Just Enough Robotics

Nathan Sprague

Contents

Preface 5

Acknowledgments 7

1 Controlling Physical Systems 9

1.1 Introduction 9

1.2 Open Loop Control 9

1.3 Closed Loop Control 10

1.3.1 Proportional Control 11

1.3.2 Adding a Derivative Term 12

1.3.3 Adding an Integral Term 15

1.4 Proportional Robot Control 17

1.5 Limitations of PID Controllers 20

1.6 References and Further Reading 21

1.6.1 Closed-Loop Controllers 21

1.6.2 General Robotics Resources 21

2 Coordinate Frames 23

2.1 Introduction 23

2.2 Conventions 24

2.3 Poses and Coordinate Frames 25

2.3.1 Translations 26

2.3.2 Euler Angles . . . 27
2.3.3 Axis Angle . . . 29
2.3.4 Quaternions . . . 29
2.3.5 Rotation Matrices . . . 30
2.4 References and Further Reading 31

3 Path Planning . . . 33
3.1 Configuration Spaces 33
3.1.1 The Path Planning Problem . . . 35
3.1.2 Constraints . . . 35
3.2 Planning Algorithms 38
3.3 Grid-Based Search 39
3.3.1 Returning A Path . . . 42
3.3.2 Dijkstra's Algorithm . . . 45
3.3.3 A* . . . 47
3.3.4 The Efficiency of Grid-Based Search . . . 49
3.4 Sampling-Based Search 50
3.4.1 Rapidly Exploring Random Trees . . . 51
3.4.2 Probabilistic Roadmaps . . . 54
3.5 References and Further Reading 56

4 Probabilistic Localization . . . 57
4.1 Introduction 57
4.2 Discrete Probability Distributions 58
4.2.1 Joint Probabilities . . . 58
4.2.2 Conditional Probabilities . . . 61
4.2.3 Bayes' Rule . . . 63
4.3 Recursive State Estimation 66
4.3.1 Efficiency Considerations . . . 68
4.4 Kalman Filter 68
4.4.1 Linear State Dynamics . . . 69
4.4.2 Linear Sensor Model . . . 70
4.4.3 Kalman Filter Algorithm . . . 70
4.5 References and Further Reading 72

Bibliography . . . 73

Preface

This book is intended to be an accessible introduction to key algorithmic concepts in autonomous robotics targeted toward undergraduate computer science students. It is not intended to be an exhaustive reference or to provide rigorous mathematical foundations, derivations or proofs. The focus is on clear explanations with extensive examples and visualizations.

This book uses a Python-based pseudocode that is generally very close to executable Python. I have attempted to steer clear of any idiomatic Python constructs so that the algorithm listings should be easy to follow even for those without a background in the Python language.

This book is a work in progress! It covers several of the central algorithmic ideas in autonomous robotics, but there are still significant missing pieces. In particular, there is currently no coverage of particle filters, mapping algorithms, or simultaneous localization and mapping (SLAM). Contributions are welcome!

Acknowledgments

The development of this book was supported by a grant from the Virtual Library of Virginia (VIVA).

I would also like to thank Dr. Kevin Molloy for providing feedback on portions of this book and helping me to pilot it in multiple semesters of CS 354 At James Madison University.

1. Controlling Physical Systems

1.1 Introduction

Computer programmers are accustomed to instructions that operate *instantaneously* and *reliably*. Both assumptions fail when writing code that controls a physical system. In this chapter we will take the first steps towards writing programs that can reliably control physical systems in an unpredictable world.

1.2 Open Loop Control

Consider the problem of programming a controller for the self-driving locomotive in Figure 1.1. In this figure x indicates the starting position of the locomotive and g indicates the goal location.

Our first attempt at developing a controller might look something like the following:

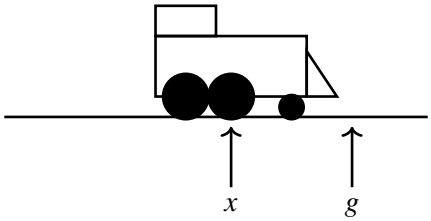


Figure 1.1: A self-driving locomotive.

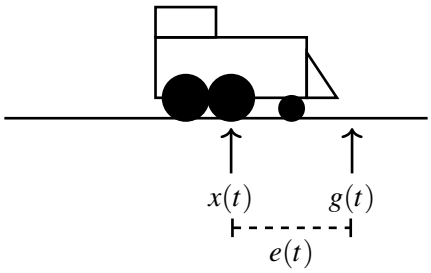


Figure 1.2: A self-driving locomotive.

Listing 1.1: Open Loop Control Algorithm

```
def open_loop(x, g):

    # Calculate the distance to travel.
    d = g - x

    # Cover that distance in one second.
    for one second:
        drive forward at a speed of d/second
```

Algorithm 1.1 is an example of an **open loop controller**. Open-loop control involves sending a sequence of control signals that, based on our understanding of the system we are controlling, should move the system into the target configuration.

There are problems with Algorithm 1.1. First, locomotives are *heavy*. The success of this algorithm relies on the unrealistic assumption that we can instantaneously changing the speed from zero to the desired value of d/s. Second, even after the locomotive reaches the target speed, factors like friction and mechanical imperfections will make it impossible to perfectly maintain that speed. Over time, small errors in speed will result in significant errors in the final position.

1.3 Closed Loop Control

The problems with the naive controller in Algorithm 1.1 can be avoided by using **closed loop control**. Closed loop controllers continuously monitor the current error in the system and update the control signal to push the error toward zero. This approach tends to be more reliable because the controller responds to the actual state of the system and is able to make adjustments when the system fails to behave as expected.

The PID (**P**roportional, **I**nverse, **D**erivative) controller is the classic example of closed loop control. The next several sections will introduce the PID controller by describing each of these three terms. The following table outlines the notation that will be used (Figure 1.2 also uses this notation).

$x(t)$ The state of the system at time t. In the case of the locomotive this is the position on the track. More generally, this could describe any state variable that we are interested in controlling. This could be the temperature of a room or the altitude of a rocket.
For now, we will assume that state information is provided by a reliable sensor. In future chapters we will consider the problem of estimating this value when sensors are absent or unreliable.

$g(t)$ Goal state at time t.

$e(t)$ Error at time t.
We will let $e(t) = g(t) - x(t)$. In the case of the locomotive, this value is zero when the locomotive is at the goal position, positive when the locomotive is to the left of the goal, and negative if the locomotive overshoots and ends up to the right of the goal.

$u(t)$ The control signal at time t. The interpretation of $u(t)$ depends on system we are attempting to control. In some cases $u(t)$ might represent a low-level control signal like the voltage sent to a motor. In other cases we may be working with a robot that allows us to directly specify a desired velocity or acceleration.
In the case of our hypothetical locomotive, we will assume we have an API provides a `throttle` function that takes a number in the range (-100, 100) where +100 represents "full speed ahead" and -100 represents "full reverse".

1.3.1 Proportional Control

Mathematically, we can think of the problem of developing a controller as finding an expression for $u(t)$ in terms of $e(t)$. One simple possibility is to follow the intuition that the magnitude of the control signal should be proportional to the current error. In the locomotive example, this means we should apply more throttle when the locomotive is far from the goal location, and ease off as the locomotive gets closer. This idea can be expressed as follows:

$$u(t) = K_p e(t) \tag{1.1}$$

The value K_p is referred to as a **gain** term. This is a constant that determines how large the control signal will be for a particular error value. Doubling the gain doubles the magnitude of the control signal. Developing a successful controller involves selecting an appropriate gain value, either by analyzing the system or through trial and error.

Algorithm 1.2 shows how we can implement a proportional controller for the locomotive example.

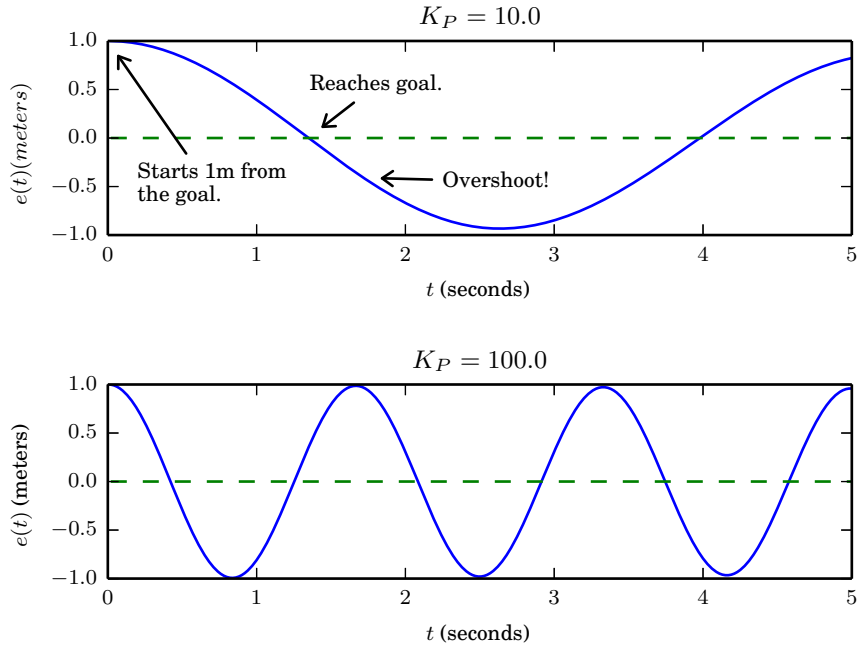


Figure 1.3: Locomotive position error over time for two different values of K_P.

Listing 1.2: Proportional Control Algorithm

```
def p_controller(train, g, K_P):
    while True:
        e = g - train.x
        u = K_P * e
        train.throttle(u)
```

Figure 1.3 shows the result of using a proportional controller to implement our locomotive controller using two different values of K_P.

These results are not very satisfying. The locomotive overshoots the goal and then over-corrects, driving back and forth indefinitely. Changing the value of K_P doesn't solve the problem, it only changes the period of the oscillations.

1.3.2 Adding a Derivative Term

One problem with our proportional controller is that it only considers the position of the locomotive, not the speed. Intuitively, it seems that if the error is already decreasing quickly, we should reduce the control signal to avoid overshooting the goal. Conversely, if the error is still increasing in spite of the proportional control, we should further increase the magnitude of the control signal. These intuitions

can be captured by adding a derivative term to the controller:

$$u(t) = K_p e(t) + \overbrace{K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t}}^{\text{derivative term}} \tag{1.2}$$

The term $\frac{\mathrm{d}e(t)}{\mathrm{d}t}$ describes the rate of change in the error. This is negative if the error is decreasing and positive if the error is increasing. The value K_d is a gain that is used to tune the impact of the derivative term.

Using the derivative term requires us to know $\frac{\mathrm{d}e(t)}{\mathrm{d}t}$, but sensors don't usually provide direct access to this value. Instead, we need to estimate it by tracking the change in error over time. Even though the physical systems we are controlling operate continuously, our algorithms necessarily perform their steps at discrete time intervals. Assuming our controller is updated every Δt seconds, we can estimate the derivative as the slope between the two most recent error values:

$$\frac{\mathrm{d}e(t)}{\mathrm{d}t} \approx \frac{e(t) - e(t - \Delta t)}{\Delta t} \tag{1.3}$$

Discrete approximations like this are common in robotics and in other areas of scientific computing but they aren't always made explicit. It takes some experience to get comfortable moving from continuous to discrete formulations. Algorithm 1.3 illustrates how we can use the discrete approximation in Equation 1.3 to implement the control algorithm described in Equation 1.2.

Listing 1.3: Proportional Derivative Control Algorithm

```
def pd_controller(train, g, K_P, K_D):
    e_prev = g - train.x
    while True:
        e = g - train.x
        dedt = (e - e_prev) / train.dt  # Equation 1.3
        u = K_P * e + K_D * dedt
        train.throttle(u)
        e_prev = e
```

Figure 1.4 shows the result of introducing the derivative term in our controller. For appropriate values of K_D the oscillations are damped, and the locomotive settles at the goal location.

Stop and Think

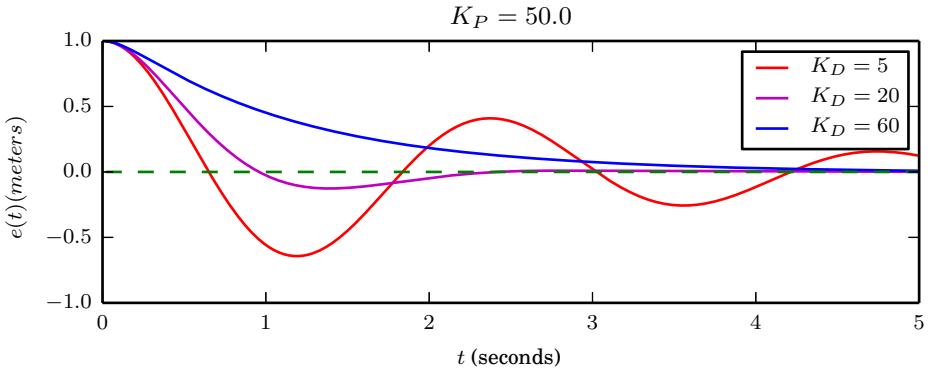


Figure 1.4: Locomotive position error over time for three different values of K_D.

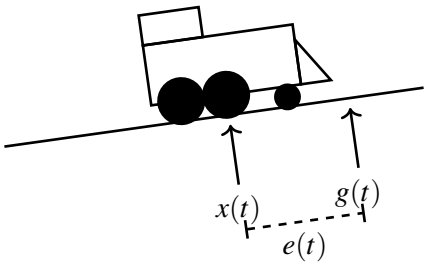


Figure 1.5: Locomotive on a hill.

1.1 The `while` loop in Listing 1.3 does not include any explicit delays. It is written under the assumption that methods called on the `train` object will only return after an appropriate delay. What could go wrong if this is not the case? In other words, what will happen if the execution time of the loop is much shorter than `train.dt`? ■

Droop

What happens when we try to apply the same controller when the locomotive is located on a slight incline as shown in Figure 1.5? In this situation, which is illustrated in Figure 1.6, the locomotive never quite reaches the goal position. The locomotive comes to rest at the point where the proportional force applied by the controller is exactly counterbalanced by gravity. Increasing K_p will move the stationary point closer to the goal, but this controller will never drive the error all the way to zero. The situation where the proportional term is not sufficient to drive the error term to zero is sometimes referred to as "droop".

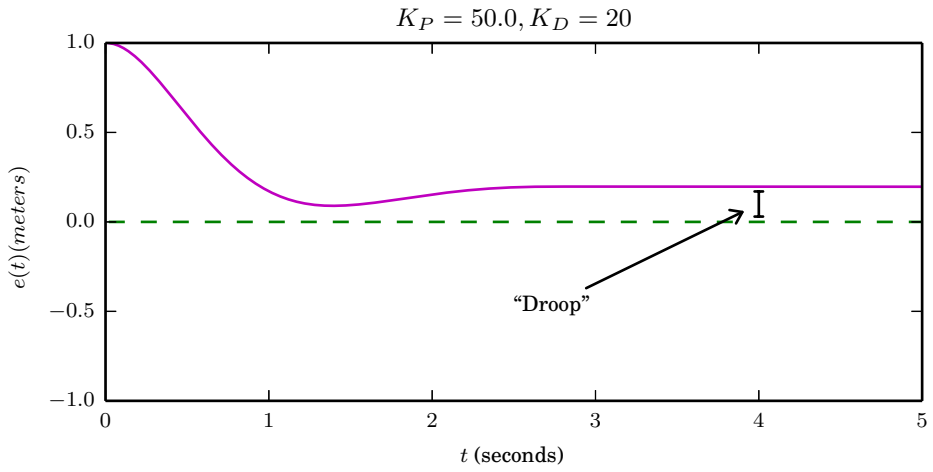


Figure 1.6: Locomotive position error over time for the PD locomotive controller when the locomotive is on an incline. The locomotive stops short of the goal at the point where the control output is counter-balanced by gravity.

1.3.3 Adding an Integral Term

The problem of droop can be addressed by adding one more term to our controller:

$$u(t) = K_P e(t) + \boxed{K_I \int_0^t e(\tau)d\tau} + K_D \frac{de(t)}{dt} \quad \text{(integral term)} \tag{1.4}$$

Where the derivative term allows the controller to look forward in time, the integral term allows the controller to look backwards in time. The integral $\int_0^t e(\tau)d\tau$ essentially "stores up" the error that the system sees over time. As long as the error fails to reach zero, $\int_0^t e(\tau)d\tau$ will steadily increase in magnitude.

As with the derivative, the integral value is not available directly, but must be estimated from discrete samples. In this case, the integral can be estimated using a summation that adds the current error value at each time step:

$$\int_0^t e(\tau)d\tau \approx \sum_{i=0}^{\boxed{t/\Delta t}} \boxed{e(i\Delta t)}\,\Delta t \tag{1.5}$$

($t/\Delta t$: # of steps before time t; $e(i\Delta t)$: Error at time step i)

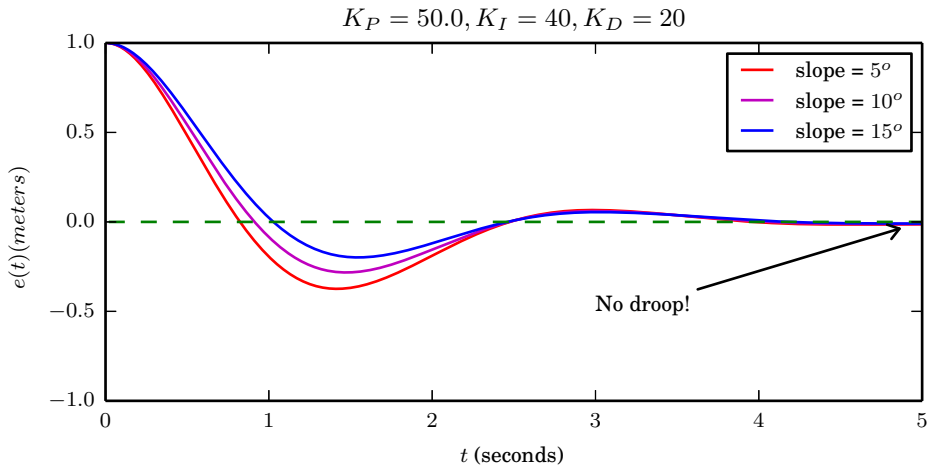


Figure 1.7: Locomotive position error over time for a different slopes. With appropriate gain values the PID controller reliably moves the locomotive to the goal location.

Listing 1.4: PID Control Algorithm

```
def pid_controller(train, g, K_P, K_I, K_D):

    e_prev = g - train.x
    e_sum = 0              # accumulator for integral term
    while True:
        e = g - train.x
        e_sum = e_sum + e * train.dt    # From Equation 1.5
        dedt = (e - e_prev) / train.dt # Equation 1.3
        u = K_P * e    +    K_I * e_sum    +    K_D * dedt
        train.throttle(u)
        e_prev = e
```

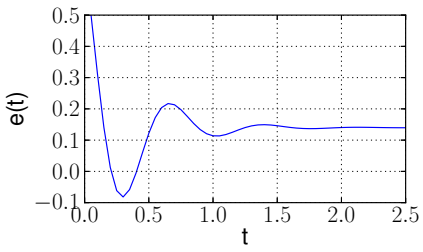
Algorithm 1.4 illustrates a complete PID controller. Figure 1.7 shows the result of adding an integral term to our locomotive controller. With an appropriate value of K_I, the resulting controller reliably moves the locomotive to the goal regardless of the slope.

Stop and Think

1.2 Practical implementations of PID controllers often place a cap on the amount of error that can be accumulated in the integral term of the controller. Why do you think this is necessary? What could go wrong if the controller were to start out far from the goal configuration, accumulating a large amount of error before the goal is reached? ■

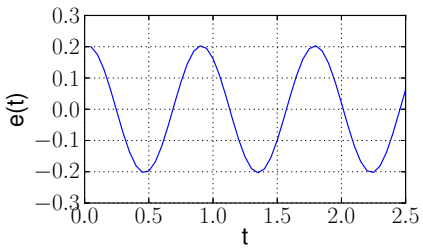
1.3 The process outlined above for tuning the PID gain terms is ad-hoc: we just experimented with different values until the behavior looked right. A more rigorous approach would require a quantitative way to evaluate the success of a particular PID controller. Can you think of some quantitative measurements that might be useful for comparing two controllers? ■

1.4 Consider the following graph of error as a function of time. Assuming that this system is being controlled by a PID controller, how would you suggest what the gain terms be modified?



■

1.5 Consider the following graph of error as a function of time. Assuming that this system is being controlled by a PID controller, how would you suggest what the gain terms be modified?



■

1.4 Proportional Robot Control

You may feel that a self-driving locomotive is a disappointingly simple robot: all it can do is move backward and forward along a linear track. Don't fear, this book will mostly focus on mobile robots that are able to move freely in multiple dimensions. In this section we will develop a proportional controller for driving a wheeled robot to a goal location as illustrated in Figure 1.8.

This is an example of a **differential drive** robot. This type of robot has two independently controllable wheels. When both wheels are rotated in the same direction at the same speed the robot moves directly forward or backward. If both wheels are turned in opposite directions the robot will rotate in place. The robot can follow a curved path by rotating each wheel at a different speed.

Since it is not intuitive to steer a differential drive robot by directly controlling the wheel velocities, the driver software for such robots often accepts commands specifying two velocity values: v and w, where v is the forward velocity in meters/second and w is the rotational velocity in radians/second.

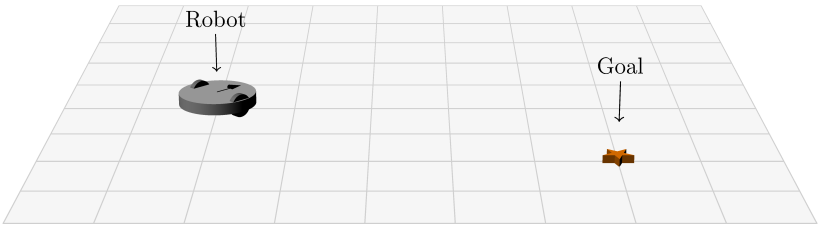


Figure 1.8: Differential drive robot.

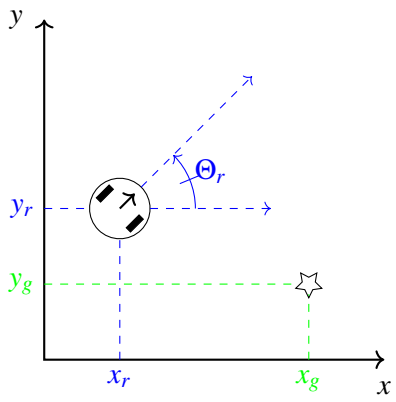


Figure 1.9: Overhead view of the robot and goal.

Figure 1.9 illustrates the parameterization of the control problem. The pose of the robot is specified with three numbers: (x_r, y_r, Θ_r), where the x_r and y_r coordinates specify the position of the center of the robot with respect to some fixed point in the room and the Θ_r value indicates the robot's heading. Heading values (in radians) are in the interval $[-\pi, \pi]$, where $\Theta_r = 0$ indicates that the robot is pointing along (or parallel to) the x-axis. As the robot turns counterclockwise, Θ_r increases. The value of Θ_r in figure Figure 1.9 is approximately $\frac{\pi}{4}$ (or 45^o).

We can solve this control problem by creating two separate proportional controllers that will operate simultaneously: one controller to output a rotational velocity that turns the robot toward the goal, the other to output a forward velocity that keeps the robot moving forward until it reaches the goal.

The first step in developing the rotational controller is determining the desired angle for the robot. The problem is illustrated in Figure 1.10. Determining Θ_g from the positions of the robot and goal requires some trigonometry:

$$\Theta_g = \tan^{-1} \frac{y_g - y_r}{x_g - x_r} \tag{1.6}$$

Actually, this won't *quite* give us what we want. Equation 1.6 doesn't distinguish between the case when both the numerator and denominator are positive and the case when they are both negative. This means that a goal above and to the right of the robot will result in the same angle as an object below and to the left. The math libraries for many programming languages address this difficulty by providing

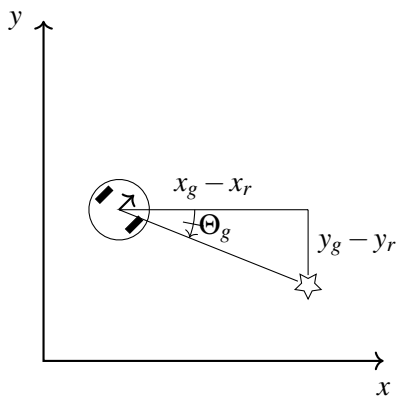


Figure 1.10: Goal angle calculations

a function named `atan2` that takes the numerator and denominator as separate arguments and correctly calculates the angle taking the signs of the arguments into account. A Python implementation might look something like the following:

```
theta_g = math.atan2(y_g - y_r, x_g - x_r)
```

Given that we can calculate a target angle, our rotational controller can be specified using the formula for a proportional controller (dropping the time index for clarity):

$$w = K_{P_w}(\Theta_g - \Theta_r) \tag{1.7}$$

Unfortunately, subtracting angles in a sensible way requires some caution. Consider the situation in Figure 1.11. Naively calculating the error by subtracting the two angles gives us $.9\pi - (-.9\pi) = 1.8\pi$. Plugging this into our proportional controller will result in a hard left turn. This isn't exactly wrong: it *is* possible for the robot to point toward the goal by turning to the left. However, it clearly makes more sense to select the smaller angle between our current heading and the target heading and have the robot turn right. We will use the symbol $\ominus$ to indicate an angle subtraction operation that has this effect. With this modification we can completely specify the control scheme for the differential drive robot.

$$w = K_{P_w}(\Theta_g \ominus \Theta_r) \tag{1.8}$$

$$v = K_{P_v}\sqrt{(y_g - y_r)^2 + (x_g - x_r)^2} \tag{1.9}$$

Equation 1.9 expresses the idea that the robot's forward velocity should be proportional to its Euclidean distance from the goal. Since this velocity control is independent of the angle, the robot may initially move away from the goal. We could address through a more complicated control mechanism, but the point here is to illustrate a simple example of proportional control. Figure 1.12 shows several example trajectories of a simulated differential drive robot under the control scheme presented in Equations 1.8 and 1.9.

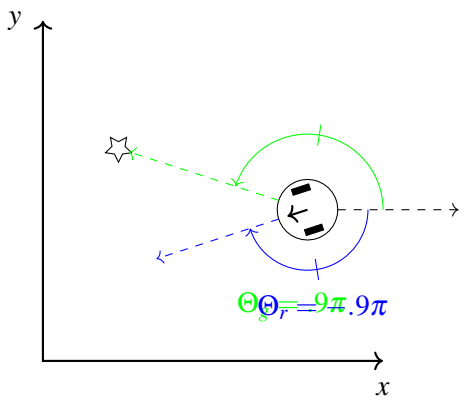


Figure 1.11: The robot can either turn $.2\pi$ radians clockwise, or 1.8π radians counterclockwise to face the goal. We want a difference operation that always returns the smaller angle.

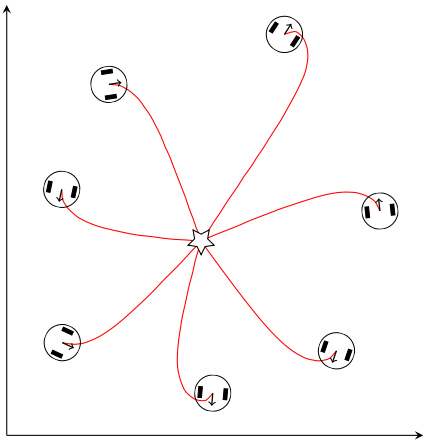


Figure 1.12: Example trajectories for a proportional controller that moves a differential drive robot to a goal location. The robot images indicate the starting positions for seven simulated navigation trials.

Notice that, unlike the locomotive example, it wasn't necessary to include integral or derivative terms in this controller. They weren't needed because we were able to set the speed of the robot directly. This means we didn't need to worry as much about overshooting the goal or applying too little force to reach it. It often makes sense to start with a pure proportional controller and add integral or derivative terms if they prove necessary.

1.5 Limitations of PID Controllers

PID control is widely used because it is simple to implement and can be applied in a wide range of situations. We can use PID control even when we don't have an accurate model of the system we are controlling. In this chapter we used trial and error to develop a PID controller for a simulated locomotive. It wasn't necessary to understand the physics involved: we didn't need to estimate frictional forces, we didn't need an exact specification of the forces created by the throttle. We only needed to tune the three gain parameters until we were satisfied with the performance.

The ad-hoc nature of PID control can also be seen as a disadvantage. PID controllers tend to work well in practice, but they don't provide any optimality guarantees. In cases where we *do* have an accurate system model, it's possible to develop more principled controllers that explicitly minimize a cost function related to the quality of the solution. This could allow us to create a controller that reaches the goal in the minimum time or with minimal energy expenditure. Such controllers are beyond the scope of this book, but the classic optimal controller is the **Linear Quadratic Regulator** or LQR.

More significantly, the kind of low-level control discussed in this chapter is obviously not appropriate for complex control problems that extend over longer periods of time. A PID controller will not save the day for the robot in Figure 1.13. Problems like this will require planning algorithms of the sort described in Chapter 3.

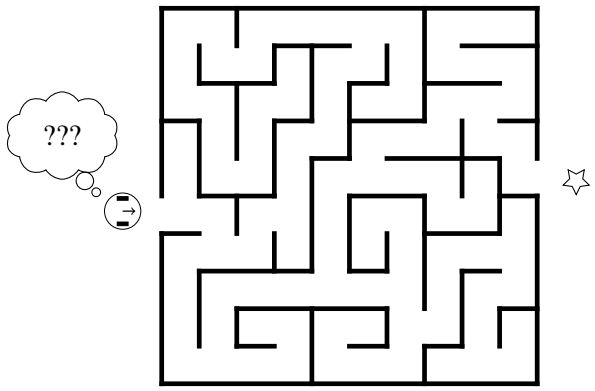


Figure 1.13: A control problem that can't be solved with the tools in this chapter.

1.6 References and Further Reading

1.6.1 Closed-Loop Controllers

Elements of the PID control model have been in place since the time of the industrial revolution. A history of the development of PID control theory since 1900 is provided in (Bennett, 2001). PID controllers are widely used in both robotics and industrial automation, and as such, there is a huge literature on tuning and implementing PID controllers. A recent 500-page reference can be found in (Johnson & Moradi, 2006).

A standard reference on the Linear Quadratic Regulator is (Anderson & Moore, 1990).

1.6.2 General Robotics Resources

There are a huge number of robotics textbooks and references. I will highlight a few here that are particularly notable and were useful in the development of this book.

The most widely used artificial intelligence textbook is (Russell & Norvig, 2010). While that book has a broader scope than just robotics, many areas of AI are directly relevant to robotics work. Another excellent general AI textbook is (Poole & Mackworth, 2017). This book has the advantage of being available online at no cost: https://artint.info/.

Computational Principles of Mobile Robotics (Dudek & Jenkin, 2010) is a robotics textbook that has a similar focus to this work: it is targeted to computer scienists and focuses on algorithmic issues in

robotics. An excellent open-access robotics textbook is (Correll et al., 2022). The definitive work on probalistic methods in robotics is *Probalistic Robotics* (Thrun et al., 2005). An excellent, and authoritative, reference on many topics in robotics is provided by the *Springer Handbook of Robotics* (Siciliano & Khatib, 2016).

2. Coordinate Frames

2.1 Introduction

A key requirement in robotics programming is keeping track of the positions and velocities of objects in space. For example, consider the situation in Figure 2.1. This robot has been programmed to find the blue teapot and report its position to a remote user. The robot's vision system has detected the teapot directly in front of the camera at a distance of 1 meter.

In this situation it would not be very helpful to report to the user that the teapot has been located `ONE METER IN FRONT OF MY CAMERA`. Presumably, the user wants to know where *in the room* the teapot is located. Providing this information requires us to know each of the following:

1. The position of the teapot relative to the camera.
2. The position and orientation of the camera relative to the base of the robot.

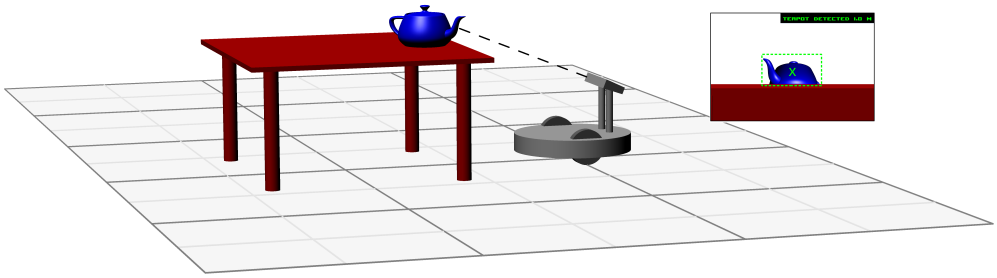


Figure 2.1: This robot has located a teapot and must report the location to a user. The inset image shows the view from the robot's camera.

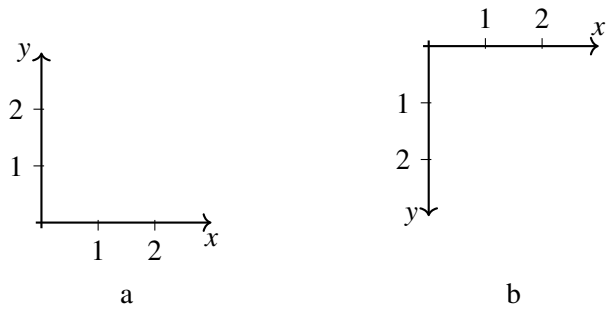


Figure 2.2: There are several ways we can draw two-dimensional coordinate systems. a. The origin is drawn in the lower-left corner. b. the origin is drawn in the upper-left.

3. The position of the robot in the room.

Determining items 1 and 3 may be challenging problems on their own, but for now, we will assume that we have access to this information. The goal in this chapter is to develop a framework for representing these relative locations and using that information to translate coordinates from one frame of reference to another. In this case, from a frame of reference defined by the position of the camera (`THE TEAPOT IS ONE METER IN FRONT OF MY CAMERA`), to a fixed frame of reference defined relative to the room (`THE TEAPOT IS NEAR THE SOUTH-WEST CORNER OF THE ROOM`).

2.2 Conventions

As a first step we need to establish some conventions for describing locations and orientations in three dimensions.

You certainly have experience working with two-dimensional coordinate systems like the one illustrated in Figure 2.2a. This figure follows the common convention of placing the origin is at the lower-left with the positive x-axis draw horizontally and the positive y-axis drawn vertically.

It is possible to draw this coordinate system differently. For example, the convention when working with coordinates on a computer screen is to place the origin at the upper-left corner as shown in Figure 2.2b.

In a sense, Figures 2.2a and 2.2b are the same coordinate system drawn in two different ways. You can imagine picking up the axes in Figure 2.2a, flipping them over and placing them back on top of the axes in Figure 2.2b so that the axes are aligned.

The situation is different in three dimensions. Depending on how the axes are arranged we can end up with one of two fundamentally incompatible coordinate systems as illustrated in Figure 2.3. The coordinate system on the left is referred to as a **left-handed coordinate system**, while the one on the right is a **right-handed coordinate system**. In a right-handed coordinate system we determine the direction of the z-axis by aiming the pointer finger of the right hand along the positive x-axis and curling our palm toward the positive y-axis. The thumb will then point in the direction of positive z. For a left-handed coordinate system we follow the same procedure using the left hand.

There is no way to rotate these two coordinate systems so that they align. They represent two incompatible ways of representing three-dimensional coordinates. This means that whenever we

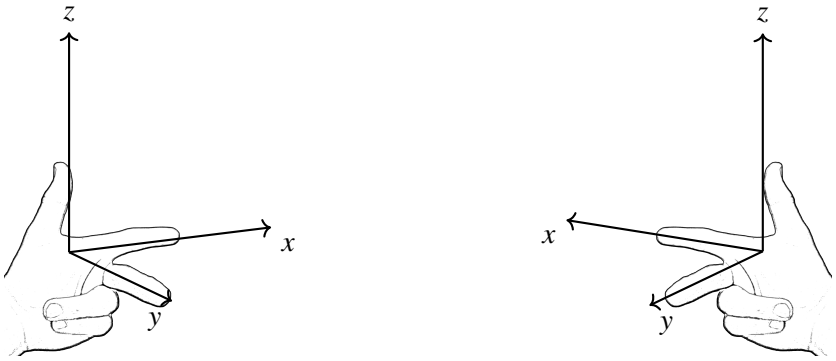


Figure 2.3: Left and right-handed coordinate systems.

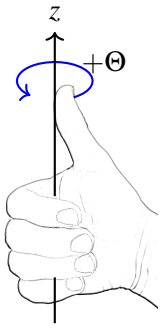


Figure 2.4: Right-hand rule for determining the direction of positive rotations around an axis.

provide coordinates in three dimensions we must specify whether we are using a left-handed or right-handed system. There is no universal convention for which should be used, but right-handed systems are more common in robotics. All of the examples in this book will assume right-handed coordinate systems.

We also need to establish a convention for describing the direction of a rotation. Again, we will follow a right-hand rule. To determine the direction of positive rotation around a particular axis we point the thumb of the right hand along the axis in question. The fingers will then curl around the axis in the direction of positive rotation. This is illustrated in Figure 2.4.

2.3 Poses and Coordinate Frames

The position and orientation of an object in space is referred to as its **pose**. Any description of an object's pose must always be made in relation to some to some **coordinate frame**. It isn't helpful to know that my teapot is at position (-2, 2.2, 1.6) unless I know where (0, 0, 0) is and which directions the three axes are pointing.

In robotics, it is often convenient to keep track of multiple coordinate frames. Figure 2.5 illustrates three potentially useful coordinate frames related to the teapot scenario described above. The "world" coordinate frame is indicated with the subscript w. This coordinate frame is fixed at a known location in space and assumed not to move over time. The "robot" coordinate frame is indicated with the subscript r. We can envision this coordinate frame as being attached to the base of the robot so that the origin of

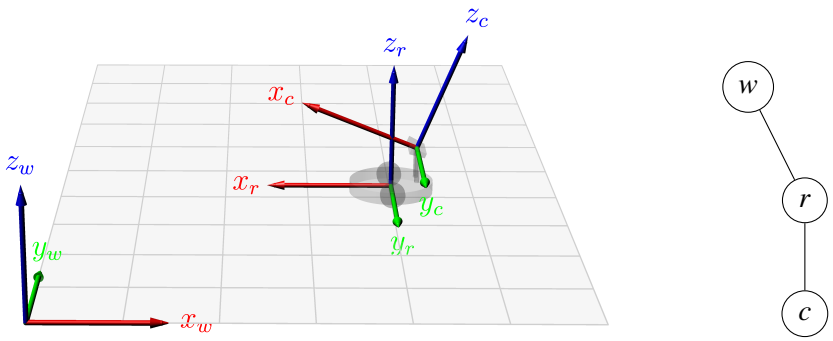


Figure 2.5: Left: three coordinate frames from the teapot scenario illustrated in Figure 2.1. Right: tree representing the relationship between the three coordinate frames.

the frame moves as the robot moves. In other words, the robot is always located at (0, 0, 0) in its own coordinate frame. Similarly, the "camera" coordinate frame is attached to the camera.

This figure follows the common convention of drawing the x-axis in red, the y-axis in green and the z-axis in blue. In the case of robots (and many other objects) we have a natural notion of directions corresponding to "forward", "left" and "up". Throughout this book these will correspond to positive x, positive y and positive z respectively.

We can organize these three coordinate frames into a tree structure with the "world" coordinate frame as the root. Each edge in the tree represents the pose of a particular coordinate frame relative to its parent. Ideally, this tree structure should mirror the physical connections between the objects involved. In this case it makes sense to describe the pose of the camera in the robot's coordinate frame, because the two are physically attached and the camera is constrained to move along with the robot. When the robot moves, it is only necessary to update the pose of the robot coordinate frame. The camera remains stationary relative to the robot.

Assuming a connected tree, it will be possible to translate a point between any two coordinate frames. For example, given the coordinates of the teapot in the camera coordinate frame, we can determine its coordinates in the robot coordinate frame, and then in the room coordinate frame. In order to accomplish this we need to specify how poses are represented. We also need a mechanism for converting from parent to child coordinate frames and vise-versa.

2.3.1 Translations

Recall that the pose of an object (or a coordinate frame) includes both its position and orientation relative to the parent coordinate frame. Representing the position is straightforward. Three numbers may be used to represent a translation of the object along each of the three axes of the parent coordinate frame.

Figure 2.6 shows two coordinate frames separated by a simple translation. In this example, the child coordinate frame is located at position (1.5, 1.0, .5) in the parent coordinate frame.

We will use subscripts to indicate the coordinate frame associated with a point or pose. For example, $(0,0,.5)_c$, refers to the point $(0,0,.5)$ in the child coordinate frame.

When coordinate frames are separated by a pure translation transforming points between frames is

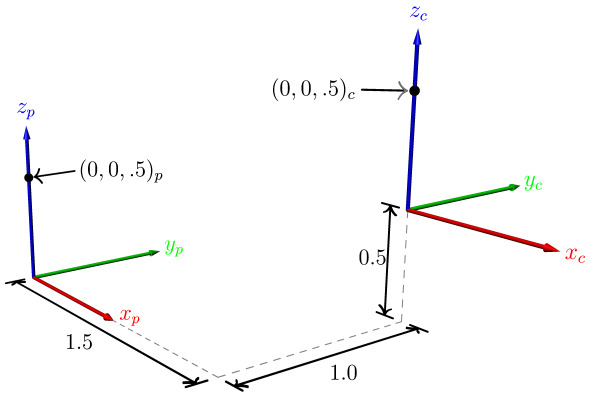


Figure 2.6: Two coordinate frames separated by a simple translation. The parent frame is labeled p, the child frame is labeled c.

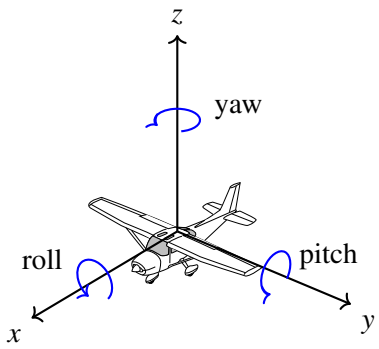


Figure 2.7: Euler angles.

straightforward: we only need to add or subtract the three coordinate offsets. For example, the point $(0,0,.5)_c$ is located at $(0+1.5,0+1.0,.5+.5) = (1.5,1.0,.5)$ in the parent coordinate frame. Moving from the parent to the child requies us to subtract instead of add: $(0,0,.5)_p$ is located at $(-1.5,-1.0,0)$ in the child frame.

2.3.2 Euler Angles

Specifying orientation in three dimensions is more complicated than specifying position. There are several approaches, each with its own strengths and weaknesses. **Euler angles** are probably the most intuitive representation. Figure 2.7 illustrates the basic idea. Orientation is expressed as a sequence of three rotations around the three coordinate axes. These three rotations are traditionally referred to as **roll**, **pitch** and **yaw**.

When working with Euler angles it is necessary to specify the order that the rotations will be applied: a 90^o rotation around the x-axis followed by a 90^o rotation around the y-axis does *not* result in the same orientaion as the same rotations applied in the opposite order. There are, in fact, *twelve* valid rotation orderings: xyz, yzx, zxy, xzy, zyx, yxz, zxz, xyx, yzy, zyz, xzx, and yxy.

It is also necessary to specify whether the rotations are relative to the parent coordinate frame, or whether each rotation is performed around the axes of a coordinate frame aligned with the earlier

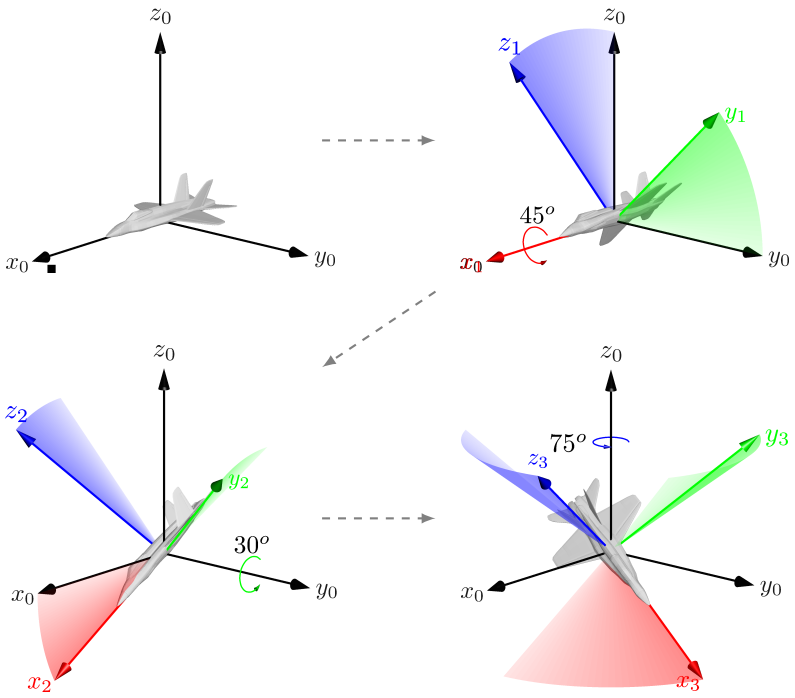


Figure 2.8: Static

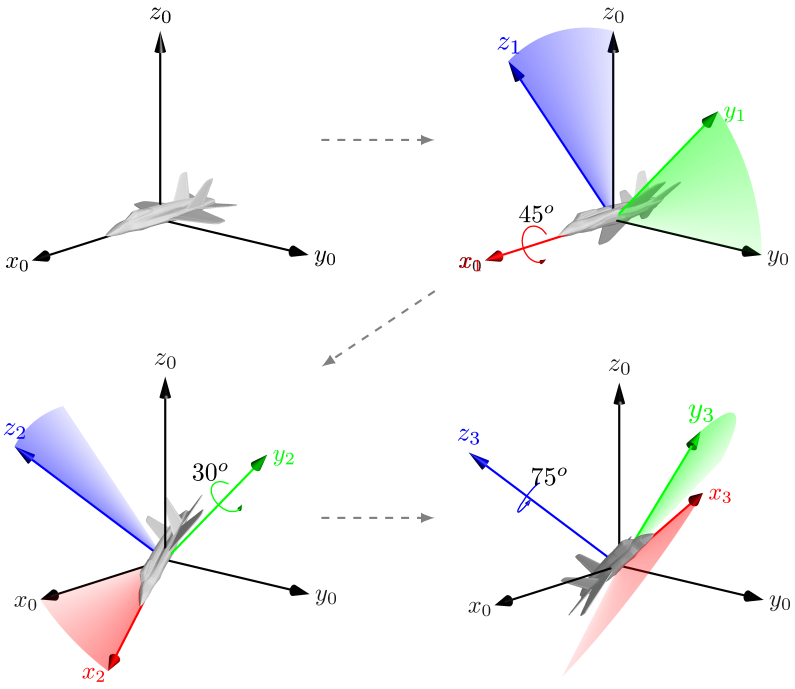


Figure 2.9: Non static

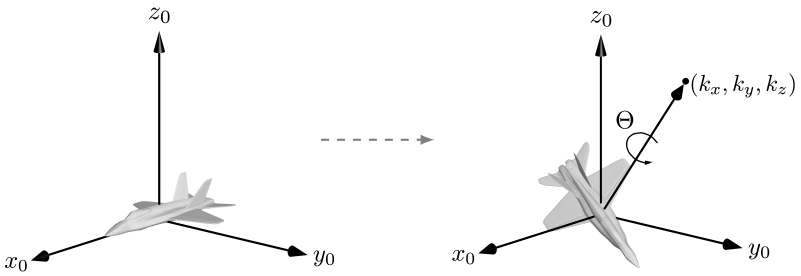


Figure 2.10: Axis-angle representation of orientation.

rotations. These two alternatives are illustrated in Figures 2.8 and 2.9. The first alternative is sometimes referred to as "static" or "extrinsic" rotations, while the second may be referred to as "relative" or "intrinsic" rotations. Combined with the choice of axis orderings, this means that there are twenty-four possible conventions for specifying Euler angles.

Euler angles are relatively easy to visualize, but they are inconvenient to work with from a mathematical point of view. The key problem is that the mapping from spatial orientations to Euler angles is discontinuous: small changes in orientation may cause big jumps in the required representation. This can cause difficulties when we need to smoothly update the orientation of a moving object over time.

2.3.3 Axis Angle

Euler angles encode an orienation using three rotations around pre-defined axes. An alternate approach is to encode an orientation as a *single* rotation Θ around an arbitrary unit vector $[k_x, k_y, k_z]^T$. This is referred to as the **axis-angle** representation. The idea is illustrated in Figure 2.10.

2.3.4 Quaternions

Probably the most widely used method for encoding orientation is the **quaternion**. There is a close relationship quaternions and the axis-angle representation described above. If Θ and $[k_x, k_y, k_z]^T$ describe the axis-angle representation of an orientation, then the quaternion represetation is a four-tuple (x, y, z, w) such that:

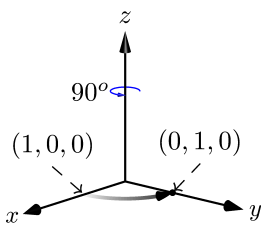
$$x = k_x \sin\frac{\Theta}{2} \tag{2.1}$$

$$y = k_y \sin\frac{\Theta}{2} \tag{2.2}$$

$$z = k_z \sin\frac{\Theta}{2} \tag{2.3}$$

$$w = \cos\frac{\Theta}{2} \tag{2.4}$$

Notice that the x, y and z components point in the same direction as the original axis of rotation. A quaternion constructed according to Equations 2.1-2.4 is also referred to as a unit-quaternion because it has the property that $\sqrt{x^2 + y^2 + z^2 + w^2} = 1$.



$$R_z(90^o) \times \begin{bmatrix}1\\0\\0\end{bmatrix} = \begin{bmatrix}\cos 90^o & -\sin 90^o & 0\\ \sin 90^o & \cos 90^o & 0\\ 0 & 0 & 1\end{bmatrix} \times \begin{bmatrix}1\\0\\0\end{bmatrix}$$
$$= \begin{bmatrix}0 & -1 & 0\\ 1 & 0 & 0\\ 0 & 0 & 1\end{bmatrix} \times \begin{bmatrix}1\\0\\0\end{bmatrix}$$
$$= \begin{bmatrix}0\\1\\0\end{bmatrix}$$

Figure 2.11: Example rotation

Quaternions are *not* particularly intuitive or easy to interpret. Their popularity is based on the fact that they support efficient computation and they avoid the troublesome discontinuities associated with Euler angles.

2.3.5 Rotation Matrices

Until now we have represented spatial coordinates as ordered triples: (x,y,z). Equivalently, we may think of points in space as three-dimensional column vectors: $\begin{bmatrix}x & y & z\end{bmatrix}^T$. This view is convenient because certain spatial transformations may be accomplished through matrix operations. In particular, any rotation can be encoded as a 3×3 **rotation matrix**. Pre-multiplying the matrix by a point will have the effect of performing the desired rotation around the origin.

The following three matrices correspond to rotations around the x, y, z axes respectively:

$$R_x(\Theta) = \begin{bmatrix}1 & 0 & 0\\ 0 & \cos\Theta & -\sin\Theta\\ 0 & \sin\Theta & \cos\Theta\end{bmatrix} \tag{2.5}$$

$$R_y(\Theta) = \begin{bmatrix}\cos\Theta & 0 & \sin\Theta\\ 0 & 1 & 0\\ -\sin\Theta & 0 & \cos\Theta\end{bmatrix} \tag{2.6}$$

$$R_z(\Theta) = \begin{bmatrix}\cos\Theta & -\sin\Theta & 0\\ \sin\Theta & \cos\Theta & 0\\ 0 & 0 & 1\end{bmatrix} \tag{2.7}$$

Figure 2.11 shows the example of rotating the point $\begin{bmatrix}1 & 0 & 0\end{bmatrix}^T$ by 90^o around the z-axis.

It is possible to represent any orienation as a product of three rotation matrices around the x, y and z axes. It is straightforward to convert from an Euler angle representation to the corresponding rotation

matrix. For static rotations, the three elementary rotation matrices must be multiplied in the order that the rotations should be applied. This means that the rotation matrix that corresponds to the Euler angle rotations illustrated in Figure 2.8 would be:

$$R_{static} = R_x(45^o) \times R_y(30^o) \times R_z(75^o)$$

For Euler angles represented using relative rotations, the order is reversed. The rotation matrix corresponding to Figure 2.9 would be:

$$R_{relative} = R_z(75^o) \times R_y(30^o) \times R_x(45^o)$$

The main advantage of rotation matrices is that they provide a convenient mechasism for composing rotations and for applying rotations to a point. The downside is that this is a highly redundant represenatation. A 3×3 rotation matrix contains 9 values, but as we saw in Section 2.3.2 three numbers are sufficient to represent any orientation in a three dimensional space.

2.4 References and Further Reading

Jennifer Kay provides an excellent and very accessible tutorial on kinematics and coordinate transforms in (Kay, 2020). That tutorial also describes **homogeneous coordinates** which provide a convenient mechanism for combining rotations and translations into a single matrix product.

A widely used open-source software library for managing coordinate frames and performing coordinate transforms is tf/tf2 (http://wiki.ros.org/tf2). The tf2 library tracks time-stamped information about the relative positions and orientations of a tree of coordinate frames and supports efficient transforms between any two frames. Introducing a time dimension is valuable because robots move over time and must interact with dynamic environments. The design and implementation of the tf library is described in (Foote, 2013).

3. Path Planning

Chapter 1 explored the problem of selecting a control signal to drive a one-dimensional system into a desired goal state. This Chapter will explore the more general problem of controlling multi-dimensional systems in the presence of obstacles.

Broadly speaking, control algorithms can be broken into two categories: **reactive** and **planning-based**. In a reactive system the control signal is determined solely by comparing the current state of the system to the goal configuration. The PID controller introduced in Chapter 1 is an example of a reactive controller.

In contrast, planning-based controllers explicitly search for a sequence of steps that will move the state of the system into a goal configuration. Planning-based control tends to be more appropriate in constrained control problems such as navigating in the presence of obstacles. In these scenarios a reactive controller may drive the system into a dead end, where a planning-based controller is able to look ahead to avoid actions that won't lead to the goal. This chapter will focus on planning-based controllers.

3.1 Configuration Spaces

Robots come in all shapes and sizes, from tiny puck-shaped robots to humanoid robots with dozens of degrees of freedom. **Configuration Spaces** provide a common framework for expressing the state of a robot within its environment. A robot's configuration $\mathbf{q}$ is a vector that contains all of the information necessary to completely specify the location of a robot and all of its constituent parts. For the locomotive robot from Chapter 1, $\mathbf{q}$ would be a single scalar value indicating the robot's location on the tracks. For a humanoid robot, $\mathbf{q}$ would include all of the robot's joint angles as well as the robot's position and orientation within its workspace. The full space of possible configurations is referred to as the robot's **Configuration Space** or **C-space** and is expressed as $\mathbf{q} \in \mathcal{C}$.

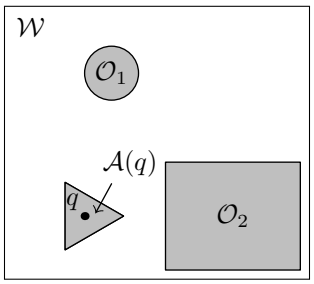


Figure 3.1: Example of a triangular robot in a planar environment.

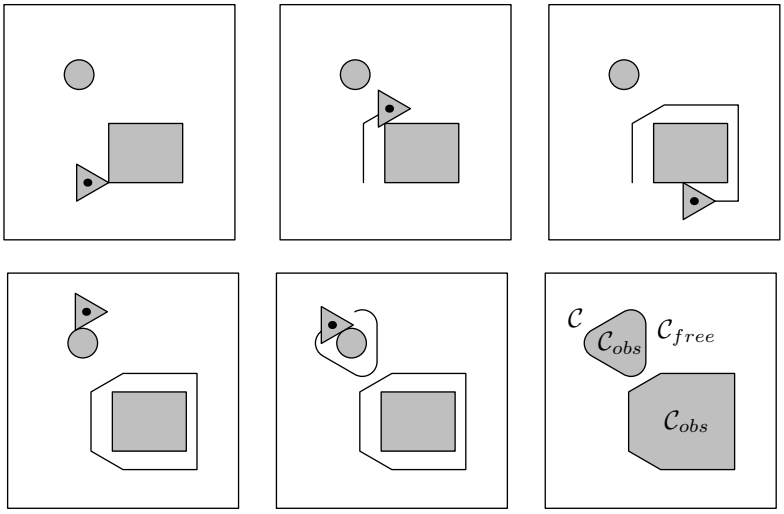


Figure 3.2: Determining $\mathcal{C}_{obs}$. The bottom-right figure illustrates $\mathcal{C}_{obs}$ and $\mathcal{C}_{free}$ for the environment shown in Figure 3.1.

While $\mathbf{q}$ may be high-dimensional, the world that the robot inhabits, $\mathcal{W}$, has only three spatial dimensions (or two if we are considering a planar path-planning problem). The mapping from a configuration $\mathbf{q}$ to the region of space occupied by a robot is denoted $\mathcal{A}(\mathbf{q}) \subset \mathcal{W}$ where $\mathcal{W} = \mathbb{R}^3$ or $\mathcal{W} = \mathbb{R}^2$.

In most workspaces there will be configurations that are impossible because they would place some part of the robot inside an obstacle. We can denote the region of space occupied by obstacles as $\mathcal{O} \subset \mathcal{W}$. For the purposes of planning it is useful to map these obstacles into the configuration space of the robot. The result is called the **C-space obstacle region** $\mathcal{C}_{obs}$ and can be expressed as $\mathcal{C}_{obs} = \{\mathbf{q} \in \mathcal{C} \mid \mathcal{A}(\mathbf{q}) \cap \mathcal{O} \neq \emptyset\}$. The unobstructed portion of of the configuration space is denoted $\mathcal{C}_{free}$ where $\mathcal{C}_{free} = \mathcal{C} - \mathcal{C}_{obs}$.

Figure 3.1 provides an example of a triangular robot in a two-dimensional workspace containing two obstacles. This robot may translate within the workspace but may not rotate. This example is atypical in that both $\mathcal{W}$ and $\mathcal{C}$ are two-dimensional. If we allowed the robot to rotate, $\mathbf{q}$ would require an additional dimension to represent the robot's orientation. The resulting configuration space would be expressed as $\mathcal{C} = \mathbb{R}^2 \times \mathbb{S}^1$, where $\mathbb{S}$ represents the set of real-valued numbers representing an angle or a point on a circle. If we were to additional appendages to the robot, like a trailer and an articulated arm, the dimensionality of $\mathcal{C}$ would increase still more.

Figure 3.2 illustrates the relationship between the shape of the robot, the locations of the obstacles

and $\mathcal{C}_{obs}$. In this case, we can imagine determining $\mathcal{C}_{obs}$ by tracking $\mathbf{q}$ as we slide the robot around the boundaries of the two obstacles.

The example in Figure 3.2 is only intended to illustrate the relationship between $\mathcal{A}(\mathbf{q})$ and $\mathcal{C}_{obs}$. It doesn't represent a general algorithm for solving the problem. The problem of determining $\mathcal{C}_{obs}$ for arbitrary robots and obstacle shapes is not straightforward, particularly in higher-dimensional configuration spaces. In practice, we don't typically attempt to calculate $\mathcal{C}_{free}$ before searching for a plan. It is usually more computationally efficient to check configurations as-needed when they arise during the planning process.

One shortcut for approximating $\mathcal{C}_{obs}$ is to represent both obstacles and $\mathcal{A}(\mathbf{q})$ as spheres, or as sets of spheres. As long as the spheres are large enough to completely enclose the objects, any non-overlapping configuration is guaranteed to be in $\mathcal{C}_{free}$. This reduces collision detection to the problem of calculating distances between object centers.

Figure 3.3 provides a second example of a robot and the corresponding configuration space. This is a planar two-link articulated arm. For this robot $\mathcal{C} = \mathbb{S}^2$.

Summary of Notation

- $\mathcal{W} = \mathbb{R}^3$ (or $\mathcal{W} = \mathbb{R}^3$) - The workspace containing the robot
- $\mathbf{q}$ - The robot's configuration
- $\mathcal{C}$ - The robot's configuration space
- $\mathcal{A}(\mathbf{q}) \subset \mathcal{W}$ - The region of space occupied by robot $\mathcal{A}$ in configuration $\mathbf{q}$
- $\mathcal{O} \subset \mathcal{W}$ - The region of space occupied by obstacles
- $\mathcal{C}_{obs}$ - The subset of the configuration space that is unreachable because it involves the robot intersecting with an obstacle.
- $\mathcal{C}_{free}$ - The subset of the configuration space that is accessible to the robot

3.1.1 The Path Planning Problem

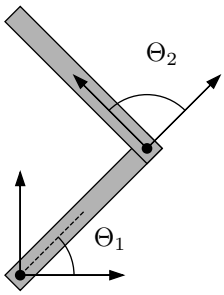
Once $\mathcal{C}_{free}$ is determined, the path-planning problem is conceptually simple. All we need to do is find a continuous path in $\mathcal{C}_{free}$ from some starting configuration $\mathbf{q}_I$ to the goal configuration $\mathbf{q}_G$. This standard formulation (sometimes called **the piano movers problem**) allows us to avoid re-thinking the path planning problem for each new robot or environment. All of robotic path planning boils down to finding a path from one point to another within a known subset of some (potentially high-dimensional) space. Figures 3.4 and 3.5 show examples of valid paths for the triangle robot and the two-link arm respectively.

3.1.2 Constraints

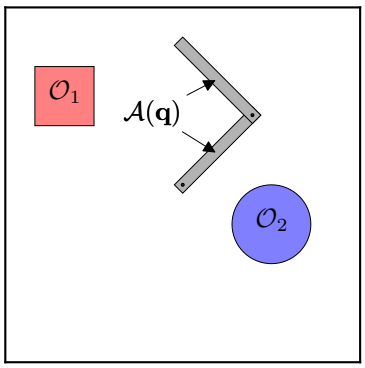
The discussion above assumes that any robot configuration is allowed, as long as the robot doesn't intersect with an obstacle. In practice, there are often additional constraints on the set of possible configurations. These constraints can be classified as **holonomic** or **non-holonomic**.

Holonomic Constraints

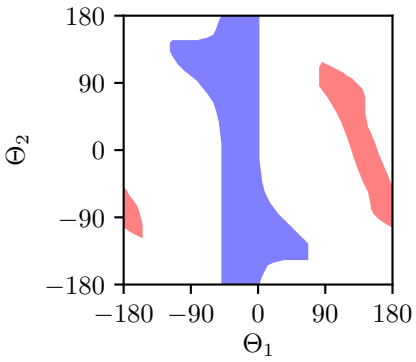
Holonomic constraints result from physical restrictions that make it impossible for the robot to enter some regions of the configuration space. For example, the elbow joint on a robotic arm may only have a 90 degree range of motion. Holonomic constraints don't significantly complicate the path planning problem: we can simply extend our notion of $\mathcal{C}_{free}$ to exclude restricted regions.



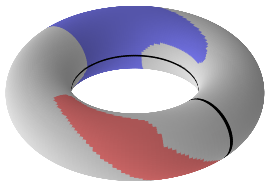
(a) 2D robot arm with two degrees of freedom. For this robot $\mathbf{q} = [\Theta_1, \Theta_2]^T$.



(b) An example of a possible configuration of the arm along with a pair of obstacles. Notice that $\mathcal{O}_2$ is close enough to the arm to prevent the first link one from making a full rotation.



(c) An illustration of $\mathcal{C}_{obs}$ for the robot configuration illustrated in 3.3b. The colors indicate configurations that intersect with the corresponding objects in 3.3b.



(d) The configuration space from 3.3c represented as a torus. The black lines are located at $\theta_1 = 180/-180$ and $\theta_2 = 180/-180$. We could imagine creating 3.3c by cutting along these lines and unwrapping the surface.

Figure 3.3: Configuration space visualizations for a planar two-link robot arm.

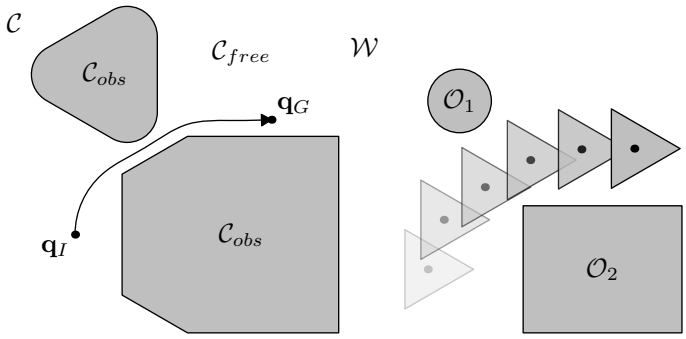


Figure 3.4: Left: A valid path from an initial configuration $\mathbf{q}_I$ to a goal configuration $\mathbf{q}_G$. Right: The robot trajectory corresponding to the indicated path.

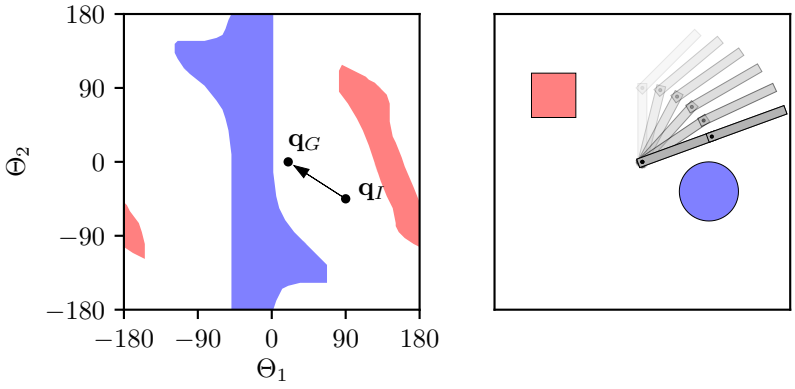


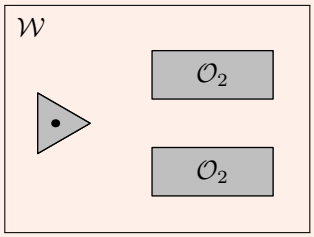
Figure 3.5: Left: A valid path from an initial configuration $\mathbf{q}_I$ to a goal configuration $\mathbf{q}_G$. Right: The robot trajectory corresponding to the indicated path.

Non-Holonomic Constraints

Non-holonomic constraints don't directly restrict which regions of the space are accessible. Instead, they restrict how the robot can move from one configuration to another. A classic example of a non-holonomic constraint is the inability of a car to slide sideways into a parking spot. There is no constraint preventing the car from being in the parking spot, but the mechanics of the vehicle prevent it from following a straight-line path to the desired configuration. Non-holonomic constraints can also arise from system dynamics: A vehicle moving at 5 miles per hour can easily make a 30 degree turn, while a vehicle moving at 50 miles per hour would roll over. Non-holonomic constraints of this sort are referred to as **kino-dynamic constraints**. Non-holonomic constraints complicate the path planning problem and require the use of specialized algorithms.

Stop and Think

3.1 Draw $\mathcal{C}_{free}$ and $\mathcal{C}_{obs}$ for the world pictured below:



Assume that this is the same non-rotating triangular robot illustrated in Figures 3.1 and 3.2. ■

3.2 The robot in Figure 3.1 has a configuration space that can be expressed as $\mathcal{C} = \mathbb{R}^2$, the robot in Figure 3.3 had a configuration space expressed as $\mathcal{C} = \mathbb{S}^2$. Imagine a more complex robot that combines attributes of these two examples. This new robot is a triangle robot that is able to translate, rotate in place, and is equipped with a two-link robotic arm. How many degrees of freedom does this robot have? How would we express configuration space for this robot? ■

3.3 Some configurations of a robotic arm may be impossible because of self-collisions. Is this an example of a holonomic or a non-holonomic constraint? ■

3.4 In the text above we used set-builder notation to express the C-obstacle region as as $\mathcal{C}_{obs} = \{\mathbf{q} \in \mathcal{C} \mid \mathcal{A}(\mathbf{q}) \cap \mathcal{O} \neq \emptyset\}$, where $\mathcal{O}$ represents the subset of $\mathcal{W}$ occupied by obstacles. What set of configurations are represented by $\{\mathbf{q} \in \mathcal{C} \mid (\mathcal{A}(\mathbf{q}) \cup \mathcal{O}) \subseteq \mathcal{O}\}$? ■

3.2 Planning Algorithms

The general path planning problem outlined above is conceptually simple but computationally difficult. Path planning has been shown to be PSPACE-Hard, making it extremely unlikely that we will ever have a polynomial time algorithm that is guaranteed to find a path if one exists. This means that practical planning algorithms necessarily involve compromises in completeness or involve attacking

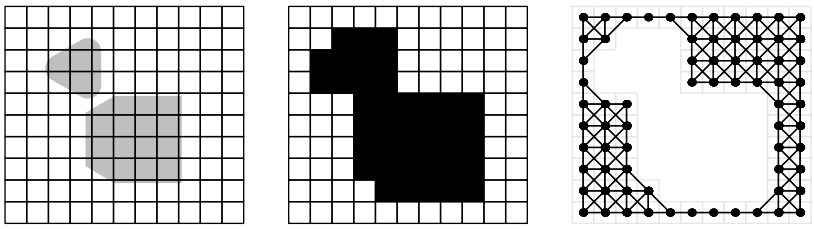


Figure 3.6: Discretizing the configuration space. The figure on the left shows a uniform grid overlaying the configuration space. In the center figure, all grid cells that overlap with $\mathcal{C}_{obs}$ have been marked as inaccessible. The figure on the right shows the state space graph that results if all accessible cells are connected to their neighbors.

simplified versions of the problem. The remainder of this chapter will explore several approaches to planning.

3.3 Grid-Based Search

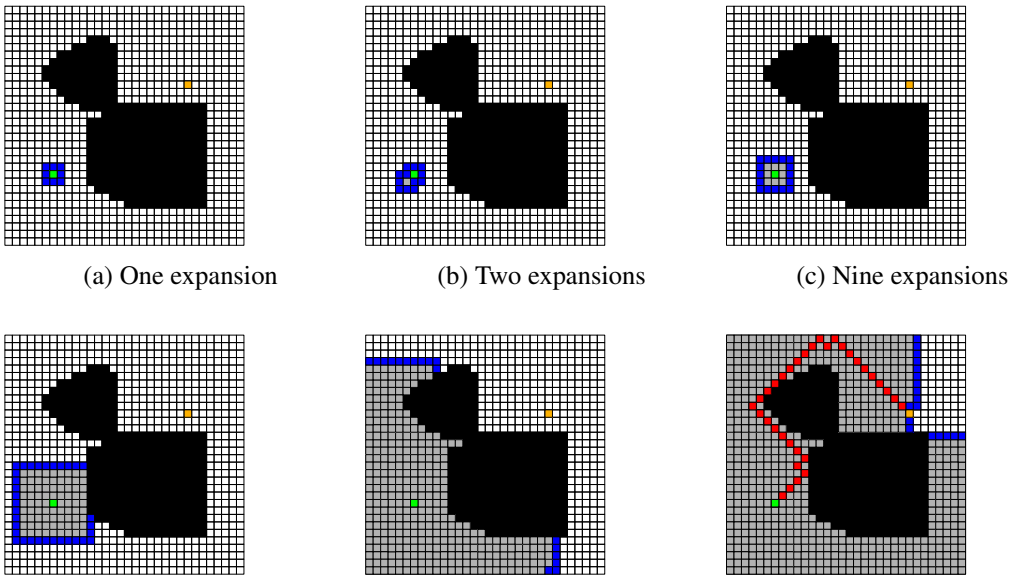
One way to simplify the planning problem is to perform a uniform discretization of the configuration space and the space of possible actions. For example, it is common to handle holonomic 2D navigation problems by overlaying the 2D configuration space with a grid. Actions are restricted to moving between four-connected or eight-connected grid cells. Figure 3.6 illustrates one possible discretization of the triangle-robot configuration space described in the previous section. The result is a graph where the vertices represent states and the edges represent actions.

Notice that the discretization in Figure 3.6 makes it impossible for the robot to find a path that passes between the two obstacles. This highlights a trade-off between the granularity of the discretization and the quality of the possible solutions. We can make our discretization arbitrarily close to the continuous version of the problem by reducing the granularity, but finer discretizations come at the cost of more grid cells which increases the cost of planning.

```python
def search(problem):
    """
    Args:
        problem: a problem instance that provides three methods:

              problem.start() -       returns the start state
              problem.goal()  -       returns the goal state
              problem.successors(s) - returns the states that are
                                      adjacent to s

    Returns:
        True if there exists a sequence of states leading from
        problem.start() to problem.goal(), or False if no such
        path exists
    """

    frontier = Collection()  # The collection of states that are
                             # currently eligible for expansion

    closed = set()           # The set of states that have already
                             # been expanded

    frontier.add(problem.start())  # Initialize the search by adding
                                   # the start state to the frontier

    while not frontier.is_empty():
        cur_state = frontier.pop()  # Select the next eligible state
                                    # to expand

        closed.add(cur_state)       # Make sure that this state can't
                                    # be added to the frontier again

        if cur_state == problem.goal():  # Success!
            return True

        else:
            # Add the neighbors of the selected state to the frontier...
            for next_state in problem.successors(cur_state):
                if (next_state not in closed and
                        next_state not in frontier):
                    frontier.add(next_state)

    return False   # No path was found!
```

Listing 1: Basic graph search algorithm.

Once a search problem has been formulated as a graph, we can use standard graph-search algorithms to find a permissible path to the goal. Listing 1 presents the basic graph search algorithm. The algorithm works outward from the start state, maintaining a collection of states on the frontier of the region we have already searched. At each iteration, we select a state from the frontier and add its neighbors back



(a) One expansion (b) Two expansions (c) Nine expansions

(d) 81 expansions (e) 350 expansions (f) 641 expansion

Figure 3.7: BFS search example. The cell containing the start state is colored green and the goal state is colored gold. States in the frontier are shown in blue. States in the closed set are shown in gray. The final path is shown in red.

into the frontier. This process continues until the goal state is selected from the frontier.

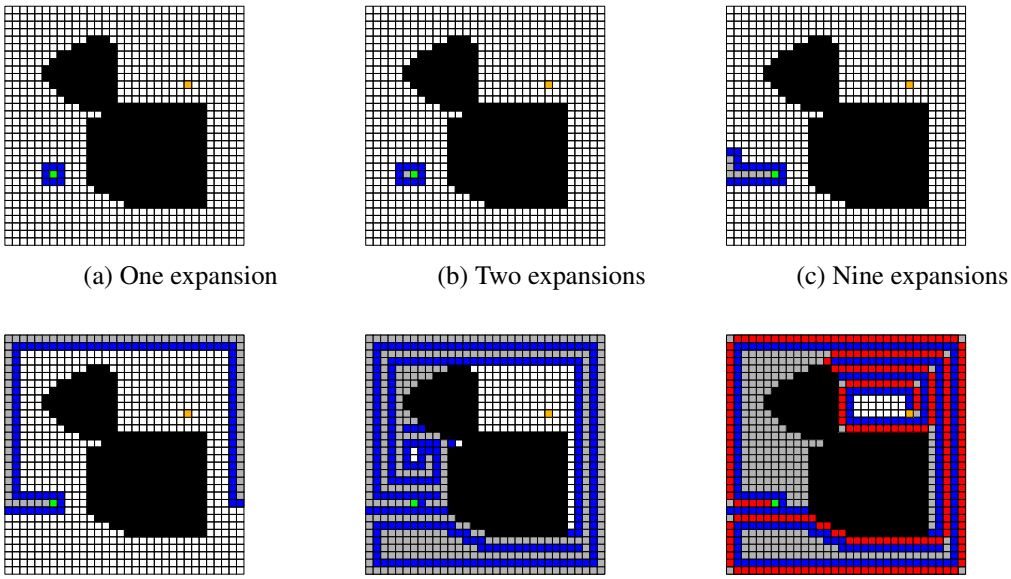
Selecting and processing a state from the frontier is referred to as *expansion*. Maintaining a *closed set* allows us to avoid re-processing states that have already been expanded. Each time a state is expanded it is added to the closed set. Once a state is closed, the check on line 39 ensures that it will never be added back into the frontier. This check prevents the algorithm from considering paths containing cycles.

The collection type used to store the frontier determines the order that the states are expanded during search. When a queue is used, the algorithm in Listing 1 performs a breadth-first-search (BFS), while the use of a stack results in depth-first-search (DFS).

For search problems with uniform step costs, BFS is both **complete** and **optimal**. A complete search algorithm is guaranteed to find a path if one exists. An optimal search algorithm is guaranteed to find the lowest-cost path. BFS is optimal in the sense that it is guaranteed to find the path with the fewest possible steps. By storing the frontier in a FIFO queue we ensure that all paths of length n are explored before we explore any paths of length $n+1$.

In contrast, DFS is neither complete nor optimal. In the case of an infinite graph, DFS could potentially perform infinite expansions away from the goal state, even if the goal could be reached within only a small number of steps. The advantage of DFS is that it requires less memory to store the frontier. The number of nodes in the frontier grows linearly for DFS but it may grow exponentially for BFS, depending on the structure of the search problem.

Figure 3.7 illustrates a BFS search in our triangle-world workspace. Figure 3.8 illustrates a DFS search.



(a) One expansion (b) Two expansions (c) Nine expansions

(d) 81 expansions (e) 350 expansions (f) 477 expansion

Figure 3.8: DFS search example.

3.3.1 Returning A Path

The careful reader will notice that the algorithm in Listing 1 doesn't actually return a path to the goal. It only determines whether such a path exists. In order to reconstruct the path we need to maintain an auxiliary data structure that stores backward references from each state to the state that precedes it. Once the goal is reached, the path can be reconstructed by following the backward references from the goal back to the start state and then reversing the steps. Listing 2 shows a possible Python implementation of the necessary data type and an appropriately updated version of the algorithm from Listing 1.

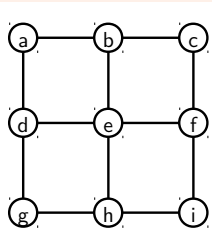
The containment check on line 40 of Listing 2 requires some clarification. The goal here is to avoid exploring multiple paths that pass through the same state. If the frontier already contains a node for a state, we don't want to add another node that represents a different route to that state. This means that the test on line 40 is *not* checking to see if the indicated Node is in the frontier, it is checking to see if any Node in the frontier contains `new_state`. Removing this check would not impact the correctness of the algorithm, but it would significantly impact efficiency for problems that allow many different paths to each state.

```
class Node:
    """The Node class stores backward references from each state
       encountered in the search to a Node representing the
       state that preceded it.
    """
    def __init__(self, state, parent_node):
        self.state = state
        self.parent = parent_node


def search(problem):
    """
    Input: problem - a problem instance that provides three methods:

              problem.start() -       returns the start state
              problem.goal()  -       returns the goal state
              problem.successors(s) - returns the states that are
                                      adjacent to s

    Returns: A sequence of states leading from problem.start() to
             problem.goal(), or None if no path exists
    """
    frontier = Collection()
    closed = set()

    frontier.add( Node(problem.start(), None) )

    while not frontier.is_empty():
        cur_node  = frontier.pop()
        cur_state = cur_node.state
        closed.add(cur_state)

        if cur_state == problem.goal():
            return construct_path(cur_node)  # path ending at this node

        else:
            for next_state in problem.successors(cur_state):
                next_node = Node(next_state, cur_node)
                if (next_state not in closed and
                        next_node not in frontier): # See text.
                    frontier.add( next_node )

    return None
```

Listing 2: Graph search algorithm updated to return a path. Steps that differ from Listing 1 are highlighted in yellow. The key difference is that the frontier stores Node objects rather than states.

Stop and Think

3.5 Consider the following graph:



Complete the table below to show the state of search algorithm 2 after each iteration of a DFS search starting at state g with the goal at state a. The tuples in the Frontier column represent Node objects where the first entry is the state and the second entry is the state associated with the parent node. Assume that the loop on line 37 accesses neighbors in alphabetical order. The first three steps are completed for you.

Expansions	Chosen	Frontier	Closed Set
0	–	$\langle g,-\rangle$	–
1	g	$\langle d,g\rangle$, $\langle h,g\rangle$	$\{g\}$
3	h	$\langle d,g\rangle$, $\langle e,h\rangle$, $\langle i,h\rangle$	$\{g,h\}$
3			
4			
5			
6			
7			

What is the final path discovered by DFS? (You should be able to reconstruct it by working backward through the parent entries starting from the point where a is selected from the frontier.) ■

3.6 Repeat the previous exercise using a BFS search. ■

3.7 The algorithm in Listing 1 is described as a graph search algorithm, but the graph is implicit: graph edges are essentially generated as needed through calls to `problem.successors`. Why might this approach make more sense than explicitly creating a complete graph structure before executing a search? ■

3.8 What if we knew that the state space being searched is actually a tree rooted at the start state? How would this allow us to simplify the search algorithm in Listing 1? Can you think of a path-planning problem that would have a tree-structured search space? ■

3.3.2 Dijkstra's Algorithm

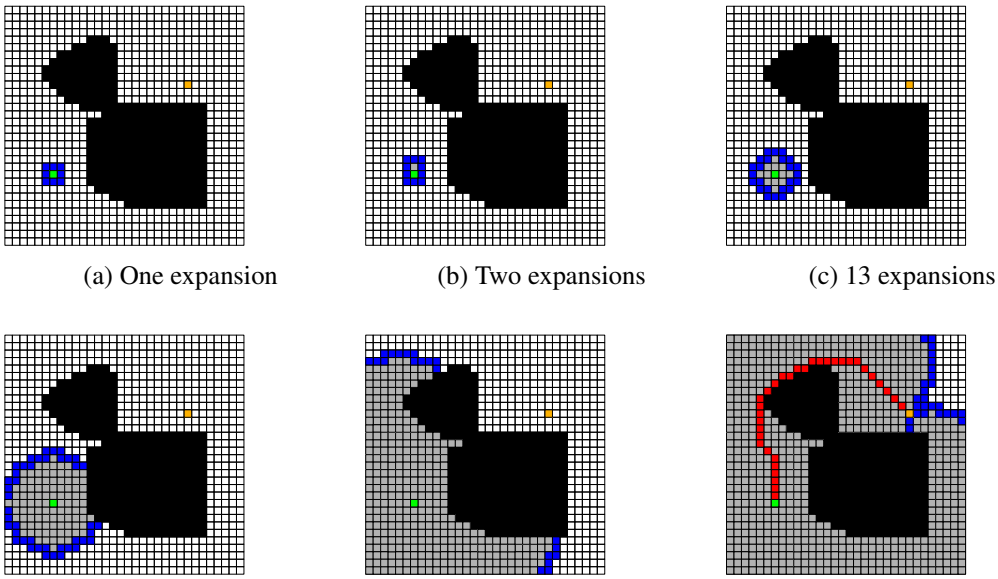
```
class Node:
    def __init__(self, state, parent_node, step_cost):
        self.state = state
        self.parent = parent_node
        self.path_cost = parent_node.path_cost + step_cost

def dijkstra(problem):
    frontier = PriorityQueue()
    closed = set()

    start_node = Node(problem.start(), None, 0.0)
    frontier.add(start_node, 0.0)

    while not frontier.is_empty():
        cur_node = frontier.pop()
        cur_state = cur_node.state
        closed.add(cur_state)

        if cur_state == problem.goal():
            return construct_path(cur_node)
        else:
            for next_state in problem.successors(cur_state):
                cost = problem.cost(cur_state, next_state)
                next_node = Node(next_state, cur_node, cost)
                if next_state not in closed:
                    frontier.add(next_node, next_node.path_cost)
    return None
```

Listing 3: Dijkstra's algorithm. Steps that differ from Listing 2 are highlighted. The key difference is that the frontier is represented by a Priority Queue that is ordered by the total cost to reach the state.

In many path-planning problems there is a notion of path cost that is distinct from the number of edges in the path. In the case of 2D navigation, the cost of a path might be the total distance traveled, the time required to reach the goal, or the amount of energy expended by the robot.

For example, if we want to find shortest paths in the grid navigation problem from Figure 3.6 we should assign weights to the edges in proportion to the distance that the robot must travel to move from cell to cell. This means that diagonal steps must have a higher weight to reflect the fact that the robot moves farther. An optimal solution to this weighted version of the problem will represent the shortest path in terms of distance traveled. Notice that the path discovered by BFS in Figure 3.7 is optimal in terms of the number of steps taken, but it is clearly not optimal in terms of distance.

With some slight modification, the algorithm in Listing 2 can be updated to take path costs into account. Listing 3 shows the updated algorithm. The main differences are that the Node implementation has been updated to store the total path cost required to reach the corresponding state, and the frontier is now represented by a Priority Queue ordered by path cost. The Priority Queue ensures that Nodes are



(a) One expansion (b) Two expansions (c) 13 expansions

(d) 100 expansions (e) 350 expansions (f) 674 expansion

Figure 3.9: Dijkstra's algorithm search example

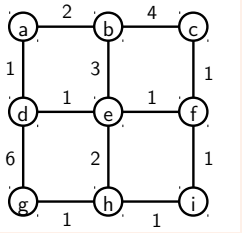
expanded in strictly increasing order of path cost. Given that all edges must have positive weights, each time an expansion occurs, all of the nodes added back to the frontier are guaranteed to have a higher cost than the node that was expanded. This means that when a node representing the goal state is expanded it must represent the lowest-cost path to the goal.

Notice that the algorithm in Listing 3 does not check if a state is already on the frontier before adding the new node for that state. From the point of view of correctness, this is fine: There is no danger of finding a longer path to the goal before a shorter path is discovered since the lower-cost node will be expanded first. However, it is more efficient to implement the frontier so that any higher-cost node is replaced when a lower-cost node is added for the same state. This prevents the wasted effort of expanding a higher-cost node (representing a longer path to the same state) after a lower-cost node.

Figure 3.9 shows an example of applying Dijkstra's algorithm to the triangle navigation problem. The resulting path is optimal for the given discretization.

Stop and Think

3.9 Consider the following weighted graph:



Complete the table below to show the state of Dijkstra's algorithm after each expansion. Again the start state is g and the goal state is a. Break priority ties using alphabetical order. I.e. in the case of a tie, state a will selected before state b. The tuples in the Frontier column represent Node objects where the first entry is the state, the second entry is the state associated with the parent node and the third entry is the path cost to reach the state. The first three steps are completed for you.

Expansions	Chosen	Frontier	Closed Set
0	–	$\langle g,-,0\rangle$	–
1	g	$\langle d,g,6\rangle$, $\langle h,g,1\rangle$	$\{g\}$
3	h	$\langle d,g,6\rangle$, $\langle e,h,3\rangle$, $\langle i,h,2\rangle$	$\{g,h\}$
3			
4			
5			
6			
7			
8			

What final path is returned? What is the path cost? ■

3.10 Compare 3.7 to 3.9. Why is the frontier more or less square in one and round in the other? ■

3.11 Take a look at the true, non-discretized, configuration space from Figure 3.1. What would the shortest path look like in that space? Why is it different from the path shown in figure 3.9? ■

3.12 Is it guaranteed that there will always be a unique minimum-cost path? If not, how could the algorithm in Listing 3 be updated to return all such paths? ■

3.3.3 A*

All of the graph search algorithms discussed so far can be described as **undirected**. This means that they "blindly" work their way out from the start state until the goal state is encountered. Intuitively, it seems that we should be able to search more quickly by preferring to expand nodes that are likely to be closer to the goal: If we know that the goal is to the east of the robot, then surely we should start the search by exploring actions that move the robot in that direction.

This intuition can be formalized through the idea of a **heuristic function** $h(s)$. A heuristic function maps from a state to the estimated cost of reaching the goal from that state. States with lower heuristic values are believed to be closer to the goal, and thus should be explored before states with higher heuristic values.

We can easily update Dijkstra's algorithm to take advantage this idea. The only change required is that the the Priority Queue representing the frontier will now be ordered by $f(s) = c(s) + h(s)$ where $c(s)$ represents the cost of reaching state s, and $h(s)$ represents the estimated cost of reaching the goal from s. The sum represents an estimate of the overall cost of the shortest path that passes through state s.

The resulting Algorithm is called A*.

The A* algorithm is guaranteed to find an optimal path as long as the heuristic function satisfies two requirements: First, it must never overestimate the true cost of reaching the goal. An overestimate could prevent a node from being expanded, even if that node is on the shortest path to the goal. Second, the heuristic function must be **consistent**. Formally, this means that $h(s) \leq h(s') + cost(s,s')$ for all states s and s' where $cost(s,s')$ represents the cost of the edge between s and s'. Informally, this means that the heuristic function should respect the actual step costs. When stepping from s to s', the heuristic function shouldn't drop by more than the cost of the step.

The "*" in A* was chosen to indicate that this is, in a sense, the last word in heuristic-based graph search algorithms. Not only is A* guaranteed to find a shortest path, it is also provably optimal in terms of the number of nodes expanded. Any algorithm that is guaranteed to find a shortest path must expand at least as many nodes as A* when using the same heuristic function.

Taking full advantage of A* requires the selection of a good heuristic function. A good heuristic function should be as close as possible to the true cost of reaching the goal, while being fast to compute and satisfying the consistency requirement. In general, it can be challenging to satisfy these requirements. However, in robotic path planning problems where the objective is to find a minimum length path, the straight-line distance to the goal is often a reasonable choice. The straight-line distance is easy to compute. It may underestimate the true cost, (because it doesn't take obstacles into account) but it can never overestimate: the straight-line distance always represents the shortest possible path to the goal. If the step cost is taken to be the distance traveled, then the straight line distance is consistent because no step can reduce the distance to the goal by more than the distance moved.

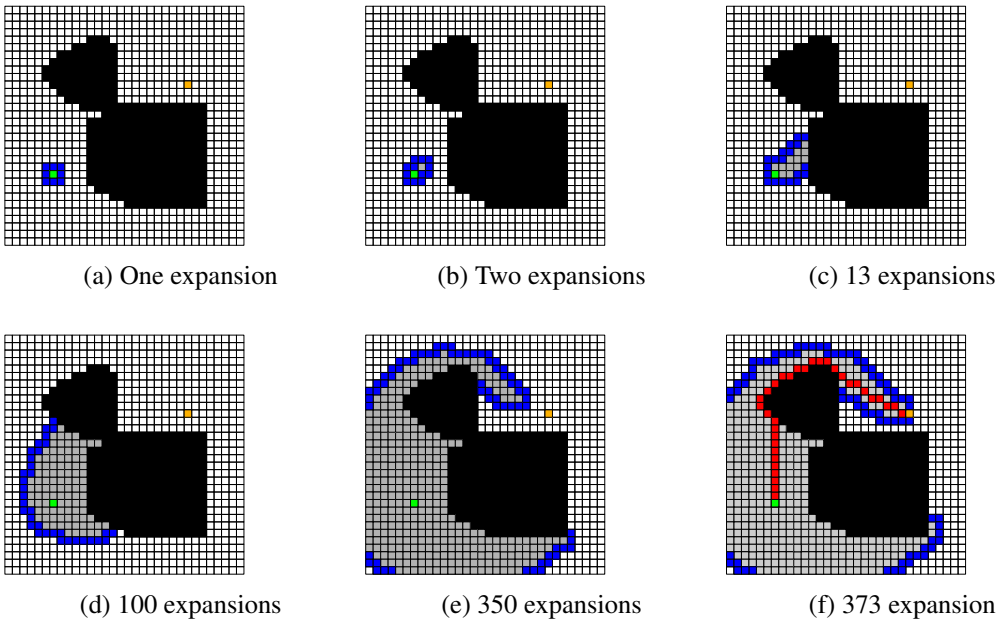
Figure 3.10 illustrates an A* search using the straight line heuristic. Notice that, while A* does not return the same path that was discovered by Dijkstra's algorithm, the two paths have the same length and both are optimal.

Stop and Think

3.13 The **Manhattan distance** is a count of the minimum number of edges between two states in a four-connected grid. Is Manhattan distance an admissible heuristic for the search problem in Question 3.9? ■

3.14 Repeat Question 3.9 using an A* search. Use the Manhattan distance to the goal as the heuristic function. For this problem, the fourth value in the Frontier tuples will represent $f(s) = c(s) + h(s)$. The first three expansions are done for you.

Expansions	Chosen	Frontier	Closed Set
0	–	$\langle g,-,0,2\rangle$	–
1	g	$\langle d,g,6,7\rangle$, $\langle h,g,1,4\rangle$	$\{g\}$
3	h	$\langle d,g,6,7\rangle$, $\langle e,h,3,5\rangle$, $\langle i,h,2,6\rangle$	$\{g,h\}$
3			
4			
5			



(a) One expansion (b) Two expansions (c) 13 expansions

(d) 100 expansions (e) 350 expansions (f) 373 expansion

Figure 3.10: A* search example.

■

3.15 In comparing Figure 3.10 to Figure 3.9 we can see that A* does appear to be "smarter". It tends to expand nodes that are closer to the goal before nodes that are farther away. On the other hand A* does expand some nodes that are in the "wrong" direction. For example, by expansion 100 it has expanded several nodes below and to the left of the start state, even those they are in the opposite direction from the goal. What's going on? Why doesn't A* always expand the node that is closest to the goal? ■

3.16 Imagine we want to encourage A* to avoid taking steps that move the robot close to an obstacle. We could attempt to accomplish this by adding 5 to the heuristic function for every state that is adjacent to an obstacle. Is it possible for the resulting heuristic to overestimate the cost to goal? Is the resulting heuristic consistent? Can you think of a better approach achieving the desired outcome? ■

3.17 Consider the heuristic function $h(s) = 0$ for all states s. Does this heuristic ever over-estimate the true cost? Is it consistent? Is it a useful heuristic? Why or why not? ■

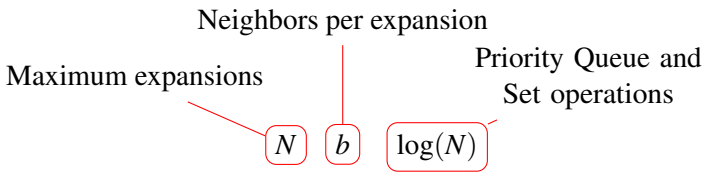
3.3.4 The Efficiency of Grid-Based Search

The worst-case time complexity of both Dijkstra's algorithm and A* is usually described as $O(d^b)$ where d is the number of steps to the goal and b is the **branching factor** – the average number of

Dimensions (d)	Branching Factor (b)	Number of States (N)
1	$3^1 - 1 = 2$	$100^1 = 100$
2	$3^2 - 1 = 8$	$100^2 = 10{,}000$
3	$3^3 - 1 = 26$	$100^3 = 1{,}000{,}000$
4	$3^4 - 1 = 80$	$100^4 = 100{,}000{,}000$
5	$3^5 - 1 = 242$	$100^5 = 10{,}000{,}000{,}000$
6	$3^6 - 1 = 728$	$100^6 = 1{,}000{,}000{,}000{,}000$

Table 3.1: Branching factor and state space size as a function of dimensionality

successors for each state. This is a valid upper-bound, but it can give a dramatic over-estimate for problems where the pruning allowed by closed the set allows us to avoid considering many alternative paths that pass through the same state. When using a closed set, the total number of expansions is, at most, equal to the number of states N, since each state can be expanded at most once. Therefore, for problems with a finite number of state, a tighter bound is $O(Nb\ \log(N)))$:



This means that both Dijkstra's algorithm and A* are actually polynomial-time algorithms as a function of the number of states. The bad news is that, for grid-based discretizations of a configuration space, the number of states and the branching factor both grow exponentially with the number of dimensions in the problem.

Table 3.1 illustrates a grid-based discretization where each dimension is divided into 100 equal increments and each grid cell is connected to its immediate neighbors. The take-home message from this table is that grid-based search works well for two-dimensional search problems, but quickly becomes impractical as the number of dimensions increases. This is a significant issue, given that practical search problems can easily have six dimensions or more. The remainder of this chapter will consider alternatives to grid-based search that are suitable for high-dimensional path planning.

3.4 Sampling-Based Search

Sampling-based search algorithms abandon the completeness and optimally guarantees of grid-based search in exchange for better performance in high-dimensional configuration spaces. Instead of creating a dense discretization, sampling based algorithm create a sparse web of nodes by placing edges between randomly generated configurations.

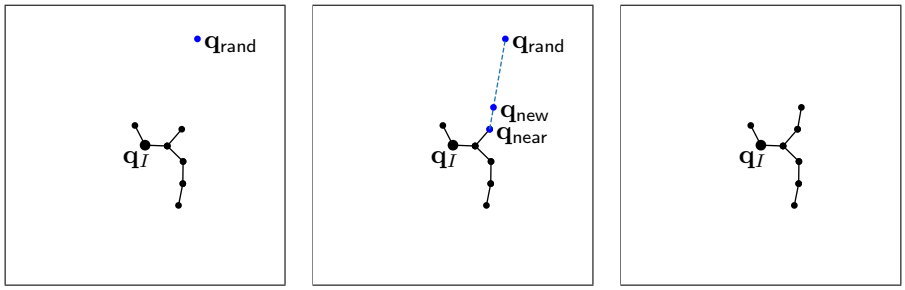


Figure 3.11: One iteration of the rapidly exploring random tree algorithm. Left: a random configuration $\mathbf{q}_{\text{rand}}$ is selected. Center: the nearest node in the existing tree $\mathbf{q}_{\text{near}}$ is selected for expansion. A new node $\mathbf{q}_{\text{new}}$ is created by running a local planner to find an action that moves in the direction of the random node. Right: The new node is added to the tree.

3.4.1 Rapidly Exploring Random Trees

The Rapidly Exploring Random Tree (RRT) algorithm is an undirected search algorithm that incrementally builds a tree outward from the starting configuration until the goal configuration is reached. Figure 3.15 illustrates one iteration of RRT. First, a configuration $\mathbf{q}_{\text{rand}}$ is sampled from the full configuration space. Next, we select the node in the tree $\mathbf{q}_{\text{near}}$ that is nearest to the randomly generated point for expansion. Finally, we add a node $\mathbf{q}_{\text{new}}$ to the tree by taking a step from $\mathbf{q}_{\text{near}}$ in the direction of $\mathbf{q}_{\text{rand}}$. The full algorithm is outlined in Listing 4.

The basic algorithm presented in Listing 4 only handles the construction of the tree. The algorithm can be augmented to return a path by adding a check to see if newly created nodes are close enough to the goal configuration to allow a connection.

```python
def rrt(problem, q_init, tree_size):
    """ Build a rapidly exploring random tree.

    Args:
        problem:   a problem instance that provides three methods:

            problem.random_state() -
               Return a randomly generated configuration
            problem.select_input(q_rand, q_near) -
               Return an action that would move the robot from
               q_near in the direction of q_rand
            problem.new_state(q_near, u) -
               Return the state that would result from taking
               action u in state q_near

        q_init:     the initial state
        tree_size:  the number of nodes to add to the tree

    Returns:
        A tree of configurations rooted at q_init

    """
    tree = Tree(q_init)  # Make the start state the root of the tree
    while tree.num_nodes() < tree_size:
        q_rand = problem.random_state()
        q_near = nearest_neighbor(tree, q_rand)
        u = problem.select_input(q_rand, q_near)
        q_new = problem.new_state(q_near, u)
        tree.add_node(q_new):
        tree.add_edge(q_near, q_new)
    return tree
```

Listing 4: The Rapidly Exploring Random Tree (RRT) algorithm.

The strength of the RRT algorithm lies in its tendency to quickly move away from the initial configuration into unexplored areas of the configuration space. The Voronoi diagram in Figure 3.12 provides an intuition for this behavior. A Voronoi diagram partitions space into regions based on the nearest neighbor boundaries. As can be seen in Figure 3.12, nodes at the edge of the tree have much larger Voronoi regions than nodes on the interior. This means that a randomly generated configuration is more likely to land within the Voronoi region of one of these edge nodes. This Voronoi bias tends to pull the tree into unexplored areas of the configuration space.

Figure 3.13 illustrates the progress of the RRT algorithm on the triangle robot example. Notice that the final path is clearly not optimal. In fact the RRT algorithm is neither optimal nor complete. It is, however, **probabilistically complete**: if a path to the goal exists, the probability of finding a path approaches one as the number of iterations approaches infinity.

In practice, RRT planners typically perform some post-processing to smooth out paths once they have been discovered. This can be done, for example, by selecting non-neighboring points on the path and

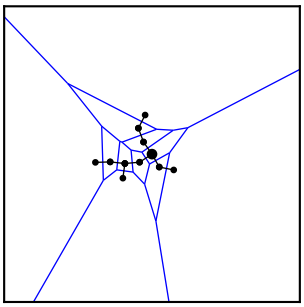


Figure 3.12: Voronoi diagram for a partially constructed random tree. The nodes at the edge of the tree have larger Voronoi regions, and so are more likely to be expanded.

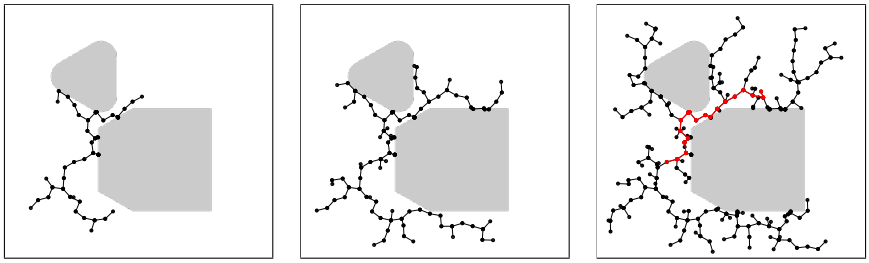


Figure 3.13: Example of applying the RRT algorithm to the triangle robot navigation problem from Figure 3.4. The final path is shown in red in the right-most figure.

attempting to find direct connections between them.

RRT's for Non-Holonomic Path Planning

The Rapidly Exploring Random Tree algorithm is well suited to non-holonomic path-planning problems. For the holonomic example in Figure 3.13, `select_input` simply creates a short line segment from $\mathbf{q}_{\text{near}}$ in the direction of $\mathbf{q}_{\text{rand}}$. In the case of non-holonomic planning, `new_state` and `select_input` must be implemented to respect the non-holonomic constraints.

Figure 3.14 shows an example of RRT path planning for a non-holonomic wheeled vehicle that may only move forward and has a maximum turning radius. In the initial configuration, the robot is located inside a narrow passage and is pointed away from the goal location. The RRT algorithm works outward from the initial configuration to the goal in steps that respect the non-holonomic constraints.

Stop and Think

3.18 Take a moment to examine the RRT algorithm in Listing 4. Which steps in the algorithm do you expect to be most computationally expensive? Why? ■

3.19 The basic RRT algorithm in Listing 4 is completely undirected. A common optimization is to bias search in the direction of the goal by selecting the goal configuration as $\mathbf{q}_{\text{rand}}$ with some small

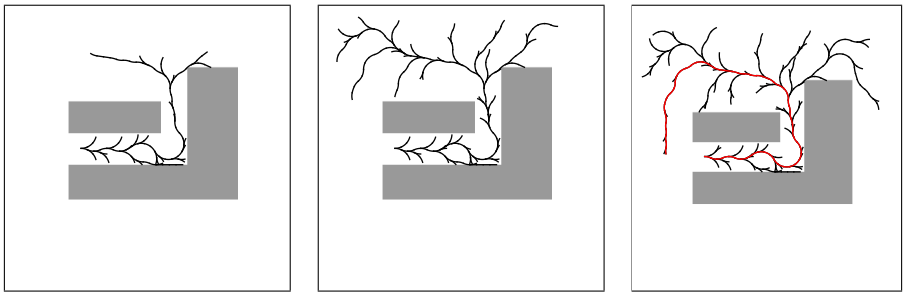


Figure 3.14: Example of applying the RRT algorithm to a non-holonomic path-planning problem.

probability. Why not select the goal configuration 100% of the time? ■

3.4.2 Probabilistic Roadmaps

One weakness of the RRT algorithm is that it must explore the configuration space from scratch for each search. In domains that involve repeated searches within the same environment, it may be more efficient to pre-compute a graph that maps the connectivity of $\mathcal{C}_{free}$. The Probabilistic Road Map (PRM) algorithm takes this approach. The algorithm proceeds by generating random configurations from $\mathcal{C}_{free}$ and attempting to connect those configurations to all existing configurations within some fixed radius. Listing 4 provides the complete algorithm and Figure 3.15 illustrates one iteration.

```
def prm(problem, delta, roadmap_size):
    """ Create a Probabilistic Roadmap.

    Args:
        problem:  a problem instance that provides two methods:

           problem.random_state() -
                Return a random state from C_free
           problem.no_collision(q1, q2) -
                Return True if there is a collision free path
                from q1 to q2

        delta: Distance threshold for connecting neighboring states
        roadmap_size: Number of nodes in the completed Roadmap

    Returns:
        A graph representing a Probabilistic Roadmap

    """
    graph = Graph()
    while graph.num_nodes() < roadmap_size:
        q_rand = problem.random_state()
        graph.add_node(q_rand)
        for q in neighbors(graph, q_rand, delta):
            if problem.no_collision(q, q_rand):
                graph.add_edge(q, q_rand)
    return graph
```

Listing 5: The Probabilistic Road Map Algorithm (PRM).

Once a roadmap has been created it can be used for path-planning by adding the start and goal configurations to the graph and using any of the graph search algorithms from Section 3.3.

The PRM algorithm is not well suited to non-holonomic path planning problems. The PRM graph must be undirected to facilitate planning from an initially unknown start configuration to an unknown goal configuration. As with the wheeled vehicle example from Figure 3.14, non-holonomic domains often do not allow for "reversible" actions of the type that can be represented in an undirected graph.

Stop and Think

3.20 What is the difference between implementing `problem.random_state` for use with the PRM algorithm vs. RRT? ■

3.21 The original formulation of the PRM algorithm only added edges between nodes that were not already in the same connected component. How would this impact the probability of finding a path in the resulting map? How would this impact the quality of the discovered paths? ■

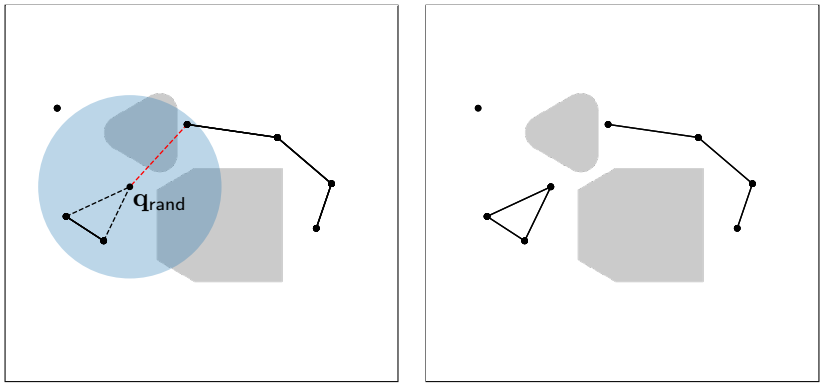


Figure 3.15: One iteration of the probabilistic Roadmap algorithm. The figure on the left shows the randomly generated configuration $\mathbf{q}_{\text{rand}}$ connected by dotted lines to the existing points that are within range. The dotted red line will not be added to the roadmap because it passes through an obstacle. The figure on the right shows the roadmap after $\mathbf{q}_{\text{rand}}$ and the appropriate edges have been added.

3.5 References and Further Reading

Discrete-space planning algorithms are a foundational idea in artificial intelligence and are covered in AI textbooks including (Russell & Norvig, 2010) and (Poole & Mackworth, 2017). The A* algorithm was invented at SRI in the 1960s for use with Shakey the robot. The Shakey project was an ambitious early effort in autonomous robotics that resulted in key innovations in planning, navigation, and computer vision. Nils J. Nilsson recounts the history of that project, along with a broader history of early research in artificial intelligence in (Nilsson, 2009). The A* algorithm was formally introduced, and proven to be optimal, in (Hart et al., 1968).

The use of configuration spaces as a formalism in planning was introduced in (Lozano-Perez, 1983). Steven LaValle's *Planning Algorithms* textbook provides a thorough introduction to configuration spaces and provides in-depth descriptions of each of the planning algorithms discussed in this chapter. (LaValle, 2006).

The rapidly exploring random tree algorithm was invented by Steven LaValle (LaValle, 1998). Many variations have been published that improve or modify some aspect of the algorithm's behavior. One notable example is the RRT* algorithm, which dynamically "re-wires" the search tree in order to make local improvements when new nodes are added (Karaman & Frazzoli, 2010). While the original RRT algorithm provides no guarantees about the quality of the final solution, it can be shown that the RRT* algorithm converges on the lowest-cost path as the number of samples approaches infinity.

The probabilistic roadmap algorithm was introduced in (Kavraki et al., 1996). As with RRT, PRM has served as the basis for many variations in the planning literature.

The Open Motion Planning Library (OMPL) https://ompl.kavrakilab.org/ is an open source library that includes high-quality implementations of sampling based planning algorithms including RRT, RRT*, PRM and many others.

4. Probabilistic Localization

4.1 Introduction

In previous chapters we assumed that the we had access to accurate information about the state of the world. For example, in Chapter 1 we developed a closed-loop controller for moving a robot to a goal by repeatedly comparing the robot's current location to the goal location. This raises the question of how we can know the exact location of the robot. The short answer is that we can't. It is possible to estimate the robot's location by using two general sources of information:

- **Sensors** - There are a wide range of sensors that can provide information about the position of a robot. Cameras may be used to detect landmarks. Depth sensors may be used to estimate the robot's position in a map. Bump sensors may be used to determine when the robot is in contact with an obstacle.
- **Dead reckoning** - Assuming the initial location of a robot is known, it's location at some later time can be estimated by considering the control signals that have been applied. If we send commands telling the robot to move forward at 1m/s for 1s, the robot should end up one meter ahead of the initial location.

There is inherent uncertainty associated with each of these sources of information. No sensor is perfect. Dead reckoning is never perfectly reliable. The goal of this chapter is to introduce a probabilistic framework that will make it possible to represent and reason about this uncertainty.

For the sake of concreteness, this chapter will focus on probabilistic representations of a robot's location, but the same mathematical tools are useful for representing any uncertain information.

4.2 Discrete Probability Distributions

A **discrete random variable**, usually expressed as an upper-case letter such as X, is a variable that can take on a fixed number of possible values. A **probability distribution**, also called a **probability mass function**, is a function that maps from each possible value of the random variable to its probability.

As a simple example, consider a Boolean random variable H that describes the possible outcomes of flipping a coin. In this case, the possible values for H are $True$ indicating that the coin landed on heads or $False$ indicating that the coin landed on tails. The probability mass function for a fair coin is then

$$P(H = True) = .5$$
$$P(H = False) = .5$$

In the case of Boolean variables we often use the more concise convention of indicating an assignment of true using a lower case variable, so that $P(H = True)$ could be expressed as $P(h)$ and $P(H = False)$ could be expressed as $P(\neg h)$.

A valid probability mass function must satisfy the following conditions:

- Probabilities must not be negative or greater than one:

$$0 \leq P(X = x_i) \leq 1 \text{ for all } x_i$$

- The probability mass function must sum to one:

$$\sum_{x_i} P(X = x_i) = 1$$

Discrete random variables need not be Boolean-valued. In particular, a random variable may be used to describe our uncertainty about the location of a robot, with each possible robot location corresponding to a value for the random variable, and the probability distribution describing our beliefs about which location is correct. For example, consider the problem of tracking the location of a robot in a one-dimensional world (perhaps our autonomous locomotive from Chapter 1). Figure 4.1 illustrates the idea. The horizontal location of each bar corresponds to a discretized value for the position variable, while the height of the bars correspond to the value of the probability mass function for that position. Notice that the height of all bars must always sum to one, since we know that the robot must be located at some location. The same information may also be presented in tabular form as illustrated in Figure 4.2.

In a more realistic scenario, the values for our random variable could be entries in a grid corresponding to possible locations in a two-dimensional workspace. This idea is illustrated in Figure 4.3.

4.2.1 Joint Probabilities

Probabilistic models of complex systems generally involve multiple, interacting, random variables. The multivariate generalization of the probability distribution is the **joint probability distribution**. A joint probability distribution maps from every possible outcome of all variables to the probability for that set of assignments. For example, In the case of our 1-d robot above, we can introduce a second random

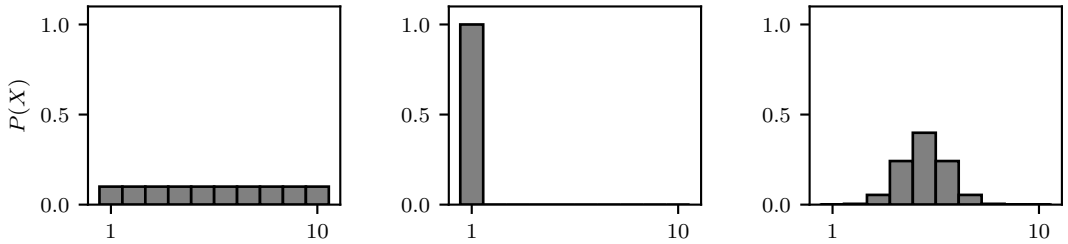


Figure 4.1: Sample histogram probability distributions describing our belief about the location of a robot in a one-dimensional environment. (a) Uniform distribution representing a complete lack of knowledge about the location of the robot. (b) Certain knowledge that the robot is in location 1. (c) Belief that the robot is most likely to be in location 5, but may be in nearby locations to the left or right.

X	P(X)
1	.1
2	.1
3	.1
4	.1
5	.1
6	.1
7	.1
8	.1
9	.1
10	.1

(a)

X	P(X)
1	1.0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0

(b)

X	P(X)
1	0
2	.004
3	.054
4	.242
5	.400
6	.242
7	.054
8	.004
9	0
10	0

(c)

Figure 4.2: Tabular representations of the probability distributions illustrated in Figure 4.1. Notice that the rows sum to one in each table.

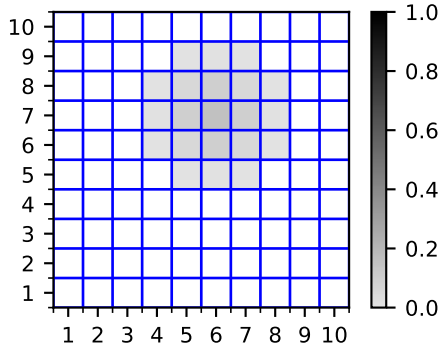


Figure 4.3: Sample two-dimensional probability distribution. In this example, shading is used to indicate the probability of the robot being located in a particular grid cell.

variable representing the output of a wall detection sensor that is designed to tell us when we are near one of the two boundaries of the hallway (state 1 or state 10). This is activated with probability .8 when the robot is at either end of the hallway, .1 when it is one step away from either end and 0 for all other locations.

In the case where we know nothing about the location of the robot, our joint probability distribution will look like the following for our location/sensor scenario:

X	Z	P(X,Z)
1	$beep$	.08
1	$\neg beep$	.02
2	$beep$	.01
2	$\neg beep$	.09
3	$beep$	0
3	$\neg beep$	.1
...		
8	$beep$	0
8	$\neg beep$	.1
9	$beep$	.01
9	$\neg beep$	.09
10	$beep$	.08
10	$\neg beep$	.02

Table 4.1: Joint probability distribution for robot position X and wall sensor output Z.

Marginalization

Given the full joint distribution we can always recover the probability distribution for an individual variable through **marginalization**, or summing out. In the general case, this can be expressed as:

$$P(A = a) = \sum_{b \in B} P(A = a,\ b) \tag{4.1}$$

This means that if we want to calculate the probability of some assignment to A using the joint distribution, we just need to sum up all of the rows in the table that match that assignment to A. For example, given our location/sensor example above, we can recover the probability that the robot is at position 2 as follows:

$$\begin{aligned} P(X = 2) &= P(X = 2,\ beep) + P(X = 2,\ \neg beep) \\ &= .01 + .09 \\ &= .1 \end{aligned}$$

Stop and Think

4.1 Based on the joint probability distribution in Table 4.1, what is $P(beep)$? What is $P(\neg beep)$? ■

Independence

Two random variables are defined to be **independent** if and only if

$$P(A \cap B) = P(A)P(B). \tag{4.2}$$

Intuitively, A and B are independent if knowing the value of A provides no information about the value of B. For example, whether or not a region will experience an earthquake on a particular day is independent of the occurrence of a tornado: there is no reason to believe that one will make the other more or less likely. On the other hand, whether a region will experience an earthquake on a given day is *not* independent of the possibility of a tsunami: earthquakes can cause tsunamis, so knowing that one has occurred increases the probability of the other.

The notion of independence is important for probabilistic reasoning. In the example from the previous section, we saw a joint probability distribution with two random variables. In general, applications involving probabilistic reasoning may involve *many* random variables. For example, consider a robot navigating through an office building with 20 doors that may each be open or closed. Assuming that the robot is unable to open doors, path planning in this environment will require reasoning about the possible states of all doors. In order to write down the full joint probability distribution describing every possible combination of closed and open doors, we would need a table with $2^{20} \approx 1,000,000$ rows. It is much more efficient, in terms of both space and computation, to make the assumption that the state of each door is independent of the state of every other door. In that case we only need to store 20 individual probability distributions with two rows each. We can then reconstruct the probability of any combination of closed and open doors by simply multiplying together the appropriate 20 probabilities.

Note that **independence assumptions** of this sort may be useful even if they aren't entirely correct. In a real office building it is unlikely that the states of individual doors will be entirely independent. If the doorway to a suite of offices is open, there is probably someone working in that suite, which would make it more likely that the individual office doors will be open as well. This is a case where it is necessary to settle for an approximately correct answer in the interest of computational tractability.

Stop and Think

4.2 The probability that it will rain today, $P(R)$ is independent of the probability of an earthquake, $P(E)$. Assuming $P(r) = .7$ and $P(e) = .001$, what is the probability that it will be a rainy day with no earthquake? $P(r, \neg e) = ??$ ■

4.2.2 Conditional Probabilities

Probability distributions like $P(X)$ are sometimes described as **prior probabilities** because they describe our initial belief about X before we have taken any evidence into account. **Conditional probabilities** allow us to express the probability for one random variable, given that we know the value of some other random variable. Conditional probability is defined as follows:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} \tag{4.3}$$

Conditional probabilities provide a useful way of thinking about the relationship between a robot's state and sensor data. For robot localization, some variables represent unknown state information that we are attempting to estimate. This unknown state information is commonly denoted X. On the other hand, some variables represent *known* values received from a sensor. These values are commonly denoted Z. For the purpose of localization, it would be useful to have access to the conditional probability distribution $P(X \mid Z)$. This is exactly the distribution over X given our known sensor value Z.

As an example of conditional probability, consider the case of a cliff detection sensor designed to prevent a home-vacuuming robot from falling down stairways. In this case, S is a Boolean random variable indicating whether a stairway is actually present, and the variable Z is true if the cliff-detection sensor has activated. The following table describes the full joint probability distribution:

S	Z	$P(S,Z)$
T	T	.0495
T	F	.0005
F	T	.095
F	F	.855

Using this table, along with Equations 4.1 and 4.3, we can calculate, for example, the false positive rate of our sensor: $P(Z = True \mid S = False)$. In other words, what is the probability that the sensor will indicate that a stairway is present when it is not.

$$
\begin{aligned}
P(Z = True \mid S = False) &= \frac{P(Z = True \cap S = False)}{P(S = False)} \\
&= \frac{.095}{P(S = False)} && \text{(From the third row of the table above)} \\
&= \frac{.095}{P(S = False, Z = True) + P(S = False, Z = False)} && \text{(Marginalization)} \\
&= \frac{.095}{.095 + .855} = .1
\end{aligned}
$$

Stop and Think

4.3 Look back at table 4.1. Apply definition 4.3 to calculate $P(X = 0 \mid Z = beep)$. ■

Conditional Independence

Section 4.2.1 introduced the idea of independent random variables. Now that we have an understanding of conditional distributions we can introduce a second notion of independence that is also a valuable tool in probabilistic reasoning.

Conditional independence is defined as follows:

A random variable A is conditionally independent of C given B if and only if

$$P(A \mid B, C) = P(A \mid B). \tag{4.4}$$

Intuitively, this means that, if we already know B, then learning something about C doesn't tell us anything additional about A. This is a bit more subtle than the definition of independence introduced earlier. Stating that A is conditionally independent from C given B is not the same as saying that any two of those variables are independent. As an example, consider the following scenario:

- A: The east door to the mall is locked
- B: The mall is open for business
- C: The west door to the mall is locked

In this case, A and C are not independent. If I learn that one door is locked, that increases my belief that the other will be locked as well. However, they *are* conditionally independent given B: if I already know whether or not the mall is open, then learning the state of the west door doesn't give me any additional information about east door.

4.2.3 Bayes' Rule

Unfortunately definition (4.3) often isn't useful for determining $P(X \mid Z)$ because it requires us to have access to the full joint probability distribution $P(X,Z)$. In practice, that full distribution is usually not easy to obtain in an explicit form. **Bayes' rule** or **Bayes' theorem** is tremendously useful here. Baye's rule can be expressed as follows:

$$P(X \mid Z) = \frac{P(Z \mid X)P(X)}{P(Z)} \tag{4.5}$$

Bayes' rule provides a recipe for taking the prior probability $P(X)$ that describes our initial beliefs about the robot's state and calculating a **posterior probability** $P(X \mid Z)$ that describes our updated belief after into account information from a sensor reading.

In order to apply Bayes' rule, we need to know two things:

- $P(Z \mid X)$ - The conditional probability of sensor readings given state information. This is often described as the **sensor model**. It simply captures what we know about how our sensor works.
- $P(X)$ - The prior distribution over our state variable.

It turns out we don't need to know $P(Z)$ explicitly. It can be calculated from $P(Z \mid X)$ and $P(X)$ using the **total probability formula**:

$$P(Z) = \sum_{x \in X} P(Z \mid X = x)P(X = x) \tag{4.6}$$

Alternatively, it is common to to replace $P(Z)$ using a **normalizing constant** α defined as

$$\alpha = \frac{1}{P(Z)}.$$

This results in the following alternate form of Bayes' rule:

$$P(X \mid Z) = \alpha P(Z \mid X)P(X) \tag{4.7}$$

This formulation takes advantage of the fact that we know that $P(Z)$ is the same for every state. We can simply solve for the value of α that makes the posterior probability sum to one.

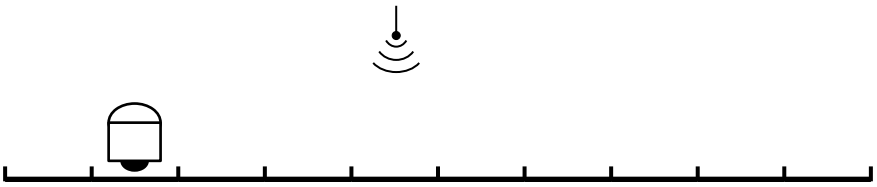


Figure 4.4: A scenario with ten discrete robot locations. There is a radio beacon at location 5.

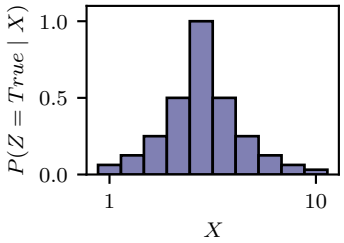


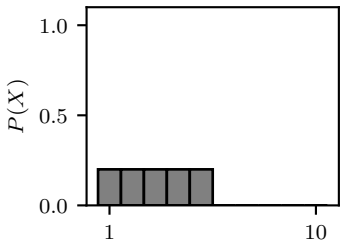
Figure 4.5: Sensor model for our radio beacon detector. The probability that the sensor will activate falls off as the robot moves further away from the beacon. Notice that this figure is *not* illustrating a probability distribution of the type shown in Figure 4.1. In this case the probabilities don't need to sum to one.

Bayes' Rule Example

Consider another one-dimensional robot that may be in one of ten discrete locations. In this case, there is a radio beacon located at location **5** and the robot has a sensor that detects the beacon with a probability that is related to distance. The scenario is illustrated in Figure 4.4.

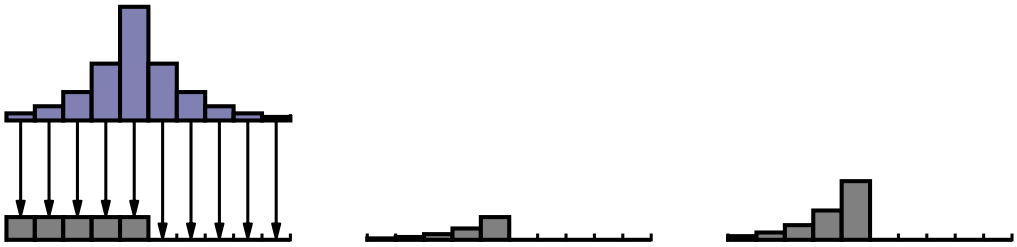
When the robot is in the same location as the beacon, it will be detected with a probability of 1. When the robot is one step away, it will be detected with a probability of .5. The probability of detection continues to drop off at a rate of 50% for each step away from the beacon. Figure 4.5 illustrates this sensor model.

In this scenario, Bayes' rule allows us to update our belief about where the robot is located based on the signal received from the radio sensor. Let's assume that our prior probability distribution over the robot's location looks like this:



This prior distribution $P(X)$ reflects that robot is equally likely to be in any of the five locations on the left, with a .2 probability of being in each.

Now assume that the robot takes a reading from its beacon-detector and the value is *True*. Figure 4.6 illustrates the steps of applying Bayes' rule to calculate the posterior distribution. First, the prior probability associated with each state $P(x_i)$ is multiplied by the value of the sensor model at that state



(a) (b) (c)

Figure 4.6: Using Bayes' rule to update state estimates based on a sensor reading. (a) The bar graph on the top represents the sensor model $P(Z = True \mid X)$. The bar chart on the bottom represents the prior distribution over the robot's location $P(x)$ The vertical arrows represent multiplication: the value of the sensor model at each state is multiplied by the prior probability for the corresponding state. (b) The resulting products. Note that these values do *not* sum to one. (c) The values at each state re-scaled (or **normalized**) to make the total sum to 1. The result is the posterior probability distribution $P(X \mid Z)$

$P(Z = True \mid X = x_i)$. The resulting values are then re-scaled to sum to one.

Let's do a detailed run-through through of the example in Figure 4.6 to see how the steps follow from the application of Bayes' rule. Bayes' rule may be used to calculate the posterior probability that the robot is in any particular state. For example, calculating the posterior for state 1 looks like the following:

$$P(X = 1 \mid Z = True) = \alpha P(Z = True \mid X = 1)P(X = 1)$$

Substituting $P(Z = True \mid X = 1) = 0.0625$ from the sensor model and $P(X = 1) = .2$ from the prior distribution yields:

$$P(X = 1 \mid Z = True) = \alpha(0.0625 \times .2) = 0.0125\alpha$$

This calculation is repeated for all values of X:

$$\begin{aligned}
P(X = 1 \mid Z = True) &= \alpha(0.0625 \times .2) = 0.0125\alpha \\
P(X = 2 \mid Z = True) &= \alpha(0.0125 \times .2) = 0.025\alpha \\
P(X = 3 \mid Z = True) &= \alpha(0.25 \times .2) \quad = 0.05\alpha \\
P(X = 4 \mid Z = True) &= \alpha(0.5 \times .2) \quad = 0.1\alpha \\
P(X = 5 \mid Z = True) &= \alpha(1.0 \times .2) \quad = 0.2\alpha \\
P(X = 6 \mid Z = True) &= \alpha(0.5 \times 0) \quad = 0 \\
&\ldots
\end{aligned}$$

All that remains is to determine α by solving for the value that makes the total probability sum to 1:

$$0.0125\alpha + 0.025\alpha + 0.05\alpha + 0.1\alpha + 0.2\alpha = 1.0$$

$$\alpha \approx 2.581$$

Substituting 2.581 for α gives us the final posterior probability distribution across states:

$$
\begin{aligned}
P(X = 1 \mid Z = True) &= 0.0125\alpha \approx 0.0323 \\
P(X = 2 \mid Z = True) &= 0.025\alpha \approx 0.0645 \\
P(X = 3 \mid Z = True) &= 0.05\alpha \approx 0.1290 \\
P(X = 4 \mid Z = True) &= 0.1\alpha \approx 0.2581 \\
P(X = 5 \mid Z = True) &= 0.2\alpha \approx 0.5161 \\
P(X = 6 \mid Z = True) &= 0 \\
&\ldots
\end{aligned}
$$

4.3 Recursive State Estimation

We now have the tools to formalize the central computational problem raised in this Chapter. The fundamental goal in robot localization is to estimate a robot's position based on the full history of sensor readings and actions taken by the robot. This can be expressed as follows:

$$Bel(X_t) = P(X_t|U_0, Z_0, U_1, Z_1, \ldots, U_t, Z_t) \tag{4.8}$$

where the subscripts indicate discrete time intervals and the U variables represent the action choices made by the robot at each time step. According to this definition, the **belief state** $Bel(X_t)$ is defined to be conditional distribution over the robot's location at time step t given full history of actions and sensor readings leading up to time t.

We are already halfway to the solution. At the beginning of this Chapter we claimed that keeping track of a robot's location involves two sources of information: sensor data and dead reckoning. As we've seen above, Bayes' rule exactly solves the problem of incorporating sensor data to update our beliefs about the robot's location.

In order to incorporate dead reckoning we need a probabilistic model of how the robot's actions impact its state. This can be expressed with the following conditional probability distribution:

$$P(X_t \mid X_{t-1}, U_t)$$

This distribution is often described as the **motion model** for the robot. Notice that this formulation involves an implicit conditional independence assumption in that the state distribution at time step t only depends on the most recent state and action. Expressed formally, we are assuming that:

$$P(X_t \mid X_{t-1}, U_t) = P(X_t \mid X_{t-1}, U_t, X_{t-2}, U_{t-1}, \ldots, X_0, U_1) \tag{4.9}$$

This is called the **Markov assumption**. This assumption may be expressed in English as "The future is independent of the past given the present."

The Markov assumption is key to developing a tractable algorithm for tracking the belief state. At first glance, the formulation of the localization problem in Equation 4.8 should be worrying. The history of sensor readings and action choices will continue to grow without bound over time. This raises the concern that the computational cost of maintaining our belief state will also continue to grow as we

accumulate more and more history that needs to taken into account. The Markov assumption provides a way out of that dilemma.

The full **recursive state estimation** algorithm can be expressed as follows:

$$Bel^{-}(X_t) = \sum_{x_i \in X_{t-1}} P(X_t \mid X_{t-1} = x_i,\, U_t) Bel(X_{t-1} = x_i) \qquad \text{(Prediction)} \qquad (4.10)$$

$$Bel(X_t) = \alpha P(Z_t \mid X_{t-1}) Bel^{-}(X_t) \qquad \text{(Correction)} \qquad (4.11)$$

In the **prediction** stage, we update the belief state based on the robot's latest action by applying the total probability theorem (Equation 4.6) to the motion model to sum across all possible values for the previous state. The $^{-}$ superscript indicates that the result is the estimated belief state before information about any sensor readings has been incorporated. This is referred to as the "prediction" step because we are predicting where the robot is likely to end up based on the action it selected.

In the **correction** stage we apply Bayes' rule to update the belief state based on the latest sensor reading, exactly as was discussed in the previous section. This is referred to as the "correction" step because we are revising our prediction by incorporating the latest sensor information.

These two stages alternate indefinitely, with the belief state calculated at each time step serving as the prior belief state for the next time step. Assuming that the Markov assumption is correct, and that the motion and sensor models are accurate, this algorithm correctly tracks the probability distribution over the robot's location over time.

One thing to keep in mind is that a perfect localization algorithm doesn't give us perfect information about the location of the robot. Our knowledge of the robot's location is fundamentally limited by the uncertainty in our sensor model, the uncertainty in our motion model, and our uncertainty about the robot's initial location when localization began. What we can guarantee is that the recursive state estimation algorithm above gives the best possible estimate given these various sources of uncertainty.

Prediction Example

Figure 4.7 illustrates the process of applying the prediction formula. In the motion model for this example, there are three possible outcomes when the robot attempts to move to the right: There is a 60% chance that the action works as expected and the robot moves one cell to the right. There is a 20% chance that the action fails and the robot stays in the same location, and there is a 20% chance that the robot overshoots and moves two positions to the right.

Figure 4.7(a) illustrates the calculation for just state 5. Given this motion model, there are three possible ways the robot could end up in state 5: it could start in state 3 and overshoot, it could start in state 4 with the expected outcome, or it could start in state 5 and fail to move. The prediction formula sums across these three possibilities:

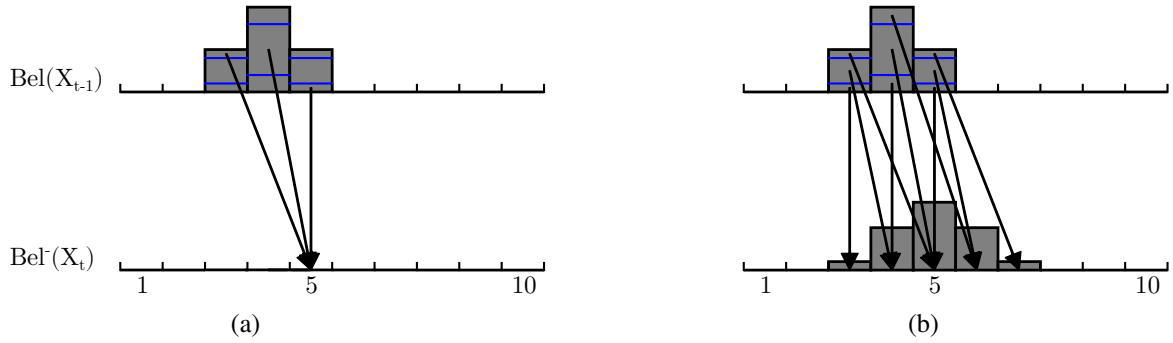


Figure 4.7: Using the prediction formula to update the belief state based on the motion model. (a) Each arrow represents a term in the sum for $Bel^{-}(X_t = 5)$. (b) The final belief distribution after all sums have been calculated.

$$
\begin{aligned}
Bel^{-}(X_t = 5) = & \sum_{x_i \in X_{t-1}} P(X_t = 5 \mid X_{t-1} = x_i,\ U_t = Right) Bel(X_{t-1} = x_i) \\
= & P(X_t = 5 \mid X_{t-1} = 3,\ U_t = Right) Bel(X_{t-1} = 3) + \\
& P(X_t = 5 \mid X_{t-1} = 4,\ U_t = Right) Bel(X_{t-1} = 4) + \\
& P(X_t = 5 \mid X_{t-1} = 5,\ U_t = Right) Bel(X_{t-1} = 5) \\
= & (.2 \times .25) + (.6 \times .5) + (.2 \times .25) \\
= & .4
\end{aligned}
$$

The full belief prediction update involves repeating this calculation for each state. The result is illustrated in figure 4.7(b)

4.3.1 Efficiency Considerations

The discrete, grid-based state representation assumed so far raises some of the same efficiency issues that were discussed in Chapter 3. In particular, the space and computational requirements grow exponentially with the dimensionality of the robot's state. Two-dimensional localization may be tractable, but once we add orientation and three-dimensional location, the problem quickly becomes un-manageable. There is also a trade-off between the granularity of the discretization and the precision of localization. Finer granularity may result in better localization, but only at the expense of additional computational cost.

In practice, it is common to use one of two alternative formulations of the recursive state estimation algorithm described above: the **Kalman filter** or the **particle filter**.

4.4 Kalman Filter

The overall form of the Kalman filter algorithm is the same as the recursive state estimation algorithm described above: There is a prediction phase in which the state estimate is projected forward through time, followed by a correction phase in which the state estimate is updated according to the latest sensor reading. The difference is that the Kalman filter uses an alternative representation of the belief state, and makes some strong assumptions about the form of the motion and sensor models:

- The belief state is represented as a multi-variate normal distribution.
- The state update and sensor models must be linear.
- The noise in both the motion and sensor models must be described by multi-variate normal distributions.

If these assumptions are satisfied, the Kalman filter provides an efficient and optimal localization algorithm.

4.4.1 Linear State Dynamics

As an example, consider the following system of difference equations that describe the motion of an object moving at a fixed velocity in two dimensions:

$$
\begin{aligned}
x_{t+1} &= x_t + \dot{x}_t dt \\
y_{t+1} &= y_t + \dot{y}_t dt \\
\dot{x}_{t+1} &= \dot{x}_t \\
\dot{y}_{t+1} &= \dot{y}_t
\end{aligned}
$$

In these equations x_t and y_t represent the x and y coordinates of the object at time t, $\dot{x}_t$ and $\dot{x}_t$ represent the instantaneous velocity, and dt represents the time interval between updates. The first two equations describe the motion of the object, while the second two equations simply describe the fact that the velocity remains constant over time.

This system can be described more concisely by representing the state information as a vector, and the series of difference equations as the matrix product:

$$\mathbf{x}_{t+1} = \mathbf{F}\mathbf{x}_t$$

Where $\mathbf{x}_t$ represents the state vector and $\mathbf{F}$ is a matrix that describes the update equations:

$$\mathbf{x}_t = \begin{bmatrix} x_t \\ y_t \\ \dot{x}_t \\ \dot{y}_t \end{bmatrix}, \qquad \mathbf{F} = \begin{bmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We may also want to represent the fact that there it is possible to apply a control signal to the object. For example, we could update our update equations with terms representing an application of force that impacts the objects velocity:

$$
\begin{aligned}
x_{t+1} &= x_t + \dot{x}_t dt \\
y_{t+1} &= y_t + \dot{y}_t dt \\
\dot{x}_{t+1} &= \dot{x}_t + \ddot{x}_t dt \\
\dot{y}_{t+1} &= \dot{y}_t + \ddot{y}_t dt
\end{aligned}
$$

We can incorporate this control signal by adding an additional term to our linear state update equation:

$$\mathbf{x}_{t+1} = \mathbf{F}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t$$

where

$$\mathbf{u}_t = \begin{bmatrix} \ddot{x}_t \\ \ddot{y}_t \end{bmatrix}, \qquad \mathbf{F} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ dt & 0 \\ 0 & dt \end{bmatrix}.$$

Finally, we need to take into account that the motion model should not be completely deterministic. We don't expect any model to make perfect predictions in a real-world system. The Kalman filter works under the assumption that system noise is normally distributed. We can incorporate noise by adding one final term to our state update equation:

$$\mathbf{x}_t = \mathbf{F}\mathbf{x}_{t-1} + \mathbf{B}\mathbf{u}_{t-1} + \mathbf{w}_{t-1} \tag{4.12}$$

Where $\mathbf{w}$ is a random vector drawn from a normal distribution with mean zero and covariance Q:

$$\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q})$$

4.4.2 Linear Sensor Model

The sensor model for a Kalman filter is also represented by a linear model corrupted with normally-distributed noise. The sensor model has the following form:

$$\mathbf{z}_t = \mathbf{H}\mathbf{x}_t + \mathbf{v}_t \tag{4.13}$$

Where $\mathbf{z}_t$ represents the observed sensor reading and $\mathbf{v}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{R})$ represents normally distributed sensor noise with covariance $\mathbf{R}$.

Continuing the example from above, we will assume that we have access to a sensor that provides estimates about the coordinates of the object, but doesn't provide any direct information about the velocity. In this case, we have

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

4.4.3 Kalman Filter Algorithm

The Kalman filter stores the belief state as a normal distribution. The mean of the distribution $\hat{\mathbf{x}}$ represents the current best estimate of the system state, while the covariance $\mathbf{P}$ represents the amount (and shape) of the current uncertainty. As with the recursive state estimation algorithm described in Section 4.3, the algorithm proceeds in two stages: a prediction stage, followed by a correction stage. Again, the $^-$ superscript indicates that the result is the estimated belief state before sensor readings have been incorporated. The full algorithm is outlined in Figure 4.8.

Inputs: Initial state estimate $\hat{\mathbf{x}}_0$ and covariance $\mathbf{P}_0$.

- **Repeat forever:**
  - **Prediction**
    - Project the state forward according to the motion model:

$$\hat{\mathbf{x}}_t^- = F\hat{\mathbf{x}}_{t-1} + B\mathbf{u}_{t-1} \tag{4.14}$$

    - Project the covariance of the state estimate forward:

$$\mathbf{P}_t^- = F\mathbf{P}_{t-1}F^T + \mathbf{Q} \tag{4.15}$$

  - **Correction**
    - Compute the Kalman gain:

$$\mathbf{K}_t = \mathbf{P}_t^- H^T (H\mathbf{P}_t^- H^T + \mathbf{R})^{-1} \tag{4.16}$$

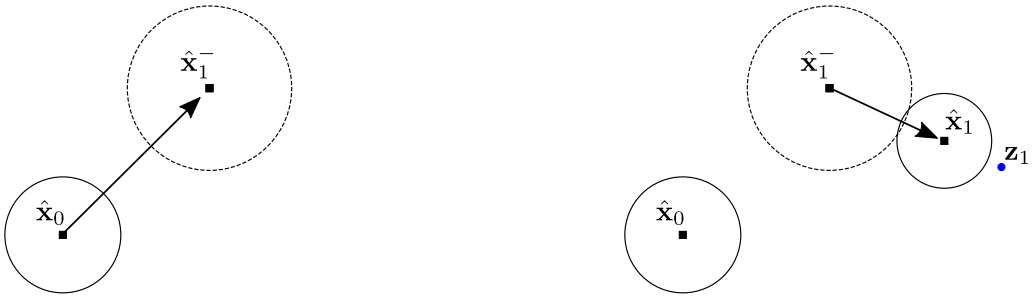
    - Use the sensor reading to update the state estimate:

$$\hat{\mathbf{x}}_t = \hat{\mathbf{x}}_t^- + \mathbf{K}_t(\mathbf{z}_t - H\hat{\mathbf{x}}_t^-) \tag{4.17}$$

    - Update the covariance of the state estimate:

$$\mathbf{P}_t = \mathbf{P}_t^- - \mathbf{K}_t H\mathbf{P}_t^- \tag{4.18}$$

Figure 4.8: Kalman filter algorithm.



(a) (b)

Figure 4.9: Kalman filter example. (a) **Prediction phase:** The state estimate is updated according to the linear motion model. The solid circle represents the initial uncertainty, while the dotted circle represents the increased uncertainty based on the known noise in the motion model. (b) **Correction phase:**. The state prediction is adjusted in the direction of the sensor reading. The uncertainty in the estimate decreases as a result of incorporating the sensor information.

The prediction phase of the algorithm updates the state estimate according to the motion model (Equation 4.14) and increases the estimated uncertainty according the noise in the motion model (Equation 4.15). This is illustrated in 4.9.

The correction phase involves updating the state estimate to account for the most recent sensor reading. Equation 4.17 can be understood as taking a weighted average between the prediction estimate and the estimate that is suggested by the latest sensor value. The **Kalman gain**, calculated in Equation 4.16, represents the optimal trade-off between those two sources of information. Intuitively, the Kalman filter places a higher weight on the sensor when the sensor noise is low relative to the uncertainty in the state estimate. Conversely, the Kalman filter places less weight on the sensor reading if the sensor noise is high relative to the current uncertainty. Put another way: the Kalman filter puts more trust in the more reliable source of information.

4.5 References and Further Reading

The mathematical foundations of probability theory are too broad and varied to survey here. Many of the key formalisms were developed by Pierre-Simon Laplace in the early 19th century.

General artificial intelligence textbooks such as (Russell & Norvig, 2010) and (Poole & Mackworth, 2017) provide an introduction to probabilistic modeling and reasoning as computational tools. The standard reference for the application of probabilistic algorithms in robotics is (Thrun et al., 2005), which provides much more detail on all of the algorithms discussed in this chapter.

The Kalman filter was introduced by R.E. Kalman in 1960 (Kalman, 1960). Greg Welch and Gary Bishop published a popular tutorial introduction that provides a concise overview of the algorithm along with a derivation (Welch & Bishop, 1995).

A history and overview of the particle filtering algorithm is provided by (Godsill, 2019), which attributes key steps in its development to Adrian Smith's research group at Imperial College London in the 1990's (Smith & Gelfand, 1992).

Bibliography

Anderson, B. D. O., & Moore, J. B. (1990). *Optimal control: Linear quadratic methods*. Dover Publications, Inc. (Cited on page 21).

Bennett, S. (2001). The past of pid controllers. *Annual Reviews in Control*, *25*, 43–53 (cited on page 21).

Correll, N., Hayes, B., Heckman, C., & Roncone, A. (2022). *Introduction to autonomous robots: Mechanisms, sensors, actuators, and algorithms* (1st). MIT Press, Cambridge, MA. (Cited on page 22).

Dudek, G., & Jenkin, M. (2010). *Computational principles of mobile robotics* (2nd). Cambridge University Press. (Cited on page 21).

Foote, T. (2013). Tf: The transform library, In *Technologies for practical robot applications (TePRA), 2013 IEEE international conference on*. (Cited on page 31).

Godsill, S. (2019). Particle filtering: The first 25 years and beyond, In *2019 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. (Cited on page 72).

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, *4*(2), 100–107 (cited on page 56).

Johnson, M., & Moradi, M. (2006). *PID control: New identification and design methods*. Springer London. (Cited on page 21).

Kalman, R. E. (1960). A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, *82*(1), 35–45 (cited on page 72).

Karaman, S., & Frazzoli, E. (2010). Incremental sampling-based algorithms for optimal motion planning. *Robotics Science and Systems VI*, *104*(2) (cited on page 56).

Kavraki, L., Svestka, P., Latombe, J., & Overmars, M. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, *12*(4), 566–580 (cited on page 56).

Kay, J. (2020). *Introduction to homogeneous transformations & robot kinematics* (technical report). Rowan University Computer Science Department. (Cited on page 31).

LaValle, S. M. (1998). *Rapidly-exploring random trees: A new tool for path planning* (technical report TR 98-11). Computer Science Dept., Iowa State University. (Cited on page 56).

LaValle, S. M. (2006). *Planning algorithms*. Cambridge university press. (Cited on page 56).

Lozano-Perez. (1983). Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, *C-32*(2), 108–120 (cited on page 56).

Nilsson, N. J. (2009). *The quest for artificial intelligence*. Cambridge University Press. (Cited on page 56).

Poole, D., & Mackworth, A. (2017). *Artificial intelligence: Foundations of computational agents* (2nd edition). Cambridge, UK, Cambridge University Press. http://artint.info/2e/html/ArtInt2e.html. (Cited on pages 21, 56, 72)

Russell, S. J., & Norvig, P. (2010). *Artificial intelligence : A modern approach* (3nd). Prentice Hall. (Cited on pages 21, 56, 72).

Siciliano, B., & Khatib, O. (Editors). (2016). *Springer handbook of robotics*. Springer Berlin Heidelberg. (Cited on page 22).

Smith, A. F. M., & Gelfand, A. E. (1992). Bayesian statistics without tears: A sampling-resampling perspective. *The American Statistician*, *46*(2), 84–88. Retrieved January 11, 2023, from http://www.jstor.org/stable/2684170 (cited on page 72)

Thrun, S., Burgard, W., & Fox, D. (2005). *Probabilistic robotics (intelligent robotics and autonomous agents)*. The MIT Press. (Cited on pages 22, 72).

Welch, G., & Bishop, G. (1995, January 1). *An introduction to the kalman filter* (technical report TR95-041). University of North Carolina at Chapel Hill, Department of Computer Science. https://sreal.ucf.edu/wp-content/uploads/2017/02/kalman_intro.pdf. (Cited on page 72)