycling her arms. Every exact method below is a way to engineer the right loop.

21.3.1 Brockett's sinusoids

The simplest system with a fiber is Choset's (12.23):

$$\dot{x}_1 = u_1, \qquad \dot{x}_2 = u_2, \qquad \dot{x}_3 = x_2 u_1,$$

with $g_1 = [1, 0, x_2]^\top$, $g_2 = [0, 1, 0]^\top$ and $[g_1, g_2] = -e_3$. Shape (x_1, x_2), fiber x_3.

STATEMENT (BROCKETT; CHOSET §12.5.1)

For $t_f = 1$ and energy $J = \frac{1}{2}\int_0^1 u^\top u \, dt$, optimal controls are quadrature sinusoids $u_1 = u_1(0)\cos\lambda_3 t - u_2(0)\sin\lambda_3 t$, $u_2 = u_1(0)\sin\lambda_3 t + u_2(0)\cos\lambda_3 t$. For a pure fiber motion — x_1, x_2 return — $\lambda_3 = 2k\pi$, and with $k = \mp 1$, $x_3(1) = \pm\big(u_1(0)^2 + u_2(0)^2\big)/4\pi$.

DERIVATION Pontryagin for the fiber

$$\dot{u}_1 = -\lambda_3 u_2, \quad \dot{u}_2 = \lambda_3 u_1, \quad x_3(1) = \mp\frac{u_1(0)^2 + u_2(0)^2}{4\pi} \quad (k = \pm 1)$$

1. **Hamiltonian.** $H = \frac{1}{2}(u_1^2 + u_2^2) + \lambda_1 u_1 + \lambda_2 u_2 + \lambda_3 x_2 u_1$.
2. **Stationarity.** $\partial H/\partial u = 0$ gives $u_1 = -\lambda_1 - \lambda_3 x_2$ and $u_2 = -\lambda_2$.
3. **Adjoint.** $\dot{\lambda} = -\partial H/\partial x$: $\dot{\lambda}_1 = 0$, $\dot{\lambda}_3 = 0$, $\dot{\lambda}_2 = -\lambda_3 u_1$.
4. **Close the loop.** Differentiate step 2: $\dot{u}_1 = -\lambda_3 \dot{x}_2 = -\lambda_3 u_2$ and $\dot{u}_2 = -\dot{\lambda}_2 = \lambda_3 u_1$ — a rotation of (u_1, u_2) at rate λ_3, which integrates to the quadrature sinusoids.
5. **Return the shape.** $\int_0^1 u_i \, dt = 0$ requires a whole number of periods: $\lambda_3 = 2k\pi$. Then $J = \frac{1}{2}(u_1(0)^2 + u_2(0)^2)$ for every k, so pick $|k| = 1$. With

$u_2(0) = 0$, $x_2(t) = u_1(0)(1 - \cos 2k\pi t)/(2k\pi)$ and $x_3(1) = \int_0^1 x_2 u_1\, dt = -u_1(0)^2/(4k\pi)$: negative for $k = 1$, positive for $k = -1$.

So to move the fiber by 0.3 choose $k = -1$ and any $(u_1(0), u_2(0))$ on the circle of radius $\sqrt{4\pi \cdot 0.3} = 1.9416$, at cost $J = 0.6\pi \approx 1.885$; the library's `steerBrockett` returns that circle's point on the u_1-axis, and integrating (12.23) under it lands on $(0, 0, 0.3)$ to round-off. The unicycle is this system in disguise: with Choset's

$$z_1 = x_3, \quad z_2 = x_1 \cos x_3 + x_2 \sin x_3, \quad z_3 = x_1 \sin x_3 - x_2 \cos x_3, \qquad v_1 = \omega, \quad v_2 = v - z_3\omega,$$

the unicycle's $(\dot{x}_1, \dot{x}_2, \dot{x}_3) = (v \cos x_3, v \sin x_3, \omega)$ becomes $\dot{z} = (v_1, v_2, z_2 v_1)$ — check it: $\dot{z}_3 = v(\cos x_3 \sin x_3 - \sin x_3 \cos x_3) + (x_1 \cos x_3 + x_2 \sin x_3)\omega = z_2 v_1$. The heading is a shape variable, the position along the heading is the other, and the sideways offset is the fiber. Brockett's cost is not a physical one for a car, which is Choset's own caveat.

21.3.2 Chained form and the sinusoid ladder

Generalize to n states and two inputs (Choset eq. 12.28):

$$\dot{x} = u_1 g_1(x) + u_2 g_2(x), \qquad g_1 = [1,\ 0,\ x_2,\ x_3,\ \ldots,\ x_{n-1}]^\top, \qquad g_2 = e_2.$$

STATEMENT (MURRAY AND SASTRY; CHOSET §12.5.2, EQ. 12.31)

The iterated brackets climb a ladder: $\mathrm{ad}^k_{g_1} g_2 = [g_1, [g_1, \ldots [g_1, g_2]]] = (-1)^k e_{k+2}$, so the system is STLC everywhere. With $u_1 = a \sin 2\pi t$, $u_2 = b \cos 2k\pi t$ on $[0, 1]$, the states $x_1, \ldots, x_{k+1}$ return and

$$x_{k+2}(1) - x_{k+2}(0) = \left(\frac{a}{4\pi}\right)^k \frac{b}{k!}.$$

Steer by driving x_1, x_2 to their goals directly, then for $k = 1, \ldots, n-2$ in ascending order choose (a, b) by (12.31) to fix x_{k+2}.

DERIVATION Why the k-th cycle touches only x_{k+2} and above

$\Delta x_j = 0\ (j \le k+1), \qquad \Delta x_{k+2} = \left(\frac{a}{4\pi}\right)^k \frac{b}{k!}$

1. **The ladder.** $[g_1, g_2] = \frac{\partial g_2}{\partial x} g_1 - \frac{\partial g_1}{\partial x} g_2 = -\frac{\partial g_1}{\partial x} e_2 = -e_3$, because g_2 is constant and the only entry of g_1 depending on x_2 is the third.

Inductively $[g_1, e_j] = -e_{j+1}$, so $\mathrm{ad}^k_{g_1} g_2 = (-1)^k e_{k+2}$ and $\{g_1, g_2, \mathrm{ad}g_2, \ldots, \mathrm{ad}^{n-2} g_2\}$ spans $\mathbb{R}^n$ — Chapter 20's lieBracket confirms the signs to better than 10^{-10} at a random point for $n = 5$.

2. **Linearity in the initial state.** Given $u_1(t)$, the chain $\dot{x}_j = x_{j-1} u_1$ is linear. A contribution of $x_m(0)$ to $x_j(t)$ is $x_m(0)\, s(t)^{j-m}/(j-m)!$ with $s(t) = \int_0^t u_1$ — a polynomial in s, and $s(1) = 0$, so every such contribution returns.
3. **The forced part.** What remains is driven by $w(t) = \int_0^t u_2 = \frac{b}{2k\pi} \sin 2k\pi t$. Each pass down the chain multiplies by $u_1 = a \sin 2\pi t$ and integrates; by the product-to-sum identities a frequency-$2k\pi$ signal multiplied k times by a frequency-2π one contains a zero-frequency (DC) term only at the k-th multiplication, i.e. in $\dot{x}_{k+2}$. States above x_{k+2} also drift, which is why the ladder is climbed upward.
4. **The constant.** Tracking the DC coefficient through the k multiplications gives $(a/4\pi)^k b/k!$ (Murray and Sastry's computation). The check runs the five-state ladder to $(1, -0.5, 0.3, 0.2, -0.4)$ and finds every cycle's increment equal to (12.31) to better than 10^{-10} — integration error, not formula error.

Hitch's four-state car $q = (x, y, \phi, \theta)$ has a chained form too (Murray and Sastry):

$$z = \left(x',\ \frac{\tan\phi}{L\cos^3\theta'},\ \tan\theta',\ y' \right), \qquad v_1 = v\cos\theta',$$

where the primes are coordinates in a chart rotated by an angle β. In the original chart the transformation is singular at $\theta = \pm\pi/2$ — which is exactly where a parking query into a north-facing bay ends. The library sets β to the bisector of the start and goal headings, so both ends sit inside $(-\pi/2, \pi/2)$; inside the motion $\theta' = \arctan z_3$ never reaches $\pm\pi/2$ by construction. On w21.1's default query, $(5.5, 1.2, 0) \to (9.25, 6.4, \pi/2)$, two sinusoid cycles land on the goal in chained coordinates to better than 10^{-9} — with three cusps and a peak steering angle of 1.22 rad, twice what Hitch can do. The method is exact *in its coordinates* and blind to everything else. Car-and-trailer systems with any number of trailers also have chained forms (Sørdalen's result, Choset ref. 393), at the price of coordinate changes that blow up at the jackknife.

21.3.3 Gradient steering, and the cross-link to Chapter 19

Exact methods need structure. Choset's §12.5.4 needs only a simulator: parameterize the control history by a finite vector $u^p \in \mathbb{R}^r$ (here, piecewise-constant (v, ω) on $K = 6$ equal intervals of $[0, 1]$ for the reduced car), call $f(x_{start}, u^p)$ the *end-state map*, and descend the error $e = f(x_{start}, u^p) - x_{goal}$.

STATEMENT (CHOSET §12.5.4)

The update $u^p \leftarrow u^p + \alpha v$ with $v^\top = -e^\top \partial f/\partial u^p$ converges for small enough α if $\partial f/\partial u^p$ has rank n everywhere — generically true for an STLC system with a rich parameterization. At a *singular control*, where the rank drops, append a *generic loop*: a random control with zero net motion.

DERIVATION **Descent, rank, and the generic loop**

$\frac{d}{d\alpha}\frac{1}{2}\|e\|^2\big|_{\alpha=0} = -\|J^\top e\|^2 < 0$ unless $e \in \ker J^\top$

1. **Descent.** With $J = \partial f/\partial u^p$, $\frac{1}{2}\|e(u^p + \alpha v)\|^2 = \frac{1}{2}\|e\|^2 - \alpha\|J^\top e\|^2 + O(\alpha^2)$, so the step decreases the error unless $J^\top e = 0$.
2. **Rank.** If $\operatorname{rank} J = n$ then $J^\top$ is injective and $J^\top e = 0$ forces $e = 0$: the only stationary point is the goal. Any error direction is correctable by a small change of u^p.
3. **The singular control.** At $u^p = 0$ every column of J is a first-order variation — a combination of g_1 and g_2 at x_{start} — so $\operatorname{rank} J = 2 < 3$ and the sideways error is invisible. A driftless system retraces itself under a control followed by its time-reversed negative, so appending such a pair changes nothing about where the car ends and everything about the Jacobian.

For a pure sideways move of 0.5 m from rest the library's `steerGradient` finds rank 2 at $u^p = 0$, appends a two-piece generic loop (rank 3), and then needs 58 backtracking gradient steps to bring $\|e\|$ below 10^{-4} — slow, because the sideways direction is a bracket direction and the Jacobian's singular value along it is small; Chapter 20's ϵ^2 reappears as conditioning. Hand the same end-state map to Chapter 19's Levenberg–Marquardt — §12.5.3's "nonlinear optimization", with no re-derivation needed — and five iterations reach round-off. Choset's two caveats carry over unchanged: a good initial guess is required, and nothing guarantees convergence; here the generic loop *is* the initial guess that makes it work.

21.3.4 Differential flatness

STATEMENT (FLIESS, LÉ VINE, MARTIN AND ROUCHON; CHOSET §12.5.5)

For the reduced car, $y = (x_1, x_2)$ — the rear-axle midpoint — is a flat output:

$$\theta = \operatorname{atan2}(\dot{y}_2, \dot{y}_1), \qquad v = \pm\sqrt{\dot{y}_1^2 + \dot{y}_2^2}, \qquad \omega = \frac{\dot{y}_1\ddot{y}_2 - \ddot{y}_1\dot{y}_2}{\dot{y}_1^2 + \dot{y}_2^2}$$

(eqs. 12.32–12.33), and for the four-state car $\tan\phi = L\kappa$ with $\kappa = \omega/v$. With a trailer hitched at the rear axle the flat output is the trailer's axle midpoint; with p trailers, the last one's.

DERIVATION Heading from the tangent, steering from curvature, the car from the trailer

$$\phi = \arctan\Big(L\,\frac{\dot{p}\times\ddot{p}}{\|\dot{p}\|^3}\Big), \qquad p = y + d\,(\cos\psi, \sin\psi), \quad \psi = \operatorname{atan2}(\dot{y})$$

1. **Heading.** No slip means the rear axle's velocity points along the body: $\dot{y} = v(\cos\theta, \sin\theta)$. Read θ and $|v|$ off $\dot{y}$; the sign of v is a choice that must stay consistent with θ (Choset's warning: $\dot{y} = (1,1)$ and $(-1,-1)$ give the same atan).
2. **Turn rate and curvature.** $\omega = \dot{\theta} = \frac{d}{dt}\operatorname{atan2}(\dot{y}_2, \dot{y}_1) = (\dot{y}_1\ddot{y}_2 - \ddot{y}_1\dot{y}_2)/\|\dot{y}\|^2$, the cross product over speed squared; $\kappa = \omega/v$ is that over speed cubed, and the front-wheel geometry gives $\tan\phi = L\kappa$.
3. **The trailer.** By (12.36)–(12.37) the trailer axle moves along the trailer's heading, so $\psi = \operatorname{atan2}(\dot{y})$ when y is the trailer axle; the hitch — the car's rear axle — sits at $p = y + d(\cos\psi, \sin\psi)$. Differentiate: $\dot{p} = \dot{y} + d\dot{\psi}(-\sin\psi, \cos\psi)$ and $\ddot{p} = \ddot{y} + d\ddot{\psi}(-\sin\psi, \cos\psi) - d\dot{\psi}^2(\cos\psi, \sin\psi)$, where $\dot{\psi}$ is step 2 applied to y and $\ddot{\psi}$ needs $\dddot{y}$.
4. **Apply steps 1–2 to p.** The car's heading needs $\dot{p}$ (two derivatives of y), its steering $\ddot{p}$ (four). One more pair of derivatives per body: "differentiate once more per trailer". Every formula divides by a speed, so the lift is singular wherever the flat output stops.

Choset's Example 12.5.1 fits $y_1 = t$, $y_2 = 3t^2 - 2t^3$ between $(0,0,0)$ and $(1,1,0)$; the library's Hermite fit with unit end speed returns exactly those polynomials. At $t = 0.25$: $y = (0.25, 0.15625)$, $\dot{y} = (1, 1.125)$, $\ddot{y} = (0, 3)$, so

$$\theta = \operatorname{atan2}(1.125, 1) = 0.844154 \text{ rad } (48.37^\circ), \quad v = \sqrt{2.265625} = 1.505199, \quad \omega = \frac{3}{2.265625} = 1.324138,$$

$\kappa = \omega/v = 0.879709$, and for wheelbase $L = 1$ the steering angle is $\phi = \arctan 0.879709 = 0.721491$ rad (41.34°). At $t = 0.5$ the curve inflects and $\omega = \phi = 0$ exactly. The chapter's check goes one step further: it takes the controls $(v(t), \phi(t))$ read off a quintic flat output, integrates the *true* car (and car-with-trailer) ODE forward from the lifted initial state, and finds the integrated

state agreeing with the lifted one along the whole trajectory to better than 10^{-11}. Flatness is not an approximation.

What it is not, either, is a planner with constraints. A polynomial of minimal degree has no freedom left to respect $|\phi| \le \gamma$ or avoid a pillar; extra coefficients plus a numerical optimizer can, and Choset calls that "a topic of current research". On w21.1's trailer query the trailer-axle quintic asks for $|\phi| = 1.39$ rad and clips an obstacle.

21.3.5 Shortest paths: Dubins and Reeds–Shepp

STATEMENT (DUBINS; REEDS AND SHEPP; CHOSET §12.5.6)

For the reduced car with $|v| \le 1$ and $|\omega| \le |v|/\rho$, a shortest path exists and lies in Choset's nine families $\{C|C|C,\ CC|C,\ C|CC,\ CC_a|C_aC,\ C|C_aC_a|C,\ C|C_{\pi/2}SC,\ CSC_{\pi/2}|C,\ C|C_{\pi/2}SC_{\pi/2}|C,\ CSC\}$ — Reeds and Shepp's forty-eight signed words, which Sussmann and Tang later trimmed to forty-six. Forward-only, the shortest path is one of the six Dubins words $\{LSL, RSR, LSR, RSL, RLR, LRL\}$.

DERIVATION From Pontryagin to six words, and LSL in closed form

$p^2 = 2 + d^2 - 2\cos(\alpha-\beta) + 2d(\sin\alpha - \sin\beta)$, $t = -\alpha + \operatorname{atan2}(\cos\beta - \cos\alpha,\ d + \sin\alpha - \sin\beta)$, $q = \beta - \operatorname{atan2}(\cdot)$

1. **Bang-bang or singular.** Minimize time with $v \in [-1,1]$, $\omega = v\kappa$, $|\kappa| \le 1/\rho$. The Hamiltonian is linear in κ, so κ sits at $\pm 1/\rho$ (bang) except on intervals where its switching function vanishes identically (singular), which forces $\kappa = 0$.
2. **Arcs and straights.** Optimal paths are concatenations of full-lock arcs C and straights S — exactly CAR GRID SEARCH's primitives, which is why Algorithm 22 loses nothing essential by discretizing the steering.
3. **Case analysis.** Reeds and Shepp's argument bounds the number of segments and the arc angles between cusps, leaving the nine families; with $v \ge 0$ forced there are no cusps and only CSC and CCC remain — Dubins' six words.
4. **Normalize.** Put the start at the origin, the goal at $(d, 0)$ with $d = \|q_1 - q_0\|_{xy}/\rho$, and measure both headings from the bearing of the goal: $\alpha = \theta_0 - \text{bearing}$, $\beta = \theta_1 - \text{bearing}$.
5. **LSL.** The start's left circle has center $c_0 = (-\sin\alpha, \cos\alpha)$, the goal's $c_1 = (d - \sin\beta, \cos\beta)$. Two equal circles turning the same way are joined

by their external tangent, parallel to $c_1 - c_0$ and of the same length: $p^2 = \|c_1 - c_0\|^2 = (d - \sin\beta + \sin\alpha)^2 + (\cos\beta - \cos\alpha)^2$, which expands to the boxed formula. The tangent's direction is $\operatorname{atan2}(\cos\beta - \cos\alpha,\ d + \sin\alpha - \sin\beta)$; the first arc turns from α to it (t), the last from it to β (q), both taken mod 2π.

6. **The other five.** Reflect ($y \to -y$, headings negated) and LSL becomes RSR. Reverse time and read the word backward and LSR and RSL swap; the internal tangent replaces the external one for the mixed words ($p^2 = d^2 - 2 + 2\cos(\alpha - \beta) + 2d(\sin\alpha + \sin\beta)$ for LSR, infeasible when negative), and three mutually tangent circles give RLR and LRL, infeasible when the circles are more than 4ρ apart.

The micro-example. $q_0 = (0, 0, 0)$, $q_1 = (3, 3, \pi/2)$, $\rho = 1$. The start's left circle is centered at $(0, 1)$, the goal's at $(3, 3) + (-\sin\frac{\pi}{2}, \cos\frac{\pi}{2}) = (2, 3)$; the centers are $2\sqrt{2}$ apart along direction $\pi/4$, so LSL has $t = \pi/4$, $p = 2\sqrt{2}$, $q = \pi/4$ and length $\pi/2 + 2\sqrt{2} = 4.39922$. By integration: after the first arc Hitch is at $(0.70711, 0.29289)$ heading $\pi/4$; the straight adds $(2, 2)$; the last arc about $(2, 3)$ ends at $(3, 3)$ heading $\pi/2$. The others are longer — $LSR = RSL = 10.56698$, $LRL = 10.99557$, $RSR = 16.65243$ — and RLR is infeasible. Reeds–Shepp returns the same $L^+S^+L^+$: here a cusp cannot help.

Reeds–Shepp by symmetry. Nobody codes forty-eight formulas. Reeds and Shepp's §8 gives a handful of base formulas — $L^+S^+L^+$, $L^+S^+R^+$, L^+R^-L, $L^+R_a^+L_a^-R^-$, $L^+R_a^-L_a^-R^+$, $L^+R_{\pi/2}^-S^-L^-$, $L^+R_{\pi/2}^-S^-R^-$, $L^+R_{\pi/2}^-S^-L_{\pi/2}^-R^+$ — and three symmetries of the normalized goal (x, y, φ) generate the rest:

time-flip: $(x, y, \varphi) \mapsto (-x, y, -\varphi)$, reflect: $(x, y, \varphi) \mapsto (x, -y, -\varphi)$, backwards: $(x, y, \varphi) \mapsto (x\cos\varphi + y\sin\varphi,\ x\sin\varphi - y\cos\varphi,\ \varphi)$.

Time-flip negates every segment ($L^+ \leftrightarrow L^-$), reflect swaps L and R, backwards reads the word right to left. The library evaluates eleven base formulas (four of them backwards readings) under the four flip/reflect combinations — forty-four evaluations, the construction OMPL's `ReedsSheppStateSpace` uses. Several base formulas leave the sign of their last segment free, so the forty-four evaluations cover the forty-eight words; the module does not prove that itself, and the check tests the two consequences that would fail if it were false. On 400 seeded pairs, every one of the 2,700 candidate words integrates to its goal to round-off (worst error 10^{-14}), and the shortest Reeds–Shepp length never exceeds the shortest Dubins length (Dubins paths are Reeds–Shepp candidates, so a missing family or a mistyped formula would show up as a violation). All nine families appear among the optimal words. The Reeds–Shepp length is also symmetric — $d(q_0, q_1) = d(q_1, q_0)$ — because reversing time maps a path to a path; that is what makes it a *metric* on $SE(2)$, and the reason PRM can use it as an undirected local planner while a Dubins-steered PRM needs directed edges.

21.3.6 Local planners with the topological property

STATEMENT (LAUMOND; SEKHAVAT AND LAUMOND; CHOSET §12.5.6)

If the free-flying car body has a free path that keeps an open free neighborhood, then recursive bisection with a Reeds–Shepp local planner terminates with a feasible car path, and PRM with a Reeds–Shepp local planner is probabilistically complete. Parking in a space ϵ longer than the car needs $\Theta(1/\epsilon^2)$ cusps.

DERIVATION STLC, the topological property, and compactness

$\exists \epsilon > 0 : \|q - q'\| < \epsilon \Rightarrow \mathrm{RS}(q, q') \subset B_\delta(q)$

1. **The topological property** (Choset Fig. 12.28). For every ball $B_\delta(q)$ there is a ball $B_\epsilon(q)$ such that the local planner's path from q to any $q' \in B_\epsilon(q)$ stays inside $B_\delta(q)$. Reeds–Shepp has it because the car is STLC and the shortest path between close poses is short: its length is $O(\|q - q'\|^{1/2})$ in the sub-Riemannian sense, so it cannot wander far.
2. **Uniformity.** The free-flying path is compact and at positive clearance δ from obstacles; by compactness one ϵ works along its whole length.
3. **Bisection terminates.** Halve the parameter interval until consecutive samples are within ϵ; each Reeds–Shepp piece then stays in a δ-ball around a free configuration, hence is free.
4. **PRM.** Chapter 11's completeness proof covered the path with a chain of balls and needed only that the local planner connects samples in consecutive balls; it never used that the planner's path is a straight line, nor that it is symmetric. The cusp count: each reversal of a car of length ℓ in a slot of length $\ell + \epsilon$ gains $O(\epsilon^2)$ of sideways motion (Chapter 20's bracket), so a fixed sideways offset needs $\Theta(1/\epsilon^2)$ of them.

The library implements both: `ReedsSheppLocal` and `DubinsLocal` satisfy Chapter 11's `Steer<Pose2>` interface (symmetric and not, respectively), and `pathTransform` is Laumond's bisection. Their price is the next honesty item: the curvature jumps at every arc–straight junction, so either the steering wheel turns instantaneously, the car stops at each junction, or execution has error. Smoothed primitives — clothoids, or flat-output polynomials — are the standard repair.

21.3.7 CAR GRID SEARCH

STATEMENT (BARRAQUAND AND LATOMBE; CHOSET ALG. 22)

Best-first search over $\{L_\pm, R_\pm, S_\pm\}$, with a d^3 occupancy grid on (x, y, θ) and cost $a \cdot$ steps $+ b \cdot$ steering changes $+ c \cdot$ cusps, finds a path to the goal region $G(q_{goal})$ if one exists at a small enough step, and the path it finds is optimal for the cost at the given step and resolution. Any path of the full car with p cusps can be followed arbitrarily closely by a path of full-lock arcs with p cusps.

DERIVATION Termination, optimality, and the bucketed OPEN list

#\{\text{expansions}\} \le d^3,\qquad \text{pop order} = \text{nondecreasing integer cost}

1. **Termination.** A node is expanded only if its cell has not been marked; there are d^3 cells, so at most d^3 expansions and $6d^3$ children. The step must be long enough to leave the current cell, and the goal region must be larger than a cell — otherwise the search can loop inside a cell or jump over the goal.
2. **Best-first on integer costs is Dijkstra.** With nonnegative integer a, b, c, every edge cost is a nonnegative integer, and popping the cheapest open node is Chapter 6's Dijkstra on the tree. Testing the goal at *pop* time (line 4), not at generation, is what makes the returned path the cheapest one: Choset's Fig. 12.25 has a node in the goal region waiting in OPEN behind a cheaper one.
3. **Constant-time insertion.** Integer costs index an array of FIFO buckets; insertion is an append, and the pop scans forward from the lowest non-empty bucket, which only ever moves up.
4. **Full lock suffices.** By the Pontryagin argument of the previous section, optimal paths between cusps are full-lock arcs and straights; Barraquand and Latombe show more — that any p-cusp path is approximable by a p-cusp full-lock path — so minimizing cusps with $a = b = 0$ loses nothing to the discretization of steering.

Choset's description includes two practicalities the implementation keeps. The occupancy grid is checked when a node is *popped* (line 7), so a cell can be reached several times but expanded once; and with a trailer the grid gains a fourth axis for the hitch angle, the trailer's change over one action being "numerically integrated, or stored in a lookup table" — here by Chapter 2's `Hitch.step`, whose

RK4 trailer integration is the lookup table computed on demand. On the Lot's reverse-parking query the search with $a = 1, c = 0$ and the search with $c = 300$ both find a 20-step path with one cusp; the second expands half as many nodes, because the cusp penalty prunes every branch that reverses early. Choset's remark that the planner "actually runs faster in cluttered spaces because the obstacles prune the search tree" is the same effect from the other side.

21.3.8 Forward propagation

STATEMENT (LAVALLE AND KUFFNER; CHOSET §7.5.1)

With prop$(x, u, \Delta t)$ Lipschitz in x and controls sampled with full support, the single tree that extends the nearest node toward a random state by the best of k sampled controls reaches any goal region of nonzero measure with probability tending to one.

DERIVATION A tube, a chain of cross-sections, and the Voronoi bias

$P[\text{no node in } G \text{ after } n \text{ extensions}] \to 0$

1. **A tube.** Take a feasible trajectory reaching the interior of G with clearance. Lipschitz continuity of prop in x (and continuity in u) gives a tube of nearby trajectories: from any state within r_i of the i-th waypoint, some open set of controls lands within r_{i+1} of the next.
2. **Cross-sections.** Discretize the trajectory into K steps of Δt and ball B_i around each waypoint.
3. **One step has positive probability.** If the tree has a node in B_i, an extension succeeds in placing one in B_{i+1} with probability at least $p > 0$: the random sample lands in the Voronoi region of that node with positive probability (Chapter 12's bias), and some sampled control falls in the open good set with positive probability.
4. **Chain.** The event "progress from B_i to B_{i+1} within m extensions" fails with probability at most $(1-p)^m$; multiply over K stages and let $m \to \infty$.
5. **What the proof never used.** Symmetry of the edge relation, reflexivity, or any ability to steer *to* a state — the theorem holds for the asymmetric, forward-only relation "some u drives x to x' in Δt".

Two honesty items follow from the proof. The constant p depends on the metric through step 3 — a metric that ranks a node with the wrong heading as "near" wastes extensions, which is the w21.5 slider — and nothing in the proof bounds path *quality*. On the Lot query from w21.1, Hitch's dynamic model

(acceleration and steering-rate inputs, speed and steering angle in the state) with best-of-eight controls and $\Delta t = 0.8$ s first lands in the goal region at extension 106 for seed 21, and re-running the simulator on the stored controls reproduces every state bit for bit. Time can be added to the state to handle moving obstacles (Choset ref. 195); the theorem does not change.

21.3.9 Kinodynamic RRT*: optimality needs a steer

Chapter 13's RRT* is asymptotically optimal because its rewiring step asks, for a node v near a new node x, whether reaching v *through* x is cheaper — a question about the exact optimal cost $c^*(x, v)$ and the trajectory achieving it. Forward propagation cannot answer it. Webb and van den Berg's Kinodynamic RRT* answers it for linear dynamics $\dot{x} = Ax + Bu$ with cost $\int_0^\tau (1 + u^\top R u)\, dt$: for a fixed arrival time τ the minimum-energy cost is

$$c(\tau) = \tau + \big(x_1 - e^{A\tau}x_0\big)^\top G(\tau)^{-1}\big(x_1 - e^{A\tau}x_0\big), \qquad G(\tau) = \int_0^\tau e^{A(\tau-t)}BR^{-1}B^\top e^{A^\top(\tau-t)}\, dt,$$

and the free final time is $\tau^* = \arg\min_\tau c(\tau)$. For the planar double integrator with $R = rI$ each axis gives $G(\tau) = \frac{1}{r}\begin{bmatrix} \tau^3/3 & \tau^2/2 \\ \tau^2/2 & \tau \end{bmatrix}$ and $c(\tau) = \tau + r\sum(12\Delta p^2/\tau^3 - 12\Delta p\,\Delta v/\tau^2 + 4\Delta v^2/\tau)$ with $\Delta p = p_1 - p_0 - v_0\tau$, $\Delta v = v_1 - v_0$. From $(0, 0, 1, 0)$ to $(2, 1, 0, 0.5)$ with $r = 0.5$ the library finds $\tau^* = 2.3893$ s at cost 3.0075, and integrating the closed-form optimal control $u(t) = R^{-1}B^\top e^{A^\top(\tau-t)}G(\tau)^{-1}d$ lands on the target to round-off. For the reduced *kinematic* car the exact steer is Reeds–Shepp, which is what w21.5's toggle uses; Hitch's dynamic model has neither, and Exercise 6 asks you to linearize it about rest and use the LQR edge anyway.

21.3.10 Kinematic reductions and decoupling vector fields

Chapter 20 left Reach's dead-motor arm with an uncomfortable verdict: accessible from rest, with Lewis–Murray unable to certify STLC. §12.4.2 changes the question from "can it get anywhere?" to "can I plan it like a kinematic system?".

STATEMENT (LEWIS; BULLO AND LYNCH; CHOSET THEOREMS 12.4.5, 12.4.7)

A driftless $\dot{q} = \sum V_i w_i$ is a *kinematic reduction* of $\nabla_{\dot{q}}\dot{q} = \sum Y_i u_i$ if all its trajectories are feasible for the mechanical system. The system is *maximally reducible* iff $\mathrm{Sym}(\mathcal{Y}) = \mathrm{span}(\mathcal{Y})$ (Theorem 12.4.5). A single field V is *decoupling* — followable at any speed profile $w(t)$ — iff

$$V \in \mathrm{span}(\mathcal{Y}) \quad \text{and} \quad \nabla_V V \in \mathrm{span}(\mathcal{Y})$$

(Theorem 12.4.7). If some set of decoupling fields has $\mathrm{Lie}(\mathcal{V})(q) = T_q\mathcal{Q}$ the system is STLKC at q, hence equilibrium controllable by decoupling

segments with stops at the switches.

DERIVATION **Why V and ∇_V V must both be actuated**

$\nabla_{\dot q}\dot q = V\dot w + \nabla_V V\, w^2 \in \operatorname{span}(\mathcal{Y})\ \ \forall\, w, \dot w$

1. **Substitute.** Along $\dot q = V(q)w$, $\ddot q = V\dot w + \frac{\partial V}{\partial q}Vw^2$ (Choset 12.16–12.17).
2. **Covariant form.** $\nabla_{\dot q}\dot q = \ddot q + M^{-1}\dot q^{\top}\Gamma\dot q = V\dot w + \big(\frac{\partial V}{\partial q}V + M^{-1}V^{\top}\Gamma V\big)w^2 = V\dot w + \nabla_V V\, w^2$.
3. **Feasibility for all speed profiles.** The left side must equal $\sum Y_i u_i$ for *some* u, for every w and $\dot w$ independently: the $\dot w$ term forces $V \in \operatorname{span}(\mathcal{Y})$, the w^2 term forces $\nabla_V V \in \operatorname{span}(\mathcal{Y})$. Conversely, both memberships let you solve for u.
4. **Finding them.** Write $V = \sum h_i(q)Y_i$ and solve $\langle X_c, \nabla_V V\rangle = 0$ for the M-orthogonal complement X_c of $\operatorname{span}(\mathcal{Y})$ — quadratic equations in the h_i. Then plan in $\boldsymbol{Q}$ and time-scale each segment with Chapter 18.

For the planar body with thrusters Choset finds $V_1 = Y_1$ (translation along the thrust line) and $V_2 = Y_2$ (rotation about the center of percussion at distance $1/d$), with $\nabla_{V_1}V_1 = 0$ and $\nabla_{V_2}V_2 \in \operatorname{span}(Y_1)$; but $\langle Y_1 : Y_2\rangle \notin \operatorname{span}(\mathcal{Y})$, so $V_1 + V_2$ is *not* decoupling (Example 12.4.8). For the 3R arm with $u_3 = 0$ the free third link is that body, its "thrusters" the forces joints 1 and 2 apply at joint 3. The two decoupling fields are translation along link 3 and rotation about its center of percussion with respect to joint 3, at distance

$$k = \frac{I_3 + m_3 r_3^2}{m_3 r_3} = \frac{0.004125 + 0.5 \cdot 0.15^2}{0.5 \cdot 0.15} = 0.205\ \text{m}$$

for Choset's Table 12.1 robot. In joint coordinates both are built through the inverse of the 2×2 Jacobian of joint 3's position. The check evaluates Theorem 12.4.7 numerically with Chapter 20's kinetic-energy connection at three configurations: both fields pass with residuals below 5×10^{-10}; rotation about the center of *mass* fails (it is not even in $\operatorname{span}(\mathcal{Y})$ — a rotation about the COM needs a torque the dead joint cannot supply); and the sum of the two fails the second condition, exactly as for the free body.

21.3.11 Modern descendants: lattices and hybrid A*

Choset's chapter ends in 2005 with Algorithm 22 and PRM-with-Reeds–Shepp. The planners that parked the DARPA Urban Challenge vehicles, and their successors, are those two ideas recombined, with three deltas:

1. **Lattice-consistent primitives** (state lattices, Pivtoraiko, Knepper and Kelly). Choose primitives that start and end *exactly* on lattice states — grid points times a set of headings — so the search graph is a real graph and A* runs unchanged. The library builds them with this chapter's Dubins solver: for each start heading and each category (straight, ±45°, ±90°, lane change), the cheapest Dubins path to a lattice endpoint whose total turning equals the heading change. The cost of exactness is that the start and goal must be lattice states.
2. **One continuous state per cell** (hybrid A*, Dolgov, Thrun, Montemerlo and Diebel). Keep Algorithm 22's free-running continuous configurations and its occupancy grid, but let each cell hold the cheapest state that reached it rather than the first.
3. **Analytic expansions and a dual heuristic** (hybrid A*). Every N expansions try a Reeds–Shepp shot from the popped state straight to the goal and accept it if it is free — the PRM local planner, inside the search. Order OPEN by max(Reeds–Shepp length ignoring obstacles, 2-D shortest distance ignoring kinematics): each term is a lower bound on the true cost, so their maximum is too. (The library's 2-D table is scaled by 1/1.0824, the worst octile-to-Euclidean ratio, to keep it admissible.)

On the Lot, reversing into bay 3 from $(2, 1, 0)$ to $(9.5, 8.5, -\pi/2)$ at 0.5 m resolution with the dual heuristic: the six-primitive state lattice expands 298 states and returns a 24.368 m path with three cusps; hybrid A* with a shot every ten expansions expands 161, fires seventeen shots, and returns 17.107 m with three cusps, ending *exactly* on the goal. Both are at least the obstacle-blind Reeds–Shepp distance, 15.189 m. Hybrid A* is not optimal (the cell pruning discards states that might have led to cheaper paths) and the lattice is optimal only over its primitives; w21.3 lets you watch both trade-offs.

21.4 The algorithm

ALGORITHM Algorithm 22 -- CAR GRID SEARCH (Barraquand and Latombe) COST at most d^3 expansions (d^4 with a trailer), six motions each; O(1) OPEN insertion by cost buckets

in start q_start, goal region G(q_goal), action set {L±, S±, R±}, step Δ, grid resolution d, integer weights (a, b, c), MAXTREESIZE, collision checker

out a path of actions from q_start into G(q_goal), or FAILURE

1: initialize the tree T and the bucket array OPEN with q_{start} at cost 0
2: **while** OPEN is not empty and $|T|$ < MAXTREESIZE:

3: $q \leftarrow$ first node of the lowest non-empty bucket; remove it
4: **if** $q \in G(q_{goal})$: **return** SUCCESS and the path root $\rightarrow q$
5: **if** the cell of q in the d^3 grid is not marked:
6: mark it
7: **for** each action $\in \{L_\pm, R_\pm, S_\pm\}$: integrate for arc length Δ (Hitch.step, with the trailer if hitched) to q_{new}, checking the footprint at substeps
8: **if** the motion is free: add q_{new} under q; cost = cost(q) + a + b [steer changed] + c [direction changed]; append to that cost's bucket
9: **return** FAILURE

ALGORITHM `steer_sinusoids(x_start, x_goal) -- Choset §12.5.2` COST `n − 1 integrations of the chain`

in chained-form start and goal in $\mathbb{R}^n$, optional amplitude cap
out n − 1 control phases on unit time and the integrated trace

1: phase 0: $u_1 = x_{goal,1} - x_{start,1}$, $u_2 = x_{goal,2} - x_{start,2}$, constant on $[0, 1]$
2: **for** $k = 1, \ldots, n-2$: $\Delta \leftarrow x_{goal,k+2} - x_{k+2}$; $a = b = (|\Delta|\, k!\, (4\pi)^k)^{1/(k+1)}$ with the sign of Δ on b; if a exceeds the cap, clamp it and set $b = \Delta\, k!\, (4\pi/a)^k$
3: apply $u_1 = a \sin 2\pi t$, $u_2 = b \cos 2k\pi t$ for unit time and integrate
4: for Hitch's car: map poses to $z = (x', \tan\phi/(L\cos^3\theta'), \tan\theta', y')$ in the chart rotated to the heading bisector, and back

ALGORITHM `steer_gradient(x_start, x_goal, u^0, ε) -- Choset §12.5.4` COST `per iteration: one end-state Jacobian (2r propagations) and a backtracking line search`

in start and goal, initial parameter vector (zero by default), tolerance, a seeded RNG for generic loops
out u^p with ||f(x_start, u^p) − x_goal|| < ε, or the iterate where the line search stalled

1: **if** rank $\partial f/\partial u^p < n$: append a generic loop (random pieces, then the same pieces reversed and negated)
2: **repeat** until $\|e\| < \epsilon$: $J \leftarrow \partial f/\partial u^p$ by central differences; $v \leftarrow -J^\top e$
3: **if** $v = 0$: append another generic loop and continue
4: double α, then halve it until $\frac{1}{2}\|e(u^p + \alpha v)\|^2 \le \frac{1}{2}\|e\|^2 - 10^{-4}\alpha\|v\|^2$; accept

5: (§12.5.3 variant: hand $r(u^p) = e$ and J to Chapter 19's levenbergMarquardt)

ALGORITHM `kinodynamic_rrt(prop, x_start, G, n, metric, k)` -- Choset §7.5.1 COST per extension: one nearest-neighbor query and k propagations of substeps collision checks

in a simulator prop(x, u, Δt), a control sampler, a state sampler with goal bias, a metric, controls per extension k

out a tree whose root-to-goal branch is a list of (control, Δt) pairs that re-integrates exactly

1: tree ← $\{x_{start}\}$
2: **for** $i = 1, \ldots, n$: $x_{rand} \leftarrow$ goal sample with probability p_g, else uniform
3: $x_{near} \leftarrow \arg\min_{v \in \text{tree}} \text{metric}(v, x_{rand})$
4: **for** $j = 1, \ldots, k$: sample u_j; propagate x_{near} for Δt in substeps, discarding u_j at the first collision
5: keep the free endpoint closest to x_{rand}; add it with its control and segment
6: **if** any state of the segment is in G: **return** the branch

ALGORITHM `hybrid_a_star(q_start, q_goal, cell, h, N)` COST A* over at most (cells × heading bins) expansions; one RS evaluation per heuristic call

in start and goal poses, cell size, heuristic (Euclidean, RS, or max(RS, 2-D)), shot period N, heading bins

out arc primitives followed by one Reeds–Shepp shot, ending exactly at q_goal

1: push q_{start}; best g per cell ← ∞ except the start's
2: **while** OPEN not empty: pop the lowest $f = g + h$; **if** its cell is closed, continue; close it
3: **every** N-th expansion: RS shot to q_{goal}; **if** free, **return** path to here + shot
4: **for** each of $L_\pm, S_\pm, R_\pm$ (arc length just long enough to leave the cell and, at full lock, the heading bin — otherwise a turning child ties with its straight sibling in the same cell and is pruned): integrate, check, compute the child's cell
5: **if** the cell is open and g_{child} < best g of that cell: record it, push with $f = g_{child} + h(q_{child})$

ALGORITHM `decoupling_grid_search(V, q_start, G, ik) then time_scale -- Choset §12.5.7` COST Alg. 22 over link-3 poses with 4 primitives and lexicographic cost (switches, steps); one time scaling per segment

in decoupling primitives (translate ±, rotate ± about the center of percussion), link-3 start and goal poses, IK with a fixed elbow, collision and singularity margin

out a switch-minimizing sequence of decoupling segments and its time-optimal, rest-to-rest execution

1: best-first over link-3 poses (x_{J3}, y_{J3}, α) with a d^3 occupancy grid; children by the four primitives, valid if IK exists with $|\sin q_2| \geq$ margin and the whole arm is free
2: cost = (number of switches, number of steps) compared lexicographically; goal tested at pop
3: merge equal consecutive primitives into segments
4: **for** each segment: path $q(s)$ by IK; Chapter 18's time scaling of the two actuated rows from rest to rest (the third row is identically zero along a decoupling path)

21.5 Implementation in Rust

The `nonholonomic` crate of Chapter 20 gains a `steer/` directory, `car_grid.rs`, `lattice/` and `reduction.rs`; forward propagation gets its own crate, `kinodynamic`, which owns the `Propagate` trait Chapter 14's preview borrowed. The TypeScript port in `web/lib/nonholonomic/` — `dubins.ts`, `reeds-shepp.ts`, `chained.ts`, `gradient.ts`, `flat.ts`, `car-grid.ts`, `lattice.ts`, `kinodynamic.ts`, `reduction.ts`, `steer.ts`, `park.ts` — is the code behind every widget on this page, and its `__checks_ch21__.ts` pins every number in the prose.

RUST **crates/nonholonomic/src/steer/dubins.rs**

```rust
use manifold::SE2;

#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum Word { LSL, RSR, LSR, RSL, RLR, LRL }

/// Dubins shortest path for the reduced car (Choset eq. 12.35). Segment lengths are
/// normalized by rho -- arcs in radians, the straight in units of rho -- so the closed forms
/// are scale-free; every word returned is checked by integration in the tests.
#[derive(Clone, Copy, Debug)]
pub struct DubinsPath { pub word: Word, pub segs: [f64; 3], pub rho: f64 }

impl DubinsPath {
    pub fn len(&self) -> f64 { self.rho * self.segs.iter().sum::<f64>() }
}

fn mod2pi(a: f64) -> f64 { a.rem_euclid(std::f64::consts::TAU) }
```

```rust

/// Start at the origin, goal at (d, 0): headings measured from the bearing of the goal.
fn normalize(q0: &SE2, q1: &SE2, rho: f64) -> (f64, f64, f64) {
    let (dx, dy) = (q1.x - q0.x, q1.y - q0.y);
    let bearing = dy.atan2(dx);
    (dx.hypot(dy) / rho, mod2pi(q0.theta - bearing), mod2pi(q1.theta - bearing))
}

/// LSL: the two left circles and their external tangent (Derivation 5, step 5).
fn lsl(d: f64, a: f64, b: f64) -> Option<[f64; 3]> {
    let p2 = 2.0 + d * d - 2.0 * (a - b).cos() + 2.0 * d * (a.sin() - b.sin());
    if p2 < 0.0 { return None; }
    let th = (b.cos() - a.cos()).atan2(d + a.sin() - b.sin());
    Some([mod2pi(-a + th), p2.sqrt(), mod2pi(b - th)])
}

pub fn dubins_shortest_path(q0: &SE2, q1: &SE2, rho: f64) -> Option<DubinsPath> {
    let (d, a, b) = normalize(q0, q1, rho);
    // RSR, LSR, RSL, RLR, LRL follow from LSL by reflection (L <-> R, headings negated)
    // and by reading the word backward; each has its own feasibility test.
    [(Word::LSL, lsl(d, a, b)), (Word::RSR, rsr(d, a, b)), (Word::LSR, lsr(d, a, b)),
     (Word::RSL, rsl(d, a, b)), (Word::RLR, rlr(d, a, b)), (Word::LRL, lrl(d, a, b))]
        .into_iter()
        .filter_map(|(word, s)| s.map(|segs| DubinsPath { word, segs, rho }))
        .min_by(|x, y| x.len().total_cmp(&y.len()))
}
```

RUST `crates/nonholonomic/src/steer/reeds_shepp.rs`

```rust
use manifold::SE2;

#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum Steer { L, S, R }

/// A signed segment: arcs in radians, straights in units of rho; negative means reverse.
#[derive(Clone, Copy, Debug)]
pub struct Seg { pub steer: Steer, pub len: f64 }

/// The three symmetries of the normalized goal (x, y, phi). Time-flip negates every
/// segment, reflect swaps L and R; "backwards" (applied separately) reads the word
/// right to left. Eleven base formulas under these four maps: 44 evaluations that
/// cover the 48 Reeds-Shepp words.
const SYMMETRIES: [(f64, f64, f64, bool, bool); 4] = [
    (1.0, 1.0, 1.0, false, false),   // base
    (-1.0, 1.0, -1.0, true, false),  // time-flip
    (1.0, -1.0, -1.0, false, true),  // reflect
    (-1.0, -1.0, 1.0, true, true),   // both
];

pub fn reeds_shepp_candidates(x: f64, y: f64, phi: f64) -> Vec<Vec<Seg>> {
    let mut out = Vec::new();
    for &(sx, sy, sp, flip, refl) in &SYMMETRIES {
        let sign = if flip { -1.0 } else { 1.0 };
        let word = |w: [Steer; 3], l: [f64; 3]| -> Vec<Seg> {
            w.iter().zip(l).map(|(&s, len)| Seg {
                steer: if refl { s.mirrored() } else { s },
                len: sign * len,
            }).collect()
```

```rust
        };
        if let Some((t, u, v)) = lp_sp_lp(sx * x, sy * y, sp * phi) {
            out.push(word([Steer::L, Steer::S, Steer::L], [t, u, v]));
        }
        if let Some((t, u, v)) = lp_sp_rp(sx * x, sy * y, sp * phi) {
            out.push(word([Steer::L, Steer::S, Steer::R], [t, u, v]));
        }
        // ... L+R-L (and its backwards reading), the two CCCC, the four CCSC, the CCSCC.
    }
    out
}

/// The shortest word in world units, or None (never, for rho > 0).
pub fn reeds_shepp_shortest_path(q0: &SE2, q1: &SE2, rho: f64) -> Option<(Vec<Seg>, f64)> {
    let local = q0.inverse() * *q1;           // goal in the start's frame
    reeds_shepp_candidates(local.x / rho, local.y / rho, local.theta)
        .into_iter()
        .map(|w| { let l = rho * w.iter().map(|s| s.len.abs()).sum::<f64>(); (w, l) })
        .min_by(|a, b| a.1.total_cmp(&b.1))
}

impl Steer {
    fn mirrored(self) -> Self { match self { Steer::L => Steer::R, Steer::R => Steer::L, Steer::S => Steer::S } }
}
```

RUST `crates/nonholonomic/src/car_grid.rs`

```rust
use std::collections::VecDeque;

#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum Action { Lp, Sp, Rp, Lm, Sm, Rm }

/// Integer weights: OPEN becomes an array of FIFO buckets indexed by cost (Choset's
/// "one-dimensional array with cost as the index"), so insertion is O(1).
#[derive(Clone, Copy, Debug)]
pub struct GridSearchCost { pub a: u32, pub b: u32, pub c: u32 }

pub struct Node<C> { pub q: C, pub parent: Option<usize>, pub action: Option<Action>, pub cost: u32 }

/// C Alg. 22, generic over the car model so the trailer is one type away.
pub fn car_grid_search<C: CarModel>(
    car: &C, q_start: C::Config, goal: &GoalRegion<C::Config>, step: f64,
    cells_per_dim: usize, cost: GridSearchCost, cs: &dyn collide::Collision<C::Config>,
    max_tree: usize,
) -> Result<Vec<(C::Config, Action)>, Failure> {
    let mut tree = vec![Node { q: q_start, parent: None, action: None, cost: 0 }];
    let mut open: Vec<VecDeque<usize>> = vec![VecDeque::from([0])];
    let mut occupied = vec![false; car.cells(cells_per_dim)];    // d3, or d? with a trailer
    let mut lowest = 0usize;
    while tree.len() < max_tree {
        // Line 3: the first node of the lowest non-empty bucket.
        while lowest < open.len() && open[lowest].is_empty() { lowest += 1; }
        let Some(id) = open.get_mut(lowest).and_then(|b| b.pop_front()) else { break };
        let q = tree[id].q.clone();
```

```
        // Line 4: the goal is tested when popped, which is what makes the path cost-optimal.
        if goal.contains(&q) { return Ok(path_to(&tree, id)); }
        // Lines 7-8: expand each cell once.
        let cell = car.cell_of(&q, cells_per_dim);
        if std::mem::replace(&mut occupied[cell], true) { continue; }
        for act in Action::ALL {
            let (q_new, sweep) = car.integrate(&q, act, step);
            if !sweep.iter().all(|s| cs.is_free(s)) { continue; }   // line 11
            let prev = tree[id].action;
            let c = tree[id].cost + cost.a
                + if prev.is_some_and(|p| p.steer() != act.steer()) { cost.b } else { 0 }
                + if prev.is_some_and(|p| p.dir() != act.dir()) { cost.c } else { 0 };
            if open.len() <= c as usize { open.resize_with(c as usize + 1, VecDeque::new); }
            open[c as usize].push_back(tree.len());
            lowest = lowest.min(c as usize);
            tree.push(Node { q: q_new, parent: Some(id), action: Some(act), cost: c });
        }
    }
    Err(Failure::TreeExhausted { size: tree.len() })
}
```

RUST `crates/kinodynamic/src/rrt.rs`

```rust
use rand::Rng;
use rand_pcg::Pcg64;

/// sec.7.5.1: planning with a simulator instead of a steer.
pub trait Propagate {
    type State: Clone;
    type Control: Clone;
    fn propagate(&self, x: &Self::State, u: &Self::Control, dt: f64) -> Self::State;  // RK4 inside
    fn sample_control(&self, rng: &mut Pcg64) -> Self::Control;
}

pub struct KinoRrt<P: Propagate> {
    pub sys: P,
    pub controls_per_extend: usize,
    pub dt: f64,
    pub substeps: usize,
    pub goal_bias: f64,
    nodes: Vec<(P::State, Option<(usize, P::Control)>)>,
}

impl<P: Propagate> KinoRrt<P> {
    pub fn plan(
        &mut self, x_start: P::State, goal: &dyn Fn(&P::State) -> bool, n: usize,
        sample: &dyn Fn(&mut Pcg64) -> P::State, metric: &dyn Fn(&P::State, &P::State) -> f64,
        is_free: &dyn Fn(&P::State) -> bool, rng: &mut Pcg64,
    ) -> Option<Vec<(P::Control, f64)>> {
        self.nodes = vec![(x_start, None)];
        for _ in 0..n {
            let x_rand = sample(rng);
            // The metric is the planner's only notion of "near" -- and its most sensitive knob.
```

```
            let near = (0..self.nodes.len())
                .min_by(|&a, &b| metric(&self.nodes[a].0, &x_rand).total_cmp(&metric(&self.nodes[b].0, &x_rand)))?;
            let mut best: Option<(P::State, P::Control, f64)> = None;
            for _ in 0..self.controls_per_extend {
                let u = self.sys.sample_control(rng);
                let h = self.dt / self.substeps as f64;
                let mut x = self.nodes[near].0.clone();
                let mut free = true;
                for _ in 0..self.substeps {
                    x = self.sys.propagate(&x, &u, h);
                    if !is_free(&x) { free = false; break; }
                }
                let d = metric(&x, &x_rand);
                if free && best.as_ref().is_none_or(|b| d < b.2) { best = Some((x, u, d)); }
            }
            let Some((x_new, u, _)) = best else { continue };
            self.nodes.push((x_new.clone(), Some((near, u))));
            if goal(&x_new) { return Some(self.controls_to(self.nodes.len() - 1)); }
        }
        None
    }
}
```

RUST **crates/nonholonomic/src/reduction.rs**

```
use nalgebra::{SMatrix, SVector};

/// Theorem 12.4.7: V is decoupling iff V in span(Y) and ?_V V in span(Y).
pub fn is_decoupling<const Q: usize>(
    v: &dyn VectorField<Q>, inputs: &[&dyn VectorField<Q>],
    model: &dynamics::Model<Q>, q: &SVector<f64, Q>, tol: f64,
) -> bool {
    let y: Vec<SVector<f64, Q>> = inputs.iter().map(|f| f.eval(q)).collect();
    let vq = v.eval(q);
    // ?_V V = (?V/?q) V + M-1 (V^T Gamma V), Chapter 17's Christoffel symbols, Choset's convention.
    let nabla = v.jacobian(q) * vq + model.mass_matrix(q).try_inverse().unwrap()
        * model.christoffel_form(q, &vq, &vq);
    relative_residual(&vq, &y) < tol && relative_residual(&nabla, &y) < tol
}

/// Reach 3R with u_3 = 0: rotation of link 3 about its center of percussion with respect
/// to joint 3, at k = (I_3 + m_3r_32)/(m_3r_3) -- 0.205 m for Choset's Table 12.1 robot.
pub fn rotate_about_percussion(arm: &PlanarArm<3>) -> impl VectorField<3> + '_ {
    let k = arm.percussion_distance();
    move |q: &SVector<f64, 3>| {
        let a = q.sum();                                  // link 3's absolute heading
        let j: SMatrix<f64, 2, 2> = arm.joint3_jacobian(q); // ?(x_J3, y_J3)/?(q_1, q_2)
        let r = j.try_inverse().unwrap() * SVector::<f64, 2>::new(k * a.sin(), -k * a.cos());
        SVector::<f64, 3>::new(r[0], r[1], 1.0 - r[0] - r[1])
    }
}
```

21.5.1 The worked example, printed

RUST crates/nonholonomic/examples/dubins_check.rs

```rust
fn main() {
    let q0 = SE2::new(0.0, 0.0, 0.0);
    let q1 = SE2::new(3.0, 3.0, std::f64::consts::FRAC_PI_2);
    for p in dubins_all(&q0, &q1, 1.0) {
        println!("{:?}  t={:.6} p={:.6} q={:.6}  len={:.6}", p.word, p.segs[0], p.segs[1],
    p.segs[2], p.len());
    }
    let best = dubins_shortest_path(&q0, &q1, 1.0).unwrap();
    let end = best.integrate(&q0);
    println!("integrate(LSL) -> ({:.6}, {:.6}, {:.6})", end.x, end.y, end.theta);
    let (rs, len) = reeds_shepp_shortest_path(&q0, &q1, 1.0).unwrap();
    println!("reeds-shepp: {}  len={:.6}", word_string(&rs), len);
}
```

```
LSL  t=0.785398 p=2.828427 q=0.785398  len=4.399223
LSR  t=0.927295 p=4.000000 q=5.639684  len=10.566979
RSL  t=5.639684 p=4.000000 q=0.927295  len=10.566979
LRL  t=3.141593 p=4.712389 q=3.141593  len=10.995574
RSR  t=5.497787 p=5.656854 q=5.497787  len=16.652429
integrate(LSL) -> (3.000000, 3.000000, 1.570796)
reeds-shepp: L+S+L+  len=4.399223
```

```
t=0.25  y=(0.250000, 0.156250)  theta=0.844154 v=1.505199 omega=1.324138 kappa=0.879709
   phi(L=1)=0.721491
t=0.50  y=(0.500000, 0.500000)  theta=0.982794 v=1.802776 omega=0.000000 kappa=0.000000
   phi(L=1)=0.000000
```

```
percussion distance k = 0.205000 m
plan: T- 0.240 | R+ 1.100 | T+ 0.390 | R- 0.200   (3 switches)
time-optimal, rest to rest: 0.151 + 0.157 + 0.230 + 0.050 = 0.589 s
max |tau3| along every segment: 0 (to finite-difference accuracy); one actuator saturated
    at every node
```

`#[test] fn reproduces_micro_example()` asserts the first two blocks to 10^{-6}; `tests/words_integrate.rs` draws 400 seeded pose pairs and asserts that every Dubins and Reeds–Shepp word integrates to its endpoint within 10^{-9} and that Euclidean $\leq$ Reeds–Shepp $\leq$ Dubins length. The dead-joint example cannot reproduce Choset's 0.890 s, because his Fig. 12.33 path is drawn but not given numerically; it reproduces his *structure* — four decoupling segments, one actuator always saturated — on the query of w21.6.

21.6 Putting it together

The Integration lab is `park_six_ways` live: Hitch with its 2.5 m trailer, from $(5.5, 1.2, 0)$ into bay 3, every planner of the chapter, one scoreboard (seed 21; the trailer is simulated along each path, or produced by the method itself for the trailer-axle flat planner and Algorithm 22's trailer extension).

method	length (m)	cusps	max $\lvert\theta_1\rvert$ (rad)	clearance (m)	ends
Reeds–Shepp (obstacle-blind)	7.37	1	0.62	0.10	exactly on the goal
flat-output quintic, trailer axle	9.19	0	1.03	collides	exactly; $\lvert\phi\rvert$ up to 1.39 rad
chained-form sinusoids	27.89	3	2.39	collides	exactly, in chained coordinates
CAR GRID SEARCH (Alg. 22, trailer grid)	7.20	0	0.63	0.11	inside ±0.5 m, ±0.2 rad
hybrid A* (car), trailer towed	7.37	1	0.62	0.10	exactly (its RS shot)
forward-propagation RRT, dynamic car	6.85	0	0.66	0.13	inside 0.8 m, 0.35 rad

Read it as a summary of the chapter. Here the query is open enough that hybrid A*'s first shot is already free, so it *is* the Reeds–Shepp path; Algorithm 22 finds a slightly shorter path only because it is allowed to stop half a metre short. The flat planner's single smooth curve is the only one built around the trailer, and it is also the one that asks for more steering than Hitch has. The sinusoids are exact in the coordinates where they are exact and absurd everywhere else: the trailer swings to 2.39 rad, a jackknife, because the chained form of the four-state car simply does not contain it. The forward-propagated tree is the shortest because its goal is a region — the honest comparison is the column on the right.

Two more experiments belong to the lab. **The failure button**: shrink the bay until Hitch is only a little shorter than it and run Algorithm 22 with $c = 0$; the number of cusps in the solution grows as the slack shrinks, as Choset's $\Theta(1/\epsilon^2)$ theorem says it must, and no planner in this chapter can beat that — it is a property of the car, paid in Chapter 20's currency of brackets. **The randomized pillar**: in w21.1 turn obstacles on and re-roll the seed; a third pillar lands in the open floor, the obstacle-blind methods start reporting collisions, and Algorithm 22 routes around it. Chapter 23 reuses the trailer flat-output planner and the lattice for its jackknife tour, and Chapter 22 learns hybrid A*'s heuristic from data.

The sister book's motion-planning chapter previewed the Dubins dial and sketched hybrid A* for a car under uncertainty; this chapter is the derivation and the tested implementation behind both.

21.7 Exercises

1 FOUNDATION •• **RSL from LSL, and when CCC can never win**

Derive the RSL closed form from LSL by reflection and reversal, and show where $p^2 < 0$ makes it infeasible. Then prove that for $\|q_1 - q_0\|_{xy} \geq 4\rho$ no CCC word is optimal — note that CCC words can still be *feasible* there.

HINTS

<>Reflection $y \to -y$ maps $L \leftrightarrow R$ and negates headings; reading a path backward from the goal swaps the roles of α and β. Compose the two and LSL's external tangent becomes the internal tangent of RSL.</> <>For two circles of radius 1 turning in opposite directions the internal tangent has length p with $p^2 + 4 = \|c_1 - c_0\|^2$: infeasible exactly when the circles overlap, $\|c_1 - c_0\| < 2$. A CCC word needs its outer centers within 4ρ, so the poses themselves can be up to 6ρ apart — feasibility is not the issue.</> <>Two facts finish it: in an optimal CCC path the middle arc exceeds $\pi\rho$ (Dubins' lemma), and with $\|q_1 - q_0\| \geq 4\rho$ the outer centers are at least 2ρ apart, so the same-direction CSC word on the *same* outer circles exists; compare the two lengths arc by arc. Then test your proof against the library: among the chapter check's 20,000 seeded goals, none with $d \geq 4\rho$ has an optimal CCC word.</>

2 FOUNDATION •• **Chained forms are flat**

Prove that every chained-form system is flat with (x_1, x_n) as flat outputs (Choset Problem 25) and give φ and ψ_u explicitly for $n = 4$. Relate x_4 to Hitch's trailer flat output when the car-with-trailer is put in chained form.

HINTS

<>Wherever $u_1 = \dot{x}_1 \neq 0$, $\dot{x}_n = x_{n-1}u_1$ gives $x_{n-1} = \dot{x}_n/\dot{x}_1$; repeat down the chain.</> <>For $n = 4$: $x_3 = \dot{y}_2/\dot{y}_1$, $x_2 = \dot{x}_3/\dot{y}_1 = (\ddot{y}_2\dot{y}_1 - \dot{y}_2\ddot{y}_1)/\dot{y}_1^3$, $u_1 = \dot{y}_1$, $u_2 = \dot{x}_2$ — three derivatives of y_2.</> <>In Murray–Sastry coordinates for the four-state car, $(z_1, z_4) = (x', y')$ is the rear axle in the rotated chart — the car's flat output. With a trailer the last chained coordinates are the trailer axle's, matching the trailer-axle flat output of §12.5.5; both lose meaning at $v = 0$ and at the coordinate singularity.</>

3 CONCEPTUAL • **Predict the word, then drag**

PREDICT FIRST

In the Reeds–Shepp Atlas with $\rho = 1$, start $(0, 0, 0)$ and goal $(2, -1, 0)$: which word is the shortest Dubins path, and does allowing cusps (Reeds–Shepp) shorten it?

- ✓ RSL, length 2.287; Reeds–Shepp returns the same path
  A lane change to the right: arc right by atan(3/4) = 0.6435 rad, straight 1, arc left by the same angle — 2·atan(3/4) + 1 = 2.287. Every reversing word must cover at least the same ground plus the wasted motion of a cusp at this range.
- RSR, length about 2.2
  An RSR path ends heading the way its last right arc leaves it; to finish with heading 0 after turning right it must turn back left, so RSR would need a near-full loop — the library gives RSR = 8.52.
- RSL for Dubins, but Reeds–Shepp is shorter with one cusp
  The second-best Reeds–Shepp word here is $R^{+}L^{-}R^{-}L^{+}$ at 3.558; cusps pay off when the goal is close and beside or behind the start, not two radii ahead.

Now drag the goal to $(0.5, -1, 0)$ — beside the start rather than ahead of it. Dubins must loop (LSL, 7.401) while Reeds–Shepp switches to a $C|CC|C$ word of length 2.355: a lane change that short is made by backing up.

WORK IT OUT

Length of the shortest path to $(2, -1, 0)$ at $\rho = 1$

Answer. 2.287

4 **CONCEPTUAL** •• **Make the sinusoids swing and the flat planner fail**

In w21.1 with the trailer on, place the bay so that the sinusoid planner's trailer swing exceeds $|\theta_1| = 1.2$ rad while the flat planner stays under 0.6; then make the flat planner fail (its steering demand over 0.6 rad) by lowering the end-speed slider. Say where control constraints enter each method.

HINTS

<>The sinusoid planner's amplitude grows like $(|\Delta|\,k!\,(4\pi)^k)^{1/(k+1)}$; any bay that needs a large lateral correction y' drives big oscillations in x', and the passive trailer follows them.</> <>The flat planner's curvature scales like $1/(\text{end speed})^2$ near the ends: a slow start means a sharp turn in little arc length. Control bounds enter flatness only through such free parameters — or through an optimizer over extra coefficients.</> <>Reeds–Shepp and Algorithm 22 respect the steering limit by construction (full lock is the limit), and violate *smoothness* instead: the curvature jumps at every junction.</>

5 PRACTICAL •• CAR GRID SEARCH with a trailer, and the jackknife guard

Implement the trailer extension of `car_grid_search` with a lookup table of $\Delta\theta_1$ per primitive (indexed by the hitch angle at the start of the motion, Choset §12.5.6), check it against integration on three seeded queries, and add the jackknife guard $|\theta_1| < \pi/2 - \delta$ as pruning. Measure the expansions it saves.

HINTS

<>Over one primitive of arc length Δ the hitch angle obeys $\theta_1' = \kappa - \sin\theta_1/d$ per unit arc length, with $\kappa \in \{1/\rho, 0, -1/\rho\}$ and the sign of Δ for reverse: the change depends only on θ_1 and the action, so a 1-D table per action suffices.</> <>Interpolate the table linearly in θ_1; with 256 entries the error against RK4 is far below the grid's resolution in θ_1.</> <>The library's `CarGridSearch` takes a `jackknife` option; reversing primitives are the ones that get pruned, and the saving is largest on reverse-parking queries.</>

6 PRACTICAL ••• Kinodynamic RRT* for Hitch's dynamic model (stretch)

Implement `KinoRrtStar` for Hitch's dynamic model linearized about rest (Webb and van den Berg: fixed-final-state, free-final-time LQR edges), compare its cost curve with `KinoRrt` (best of 8) over 20 seeds, then swap the LQR edge for `steer_flat` and report which assumption each connection violates.

HINTS

<>About rest the car's linearization is uncontrollable in the lateral direction (Chapter 20: the sideways direction is a bracket, not a first-order direction), so the LQR Gramian $G(\tau)$ is singular — linearize about a moving nominal instead, or accept edges only where G is well conditioned.</> <>The library's `doubleIntegratorSteer` shows the closed form for the controllable case; check your edge by integrating its control and landing on the target.</> <>`steer_flat` is exact for the nonlinear car but neither optimal nor constraint-aware: RRT*'s rewiring then compares costs that are not c^*, and its optimality argument no longer applies.</>

21.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

Chapter 12, §12.4.2 and §12.5 (Brockett's sinusoids, chained forms, gradient steering, flatness, cars and trailers, Algorithm 22, kinematic reductions, Table 12.1) and §7.5.1 (control-based planning), followed section by section. The Souè res–Laumond synthesis and the Sussmann–Tang reduction of the Reeds–Shepp list are its refs. 395 and 402. `link`

Dubins, L. E. (1957). On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents. *American Journal of Mathematics 79(3)*.

The six-word result for the forward-only car: the CSC/CCC words implemented in dubins.ts and pinned by the micro-example. `doi:10.2307/2372560`

Reeds, J. A. and Shepp, L. A. (1990). Optimal Paths for a Car That Goes Both Forwards and Backwards. *Pacific Journal of Mathematics 145(2)*.

The forty-eight words and the §8 base formulas that reeds-shepp.ts evaluates under the time-flip and reflect symmetries. `doi:10.2140/pjm.1990.145.367`

Murray, R. M. and Sastry, S. S. (1993). Nonholonomic Motion Planning: Steering Using Sinusoids. *IEEE Transactions on Automatic Control 38(5)*.

Chained forms, the sinusoid ladder of eq. (12.31), and the chained coordinates of the kinematic car used by steerCarSinusoids. `doi:10.1109/9.277235`

Fliess, M., Lé vine, J., Martin, P., and Rouchon, P. (1995). Flatness and Defect of Non-linear Systems: Introductory Theory and Examples. *International Journal of Control 61(6)*.
Differential flatness, with the car and the car with trailers among its examples — Choset's refs. 152–153 and the basis of flat.ts. doi:10.1080/00207179508921959

Barraquand, J. and Latombe, J.-C. (1993). Nonholonomic Multibody Mobile Robots: Controllability and Motion Planning in the Presence of Obstacles. *Algorithmica 10*.
The grid search over full-lock primitives with a cusp-weighted cost that Choset presents as Algorithm 22, including the trailer extension. doi:10.1007/BF01891837

Laumond, J.-P., Jacobs, P. E., Taïx, M., and Murray, R. M. (1994). A Motion Planner for Nonholonomic Mobile Robots. *IEEE Transactions on Robotics and Automation 10(5)*.
The omnidirectional-to-nonholonomic path transformation by recursive Reeds–Shepp bisection (pathTransform) and the topological property behind its completeness. doi:10.1109/70.326564

LaValle, S. M. and Kuffner, J. J. (2001). Randomized Kinodynamic Planning. *International Journal of Robotics Research 20(5)*.
Forward-propagation RRTs for systems with differential constraints — the KinoRrt of this chapter and the metric-sensitivity discussion. doi:10.1177/02783640122067453

Bullo, F. and Lynch, K. M. (2001). Kinematic Controllability for Decoupled Trajectory Planning in Underactuated Mechanical Systems. *IEEE Transactions on Robotics and Automation 17(4)*.
Decoupling vector fields, Theorem 12.4.7 and kinematic controllability, with the 3R arm whose third joint is unactuated. doi:10.1109/70.954753

Pivtoraiko, M., Knepper, R. A., and Kelly, A. (2009). Differentially Constrained Mobile Robot Motion Planning in State Lattices. *Journal of Field Robotics 26(3)*.
State lattices with lattice-consistent primitives: delta 1 of the chapter's three deltas from Algorithm 22. doi:10.1002/rob.20285

Dolgov, D., Thrun, S., Montemerlo, M., and Diebel, J. (2010). Path Planning for Autonomous Vehicles in Unknown Semi-structured Environments. *International Journal of Robotics Research 29(5)*.
Hybrid A*: continuous states in a discrete grid, analytic Reeds–Shepp expansions and the dual heuristic — deltas 2 and 3. doi:10.1177/0278364909359210

Webb, D. J. and van den Berg, J. (2013). Kinodynamic RRT*: Asymptotically Optimal Motion Planning for Robots with Linear Dynamics. *IEEE International Conference on Robotics and Automation (preprint arXiv:1205.5088)*.
The fixed-final-state, free-final-time optimal edge that turns RRT* into a kinodynamic planner; the double-integrator closed form in kinodynamic.ts. link

Karaman, S. and Frazzoli, E. (2011). Sampling-based Algorithms for Optimal Motion Planning. *International Journal of Robotics Research 30(7)*.
RRT *and its asymptotic optimality, which Chapter 13 proves and Kinodynamic RRT* inherits once the steer is exact. doi:10.1177/0278364911406761

Part VI

Frontiers and Integration

CHAPTER 22

Learning and Planning

FRONTIERS AND INTEGRATION · Advanced · 85 min

> *The framework presented in this section enables a rigorous treatment of asymmetric reachability, nonmanifold configuration spaces, and sampling from arbitrary distributions.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), §7.4.3*

Every planner in Parts II through V was designed by hand. Uniform samples, Euclidean heuristics, Gaussian and bridge biases a person reasoned out from the geometry of narrow passages. This chapter asks the question every planning group asks in 2026, and tries to answer it without hype: where does machine learning belong in a motion planner?

The answer that organizes everything is one sentence. **Learning replaces the *proposal*, never the *verifier*.** A learned sampler changes *where* a PRM looks, never *whether* an edge is collision checked. A learned heuristic changes the *order* in which A* expands nodes, never the test that declares a path valid. A neural planner that emits waypoints is only as complete as the classical repair behind it, and a diffusion model that emits whole trajectories produces candidates that a checker still has to accept. In every case the guarantee — completeness, optimality, collision freedom — is *inherited from the verifier*, the speed is *bought from the learner*, and the exchange rate is a number you can measure.

Choset's abstract path-tiling theorem (§7.4.3, which the epigraph introduces) never needed uniform samples. We carry one constant λ through its proof, then measure what the learner actually buys on Reach's Workbench — less than the abstracts suggest, and more than nothing.

◎ AFTER THIS CHAPTER YOU CAN

- State the one rule that organizes every use of learning in this chapter — learning replaces the proposal, never the verifier — and say for each of four learned components what the proposal is and what the verifier is
- Prove that a learned sampler mixed with a uniform floor of weight λ keeps Choset's probabilistic-completeness bound with p replaced by λp, and that $\lambda = 0$ voids it
- Show that maximum-likelihood training of a sampler is minimization of D_KL(p || $\hat{p}$_θ), *and compute the four-cell example by hand:* D_KL($\hat{p}$ || u) = *0.2442 nats,* D_KL(p || $\hat{p}$) = 0.0119, D_KL(p* || u) = 0.3162
- Prove that A *with reopening and an* ε*-admissible heuristic returns a path of cost at most* $(1 + \varepsilon)C$, and audit a three-node heuristic to $\varepsilon = 0.1333$ and a bound of 5.667
- Explain why a neural planner with a classical fallback inherits the fallback's guarantee and nothing else, and why a diffusion planner is a proposal that still needs a collision checker
- Read a benchmark honestly: fixed budgets in collision checks, held-out worlds from a different generator, wall-clock including inference, and training time amortized over the queries that pay for it

ASSUMED

- A* with admissibility, consistency and reopening; backward Dijkstra as a cost-to-go oracle (Chapter 6)
- PRM, the Sampler and Steer traits, the Hybrid mixture sampler, and the three completeness analyses of Choset §7.4 (Chapter 11)
- RRT-Connect as a single-query planner (Chapter 12); the cost functional of trajectory optimization (Chapter 19)
- Reach on the Workbench and its torus of configurations (Chapters 2, 4 and 5)

22.1 The problem: a sampler that learned the bench

Chapter 11's PRM draws configurations uniformly from Reach's torus. On a cluttered Workbench most of those samples land where the eventual path never goes, and each costs a collision check plus a dozen more for every edge tried. Someone who has watched a thousand roadmaps grow knows roughly where paths go; Chapter 11's Gaussian, bridge and OBPRM samplers were three attempts to write that knowledge down as a rule.

The modern move is not to write it down at all. Generate six hundred random

Workbenches, run Chapter 11's planner on three queries in each, record every node of every path it found, and train a small network to predict, from the table's occupancy raster and the query's two endpoints, *where on the torus the path nodes will be*. Then sample from the network. The data is real: every recorded node passed Chapter 2's collision checker, every path was certified edge by edge. Nothing in the planner changes except the line that draws the next sample.

Here is what that buys, measured on thirty solvable queries on fresh Workbenches from the same random generator, every planner stopped at a fixed number of collision checks. At a budget of 100 checks the learned proposal — mixed with ten percent uniform, for a reason the mathematics will make non-negotiable — solves 60% of the queries and the uniform PRM solves 47%. At 400 checks the order *reverses*: the learned sampler solves 77% and uniform 90%. At 12,800 checks both solve every one. The paths the learned sampler finds are shorter (1.28 times the best known path against uniform's 1.33), and every millisecond of that is spent on a network that took 241 seconds of data generation and training before the first query, which no realistic number of queries on this suite pays back.

That is the chapter in miniature: the learner moved where the samples fell, could not change what the planner was able to find, and whether the move paid depends on budget, world and bookkeeping.

22.2 Building intuition: the scout and the surveyor

The chapter's picture in one line: *the learner is a scout who says "look here"; the planner is the surveyor who still measures every edge.* Each widget below puts a different scout in front of the same surveyor. Read them as a set — they are four instances of one design and one honest scoreboard.

22.2.1 Where the samples fall

Two Chapter 11 PRMs grow side by side on the same randomized Workbench, seed and query. The left pane draws uniformly; the middle pane draws from the learned proposal $\hat{p}_\theta(q \mid \mathcal{E}, q_{\text{start}}, q_{\text{goal}})$ — the grey heat under the amber C-obstacle raster — mixed with uniform at weight λ; the right pane is the Workbench.

> **INTERACTIVE · w22.1**
>
> **Learned Sampler vs Uniform**
>
> A learned sampler changes where a PRM looks, not what it can find: with $\lambda > 0$ both panes connect every solvable query, the learner moves the knee of the success curve — and on a held-out world it moves it the wrong way.
>
> Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch22-learning-and-planning#w22.1`

On the default world the learned pane joins start and goal after about half the collision checks the uniform pane spends: the heat sits on the band of the torus the path actually uses, and the samples follow it. Now press **held-out generator**. The world is replaced by one built from long thin bars at arbitrary angles — a family of shapes the training generator never produced — and on this seed the learned pane needs more than twice the checks uniform does. The heat still lands in confident blobs; they are just in the wrong place, because the network learned what blocks do to Reach's torus and is now looking at bars. Three things to notice, in order of importance.

Both panes connect. Slide the sample budget up and both roadmaps join on every solvable query you try, on both generators, at every λ the slider allows. The learner moves *where* the planner looks. It does not move what the planner can find — that is the content of Derivation 1, and the reason the λ slider stops at 0.02 instead of reaching zero.

The currency is collision checks. On Reach's torus a query is usually joined within a handful of nodes, so "samples until the join" hides the difference. The check count — the book's cost unit since Chapter 11 — shows it. The success-at-budget curves below the canvas accumulate over further seeds in the background, and they are the honest summary: on some worlds the learner wins big, on others it loses, and the curve is the average of both.

The bar chart is the chapter's first formula. It prints $D_{\mathrm{KL}}(\hat{p} \,\|\, u)$ on the four quadrants of the torus, live: how *concentrated* the proposal is — not whether the concentration is in the right place, which takes a second divergence. The four-cell example computes both by hand.

22.2.2 A heuristic that knows too much

Chapter 6's A* on a 24×24 lattice of Reach's torus, with the admissible octile distance, a learned cost-to-go $\hat{h}_\theta$, or $\hat{h}_\theta/(1+\epsilon_{val})$. Left: explored set (blue), path (purple), a red mark wherever $\hat{h}$ exceeds the true cost-to-go h^* from backward Dijkstra. Right: every reachable cell at $(h^*(v), \hat{h}(v))$ against the diagonal.

> INTERACTIVE · w22.2
>
> **Heuristic Calibration**
>
> A more informed heuristic is not a better one: the inadmissible cloud expands fewer nodes and returns a longer path — and the ε by which it overestimates is the exact bound on what you lose.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch22-learning-and-planning#w22.2

On the default world the learned heuristic does exactly what the folklore says a better heuristic does: it expands fewer nodes than octile. And it returns a *longer path* — the cost-ratio tile reads above one. The scatter shows why: a cloud of red

points above the diagonal, cells where the net guessed high. The measured ϵ is the worst of those ratios on this world, and the fourth tile prints the guarantee $(1 + \epsilon)\, C^*$ that Derivation 3 proves the realized cost can never exceed. It never does — on this world, on any world you re-roll to, at any scale.

Then try the other two settings. Slide the scale above 1 and even the octile heuristic becomes inadmissible (that is weighted A*, which Chapter 6 introduced): expansions fall, the path stays at or above C^*, and the bound line rises with the scale. Switch to the deflated heuristic and the red cloud vanishes, the path becomes optimal again — and the expansion count balloons to nearly three times octile's. Dividing by $1 + \epsilon_{val}$ buys admissibility on the validation worlds by throwing away most of what the net knew.

Across many worlds the summary is less flattering: on the audit worlds the learned heuristic and octile expand almost the same number of nodes (ratio 0.994). Octile is already good on this lattice — its mean ratio to h^* on the validation worlds is 0.917, against the net's 0.959. What does *not* vary is the bound: the learned heuristic cannot hurt you by more than its measured ϵ.

22.2.3 The net that hops

An MPNet-style policy proposes Reach's next configuration a hop $\eta = 0.35$ rad away. Each hop goes to the verifier — Chapter 11's `Steer`, fulfilled by Chapter 2's *sound* swept check — and is drawn blue if certified, red and dashed if not. Rejected stretches are bridged by Chapter 12's RRT-Connect (the thin blue mini-tree); if local repairs run out, RRT-Connect runs on the whole original query.

> **INTERACTIVE · w22.3**
>
> **Neural Planner Autopsy**
>
> The net did not learn to plan; it learned the training distribution. The repair planner behind it is the guarantee, and Derivation 4 is why the demo is complete at all.
>
> Run this simulation in the web edition: `shannon-dynamics.github.io/PRM/chapters/ch22-learning-and-planning#w22.3`

On the training tab most of the hops pass. The policy learned the shape of paths on block-shaped Workbenches, and on this one it walks most of the way by itself; one short repair splices over the stretch where it clipped a block. Switch to the held-out tab: fewer than two-thirds of the hops pass, and the tile that splits the path into *by net* and *by repair* tips the other way — the repair planner supplies most of the length. Now turn **repair off**. The tile labelled *net alone* reads *failed* on both tabs. A path is only a path if every hop is certified and the last one reaches the goal; one clipped corner is enough to have nothing. The network did not learn to plan; it learned the training distribution of paths — valuable, since on a good day it proposes most of the answer — and with the repair planner behind it the composition is exactly as complete as RRT-Connect (Derivation 4).

22.2.4 Noise into paths

Diffusion models generate by running a noising process backward. The smallest example: a trajectory $\tau \in \mathbb{R}^{16}$ is a configuration per time step, pinned to 0 at both ends; a slab blocks $|q| < 0.3$ during steps 5 through 10; demonstrations swing around it, half above and half below. Thirty-two samples start as pure noise and are denoised for $K = 40$ steps by a small net, then checked: purple is clear, red went through the slab.

INTERACTIVE · w22.4

Denoising Trajectory

Generative models produce plausible trajectories, not collision-free ones: the valid fraction is high and never one, so Chapter 2's checker runs on every sample.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch22-learning-and-planning#w22.4

Scrub the step slider slowly from 0. For the first half of the steps nothing recognizable happens; then the swing appears, *both* swings, as the samples commit to going over or under. At step 40 a good fraction are red: plausible-looking trajectories that dip into the slab for a time step or two. Turn on cost guidance — each step also descends the gradient of a smooth cost that penalizes coming near the slab — and the red fraction drops a great deal. It does not reach zero, on any seed you try. A soft penalty is not a hard constraint, which is why Chapter 2's checker runs on every sample and why a diffusion planner is a *proposal* in exactly the sense of the sampler above.

22.2.5 The board

Seven planners — four hand-designed PRM samplers, the learned sampler at $\lambda = 0.1$, RRT-Connect, and the MPNet-style planner — on thirty solvable queries from each generator, same world, query and seed, stopped at 12,800 collision checks; smaller budgets are read off the same runs.

INTERACTIVE · w22.5

Honest Benchmark Board

A benchmark without inference time and without held-out worlds is not a benchmark: the learned rows win on one side of the table and lose on the other, and training is paid for only by enough queries on similar worlds.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch22-learning-and-planning#w22.5

Toggle the suite and move the budget. The learned rows (tinted orange) win at the smallest budgets in distribution and then fall behind; their first paths

are consistently shorter; their wall-clock is *worse*. RRT-Connect, which nobody trained, beats every PRM row at almost every budget on both suites. Drag the amortization slider to 10^4 queries and the learned rows still do not overtake uniform on wall-clock: they were never faster per query. *Putting it together* reads this board line by line.

> **THE ONE SENTENCE VERSION**
>
> A learned component in a motion planner is a proposal — where to sample, which node to expand, which waypoint to try, which trajectory to check — and the planner's guarantees survive exactly when a classical verifier still certifies every candidate and a classical fallback still runs on the original query; the learner's contribution is then a change in cost, which must be measured at fixed budgets, on worlds it was not trained on, with its own inference and training time on the bill.

22.3 The mathematics

$\mathcal{E}$		environment encoding fed to a net — here an 8×8 occupancy raster of the Workbench, each cell the fraction of four probe points inside an obstacle
$u(q) = 1/\mu(\mathcal{Q})$		the uniform density on $\mathcal{Q}$ — Chapter 11's default proposal Choset §7.4.1
$\hat{p}_\theta(q \mid \mathcal{E})$, $p^*(q \mid \mathcal{E})$		the learned proposal; the target — the empirical distribution of nodes on Chapter 11 roadmap paths
$p_\lambda = (1-\lambda)\hat{p}_\theta + \lambda u$		the bounded-below mixture, $\lambda \in (0, 1]$
$D_{\mathrm{KL}}(p \,\|\, q)$ $\sum_i p_i \ln(p_i/q_i)$	=	Kullback–Leibler divergence on a partition, in nats (bits in parentheses); H(p) = −Σ p_i ln p_i is the entropy
$h^*(v)$, $\hat{h}_\theta(v)$, ϵ		true cost-to-go; learned heuristic; the smallest ε with $\hat{h} \le (1+\varepsilon)\, h^*$ on the audited nodes
π_θ, η		a waypoint policy (neural planner); its hop length on T^2
β_k, $\bar{\alpha}_k$, $\epsilon_\theta(\tau^k, k, \mathcal{E})$, $J(\tau)$		diffusion noise schedule and its cumulative product; the noise predictor over a trajectory $\tau \in \mathcal{Q}^{\mathrm{T}}$; the guidance cost Chapter 19 for J

22.3.1 Proposals and verifiers

Everything in this chapter is an instance of two definitions and one composition.

DEFINITION 22.1 — SAMPLING PROPOSAL

A **sampling proposal** is any probability density on $\boldsymbol{Q}$ from which a sampling-based planner draws candidate configurations. The **verifier** — the node collision test and the local planner Δ (`Steer`) that certifies edges — is not part of the proposal and is unchanged by it. Chapter 11's uniform, Gaussian, bridge and OBPRM samplers are proposals; so is the learned $\hat{p}_\theta(q \mid \mathcal{E}, q_{\text{start}}, q_{\text{goal}})$.

DEFINITION 22.2 — COMPLETENESS-PRESERVING PROPOSAL

A proposal p is **completeness-preserving** if there is a fixed $\lambda > 0$ with $p(q) \geq \lambda\, u(q)$ for every $q \in \boldsymbol{Q}$. Every mixture $p_\lambda = (1-\lambda)\hat{p}_\theta + \lambda u$ is one, whatever $\hat{p}_\theta$ is. A raw network output generally is not: decoders emit exact zeros, and a softmax that is technically positive everywhere can put 10^{-6} of uniform's mass on the one corridor that matters.

The constant in Definition 22.2 is the constant in the failure bound. The shipped sampler's raw softmax leaves 188 of 3,456 cells, across 24 audited worlds, below a thousandth of the uniform mass: positive, and useless as a guarantee. The mixture replaces that unknowable minimum by λ.

22.3.2 Completeness survives a floor

Recall Chapter 11's statement of Choset's Theorem 7.4.3: if PRM with local planner relation R and sampling measure μ is probabilistically complete, there are ℓ sets $A_1, \ldots, A_\ell \subset \boldsymbol{Q}_{\text{free}}$, each of positive measure, such that *any* choice $x_j \in A_j$ chains into a path $x\ R\ x_1\ R \cdots R\ x_\ell\ R\ y$; and with $p = \min_j \mu(A_j)$ the failure probability after n samples is at most $\ell(1-p)^n \leq \ell e^{-pn}$. Choset introduces the section by saying the framework covers "sampling from arbitrary distributions", and the proof bears it out: the only property of μ it uses is a *lower bound* on the mass of each tile.

$$\Pr\left[\text{FAILURE after } n \text{ samples from } p_\lambda\right] \;\leq\; \ell\,(1-\lambda\, p_u)^n \;\leq\; \ell\, e^{-\lambda\, p_u\, n}, \qquad p_u = \min_j u(A_j).$$

DERIVATION Derivation 1 — probabilistic completeness survives any bounded-below proposal

$\Pr[\text{FAILURE}] \leq \ell\,(1-\lambda p_u)^n \leq \ell\, e^{-\lambda p_u n}$

Step 1 — take the uniform tiles. Suppose uniform PRM with relation R is probabilistically complete on the query (x, y) (Chapter 11 proved this

for the straight-line planner on any space with a clearance-positive path). Theorem 7.4.3 supplies tiles $A_1, \ldots, A_\ell$ with $p_u = \min_j u(A_j) > 0$ and the chaining property. The chaining property is a statement about R alone — it does not mention the sampling measure.

Step 2 — failure needs an empty tile. If every A_j receives at least one of the n roadmap nodes, the roadmap contains a path from x to y (the nodes chain by Step 1, and PRM's connection rule finds the chain under the same hypotheses as in Chapter 11). So $\{\text{FAILURE}\} \subseteq \bigcup_j \{A_j \text{ empty}\}$.

Step 3 — each draw hits a tile with probability at least λp_u. Under the mixture, $p_\lambda(A_j) = (1-\lambda)\hat{p}_\theta(A_j) + \lambda u(A_j) \geq \lambda u(A_j) \geq \lambda p_u$, because $\hat{p}_\theta(A_j) \geq 0$ — the only thing we know about the network. The learned sampler is conditioned on the query, which is fixed before sampling begins, so the n draws are independent and identically distributed. Rejection of colliding draws only helps: the tiles lie in Q_{free}, so conditioning on acceptance can only raise their probability.

Step 4 — union bound. $\Pr[A_j \text{ empty}] \leq (1-\lambda p_u)^n$, and $\Pr[\text{FAILURE}] \leq \sum_j \Pr[A_j \text{ empty}] \leq \ell(1-\lambda p_u)^n \leq \ell e^{-\lambda p_u n}$, using $1 - t \leq e^{-t}$.

Step 5 — what was used, and what $\lambda = 0$ does. Only the lower bound $p_\lambda \geq \lambda u$ was used; the network could be anything, including adversarial. The price is visible in the exponent: the sample count for a given confidence is at most $1/\lambda$ times uniform's. Setting $\lambda = 0$ removes Step 3, and with it the bound: if the proposal puts zero mass on a corridor every path must cross, the failure probability is 1 for every n. The theorem survives a learner. It does not survive a learner with no floor — which is why the Rust constructor refuses $\lambda \leq 0$ and the widget's slider stops at 0.02. ■

22.3.3 Hitting times, divergences, and maximum likelihood

Derivation 1 is a worst-case statement — the learner can only *lose* a factor $1/\lambda$. What it can *win* is a statement about where its mass is.

DERIVATION **Derivation 2 — hitting time, KL as concentration and aim, and maximum likelihood**

$$\arg\max_\theta \mathbb{E}_{p^*}[\ln \hat{p}_\theta] = \arg\min_\theta D_{\mathrm{KL}}(p^* \,\|\, \hat{p}_\theta)$$

Step 1 — geometric waiting time. The number of independent draws until the first one lands in a region A is geometric with success probability $p(A)$, so its mean is $1/p(A)$. If the proposal puts $\hat{p}(A) = a\,u(A)$ on a corridor A, it hits the corridor a times sooner than uniform.

Step 2 — the mixture caps the loss. Under p_λ, $p_\lambda(A) \geq \lambda u(A)$, so the mean wait is at most $1/(\lambda u(A))$ — never more than $1/\lambda$ times uniform's, however

wrong $\hat{p}$ is about A. At $\lambda = 0.1$ that is a factor of ten — the regime the Board's 400-check column shows, where the queries on which the proposal aimed elsewhere are still waiting for their corridor.

Step 3 — concentration. Coarsen Q into N cells with $u_i = 1/N$. Then $D_{\mathrm{KL}}(\hat{p} \,\|\, u) = \sum_i \hat{p}_i \ln(N\hat{p}_i) = \ln N - H(\hat{p})$: zero for the uniform proposal, $\ln N$ for a proposal that puts everything in one cell. It measures how peaked the proposal is, and nothing else.

Step 4 — aim. Let p^* be the distribution of path nodes on the same partition. Then $D_{\mathrm{KL}}(p^* \,\|\, \hat{p})$ is small exactly when $\hat{p}$ has mass wherever paths go. The direction matters: p^* in the first slot makes the divergence infinite if $\hat{p}_i = 0$ for any cell a path uses, and indifferent to mass that $\hat{p}$ wastes elsewhere — the "forward" divergence penalizes the sampler's one catastrophic failure, missing the corridor, and forgives its mild one, sampling where paths do not go.

Step 5 — maximum likelihood. Training data are path nodes $q_1, \ldots, q_M \sim p^*$. The log-likelihood per sample is $\frac{1}{M}\sum_j \ln \hat{p}_\theta(q_j)$, whose expectation is $\sum_i p^*_i \ln \hat{p}_{\theta,i} = -H(p^*) - D_{\mathrm{KL}}(p^* \,\|\, \hat{p}_\theta)$. The entropy $H(p^*)$ does not depend on θ, so maximizing the likelihood — equivalently, minimizing the cross-entropy loss the network is trained on — *is* minimizing the aim divergence. Conditioning on $\mathcal{E}$ and the query changes nothing but the subscripts. ■

The training curve in these units. Uniform's cross-entropy on 144 cells is $\ln 144 = 4.970$ nats. The shipped sampler's final training cross-entropy is 3.393 nats, so by Step 5 it sits 1.576 nats closer to the path distribution, in $D_{\mathrm{KL}}(p^* \,\|\, \cdot)$, than uniform does — on its own training worlds. On fresh worlds from the same generator the check `ch22 aim` measures it directly: with $\lambda = 0.1$ and one fresh PRM path per world as p^*, the mixture is closer to p^* than uniform on 6 of 7 worlds (mean 1.946 nats against 2.729). On the held-out bars it is closer on 8 of 10 (2.186 against 2.770).

DERIVATION **Collapsible — the CVAE of Ichter, Harrison and Pavone, and why this chapter uses cells**

On Reach's two-dimensional torus a categorical over a 12×12 grid is a complete description of any density at that resolution, and it is natively multimodal: two corridors are two bright groups of cells. In seven dimensions a grid has $12^7 \approx 3.6{\times}10^7$ cells and the idea dies. Ichter, Harrison and Pavone's answer is a **conditional variational autoencoder**: a latent $z \sim \mathcal{N}(0, I)$ and a decoder $q = g_\theta(z, y)$ conditioned on $y = (\mathcal{E}, q_{\mathrm{start}}, q_{\mathrm{goal}})$, trained by maximizing the evidence lower bound

$$\ln p_\theta(q \mid y) \;\geq\; \mathbb{E}_{z\sim r_\phi(z|q,y)}\big[\ln p_\theta(q \mid z, y)\big] - D_{\mathrm{KL}}\big(r_\phi(z \mid q, y)\,\|\,\mathcal{N}(0, I)\big)$$

with an encoder r_ϕ used only during training (Kingma and Welling's construction, with conditioning). The latent is what lets one network propose *both* corridors of a world: different regions of z-space decode to different modes. Maximizing the bound is maximizing a lower bound on the likelihood, so Step 5 applies with an inequality, and the learned proposal is used in a mixture with uniform exactly as here — their paper mixes in uniform samples for the same reason Derivation 1 gives. The Rust crate keeps the categorical for T^2 and names the CVAE as the route to higher dimensions; the KL on a partition is then an estimate rather than exact.

22.3.4 The four-cell example, by hand

Coarsen Reach's torus on one Workbench into its four quadrants. Two hundred draws of the raw learned sampler ($\lambda = 0$) fall $(100, 60, 30, 10)$, so $\hat{p} = (0.50, 0.30, 0.15, 0.05)$ against $u = (0.25, 0.25, 0.25, 0.25)$:

$$D_{\mathrm{KL}}(\hat{p}\,\|\,u) = 0.5\ln 2 + 0.3\ln 1.2 + 0.15\ln 0.6 + 0.05\ln 0.2 = 0.34657 + 0.05470 - 0.07662 - 0.08047 = \mathbf{0.2442}\text{ nats } (0.352\text{ bits}).$$

Equivalently $\ln 4 - H(\hat{p})$; the check asserts the identity to 10^{-12}. Now the aim. The same world's roadmap paths put their nodes at $p^* = (0.55, 0.30, 0.10, 0.05)$, and only two terms survive because the other two ratios are one:

$$D_{\mathrm{KL}}(p^*\,\|\,\hat{p}) = 0.55\ln 1.1 + 0.10\ln\tfrac{2}{3} = 0.0524 - 0.0405 = \mathbf{0.0119}, \qquad D_{\mathrm{KL}}(p^*\,\|\,u) = \ln 4 - H(p^*) = 1.38629 - 1.07005 = \mathbf{0.3162}.$$

The learned proposal is about 27× closer to the path distribution than uniform is. Finally the floor: at $\lambda = 0.1$ the mixture is $p_\lambda = 0.9\,\hat{p} + 0.025 = (0.475, 0.295, 0.160, 0.070)$, whose smallest cell, 0.070, is above $\lambda/4 = 0.025$ — Definition 22.2 holds, with room to spare, on the cell the raw sampler nearly ignored. The w22.1 bar chart runs this computation live on the current world.

22.3.5 Heuristics that overestimate

DEFINITION 22.3 — ε-ADMISSIBLE HEURISTIC

A heuristic $\hat{h}$ is **ϵ-admissible** if $\hat{h}(v) \leq (1+\epsilon)\,h^*(v)$ for every node v, with $\hat{h}(q_{\mathrm{goal}}) = 0$. At $\epsilon = 0$ this is Choset's *optimistic* heuristic (App. H.2): "An optimistic, or admissible, heuristic always returns a value less than or equal to the cost of the shortest path from the current node to the goal node

within the graph." For a learned $\hat{h}_\theta$, ϵ is not a design parameter but a *measurement*: the largest $\hat{h}_\theta / h^* - 1$ over an audited set of nodes, with h^* from backward Dijkstra.

$$\hat{h} \le (1+\epsilon)\, h^* \implies C_{A^*} \le (1+\epsilon)\, C^*$$

DERIVATION **Derivation 3 — ε-admissible A* with reopening is (1 + ε)-optimal**

$C \le (1+\epsilon)\, C^*$

Step 1 — the open optimal node. Let $P^* = (q_{\text{start}} = v_0, v_1, \ldots, v_m = q_{\text{goal}})$ be an optimal path of cost C^*. Before the goal is expanded, some node of P^* is on OPEN with its optimal g: take the first v_i on P^* that is not closed with $g(v_i) = g^*(v_i)$. Its predecessor v_{i-1} *was* expanded with $g(v_{i-1}) = g^*(v_{i-1})$, which relaxed v_i to $g(v_i) \le g^*(v_{i-1}) + c(v_{i-1}, v_i) = g^*(v_i)$; so $g(v_i) = g^*(v_i)$, and v_i is on OPEN — if it had been closed with a worse g, the relaxation *reopened* it. (This is where reopening is used.)
Step 2 — its key is at most $(1+\epsilon)C^*$**.** $f(v_i) = g^*(v_i) + \hat{h}(v_i) \le g^*(v_i) + (1+\epsilon)\, h^*(v_i) \le (1+\epsilon)\big(g^*(v_i) + h^*(v_i)\big) = (1+\epsilon)\, C^*$, using $g^* \ge 0$ and $g^*(v_i) + h^*(v_i) = C^*$ on an optimal path.
Step 3 — the goal pops with key C**.** When the goal is popped, $f(q_{\text{goal}}) = g(q_{\text{goal}}) + 0 = C$, the cost of the path A* returns.
Step 4 — best-first order. A* pops the minimum key. At the moment the goal pops, v_i from Step 1 is still on OPEN with key at most $(1+\epsilon)C^*$, so $C \le f(v_i) \le (1+\epsilon)C^*$.
Step 5 — conclusion. The returned cost satisfies $C \le (1+\epsilon)C^*$. Nothing about *how* $\hat{h}$ was produced entered the proof: a learned heuristic, a scaled octile distance (weighted A*), and a hand-tuned guess are all covered, with ϵ the measured worst ratio. ■
The additive form. If instead $\hat{h} \le h^* + \delta$, Step 2 gives $f(v_i) \le C^* + \delta$ and the same argument gives $C \le C^* + \delta$ (Exercise 2).
The reopening caveat. Without reopening, Step 1 fails: an inconsistent heuristic can close a node of P^* with a suboptimal g that is never repaired, and the bound — even at $\epsilon = 0$ — no longer holds. Chapter 6's engine reopens, counts the reopenings, and the widget prints the count.
The measurement caveat. The proof needs $\hat{h} \le (1+\epsilon)h^*$ on the nodes of *this* search. A validation ϵ is a maximum over the audited worlds; it is a guarantee there and an estimate everywhere else.

The three-node audit. A chain $a \xrightarrow{2} b \xrightarrow{2} c \xrightarrow{1}$ goal has true costs-to-

go $h^* = (5.0, 3.0, 1.0)$, and a net outputs $\hat{h} = (4.6, 3.4, 0.8)$. The ratios are $\hat{h}/h^* = (0.92, 1.1333, 0.80)$: admissible at a and c, **violated at** b, where $3.4 > 3.0$. Consistency fails at b too, since $\hat{h}(b) = 3.4 > c(b, c) + \hat{h}(c) = 2.8$. So

$\epsilon = \max(0.92,\ 1.1333,\ 0.80) - 1 =$ **0.1333**, any path A* returns from a costs at most $1.1333 \times 5.0 =$ **5.667**.

The audit is one function, `admissibility_audit`, and the check asserts the violation set $\{b\}$, the inconsistency set $\{b\}$, ϵ and the bound.

The audit of the shipped heuristic. The net predicts $\ln(h^*/h_{oct})$, the factor by which obstacles stretch the empty-torus octile distance, so $\hat{h} = h_{oct}\, e^{\hat{y}}$ is inadmissible wherever the net guesses high. It was trained on 71,848 backward-Dijkstra labels from 240 worlds, with overestimates weighted six times in the squared loss. Auditing every reachable cell of 16 validation worlds, three goals each: 5,584 of 12,830 cells over, $\epsilon_{val} = 0.662$. On 16 held-out bar worlds, $\epsilon = 0.387$ — *smaller*: a sample maximum moves around. The check `ch22 ε audit` repeats this on eight fresh worlds ($\epsilon = 0.327$) and runs 43 searches: every one returned an optimal path. Derivation 3 is a ceiling, not a forecast.

22.3.6 Neural planners and what they inherit

DEFINITION 22.4 — NEURAL MOTION PLANNER

A **neural motion planner** is a waypoint policy $\pi_\theta(\mathcal{E}, q, q_{\text{goal}})$ iterated from q_{start}, whose output passes through a classical **repair**: each hop is certified by `Steer`; each failed stretch is bridged by RRT-Connect between the nearest free waypoints; and if a bridge fails, RRT-Connect runs on the original query.

DERIVATION **Derivation 4 — a proposal plus a verifier inherits the verifier's guarantee**

\text{guarantee}(\text{propose} \circ \text{verify} \circ \text{fallback}) = \text{guarantee}(\text{fallback})

Step 1 — case split. For a query $(q_{\text{start}}, q_{\text{goal}})$ the composition either returns a candidate whose every segment the verifier certified, or hands the *original* query to a fallback planner $\mathcal{F}$.
Step 2 — validity. In the first case the path is collision-free because the verifier is sound (the swept check of Chapter 2 cannot miss a collision; a fixed-step subdivision can, and the neural planner uses the former for that reason). In the second case it is collision-free because $\mathcal{F}$'s paths are.
Step 3 — completeness. The composition fails only if the proposal is

rejected *and* $\mathcal{F}$ fails on $(q_{\text{start}}, q_{\text{goal}})$. So Pr[fail] $\leq$ Pr[$\mathcal{F}$ fails]: if $\mathcal{F}$ is probabilistically complete with Chapter 12's exponential rate, so is the composition. The guarantee $\mathcal{G}$ is a statement about the query, and the fallback answers the same query.

Step 4 — what does not transfer. Time. The composition's running time is the proposal's plus the verification's plus, in the bad case, the whole fallback's; its worst case is worse than $\mathcal{F}$'s alone. Without the fallback the composition has no guarantee at all — only an empirical success rate measured on some distribution of worlds. ■

MPNet's schedule. Qureshi and colleagues' planner proposes bidirectionally, contracts the path, re-plans failed segments with the network and finally with a classical planner. Ours is the one-directional version; Derivation 4 only needs the last resort to run on the original query.

The check `ch22 proposal + verifier` runs the planner on 16 queries, eight from each generator, drawn without asking whether they have an answer: 12 returned a path, all 12 passed an independent dense recheck at a step of 0.005 rad, and 4 queries went to the whole-query fallback. The network *alone* solved 0 of the 8 in-distribution queries and 4 of the 8 held-out ones. Eight queries are too few to say more than that the success rate without repair is not a number anyone should plan on.

22.3.7 Diffusion planners, at the level of statements

DEFINITION 22.5 — DIFFUSION PLANNER

A **diffusion planner** samples a trajectory $\tau \in Q^T$ by iterated denoising from Gaussian noise, optionally tilted toward low cost $J(\tau)$ by guidance. It is a *proposal* in the sense of Definition 22.1: what it emits is checked afterwards, and a sample that fails the check is discarded.

DERIVATION **Derivation 5 — the denoising objective and guided sampling (statement level)**

$$\min_{\theta} \mathbb{E}\,\|\epsilon - \epsilon_\theta(\tau^k, k, \mathcal{E})\|^2$$

Step 1 — noising destroys structure. The forward process $q(\tau^k \mid \tau^{k-1}) = \mathcal{N}\big(\sqrt{1-\beta_k}\,\tau^{k-1},\ \beta_k I\big)$ has the closed form $\tau^k = \sqrt{\bar{\alpha}_k}\,\tau^0 + \sqrt{1-\bar{\alpha}_k}\,\epsilon$ with $\bar{\alpha}_k = \prod_{i\leq k}(1-\beta_i)$ and $\epsilon \sim \mathcal{N}(0, I)$. The toy model uses $K = 40$ steps with β_k linear from 10^{-3} to 0.25, so $\bar{\alpha}_K = 4.06 \times 10^{-3}$: less than half a percent of the demonstration survives, and τ^K is noise.

Step 2 — the net reverses one step. A network $\epsilon_\theta(\tau^k, k, \mathcal{E})$ is trained to predict the noise, by minimizing $\mathbb{E}_{\tau^0,k,\epsilon}\|\epsilon - \epsilon_\theta(\tau^k, k, \mathcal{E})\|^2$ over demonstrations τ^0. Ho, Jain and Abbeel showed that this objective is a reweighted variational bound on the likelihood; at statement level it is "learn to undo one step of noise".
Step 3 — chaining samples the demonstrations. Ancestral sampling runs $\tau^{k-1} = \frac{1}{\sqrt{1-\beta_k}}\left(\tau^k - \frac{\beta_k}{\sqrt{1-\bar{\alpha}_k}}\,\epsilon_\theta(\tau^k, k, \mathcal{E})\right) + \sqrt{\beta_k}\,z$ from $k = K$ down to 1, with $z \sim \mathcal{N}(0, I)$ (no noise on the last step) and the start and goal re-pinned after every step — Janner and colleagues' inpainting of the constraints. If the demonstrations are bimodal, so are the samples: nothing in the procedure averages the two swings into one that goes through the slab.
Step 4 — guidance tilts, it does not constrain. Subtracting $w_k \nabla_\tau J(\tau^k)$ from each step's mean, with $w_k \propto \beta_k$, approximately samples $p_\theta(\tau)\exp(-J(\tau))$ — the demonstrations reweighted toward low cost. In the toy, J is Chapter 19's smoothness term plus a soft penalty $\sum_t \max(0, m - |\tau_t|)^2$ inside the slab's time window. A soft penalty can be outweighed and a gradient step can overshoot; collision is a hard constraint, so the checker runs afterwards. ■

On 64 samples the toy diffuser produces 58% valid trajectories unguided — 30 swinging above the slab, 34 below — and 83% with guidance. Both numbers are high, both modes survive, and neither is one. That last fact is the entire practical content of the section.

22.3.8 What an honest benchmark is

DEFINITION 22.6 — HONEST BENCHMARK

An **honest benchmark** of a learned planning component uses (i) held-out worlds from a generator *different* from the training one, alongside in-distribution worlds; (ii) success at fixed budgets in a planner-independent cost unit, here collision checks; (iii) path cost relative to a reference incumbent; (iv) wall-clock *including* inference, with training time amortized over a stated number of queries; and (v) classical baselines run with the same seeds, the same budget and the same care.

For (iii) the Rust harness uses RRT*'s incumbent after ten seconds; the web port uses the shortest path *any* method found, including a dense reference PRM — weaker (1.0 means "as good as anything we tried"), but every ratio is at least one by construction, as `ch22 Board invariants` asserts.

22.4 The algorithm

Five boxes, one per learned component and one for the board. Each is a classical algorithm from an earlier chapter with one line replaced; the replaced line is marked.

ALGORITHM LEARNED-PROPOSAL PRM (Choset Algorithm 6, sampling line replaced) COST one forward pass, then BASIC PRM: per node one categorical draw (O(log N) by CDF search) plus k calls to Δ

in Workbench encoding $\mathcal{E}$, query (q_start, q_goal), trained net θ, floor $\lambda \in (0, 1]$, budget b in collision checks, k

out a roadmap path from q_start to q_goal, or FAILURE

1: $\pi \leftarrow \text{softmax}\big(f_\theta(\mathcal{E}, q_{\text{start}}, q_{\text{goal}})\big)$ — the cell masses, computed **once** per query
2: **if** $\lambda \le 0$ **then return** error — Derivation 1 needs a floor
3: add $q_{\text{start}}, q_{\text{goal}}$ as nodes; connect each to its k nearest
4: **while** $q_{\text{start}}, q_{\text{goal}}$ are in different components **and** checks spent $< b$ **do**
5: **with probability** λ: $q \leftarrow$ uniform on Q; **otherwise** draw cell $c \sim \pi$ and $q \leftarrow$ uniform in cell c ◀ *replaced line*
6: **if** q is collision-free **then** add q; connect to its k nearest with Δ (unchanged verifier)
7: **return** the shortest roadmap path, or FAILURE

ALGORITHM A*_ε WITH A MEASURED ε (Choset Algorithm 24 with reopening, heuristic replaced) COST audit: one backward Dijkstra per audited goal plus one forward pass per cell; search: Dijkstra's worst case

in lattice G, start s, goal g, net θ, encoding $\mathcal{E}$, audit worlds W

out a path of cost C ≤ (1 + ε)C* on audited worlds, and ε

1: **audit**: **for** each world in W and each sampled goal g': $h^* \leftarrow$ backward Dijkstra from g'
2: $\epsilon \leftarrow \max\big(\epsilon,\ \max_v \hat{h}_\theta(v)/h^*(v) - 1\big)$ over reachable v with $h^*(v) > 0$
3: **search**: $\hat{h}(v) \leftarrow h_{oct}(v)\ \exp\big(\hat{y}_\theta(\mathcal{E}, v, g)\big)$ for every free v ◀ *replaced line*
4: run A* from s to g with key $f = g + \hat{h}$, **reopening** any closed node whose g improves
5: **return** path, its cost C, expansions, reopenings, and ϵ — the bound $(1 + \epsilon)C^*$ holds on the audited worlds (Derivation 3)

ALGORITHM MPNET-STYLE PLAN WITH LAZY REPAIR (proposal: policy; verifier: Steer; fallback: Choset Algorithm 13) COST at most H forward passes, one Steer per hop, then repair trees

in encoding $\mathcal{E}$, query (q_start, q_goal), policy π_θ, hop η, repair budget r, fallback budget F
out a certified path, or FAILURE

$w_0 \leftarrow q_{\text{start}}$; **for** $i = 1, \ldots, H$: $w_i \leftarrow w_{i-1} + \eta \tanh \pi_\theta(\mathcal{E}, w_{i-1}, q_{\text{goal}})$, snapping to q_{goal} within η; re-propose with jitter if w_i collides ◀ *proposal*

for each hop (w_{i-1}, w_i): $ok_i \leftarrow \Delta(w_{i-1}, w_i) \neq$ NIL (swept, sound)

walk the hops; at a failed hop, find the next free waypoint w_j (or q_{goal}) and run RRT-Connect from w_{i-1} to w_j for r iterations; splice the bridge

if any bridge failed **then** run RRT-Connect on $(q_{\text{start}}, q_{\text{goal}})$ for F iterations — the guarantee (Derivation 4)

re-certify every segment of the result; **return** it, or FAILURE

ALGORITHM GUIDED DENOISING, THEN VERIFY (Diffuser-style, statement level) COST K forward passes per sample (O(K T c_net)), then one hard check per sample

in noise predictor ε_θ, schedule $\beta_1 \ldots \beta$_K, guidance cost J and scale s, endpoints, sample count M
out the certified subset of M sampled trajectories

---

for $m = 1, \ldots, M$: $\tau \leftarrow \mathcal{N}(0, I)$; pin $\tau_0 = q_{\text{start}}$, $\tau_{T-1} = q_{\text{goal}}$

  for $k = K, \ldots, 1$: $\mu \leftarrow \frac{1}{\sqrt{1-\beta_k}}\left(\tau - \frac{\beta_k}{\sqrt{1-\bar{\alpha}_k}}\epsilon_\theta(\tau, k)\right) - s\,\beta_k \nabla J(\tau)$

    $\tau \leftarrow \mu + \sqrt{\beta_k}\, z$ (no noise at $k = 1$); re-pin the endpoints

return $\{\tau : \tau$ passes the collision checker$\}$ — the verifier is not optional

ALGORITHM HONEST BENCHMARK (Definition 22.6) COST O(suites × n × planners × b_m)

in two world generators, n queries per suite, planners, budgets $b_1 < \ldots <$ b_m, seed
out per planner and suite: success at each budget, cost ratio, wall-clock with and without inference

---

for each suite and seed: build a world; draw a blocked query; keep it only if a dense reference PRM solves it

2: **for** each planner: run once with the same seed, stopping at b_m checks or at the first verified path
3: record checks at the first path, its cost after shortcutting, wall-clock, inference time
4: incumbent ← the shortest cost any run, or the reference, found
5: **report** success at b_i as the fraction with checks $\leq b_i$; cost ratio over the solved; time amortized over k queries

22.5 Implementation in Rust

The `learn` crate trains with `candle` and infers without it. Training is native-only (`candle-core` and `candle-nn` build the networks, losses and optimizer); each trained network is exported to `safetensors` and evaluated by `infer.rs`, a forty-line forward pass on `nalgebra`, so the WebAssembly widgets and the native planner run identical arithmetic — a test asserts candle and `infer.rs` agree to 10^{-6} on a thousand inputs. The TypeScript port that runs this page goes one step further and hand-rolls a tiny MLP, backpropagation included, in `lib/learn/mlp.ts` — two layers, tanh, Adam — so the checks can train small networks from scratch (XOR, a one-dimensional function, the toy denoiser) and the whole story runs with no dependency. The check `ch22 MLP backprop` compares its gradients to central differences on three architectures; the worst relative discrepancy is 3×10^{-8}. The crate imports and never re-defines Chapter 11's `Sampler`, `Hybrid`, `Uniform` and `Steer`, Chapter 12's `RrtConnect` and Chapter 6's `astar`.

RUST **crates/learn/src/infer.rs, info.rs**

```rust
use nalgebra::{DMatrix, DVector};

/// A dense tanh MLP as exported from candle: layers (W, b), tanh between, linear out.
/// This is the *only* inference path -- native planner, WASM widgets and the TS port agree.
pub struct MlpWeights { pub layers: Vec<(DMatrix<f32>, DVector<f32>)> }

impl MlpWeights {
    pub fn from_safetensors(bytes: &[u8]) -> Result<Self, LearnError> { /* names l0.w, l0.b, ... */ }

    pub fn forward(&self, x: &DVector<f32>) -> DVector<f32> {
        let last = self.layers.len() - 1;
        self.layers.iter().enumerate().fold(x.clone(), |h, (i, (w, b))| {
            let z = w * h + b;
            if i == last { z } else { z.map(f32::tanh) }
        })
    }
}

pub fn softmax(z: &DVector<f32>) -> DVector<f64> {
    let m = z.max();                                   // shift: exp never overflows
    let e = z.map(|v| f64::from(v - m).exp());
```

```
    let s = e.sum();
    e / s
}

/// Counts on N cells and the two divergences of Derivation 2 (nats).
pub struct CoarseHistogram<const N: usize>(pub [u64; N]);

impl<const N: usize> CoarseHistogram<N> {
    pub fn probabilities(&self) -> [f64; N] {
        let t = self.0.iter().sum::<u64>() as f64;
        self.0.map(|c| c as f64 / t)
    }
    /// D_KL(p || u) = ln N - H(p): concentration, not aim.
    pub fn kl_to_uniform(&self) -> f64 {
        let u = [1.0 / N as f64; N];
        kl(&self.probabilities(), &u)
    }
    /// D_KL(target || p): aim. Infinite if p misses a cell the target uses -- the
   failure that matters.
    pub fn kl_from(&self, target: &[f64; N]) -> f64 { kl(target, &self.probabilities()) }
    /// p_lambda = (1 - lambda) p + lambda u, cell by cell (Definition 22.2).
    pub fn mixture(&self, lambda: f64) -> [f64; N] {
        self.probabilities().map(|p| (1.0 - lambda) * p + lambda / N as f64)
    }
}

pub fn kl(p: &[f64], q: &[f64]) -> f64 {
    p.iter().zip(q).filter(|(pi, _)| **pi > 0.0).map(|(pi, qi)| {
        if *qi <= 0.0 { f64::INFINITY } else { pi * (pi / qi).ln() }       // 0 . ln 0 =
   0
    }).sum()
}
```

The sampler is a categorical over a 12×12 coarsening of the torus, conditioned on the 8×8 raster and on (cos, sin) of the four query angles — the wrap-aware encoding Chapter 5 recommends for S^1 factors. It implements Chapter 11's `Sampler` and nothing else; the free-space test inside `sample` is the one every other sampler runs. The safe version is not a new type: it is Chapter 11's `Hybrid`, and the constructor is where Derivation 1 is enforced.

RUST `crates/learn/src/sampler.rs`

```
use candle_core::{DType, Device, Tensor};
use candle_nn::{linear, loss::cross_entropy, AdamW, Module, Optimizer, ParamsAdamW,
   VarBuilder, VarMap};
use manifold::{Manifold, T2};
use rand::{rngs::SmallRng, Rng};
use sampling::{FreeSpace, Hybrid, Sampler, Uniform};
use crate::{encode::{query_features, EnvEncoding}, infer::{softmax, MlpWeights},
   LearnError};

pub const CELLS_N: usize = 12;
pub const CELLS: usize = CELLS_N * CELLS_N;
type Q = <T2 as Manifold>::Point;

/// D22.1. The raw proposal p_theta(. | ?, q_start, q_goal): draw a cell, then a uniform
   point in it.
```

```rust
pub struct LearnedSampler { probs: Vec<f64>, cdf: Vec<f64> }

impl LearnedSampler {
    /// One forward pass per query -- the cell masses do not depend on the samples drawn.
    pub fn new(net: &MlpWeights, env: &EnvEncoding, start: &Q, goal: &Q) -> Self {
        let probs: Vec<f64> = softmax(&net.forward(&query_features(env, start, goal))).iter().copied().collect();
        let cdf = probs.iter().scan(0.0, |acc, p| { *acc += p; Some(*acc) }).collect();
        Self { probs, cdf }
    }
    pub fn draw_raw(&self, rng: &mut SmallRng) -> Q {
        let u: f64 = rng.random();
        let c = self.cdf.partition_point(|&f| f < u).min(CELLS - 1);
        let (i, j) = (c % CELLS_N, c / CELLS_N);
        let w = std::f64::consts::TAU / CELLS_N as f64;
        T2::from_angles([-std::f64::consts::PI + (i as f64 + rng.random::<f64>()) * w,
                         -std::f64::consts::PI + (j as f64 + rng.random::<f64>()) * w])
    }
}

impl Sampler<T2> for LearnedSampler {
    fn sample(&mut self, _: &T2, free: &dyn FreeSpace<T2>, rng: &mut SmallRng) -> Option<Q> {
        let q = self.draw_raw(rng);
        free.is_free(&q).then_some(q)                  // the verifier is Chapter 11's, untouched
    }
}

/// D22.2 by construction: Chapter 11's `Hybrid` with `Uniform` as the floor at weight lambda.
/// Derivation 1 needs lambda > 0, so lambda = 0 is an error here, not a warning in a README.
pub fn safe_learned_sampler(learned: LearnedSampler, lambda: f64) -> Result<Hybrid<T2>, LearnError> {
    if !(lambda > 0.0 && lambda <= 1.0) { return Err(LearnError::NoUniformFloor(lambda)); }
    Ok(Hybrid { parts: vec![(Box::new(learned), 1.0 - lambda), (Box::new(Uniform), lambda)] })
}

/// Maximum likelihood on path-node cells = min D_KL(p* || p_theta) (Derivation 2). Native only.
pub fn train_sampler(x: &[Vec<f32>], cells: &[u32], epochs: usize) -> candle_core::Result<VarMap> {
    let dev = Device::Cpu;
    let vars = VarMap::new();
    let vb = VarBuilder::from_varmap(&vars, DType::F32, &dev);
    let (l0, l1) = (linear(x[0].len(), 64, vb.pp("l0"))?, linear(64, CELLS, vb.pp("l1"))?);
    let mut opt = AdamW::new(vars.all_vars(), ParamsAdamW { lr: 2e-3, ..Default::default() })?;
    let xs = Tensor::from_vec(x.concat(), (x.len(), x[0].len()), &dev)?;
    let ys = Tensor::from_slice(cells, cells.len(), &dev)?;
    for _ in 0..epochs {
        let logits = l1.forward(&l0.forward(&xs)?.tanh()?)?;
        opt.backward_step(&cross_entropy(&logits, &ys)?)?;   // full batch for brevity; the crate mini-batches
    }
    Ok(vars)                                                 // -> safetensors -> MlpWeights
```

```
}
```

The heuristic is a closure, because that is all Chapter 6's `astar` ever asked of a heuristic. The ϵ is a field that starts as `None` and is filled only by an audit — the type makes it impossible to quote a bound that was never measured.

RUST **crates/learn/src/heuristic.rs**

```rust
use search::{astar, Trace};
use cspace::TorusGrid;                                  // Ch. 4's raster, 8-connected with wraparound
use crate::{encode::{heuristic_features, EnvEncoding}, infer::MlpWeights};

/// Learned cost-to-go. ? = h_oct . exp(?): at least the admissible baseline where ? >= 0.

pub struct HeuristicNet<'a> { net: &'a MlpWeights, env: &'a EnvEncoding, grid: &'a TorusGrid,
                              goal: usize, pub epsilon: Option<f64> }

impl HeuristicNet<'_> {
    pub fn h(&self, v: usize) -> f64 {
        let base = self.grid.octile(v, self.goal);
        if base == 0.0 { return 0.0; }
        let y = self.net.forward(&heuristic_features(self.env, self.grid, v, self.goal))[0];
        base * f64::from(y).clamp(-1.0, 3.0).exp()   // clamp: one wild guess cannot overflow
    }
}

/// Derivation 3's hypothesis, measured: max over every reachable cell of ?/h* - 1.
pub fn measure_epsilon(h: &HeuristicNet, h_star: &[f64]) -> (f64, usize) {
    h_star.iter().enumerate()
        .filter(|(_, hs)| hs.is_finite() && **hs > 0.0)
        .fold((0.0, 0), |(eps, over), (v, hs)| {
            let r = h.h(v) / hs;
            if r > 1.0 + 1e-9 { (eps.max(r - 1.0), over + 1) } else { (eps, over) }
        })
}

/// Ch. 6's A*, which reopens; the bound below is Derivation 3 and holds only where epsilon was audited.
pub fn a_star_eps(grid: &TorusGrid, s: usize, g: usize, h: &HeuristicNet) -> Option<(Trace<usize>, f64)> {
    let eps = h.epsilon.expect("measure epsilon before quoting a bound");
    let t = astar(grid, s, g, |v| h.h(v))?;
    let bound = (1.0 + eps) * grid.cost_to_go(g)[s];
    debug_assert!(t.cost <= bound + 1e-9, "Derivation 3 violated on an audited world");
    Some((t, bound))
}

pub struct Audit { pub ratios: Vec<f64>, pub violated: Vec<char>, pub inconsistent: Vec<char>,
                   pub epsilon: f64, pub bound: f64 }

/// sec.3(b): a chain v_0 -> v_1 -> ... -> goal with edge costs c; both heuristics are 0 at the goal.
```

```rust
pub fn admissibility_audit(labels: &[char], h_star: &[f64], h_hat: &[f64], c: &[f64]) -> Audit {
    let ratios: Vec<f64> = h_hat.iter().zip(h_star).map(|(a, b)| a / b).collect();
    let violated = labels.iter().zip(&ratios).filter(|(_, r)| **r > 1.0 + 1e-12).map(|(l, _)| *l).collect();
    let inconsistent = (0..labels.len())
        .filter(|&i| h_hat[i] > c[i] + h_hat.get(i + 1).copied().unwrap_or(0.0) + 1e-12)
        .map(|i| labels[i]).collect();
    let epsilon = (ratios.iter().copied().fold(f64::MIN, f64::max) - 1.0).max(0.0);
    Audit { ratios, violated, inconsistent, epsilon, bound: (1.0 + epsilon) * h_star[0] }
}
```

The neural planner holds its repair planner as a field, not an `Option`: Derivation 4 says the guarantee lives there, and a planner without one has only a success rate.

RUST **crates/learn/src/neural_planner.rs**

```rust
use manifold::T2;
use rand::{rngs::SmallRng, Rng};
use sampling::{tree::{Extend, RrtConnect}, FreeSpace, Steer};
use crate::{encode::EnvEncoding, infer::MlpWeights, Q};

pub struct MpnetLike<E: Extend<T2>> { policy: MlpWeights, env: EnvEncoding, repair: E,
                                      eta: f64, max_hops: usize, local: usize, fallback: usize }

pub struct NeuralPlan { pub waypoints: Vec<Q>, pub hops_ok: Vec<bool>,
                        pub fell_back: bool, pub path: Option<Vec<Q>> }

impl<E: Extend<T2> + Clone> MpnetLike<E> {
    pub fn plan(&self, free: &dyn FreeSpace<T2>, s: &Q, g: &Q, rng: &mut SmallRng) -> NeuralPlan {
        // D22.1: <= max_hops of q + eta.tanh(pi_theta), re-proposed with jitter where the endpoint collides.
        let waypoints = self.propose(free, s, g, rng);
        let hops_ok: Vec<bool> = waypoints.windows(2)          // the verifier, one hop at a time
            .map(|w| self.repair.steer(&T2, free, &w[0], &w[1]).is_some()).collect();
        let mut out = vec![*s];
        let (mut i, mut failed) = (0, false);
        let last = waypoints.len() - 1;
        while i < last {
            if hops_ok[i] { out.push(waypoints[i + 1]); i += 1; continue; }
            let j = (i + 1..last).find(|&j| free.is_free(&waypoints[j])).unwrap_or(last);
            match self.connect(free, &waypoints[i], &waypoints[j], self.local, rng) {
                Some(bridge) => { out.extend_from_slice(&bridge[1..]); i = j; }
                None => { failed = true; break; }
            }
        }
        // Derivation 4: the last resort answers the *original* query, so its guarantee is ours.
        let (path, fell_back) = if failed { (self.connect(free, s, g, self.fallback, rng), true) }
                                else { (Some(out), false) };
        let path = path.filter(|p| p.windows(2).all(|w| self.repair.steer(&T2, free, &w[0], &w[1]).is_some()));
        NeuralPlan { waypoints, hops_ok, fell_back, path }
```

```
    }

    fn connect(&self, free: &dyn FreeSpace<T2>, a: &Q, b: &Q, iters: usize, rng: &mut SmallRng) -> Option<Vec<Q>> {
        RrtConnect::new(T2, a.clone(), b.clone(), self.repair.clone(), self.eta, rng.random()).merge(free, iters)
    }
}
```

The worked example prints the two micro-examples exactly — they are arithmetic, and the test asserts them — and then trains on randomized Workbenches.

RUST **crates/learn/examples/learned_sampler.rs**

```
use learn::{admissibility_audit, data, CoarseHistogram};

fn main() -> anyhow::Result<()> {
    // sec.3(a): 200 raw draws on four quadrants of one Workbench's torus.
    let hat = CoarseHistogram([100u64, 60, 30, 10]);
    let target = [0.55, 0.30, 0.10, 0.05];
    println!("counts {:?}  KL(p_hat||u) = {:.4} nats ({:.3} bits)  KL(p*||p_hat) = {:.4}  KL(p*||u) = {:.4}",
             hat.0, hat.kl_to_uniform(), hat.kl_to_uniform() / std::f64::consts::LN_2,
             hat.kl_from(&target), learn::kl(&target, &[0.25; 4]));
    println!("p_lambda(0.1) = {:.3?}", hat.mixture(0.1));

    // sec.3(b): a ->2-> b ->2-> c ->1-> goal.
    let a = admissibility_audit(&['a', 'b', 'c'], &[5.0, 3.0, 1.0], &[4.6, 3.4, 0.8], &[2.0, 2.0, 1.0]);
    println!("h* [5.0 3.0 1.0]  h_hat [4.6 3.4 0.8]  violated {:?}  inconsistent {:?}  epsilon {:.4}  bound {:.3}",
             a.violated, a.inconsistent, a.epsilon, a.bound);

    // The pipeline: 600 random Workbenches * 3 queries -> paths -> sampler, policy; Dijkstra -> heuristic.
    let seed = std::env::args().skip_while(|a| a != "--seed").nth(1).map_or(Ok(7), |s| s.parse())?;
    let ds = data::generate_dataset(data::Generator::Blocks, 600, 3, seed);
    println!("{}/{} PRM queries solved . {} sampler targets . {} policy targets",
             ds.solved, ds.attempted, ds.sampler.len(), ds.policy.len());
    Ok(())
}
```

```
counts [100, 60, 30, 10]  KL(p_hat||u) = 0.2442 nats (0.352 bits)  KL(p*||p_hat) = 0.0119  KL(p*||u) = 0.3162
p_lambda(0.1) = [0.475, 0.295, 0.160, 0.070]
h* [5.0 3.0 1.0]  h_hat [4.6 3.4 0.8]  violated ['b']  inconsistent ['b']  epsilon 0.1333  bound 5.667
954/1800 PRM queries solved . 21508 sampler targets . 44553 policy targets
```

The first three lines are exact and are what the tests below assert. The last is the TypeScript port's run of the same pipeline (`lib/learn/__train__.ts`, seed 7), which the page's numbers come from; a Rust run reproduces the pattern, not the digits, because the random streams differ. As recorded there: the sampler's

cross-entropy fell to 3.393 nats in 50 epochs (uniform: 4.970); generating the paths took 147 s and training the sampler 93 s — 241 s in all, the number the Board amortizes.

RUST crates/learn/tests/micro.rs

```rust
use learn::*;

#[test]
fn kl_four_cell_matches_hand_computation() {
    let h = CoarseHistogram([100u64, 60, 30, 10]);
    assert!((h.kl_to_uniform() - 0.2442).abs() < 1e-4);
    let entropy = -h.probabilities().iter().map(|p| p * p.ln()).sum::<f64>();
    assert!((h.kl_to_uniform() - (4f64.ln() - entropy)).abs() < 1e-12);       // ln N - H(p)
    assert!((h.kl_from(&[0.55, 0.30, 0.10, 0.05]) - 0.0119).abs() < 1e-4);
}

#[test]
fn eps_admissibility_three_nodes() {
    let a = admissibility_audit(&['a', 'b', 'c'], &[5.0, 3.0, 1.0], &[4.6, 3.4, 0.8], &[2.0, 2.0, 1.0]);
    assert_eq!(a.violated, vec!['b']);
    assert_eq!(a.inconsistent, vec!['b']);
    assert!((a.epsilon - 0.13333).abs() < 1e-4 && (a.bound - 5.6667).abs() < 1e-3);
}

#[test]
fn mixture_never_below_lambda_uniform() {
    let h = CoarseHistogram([100u64, 60, 30, 10]);
    for lambda in [0.05, 0.1, 0.3] { assert!(h.mixture(lambda).iter().all(|&m| m >= lambda / 4.0 - 1e-12)); }
    assert!(safe_learned_sampler(LearnedSampler::uniform_stub(), 0.0).is_err());
}
```

The crate's fourth test, `a_star_eps_cost_within_bound`, runs Derivation 3 as a property on 500 seeded lattices with $\hat{h} = h^*(1 + 0.3\,r)$, $r \sim U[0, 1]$. The web port's checks are the same tests in TypeScript — its property test saw a worst realized ratio of 1.0711 over 30 searches, with 13 reopenings — plus the mixture floor with 100,000 empirical draws, the ϵ audit, the proposal-plus-verifier recheck, the toy diffuser, and the Board's invariants.

22.6 Putting it together: reading the board

The integration lab is the Honest Benchmark Board above, read the way Definition 22.6 says to: thirty solvable queries per suite, from the training generator (axis-aligned blocks, like the real Workbench's block, post and shelf) and the held-out one (thin bars at arbitrary angles), budgets from 100 to 12,800 collision checks, the same world, query and seed for every planner.

Success at budget. In distribution, at 100 checks, the learned sampler at $\lambda = 0.1$ solves 60% against uniform's 47%; RRT-Connect solves 57%. At 400

checks the learned sampler has fallen behind: 77% against uniform's 90%, the bridge sampler's 80% and RRT-Connect's 93%. That shape — ahead early, behind later — is Derivation 2's cap in action. When the proposal aims right, it hits the corridor sooner than uniform; when it aims wrong, only the λ floor keeps it searching, at up to ten times uniform's waiting time, and by 400 checks those queries are what the column counts. At 12,800 checks the learned and uniform rows both solve every query in both suites — Derivation 1, on the board.

The held-out suite. We expected the learned sampler to lose its edge on the other generator. At $\lambda = 0.1$ it did not lose more than it had: held-out at 100 checks it solves 63% against uniform's 60%, and at 400 checks both solve 90% (bridge 80%, RRT-Connect 100%). Single worlds show the reversal — w22.1's default held-out seed costs the learner more than twice uniform's checks — but over thirty queries the bars were not harder for it than the blocks. Thirty queries is a small suite, and "did not degrade on one other generator" is not "generalizes".

Path cost. The learned sampler's first paths are consistently shorter: 1.28 times the incumbent in distribution against uniform's 1.33, and 1.23 against 1.43 held-out. That is Derivation 2's aim paying off in a currency nobody optimized for — the targets were nodes on *shortest* roadmap paths, so the proposal concentrates where short paths go. RRT-Connect, at 1.07 in distribution, beats every PRM row on cost as well.

Wall-clock. Here the learned row loses outright. A query's forward pass costs about 0.10 ms, so inference is not what makes the learned PRM's mean wall-clock in distribution 12 ms against uniform's 10 ms; the difference is in the planning loop itself. Amortizing the 241 s of data generation and training over k queries can only make that worse, and the break-even against uniform is *never*. The MPNet-style planner pays 1.1 ms of inference per query and 53 ms of wall-clock, most of it in repairs, and reaches 93% at the full budget in both suites; on the remaining queries the fallback either ran out of iterations or overran the check budget.

What a fair reading concludes. On Reach's two-dimensional torus, with these small networks, a learned proposal buys a better first path and a head start at the smallest budgets, and it costs wall-clock and four minutes of offline work before the first query. A hand-designed single-query planner, RRT-Connect, is better than all of it. None of that is a verdict on learning in planning — the literature's wins are in seven-dimensional arms and repeated queries on similar scenes, where uniform sampling is hopeless and a training run is amortized over many queries — but it is a verdict on *this* setting, and the board's purpose is that such verdicts are possible at all.

22.6.1 What is principled, and what is still alchemy

The chapter closes with a ledger, because the field needs one more than it needs another leaderboard.

Principled. Any completeness-preserving proposal keeps Choset's guarantee

with its constant multiplied by λ (Derivation 1), and a constructor that refuses $\lambda = 0$ makes that a property of the code. Maximum-likelihood training of a sampler is minimization of the forward KL divergence from the path distribution (Derivation 2), so "the sampler is aimed well" has a definition and a number. A heuristic's overestimate bounds the path cost exactly (Derivation 3), provided ϵ is *measured*, and provided one says aloud that a validation ϵ is a guarantee on the validation worlds and an estimate elsewhere — on this chapter's worlds the held-out ϵ happened to be smaller than the validation one, which is luck, not a law. A proposal behind a sound verifier and a fallback on the original query inherits the fallback's guarantee (Derivation 4). And a benchmark that includes inference time, training time and held-out worlds (Definition 22.6) can falsify a claim, which is the only kind of benchmark worth running.

Still alchemy in 2026. *Generalization across world generators*: our held-out suite is one other generator, and no theorem says what the next one does. *Validation ϵ as a deployment guarantee*: it is not one. *Neural planners' success rates without repair*: an empirical rate on a distribution of worlds, and the check that runs the net alone saw 0 of 8 in one suite and 4 of 8 in the other. *Diffusion guidance through a non-differentiable constraint*: collision is a set, not a cost; guidance raised the toy's valid fraction from 58% to 83% and no setting makes it one. *The circularity*: the sampler is trained on paths from the very PRM it is benchmarked against, so at best it reproduces PRM-with-a-large-budget, faster — it cannot find a passage the teacher never found. And *amortization*: a training run is paid for only by enough queries on *similar* worlds, and on this board the break-even never arrives.

Chapter 23's planning stack will carry a "learned sampler" toggle in Reach's panel, wired through exactly the `Hybrid` constructor above, and cite this board in its retrospective. The sister book's Chapter 25 applies the same principled-or-alchemy discipline to learning inside an estimator, where the verifier is a likelihood rather than a collision checker and the inheritance argument has a different shape.

22.7 Exercises

1 FOUNDATION •• **λ through the whole of Chapter 11**

Carry λ through Chapter 11's $(\epsilon, \alpha, \beta)$-expansiveness proof (Theorem 7.4.2), not only through the tiling bound of Derivation 1. Every volume in that proof is a uniform measure of a reach set or a lookout; under p_λ each is bounded below by λ times itself. Name the constant of the theorem that absorbs the change. Then show that a proposal with $\hat{p}(A) = 0$ on one tile A makes the failure probability of Derivation 1 bounded below by a positive constant independent of n.

WORK IT OUT

Chapter 11's ball-tiling example (path length 1.2, clearance 0.1, unit square) needs 892 uniform samples for an exponential-form failure bound of 1%. With the same tiles and a mixture at $\lambda = 0.1$, how many samples does the same bound require?

Answer. 8920 In the exponential form the requirement is $n \geq \ln(\ell/\delta)/p$; replacing p by λp multiplies it by $1/\lambda = 10$. The learned sampler may need far fewer in practice — the bound is the price of knowing nothing about it.

HINTS

<>Theorem 7.4.2's sample bound has the form of a logarithm divided by a product of the expansiveness constants.</> <>ε and α are fractions of μ(Q_free); under p_λ both measured masses can only shrink by a factor λ, never more.</> <>For the converse: if a tile has zero proposal mass and $\lambda = 0$, the probability that it stays empty is 1 for every n.</>

2 **FOUNDATION** •• **The additive bound, and A* without reopening**

Prove the additive form of Derivation 3: if $\hat{h} \leq h^* + \delta$ everywhere, A* with reopening returns $C \leq C^* + \delta$. Then build a four-node graph — start, two intermediate nodes, goal — with an *admissible but inconsistent* heuristic on which A* *without* reopening returns a suboptimal path, and explain which step of the derivation breaks.

HINTS

<>Only Step 2 changes: f(v) ≤ g(*v*) + *h*(v) + δ.</> <>You want a node on the optimal path to be closed first through an expensive edge, because the cheap route to it passes a node whose heuristic is high — but still admissible.</> <>Try edges s→a 1, a→b 1, s→b 3, b→t 3 with $\hat{h}$(s) = $\hat{h}$(b) = 0 and $\hat{h}$(a) = 4. Every value is admissible (h* = 5, 4, 3, 0 for s, a, b, t), but $\hat{h}$(a) exceeds c(a, b) + $\hat{h}$(b) = 1. Trace the pops: b is closed with g = 3 before a reveals g = 2, and without reopening the goal pops at 6, not 5. Step 1's open optimal node is the one that was closed and never reopened.</>

3 CONCEPTUAL • **Predict the held-out sampler, then verify**

PREDICT FIRST

In Learned Sampler vs Uniform, set λ to its minimum, 0.02, and switch to the held-out generator. Before pressing play: across the seeds the success-at-budget curve accumulates, what happens to the learned curve compared with uniform's at the largest budgets?

- It never reaches uniform's level — with a tiny floor some held-out queries stay unsolved

  Derivation 1 still applies at $\lambda = 0.02$: the bound decays fifty times more slowly than uniform's, but it decays. Given enough checks every solvable query is joined.

- ✓ It reaches the same level, later on the worlds where the net aims wrong

  Right. The floor guarantees the knee exists; at $\lambda = 0.02$ a mis-aimed proposal waits up to fifty times uniform's time for the corridor, so the learned curve rises later on those seeds and catches up only at large budgets.

- It is identical to uniform's — the net is useless on worlds it never saw

  Too strong. The held-out bars still share the table, the base clearance and the arm with the training worlds; the check `ch22 aim` finds the proposal closer to p* than uniform on 8 of 10 held-out worlds. The curve differs seed by seed.

Now raise λ to 0.3 and watch the gap at the larger budgets close: a heavier floor caps the loss at a smaller factor, $1/\lambda \approx 3$, at the cost of diluting the head start on the worlds where the network was right.

4 CONCEPTUAL •• **The deflated heuristic**

PREDICT FIRST

In Heuristic Calibration, switch from the learned $\hat{h}$ to $\hat{h}$ / (1 + ε_val), with ε_val = 0.662 from the validation worlds. Compared with the admissible octile heuristic, the deflated one will expand. . .

- Fewer nodes — it is the learned heuristic made safe
  Dividing by 1.662 shrinks every estimate, including the many where the net was accurate or low. Most of the deflated values fall well below octile's.
- About as many nodes
  Only if the net were systematically 1.66 times too high, which it is not: its mean ratio to h* is 0.959.
- ✓ Several times as many nodes, and return an optimal path
  Right. $\hat{h}/1.662$ is admissible on the validation worlds and is a much weaker lower bound than octile almost everywhere, so A* drifts toward Dijkstra. Admissibility bought by uniform deflation throws away what the net knew.

The cure is a better-shaped loss — an asymmetric objective (this net already weights overestimates six times), or a correction clipped at the admissible baseline. Neither has a guarantee off the audited worlds; both waste less.

5 PRACTICAL •• **A learned sampler for Rusty**

Implement `Sampler` on Rusty's $\mathbb{R}^2$ configurations in the Apartment for a `LearnedSampler` conditioned on a 64×64 occupancy raster, trained on 500 randomized floorplans (move interior walls and doorways; keep the outer walls). Wrap it with `safe_learned_sampler` and add its row to the Board, with a held-out generator that moves doors to walls the training generator never opened. Does the held-out gap shrink or grow with the larger raster? Report success at three budgets and the break-even k, and say which of the two you believe more.

6 PRACTICAL ••• **Stretch: a learned heuristic for hybrid A***

Train a heuristic for Chapter 21's hybrid A* on the Lot, with labels from backward Dijkstra on the (x, y, θ) lattice for Hitch. Measure ϵ on held-out Lots (different parking-row layouts) with `measure_epsilon`, run the planner, and compare the realized cost ratio to the bound $(1 + \epsilon)$ and to the classical $\max(h_{RS}, h_{holonomic})$ — the Reeds–Shepp distance and the obstacle-aware 2-D distance. Which of the three heuristics expands the fewest nodes, and which would you ship?

22.8 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.
§7.1.3's hand-designed biased samplers (OBPRM, Gaussian, bridge) are this chapter's baselines; §7.4.3's abstract path tiling, whose framework covers sampling from arbitrary distributions, is the theorem Derivation 1 extends; Appendix H.2's optimistic heuristic is Definition 22.3 at $\varepsilon = 0$. link

Ichter, B., Harrison, J., and Pavone, M. (2018). Learning Sampling Distributions for Robot Motion Planning. *IEEE International Conference on Robotics and Automation (ICRA)*.
The conditional variational autoencoder as a learned sampling distribution, mixed with uniform samples — the design this chapter's sampler simplifies to a categorical on T^2. link

Qureshi, A. H., Simeonov, A., Bency, M. J., and Yip, M. C. (2019). Motion Planning Networks. *IEEE International Conference on Robotics and Automation (ICRA)*.
MPNet: a learned waypoint policy with lazy states contraction, neural re-planning and a classical fallback — the instance of Derivation 4. link

Bhardwaj, M., Choudhury, S., and Scherer, S. (2017). Learning Heuristic Search via Imitation. *Conference on Robot Learning (CoRL)*.
Learning which node to expand by imitating an oracle with full knowledge of the cost-to-go — the sequential-decision view of a learned heuristic. link

Janner, M., Du, Y., Tenenbaum, J. B., and Levine, S. (2022). Planning with Diffusion for Flexible Behavior Synthesis. *International Conference on Machine Learning (ICML)*.
Diffuser: trajectory-level denoising with inpainted start and goal and gradient guidance by a cost — the model Derivation 5 states and the toy of w22.4 shrinks to one dimension. link

Ho, J., Jain, A., and Abbeel, P. (2020). Denoising Diffusion Probabilistic Models. *Advances in Neural Information Processing Systems (NeurIPS)*.
The noise-prediction objective and the ancestral sampling step used in Derivation 5, with the variational bound it reweights. link

Kingma, D. P. and Welling, M. (2014). Auto-Encoding Variational Bayes. *International Conference on Learning Representations (ICLR)*.
The evidence lower bound and the reparameterized encoder behind the CVAE sampler of the collapsible note. link

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics 4(2), 100–107*.
A* and the admissibility theorem whose ε-relaxation is Derivation 3. doi:10.1109/TSSC.1968.300136

Karaman, S. and Frazzoli, E. (2011). Sampling-based Algorithms for Optimal Motion Planning. *International Journal of Robotics Research 30(7), 846–894*.
The asymptotic-optimality baseline (RRT, *PRM*) a learned proposal must not pretend to beat; its incumbent is the Rust harness's cost reference. doi:10.1177/0278364911406761

CHAPTER 23

Capstone: A Planning Stack from Sensor to Motion

FRONTIERS AND INTEGRATION · Advanced · 75 min

> *The distinction between offline algorithms and online sensor-based algorithms can be somewhat murky; if an offline planner runs quickly enough, for example, then it can be used in a feedback loop to continually replan when new sensor data updates the environment model.*
>
> **Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun**
>
> *Principles of Robot Motion (2005), Chapter 1*

Twenty-two chapters built planners. Each one came with its own world, its own query, and a widget that ran it in isolation: a tree growing on a torus, a wave front crossing a grid, a car tracing Reeds–Shepp arcs between two poses nobody would ever drive between. This chapter builds the thing those planners are for — *the stack* — three times over, and runs all three at once.

Rusty is dropped into the Apartment with the architect's drawing and nothing else. Nobody told it where the furniture is, or which doors are shut. It localizes against the drawing with Chapter 16's particle filter, plans on Chapter 6's grid with D* Lite, and tracks the plan with Chapter 19's receding-horizon controller; when a door it was counting on turns out to be closed, two scans confirm it, the grid grows a few amber cells, and the plan repairs. Reach makes three picks on the Workbench: Chapter 12 finds a joint path, Chapter 13 improves it inside a budget, Chapter 18 gives it a clock that saturates the motors, Chapter 19 smooths it — and a verifier throws the smoothed version away whenever it touches the block. Hitch tows a trailer across the Lot into bay 3 with Chapter 21's hybrid A*, while a guard built from Chapter 20's trailer kinematics watches the hitch angle.

The argument of the chapter is architectural, and it has mathematics. A stack is a chain of **contracts**; each stage lives in a different cell of Choset's task × robot

× algorithm table, and the composed system has exactly the guarantee of its weakest stage — no more, and only that much if every downstream stage either checks its input or falls back to it. Staleness adds up along the chain and turns into a braking distance. Recoveries are not luck: each one is a statistic the book already derived, crossing a threshold someone has to defend. And the claim the book has been building toward is literal here: **everything you studied is running on this page, unmodified, at real time, in the browser.**

◎ AFTER THIS CHAPTER YOU CAN

- Read a planning stack as a chain of contracts — input type, output type, frame, period, and the guarantee a stage gives when its input is valid — and make the path/trajectory distinction a type nothing can cross by accident
- Prove the guarantee-composition rule: a pipeline is as complete as its weakest rejecting stage, and only that complete if every post-processor verifies its output or falls back to its input
- Derive the replanning budget $v \cdot T_{react} + v^2/2a < d_{detect}$ from the staleness of each stage, compute the safe speed and the validity horizon for the book's default rates, and see the D23.2 check catch what the budget alone would miss
- Turn two statistics the book already owns into guards with thresholds you can defend: the manipulability $\mu = l_1 l_2 |\sin \theta_2|$ of Chapter 4 and the hitch angle of Chapter 20
- Compose Chapters 2–22 in Rust as stages behind one trait, scheduled on threads natively and by a deterministic round-robin in WASM, with a supervisor whose Mode the compiler keeps exhaustive
- Map every stage onto its counterpart in OMPL, MoveIt and Nav2, and write the retrospective honestly: what the type system caught, and what it cost

ASSUMED

- The whole book. The load-bearing chapters are 4 (Jacobian, inflation), 6 (D *Lite*), 12–13 (*RRT-Connect*, *Informed RRT*), 16 (particle belief, replan_on_belief), 18 (time scaling), 19 (CHOMP, MPC), 20–21 (the trailer, hybrid A*)
- Chapter 1's completeness vocabulary — complete, resolution-complete, probabilistically complete — which this chapter composes
- Rust traits, enums with data, and exhaustive match

23.1 Sixty seconds, three robots

INTERACTIVE · w23.1

The Grand Plan

An autonomy demo is not one clever algorithm: it is three chains of stated contracts, and every failure is a contract violation you can point at.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch23-capstone#w23.1

Nothing in that widget is recorded. Each animation frame advances one `Mission` by roughly one 50 ms tick of simulated time: fifteen stages on one round-robin scheduler, each running only when its period has elapsed. The particle cloud is being reweighted ten times a second against the architect's drawing; D* Lite repairs its cost-to-goal field twice a second; Informed RRT* is doing forty iterations per tick on the torus until its budget runs out; the trailer's heading is being integrated by the same RK4 that Chapter 2's simulator uses. Every number in the inspectors is read straight out of the running stages.

Three things are worth watching before any button is pressed.

Rusty is drawn where it *believes* it is. The orange triangle is the mode of a particle cloud localized against the blueprint; the purple ellipse is three standard deviations of that cloud. Switch on *ground truth* and a gray outline appears where Rusty really is — a few centimeters away, because the wheels slip and the belief only ever learns about it from the scans. Every downstream stage plans from the belief. If that sounds like a detail, open the Belief tab and read the line *map trusts scans*: the mapping stage refuses to paint an obstacle while the cloud is wider than 5 cm, because a scan projected through a loose belief puts walls where there are none.

Reach's motion has a clock, and the clock is tight. The strip under the torus is the phase plane $(s, \dot{s})$ of Chapter 18. The purple profile touches the amber velocity-limit curve: at every instant one of the two motors is at its torque limit. That is what *time-optimal* means, and it is why the smoothed path Reach actually executes is a few hundredths of a second faster than the shortcut path it came from — CHOMP lowered the curvature, which raised the limit curve.

Hitch reverses first. It starts facing the east curb with the bay behind it, so the only way in is to back out, and reversing is exactly when a trailer is unstable. The hybrid A* path is accepted only because the trailer, simulated along it, never folds past $\phi_{plan} = 0.85$ rad. The gauge in the Guards tab shows the hitch angle the whole way.

Now press the three chaos buttons. *Close Door* shuts the next door on Rusty's path; *Singular Goal* moves Reach's next pick out to full stretch; *Reverse Hard* throws Hitch into full reverse at full lock. Each robot recovers, and the left rail shows the mode machine that did it. The rest of the chapter explains every panel and every

recovery.

> **THE ONE SENTENCE VERSION**
>
> A planning stack is a chain of typed contracts: each stage promises something only about valid input, the composed guarantee is the weakest promise in the chain, and every recovery is a named statistic crossing a threshold followed by a mode with a plan.

23.2 Why not one planner?

The first question, having built twenty planners, is why the capstone needs three different pipelines rather than one good one. Chapter 1 already gave the answer in Choset's own form: a motion-planning problem is a point in a space of *task* × *robot* × *algorithm*, and the algorithm column is decided by the other two. Our three robots sit in three different rows.

	Rusty	**Reach**	**Hitch (+ trailer)**
Configuration space	$\mathbb{R}^2$ (disc)	T^2	$SE(2) \times S^1$
Constraint	holonomic, slips	dynamics, torque limits	nonholonomic, jackknife
What it knows	a blueprint; not the furniture, not the doors	the Workbench, exactly	the Lot, exactly
Task	navigate, replan	pick, fast and smooth	park, towing
Algorithm family	grid search on a belief	sampling, then time scaling	lattice search with analytic shots
What it cannot promise	anything, if the belief is lost	a solution in finite time	a path the car search never proposes

Each column is a different chapter of this book, and each bottom row is a different *kind* of incompleteness. D* Lite on an inflated grid is resolution-complete for a robot whose position it knows; sampling is probabilistically complete, which promises nothing about any particular deadline; hybrid A* is resolution-complete with respect to its own expansion primitives and to nothing else. There is no single planner that is good at all three, and even if there were, the guarantees would still be different statements about different objects.

The second reason is time. A plan is computed for the world as it was when the planner last looked. By the time it reaches the wheels, the scan it was built on is a tenth of a second old, the belief a little older, the map older still; if the door shut in between, the plan is a confident description of a world that no longer exists. A stack is how a robot keeps those ages bounded — and the mathematics of this chapter is mostly the mathematics of adding them up.

INTERACTIVE · w23.2

Stack Anatomy

An architecture diagram is not decoration: the arrows carry measured traffic, and removing a box has a visible, causal consequence.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch23-capstone#w23.2

The block diagram above is not a picture of the stack; it is the stack, replaying a recorded mission. The numbers on the arrows are the messages per second each stage actually emitted over the last simulated second, and the amber tint on each block is how old its newest input was when it last ran. Rusty's chain is the deepest — scan, belief, map, plan, control — and its control stage at 20 Hz reads a plan that can be half a second old. Reach's stages run at the base rate but emit only when they have work: the planner is silent between picks. Switch a block off and the mission re-records with that stage dead from three seconds in, which turns each sentence in the contract card into something you can watch happen.

Honesty item, stated once and kept: **the three panels do not share a world.** This is three stacks on one scheduler, not a team. Nothing Reach does can block Rusty; nothing Hitch does needs Rusty's permission. Coupled robots are Chapter 14's subject, and the composite configuration spaces there are exactly what three-robot coordination would require.

23.3 Every chapter, running

The capstone imports; it does not reimplement. The table is exhaustive. *Artifact* means the chapter's code is linked into `lib/capstone` and executed by the browser port (and into `crates/capstone` on the Rust side); *background* means the chapter's idea is load-bearing but no code from it runs.

Chapter	Contribution to the capstone	Kind
Ch. 1 (Chapter 1)	Task × robot × algorithm taxonomy (Choset Table 1.1); path vs trajectory; the completeness vocabulary the composed guarantee is written in	background
Ch. 2 (Chapter 2)	The three robots and worlds; the `Collision` contract (disc, chain, footprint checkers) that every dense recheck calls; the range sensor	artifact
Ch. 3 (Chapter 3)	Tangent Bug as Rusty's complete local fallback when the inflated grid has no path	artifact

Chapter	Contribution to the capstone	Kind
Ch. 4 (Chapter 4)	Disc inflation for Rusty's grid; Reach's Jacobian, $\lvert\det J\rvert$ and σ_{min} — the singularity guard	artifact
Ch. 5 (Chapter 5)	The manifolds every stage is generic over: T^2 for Reach's planner and shortcut, $\mathbb{R}^2$ for Rusty's	artifact
Ch. 6 (Chapter 6)	D* Lite on Rusty's inflated grid; the grid raster of the blueprint; A* for the final-map optimum	artifact
Ch. 7 (Chapter 7)	The brushfire distance transform under CHOMP's distance field	artifact
Ch. 8 (Chapter 8)	Accessibility, connectivity, departability — the vocabulary of a stage contract	background
Ch. 9 (Chapter 9)	Why Rusty does not trace the GVG online: at Apartment scale a repaired grid plan is cheaper	background
Ch. 10 (Chapter 10)	Coverage — the one Table 1.1 task the capstone does not run	background
Ch. 11 (Chapter 11)	The `Steer` contract and greedy shortcutting; Reach's `FreeSpace` over the chain checker	artifact
Ch. 12 (Chapter 12)	RRT-Connect: Reach's first feasible path and the source of its probabilistic completeness	artifact
Ch. 13 (Chapter 13)	Informed RRT*, anytime inside an iteration budget	artifact
Ch. 14 (Chapter 14)	Composite manifolds and coupling — why three robots in three worlds are three stacks, not a team	background
Ch. 15 (Chapter 15)	The 3σ margin and the covariance vocabulary of the belief gate (the browser port reads σ from Chapter 16's cloud)	background
Ch. 16 (Chapter 16)	The particle localizer Rusty plans from; `replan_on_belief` scoring the first move on D* Lite's field	artifact
Ch. 17 (Chapter 17)	Reach's $M(q)$, $C(q, \dot{q})$, $g(q)$ and torque limits	artifact
Ch. 18 (Chapter 18)	Path-constrained dynamics, the time-optimal Time-Scaling Algorithm, the `Trajectory<M>` type	artifact
Ch. 19 (Chapter 19)	CHOMP smoothing for Reach; the receding-horizon iLQR tracker for Rusty	artifact

Chapter	Contribution to the capstone	Kind
Ch. 20 (Chapter 20)	The car-with-trailer drive field behind the hitch-angle rate of the jackknife guard	artifact
Ch. 21 (Chapter 21)	Hybrid A* with Reeds–Shepp shots; the trailer simulated along a car path; the path scorer	artifact
Ch. 22 (Chapter 22)	The optional learned sampler for Reach's RRT-Connect, shipped only inside a uniform floor	artifact
App. A–F	Rust idioms, topology card, collision reference, completeness definitions, linear-systems card, simulator framework	background

Two rows deserve a sentence. Chapter 15 is *background* in the browser port: the belief Rusty plans from is Chapter 16's particle cloud, and the 0.15 m inflation margin is three standard deviations of a 5 cm localization error — Chapter 15's vocabulary, applied to a cloud. And Chapter 3 is an artifact that the default missions never exercise: Tangent Bug takes over only when D* Lite reports no path on the inflated grid, which happens when the map has sealed off a route the true world still has. A fallback that never runs in the demo is still part of the contract, and the contract card in the Stack Anatomy says so.

The capstone's own worlds add exactly two things to Chapter 2's. The Apartment's floor plan is a tree — every room has one door — so a shut door would simply end Rusty's mission; the architect's drawing Rusty is given therefore has one more opening, a connecting door between the study and the bedroom. And the furniture is new: four curated placements, of which a seed picks two (seed 7: a cart in the corridor and a basket in the bedroom). Curated, not uniform, because a box dropped in a doorway makes the mission infeasible, and the capstone's subject is replanning, not infeasibility.

23.4 The mathematics

Choset has no systems chapter, so the formal content here is new, but it is small: four definitions and four derivations, each of which composes results proved elsewhere in the book. The notation adds to TOC §2 (Chapter 1) and changes nothing in it.

$\mathcal{S}_i$, τ_i, ς_i	stage i; its period; its worst-case staleness as a producer ς_i = τ_i for a periodic stage
T_{react}	worst-case sense-to-command delay through a pipeline F2

d_{detect}, v_{max}, a_{max}	distance at which a new obstacle is first trusted; Rusty's speed and braking limits 2.0 m, 0.6 m/s, 1.0 m/s^2
L_{valid}	validity horizon: the path prefix re-checked against the newest map every control tick D23.2
$\mu(q) = \lvert\det J(q)\rvert = l_1 l_2 \lvert\sin\theta_2\rvert$	Reach's manipulability (2R, square Jacobian) Ch. 4; μ_min = 0.25
$\sigma_{min}(J)$	smallest singular value of the Jacobian $\lVert\dot{q}\rVert \le \lVert\dot{x}\rVert / \sigma$_min
$\phi = \theta - \psi$	hitch angle: car heading minus trailer heading Ch. 2's convention; d = 2.5 m
$\phi_{plan} < \phi_{warn} < \phi_{max}$	trailer filter bound; guard threshold; mechanical jackknife 0.85, 0.90, 1.30 rad
$\mathcal{G}$	a guarantee: complete $\succ$ resolution-complete $\succ$ probabilistically complete $\succ$ none a total order

23.4.1 Four definitions

D23.1 — Stage and contract. A stage $\mathcal{S}_i$ consumes stamped inputs and emits one stamped output at period τ_i. Its *contract* names the input type, the output type, the frame both are expressed in, and the guarantee $\mathcal{G}_i$ the stage provides **given valid inputs**. The types are the contract: a `Path<M>` is a geometric curve $c : [0,1] \to \mathcal{Q}_{\text{free}}$; a `Trajectory<M>` is a path with a clock, $q(t)$ for $t \in [0, t_f]$ (Choset §1.3 (Chapter 1), Chapter 18). Nothing executes a `Path`.

D23.2 — Validity horizon. $L_{valid} = v_{max}\,(\tau_{plan} + \varsigma_{map})$: the prefix of the current path the robot may drive before the next plan can possibly arrive. Every control tick, that prefix is re-checked against the *newest* map — which may be newer than the map the path was planned on — and the robot brakes if any cell in it is occupied.

D23.3 — Composed guarantee. Call a stage *rejecting* if it can answer "no" to a query (a planner, a filter). Then

$$\mathcal{G}(\text{pipeline}) \;=\; \min_{i\,:\,\mathcal{S}_i \text{ rejects}} \mathcal{G}_i$$

provided every post-processor downstream of a planner either verifies its output or falls back to its input; if any post-processor can silently break its input's property, $\mathcal{G}$ = none.

D23.4 — Mode. The supervisor's discrete state per robot is an exhaustive enum, `Plan | Execute | Replan | Recover(kind) | Done | Failed(reason)`, and its transition function is total: every (mode, event) pair lands on a mode. Recoveries carry parameters — `PullGoalInward { delta }`, `PullForward { until_phi }`, `TangentBug` — because "recover" is not a plan.

23.4.2 F1 — Guarantee composition: the weakest link, with verified post-processors

Statement. Rusty's pipeline is resolution-complete *given a localized belief and a world that is static between detections*; Reach's is probabilistically complete; Hitch's is resolution-complete for the car alone and promises **nothing** for the car with its trailer, because the trailer filter can reject every candidate the car search proposes.

The sketch has four steps. List the stages and their $\mathcal{G}_i$. Identify which of them can reject a query. Check each post-processor for a fallback. Take the minimum. The interesting work is in the third step, and in being honest in the first.

DERIVATION **F1 for the three pipelines, stage by stage**

Rusty: resolution-complete (conditional) · Reach: probabilistically complete · Hitch: none for the trailer

Rusty. The stages and their roles:

Stage	Role	$\mathcal{G}_i$ / property
scan (Ch. 2)	source	—
belief (Ch. 16)	source	localized only while $\sigma \leq 5$ cm, which the map stage enforces
inflated grid (Ch. 4, Ch. 6)	source	the C-obstacle of the disc, sampled at cell centres
D* Lite (Ch. 6)	rejecting	resolution-complete on the inflated grid it is given
D23.2 check + MPC (Ch. 19)	verified post-processor	never drives a prefix the newest map marks occupied; falls back to braking and replanning
Tangent Bug (Ch. 3)	fallback	complete for a point with clearance W^* in a static plane

The only rejecting stage is D* Lite, so the pipeline inherits *resolution completeness on the grid it was given*. Two conditions hide in that phrase, and the text states both rather than letting the reader infer more. First, the grid is built by projecting scans through the belief; if the belief is lost the grid is wrong and no completeness statement about it says anything about the Apartment. The map stage therefore refuses to insert obstacles while $\sigma > 5$ cm — the condition is enforced, not assumed. Second, D* Lite is resolution-complete for a static graph; between two repairs the world must not change faster than the budget F2 allows. Tangent Bug is the fallback for the case in which the grid says "no path" but the world disagrees.

Reach.

Stage	**Role**	$\mathcal{G}_i$ **/ property**
singularity guard (Ch. 4)	guard	rejects a goal with $\mu < \mu_{min}$, recovers by moving it
RRT-Connect (Ch. 12)	rejecting	probabilistically complete
Informed RRT* (Ch. 13)	anytime improver	returns the better of its own best and RRT-Connect's path
shortcut (Ch. 11)	verified post-processor	every replaced piece is a steered, checked segment
time scaling (Ch. 18)	verified post-processor	the spline is densely rechecked; if it cuts a corner, the polyline is scaled segment by segment, rest to rest
CHOMP (Ch. 19)	**unverified** alone	can push a waypoint into the block
verify (Ch. 2)	guard	swept recheck of CHOMP's trajectory; on failure, falls back to the time-scaled shortcut

CHOMP alone would make the composed guarantee *none*: it is a local optimizer of a soft penalty, and nothing in its objective forbids collision. Wrapped by the verifier it becomes a verified post-processor, and the pipeline is probabilistically complete — RRT-Connect's guarantee. Switch the verifier off in Stack Anatomy and the $\mathcal{G}$ printed under Reach's row drops to *none* whether or not this particular seed collides, because the guarantee is a property of the pipeline, not of the run.
The Chapter 22 toggle leaves this row unchanged. The learned sampler only ever ships inside the mixture $(1-\lambda)\hat{p}_\theta + \lambda u$ with $\lambda > 0$, so every set A of positive measure is still sampled with probability at least $\lambda\,\mu(A)/\mu(\boldsymbol{Q})$ per draw — the hypothesis probabilistic completeness needs.
Hitch.

Stage	Role	$\mathcal{G}_i$ / property
hybrid A* + RS shots (Ch. 21)	rejecting	resolution-complete w.r.t. its primitives, *for the car*
trailer filter (Ch. 20)	rejecting	none: it tests, it does not search
jackknife guard (Ch. 20)	guard	fires on $\lvert\phi\rvert > \phi_{warn}$ while reversing
segment tracker (Ch. 21)	executor	replays the words exactly on Chapter 2's `Hitch.step`

Chapter 21's hybrid A* expands the car; the trailer is simulated along each candidate path afterwards and the candidate is rejected if the trailer touches anything or folds past ϕ_{plan}. A filter that can say "no" without searching for an alternative is itself a rejecting stage with $\mathcal{G}$ = none, so the minimum is none. The retry ladder (finer cells, reverse penalties, staging poses) widens the set of candidates the filter sees; it does not close the gap. What would close it is a search with the trailer in its state: Chapter 21's CAR GRID SEARCH with the hitch angle as a fourth grid axis and jackknife pruning is resolution-complete *for the trailer*, at the price of two orders of magnitude more expansions. The panels display `composed_guarantee` — none — rather than letting the reader infer more from a successful run. (Switch the filter *off* in Stack Anatomy and the displayed guarantee rises to resolution-complete: completeness for the car, bought by giving up every claim about the trailer. Completeness and soundness are different promises.)

23.4.3 F2 — The replanning budget: staleness plus braking

A corridor that becomes blocked is first *trusted* at some distance d_{detect} ahead: the sensor sees it earlier, but two consecutive scans must agree before a cell

becomes an obstacle, and the stack trusts returns only within 2 m, where its 5° beam spacing still puts five rays across a 0.9 m door. From that moment, the information has to travel scan → belief → map → plan → control, and then the wheels have to stop.

$$v_{max}\,T_{react} + \frac{{v_{max}}^2}{2\,a_{max}} < d_{detect}, \qquad T_{react} = \varsigma_{scan} + \varsigma_{bel} + \varsigma_{map} + \tau_{plan} + \tau_{ctrl}.$$

DERIVATION **Chaining the delays**

T_react = Σ periods; stop = v T_react + v²/2a

1. **Worst-case staleness per stage.** A periodic stage that reads its input just *before* the producer publishes sees a message almost one producer period old. So each link in the chain can add up to one period: $\varsigma_{scan} = 1/f_{scan}$, $\varsigma_{bel} = 1/f_{bel}$, $\varsigma_{map} = 1/f_{map}$. The planner itself takes up to one of its own periods to produce the repaired path, and the controller up to one of its periods to act on it: τ_{plan}, τ_{ctrl}. Summing gives T_{react}.
2. **Reaction distance.** During T_{react} the robot keeps its current speed, at most v_{max}, and covers $v_{max}\,T_{react}$.
3. **Braking distance.** Decelerating at a_{max} from v_{max} takes $v_{max}^2/2a_{max}$. The corridor is handled iff the sum is less than the distance at which the blockage was trusted.

The **moving-obstacle variant** adds the obstacle's own closing distance, $v_{obs}T_{react}$, to the left side; Exercise 1 asks for the fastest walker the default rates tolerate.

The micro-example, with the book's default rates — range sensor 10 Hz, Chapter 16 belief 10 Hz, inflated-grid update 5 Hz, D* Lite 2 Hz, MPC 20 Hz — and Rusty's $v_{max} = 0.6$ m/s, $a_{max} = 1.0$ m/s^2, $d_{detect} = 2.0$ m:

$$T_{react} = 0.10 + 0.10 + 0.20 + 0.50 + 0.05 = 0.95\ \text{s}, \qquad 0.6 \times 0.95 + \frac{0.6^2}{2 \times 1.0} = 0.570 + 0.180 = \mathbf{0.750\ m} < 2.0\ \text{m}.$$

The margin is 1.25 m. The fastest Rusty could safely drive with these rates solves $v^2/2 + 0.95\,v - 2.0 = 0$:

$$v_{safe} = -0.95 + \sqrt{0.9025 + 4.0} = \mathbf{1.264\ m/s}, \qquad 1.264 \times 0.95 + \tfrac{1.264^2}{2} = 1.201 + 0.799 = 2.000.$$

And the validity horizon of D23.2 is $L_{valid} = 0.6 \times (0.50 + 0.20) = \mathbf{0.42\ m}$. These are the first line the worked example prints (`replanning_budget`), and the check file pins all five numbers to 10^{-3}.

Why have both F2 and D23.2? Because F2 is a statement about the *worst case of a schedule*, and D23.2 is a run-time check of *this* path against *this* map. Throttle the replanner to 0.5 Hz in the Failure Tour and the budget still closes — T_{react} grows to 2.45 s, the stopping distance to 1.65 m — but the validity horizon grows to $0.6 \times 2.2 = 1.32$ m, and it is the D23.2 check, not the planner, that brakes Rusty in front of the door. The budget says the stack *can* stop in time; the check is what makes it actually do so when the plan is stale.

23.4.4 F3 — The singularity guard, from Chapter 4's Jacobian

Statement. For a commanded tip velocity $\dot{x}$, the joint velocity satisfies $\|\dot{q}\| \leq \|\dot{x}\|/\sigma_{min}(J)$, and for the 2R arm $\sigma_{min} \to 0$ exactly when $\mu(q) = l_1 l_2 |\sin\theta_2| \to 0$. The guard rejects a goal with $\mu(q_{goal}) < \mu_{min}$.

DERIVATION **From the SVD to a threshold you can defend**

$\mu_m in = 0.25 \Rightarrow \sigma_m in = 0.1127 \Rightarrow \|\dot{q}\| \leq 1.775 rad/sat 0.2 m/s$

1. **Resolved rates.** Away from singularities $J(q)$ is invertible and the approach controller commands $\dot{q} = J^{-1}\dot{x}$ (Chapter 4).
2. **The bound.** With the SVD $J = U\Sigma V^\top$, $J^{-1} = V\Sigma^{-1}U^\top$ and $\|J^{-1}\|_2 = 1/\sigma_{min}$, so $\|\dot{q}\| \leq \|\dot{x}\|/\sigma_{min}$, with equality when $\dot{x}$ points along the left singular vector of σ_{min} — for an arm at full stretch, *radially*.
3. **The 2R determinant.** Chapter 4 computed $\det J = l_1 l_2 \sin\theta_2$; since $|\det J| = \sigma_1\sigma_2$ and σ_1 is bounded above by the arm length, $\sigma_{min} \to 0$ if and only if $\sin\theta_2 \to 0$: arm straight or folded.
4. **The threshold.** Reach's last 12 cm of every pick is a straight Cartesian approach at $\|\dot{x}\| = 0.2$ m/s, and its joints are limited to 2 rad/s. At $\mu_{min} = 0.25$ the smallest singular value is $\sigma_{min} = 0.1127$, so the approach needs at most $0.2/0.1127 = 1.775$ rad/s — under the limit. Any goal the guard admits is one the approach can reach without saturating a joint; any goal it rejects is one where the bound is no longer a promise.

The **recovery** is `Recover(PullGoalInward { delta: 0.15 })`: move the tip target 15 cm toward the base and run the guard again. A full-stretch target at $r = 2$ m becomes $r = 1.85$ m, where $\cos\theta_2 = (r^2 - 2)/2$ gives $\mu = 0.703$. Nothing else in the pipeline changes: joint-space planning and time scaling are singularity-agnostic — their limits are torques, not Cartesian speeds — which is exactly why the guard belongs in front of the Cartesian approach and nowhere else.

For the redundant 3R variant the same argument runs with $\mu = \sqrt{\det JJ^\top}$, the product of the two singular values of the 2×3 Jacobian.

23.4.5 F4 — The jackknife guard, from Chapter 20's trailer field

Statement. With the trailer hitched at the rear axle, Chapter 20's drive field $g_1 = [\cos\theta,\ \sin\theta,\ 0,\ \tan\delta/L,\ \sin(\theta-\psi)/d]^\top$ gives the hitch-angle rate

$$\dot{\phi} \;=\; \frac{v}{L}\tan\delta \;-\; \frac{v}{d}\sin\phi \quad \xrightarrow{\delta=0,\ \phi\approx 0} \quad \dot{\phi} \approx -\frac{v}{d}\,\phi,$$

stable forward ($v > 0$), unstable in reverse with rate $|v|/d$. The guard fires on $|\phi| > \phi_{warn}$ while $v < 0$.

DERIVATION **Why reversing a trailer is the dangerous direction**

$\phi(s) \approx \phi_0 e^(s/d) in reverse; e - folding distance d = 2.5m$

1. **Two headings.** The car turns at $\dot{\theta} = (v/L)\tan\delta$; the trailer, dragged by the hitch at the rear axle, turns at $\dot{\psi} = (v/d)\sin(\theta-\psi)$ — Chapter 2's simulator integrates exactly this.
2. **Their difference.** $\dot{\phi} = \dot{\theta} - \dot{\psi}$ is the displayed equation; the check file evaluates it through Chapter 20's `carTrailer` field and compares with the closed form.
3. **Linearize.** With the wheel straight and ϕ small, $\dot{\phi} \approx -(v/d)\phi$. Per metre travelled, $d\phi/ds = -\operatorname{sign}(v)\,\phi/d$.
4. **Sign of v.** Forward, ϕ decays like $e^{-s/d}$; in reverse it grows like $e^{s/d}$ — every 2.5 m of straight reversing multiplies a small hitch angle by e.

The **recovery** is `Recover(PullForward { until_phi: 0.15 })`: drive forward — the stable direction — steering toward the trailer to straighten it faster, until $|\phi| < \phi_{ok} = 0.15$ rad, then replan from where the car is, with the trailer's current heading. The thresholds are ordered on purpose: $\phi_{plan} = 0.85 < \phi_{warn} = 0.90 < \phi_{max} = 1.30$ rad. The planner leaves headroom, the guard sits in it, and the hard bound is the mechanism's. ϕ_{max} is a *state* constraint layered on the Pfaffian one: the nonholonomic constraint says which velocities exist; the jackknife bound says which configurations are allowed, and only a guard or a planner that knows about it enforces it.

23.5 The algorithms

Every box below is a composition of algorithms numbered elsewhere in the book; the line that does the real work is always a call. What is new is the order, the rates, and what happens when a line says no.

ALGORITHM `pipeline_tick(bus, stages, now) -- the WASM round-robin` COST the sum of the due stages' costs; 60 fps kept by time-slicing the expensive ones

in the stamped bus, stages in topological order with periods τ_i and phases, the simulated time now (integer ms)

out every due stage run once; their outputs published with stamp now

1: **for** each stage $\mathcal{S}_i$ in topological order (producers before consumers):
2: **if** $\mathcal{S}_i$ is enabled **and** $(now - phase_i) \bmod \tau_i = 0$:
3: read the newest message on each input topic; record its age $now - t$ (the measured ς)
4: out $\leftarrow \mathcal{S}_i$.tick(inputs, now) — a *bounded* chunk of work: 40 tree iterations, 10 CHOMP steps, one hybrid A* rung
5: **if** out is some message: publish it stamped (now, frame of $\mathcal{S}_i$'s output)
6: advance now by the base period (50 ms); integer arithmetic, so two replays never drift apart

ALGORITHM `rusty_stack -- one round of Rusty's pipeline` COST belief O(N·beams); D* Lite incremental in the changed cells; MPC O(H) per iteration at 20 Hz

in the blueprint, the belief (Ch. 16), the newest scan, the inflated grid, D* Lite's state (Ch. 6), the goal

out a twist (v, ω) for the wheels, or a mode event

1: **scan** (10 Hz): 72 beams of the range sensor in the *true* world
2: **belief** (10 Hz): predict with odometry, correct with every third beam against the blueprint, resample when ESS falls below N/2
3: **map** (5 Hz): **if** $\sigma_{belief} > 5$ cm: trust nothing this round; **else** project returns within d_{detect} through the belief, keep those the blueprint does not explain, and inflate by $r + 0.15$ m every cell hit in two consecutive rounds
4: **plan** (2 Hz): set D* Lite's start to the belief's cell; UpdateCells(new changes) — Koenig and Likhachev's repair; path $\leftarrow$ greedy descent, shortcut with Chapter 11's Steer; score the first move with `replan_on_belief`; **if** no path: Recover(TangentBug)
5: **control** (20 Hz): **if** the D23.2 prefix of length L_{valid} hits an occupied cell of the newest map: Blocked $\rightarrow$ Replan, brake; **else** iLQR over 10 steps toward reference points along the path (turn in place first if the path doubles back)

ALGORITHM `reach_stack -- one pick` COST RRT* O(n log n) for n ≤ 400; time scaling O(N_s) on 400 grid nodes; CHOMP O(k N_w) for 30 steps on 24 waypoints

in a tip target, Reach's state q, the Workbench, torque limits (20, 12) N·m

out an executed, verified Trajectory<T²> ending in a straight 12 cm Cartesian approach

1: IK both elbow branches for the target and the pre-grasp 12 cm before it; keep the collision-free branch nearest q
2: **singularity_guard**: **if** $\mu(q_{goal}) < \mu_{min}$: Recover(PullGoalInward δ); move the target; goto 1
3: RRT-Connect to the pre-grasp (Ch. 12), 40 attempts per tick; then Informed RRT* (Ch. 13) for a budget of 400 iterations
4: path ← the cheaper of the two; shortcut it (Ch. 11)
5: spline through the shortcut path resampled at 26 knots; **if** a dense swept check fails: scale the polyline segment by segment, rest to rest; T_1 ← time_scale (Ch. 18)
6: CHOMP (Ch. 19) from the same 24 waypoints, 10 steps per tick; spline; T_2 ← time_scale
7: **verify**: **if** the T_2 trajectory passes the swept recheck **then** execute it **else** execute T_1
8: approach: $\dot{q} = J^{-1}\dot{x}$ at 0.2 m/s for 12 ticks; **if** $\|\dot{q}\|_\infty > 2$ rad/s: Fault(JointRate)

ALGORITHM `hitch_stack -- plan, filter, guard` COST one hybrid A* rung per tick (≤ 4000 expansions); filter O(samples); guard O(1) at 20 Hz

in Hitch's configuration (x, y, θ, ψ), the Lot, the goal pose in bay 3, the retry ladder

out a segment list executed on Chapter 2's Hitch.step, or Failed(reason)

1: **plan**: hybrid A* with Reeds–Shepp shots on the car (Ch. 21), rung k of the ladder: (cell, reverse penalty, optional staging pose)
2: **filter**: simulate ψ along the candidate from the trailer's current heading; **if** the trailer touches anything **or** max$|\phi| > \phi_{plan}$: reject, $k \leftarrow k + 1$; **if** the ladder is exhausted: Failed(NoPath)
3: **track**: replay the words at $|v| = 1$ m/s, steering $\pm\phi_{steer,max}$ or straight, never overshooting a segment
4: **guard** (20 Hz): **if** $v < 0$ **and** $|\phi| > \phi_{warn}$: Recover(PullForward); **if** $|\phi| > \phi_{max}$: Failed(Jackknife)
5: **recover**: forward at 0.8 m/s steering toward the trailer until $|\phi| < \phi_{ok}$; then $k \leftarrow 0$ and replan from here

ALGORITHM `supervisor_step(mode, event) -- D23.4` COST `O(1); total over 6 modes × 10 events`

in the robot's mode, one event from its stages

out the next mode

1: **if** event is Fault(reason) **and** mode is live: **return** Failed(reason)
2: **match** mode: Plan or Replan — PlanReady → Execute; NoPath(r) → Recover(r) or Failed(NoPath); Singular(δ) → Recover(PullGoalInward δ); Jackknife → Recover(PullForward)
3: Execute — Blocked → Replan; Arrived → Done; Jackknife → Recover(PullForward)
4: Recover — Recovered → Replan; PlanReady → Execute (Tangent Bug handing back to the grid)
5: Done — NextTask → Plan; Failed — absorbing
6: every other pair leaves the mode unchanged — and the compiler, not this list, is what proves there is no pair left out

23.6 Implementation in Rust

The capstone crate adds one dependency to the workspace — `crossbeam-channel` 0.5, for the native bus — and otherwise only imports: `nalgebra` 0.35, `parry2d` 0.30, `petgraph` 0.8 and `rand` 0.9 arrive through the Chapter 2–22 crates it links. The module plan:

```
crates/capstone/src/
  bus.rs          Stamped<T, F>, frame markers, the native channels and the WASM slots
  stage.rs        trait Stage, Guarantee, composed_guarantee, ThreadRunner (native),
   RoundRobin (wasm)
  stacks/rusty.rs Ch. 16 belief -> Ch. 4 inflation -> Ch. 6 D* Lite -> D23.2 -> Ch. 19
   MPC; Ch. 3 fallback
  stacks/reach.rs Ch. 4 guard -> Ch. 12 -> Ch. 13 -> Ch. 11 -> Ch. 18 -> Ch. 19 -> verify
    (Ch. 2)
  stacks/hitch.rs Ch. 21 hybrid A* -> Ch. 20 trailer filter -> tracker; jackknife guard
  guards.rs       singularity_guard (F3), jackknife_guard (F4), corridor_blocked (D23.2)
  budget.rs       replanning_budget (F2) and RateTable::DEFAULT
  supervisor.rs   Mode, Recover, supervisor_step
  mission.rs      MissionCfg, run_mission, MissionReport, format_mission
```

23.6.1 The contract, as types

The first listing is the whole of D23.1 and D23.3. One deviation from the design sketch is deliberate: frames are **zero-sized marker types** rather than a runtime `FrameId(&'static str)`. A string frame can only be checked when the program runs; a marker type makes "a map-frame pose handed to the odom-frame belief" a compile error, which is the third vignette of the retrospective.

RUST **crates/capstone/src/stage.rs**

```rust
use std::marker::PhantomData;

/// Simulated time in integer milliseconds. Floating-point seconds would turn
/// 0.05 + 0.05 + ... into 0.30000000000000004 and two stages with equal periods
/// would drift out of phase; integers keep replays bit-identical.
pub type SimTime = u64;

/// Frames are types, not strings: a mix-up is rejected by rustc, not at run time.
pub trait Frame: Copy + Send + 'static {
    const NAME: &'static str;
}
#[derive(Clone, Copy, Debug)] pub struct Map;
#[derive(Clone, Copy, Debug)] pub struct Odom;
#[derive(Clone, Copy, Debug)] pub struct Base;
#[derive(Clone, Copy, Debug)] pub struct Torus;
#[derive(Clone, Copy, Debug)] pub struct Lot;
impl Frame for Map { const NAME: &'static str = "map"; }
impl Frame for Odom { const NAME: &'static str = "odom"; }
// ... Base, Torus, Lot likewise.

/// D23.1: nothing crosses a stage boundary without a time and a frame on it.
#[derive(Clone, Debug)]
pub struct Stamped<T, F: Frame> {
    pub t: SimTime,
    pub v: T,
    frame: PhantomData<F>,
}

impl<T, F: Frame> Stamped<T, F> {
    pub fn new(t: SimTime, v: T) -> Self { Self { t, v, frame: PhantomData } }
    pub fn age(&self, now: SimTime) -> SimTime { now.saturating_sub(self.t) }
}

/// ?, weakest first, so `Ord::min` over stages is "the weakest link".
#[derive(Clone, Copy, Debug, PartialEq, Eq, PartialOrd, Ord)]
pub enum Guarantee { None, ProbabilisticallyComplete, ResolutionComplete, Complete }

#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum Role { Source, Rejecting, Verified, Unverified, Executor, Guard }

pub trait StageInfo {
    fn id(&self) -> &'static str;
    fn period_ms(&self) -> SimTime;
    fn role(&self) -> Role;
    /// What this stage promises *given valid input* (D23.1).
    fn guarantee(&self) -> Guarantee;
}

/// One pipeline stage. The same impl runs on its own thread (native) or
/// inside the round-robin (wasm): it cannot tell which.
pub trait Stage: StageInfo + Send {
    type In;
    type Out;
    type FIn: Frame;
    type FOut: Frame;
    fn tick(&mut self, input: &Stamped<Self::In, Self::FIn>, now: SimTime)
        -> Option<Stamped<Self::Out, Self::FOut>>;
}

```

```rust
/// D23.3: the minimum over rejecting stages -- and nothing at all if any
/// post-processor can silently break the property it was handed.
pub fn composed_guarantee(stages: &[&dyn StageInfo]) -> Guarantee {
    if stages.iter().any(|s| s.role() == Role::Unverified) {
        return Guarantee::None;
    }
    stages
        .iter()
        .filter(|s| s.role() == Role::Rejecting)
        .map(|s| s.guarantee())
        .min()
        .unwrap_or(Guarantee::None)
}
```

`Path<M>` and `Trajectory<M>` are Chapter 18's, unchanged; the capstone adds no types there. What it adds is the *place* where the distinction bites: Reach's tracker is a `Stage` whose `In` is `Trajectory<Torus2>`, so wiring the shortcut stage's `Path<Torus2>` output into it does not compile. In the browser port, where TypeScript has no phantom frames and no exhaustiveness the compiler enforces, the same mistake is a runtime refusal: switch the time-scaling block off in Stack Anatomy and the tracker fails the task with the rustc message as its reason.

23.6.2 Two runtimes, one schedule

RUST **crates/capstone/src/stage.rs (continued)**

```rust
/// Native: one thread per stage, a bounded channel per edge. A full channel
/// drops the *oldest* message -- a stage always wants the newest one.
pub struct ThreadRunner { handles: Vec<std::thread::JoinHandle<()>> }

impl ThreadRunner {
    pub fn spawn<S>(mut stage: S, rx: crossbeam_channel::Receiver<Stamped<S::In, S::FIn>>,
                    tx: crossbeam_channel::Sender<Stamped<S::Out, S::FOut>>, clock: Clock
    ) -> Self
    where S: Stage + 'static, S::In: Send + 'static, S::Out: Send + 'static {
        let period = std::time::Duration::from_millis(stage.period_ms());
        let h = std::thread::spawn(move || loop {
            let Some(input) = rx.try_iter().last() else { std::thread::sleep(period);
    continue };
            if let Some(out) = stage.tick(&input, clock.now()) { let _ = tx.try_send(out)
    ; }
            std::thread::sleep(period);
        });
        Self { handles: vec![h] }
    }
}

/// WASM: no threads. Every stage whose period divides `now - phase` runs once,
/// in topological order, inside the animation frame -- same stage code, same
/// messages, same seeds, so a mission replays bit for bit.
pub struct RoundRobin { entries: Vec<Box<dyn Runnable>> }

impl RoundRobin {
    pub fn tick(&mut self, bus: &mut Bus, now: SimTime) {
```

```
        for e in self.entries.iter_mut().filter(|e| e.enabled()) {
            if now >= e.phase() && (now - e.phase()) % e.period_ms() == 0 {
                e.run(bus, now); // reads the newest slot, publishes Stamped::new(now,
    ...)
            }
        }
    }
}
```

The native runner is *nondeterministic* — thread interleavings differ run to run — which is why the book's regression mission runs on the round-robin even natively, and why the chapter never claims the browser demonstrates real-time scheduling. What it demonstrates is throughput: fifteen stages keep their rates inside a frame budget, with every expensive one sliced into bounded chunks.

23.6.3 The supervisor and its guards

RUST **crates/capstone/src/supervisor.rs**

```rust
use crate::guards::{Jackknife, Singular};

#[derive(Clone, Debug, PartialEq)]
pub enum Recover { PullGoalInward { delta: f64 }, PullForward { until_phi: f64 },
   TangentBug }

#[derive(Clone, Debug, PartialEq)]
pub enum Mode { Plan, Execute, Replan, Recover(Recover), Done, Failed(FailReason) }

#[derive(Clone, Debug, PartialEq)]
pub enum FailReason { NoPath, Collision, Timeout, Jackknife, JointRate, Boxed }

pub enum Event {
    PlanReady, NoPath(Option<Recover>), Blocked { at: f64 }, Arrived, NextTask,
    Singular(Singular), Jackknife(Jackknife), Recovered, Fault(FailReason),
}

/// D23.4, total by construction: add a variant to `Mode`, `Recover` or `Event`
/// and every match that forgot it stops compiling (E0004).
pub fn supervisor_step(mode: Mode, ev: Event) -> Mode {
    use Mode::*;
    match (mode, ev) {
        (Failed(r), _) => Failed(r),
        (Done, Event::NextTask) => Plan,
        (Done, _) => Done,
        (_, Event::Fault(r)) => Failed(r),
        (Plan | Replan, Event::PlanReady) => Execute,
        (Plan | Replan, Event::NoPath(Some(r))) => Recover(r),
        (Plan | Replan, Event::NoPath(None)) => Failed(FailReason::NoPath),
        (Plan | Replan | Execute, Event::Singular(_)) => Recover(Recover::PullGoalInward
   { delta: 0.15 }),
        (Plan | Replan | Execute, Event::Jackknife(_)) => Recover(Recover::PullForward {
   until_phi: 0.15 }),
        (Execute, Event::Blocked { .. }) => Replan,
        (Execute, Event::Arrived) | (Recover(_), Event::Arrived) => Done,
        (Execute, Event::NoPath(_)) => Replan,
```

```
        (Recover(_), Event::Recovered) => Replan,
        (Recover(Recover::TangentBug), Event::PlanReady) => Execute,
        (m, _) => m,
    }
}
```

There is no `PathNotTrajectory` variant in the Rust `FailReason`, and that absence is the point: the browser port needs one because TypeScript lets the mistake reach run time; the Rust supervisor never sees it, because the program containing it does not build.

RUST **crates/capstone/src/guards.rs**

```rust
use nalgebra::Matrix2;
use robots::{Reach, T2};

pub struct Singular { pub mu: f64, pub sigma_min: f64 }
pub struct Jackknife { pub phi: f64 }
pub struct Blocked { pub cell: usize, pub at: f64 }

/// F3. mu = |det J| = l_1l_2|sin theta_2|; ||q_dot|| <= ||x_dot||/sigma_min. Chapter 4
   owns `jacobian2`.
pub fn singularity_guard(arm: &Reach, q: &T2, mu_min: f64) -> Result<(), Singular> {
    let j: Matrix2<f64> = cspace::jacobian2(arm, q);
    let mu = j.determinant().abs();
    if mu >= mu_min { return Ok(()); }
    let sigma_min = j.svd(false, false).singular_values.min();
    Err(Singular { mu, sigma_min })
}

/// F4. Reversing is the unstable direction (phi_dot ~= -(v/d)phi), so only reverse can
   trip it.
pub fn jackknife_guard(phi: f64, v: f64, phi_warn: f64) -> Result<(), Jackknife> {
    if v < 0.0 && phi.abs() > phi_warn { Err(Jackknife { phi }) } else { Ok(()) }
}

/// D23.2: the first occupied cell of the *newest* map within L_valid along the path.
pub fn corridor_blocked(path: &Path<R2>, grid: &OccupancyGrid, from: R2, l_valid: f64) ->
    Option<Blocked> {
    let s0 = path.project(from);
    let step = grid.cell_size / 2.0;
    (1..)
        .map(|k| s0 + k as f64 * step)
        .take_while(|&s| s <= (s0 + l_valid).min(path.length()))
        .map(|s| (s - s0, grid.cell_at(path.point_at(s))))
        .find(|&(_, c)| grid.occupied(c))
        .map(|(at, cell)| Blocked { cell, at })
}
```

23.6.4 Verify or fall back

D23.3 lives or dies on post-processors that cannot silently break their input. Reach's smoothing stage is the one place in the stack where that is a design

decision rather than a property of the algorithm, so it is written out:

RUST **crates/capstone/src/stacks/reach.rs**

```rust
/// CHOMP is a local optimizer of a *soft* collision penalty: nothing in its objective
/// forbids contact. Wrapped like this it is a verified post-processor, and Reach's
/// composed guarantee stays RRT-Connect's (probabilistically complete).
pub struct ChompVerify { pub checker: ChainChecker, pub problem: ChompProblem, pub iters:
    usize }

impl Stage for ChompVerify {
    type In = Trajectory<Torus2>;       // T_1, already time-scaled (Ch. 18)
    type Out = Trajectory<Torus2>;
    type FIn = Torus;
    type FOut = Torus;

    fn tick(&mut self, t1: &Stamped<Trajectory<Torus2>, Torus>, now: SimTime)
        -> Option<Stamped<Trajectory<Torus2>, Torus>> {
        let xi = trajopt::chomp(&self.problem, &t1.v.waypoints(24), self.iters);
        let smoothed = CubicSpline::through(&xi);
        let t2 = trajectory::time_scale(&smoothed, &REACH_LIMITS).ok()?;
        // Dense swept recheck of every consecutive pair of samples (Ch. 2's sphere
   marching).
        let ok = t2.samples().windows(2).all(|w| self.checker.edge_free(&w[0].q, &w[1].q)
   );
        Some(Stamped::new(now, if ok { t2 } else { t1.v.clone() }))
    }
}
// StageInfo for ChompVerify: role() = Role::Verified. Remove the recheck and it is
// Role::Unverified -- and composed_guarantee() returns Guarantee::None, as it should.
```

23.6.5 The budget and the mission

RUST **crates/capstone/src/budget.rs**

```rust
pub struct RateTable { pub scan_hz: f64, pub belief_hz: f64, pub map_hz: f64, pub plan_hz:
    f64, pub ctrl_hz: f64 }

impl RateTable {
    pub const DEFAULT: Self = Self { scan_hz: 10.0, belief_hz: 10.0, map_hz: 5.0, plan_hz:
    2.0, ctrl_hz: 20.0 };
    /// T_react = ?_scan + ?_bel + ?_map + tau_plan + tau_ctrl: one period per link,
   worst case.
    pub fn t_react(&self) -> f64 {
        [self.scan_hz, self.belief_hz, self.map_hz, self.plan_hz, self.ctrl_hz].iter().
   map(|f| 1.0 / f).sum()
    }
}

pub struct BudgetReport { pub t_react: f64, pub stop: f64, pub margin: f64, pub v_safe:
   f64, pub l_valid: f64 }

/// F2. v T + v2/2a < d; v_safe is the positive root of v2/2a + T v - d = 0.
pub fn replanning_budget(r: &RateTable, v_max: f64, a_max: f64, d_detect: f64) ->
   BudgetReport {
    let t = r.t_react();
```

```
    let stop = v_max * t + v_max * v_max / (2.0 * a_max);
    let v_safe = a_max * (-t + (t * t + 2.0 * d_detect / a_max).sqrt());
    let l_valid = v_max * (1.0 / r.plan_hz + 1.0 / r.map_hz);
    BudgetReport { t_react: t, stop, margin: d_detect - stop, v_safe, l_valid }
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn budget_matches_hand_computation() {
        let b = replanning_budget(&RateTable::DEFAULT, 0.6, 1.0, 2.0);
        assert!((b.t_react - 0.950).abs() < 1e-12 && (b.stop - 0.750).abs() < 1e-12);
        assert!((b.v_safe - 1.264).abs() < 1e-3 && (b.l_valid - 0.42).abs() < 1e-12);
    }
}
```

The worked example runs all three pipelines on seed 7 with the three chaos events of the hook — Close Door at 9 s, which shuts the next door on Rusty's path (here the study–bedroom door; the design sketch said 12 s, but by then Rusty is already inside d_{detect} of that door), the singular goal at 4 s, Reverse Hard at 2.5 s — and prints the budget line, the event timeline, and the report:

RUST **crates/capstone/examples/mission.rs**

```
use capstone::{format_mission, run_mission, ChaosEvent::*, MissionCfg, RateTable};

fn main() {
    let cfg = MissionCfg {
        seed: 7,
        rates: RateTable::DEFAULT,
        chaos: vec![CloseDoor { t: 9_000 }, SingularGoal { t: 4_000 }, ReverseHard { t: 2
   _500 }],
    };
    let report = run_mission(&cfg);
    for line in format_mission(&report) {
        println!("{line}");
    }
}
```

```
t_react 0.950 s  stop 0.750 m  margin 1.250 m  v_safe 1.264 m/s  l_valid 0.42 m
  2.20 s  reach  pick 1 done at (-1.50, -0.55)
  2.50 s  hitch  chaos: Reverse Hard -- full lock, full reverse
  4.00 s  reach  chaos: the next pick is pushed out to full stretch
  4.70 s  hitch  jackknife guard: |phi| = 0.91 rad > phi_warn = 0.9 while reversing
  5.00 s  reach  pick 2 done at (1.55, -0.55)
  5.55 s  reach  singularity guard: mu(q_goal) = 0.000 < mu_min = 0.25 (sigma_min =
   0.000)
  7.20 s  hitch  recovered: |phi| = 0.15 < phi_ok = 0.15; replanning from here
  7.25 s  hitch  trailer filter rejects hybrid A* (cell 0.5 m, reverse *1): |phi| reaches
    0.85 > phi_plan
  7.30 s  hitch  trailer filter rejects hybrid A* (cell 0.4 m, reverse *3): |phi| reaches
    0.85 > phi_plan
  7.35 s  hitch  hybrid A* (cell 0.5 m, reverse *1, via (12.4, 4.2)): 11.02 m, 2 cusps, 2
    expansions, max |phi| 0.69 rad
```

```
   7.90 s  reach  pick 3 done at (-0.50, 1.78)
   9.00 s  rusty  chaos: the study-bedroom door swings shut
  11.00 s  rusty  D* Lite repaired 3 times, 24-380 vertices each (until 12.00 s)
  18.35 s  hitch  parked: 0.000 m, 0.00?
  18.50 s  rusty  D* Lite repaired 7 times, 14-117 vertices each (until 24.50 s)
  24.90 s  rusty  D23.2: occupied cell 0.40 m ahead inside L_valid = 0.42 m -- braking
  25.00 s  rusty  D* Lite repaired 3 times, 14-32 vertices each (until 26.50 s)
rusty  Done: driven 14.54 m vs 11.02 m final-map optimum (ratio 1.32), 13 repairs
reach  Done: 3 picks; pick 1 cost 2.14 rad, t_f 0.97 s -> 0.93 s after CHOMP; max |tau|/
   tau_max 1.00 at the grid nodes
hitch  Done: parked within 0.0 cm / 0.0?, 3 cusps, max |phi| 0.91 rad
dense recheck: rusty true, reach true, hitch true
```

Every line of that output is produced by the browser port's `formatMission` and pinned, line for line, by the chapter's check file; the same file pins the no-chaos run's report (Rusty 10.76 m against a 10.99 m final-map optimum, ratio 0.98, 3 repairs; Hitch 1 cusp, max $|\phi|$ 0.77 rad) and asserts that two runs with the same seed produce byte-identical reports. The design sketch's numbers were targets — 18.4 m against 16.9 m, 1.84 s → 1.71 s, 4 cm and 1.8°; these are what the implementation actually does, and they differ for reasons worth reading.

- **Rusty's ratio is 1.32 with the door and 0.98 without.** The final-map optimum is computed on the world as it *ended*, door shut: an omniscient planner would never have gone into the study. Rusty did, because the door was open when it planned, and the 3.5 m of detour is the price of not being omniscient. Without chaos the ratio dips below one because Rusty's MPC rounds corners inside the inflation margin that the any-angle optimum respects.
- **Reach's torque ratio is exactly 1.00 at the grid nodes** — a time-optimal profile saturates a motor by construction. Sampled every 50 ms the replayed torques overshoot by a few percent, because the stage acceleration is held constant between nodes; Chapter 18's construction is exact only at them.
- **Hitch parks with zero error** because the last word of every hybrid A* path is an analytic Reeds–Shepp shot and the tracker replays it on the same exact-arc kinematics. The simulator grades its own homework here, and the retrospective says what evaluation looks like on hardware.

23.7 Putting it together: the failure tour

The Grand Plan lets you break the stack whenever you like; the Failure Tour breaks it the same way every time, records the result, and lets you scrub to the instant a statistic crosses its threshold. Each tab is one robot, one injection, one contract.

> INTERACTIVE · w23.3
>
> **Failure Tour**

> Recovery is not magic: each one is a named statistic crossing a threshold, followed by a mode with a plan.
>
> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch23-capstone#w23.3

Blocked Corridor. At 9 s the study–bedroom door shuts. Rusty is still in the study, heading for it. Nothing happens for almost two seconds — the door is outside the 2 m the stack trusts, and then a cell has to be hit by two consecutive scans — and at 11 s the map inflates a handful of cells across the doorway. The next planning tick hands those changes to D* Lite, which repairs **380 of the grid's 10,800 vertices** and returns a path that turns around, leaves the study by its corridor door and enters the bedroom from the corridor. The tracker turns in place first — a differential drive's privilege — and follows. Tick *throttle replanner* and run it again: the planner now ticks every two seconds, the validity horizon is 1.32 m, and you will see the D23.2 check brake Rusty in front of a door the planner has not yet heard about. The budget still closes ($0.6 \times 2.45 + 0.18 = 1.65$ m < 2.0 m); what the throttle costs is time, not safety, because the check is there.

Singular Arm. At 4 s the next pick is pushed out to full stretch. When Reach finishes its current pick and the guard evaluates the new goal, $\mu(q_{goal})$ dives to zero — the arm would be straight — and the mode machine reads `Plan` → `Recover(PullGoalInward { delta: 0.15 })` → `Replan` within one tick. The target moves 15 cm toward the base, μ comes back at 0.703, and the joint-space pipeline below it runs exactly as it did for every other pick. Switch the guard off in Stack Anatomy to see the other branch: the approach runs into the singularity, $\|\dot{q}\|$ passes 2 rad/s just before the tip reaches its target, and the task fails with `Failed(JointRate)` — the F3 bound arriving as a fault instead of as a threshold.

Jackknife. At 2.5 s Hitch is thrown into full reverse at full lock, steering the way that folds the trailer. The hitch angle grows for about two seconds — reversing is the unstable direction, and the steering makes it worse — and at 4.70 s crosses $\phi_{warn} = 0.90$ rad. The guard switches to `Recover(PullForward { until_phi: 0.15 })`; driving forward, the angle decays exactly as F4 predicts, and at 7.20 s Hitch is straight enough to replan. The first two rungs of the ladder propose the obvious Reeds–Shepp shot to the bay, and the trailer filter rejects both — simulated along them, the trailer would reach 0.85 rad, past ϕ_{plan}. The third rung plans to a staging pose east of the bays first and then to the bay: 11.02 m, 2 cusps, never above 0.69 rad. Hitch parks at 18.35 s.

The pattern is the same in all three tabs, and it is the chapter's second key idea.

> **RECOVERY IS A MECHANISM**
>
> Each recovery is a statistic the book already derived — scan returns the

blueprint cannot explain, the manipulability of Chapter 4, the hitch angle of Chapter 20 — crossing a threshold chosen from a bound you can state, followed by a mode whose parameters say what the recovery will do and when it is over.

23.7.1 The same contracts in production

None of this is peculiar to a textbook. Production motion stacks are chains of the same contracts — a state space, a validity checker, a planner that can say no, post-processors, time parameterization, a controller, recoveries — and the figure maps our stages onto three widely used open-source ones.

INTERACTIVE · f23.2

fig-23-library-map

Our stages are not toys with no relatives: each contract has a production counterpart, and a dash where a library deliberately has none.

Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch23-capstone#f23.2

Three columns, three philosophies. OMPL is a *planning library*: it owns state spaces, validity checking, planners and path simplification, and deliberately nothing else — no map, no controller, no clock. MoveIt is a *manipulation pipeline* around a planning scene: planner plugins (OMPL among them), then request adapters that add exactly the stages our Reach pipeline has — time parameterization and optimization-based smoothing — then execution. Nav2 is a *navigation system*: layered costmaps with an inflation layer, a planner server, a controller server, and a behavior tree that runs recoveries such as backing up or spinning when a statistic says the robot is stuck. The dashes are as informative as the names: a library that does not own a job will not quietly do it for you.

23.8 What Rust caught, and what it cost

The design brief for this book promised a retrospective with the costs reported as carefully as the wins. Here are both, in panels of the same size.

INTERACTIVE · w23.4

Retrospective Scorecard

Rust is not free safety: what the compiler caught and what it cost get the same amount of space.

> Run this simulation in the web edition: shannon-dynamics.github.io/PRM/chapters/ch23-capstone#w23.4

Read the left panel as five contracts the compiler enforced while the capstone was being written. The Manifold parameter keeps an arm's joint path away from a car's tracker; D23.1 is a type, so a curve without a clock cannot be executed; frames are marker types, so the map/odom mix-up that drifts a real robot silently never compiles; `Mode` is exhaustive, so adding the jackknife recovery broke every supervisor that had not thought about it; and the borrow checker refused to let the replanner mutate the grid the tracker was reading — the very race the D23.2 check exists to catch at run time, caught at compile time instead.

Read the right panel with the same attention. Generic code is monomorphized once per robot, and that is paid at every build; this edition did not measure the cost, so it prints no number, and a number you should trust would need clean and incremental build times of the workspace measured on stated hardware. WASM has no threads, so the browser runs fifteen stages cooperatively and demonstrates throughput, never preemption. Chapter 22 trains with `candle` but ships a forty-line forward pass so the learned sampler fits in the page. Chapter 11 hand-rolls its kd-tree because nearest neighbours on a torus need periodic axes. And the browser port is a second implementation kept honest by hand: its supervisor is total by a `switch` with an `assertNever`, which is a shadow of exhaustiveness rather than the thing itself.

Four further honesty items belong in the same ledger.

1. **The simulator grades its own homework.** Rusty's path ratio is measured against an optimum computed on the true final world, and Hitch's parking error against a goal pose in a simulator whose kinematics are exactly the planner's model. Both exist because we own the world. On hardware the same numbers need an external reference — motion capture, surveyed markers — and Hitch would not park with zero error, because no tire follows an arc exactly.
2. **"Complete" is conditional.** Rusty's resolution completeness holds for a localized belief and a world that changes slower than F2; Reach's probabilistic completeness says nothing about the iteration budget it actually runs; Hitch's composed guarantee is *none*, and the panels say so.
3. **Three stacks are not a team.** The panels share a clock and nothing else.
4. **The browser proves throughput, not scheduling.** The round-robin is deterministic by design, which is what makes the regression mission reproducible — and is exactly what a preemptive real-time system is not.

23.9 Where to go next

The book stops here; the subject does not. Four directions follow naturally from the last chapter.

- **Onto hardware.** The stage trait maps onto ROS 2 nodes almost one for one — a period, typed messages, a frame on every message. Rust bindings for ROS 2 exist (`rclrs` and `r2r` were the active ones in October 2026; check before you rely on either). What changes is everything this chapter could assume: time is not integer milliseconds, messages are lost, and the belief can be wrong in ways a simulator never invents.
- **Coupling the robots.** Put Rusty in the Lot with Hitch and the stacks stop being independent: Chapter 14's composite configuration spaces and prioritized planning are the starting point, and the composed-guarantee rule needs a new row for the coordinator.
- **Belief all the way down.** Rusty plans on a belief but tracks as if its pose were known. The sister book's capstone, *A Complete Autonomous Robot*, builds the stack from the estimation side — mapping, exploration and chance-constrained margins — and is the natural companion to this chapter.
- **Guarantees you can check at run time.** Every guard here is a hand-written monitor of one statistic. Runtime verification turns the contracts themselves — "never execute a Path", "every post-processor verifies" — into monitors generated from specifications.

23.10 Exercises

1 FOUNDATION •• **A walker in the corridor**

Extend F2 to an obstacle that approaches Rusty at speed v_{obs} from the moment it is first trusted at d_{detect}: during T_{react} both close the gap, and then Rusty brakes. Derive the inequality, then compute the fastest walker the default rates tolerate at $d_{detect} = 2.0$ m and $v_{max} = 0.6$ m/s.

WORK IT OUT

Fastest walker speed the default rates tolerate (m/s)

Answer. 1.316 m/s $v_{max}T_{react} + v_{max}^2/2a_{max} + v_{obs}T_{react} < d_{detect}$, so $v_{obs} < (2.0 - 0.750)/0.95 = 1.316$ m/s — a brisk walk. The budget function's `maxWalkerSpeed` returns the same number.

HINTS

The walker adds a third distance to the left side of F2, and it accrues only during the reaction time. After Rusty has stopped, a walker still approaching is the walker's problem; F2 is about Rusty's motion. Solve $0.750 + 0.95\,v_{obs} = 2.0$.

2 FOUNDATION • • • **Three chains, three guarantees**

Write all three pipelines as D23.3 chains — every stage, its role, its $\mathcal{G}_i$ — and prove each composed guarantee from F1's rule. Then decide which *single* stage you would upgrade first, and say what the upgrade costs. (For Rusty, the cost is measured in T_{react}; for Hitch it is measured in expansions per tick, because Hitch has no moving obstacles and therefore no F2.)

<details> <summary>A sketch of one answer</summary> Rusty: one rejecting stage, D* Lite, resolution-complete; the composed guarantee is that, *given* a localized belief and a world that changes slower than F2 allows. Reach: RRT-Connect, probabilistically complete, preserved because CHOMP is verified. Hitch: the trailer filter rejects without searching, so the composed guarantee is none. The obvious first upgrade is Hitch's: put the hitch angle into the search state (Chapter 21's CAR GRID SEARCH with a fourth axis and jackknife pruning), which makes the filter a verifier of a property the planner already guarantees and lifts the composed guarantee to resolution-complete — at roughly a hundred times the expansions, so the search must be sliced across many ticks. </details>

3 CONCEPTUAL •• **How slow may the replanner be?**

PREDICT FIRST

In the Failure Tour, tick 'throttle replanner' (0.5 Hz, Rusty at 0.6 m/s). Before you watch it: does the stack still satisfy F2?

- ✓ Yes — the stopping distance grows but stays under 2 m
  T_react = 0.10 + 0.10 + 0.20 + 2.00 + 0.05 = 2.45 s, and 0.6 × 2.45 + 0.18 = 1.65 m < 2.0 m. What grows past comfort is L_valid = 0.6 × 2.2 = 1.32 m: the D23.2 check, not the planner, ends up braking Rusty.
- No — at 0.5 Hz the planner is too slow and Rusty reaches the door
  Put the numbers in: the reaction distance is 1.47 m and braking adds 0.18 m, still inside the 2 m at which the door is trusted.
- It depends on the furniture the seed picked
  F2 is a worst case over the schedule; it does not depend on where the obstacle is, only on when it is first trusted.

Now compute the slowest replanner F2 allows if Rusty drove at 1.0 m/s, everything else at its default rate.

WORK IT OUT

Slowest replanner rate F2 allows at v_max = 1.0 m/s (Hz)

Answer. 0.952 Hz Braking takes 0.5 m, leaving 1.5 m, so $T_{react} \leq 1.5$ s. The other four terms are 0.45 s, so $\tau_{plan} \leq 1.05$ s, i.e. $f_{plan} \geq 0.952$ Hz. The planner's period is the dominant term — which is why the throttle is the knob the Failure Tour gives you.

4 CONCEPTUAL • **Switch the filter off**

PREDICT FIRST

In Stack Anatomy, switch off Hitch's trailer filter. What happens to the $\mathcal{G}$ printed under Hitch's row?

- It drops further, from none to something worse
  There is nothing below none in the order; and removing a rejecting stage can only remove a term from the minimum.
- ✓ It rises to resolution-complete
  The filter was the rejecting stage with $\mathcal{G}$ = none. Without it the minimum is hybrid A*'s, resolution-complete for the car — completeness bought by giving up every claim about the trailer.
- It stays at none
  D23.3 is a minimum over rejecting stages; the remaining one is hybrid A*, which is resolution-complete w.r.t. its primitives.

Completeness and soundness are different promises: an unfiltered planner answers more queries, and some of its answers jackknife.

5 PRACTICAL •• **Add Mode::ReturnHome**

After `Done`, Rusty should plan back to q_{start} and drive there. Add the variant and its transitions, touching only `supervisor.rs` and `mission.rs`. Let the compiler find every `match` that now fails with E0004 — the fourth vignette of the Retrospective Scorecard, firsthand — and decide for each whether `ReturnHome` behaves like `Execute`, like `Plan`, or like neither.

HINTS

Start with `supervisor_step`: which events should `ReturnHome` accept, and which mode does `Arrived` lead to now? The planner stage needs a goal that depends on the mode; that is a change to the stage's input type, not to D *Lite.* *D* Lite's goal is fixed at construction; returning home is a new search with start and goal swapped, not a repair.

6 PRACTICAL ••• **Swap Hitch's planner behind the same contract**

Replace hybrid A* plus the trailer filter with Chapter 21's CAR GRID SEARCH with the hitch angle in the state and jackknife pruning at ϕ_{plan}, behind the same `Stage` trait and the same jackknife guard; slice it so that one tick does at most a few thousand

pops. Re-run seed 7 and the *Jackknife* tour. Does `composed_guarantee` change? What happened to the parking error, and why?

> **HINTS**
>
> The new planner is a rejecting stage whose guarantee covers the trailer; the filter becomes a verifier of a property the planner already guarantees. CAR GRID SEARCH stops in a goal *region*; finish with a Reeds–Shepp shot only if the trailer filter accepts it. Resolution-complete for the trailer, and a parking error of the order of the goal region's tolerance instead of zero.

23.11 References

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E. and Thrun, S. (2005). Principles of Robot Motion: Theory, Algorithms, and Implementations. *MIT Press*.

Chapter 1's taxonomy, the offline/online distinction of the epigraph, and the path–trajectory distinction this chapter turns into types; the book has no systems chapter, which is why this one exists. `link`

Şucan, I. A., Moll, M. and Kavraki, L. E. (2012). The Open Motion Planning Library. *IEEE Robotics & Automation Magazine 19(4)*.

The planner / state-space / validity-checker separation that the Stage contract mirrors, and the OMPL column of the library map. `doi:10.1109/MRA.2012.2205651`

Koenig, S. and Likhachev, M. (2005). Fast Replanning for Navigation in Unknown Terrain. *IEEE Transactions on Robotics 21(3)*.

D* Lite, the repair Rusty runs twice a second; Chapter 6 derives it. `doi:10.1109/TRO.2004.838026`

Kuffner, J. J. and LaValle, S. M. (2000). RRT-Connect: An Efficient Approach to Single-Query Path Planning. *IEEE International Conference on Robotics and Automation*.

Reach's first feasible path, and the source of its pipeline's probabilistic completeness. `doi:10.1109/ROBOT.2000.844730`

Karaman, S. and Frazzoli, E. (2011). Sampling-based Algorithms for Optimal Motion Planning. *International Journal of Robotics Research 30(7)*.

RRT*, the anytime improver inside Reach's planning budget. `doi:10.1177/0278364911406761`

Gammell, J. D., Srinivasa, S. S. and Barfoot, T. D. (2014). Informed RRT*: Optimal Sampling-based Path Planning Focused via Direct Sampling of an Admissible Ellipsoidal Heuristic. *IEEE/RSJ International Conference on Intelligent Robots and Systems*.
The informed sampler Chapter 13 implements and Reach runs. doi:10.1109/IROS.2014.6942976

Pham, H. and Pham, Q.-C. (2018). A New Approach to Time-Optimal Path Parameterization Based on Reachability Analysis. *IEEE Transactions on Robotics 34(3)*.
TOPP-RA, the modern counterpart of the Time-Scaling Algorithm Chapter 18 uses for Reach, and the kind of stage MoveIt's time-parameterization adapters run. doi:10.1109/TRO.2018.2819195

Zucker, M., Ratliff, N., Dragan, A. D., Pivtoraiko, M., Klingensmith, M., Dellin, C. M., Bagnell, J. A. and Srinivasa, S. S. (2013). CHOMP: Covariant Hamiltonian Optimization for Motion Planning. *International Journal of Robotics Research 32(9–10)*.
The smoother that F1 shows must be verified before it can be trusted with a guarantee. doi:10.1177/0278364913488805

Dolgov, D., Thrun, S., Montemerlo, M. and Diebel, J. (2010). Path Planning for Autonomous Vehicles in Unknown Semi-structured Environments. *International Journal of Robotics Research 29(5)*.
Hybrid A *with analytic Reeds–Shepp expansions — Hitch's planner, and the ancestor of Nav2's Smac Hybrid-A*. doi:10.1177/0278364909359210

LaValle, S. M. and Kuffner, J. J. (2001). Randomized Kinodynamic Planning. *International Journal of Robotics Research 20(5)*.
Forward propagation on a control system — the lineage of Hitch's planners (Choset §7.5.1) and of Chapter 21's kinodynamic trees. doi:10.1177/02783640122067453

Nav2 contributors (2026). Nav2 Documentation.
The navigation-system column of the library map: costmap layers, planner and controller servers, behavior-tree recoveries. Accessed October 2026; component names change between releases. link

MoveIt contributors (2026). MoveIt Motion Planning Framework.
The manipulation column of the library map: planning scene, planner plugins, time parameterization and smoothing adapters. Accessed October 2026. link