A Handbook for Robotics

Irakli Bakuradze
Department of Mathematics, WPI

William Sobral
Department of Robotics, WPI

December 18, 2025

Abstract

This deliverable presents a beginner friendly robotics handbook for beginner to intermediate students focused on turning ideas into working programs. The book covers common robotics topics in a concrete, digestible way, emphasizing clear reasoning and practical implementation. In several chapters, the material is developed through a problem-first workflow: we start from a simple scenario, make an honest attempt, identify what breaks, and refine the approach into something more reliable. Throughout, examples are chosen to be easy to visualize so readers can focus on building transferable problem-solving habits. As an additional resource, we provide a lightweight simulation environment and example scenarios for hands-on experimentation and iteration before moving to hardware.

Acknowledgements

We thank Professor Vincent Aloi for his guidance, feedback, and encouragement throughout this robotics handbook project. We also thank Lorenzo Hess for technical support and helpful insight into the book's formatting.

Contents

1 Introduction

This handbook is written for beginner-to-intermediate robotics students who want to get better at turning ideas into working programs. Robotics is inherently interdisciplinary: it combines geometry, physics, algorithms, and software engineering, and it often requires coordinating many moving pieces at once. For that reason, a key skill is abstraction: deciding what to model, what to ignore, and what level of detail is appropriate for the decision you are trying to make.

The goal of this book is to present common, useful topics in a way that is concrete, digestible, and practical, while staying intellectually honest and relatable about how solutions are actually discovered. Many of the problems we use are intentionally simple to state and easy to visualize; the point is not the particular scenario, but the underlying principles and habits of reasoning that transfer to more complex systems.

Note: AI aided in the creation of this book and the simulations.

1.1 What this book is (and is not)

This is not meant to be an exhaustive robotics textbook, and it is not a reference manual. Instead, it is a guided collection of problems, techniques, and implementation patterns that we have found repeatedly useful when building simple robotic behaviors. Whenever possible, we emphasize the underlying reasoning that connects a problem statement to a working approach, rather than presenting final answers in a polished "from thin air" form.

1.2 A problem-first style and "honest attempts"

Many chapters intentionally begin from a problem and work forward through a natural solution process: proposing an approach, testing it against edge cases, discovering what fails, and refining the idea until it becomes a reliable method. We call these "honest attempts"—not because they are perfect, but because they reflect the real cognitive workflow of engineering: making assumptions explicit, debugging your own reasoning, and improving designs iteratively.

The hope is that, by watching the reasoning process (including the missteps), you gain reusable intuition: how to structure a robotics problem, how to choose what to model and what to ignore, and how to build solutions that are robust rather than merely correct on a single example.

1.3 How to use this book

To get the most out of the chapters, try to actively follow the logic rather than only reading the final result. In practice, that means: (i) pausing to predict the next step before reading it, (ii) translating key ideas into your own words, and (iii) implementing small variations of the same idea (different parameters, different geometry, different noise, different constraints) to see what breaks.

If a derivation or implementation detail feels unfamiliar, treat it as an opportunity to build a small personal "toolbox": a few trusted patterns for coordinate handling, state updates, debugging, and testing. Those skills transfer across essentially every robotics project.

1.4 A simple simulation environment as a bridge

Alongside the text, we provide a small simulation environment for running basic robotics scenarios. It is intentionally simple and beginner-friendly: we simulate in 2D (top-down) because that choice removes a large amount of complexity that often distracts from the core learning goals. Many high-fidelity simulators exist, but they are frequently heavier than what is needed when your objective is to practice reasoning, control logic, and implementation discipline.

We view the simulator as an intermediate bridge between:

- **theory and code** (ideas you can write down and implement), and
- **reality** (hardware quirks, calibration issues, unreliable sensors, and integration overhead).

Real robots introduce many additional failure modes that are important to learn, but they can also make it hard to isolate and train the core skill of algorithmic problem solving. The simulator provides a controlled place to iterate quickly and build confidence before dealing with the full complexity of hardware.

The simulation environment is available at:

https://github.com/homeless-vibecoder/simulation_environment_for_basic_robotics_scenarios

The repository includes setup instructions and example scenarios (e.g., a line-following robot) that you can run immediately and then modify.

1.5 What we hope you take away

By the end of this book, our hope is that you will be able to approach unfamiliar robotics problems more calmly and systematically: identify the essential variables, select a workable model, write and test control logic, and iterate from failures toward robust behavior. If the book succeeds, you will not only know a handful of techniques—you will also have a clearer sense of how to think when you do not yet know the answer.

2 Control and PID

2.1 The basic problem: controlling what you cannot directly set

In many robotics problems, the quantity we care about is not something we can directly command. For example, we want a robot to be at a particular position, but we can only command its motor voltage. We want a wheel to spin at a specific angular velocity, but we only control motor power / current. We want a drone to hold a fixed altitude, but we can only change thrust.

So the general situation is that there is some variable we care about (call it x, such as position, angle, or speed) and something we can directly command (call it u, such as motor power or torque). The physics of the robot gives us a relationship: "When I change u, x slowly responds over time." We will call x the state (or controlled variable), and u the control input.

The job of a controller is to look at how x is doing (or how we estimate it), compare it to the goal x_{goal}, and decide what u should be next. You can think of a controller as a piece of code that continuously answers:

> Given what I want (x_{goal}) and what I see (my estimate of x), what motor command u should I send right now?

Later we will introduce PID, but we will get there by starting from very simple controllers and seeing their strengths and weaknesses.

2.2 A concrete example: controlling motor speed

To make the discussion specific, we will mostly use motor speed control as the running example. Let x be the angular velocity of the motor (our target is x_{goal}) and u be the command we send to the motor driver. For simplicity, let u be a number in the range $[-1, 1]$, where -1 is full reverse, 1 is full forward, and 0 is "no power".

In reality, motor power roughly controls torque, which affects acceleration, and friction effects grow with speed, so the relationship between u and x is nonlinear. For this chapter we will keep the mental model simple: if we increase u, the motor tends to speed up; if we decrease u, it tends to slow down; and there is a limit on how fast the speed can change. This is enough to reason about basic controllers and PID.

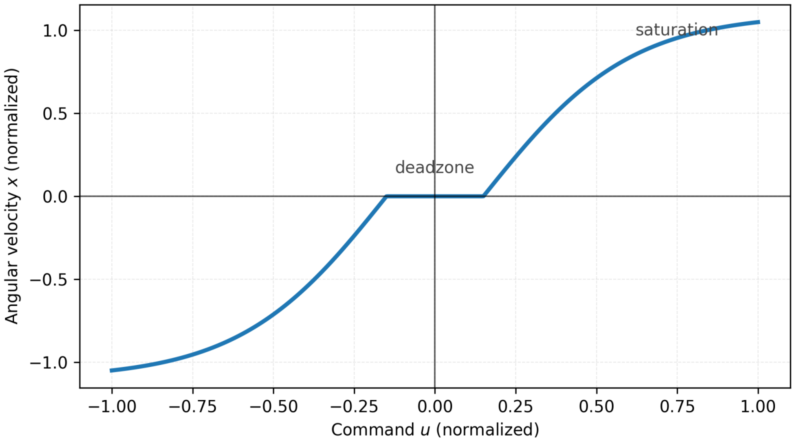

Figure 1: Rough sketch of angular velocity x vs command u. At small u the motor barely moves; as u increases, x rises, but eventually saturates because friction and back-EMF balance the torque.

2.3 Attempt 1: bang-bang control

We start with the simplest possible controller.

Suppose:

- We can measure the current speed x.
- We have a desired speed x_{goal}.

A bang-bang controller uses only the sign of the error:

$$e = x_{\text{goal}} - x.$$

The rule is:

- If $e > 0$ (we are too slow), command maximum forward power.
- If $e < 0$ (we are too fast), command maximum reverse power.
- If $e = 0$, command zero.

In code:

```
def bang_bang_control(x, x_goal):
    error = x_goal - x

    if error > 0:
        u = +1.0 # full forward power
    elif error < 0:
        u = -1.0 # full reverse power
    else:
        u = 0.0

    return u
```

This controller is very simple and aggressive, as it always uses maximum effort in whichever direction it thinks is correct.

If we run this in a loop, when the motor is slower than the target, we drive it forward at full power. It speeds up and eventually overshoots the target. Once it is above the target, we slam power in the other direction. The motor slows down, crosses below the target, and we flip again. The result is a persistent oscillation of x around x_{goal}.

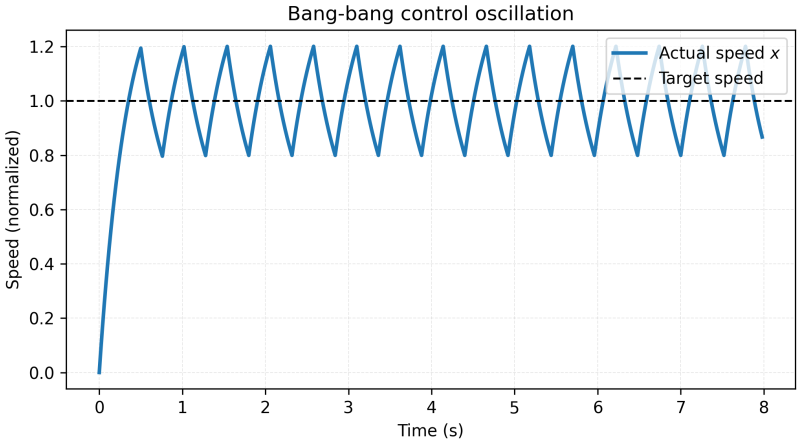

Figure 2: Bang-bang speed control causing oscillations around the target speed. The target speed is a horizontal line, and the actual speed oscillates above and below it as the bang-bang controller alternates between full positive and full negative effort.

2.4 Where the oscillation comes from

When students first see this behavior, a common explanation is:

> We overshoot because of momentum; we cannot stop instantly.

That is partly true for systems where we control a *second derivative* of the thing we care about (for example: controlling position, where your command affects acceleration). In those cases, momentum/inertia makes it hard to stop exactly at a target.

But in this motor-speed example, the controlled variable is *speed* x, and our command u affects something close to the *first derivative* x':

- Increasing u increases torque, which increases angular acceleration.
- In a simplified model we can think of u as almost equal to x' (the rate of change of speed).

So in principle, if we could react instantly, we could set u to 0 and stop changing the speed (stop accelerating) immediately. In other words: this oscillation is not mainly a "momentum" story.

Why is there still oscillation? Two key reasons: delay / sample time (we only update u at discrete times, so the motor keeps responding to the old command between updates) and all-or-nothing effort (even if we are very close to the target, bang-bang still uses maximum effort).

Consider this simple discrete-time picture. At time t_k we see that x is a tiny bit below x_{goal}, so we set $u = +1$ (full power). For the entire next sample period of length D, the motor keeps accelerating. By the time we measure again at t_{k+1}, the speed has increased by about $x'_{\max}D$. If the remaining error was smaller than that, we will overshoot before we even get a chance to change u. Then we flip to $u = -1$, overshoot on the other side, and repeat. This creates a small "limit cycle" around the target even without needing a momentum-based explanation.

Takeaway: With bang-bang control, even tiny errors near the target produce maximum effort, and with any delay at all this leads to oscillations.

2.5 Attempt 2: proportional control (P)

To reduce oscillations, we would like:

- Big corrections when the error is large.
- Small corrections when the error is small.

A natural idea is:

> Make the control effort proportional to the error.

Instead of $u = \pm 1$, we set:

$$u = K_p \cdot e = K_p \cdot (x_{\text{goal}} - x).$$

Here, K_p is a proportional gain (a positive constant). When $|e|$ is large, $|u|$ is large; when $|e|$ is small, $|u|$ is small.

Because our hardware has limits, we must clip u to the allowed range:

```python
def proportional_control(x, x_goal, K_p, u_min=-1.0, u_max=1.0):
    error = x_goal - x
    u = K_p * error

    # Clip to the allowed range
    if u > u_max:
        u = u_max
    elif u < u_min:
        u = u_min

    return u
```

This controller behaves like bang-bang when the error is very large (if $K_p|e|$ is bigger than 1, the command saturates to ± 1). But near the target, the error is small, so $|u|$ is small. That means gentler accelerations and much smaller overshoot.

Takeaway: Proportional control encodes a simple and powerful idea: "The bigger the error, the harder I push; the smaller the error, the more gently I approach."

2.6 Many possible feedback laws

When we switched from bang-bang to proportional control, it might have felt like there was one "correct" formula:

$$u = K_p(x_{\text{goal}} - x).$$

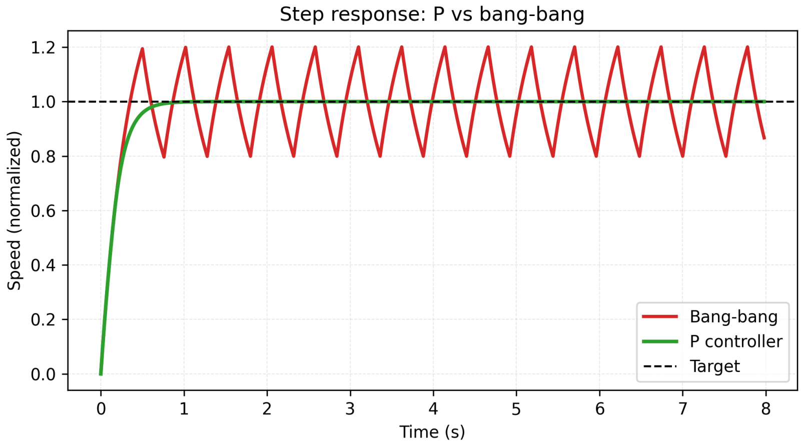

Figure 3: Step response of a P controller vs a bang-bang controller. The P controller approaches the target smoothly and overshoots less.

But this is just one choice among many possible feedback laws of the general form:

$$u = f(e), \quad e = x_{\text{goal}} - x,$$

where f is any function that:

- Pushes in the right direction (sign of u matches sign of e).
- Grows larger when $|e|$ is larger.

For example, we could choose:

$$u = k \, \text{sign}(e) \, |e|^p,$$

where $\text{sign}(e)$ is $+1$ if $e > 0$, -1 if $e < 0$ (and 0 if $e = 0$).

with parameters:

- $k > 0$ (overall gain / scaling).
- $p > 0$ (nonlinearity).

With clipping to $[-1, 1]$, this gives: for large error, $|e|^p$ is big and u saturates to ± 1 (like bang-bang). For small error, if $p > 1$, $|e|^p$ shrinks faster than $|e|$, so corrections are very gentle. If $0 < p < 1$, $|e|^p$ shrinks slower than $|e|$, so corrections stay relatively strong near the target.

This shows an important point: there is usually not a single “mathematically correct” feedback law. Instead, there is a family of reasonable choices. We typically pick simple, tunable functions (like $K_p e$) that work well enough and are easy to understand.

Takeaway: In control design, do not assume there is a unique “magic formula”. Often many different feedback laws work, and we choose ones that are simple and tunable.

2.7 From P to PI and PID

Proportional control is a good start, but it has limitations.

The problem of steady-state error

In the real motor:

- Friction and back-EMF mean that:
- If we hold u constant, the speed x will settle to some value where torque = opposing forces.

With a pure P controller, it is common to see:

- The motor speed settles close to x_{goal}, but not exactly.
- There is a small steady-state error: $e_{\text{steady}} \neq 0$.

Intuitively:

- When x is slightly below x_{goal}, the error is small.
- So $u = K_p e$ is also small (K_p is just a constant to be tuned).
- That small u is just enough to balance friction, but not enough to push all the way to the true target.

We would like a controller that remembers this persistent error and keeps increasing the command until the steady-state error is driven to (almost) zero.

Integral action (I)

Integral control adds a term that looks at the accumulated past error:

$$I(t) = \int_0^t e(\tau)\, d\tau.$$

Then we add a term $K_i I(t)$ to the control input:

$$u(t) = K_p e(t) + K_i I(t).$$

Where K_p, K_i are constants (to be tuned).

In discrete time, running at time steps of length Δt, we can approximate:

```
I = I + error * dt # accumulate error over time
u = K_p * error + K_i * I
```

Intuitively, if the error stays positive for a long time, the integral $I(t)$ grows larger and larger, and so does the contribution $K_i I(t)$. Over time, this accumulated push lets the controller overcome constant biases (friction, slopes, miscalibration, etc.) and drive the steady-state error closer to zero.

However, if we let I grow without limits, we may get integral windup: when the actuator saturates (e.g. u clipped at 1), the error can stay large, causing the integral term to continue growing. Once we finally leave saturation, u can be much too large, causing big overshoot. We typically handle this with some form of integral limiting.

Derivative action (D)

So far we used:

- P: reacts to the current error.
- I: reacts to the accumulated past error.

Derivative action reacts to how fast the error is changing:

$$D(t) \approx \frac{de}{dt}.$$

We add a term:

$$u(t) = K_p e(t) + K_i I(t) + K_d D(t).$$

In discrete time:

```
D = (error - prev_error) / dt
u = K_p * error + K_i * I + K_d * D
prev_error = error
```

Intuition: if the error is decreasing rapidly, D is negative. The derivative term subtracts from u when the system is moving quickly toward the target. This acts like damping: it tells the controller to slow down corrections when we are already moving in the right direction. Derivative action is especially useful in systems with second-order behavior, where momentum and overshoot are big problems (for example, position control where you command acceleration).

The PID controller

Putting it all together:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de}{dt}.$$

This is called a PID controller: P is proportional to current error, I is proportional to accumulated error, and D is proportional to the rate of change of error. Many practical controllers are P-only or PI for relatively simple, first-order systems (like speed control), and PID (or carefully tuned PD) for systems where overshoot and oscillations are a big concern.

Takeaway: PID adds memory (I) and prediction (D) to simple proportional control, giving us a flexible and powerful family of controllers.

2.8 Implementing a discrete-time PI/PID for motor speed

We now put the pieces together into a simple discrete-time controller for motor speed.

Assume:

- We run a loop every Δt seconds.

- Each iteration we can:

- Read the current speed x from an encoder.

- Compute the error $e = x_{\text{goal}} - x$.

- Update P, I, D terms.

- Compute a command u, then clip it to $[-1, 1]$.

Basic structure with PI

```
def speed_controller_step(x, x_goal, state, dt):
    # state holds integral and previous error between calls
    error = x_goal - x

    # Proportional term
    P = state.K_p * error

    # Integral term with simple anti-windup clamp
    state.I += error * dt
    state.I = max(min(state.I, state.I_max), -state.I_max)
    I_term = state.K_i * state.I

    # (Optional) Derivative term -see below
    D = 0.0
    if state.use_derivative:
        D = (error - state.prev_error) / dt
    D_term = state.K_d * D

    # Combine terms
    u = P + I_term + D_term

    # Clip to allowed range
    u = max(min(u, 1.0), -1.0)

    # Remember for next step
    state.prev_error = error

    return u
```

Key practical points:

- We limit the integral term to prevent windup.
- For speed control, many systems work well with PI only ($K_d = 0$).
- If derivative noise is a problem (it usually is), we can:
- Apply a small low-pass filter to D.
- Or approximate derivative of a filtered version of the speed.

Relation to time-scales and order of control

For motor speed:

- Our command u is close to acceleration (through torque).
- The speed x behaves roughly like a first-order system.

That means:

- Proportional + integral action typically give:
- Good tracking with small steady-state error.
- Acceptable overshoot if gains are chosen moderately.
- Derivative action is less critical than in position control, where we command acceleration and get second-order position dynamics.

This connects back to the general idea of order of control: if we directly control a velocity-like quantity, PI is often enough. If we control acceleration and care about position, PID (or at least PD) becomes more important to manage momentum and overshoot.

2.9 Practical tuning and pitfalls

Even with a correct structure, the numbers (K_p, K_i, K_d) matter a lot.

A simple manual tuning procedure (for PI)

One common approach for motor speed:

1. Start with P only:
   - Set $K_i = 0$, $K_d = 0$.
   - Increase K_p from 0 until:
   - The system responds fast enough to step changes in x_{goal}.
   - But does not oscillate wildly.
2. Add a bit of I:
   - Gradually increase K_i from 0.
   - Watch how the controller eliminates steady-state error.
   - Stop before it creates large overshoot or slow oscillations.
3. (Optional) Add D if needed:
   - If the response is still too oscillatory, or overshoots too much, a small K_d can help.
   - Be careful: D is sensitive to measurement noise.

Throughout tuning:

- Keep an eye on actuator saturation (is u often stuck at ± 1?).
- Adjust the integral limit to prevent windup.

Anti-windup strategies

Integral windup is especially problematic when:

- The controller asks for more effort than the actuator can produce.
- The output u is being clipped for an extended period.

Common strategies:

- Clamp the integral state (as in the example).
- Freeze or slow down integral accumulation when:

- The output is saturated.
- And the error is still pushing integral in the same direction.
- Use more advanced schemes that back-calculate what the integral *should* be given the saturated output.

Noise and derivative filtering

Real speed measurements have noise:

- Encoder quantization.
- Mechanical vibrations.

If we compute a raw derivative:

```
D = (error - prev_error) / dt
```

small measurement noise can cause large spikes in D when Δt is small.

Simple mitigation:

- Use an exponential moving average (EMA) on the measured speed or on D:

```
alpha = 0.2
smoothed_D = (1 - alpha) * state.smoothed_D + alpha * D
state.smoothed_D = smoothed_D
```

- Use the smoothed value in the D term.

Takeaway: Good controllers are not just about formulas; they also require careful thinking about actuator limits, noise, sampling rates, and how to tune and protect the integral term.

2.10 A brief optimal-control perspective (optional)

So far we have focused on heuristic feedback laws that work well in practice.

There is also a more mathematical view:

- We define a cost function that measures how "bad" a control strategy is.

- We then ask: “Among all possible control inputs $u(t)$, which one makes the cost as small as possible?”

For example, suppose we want to:

- Start at speed $x(0) = 0$.
- Reach and hold a target speed x_{goal}.
- Penalize large deviations from the target over time.

One simple cost is the squared error over time:

$$J = \int_0^T \big(x(t) - x_{\text{goal}}\big)^2 dt.$$

We might also penalize very large control inputs (to avoid aggressive behavior):

$$J = \int_0^T \Big[q\big(x(t) - x_{\text{goal}}\big)^2 + r\,u(t)^2\Big]\,dt,$$

with weights $q, r > 0$.

Solving for the exact optimal $u(t)$ requires:

- A reasonably accurate dynamic model of the system.
- Some calculus of variations / optimal control math.

In many classical cases, the solution turns out to be a linear feedback law, which looks suspiciously similar to PID or state-feedback controllers.

We will not go deeper here, but it is useful to know:

- PID can be viewed as a practical, robust approximation to more formal optimal controllers.
- The cost-function viewpoint gives a principled way to talk about trade-offs:
- Fast response vs. overshoot.
- Small error vs. small control effort.

2.11 Cascading control

Many robots have several variables that must be controlled at once, and they respond on *different time-scales*.

Examples:

- A motor has fast current / torque dynamics and slower speed dynamics.
- A drone must keep its attitude stable very quickly, while its altitude and position change more slowly.
- An elevator needs smooth position profiles, which depend on controlling velocity and acceleration.

Trying to make one single PID controller handle all of these effects at once is often hard to tune, sensitive to disturbances, and confusing to reason about.

Cascading control (cascade control) solves this by stacking loops: an outer (slow) loop computes a setpoint for an inner (faster) loop, which may compute a setpoint for an even faster loop, and so on. Each inner loop is closer to the actuator and is tuned to respond much faster than the loop above it, so the outer loop can treat the inner loop as an almost-ideal "actuator" for its time-scale.

At each stage, the inner loop tries to track its setpoint much faster than the loop above it changes that setpoint. The outer loop can then pretend the inner loop is an almost-ideal actuator.

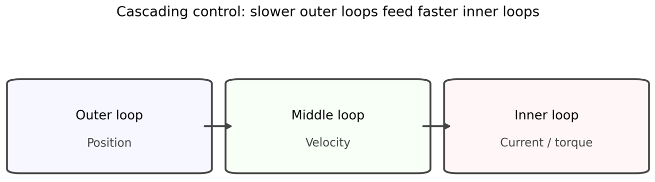

Figure 4: Conceptual cascading control: an outer loop (position) feeds a setpoint to a middle loop (velocity), which feeds a setpoint to an inner loop (current/torque). Each loop has its own sensor and controller.

Basic structure of a cascade

A simple feedback loop contains:

- A measured variable (from a sensor).
- A controller (for example, PID).
- A final control element (the plant / actuator).

In a cascading controller we stack several such loops: each loop has its own measured variable and controller, the output of an outer loop becomes the setpoint of the inner loop, and the innermost loop directly commands the actuators.

This lets us give each loop a clear job on its own time-scale, use simple controllers in each loop instead of one giant hard-to-tune controller, and improve disturbance rejection (since inner loops can react quickly to disturbances before they grow into large outer-loop errors).

Takeaway: Cascading control decomposes a complex control problem into several easier problems, each handled by its own loop and time-scale.

Example: motor current, speed, and position

Consider an electric motor in an elevator lift.

We ultimately care about:

- Position of the elevator car (floor height).
- Comfort: no sharp jerks or sudden accelerations.

Physically, the motor obeys:

- Commanded voltage / current → changes in torque.
- Torque → changes in angular acceleration.
- Acceleration → changes in speed and position.

If we try to control position with a single PID that directly outputs motor current:

- Large position errors produce large current commands.
- The motor can accelerate very quickly at the start of a move.
- The resulting speed profile often has large peaks and high jerk (uncomfortable for passengers).

Instead we can build a cascade:

1. Trajectory generator (outermost):
    - Takes start and target positions.
    - Produces a *smooth* desired profile of
    - position $x_{\text{des}}(t)$, velocity $v_{\text{des}}(t)$, and maybe acceleration $a_{\text{des}}(t)$.
    - For comfort, we can use trapezoidal or S-curve profiles that limit acceleration and jerk.
2. Position loop:
    - Measures actual position x.
    - Compares it to x_{des} and outputs a desired velocity v_{set}.
    - This loop is relatively slow.
3. Velocity loop:
    - Measures actual velocity v (for example from an encoder).
    - Compares it to v_{set} and outputs a desired torque / current command i_{set}.
    - This loop is faster and tries to track v_{set} tightly.
4. Current (torque) loop (innermost):
    - Controls motor current directly using current sensors.
    - Runs at the highest frequency and deals with fast electrical dynamics.

Each loop only has to solve a *local* problem: the position loop does not worry about detailed motor physics, just asking for a reasonable velocity; the velocity loop does not worry about jerk-limited moves, just tracking v_{set}; and the current loop only worries about the motor's electrical behavior.

Example: quadcopter attitude and altitude

A quadcopter flying at a fixed altitude needs to:

- Keep its attitude (roll, pitch, yaw) near level to avoid tipping.
- Control its altitude (height) above the ground.

- Often also track a desired horizontal trajectory.

The typical cascade looks like:

1. Outer position / altitude loop:
   - Uses GPS, barometer, or range sensors to estimate position and height.
   - Outputs desired attitude and total thrust that would move the drone toward its target.
2. Inner attitude loop:
   - Uses IMU (gyro + accelerometer) measurements.
   - Controls roll, pitch, and yaw rates very quickly.
   - Outputs motor commands (or thrust setpoints) to achieve the desired attitude.

The key idea is the same:

- The attitude loop runs very fast (hundreds of Hz) and stabilizes the drone.
- The outer loop runs slower and assumes that when it asks for a certain attitude, the inner loop will achieve it quickly.

Takeaway: In multirotor control, a fast inner attitude loop gives the outer altitude/position loop something that behaves almost like an ideal “tilt and thrust” actuator.

Design guidelines for cascades

When designing cascaded loops, it helps to:

- Choose variables in order of proximity to the actuator:
  - Innermost: quantities directly affected by actuator commands (current, torque).
  - Middle: their integrals (velocity).
  - Outermost: slowest variables (position, trajectory).

  Separate time-scales:

  - Inner loops should be significantly faster than outer loops.

  - Outer loops should not change their setpoints faster than inner loops can track.

  Tune from inside out:

  - First tune the innermost loop until it behaves well.
  - Then treat it as part of the plant for the next loop, and tune that loop, and so on.

Takeaway: A good cascade respects both physics and time-scales: each loop controls the most appropriate variable on its own time-scale, and we tune loops from the inside out.

2.12 Summary and takeaways

In this chapter, we looked at control and PID through the lens of a simple motor-speed example.

- Start simple: Bang-bang control is easy to write, but it naturally produces oscillations because it always uses maximum effort, even very close to the target.
- Add memory and prediction: The integral term learns and cancels biases (friction, gravity, miscalibration), while the derivative term damps motion by reacting to how quickly the error is changing.
- Respect hardware and noise: Saturation, integral windup, sensor noise, and discrete-time sampling all affect how a controller behaves in the real world.
- Think in terms of time-scales and order of control: Whether we directly control speed, acceleration, or something else changes which parts of PID are most important and how we tune them.

As you build more complex controllers (for line following, arm motion, balancing robots, and more), you will see the same themes repeat: use feedback that depends on the difference between desired and actual state, choose simple, tunable laws (like PI/PID) that respect the system's physics and constraints, and refine them with estimation, filtering, and careful tuning to get robust, smooth behavior.

3 Line follower robot

3.1 The problem: staying on the line

Imagine a small two-wheeled robot driving on a white floor with a black line painted on it. It rides on two drive wheels on the same axle, and near each wheel sits a simple line sensor that only reports “line” (1) or “no line” (0).

The goal is simple to state: keep the robot roughly centered over the line, keep making forward progress, and avoid losing the line completely.

This makes a great example because it is easy to visualize (you can literally draw it on paper) and the robot has very limited sensing (just a couple of bits). Making it work well still forces us to think about the geometry of differential-drive motion, what a naive controller does and why it struggles, the relevant time-scales, the order of control (position vs velocity vs acceleration), and how to use state and estimation to do better.

We will first build a very simple controller that more or less “works”, then look carefully at where it breaks, and finally move to a slightly smarter controller that uses more information.

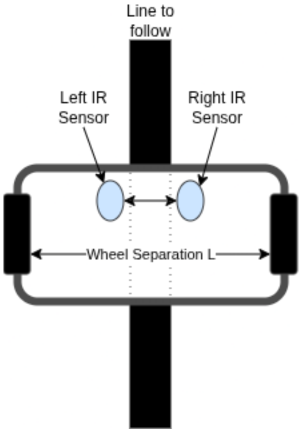

Figure 5: Top-down view of the line follower robot with sensors, line, and wheelbase labeled.

3.2 How a differential-drive robot moves

Before we talk about control, we need a simple mental model for how the robot moves when we command different wheel speeds.

Let:

- v_l = linear velocity of the left wheel (m/s).
- v_r = linear velocity of the right wheel (m/s).
- L = distance between the wheels (wheel separation).

We can break the motion into:

- Forward speed v of the robot's center.
- Angular speed w (how fast it rotates around its center, positive for counterclockwise).

The standard differential-drive kinematics say:

$$v = \frac{v_r + v_l}{2}, \qquad w = \frac{v_r - v_l}{L}.$$

Some sanity checks:

- If $v_l = v_r$, then $w = 0$ and the robot goes straight.
- If $v_l = -v_r$, then $v = 0$ and the robot spins in place.

We can also go the other way:

Given a desired v and w, we can compute:

$$v_l = v - \frac{L}{2}w, \qquad v_r = v + \frac{L}{2}w.$$

This is how we will usually think about it in code:

- Decide on a forward speed v.
- Decide how much we want to turn (angular speed) w.
- Convert (v, w) into (v_l, v_r) and send those to the motors.

3.3 Attempt 1: bang-bang control with two binary sensors

We now return to the line follower.

The robot uses two binary line sensors, one near each wheel. Each sensor reports 1 when it sees the black line and 0 when it only sees the white floor. So at each time step

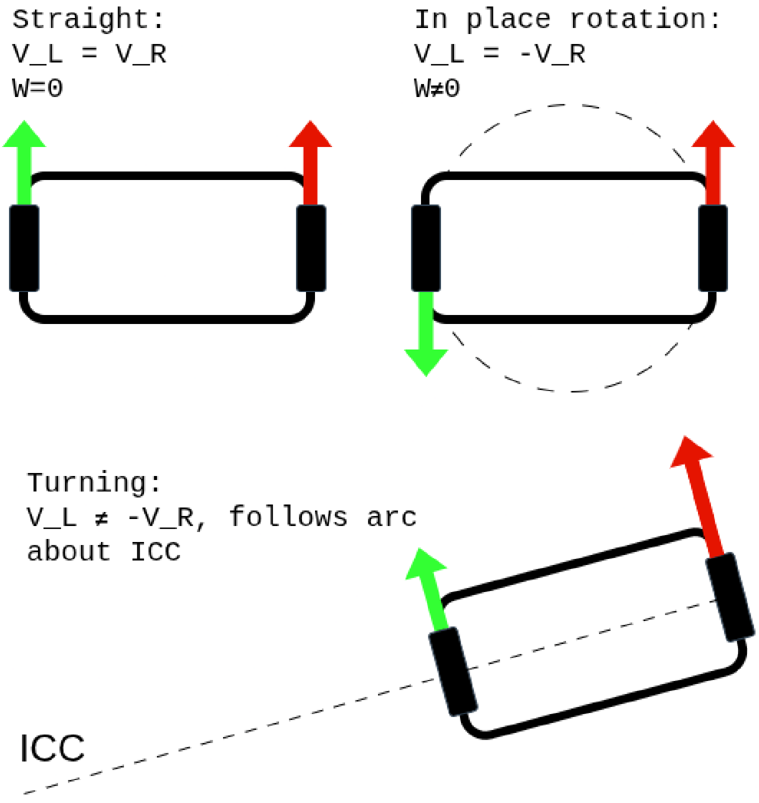

Figure 6: Differential-drive robot motion: pure translation, pure rotation, and turning around an instantaneous center of curvature.

we simply see a pair of bits: 00 if neither sees the line, 01 if only the right sensor sees it, 10 if only the left one does, and 11 if both do.

A very simple strategy

We will start with a bang-bang (on/off) steering rule:

- Pick a base forward speed v (try to move straight ahead).
- Use the sensor readings only to decide whether to turn left, turn right, or not turn.

Intuitively:

- If the right sensor sees the line and the left does not (01), we are drifting right of the line, so we should turn left (counter-clockwise).
- If the left sensor sees the line and the right does not (10), we are drifting left of the line, so we should turn right (clockwise).

- If both see the line (11) or both see nothing (00), we go straight for now.

Translated into a rule for w:

- 00 or 11: $w = 0$.
- 01: $w < 0$ (turn clockwise).
- 10: $w > 0$ (turn counter-clockwise).

Pseudocode

In a simple control loop, we might write:

```
while robot_is_running():
    # Base forward speed (pure straight motion if w = 0)
    v = base_speed # for example, 0.5 m/s

    # How aggressively we turn when we see the line
    rotation_speed = 0.5 # in rad/s, tuned experimentally

    # Read sensors as a pair of bits (L, R)
    sensors = read_line_sensors() # returns (0, 1), (1, 0), etc.

    if sensors == (0, 1): # only right sees the line
        w = -rotation_speed # turn right (clockwise)
    elif sensors == (1, 0): # only left sees the line
        w = rotation_speed # turn left (counter-clockwise)
    else: # (0, 0) or (1, 1)
        w = 0.0 # go straight

    # Convert (v, w) to wheel speeds
    v_l = v - 0.5 * L * w
    v_r = v + 0.5 * L * w

    send_to_motors(v_l, v_r)
```

This controller is extremely simple: it ignores how fast the robot is currently spinning, uses only the current sensor readings, and makes no attempt to predict where the line will be next.

But surprisingly, on a gentle track, it often kind of works: the robot zig-zags around the line and usually stays close enough not to get lost.

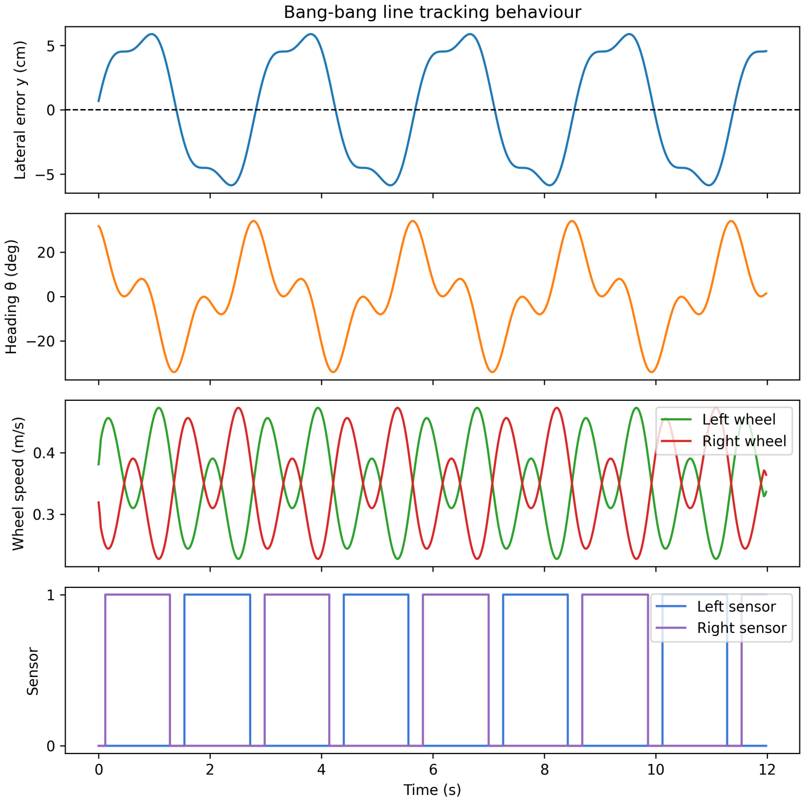

Figure 7: Bang-bang tracking: lateral error, heading, wheel speeds, and sensor states over time.

3.4 What works and what breaks in Attempt 1

Attempt 1 is a useful starting point, but it has several important limitations. We will look at three of them:

1. Wheel saturation (speed limits) and how that distorts our intended motion. 2. Ignoring time-scales and actuator delays. 3. Making decisions from too little information (only current sensor bits, no memory).

Wheel speed limits and saturation

In real robots, wheel commands are bounded; we might only be allowed to command $v_l, v_r \in [-1, 1]$. If we pick a maximum forward speed ($v = 1.0$) and try to add a turn, one of the requested wheel speeds will exceed 1.0. For example, if $v = 1.0$ and we want a turn that requires adding 0.5 to the right wheel, we get $v_r = 1.5$. In practice, the motor driver will likely clip this to 1.0. As a result, the actual motion is not what we assumed when we wrote down the kinematics.

One simple fix is to compute the theoretical v_l, v_r, check if either is out of bounds, and if so, scale both down so that the largest one hits the limit.

```
v_l = v - 0.5 * L * w
v_r = v + 0.5 * L * w

max_mag = max(abs(v_l), abs(v_r))
if max_mag > max_speed:
    scale = max_speed / max_mag
    v_l *= scale
    v_r *= scale
```

This keeps the ratio between v_l and v_r (and therefore the turn rate) roughly correct, while respecting hardware limits.

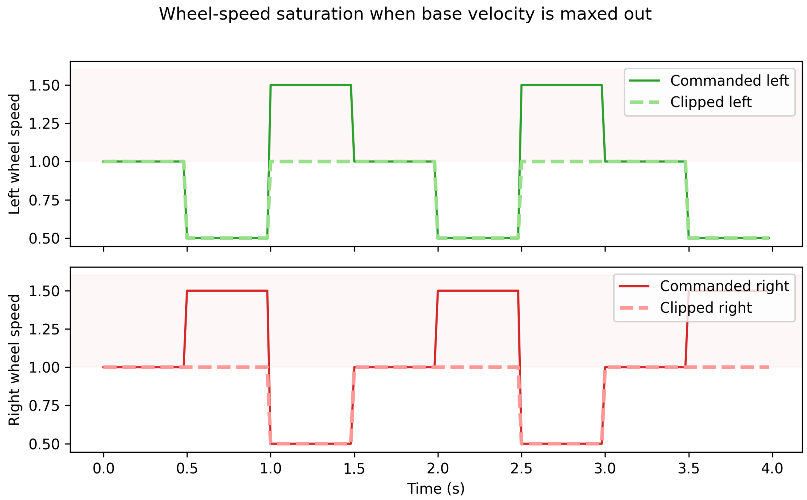

Figure 8: Commanded vs clipped wheel speeds for a saturated base velocity.

Ignoring dynamics and time-scales

In the pseudocode above, we pretend that as soon as we change v_l and v_r in software, the wheel speeds instantly change. Reality is more complicated. The microcontroller updates the motor command almost instantly, but the motor driver and coils respond with a small delay (e.g. 20 ms). The robot body (mass, friction, inertia) takes significantly longer—hundreds of milliseconds—to noticeably change speed or heading.

If we think about the robot's forward speed v:

- A large change (for example from 0.0 to 0.8 m/s) may take 0.5 seconds or more.
- A tiny tweak (from 0.50 to 0.52 m/s) is faster.

The key point is that we can change motor current / acceleration very quickly, but the actual velocity and position change more slowly. These are different time-scales in the same system. Just like in the filters chapter, we must decide which time-scales matter. If we update the bang-bang controller very quickly and always assume instantaneous response, we may ask the robot to do things it physically cannot do that fast.

Decisions from just the current bits

In Attempt 1 we look only at the current sensor pair (L, R), remember nothing about where the line was a moment ago, and ignore any estimate of how far we are from the line or how quickly we are drifting. With just two bits and no memory, decisions stay coarse: turn sharply left or right when a sensor flips, and go straight otherwise. The result is zig-zag paths, over-corrections even when we are nearly centered, and occasional loss of the line on corners or at higher speeds. To do better we need some notion of state (an estimate of where we are relative to the line) and a controller that uses that state more smoothly instead of reacting only when bits flip.

Takeaway: Attempt 1 is simple and can "sort of" follow a line, but it ignores wheel limits, physical delays, and any memory of the recent past. All three will matter when we try to make the robot behave more smoothly and robustly.

3.5 Time-scales: how quickly can we really change things?

Let us look more closely at the idea of time-scales in this specific robot.

Consider the forward speed v of the robot:

- We command a motor current (or a motor power value).
- Motor current is roughly proportional to acceleration (how quickly speed changes).
- Suppose the maximum acceleration we can achieve is limited: $|v'| \leq a_{\max}$.

If we want to change speed by Δv, the fastest we can possibly do it is roughly:

$$\text{time} \approx \frac{\Delta v}{a_{\max}}.$$

Heavier robots have:

- The same motor power, but more mass to move.
- So their maximum acceleration $a_{\max}$ is smaller.
- That means they respond more slowly to the same change in command.

For our line follower, this means:

- Quick changes in commanded wheel speed may not produce quick changes in actual wheel speed or position.
- There is a slow mechanical time-scale on top of the very fast electronic one.

When we design controllers, it often helps to:

- Treat very fast time-scales (microseconds to a few milliseconds) as essentially "instant".
- Focus on the slowest time-scale that dominates behavior we care about (for example, "how quickly can the robot move from one side of the line to the other?").

Takeaway: When you think about how much control you have over a variable (like speed or heading), always ask "how long does it actually take to change this by a meaningful amount?", not just "how fast can I change the command in code?".

3.6 Order of control: position, velocity, and momentum

Another useful idea is order of control:

- Do we directly control position x?
- Or do we control its velocity x'?
- Or only its acceleration x''?

Car analogy: stopping at a finish line

Imagine you are driving a car and want to stop exactly at a finish line.

If you used a naive rule like:

- “If I am left of the finish line, press the gas pedal fully forward.”
- “If I am right of the finish line, press the gas pedal fully backwards.”

you would:

- Overshoot the line.
- Then slam into reverse.
- Overshoot again in the other direction.
- And keep oscillating back and forth.

Why?

- The car’s motion has momentum.
- You do not control position directly; you control acceleration (through engine/brake forces).

If instead you could magically set your position directly (first-order control), you could simply “snap” to the line with much less overshoot.

First-order vs second-order behavior

In many robotics problems:

- We have relatively direct control over a velocity-like quantity.
- Position then comes from integrating that velocity.

For the line follower:

- We can change motor current quickly → change acceleration.
- That changes wheel speed v more slowly.
- That changes the robot’s lateral position and heading even more slowly.

So from the point of view of the line's lateral error y (distance from robot center to the line). We effectively have second-order control (position comes from integrating speed, which comes from integrating acceleration).

Higher "order of control" tends to mean:

- More momentum-like behavior.
- Harder to stop exactly at a target.
- Controllers need to anticipate and slow down near the goal instead of always pushing at full power.

This is why many practical controllers:

- Do not just slam controls to extremes.
- Use smooth feedback (for example, proportional or PID control) that reduces effort as you get close to the desired state.

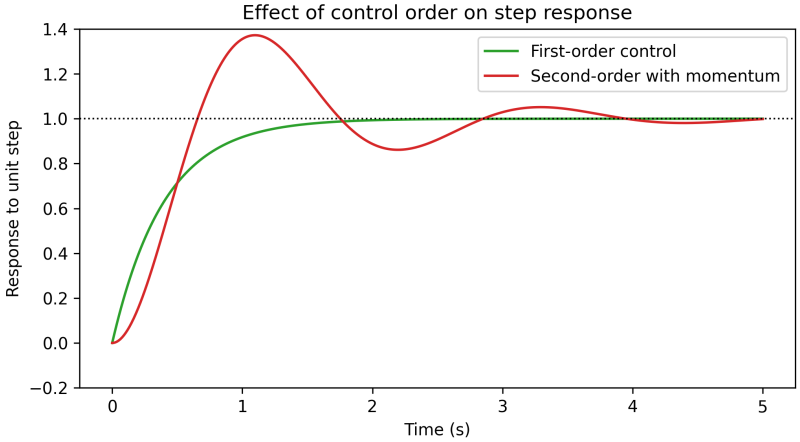

Figure 9: First-order vs underdamped second-order step responses.

Takeaway: For the line follower, we should remember that we are not directly controlling "distance to the line"; we influence it through wheel acceleration and heading. This second-order nature explains much of the overshoot and oscillation we see with naive bang-bang control.

3.7 Attempt 2: use estimated state (y, θ, v, w)

To improve on Attempt 1, we will assume we have access to a few state variables describing the robot relative to the line, and then use a smoother feedback law that adjusts turning and speed based on those states.

Let y be the lateral distance from the robot center to the line, θ be the heading error, v be the forward speed, and $w = \theta'$ be the angular speed. In a real robot, we do not measure all of these directly; we typically estimate them from encoders and line sensors. For now, we pretend some estimator gives us (y, θ, v, w) each time step.

A simple feedback law

We will:

- Pick a base forward speed v_{base}.
- Adjust the turn rate w based on a weighted combination of:
- Lateral error y.
- Heading error θ.
- Current spin rate w itself.

Intuitively:

- If y is positive (robot is to the right of the line), we want to turn left.
- If θ is positive (robot is rotated counter-clockwise relative to the line), we may want to turn back clockwise.
- If w is already large, we may want to reduce it to avoid over-rotating.

We can write:

$$w_{\text{command}} = -c_y y - c_\theta \theta - c_w w,$$

where c_y, c_θ, c_w are tunable gains.

We also slow down when we are far from the line or badly misaligned, to give the robot more time to correct:

$$v_{\text{forward}} = \text{limit}\Big(v_{\text{base}} - k_y|y| - k_\theta|\theta|, 0,\ v_{\text{base}}\Big),$$

where k_y, k_θ are “slow-down” gains.

Pseudocode

Putting this together:

```
def control_step(state):
    # Unpack estimated state relative to the line
    y = state.y # lateral error (m)
    theta = state.theta # heading error (rad)
    v = state.v # current forward speed (m/s)
    w_curr = state.w # current angular speed (rad/s)

    # Tuning parameters
    base_speed = 0.5
    position_gain = 1.0 # c_y
    angle_gain = 0.8 # c_theta
    spin_gain = 0.2 # c_w
    slow_gain_y = 0.8 # k_y
    slow_gain_th = 0.8 # k_theta

    # Compute a turn command from position, angle, and current spin
    turn_from_y = -position_gain * y
    turn_from_th = -angle_gain * theta
    turn_from_spin = -spin_gain * w_curr

    w_cmd = turn_from_y + turn_from_th + turn_from_spin

    # Slow down when we are far from the line or badly misaligned
    v_cmd = base_speed
    v_cmd -= slow_gain_y * abs(y)
    v_cmd -= slow_gain_th * abs(theta)
    v_cmd = limit(v_cmd, 0.0, base_speed)

    # Convert (v_cmd, w_cmd) to wheel speeds and enforce limits
    v_l = v_cmd - 0.5 * L * w_cmd
    v_r = v_cmd + 0.5 * L * w_cmd

    v_l, v_r = limit_wheels(v_l, v_r, max_speed)
    send_to_motors(v_l, v_r)
```

Compared to Attempt 1, this controller reacts continuously to how far we are from the line and how we are angled, slows down when things look bad instead of charging ahead, and takes into account the current spin w to avoid over-rotating. With reasonable gains, the robot tends to follow the line more smoothly, overshoot less on corners, and recover more gracefully after small disturbances.

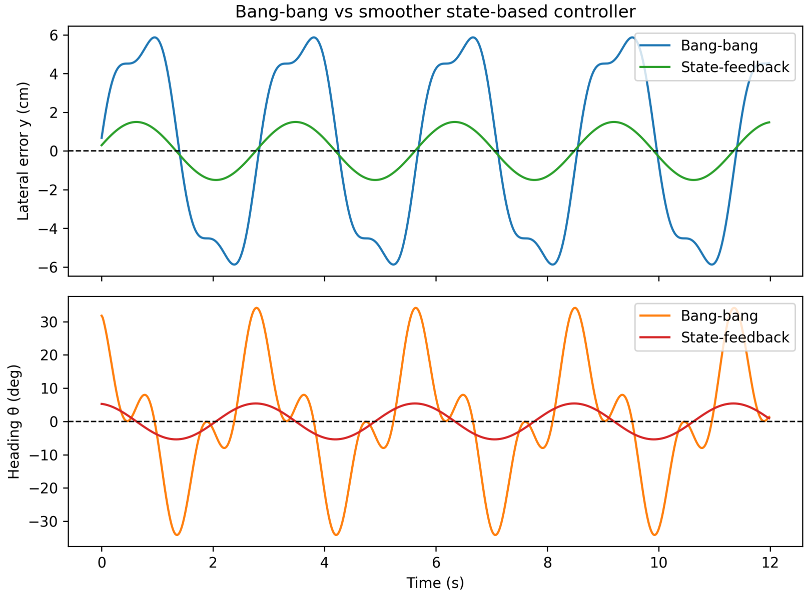

Figure 10: Improved controller tracking: smoother lateral error and heading vs. bang-bang.

Takeaway: By estimating a small state (y, θ, v, w) and using it in a simple feedback law, we turn a twitchy bang-bang line follower into something that behaves much more like a smooth, well-damped system.

3.8 Sketch: estimating (y, θ, v, w) from encoders and line sensors

We have so far pretended we know y, θ, v, w. Now we will briefly sketch how we might estimate them in practice.

We assume:

- We can read wheel speeds v_l, v_r from encoders (or at least estimate them).

- We still have the two binary line sensors near the front.
- We run a control loop at a fixed time step Δt.

Getting v and w from encoders

From the differential-drive kinematics:

$$v = \frac{v_r + v_l}{2}, \qquad w = \frac{v_r - v_l}{L}.$$

If we update this every control cycle:

- We get reasonably good short-term estimates of forward speed and angular speed.
- Encoders are usually quite reliable over short time-scales.

Predicting y and θ by dead-reckoning

We define y and θ relative to the line:

- y: lateral distance from robot center to the line (positive to one side).
- θ: heading error between robot's direction and the line direction.

If during one small time step Δt we move with speeds v and w, we can predict how y and θ change:

$$\theta \leftarrow \theta + w\Delta t,$$

$$y \leftarrow y + v \sin(\theta)\, \Delta t.$$

For small θ, we often approximate $\sin(\theta) \approx \theta$, which keeps the math simple and still captures how heading error slowly turns into lateral error.

So a basic predict step is:

```
theta = theta + w * dt
y = y + v * math.sin(theta) * dt # or y += v * theta * dt for small theta

# Keep theta in [-pi, pi] to avoid numerical drift
theta = wrap_angle(theta)
```

This is called dead-reckoning: we update our state based on how we think we moved, without looking at sensors yet.

Correcting using the line sensors

The two line sensors sit at known lateral offsets from the robot center:

- Left sensor: at $+d/2$.
- Right sensor: at $-d/2$.

When a sensor reports "line", we can treat it as a noisy measurement that the line passes under that sensor's lateral position.

We can turn the two bits (L, R) into a crude measurement of y:

```
def lateral_measurement_from_sensors(L, R, d):
    if (L, R) == (1, 0): # line under left sensor only
        return +0.5 * d
    elif (L, R) == (0, 1): # line under right sensor only
        return -0.5 * d
    elif (L, R) == (1, 1): # both see line -> roughly centered
        return 0.0
    else: # (0, 0): no direct measurement
        return None
```

At each time step:

1. We run the predict step to update y and θ. 2. We read sensor bits (L, R) and turn them into a tentative measurement `y_meas`. 3. If `y_meas` is not `None`, we blend it with our predicted y.

The blending can be as simple as an exponential moving average:

```
alpha = 0.3 # between 0 and 1; how much we trust the sensor

if y_meas is not None:
    y = (1.0 - alpha) * y + alpha * y_meas
```

This is a tiny predict–correct filter:

- If the sensors agree with dead-reckoning, we only nudge y a little.

- If they disagree, y slowly drifts toward what sensors say.
- If sensors see nothing for a few steps, we rely on prediction alone.

In this simple sketch:

- We mainly correct y using the line sensors.
- We let θ evolve from the encoder-based w.
- With a bit more hardware (for example, a third sensor further forward, or a gyro), we could also correct θ explicitly.

Putting it all together

A single estimator step might look like:

```
def estimator_step(state, v_l, v_r, sensors, dt):
    # 1. Compute v and w from wheel speeds
    v = 0.5 * (v_r + v_l)
    w = (v_r - v_l) / L

    # 2. Predict theta and y from motion
    state.theta += w * dt
    state.theta = wrap_angle(state.theta)

    state.y += v * math.sin(state.theta) * dt

    # 3. Correct y using line sensors, if available
    L_bit, R_bit = sensors
    y_meas = lateral_measurement_from_sensors(L_bit, R_bit, sensor_spacing_d)

    if y_meas is not None:
        state.y = (1.0 - alpha_y) * state.y + alpha_y * y_meas

    # 4. Update stored v and w for the controller
    state.v = v
    state.w = w
```

This estimator is intentionally simple:

- It ignores many real-world complications (slip, curvature of the line, sensor noise patterns).
- But it already turns raw encoder ticks and two bits of line information into a continuous state that the controller can use.

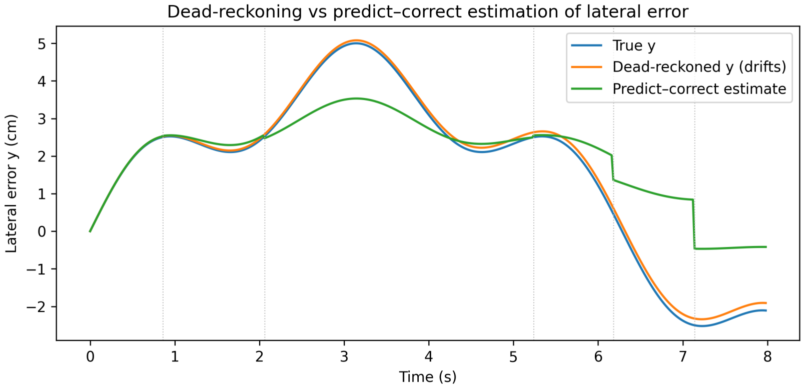

Figure 11: Predict–correct estimation of lateral error using encoders and line sensors.

Takeaway: By combining encoder-based prediction with occasional, coarse line-sensor measurements, we can maintain a usable estimate of where the robot is relative to the line – even though each sensor alone is very limited.

3.9 Summary and takeaways

In this chapter, we used the line follower as a small but rich example of control in robotics.

- Start simple: A bang-bang controller that turns only when a sensor sees the line can often follow a track, but it naturally zig-zags and is sensitive to speed, curvature, and delays.
- Respect physics: Wheel saturation, actuator limits, and slow mechanical dynamics mean we cannot instantly get the motion we ask for in code.
- Think in time-scales and order of control: Understanding what we control directly (acceleration vs velocity vs position) explains overshoot and oscillation.
- Use state and estimation: A small estimated state (y, θ, v, w), combined with a smooth feedback law and a simple predict–correct estimator, already yields a much more robust and graceful line follower.

As you build more complex robots, you will see the same pattern repeat: start with a naive controller to understand the problem, notice where it breaks, and then introduce state, estimation, and smoother feedback to tame time-scales and momentum.

4 Filters and Noisy Sensors

4.1 The problem: noisy sensors

When you write your first robot program, it is tempting to think that you can simply read a sensor once, get the distance, angle, or speed, and immediately use that number to decide what to do. Reality is much less kind.

Sensors lie all the time – not in a dramatic way, but in a small, jittery, annoying way. If we take every single reading literally, the robot ends up twitchy, overreacting to every tiny blip.

In this chapter, we will talk about filters: small pieces of code that take a messy sequence of sensor readings and turn them into smooth, believable numbers you can actually control with.

We will keep the examples very simple and self-contained. Just like in the line follower chapter, we will keep coming back to a few concrete, easy-to-visualize scenarios:

1. A robot with a noisy distance sensor pointing at a wall. 2. A motor with an encoder whose speed measurement jumps around a bit. 3. A simple robot with line sensors trying to follow a (black) line on the (white) floor.

Example: a shaky distance sensor

Imagine a small robot sitting still, facing a wall. It has a distance sensor on the front that tells you "how many centimeters to the wall".

In the real world, if the robot is not moving, the true distance is constant – for example, exactly 1.0 m.

But when you log the sensor readings over time, you might see something like:

- 0.94, 1.02, 1.10, 0.98, 1.01, 0.93, 1.06, 0.99, . . .

Nothing dramatic is happening in the world – the robot is not moving and the wall is not moving – but the readings jump around by a few centimeters because of noise, reflections, small changes in lighting, etc.

If, each control loop, you directly use the latest reading to decide what to do ("If distance < 0.95, back up!"), the robot might nervously twitch in and out of backing-up mode, even though nothing meaningful changed.

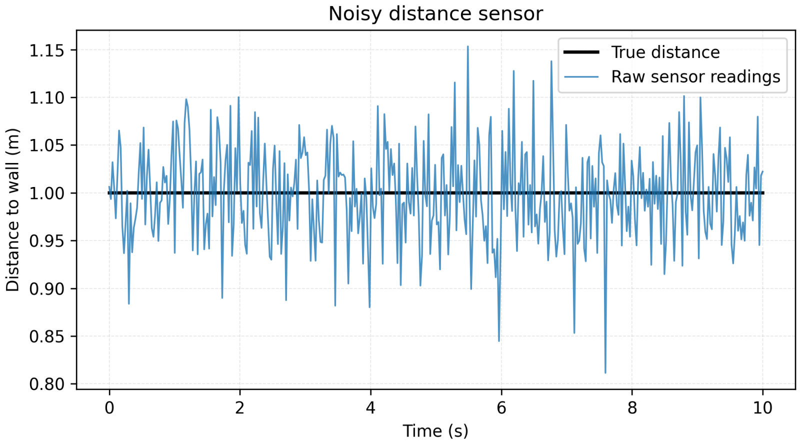

Figure 12: Noisy distance sensor readings over time: a jagged line bouncing around the true constant distance.

Example: noisy motor speed from an encoder

Now imagine you have a wheel with an encoder. You send a fixed power command to the motor (for example, 0.4 on a scale from -1 to 1), and you expect the wheel speed to settle somewhere. You then compute speed every 20 ms from the encoder ticks.

If you plot the computed speed, you might see:

- Sometimes 0.35 m/s, sometimes 0.39, sometimes 0.32, sometimes 0.37. . .

Again, the real physical speed is almost constant, but your estimated speed jumps around because of quantization (integer tick counts), friction changes, and small timing variations. If you feed this noisy speed straight into a controller (for example, one that tries to keep speed close to a goal), the controller will "see" imaginary changes and keep overreacting.

4.2 Filters: smoothing and "controlled forgetfulness"

The core idea in this chapter is simple: do not trust a single reading. Instead, combine many readings over time into a smoother, more believable value.

We will start with simple averages over time, move to leaky memory (exponential moving averages), and then think in terms of time-scales (what changes fast vs slow). From there we will combine prediction (what we expect) with correction (what sensors say) and finally show how to combine different sensors using a simple "complementary filter".

Throughout, we will focus on intuition and graphs, not formal control theory.

4.3 A simple noise model and why averaging helps

Before we build any filters, it is useful to have a very simple mental model of what "noise" means.

Let x_{true} be the quantity we care about (for example, the real distance to a wall), and let z be what the sensor reports. A very common model is:

$$z = x_{\text{true}} + n,$$

where n is noise:

- Sometimes n is positive, sometimes negative.
- Over time, it tends to *average out* toward zero.
- Its exact value at each time step is mostly out of our control.

If we take many independent noisy measurements $z_1, z_2, \ldots$ of the *same* underlying value x_{true}, then each individual measurement is off by its own noise n_k, but the average of many measurements tends to move closer to x_{true} because positive and negative noise cancel.

This is the statistical reason why averaging reduces noise. We will not go deep into probability theory here; for our purposes it is enough to remember:

> If the underlying quantity is not changing quickly, and the sensor noise wobbles around zero, then averaging several readings gives a better estimate than any single reading.

The rest of this chapter shows different ways to exploit that idea while keeping lag under control.

4.4 First tool: simple averaging over time

The simplest way to make a noisy number calmer is to average the last few readings. Instead of using just the latest measurement, we keep a short memory of the recent past and take the average.

This already gives us a very useful filter.

Moving average (box filter)

Suppose our distance sensor gives readings

- $d_1, d_2, d_3, \ldots$

At time step k, instead of using just d_k, we use the average of the last N readings:

$$\bar{d}_k = \frac{d_k + d_{k-1} + \cdots + d_{k-N+1}}{N}.$$

This is called a moving average (or a box filter).

- If $N = 1$, we are not filtering at all – we just use the latest reading.
- If N is bigger, we smooth more aggressively.

Example with the distance sensor

Imagine we take a moving average with $N = 5$. Every control cycle, we:

1. Read the latest sensor value.
2. Put it into a small buffer of the last 5 readings.
3. Compute the average of those 5 numbers.
4. Use this average for our decisions instead of the raw reading.

Now, a single "bad" reading (for example, 0.90 instead of 1.0) does not instantly make your robot panic and back away. It only pulls the average slightly down.

What do we gain and what do we lose?

The moving average gives us two main effects: less noise (random spikes get averaged out) but more lag (the filtered value reacts more slowly to real changes).

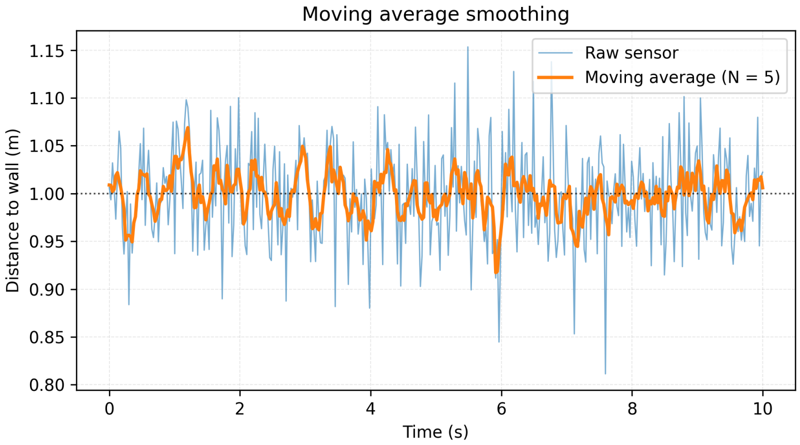

Figure 13: Raw noisy distance vs 5-sample moving average: jagged raw line and smoother averaged line.

This tradeoff shows up everywhere in filtering. A large N means the filter is very smooth but slow to react; a small N makes it less smooth but quick to respond. You can think of it as the filter's "stubbornness": with large N, it remembers a lot of the past and does not change its mind easily. With small N, it is "forgetful", changing quickly but being easily fooled by noise.

Implementation sketch

Here is a simple way to implement a moving average over the last N readings for any 1D sensor:

```
N = 5
buffer = [0.0] * N
index = 0
count = 0 # how many valid entries we have so far

def update_filter(new_value):
    global index, count
    buffer[index] = new_value
    index = (index + 1) % N
    count = min(count + 1, N)

    # compute average of valid entries
    total = 0.0
    for i in range(count):
        total += buffer[i]
    return total / count
```

This is not the most efficient possible implementation, but it is very clear and easy to adapt.

Takeaway: The moving average is the simplest filter: it looks at a fixed-sized window in time, smooths out noise, and introduces some lag.

Worked numeric example

To make this concrete, imagine the true distance to a wall is exactly 1.00 m, but the sensor readings (in meters) over time are:

$$0.94,\ 1.02,\ 1.10,\ 0.98,\ 1.01.$$

Let $N = 5$ and compute the average over these five readings:

$$\bar{d} = \frac{0.94 + 1.02 + 1.10 + 0.98 + 1.01}{5} = \frac{5.05}{5} = 1.01 \text{ m}.$$

Individually, some readings are off by as much as 6 cm, but the average is only 1 cm away from the truth. If we kept adding more readings from the same still robot, the running average would usually hover very close to 1.0 m, even though any single measurement is noisy.

This is exactly the “noise cancels, truth remains” effect that moving averages and exponential filters take advantage of.

4.5 Second tool: exponential moving average (“leaky memory”)

The moving average uses a hard window: last N readings all count equally; anything older is forgotten completely. == There is another very common and often more convenient filter:

$$x_{\text{filtered}} \leftarrow (1-\alpha)\, x_{\text{filtered}} + \alpha\, x_{\text{new}}.$$

This is called an exponential moving average or a first-order low-pass filter. We will think of it as leaky memory: we keep a memory of the past value, and every time we get a new reading, we move our memory a little bit towards the new value.

The update rule

Assume we maintain a variable `filtered` that stores our current best guess. Each time a new measurement `m` arrives, we do:

```
alpha = 0.2 # between 0 and 1

filtered = (1.0 - alpha) * filtered + alpha * m
```

You can also write it as:

```
filtered += alpha * (m - filtered)
```

which reads: "take a step of size `alpha` towards the new measurement".

Interpreting alpha

The parameter α controls how quickly the filter reacts. If α is small (e.g. 0.05), the filter changes slowly and has long memory, resulting in a very smooth signal but more lag. If α is large (e.g. 0.5), the filter changes quickly and has short memory, making it less smooth but more responsive.

You can think of `filtered` as a heavy object on a table, and `m` as a hand that nudges it toward a new position every time step. α is how hard you push: the harder you push, the faster it moves.

Step response: reacting to real changes

To understand lag, imagine the true distance suddenly changes:

- At time 0, the wall suddenly moves closer, and the true distance jumps from 1.0 m to 0.5 m.

Assume for the moment that our sensor is perfect (no noise) and instantly reports the new value 0.5 m. Even then, the filtered value won't jump immediately; it will approach 0.5 gradually.

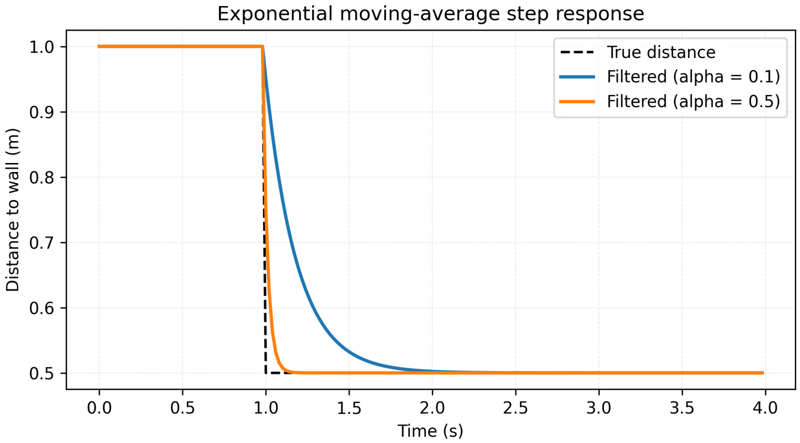

Figure 14: Step response of exponential moving average: true distance jumps from 1.0 to 0.5, filtered curves for different α.

This shows clearly:

- Smaller α means slower adaptation to real changes.
- Larger α means faster adaptation, but it also lets more noise through.

Noisy constant signal: reacting to noise

Now go back to the original situation where the true distance is constant at 1.0 m, but the readings bounce around.

If we apply the exponential moving average to the noisy readings:

- For small α, the filtered line is very smooth and stays near 1.0, ignoring most random jumps.
- For large α, the filtered line still follows the noise somewhat, but less than the raw signal.

Comparison with moving average

Both the moving average and the exponential filter reduce noise and introduce lag, but they differ in implementation. The moving average needs to store the last N samples,

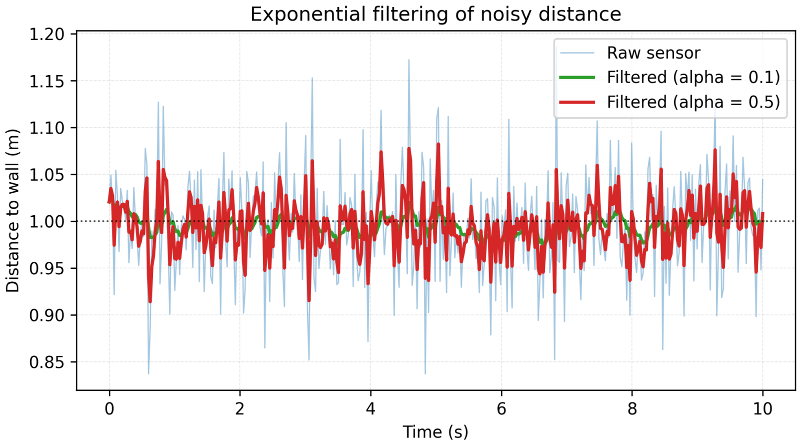

Figure 15: Noisy distance readings with exponential moving average filters for different α values.

gives them equal weight, and completely forgets anything older than the window. The exponential filter needs to store only one number (`filtered`), gives higher weight to recent samples, and lets older samples fade smoothly without a sharp cutoff.

For many robotics applications, the exponential filter is a very convenient default: it is easy to implement, uses little memory, and lets you easily tune how quickly the filter "believes" new readings.

Takeaway: Exponential filters are like a memory that slowly forgets the past. The parameter α is how quickly it forgets.

4.6 Time-scales: what we keep and what we ignore

So far, we have mostly thought, "we want less noise", and we used filters to make graphs look smoother. There is another way to think about filters that is very useful in robotics:

> What time-scales do we care about?

In other words:

- What changes fast that we want to ignore?
- What changes slowly that we want to keep?

Motor speed: fast wiggles vs slow changes

Return to the motor speed example. You send a fixed power command and measure speed with an encoder every 20 ms.

If you zoom into the data, you might see:

- Very fast wiggles every 20–40 ms, due to individual encoder ticks.
- Much slower changes over hundreds of milliseconds or seconds, due to friction changes, battery voltage, or you commanding a new target speed.

If your control loop runs at 50 Hz (every 20 ms), you might ask:

- Do I really care about tiny changes that happen within one or two time steps?
- Or am I mostly interested in "the average speed over a few tenths of a second"?

If you decide that only slower changes matter, then it is natural to deliberately remove fast wiggles with a filter.

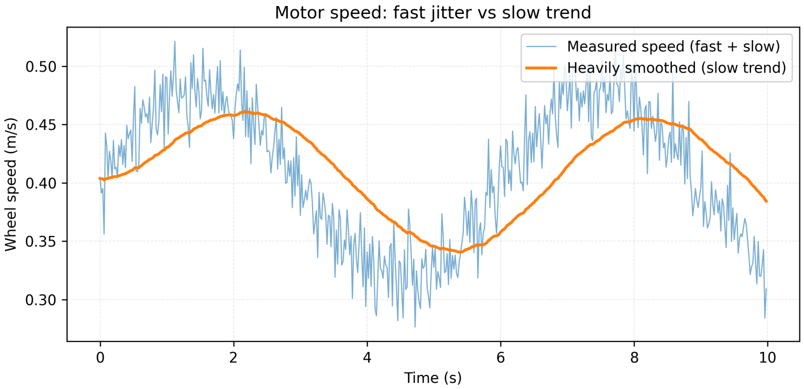

Figure 16: Motor speed signal split into fast noise and slow trend.

Distance sensor: robot approaching a wall

Consider the distance robot again, but now it is driving towards the wall at a slow, steady speed.

The true distance vs. time is a smooth, gently decreasing curve. On top of that, the raw sensor gives noisy, jittery samples.

If you choose α so that the filter reacts on roughly the same time-scale as "how quickly we approach the wall", then:

- The filtered distance is smooth enough to ignore random noise.
- But it still tracks the actual closing distance reasonably quickly.

If you choose α too small (filter too slow):

- The filter might report "still far away" even when you are already very close, because it is still stuck remembering the past.

If you choose α too large (filter too fast):

- You get almost no benefit; the filtered output still jumps almost as much as the raw data.

The right choice depends on:

- How fast the real world is changing.
- How quickly your control loop reacts to changes.

Filters as time-scale selectors

One useful mental model is that a filter is a time-scale selector: it passes through changes that are slower than its "reaction time" while suppressing changes that are faster.

In practice, when you choose α or N, you are answering:

> How many control-loop steps should it take for my estimate to respond to
> a real change?

Takeaway: Instead of thinking "I just want less noise", think "I want to ignore changes faster than X, and still see changes slower than X". Then pick filter parameters that roughly match that time-scale.

4.7 Predict and correct: filtering as "best guess of hidden state"

So far, our filters have only used current measurements and past filtered values. We have not used any knowledge of how the robot is *supposed* to move.

But in many robotics problems, we know something about how the world behaves:

- If a robot moves with constant speed v, its position changes by $v\Delta t$ every time step.
- If we are facing a wall and we know our forward speed, we can predict how the distance should shrink.

We can use this to build a slightly smarter filter:

1. Predict what we expect the state to be, based on the previous state and our motion.
2. Correct that prediction using a (noisy) measurement.

This is still a filter, but now it also uses a simple model of the world.

1D example: tracking distance to a wall

Consider again the robot driving straight toward a wall. We want to estimate the distance d over time.

We have:

- Control input: forward speed command v_{cmd}.
- Unknown true state: actual distance to the wall d.
- Noisy measurements: sensor readings z (distance sensor).

We will keep a variable `d_est` – our estimated distance.

Every control loop (every Δt seconds) we do:

1. Predict how distance changes due to motion. 2. Correct that prediction based on the noisy sensor.

Predict step

If the robot is moving roughly straight towards the wall at speed v, distance should shrink by $v\Delta t$ each step. So we can predict:

```
d_est = d_est - v * dt
```

This uses our model: "distance decreases at rate v".

Of course, this prediction is not perfect:

- Maybe the floor is slippery and the wheel slips.
- Maybe the robot slightly slows down going up a ramp.

But it still gives us a reasonable guess of what *should* happen if everything is ideal.

Correct step: blending with measurement

Next, we read the noisy distance measurement `z` from the sensor.

We could directly replace `d_est` with `z`, but then we would throw away our prediction. Instead, we combine them:

```
beta = 0.2 # how much we trust the sensor vs our prediction

innovation = z - d_est # how wrong our prediction was
d_est = d_est + beta * innovation
```

Again, this is just an exponential filter, but now applied to the prediction error. We are saying:

- If the sensor reading mostly agrees with our prediction, we change `d_est` only a little.
- If the sensor reading strongly disagrees, we move `d_est` more toward the sensor.

Putting it together

Every loop:

```
# 1. Predict
d_est = d_est - v * dt

# 2. Measure
z = read_distance_sensor()

# 3. Correct
beta = 0.2
d_est = d_est + beta * (z - d_est)
```

This is a tiny “predict–correct” filter.

Why bother predicting?

At first glance, this might feel like extra work. Why not just filter the sensor directly with an exponential average?

The advantage of prediction shows up when:

- Sensors are temporarily unavailable (for example, the distance sensor gets saturated at very close range).
- Sensors are extremely noisy in some situations.
- You have a decent idea of how the state should evolve (for example, wheel encoders are reliable in the short term).

When the sensor is missing or very noisy, prediction alone can carry the estimate for a short time. When the sensor is available, we gently pull the estimate back toward reality.

A scalar Kalman-style view (optional)

If you have seen the Kalman filter before, you might recognize the structure of the predict–correct code. In a very simple 1D case the Kalman filter looks almost identical, but with a carefully chosen gain.

Suppose we track distance d to a wall with a constant-velocity model and noisy distance measurements:

- State estimate: d_{est}.
- Process model: d decreases by $v\,\Delta t$ each step.
- Measurement: $z = d + n$, with some measurement noise n.

We can write:

- Predict:

$$d_{\text{pred}} = d_{\text{est}} - v\,\Delta t.$$

- Correct:

$$d_{\text{est}} \leftarrow d_{\text{pred}} + K\,(z - d_{\text{pred}}),$$

- where K is a gain between 0 and 1.

This is just like the earlier code where we used a parameter `beta` to blend prediction and measurement. The full Kalman filter goes further:

- It keeps track of how uncertain the estimate is.
- It automatically adjusts K based on how noisy the sensor is and how uncertain the prediction is.

Conceptually, though, it is still the same predict–correct loop you have already seen: use a model to predict, use a measurement to correct.

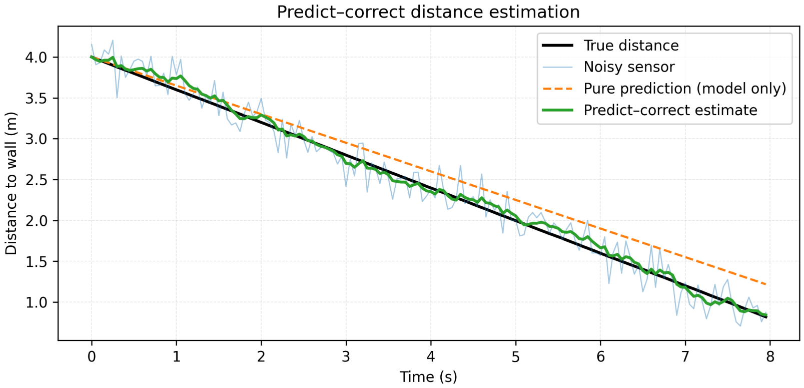

Figure 17: Predict-correct distance estimation: true distance, raw sensor, predicted distance, and filtered estimate.

Takeaway: You can think of filtering as "What is my best guess of the hidden state, combining what my model predicts and what my sensors say?"

4.8 Combining sensors with a complementary filter

So far, we have mostly talked about a single sensor. In many robots, we measure the same physical quantity in more than one way, using different sensors.

For example, consider estimating the tilt angle (pitch or roll) of a robot:

- An accelerometer can estimate tilt by measuring gravity, but its reading is very noisy in the short term and sensitive to linear acceleration.
- A gyroscope measures angular velocity very well in the short term, but its integration over time drifts slowly (small bias accumulates).

We have:

- Accelerometer: good long-term truth, bad short-term noise.
- Gyro: good short-term changes, bad long-term drift.

We would like an estimate that is good in both senses.

Intuition: two friends with different strengths

Imagine two friends. Friend A (accelerometer) is calm and reliable in the long run, but their instant reactions are a bit jittery. Friend B (gyro) reacts quickly and tracks rapid changes well, but gets slowly confused over many seconds.

To get the best overall advice, you mostly listen to Friend A for slow trends and Friend B for fast changes. This is exactly what a complementary filter does.

Basic complementary filter for tilt

Let:

- θ_{gyro} be the tilt estimate from integrating the gyro.
- θ_{accel} be the tilt estimate from the accelerometer.
- θ be our combined estimate.

A simple complementary filter update is:

1. Integrate the gyro to predict the new angle:

```
theta = theta + gyro_rate * dt
```

2. Correct the slow drift using the accelerometer reading:

```
k = 0.02 # small correction factor
theta = (1.0 - k) * theta + k * theta_accel
```

This says:

- Most of the time, follow the gyro (fast, responsive).
- Very slowly, nudge the estimate back toward the accelerometer's long-term truth.

Viewing it as combining filters

Another way to see it is that the gyro-based angle is effectively high-pass filtered (capturing fast changes but drifting at low frequencies), while the accelerometer-based angle is low-pass filtered (noisy instant-by-instant but correct on average). The complementary filter simply adds a high-pass part from the gyro and a low-pass part from the accelerometer. They are "complementary" because one covers what the other is weak at.

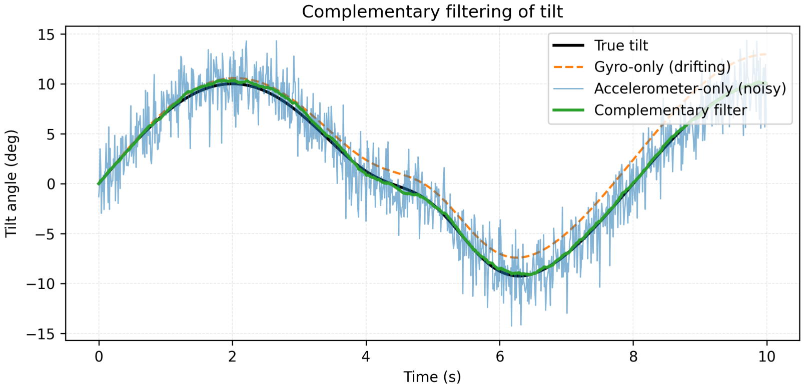

Figure 18: Complementary filter for tilt: gyro-only, accelerometer-only, and combined estimates over time.

Connections to earlier ideas

Notice how the complementary filter is built from the same ingredients as before: it uses prediction (integrating the gyro) and correction (blending toward accelerometer), where the correction step is just another exponential filter with a small gain k. It relies on time-scales, trusting different sensors at different speeds. The main new idea is that when you have multiple sensors, you can design filters that intentionally mix their strengths.

Takeaway: Complementary filters are simple, powerful ways to combine a "fast but drifting" sensor with a "slow but stable" one.

4.9 Using filters in simple robotics problems

Let us briefly return to our earlier two examples and see how we might actually use filters in code.

Distance-based stopping

Suppose you want the robot to drive towards a wall and stop smoothly at 0.3 m.

Without filtering:

- You might write `if distance < 0.3:  stop`.
- With noisy readings, sometimes you see 0.29 m early and stop too soon, or 0.31 then 0.28 and jitter.

With a simple exponential filter:

1. Maintain `d_filtered` using the exponential rule. 2. Decide based on `d_filtered`:

```
  if d_filtered < 0.3:
      stop()
```

Now:

- Random spikes below 0.3 m are less likely to trigger a stop.
- The robot responds to a trend (distance really shrinking) rather than a single lucky reading.

Note that you might need to slightly alter (increase) threshold, so that `d_filtered` actually stops at 0.3.

Motor speed control

Suppose you want to control motor speed using encoder feedback. You measure speed every control loop and compare it to a desired speed.

If you feed the raw speed into your controller:

- The controller "thinks" speed is bouncing, so it keeps changing the motor command.
- The motor command becomes noisy, which can make the actual speed noisier too.

If you instead filter the measured speed:

```
speed_measured = compute_speed_from_encoder()
speed_filtered = (1.0 - alpha) * speed_filtered + alpha * speed_measured

```

```
error = target_speed - speed_filtered
command = k_p * error # simple proportional controller, for example
send_to_motor(command)
```

Then:

- The controller reacts to smoothed speed.
- Small, high-frequency measurement noise does not cause large corrections.

You still have to choose α carefully:

- Too small (α very low): the controller reacts slowly when you actually change the target speed.
- Too large: you get less filtering and more noisy commands.

Takeaway: A very common pattern is: filter the measurement before giving it to the controller. This usually works better than filtering the command.

4.10 Summary, practical recipes, and common pitfalls

To close the chapter, here are some simple, practical guidelines.

When to use a simple average vs exponential filter

If you:

- Are just starting out,
- Have a signal that you read at a fixed rate,
- And want something easy to tune,

then start with an exponential filter.

If:

- You like the idea "keep the last N samples and average them",
- And you don't mind a bit of extra memory,

then a moving average is also fine, especially for slower signals like battery voltage or very slow distance changes.

In many robotics systems, exponential filters are preferred because they are:

- Simple (one line of code).
- Memory-light.
- Easy to adjust by changing a single number α.

How to pick alpha (or N)

A rough way:

- Decide how quickly you want the filtered value to react to a real step change (for example, "be close to the new value in about 0.5 seconds").
- Convert that feeling into a proportion of your control-loop rate.

For example:

- If your control loop runs at 50 Hz (every 0.02 s) and you want the filter to react on the order of half a second (about 25 steps), you might start with α around 0.1–0.2.
- Then log data, plot it, and adjust until it "feels right" for your application.

There is a lot of theory about optimal choices, but in practice, a bit of plotting and intuition goes a long way.

Common pitfalls

- Filter is too slow (too much lag).
  - Symptom: the robot reacts late to real changes (for example, braking too late because filtered distance says you are still far away).
  - Fix: increase α or decrease N so the filter reacts faster.
- Filter is too fast (almost no smoothing).
  - Symptom: filtered signal still looks very noisy; controller output is jittery.
  - Fix: decrease α or increase N.
- Filtering the wrong thing.

  - Symptom: Robot still behaves jittery or unstable, even though you applied a filter somewhere.
  - Fix: Usually, you want to filter measurements (sensor values or estimated states), not the commands you send to actuators.

- Ignoring time-scales.
  - Symptom: You pick α blindly and later realize that either you are filtering away changes you actually care about, or you are still responding to noise you did not care about.
  - Fix: Think explicitly: "What is the fastest change in this quantity that I care about?" and set the filter to be just a bit faster than that.

Implementation checklist

When you actually add filters to a robot project, it helps to follow a small checklist:

- Decide where filtering belongs in the loop:
  - Read raw sensors.
  - Update filter state to get filtered values.
  - Feed filtered values into controllers or estimators.

  Keep filter state next to the signal it belongs to:
  - For each sensor or estimated variable, store its own `filtered` value and any extra state (buffers, sums).

  Log both raw and filtered values:
  - Plot them together to see noise vs lag.
  - Adjust α or N until the trade-off looks reasonable.

  Match update rates:
  - Run the filter at the same rate as the control loop or sensor.
  - If you downsample, be explicit about the new effective Δt.

Final thoughts

Filters are not magic. They cannot create information that is not there. But they are extremely useful at:

- Ignoring small, fast, meaningless fluctuations.
- Producing calm, believable numbers you can use for control.
- Combining knowledge of how the world should move with noisy sensor readings.

As you build more complex robots, you will encounter more advanced filters (like the Kalman filter), but the core ideas remain the same as in this chapter: average over time, decide which time-scales you care about, predict what should happen, and correct based on what you actually see.

If you keep these simple pictures in mind – noisy plots, smoothed plots, and step responses – you will already have strong intuition for how filters behave, long before you see any heavy math.

5 Geometric Foundations for Robotics

Geometry provides the mathematical language necessary to describe the position, orientation, movement, and interaction of robotic systems with their environment.

Introduction: Role of Geometry in Robotics Geometry forms the basis of robotics across various subdisciplines in many regards. This chapter will focus on:

1. Kinematics: Describing the motion of robot links and joints without considering the forces causing the motion.

2. Planning: Generating trajectories and sequences of movements that avoid obstacles and reach target configurations.

3. Perception: Interpreting sensory data (e.g., from cameras or LiDAR) to understand the robot's environment and its own state.

5.1 Transformations

Transformations allow the robot to relate different coordinate systems (e.g., the robot's base frame, a tool frame, and the world frame). A transformation can be either a translation rotation, or both. Translation changes position without changing orientation and rotations change the orientation around an axis. Rotations can be represented in several ways including a rotation matrix, Euler angles, or quaternions. The three options all have their relative benefits.

Rotation matrices are 3x3 orthogonal matrices with a determinant of 1. They are the most common rotation representation as they are easy to use, preserve vectors, and don't suffer gimbal lock. Being a 3x3 matrix there are redundant parameters and is not the most computationally efficient of the rotation representations.

*Gimbal lock is a singularity in Euler angle representation where two rotation axes align, causing the loss of one degree of freedom and preventing certain orientations from being uniquely represented.

Euler angles are another common rotation representation that use complex formulas to represent 3 angles: roll, pitch, and yaw. Euler angles are more compact than rotation matrices being composed of only three values, making it a more compact storage solution. The key disadvantage is that it does suffer from gimbal lock where it contains singularities and non-unique solutions.

Most commonly found in kinematic derivations for robotic manipulators, the homogeneous transformation is a matrix of the shape 4x4 containing both the rotation matrix and the translation vector. The format is particularly useful when a kinematic system undergoes many transformations in order to solve for the position of the end effector. To find the transformation between any two joints a manipulator transforms, the individual transformations, often derived from the Denavit-Hartenberg convention, are multiplied together to yield a complete transformation. See the next section.

Rigid transforms and the groups SE(2) and SE(3)

Homogeneous transforms live in mathematical objects called *special Euclidean groups.* SE(2) represents planar rigid motions (position (x, y) and heading θ), while SE(3) represents full 3D rigid motions (position (x, y, z) and 3D orientation).

An element of SE(3) is a matrix of the form

$$T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix},$$

where:

- R is a 3×3 rotation matrix with

$$R^{\top} R = I, \qquad \det R = 1,$$

- so it preserves lengths and angles.
- p is a 3D translation vector.

Two key properties make SE(2) and SE(3) so useful: composition (if T_1 maps from frame A to B and T_2 from B to C, then $T_2 T_1$ maps directly from A to C) and inverse (every transform has an inverse that undoes it; if T maps A to B, then T^{-1} maps B back to A).

This "closed under composition and inverse" behavior is why we call SE(2) and SE(3) *groups.* In practice, it justifies the very common pattern in robotics code: store each joint or link as a homogeneous transform, multiply them to move between frames, and invert them when you need to go backwards along the chain.

5.2 Forward Kinematics

The Denvevit-Hartenberg (DH) convention is a standard for assigning coordinate frames to robot manipulator links and joints to describe forward kinematics. The DH convention is not intuitive to the spatially challenged and those often toil with the ambiguity of the convention in edge cases. Fear not, the idea is simple: to encode individual joint properties into transformations such that we can chain them together and control the position of the end effector. The complete forward kinematics of the system can be derived using the following parameters.

The DH convention has four parameters for joint/link;

1. a_i Distance along x_{i-1} from z_{i-1} to z_i

2. α_i Twist angle around x_{i-1} from z_{i-1} to z_i

3. d_i Distance along z_{i-1} from x_{i-1} to x_i

4. θ_i Joint angle around z_{i-1} from x_{i-1} to x_i

The parameters are determined per joint in a robotic manipulator and are then used to construct a DH table containing the complete set of joint parameters in the manipulator.

i	a_i	α_i	d_i	θ_i
1	0.4	0	0	θ_1
2	0.3	0	0	θ_2
3	0.2	0	0	θ_3

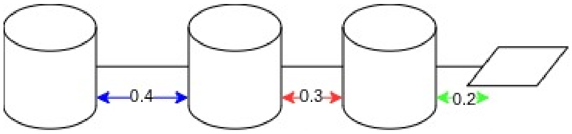

Figure 19: Denavit–Hartenberg parameter table for a simple 3-DOF planar arm with three revolute joints. Here a_i (highlighted along the second column) are the link lengths along the common normal (for a planar arm, they coincide with in-plane link lengths), α_i (third column) are the twist angles between successive z-axes (all zero for a planar arm), d_i (fourth column) are the offsets along each joint axis (zero for purely revolute, planar joints), and θ_i (fifth column) are the joint variables (the rotation angles about each z-axis)

The DH convention specifies rules for frame assignment:

1. Z_i axis is along the joint axis i

2. X_i axis is along the common normal between Z_{i-1} and Z_i
3. If joints are parallel choose X_i to minimize d_i
4. If joints intersect, X_i is perpendicular to the Z_{i-1} - Z_1 plane.
5. Use the right hand rule to assign Y_i
6. For revolute joints, θ_i is variable, and for prismatic joints, d_i is variable.

Determining the parameters seems straightforward enough, but it becomes difficult to keep track of manipulators with many joints. Additionally, this task can be made much more difficult with poorly assigned reference frames and while there is little choice in assigning reference frames, the few decisions that are made can have a large impact on the ease of assigning the parameters. Luckily, there are a few good tricks here that will help alleviate some headache.

The first step to extracting DH parameters requires drawing a diagram of the manipulator and drawing the first coordinate frame. Assign the coordinate frame following the rules above. Beginning at joint 0, first add the Z_0 axis. This axis should point align with the axis of rotation for a revolute joint or in the direction of linear motion for the prismatic joint. The assignment of the X_0 axis is assigned arbitrarily as we have two directions to choose from. However, the X_i axis should be chosen carefully as it will affect later frame assignments. If we fast forward and look at joint 1 and think about the rules for frame assignments, it should be considered that when X_{i-1} and Z_i are not parallel, X_i must point along the common perpendicular. The direction is chosen so that the parameter a_{i-1} is measured from Z_{i-1} to Z_1 along X_i and is positive or zero. If Z_{i-1} and Z_i are parallel or coincident the common perpendicular is not uniquely defined. In this case, the direction of X_i is arbitrary. Once the X_0 axis is assigned, the Y_0 axis can be assigned using the right hand rule. After all of the joint and links are accounted for in the DH table the end effector frame should be assigned. Unless some specific end-effector orientation is required, it is often easier to simply translate the previous frame to the end effector position, not doing any rotation.

Remember, that we are trying to encode the joint positions and orientations for the end-effector position in a series of transformation matrices. As such, the transformation between one frame to another is what is being captured, and the number of rows in the DH tables reflects that there are n, number of frames - 1 rows of parameters in the

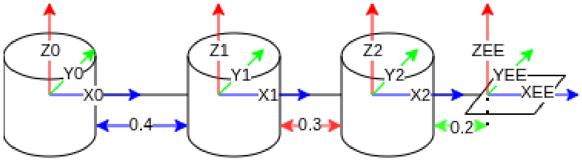

Figure 20: Example of a 3-link planar manipulator with reference frames.

table.

Once the DH parameters are extracted we can construct the homogeneous transformation matrix per row of parameters.

$$T_i^{i-1} = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} = \left[\begin{array}{ccc|c} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

where the left 3×3 block R is the rotation submatrix and the right 3×1 block p is the translation vector, with DH parameters a_i, α_i, d_i, θ_i in their standard positions. To find the complete transformation from T_0 to T_{EE}, all of the transformations from $T_{0,1}$ to $T_{N,EE}$ are multiplied together.

Example: 2-link planar arm forward kinematics

To see DH parameters in action, consider a simple 2-link planar arm lying in the x–y plane:

- Link 1 has length L_1 and joint angle θ_1 (revolute).
- Link 2 has length L_2 and joint angle θ_2 (revolute).
- Both joints rotate about the z-axis pointing out of the page.

Using the standard DH convention for this planar arm, one valid choice of parameters is:

i	a_i	α_i	d_i	θ_i
1	L_1	0	0	θ_1
2	L_2	0	0	θ_2

For each row we build the homogeneous transform ${}^{i-1}T_i$. With $\alpha_i = 0$ and $d_i = 0$, the standard DH matrix simplifies to a pure rotation plus translation along x:

$$T_i^{i-1} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_i \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

So for link 1 and link 2 we have:

$$T_0^1 = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & L_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad T_1^2 = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & L_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The complete transform from the base to the end-effector is:

$$T_0^2 = T_0^1 \, T_1^2.$$

Multiplying the rotation and translation parts, the end-effector position $(x_{\text{ee}}, y_{\text{ee}})$ in the base frame comes out to the familiar planar-arm formulas:

$$x_{\text{ee}} = L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2),$$

$$y_{\text{ee}} = L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2).$$

This example shows how the abstract DH table and homogeneous transforms reduce to concrete trigonometric expressions for a very common robot arm.

Special DH cases: parallel or coincident z-axes

The standard DH rules work perfectly when consecutive z-axes are skew or intersecting, but three common situations need extra care:

Case	What you see in the robot	How to handle it in DH
Parallel z-axes ($z_{i-1} \parallel z_i$, not coincident)	Two revolute joints with parallel axes	Common perpendicular is well-defined. Set a_{i-1} = distance between the axes. Choose x_i along that perpendicular
Coincident z-axes	Two revolute joints sharing the same axis	Common perpendicular undefined → set $a_{i-1} = 0$, $\alpha_{i-1} = 0$. Choose x_i arbitrarily.
Prismatic joint followed by revolute with parallel/co-incident z	Typical in cylindrical robots or many industrial arms	Same rule as parallel case: a_{i-1} is the offset along the common perpendicular.

Table 1: Special DH cases when z-axes are parallel or coincident.

These are the only situations where DH assignment feels ambiguous, but the recipe is simple: when the common perpendicular disappears, set the corresponding twist $\alpha = 0$ and link length $a = 0$ (or the actual offset) and pick the x-axis direction consistently.

5.3 Inverse Kinematics — Getting Joint Angles from a Desired Pose

Up to now we have only ever gone one direction:

> "Here are some joint angles → where is the end-effector?"

That was easy. Just chain a bunch of homogeneous transformations together and you're done.

Now we want the far more useful direction:

> "I want the gripper exactly here with this exact orientation → what joint angles do I need?"

Welcome to inverse kinematics. It is almost never as clean as forward kinematics.

Why inverse kinematics feels painful the first time

In practice, inverse kinematics is where most beginners lose weeks of their life. A normal 6-DOF arm has infinitely many solutions (elbow-up, elbow-down, wrist flipped, etc.), and sometimes zero solutions if the pose is outside the reachable workspace. Near singularities, the same tiny motion of the end-effector can demand enormous joint speeds, and real robots have joint limits so even mathematically perfect solutions can be illegal.

Example: numeric IK for a 2-link planar arm

Before tackling full 6-DOF arms, it helps to see inverse kinematics on the simple 2-link planar arm from the forward-kinematics example.

Assume:

- Link lengths $L_1 = 1.0\,\mathrm{m}$ and $L_2 = 0.7\,\mathrm{m}$.
- Desired end-effector position $x_{\mathrm{ee}} = 1.2\,\mathrm{m}$, $y_{\mathrm{ee}} = 0.5\,\mathrm{m}$.

From planar geometry we know:

$$x_{\mathrm{ee}} = L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2),$$

$$y_{\mathrm{ee}} = L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2).$$

We first solve for the elbow angle θ_2 using the law of cosines on the triangle from the shoulder to the wrist:

$$r^2 = x_{\mathrm{ee}}^2 + y_{\mathrm{ee}}^2, \qquad r = \sqrt{1.2^2 + 0.5^2} \approx 1.30 \text{ m},$$
$$\cos\theta_2 = \frac{r^2 - L_1^2 - L_2^2}{2L_1L_2}.$$

Plugging in the numbers:

$$\cos\theta_2 \approx \frac{1.30^2 - 1.0^2 - 0.7^2}{2 \cdot 1.0 \cdot 0.7} \approx \frac{1.69 - 1.00 - 0.49}{1.4} = \frac{0.20}{1.4} \approx 0.143.$$

So one solution for θ_2 is

$$\theta_2 = \arccos(0.143) \approx 1.43 \text{ rad} \approx 82^\circ.$$

(The other solution, with the elbow “flipped”, is $-\arccos(0.143)$.)

Next we solve for the shoulder angle θ_1. One common formula is

$$\theta_1 = \operatorname{atan2}(y_{\mathrm{ee}}, x_{\mathrm{ee}}) - \operatorname{atan2}\big(L_2 \sin\theta_2,\ L_1 + L_2 \cos\theta_2\big).$$

Using the numbers above and the "elbow-up" θ_2:

$$\operatorname{atan2}(y_{\text{ee}}, x_{\text{ee}}) = \operatorname{atan2}(0.5, 1.2) \approx 0.39 \text{ rad},$$

$$\operatorname{atan2}\big(L_2 \sin\theta_2,\ L_1 + L_2\cos\theta_2\big) \approx 0.56 \text{ rad}.$$

So

$$\theta_1 \approx 0.39 - 0.56 \approx -0.17 \text{ rad} \approx -10^\circ.$$

If you plug these angles back into the forward-kinematics formulas, you will recover an end-effector position very close to $(1.2, 0.5)$ up to rounding. This small example shows the same elbow-up / elbow-down ambiguity and trigonometric structure that appears, on a larger scale, in 6-DOF analytic IK.

The good news is almost every real robot you will ever touch makes it easy. Look at any modern 6-DOF industrial or collaborative arm (UR5, UR10, Franka Emika Panda, KUKA IIWA, Fanuc, etc.). You will notice something deliberate: the last three joint axes intersect at a single point.

This is called a spherical wrist and it is the greatest gift the mechanical designers ever gave us.

Because axes 4–6 meet at one point, the problem splits cleanly in two:

1. Joints 1–3 only have to position the wrist center (a pure 3D positioning problem).
2. Joints 4–6 only have to orient the end-effector around that fixed point (a pure orientation problem).

Suddenly inverse kinematics becomes something you can solve with high-school trigonometry. d

The pattern in every analytic IK for 6DOF arms

1. Subtract the last link length along the desired approach direction to find the wrist-center position:

$$\mathbf{p}_{\text{wc}} = \mathbf{p}_{\text{ee}} - d_6\, \mathbf{a}_z$$

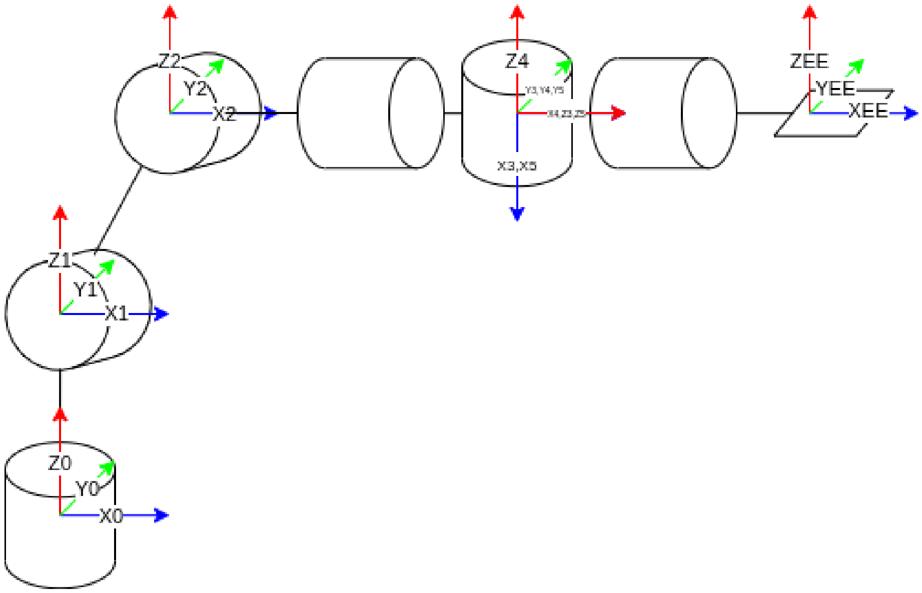

Figure 21: 6DOF manipulator with spherical wrist.

(where $\mathbf{a}_z$ is the third column of the desired rotation matrix).

2. Solve for joint 1 — it is just the base rotation:

$$\theta_1 = \text{atan2}(p_{\text{wc},y},\ p_{\text{wc},x})$$

   One `atan2`, done.

3. Treat joints 2 and 3 as a 2D two-link planar arm that has to reach from the shoulder to the wrist center.

   - Cosine law → elbow angle (joint 3).
   - One more `atan2` → shoulder angle (joint 2).
   - You automatically get two solutions: elbow-up and elbow-down.

4. Once joints 1–3 are known, compute the rotation matrix of frame 3 relative to the base.

5. The remaining orientation the wrist has to provide is

$$\mathbf{R}_6^3 = (\mathbf{R}_3^0)^\top \mathbf{R}_{\text{des}}$$

6. Extract joints 4–6 from $\mathbf{R}_6^3$ — again just a few `atan2` calls or a canned spherical-

wrist formula.

That's it. No Jacobian inversion, no numerical iteration, no mystery.

If you can solve inverse kinematics for a 2-link planar arm, you can solve it for every 6-DOF collaborative robot on the market.

Practical advice that will save you weeks

Do not derive the full 6-DOF analytic IK from scratch unless it's a homework assignment; the positioning part (joints 1–3) is the only interesting bit, while the wrist part is always the same handful of lines. Google “UR5 analytic inverse kinematics” or “Franka inverse kinematics github” and you will find dozens of battle-tested implementations that differ only in variable names. Always compute and return both elbow-up and elbow-down solutions, as randomly flipping between them makes your trajectories look drunk. Clamp every solution to joint limits immediately (never let the controller try to go to 370°), and test aggressively: generate 10,000 random valid poses, run forward → inverse → forward, and check that the error is small. If it isn't, you have a sign bug somewhere.

Singularity detection with the Jacobian

Even though analytic IK gives you perfect joint angles, the robot can still blow up in speed near two classic singularities:

1. Shoulder/elbow singularity — the arm is fully stretched or folded (joint 2 and joint 3 aligned).
2. Wrist singularity — joints 4 and 6 are aligned (the classic “wrist flip” degeneracy).

Near these configurations a tiny motion of the end-effector demands huge joint velocities. Your motors will scream and your torque limits will cry.

The cheapest, most reliable way to catch this is to look at the smallest singular value of the 6×6 geometric Jacobian at the current joint angles.

You do *not* need to derive the full analytical Jacobian (it's horrible). Just use a library:

```
J = robot.jacobian(q) # 6xn geometric Jacobian
min_sv = np.linalg.svd(J, compute_uv=False)[-1] # smallest s.v.

```

```
if min_sv < 0.015: # tune this once on your robot
    flag_as_singular() # slow down, replan, or switch to joint motion
```

In practice you do one (or more) of the following when the flag triggers: slow the Cartesian velocity command way down (0.1–0.2× normal speed), temporarily switch to pure joint-space motion, or let your motion planner avoid the region entirely.

What we are deliberately skipping for now

- 7-DOF and redundant arms → you switch to damped pseudo-inverse Jacobian control.
- Full analytical Jacobian derivation → not worth the pain.
- Advanced singularity-robust methods (SR-Inverse, damped least-squares, etc.) → later.

Bottom line: analytic IK gives you the angles, the Jacobian tells you whether those angles are safe to use at full speed. Never ship a 6-DOF arm without both.

Next — we finally have nice smooth joint trajectories that respect singularities. Now we have to make the real motors follow them without oscillating, overshooting, or catching fire. That's where all the control tricks from the line-follower and motor-speed chapters come back in full force.

Coding patterns for transforms and kinematics

In real code, it helps to give geometric objects first-class representations instead of constantly passing raw matrices around. A few common patterns include representing a transform as a small struct or class with fields `R` (rotation) and `p` (translation) and helper functions like `compose`, `inverse`, and `apply`. Storing the robot's kinematic chain as an ordered list of such transforms makes forward kinematics a simple loop of compositions. For DH-based arms, keeping both the DH table and a function that turns one row of parameters into a transform makes it easy to regenerate all link transforms when joint variables change.

These small abstractions make it much easier to reuse the same geometric ideas across mobile robots, manipulators, and perception pipelines without getting lost in index gymnastics.

6 Path Planning and Motion Planning

6.1 Why geometry shows up in path planning

When you ask a robot to "go from here to there without crashing into anything," you are really asking it to solve a geometric puzzle. The robot has a shape (a point, a circle, a rectangle, a full arm with joints), the world has obstacles (walls, table legs, other robots, people), and the robot is only allowed to move in ways that respect its constraints (joint limits, maximum turning rate, non-slipping wheels, safety margins).

Path planning is about turning this physical setup into a mathematical representation that we can search. We want to find a path (or trajectory) from a start configuration to a goal configuration, such that the robot does not collide with anything along the way, and preferably does so in a way that is "good enough" by some measure (short, smooth, fast, or energy-efficient).

From this point of view, geometry is not decoration – it is the language we use to describe where the robot is, what is allowed, and what is forbidden.

6.2 Configuration space: one point for each possible robot pose

In everyday thinking, we picture robots moving around in 2D or 3D space: a small differential-drive robot on the floor has a position (x, y) and a heading angle θ, a simple robotic arm might have two joint angles (q_1, q_2), and a drone might have position, orientation, and sometimes extra internal variables.

For path planning, it is often more convenient to move to an abstract space called configuration space (often shortened to C-space). A configuration is one complete description of the robot's pose (for example, all its joint angles and base position), and the configuration space is the set of all configurations the robot could possibly take on.

Examples:

- A point robot moving in the plane has configuration (x, y). Its configuration space is the usual 2D plane $\mathbb{R}^2$.
- A rigid robot moving in the plane with position and heading has configuration

(x, y, θ). Its configuration space is 3-dimensional, often written $\mathbb{R}^2 \times S^1$ ($\mathbb{R}^2$ is the 2D plane, each point identified with (x, y) and S^1 is a (unit) circle, each point identified with θ).

- A planar 2-link arm has configuration (q_1, q_2) (two joint angles). Its configuration space is a 2D surface that wraps around like a torus (because angles repeat every 2π).

The key idea is:

- Each configuration is a single point in C-space.
- A motion of the robot is a continuous curve through C-space.

Obstacles also become regions in C-space:

- Every obstacle in the world corresponds to a set of configurations where the robot would collide with it:

$$C_{\text{obs}} = \{\, q \in C \mid \text{robot at } q \text{ is in collision} \,\}.$$

- The remaining configurations are called C-free (free space):

$$C_{\text{free}} = C \setminus C_{\text{obs}}.$$

The path planning problem now becomes:

> Find a continuous curve $\gamma : [0, 1] \to C_{\text{free}}$ from the start configuration to the goal configuration, with $\gamma(0) = q_{\text{start}}$ and $\gamma(1) = q_{\text{goal}}$.

This is a purely geometric question about curves and regions in an abstract space.

6.3 From the real world to a search problem

Once we think in configuration space, planning a path looks a lot like navigating a maze: the maze is C-free (the set of safe configurations), the walls are C-obs (configurations that would cause collisions), and the start and goal are two specific points in the maze.

In principle, we could explicitly build a geometric model of C-obs and C-free, then run a graph search algorithm like breadth-first search or A* to find a path. In practice, fully building C-space is often too hard because the space can have many dimensions, exact collision boundaries can be complicated curved shapes, and we may only have an approximate map of the world.

Instead, planners usually combine a collision checker (given a configuration, tell me if it is free or in collision), a way to generate neighboring configurations (small moves the robot could attempt), and a search strategy (which part of C-space to explore first). Different planning algorithms mostly differ in how they choose which configurations to try and how they connect them.

6.4 Grid-based planning: discretize and search

One of the simplest and most intuitive approaches is to approximate C-space by a grid. Imagine a small robot constrained to a flat lab floor with known obstacles. The world is represented as an occupancy grid (a 2D array where each cell is free or occupied) and the robot's configuration is approximated by the cell it occupies.

We can now treat each free grid cell as a node in a graph, connect neighboring cells (up, down, left, right, and maybe diagonals), and use A* or a similar search algorithm to find a path of cells from start to goal that minimizes a discrete cost.

$$J_{\text{grid}} = \sum_{k=0}^{N-1} c\big(s_k, s_{k+1}\big),$$

where s_k are grid cells on the path and c is the step cost (often 1 for orthogonal moves and $\sqrt{2}$ for diagonals).

This has some nice properties:

- It is conceptually simple and easy to visualize.
- A* can incorporate a cost for each step (for example, prefer shorter paths or prefer staying away from walls).

But it also has limitations:

- Resolution: a coarse grid might miss narrow passages; a fine grid explodes in size.
- Kinematics: many real robots cannot move arbitrarily from one cell to any neighbor (think of cars that cannot slide sideways).

- Higher-dimensional C-spaces (adding heading, joints, etc.) lead to enormous grids.

Even so, grid-based planning remains a very common teaching example and is used in many practical navigation systems with moderate environments.

A* search in pseudocode

On a grid, A* is a practical way to find a lowest-cost path from a start cell to a goal cell. At a high level it keeps track of:

- $g(s)$: cost to reach a cell s from the start.
- $h(s)$: heuristic estimate of remaining cost to the goal (for example, Euclidean distance).
- $f(s) = g(s) + h(s)$: optimistic total cost through s.

A simple version of A* in pseudocode looks like:

```
open_set = {start} # frontier to explore, typically a priority queue
came_from = {} # map: cell -> predecessor on best path so far
g[start] = 0.0

while open_set is not empty:
    s = cell in open_set with smallest f(s) = g[s] + h[s]

    if s == goal:
        return reconstruct_path(came_from, s)

    remove s from open_set

    for each neighbor n of s:
        if n is an obstacle: continue

        tentative_g = g[s] + cost(s, n)

        if n not visited before or tentative_g < g[n]:
            came_from[n] = s
            g[n] = tentative_g
            add or update n in open_set
```

If $h(\cdot)$ is *admissible* (never overestimates the true remaining cost) and *consistent*, A* is guaranteed to find an optimal path on the grid.

Tiny 5x5 example

Consider a 5×5 grid where $(0, 0)$ is the top-left and $(4, 4)$ is the bottom-right. Let the start be at $(0, 0)$ and the goal at $(4, 4)$. Suppose the middle column $(2, 1)$, $(2, 2)$, $(2, 3)$ is blocked by obstacles.

If we restrict moves to the four cardinal neighbors and use unit step cost plus Euclidean-distance heuristic:

- A* will first expand nodes close to the diagonal toward the goal.
- When it reaches the blocked column, it will "flow" around the obstacles either above or below.
- The final path is a shortest 8-step route that detours around the column and then comes back toward $(4, 4)$.

Walking through the algorithm by hand on such a small grid is an excellent way to develop intuition for how the open set, g-costs, and heuristic interact.

6.5 Sampling-based planning: explore C-space without a full grid

When configuration spaces get bigger or more complicated, explicitly building a full grid becomes impractical. Sampling-based planners take a different approach:

- They randomly sample configurations in C-space.
- They keep the ones that are in C-free (collision-free).
- They try to connect these samples with short, simple motions.

Over time, the samples and their connections form a roadmap of the free space that the planner can search.

Two widely used sampling-based methods are:

- Probabilistic Roadmaps (PRM):
- Sample many configurations in C-free (for example, uniformly in a bounding box or according to some bias).
- Try to connect nearby ones with straight-line paths in C-space, often using a metric like the Euclidean distance

$$d(q_i, q_j) = \|q_j - q_i\|,$$

and keeping edges only if those straight-line segments remain in C_{free}.

- Once the roadmap is built, connect start and goal to it and run a graph search over the roadmap.
- Good for multi-query settings where you plan many paths in the same environment.
- Rapidly-exploring Random Trees (RRT):
- Start a tree at the start configuration q_{start}.
- Repeatedly sample a random configuration q_{rand} in C-space.
- Find the nearest node q_{near} in the tree (for example, minimizing $d(q_{\text{near}}, q_{\text{rand}})$).
- Compute a new configuration q_{new} by moving a limited step δ toward q_{rand}:

$$q_{\text{new}} = q_{\text{near}} + \delta \, \frac{q_{\text{rand}} - q_{\text{near}}}{\|q_{\text{rand}} - q_{\text{near}}\|}.$$

- If the new edge is collision-free, add q_{new} to the tree.
- Stop when the tree reaches the goal region.
- Good for single-query problems with complex dynamics or high dimensions.

These methods have a few shared strengths: they do not need a full, explicit representation of C-space, they can handle high-dimensional robots and complex obstacle shapes, and given more time, they tend to find better and better paths (or at least more coverage of C-free).

Their main cost is that each sample and edge requires a collision check, which can be expensive. Also, paths are often somewhat jagged and may need post-processing (smoothing, shortcutting) before execution.

Basic RRT pseudocode

A minimal RRT in pseudocode looks like:

```
tree = { start }

for k in range(max_iterations):
    q_rand = sample_random_config()
    q_near = nearest_neighbor(tree, q_rand)
    q_new = steer(q_near, q_rand, step_size)

    if edge_is_collision_free(q_near, q_new):
        add q_new to tree with parent q_near

        if q_new is in the goal region:
            return extract_path(tree, q_new)
```

The key ingredients are:

- Sampling: `sample_random_config()` explores new parts of C-space.
- Nearest neighbor: we always grow the tree from the closest known configuration.
- Steering: `steer` limits how far we move in one step, which helps respect dynamics.
- Collision checking: we reject edges that intersect obstacles.

RRT is *probabilistically complete*: if a solution exists and we keep sampling forever, the probability that we eventually find a path goes to 1.

6.6 Kinematic and dynamic constraints: not every path is drivable

So far, we have mostly treated the robot as if it could move along any curve in C-free. Real robots have additional kinematic and dynamic constraints. A car-like robot cannot move sideways; it must drive forward or backward with limited steering angle. A differential-drive robot has bounded wheel speeds and can only turn so fast. A robotic arm has joint limits, maximum joint velocities, and sometimes torque limits.

These constraints change which paths are actually feasible. A curve that looks fine in C-space might require instantaneous turns or speeds beyond what the robot can produce, or seemingly short paths may involve sharp, jerky motions that are unsafe for the hardware.

Motion planning is often described in two layers:

- Path planning: find a collision-free geometric path in C-free.
- Trajectory planning: turn that path into a time-parameterized motion that respects speed, acceleration, and sometimes force limits.

In practice, planners may:

- Directly build constraints into the planning process (for example, RRT variants that extend using only feasible motions).
- Or first find a coarse geometric path, then refine it into a feasible trajectory using separate timing and smoothing steps.

6.7 Simple cost functions: what makes one path "better" than another?

Once multiple feasible paths exist, we need a way to choose between them. This is where cost functions come in.

Common examples:

- Path length: shorter is better. For a continuous path $\gamma(t)$ in C-space, one simple length cost is
$$J_{\text{len}}(\gamma) = \int_0^1 \big\|\dot{\gamma}(t)\big\| \, \mathrm{d}t.$$
- Time to reach goal: faster is better, subject to speed limits. If a trajectory is parameterized over time $t \in [0, T]$, the simplest time cost is just
$$J_{\text{time}} = T.$$
- Clearance: prefer paths that stay farther away from obstacles.
- Energy use: penalize large accelerations, high speeds, or steep climbs.

In search-based planners like A*, cost appears in:

- The g-cost: cost accumulated so far along the path,

$$g(s_k) = \sum_{i=0}^{k-1} c\big(s_i, s_{i+1}\big).$$

- The h-cost (heuristic): optimistic guess of the remaining cost to the goal, often something simple like Euclidean distance

$$h(s) = \big\|x_{\text{goal}} - x(s)\big\|.$$

In sampling-based planners, cost may be used to:

- Bias tree growth toward lower-cost regions.
- Select which edges to keep or prune.
- Post-process a found path (for example, "shortcut" it by removing unnecessary detours).

There is rarely a single "right" cost function. Instead, you choose one that matches your task:

- A delivery robot in a crowd might care about safety margins and smoothness more than pure shortest distance.
- A competition robot with a time limit might prioritize fastest completion even if the path passes close to obstacles.

6.8 From planned path to executed motion

A planned path is only a suggestion. The real world will not exactly match your map: obstacles may move (people, other robots), sensors may be noisy or delayed, and the robot may slip, drift, or be pushed.

To actually execute a path, we almost always need a local controller (like pure pursuit or PID) that tracks the desired path and corrects deviations, and a way to update the plan when the world changes enough that the old path is no longer safe or reasonable.

This leads to a common architecture:

1. A global planner computes a high-level path through the environment (for example, using a grid or PRM). 2. A local planner/controller follows this path while reacting to

short-term changes (small obstacles, minor map errors). 3. If the path becomes blocked or clearly suboptimal, the global planner is asked to replan.

Costmaps and obstacle inflation

In many mobile-robot systems, the global planner does not work directly on a raw occupancy grid. Instead it uses a costmap. We start from an occupancy grid, inflate obstacles by a radius related to the robot size and a safety margin (so paths naturally stay away from walls), and assign higher costs to cells near obstacles.

When A* or another search algorithm runs on this costmap, the planner prefers short paths *and* paths through low-cost (safer) regions, providing a built-in buffer against small map errors and localization drift.

Global and local planner stack

A simple but effective navigation stack for a differential-drive robot might look like:

- Global planner (slow, e.g. 1–2 Hz):
  - Runs A* or a sampling-based method on a coarse costmap.
  - Produces a sequence of waypoints or a coarse path from start to goal.

  Local planner/controller (fast, e.g. 10–50 Hz):
  - Looks at a short window of the global path ahead of the robot.
  - Uses a path-following controller (for example, pure pursuit or a lateral-error PID) to generate velocity commands.
  - Incorporates very recent obstacle information from nearby sensors to avoid sudden collisions.

Replanning is typically triggered when new obstacles are detected that intersect or closely approach the current global path, the robot deviates too far from the planned path (for example, due to slip or being pushed), or the costmap changes significantly (doors opening or closing, moved furniture).

In this way, geometry (configuration space, obstacles, constraints) and control (feedback, filtering, state estimation) work together: geometry gives the robot a map of the possibilities, and control makes the robot stick to a chosen path despite noise and disturbances.

6.9 Takeaways

- Path and motion planning turn “move safely from here to there” into a geometric problem in configuration space.
- Obstacles become forbidden regions; safe motions are continuous curves through the free space.
- Different planners (grid-based, sampling-based, constraint-aware) offer different tradeoffs between simplicity, efficiency, and realism.
- Planning and control are tightly coupled: a good plan is easier to track, and good feedback control makes planning more robust to the messy reality of robots in the world.

7 Localization and Mapping

Robots cannot follow a planned path or execute a clever controller unless they know *where they are* and have at least a rough idea of *what the world around them looks like.*

In earlier chapters we saw:

- In the filters chapter, how to turn noisy sensor readings into smoother, more believable signals using prediction and correction.
- In the line follower chapter, how even a small estimate of state (like lateral error to a line) lets us build much better controllers.
- In the path-planning chapter, how to find paths through a geometric world once we have a map.

This chapter ties those ideas together for mobile robots:

- Odometry: how a robot can estimate its own motion just from its wheel encoders, and why this always drifts over time.
- Maps: simple ways to represent where obstacles are, especially 2D occupancy grids.
- Localization: how to combine motion and sensor information to keep track of pose in a known map.
- SLAM: a high-level view of building a map while simultaneously localizing within it.

We will keep the math light and focus on pictures, intuition, and a few small update rules; think of this chapter as a bridge between the filters and path-planning chapters.

7.1 Odometry and dead reckoning

Imagine a small differential-drive robot like the one in the line follower chapter: two drive wheels separated by distance L, wheel encoders that report speed, and no external reference like GPS or cameras. From the encoders we can estimate the linear velocity of each wheel (v_l, v_r) and combine them to find the robot's forward speed v and angular speed ω.

As in the line follower chapter, the kinematics say:

$$v = \frac{v_r + v_l}{2}, \qquad \omega = \frac{v_r - v_l}{L}.$$

Updating pose from velocity

We describe the robot's pose in the plane by:

- x: position along some horizontal axis.
- y: position along some vertical axis.
- θ: heading angle (orientation) of the robot, measured from the x-axis.

If at some instant the robot is at (x, y, θ) and moving with forward speed v and angular speed ω, then over a short time step Δt we can approximate:

$$x \leftarrow x + v \cos\theta \, \Delta t,$$

$$y \leftarrow y + v \sin\theta \, \Delta t,$$

$$\theta \leftarrow \theta + \omega \, \Delta t.$$

In code-like pseudocode:

```
x = x + v * cos(theta) * dt
y = y + v * sin(theta) * dt
theta = theta + omega * dt
theta = wrap_angle(theta) # keep in [-pi, pi], for example
```

If we repeat this update every control cycle, we get an estimate of how the robot has moved over time using only the encoders. This process is called odometry or dead reckoning.

Why odometry alone always drifts

Odometry is extremely useful because it works indoors, has no external dependencies, and is often quite accurate over short distances. However, there are many small sources of error: wheel slip, slightly different wheel radii, timing jitter, or uneven floors.

Each step of the odometry update is just a tiny bit wrong, but those tiny errors accumulate. Position error grows with distance traveled, and heading error grows especially when the robot turns. After a long trip, the odometry estimate may say the robot has returned to its starting point even though it is physically tens of centimeters away.

If you have ever drawn a treasure map by walking around your house, counting steps, and recording turns, you have experienced odometry drift: by the time you finish the loop, your hand-drawn starting point does not exactly line up with the real one.

Takeaway: Odometry is a great short-term motion estimate, but by itself it cannot tell you where you are in a building after minutes of driving. We need something that can occasionally "pull" our estimate back toward reality.

7.2 Maps and occupancy grids

To correct odometry and plan paths, the robot needs some representation of its environment — a map.

There are many kinds of maps (point clouds, polygonal maps, semantic maps), but for indoor ground robots a very simple and popular choice is the occupancy grid.

What is an occupancy grid?

An occupancy grid divides the world into a 2D grid of small square cells. Each cell represents a tiny patch of floor and stores a score indicating how likely it is to be occupied vs free. Conceptually, high-occupancy cells correspond to walls and furniture, low-occupancy cells are safe floor, and unobserved cells start in an "unknown" middle state. This matches the grid maps we used in path planning, where A* treats free cells as traversable nodes.

Updating the grid with range sensors

How does the robot build or refine such a grid?

Imagine the robot has a range sensor (like a 2D LiDAR) and an odometry-based pose estimate. Each time the sensor takes a measurement, we cast rays outwards from the robot's pose. Cells along the ray (up to the measured distance) are likely free, while the cell at the hit point is likely occupied. The map updates its belief about each touched cell, nudging the occupancy score up or down.

Over time, walls and furniture show up as bands of highly occupied cells, doorways appear as gaps, and open floor stays free. We do not need the exact updating formulas to understand the idea; the important part is that each measurement slightly adjusts nearby cells rather than overwriting the whole map.

A simple log-odds update rule

One common way to maintain occupancy probabilities is to work with log-odds instead of raw probabilities. For each cell we keep a value

$$\ell = \log \frac{p}{1-p},$$

where p is the probability that the cell is occupied. Large positive ℓ means "very likely occupied", large negative ℓ means "very likely free".

When a new measurement suggests a cell is *occupied*, we add a small positive constant L_{occ} to ℓ. When a beam passes through a cell (evidence for *free*), we add a small negative constant L_{free} instead:

$$\ell \leftarrow \ell + L_{\text{occ}} \quad \text{or} \quad \ell \leftarrow \ell + L_{\text{free}}.$$

Over time, cells that are repeatedly hit by beams accumulate large positive ℓ (very likely obstacles), cells that are frequently traversed accumulate large negative ℓ (very likely free), and mixed evidence keeps ℓ near zero (unknown). This log-odds trick turns "multiply probabilities" into simple additions, which is numerically stable and easy to implement.

7.3 Localization with a known map

Suppose we have a reasonably accurate map, odometry estimates, and sensor data. Our goal is to track the robot's pose (x, y, θ) *within that map.* We know odometry alone will drift, and the sensor alone is often ambiguous (a single scan might look the same in two different corridors). The solution is to combine them in a predict–correct loop, just like in the filters chapter.

A 1D Bayes-filter picture

To keep the math simple, imagine the robot can only move along a line in a corridor. Its position along that line is x, and we maintain a belief $p(x)$ over where it might be.

Each time step we:

- Predict how the belief moves when the robot drives forward.
- Correct the belief using a sensor reading that depends on x and the map.

In probability notation this looks like:

- Prediction with control u (for example, commanded forward motion):

$$p(x_t \mid u_t) = \sum_{x_{t-1}} p(x_t \mid x_{t-1}, u_t)\, p(x_{t-1}), \tag{1}$$

  which says: "to get the new belief at x_t, consider all previous positions x_{t-1} and how likely it is to move from there to here." In continuous space, the sum becomes an integral.

- Correction with measurement z_t:

$$p(x_t \mid z_t) \propto p(z_t \mid x_t)\, p(x_t \mid u_t), \tag{2}$$

  which says: "poses that make the measurement z_t likely get their probability boosted; others are down-weighted."

We do not need to carry these formulas into code line by line, but they capture the same story as before:

> Prediction moves and spreads the belief according to the motion model; correction reshapes it according to how well each pose explains the sensor data in the map.

Belief over pose

Instead of claiming "the robot is exactly here", we maintain a fuzzier notion:

- A belief over possible poses.

- In a coarse grid this might be a score for each cell and heading bin: how likely it is that the robot is in that cell facing that direction.
- In a continuous representation it might be a mean pose with uncertainty ellipses around it.

You can picture this belief as a blurry dot on the map:

- A small, sharp dot means we are very sure about where we are.
- A smeared-out cloud means we are less certain and must be more careful when planning.

Predict: move the belief with odometry

When the robot moves forward and turns a little, the whole belief distribution should shift:

- If the robot commands "drive straight 0.2 m", the cloud of possible poses should move forward in the map by roughly 0.2 m.
- If the robot turns slightly, the cloud rotates accordingly.
- At the same time, we *widen* the cloud a bit to reflect the uncertainty added by odometry drift.

This is the prediction step: we let odometry push our belief forward in time.

Correct: compare expected and actual sensor readings

Next we ask: *If the robot were actually at each of these candidate poses, what would its sensor readings look like in this map?*

- For each pose in our belief (or for many random samples drawn from it), we imagine casting synthetic sensor rays into the map.
- We compare these predicted ranges to the real sensor measurements.
- Poses whose predictions agree with the real readings become more likely; those that disagree become less likely.

This is the correction step:

- The map acts as a reference.
- Sensor readings tell us which parts of the map look like where we currently are.
- Our belief shifts and tightens around consistent poses.

The exact math (Bayes' rule, probabilities, and normalization) is important for implementation, but the core story is simple:

> Use odometry to guess where you went; use the map and sensors to gently pull that guess back toward places that actually match what you see.

Connection to filters on 1D signals

This is conceptually the same as the 1D predict–correct filter from the filters chapter:

- There we estimated a *distance to a wall* using a motion model and noisy range measurements.
- Here we estimate a *2D pose* using a motion model (odometry) and noisy scans.
- In both cases:
  - Prediction uses knowledge of how the world tends to change.
  - Correction uses sensors to keep the estimate anchored to reality.

7.4 Mapping and localization together (SLAM)

So far we assumed the map is known in advance. But in many real applications—like a search-and-rescue robot entering a collapsed building—the map is not known. The robot must build a map *and* localize itself within that evolving map at the same time. This is the simultaneous localization and mapping (SLAM) problem.

The core SLAM loop

At a high level, most SLAM systems repeat a familiar cycle: predict pose forward using odometry, incorporate new sensor data to update both the map and the pose estimate, and refine the map when loops are detected (loop closure). You can think of SLAM as running a giant predict–correct filter where the state includes both the robot trajectory and the layout of the environment.

Modern SLAM systems (2D LiDAR SLAM, visual-inertial SLAM, graph-based SLAM) use more advanced math and optimization, but the story remains the same.

Kalman filters vs. particle filters (very brief)

There are many concrete algorithms that implement the Bayes-filter idea. Two important families are:

- Kalman filters: assume state and noise are well modeled by Gaussians, so the belief can be summarized by a mean μ and covariance Σ. In a linear system with Gaussian noise the classic Kalman filter equations are:

$$\mu_{t|t-1} = A\mu_{t-1} + Bu_t$$

$$\Sigma_{t|t-1} = A\Sigma_{t-1}A^\top + Q$$

$$K_t = \Sigma_{t|t-1}C^\top(C\Sigma_{t|t-1}C^\top + R)^{-1}$$

$$\mu_t = \mu_{t|t-1} + K_t(z_t - C\mu_{t|t-1})$$

$$\Sigma_t = (I - K_tC)\Sigma_{t|t-1}.$$

- Extended and unscented Kalman filters adapt these ideas to mildly nonlinear systems.
- Particle filters: represent the belief by a cloud of samples $\{x_t^{(i)}\}$ with weights $w_t^{(i)}$. Each step:

  1. Sample new particles from the motion model.
  2. Weight them by the likelihood of the measurement $p(z_t \mid x_t^{(i)})$.
  3. Resample to focus on high-weight particles.

Kalman-style methods are efficient when the state is small and the world is close to linear-Gaussian. Particle filters handle multi-modal beliefs and strong nonlinearities, at the price of more computation.

Practical examples

Some practical examples of SLAM-like behavior:

- A low-cost vacuum robot gradually learns which areas of the floor are open and which are blocked, while keeping track of its approximate location.
- A drone with a camera and IMU flies indoors, fusing inertial predictions with visual landmarks to avoid walls and maintain a stable trajectory.
- A research robot builds a 2D occupancy grid from a spinning LiDAR while simultaneously estimating its pose on that grid.

7.5 Practical considerations and common failure modes

Localization and mapping in the real world come with many pitfalls. Here are a few important ones:

Sensor limitations (limited field of view, shiny surfaces, noise) mean we often fuse multiple sensors. The "kidnapped robot" problem—where a robot is moved without knowing it—requires filters that can recover from being totally lost, often by relocalizing from scratch. Ambiguous environments like long corridors can confuse localization, leading designers to add artificial landmarks. Finally, bad odometry calibration systematically biases predictions, making mapping much harder.

Regular calibration and simple checks (drive a square or a straight line and measure the drift) go a long way toward making localization and SLAM more reliable.

7.6 Implementation sketch and tuning tips

To tie the pieces together, here is a minimal loop for odometry plus occupancy-grid mapping, ignoring many engineering details but showing where each idea fits:

```
while robot_is_running():
    # 1. Read wheel encoders and update pose estimate (odometry)
    v_l, v_r = read_wheel_speeds()
    v = 0.5 * (v_r + v_l)
    omega = (v_r - v_l) / L

    x += v * cos(theta) * dt
    y += v * sin(theta) * dt
    theta = wrap_angle(theta + omega * dt)

    # 2. Read range sensor and update occupancy grid in a local window
    scan = read_range_sensor()
    for each beam in scan:
        mark_cells_along_beam_free(grid, x, y, theta, beam)
        mark_end_cell_occupied(grid, x, y, theta, beam)

    # 3. (Optional) Run a localization update using the map and scan
    # e.g., update belief over pose with a particle filter or EKF
    update_pose_belief_from_map_and_scan()

    # 4. Use current pose + map for planning and control
    follow_planned_path_with_controller()
```

Some light-touch tuning advice:

- Grid resolution: start with cells roughly the size of the robot's footprint or slightly smaller; finer grids give prettier maps but cost more memory and CPU.
- Log-odds increments: choose modest L_{occ} and L_{free} so that a few consistent readings can override occasional outliers, but beliefs do not saturate instantly.
- Beam subsampling: on high-rate lidars, you can often use every second or fifth beam without losing the big picture, reducing compute load.
- Pose vs. map trust: if odometry is poor, let the localization module move the pose more aggressively; if the map is uncertain, be more conservative with pose corrections.

7.7 Summary and connections

In this chapter we:

- Saw how odometry and dead reckoning can estimate a robot's motion from wheel encoders, but inevitably drift over time.
- Introduced occupancy grids as a simple, powerful way to represent maps for path planning and collision checking.
- Described localization in a known map as a predict–correct process that combines

odometry with sensor readings.

- Outlined SLAM as the joint problem of building a map and localizing within it, using the same ideas at a larger scale.

Taken together with the filters, line follower, and path-planning chapters, this gives a coherent picture: filters smooth noisy signals, localization and mapping apply those ideas to pose and environment, path planning uses the map to find safe routes, and control makes the motors follow those routes.

Whenever you think about where a robot is and how it moves through a space, keep this loop in mind: estimate state → plan a path → execute with feedback → update the estimate. Localization and mapping are the essential glue between sensing and motion in that loop.

References

[1] Peter Corke. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*. Springer, Cham, 2nd edition, 2017.

[2] John J. Craig. *Introduction to Robotics: Mechanics and Control*. Pearson, Boston, 4th edition, 2018.

[3] Kevin M. Lynch and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, Cambridge, 2017.

[4] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. Springer, Cham, 2nd edition, 2016.

[5] Mark W. Spong, Seth Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. Wiley, Hoboken, NJ, 2006.

[6] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, Cambridge, MA, 2005.

List of Figures