ECE 411: Sensor Fusion for Robotics

Bryan Van Scoy

November 12, 2024

Contents

Part I

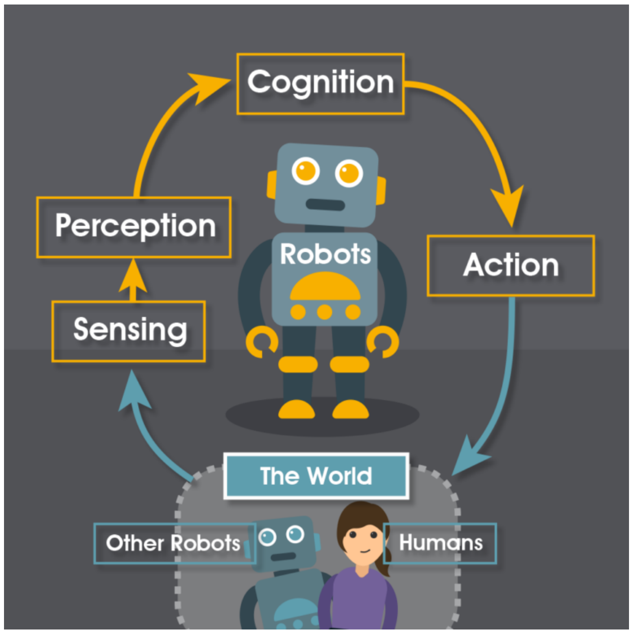
Introduction to Robotic Algorithms

1

Introduction

1.1 Robotics

Robotics is the science of perceiving and manipulating the physical world through computer-controlled devices. The robot uses sensors to perceive its environment, it uses a computer to process and interpret the information, and then uses actuators to manipulate the environment (which then affects what it senses!). This creates a feedback loop between the robot and its environment as shown below.



Types of robots

There are two main categories of robots: manipulators and mobile robots.

- **Manipulators** such as robotic arms work in constrained environments such as factories. These environments are typically designed specifically for the robot (for instance, they may be clear of clutter and the robot may know the exact location of objects in its surroundings). As manipulators are fixed to their environment, they are typically able to measure absolute positions of their joints using encoders.

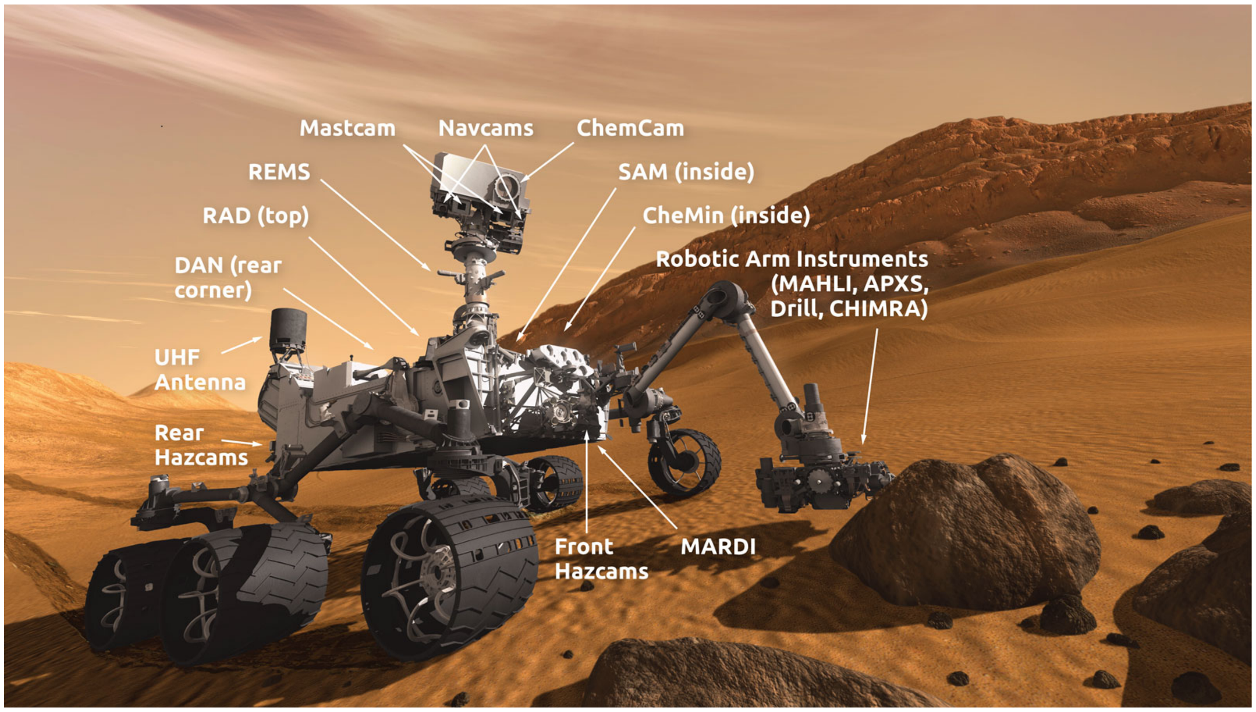
- **Mobile robots** on the other hand, often operate in unconstrained environments such as on roads, in the ocean, in the air, etc. Mobile robots may have access to their global position within the environment (via GPS, for example), although many robots must operate using only relative measurements of their position (for example, using encoders). For this reason, sensing in mobile robotics is critical for the robot to be able to reason about the environment and itself within the environment.

Manipulator

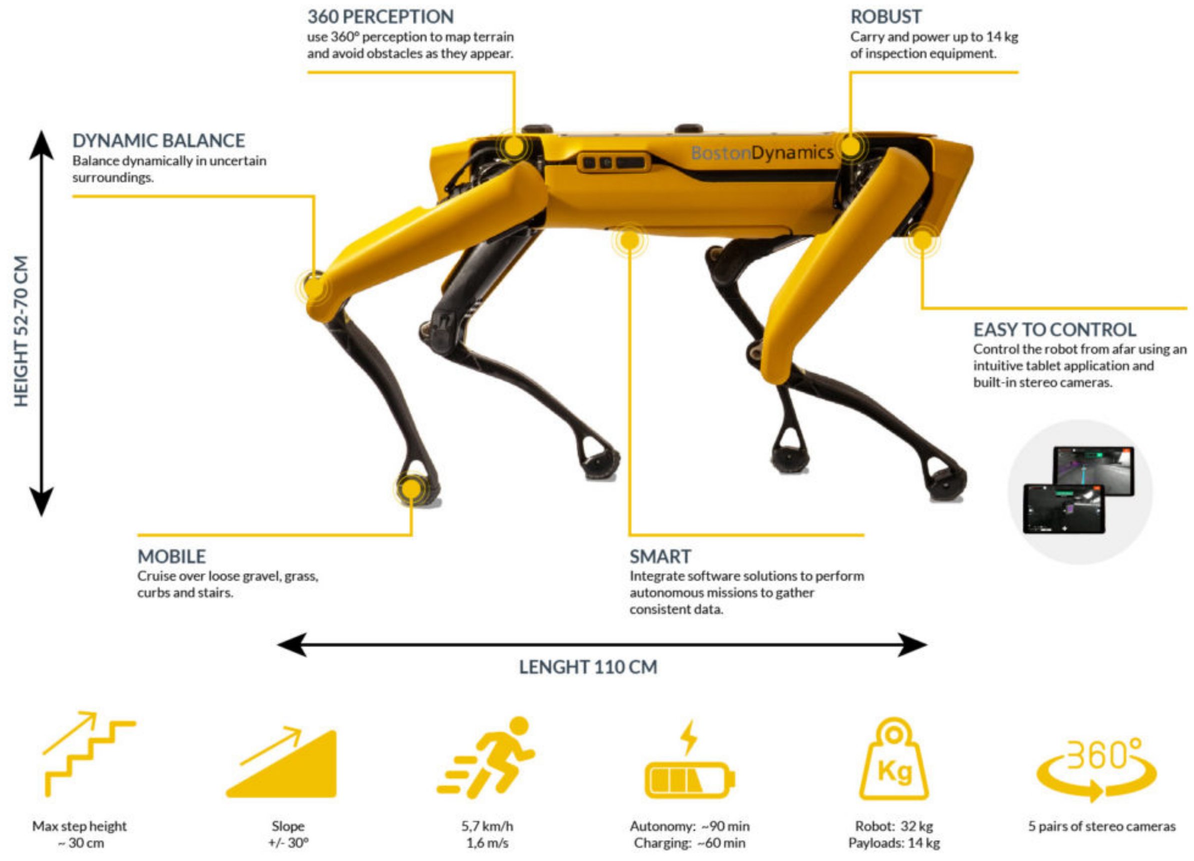
Mobile robot

Robotics applications

Mars Curiosity Rover.



Spot from Boston Dynamics.


360 PERCEPTION
use 360° perception to map terrain and avoid obstacles as they appear.
ROBUST
Carry and power up to 14 kg of inspection equipment.
DYNAMIC BALANCE
Balance dynamically in uncertain surroundings.
HEIGHT 52-70 CM
EASY TO CONTROL
Control the robot from afar using an intuitive tablet application and built-in stereo cameras.
MOBILE
Cruise over loose gravel, grass, curbs and stairs.
SMART
Integrate software solutions to perform autonomous missions to gather consistent data.
LENGHT 110 CM
Max step height ~ 30 cm
Slope +/- 30°
5,7 km/h 1,6 m/s
Autonomy: ~90 min Charging: ~60 min
Robot: 32 kg Payloads: 14 kg
5 pairs of stereo cameras

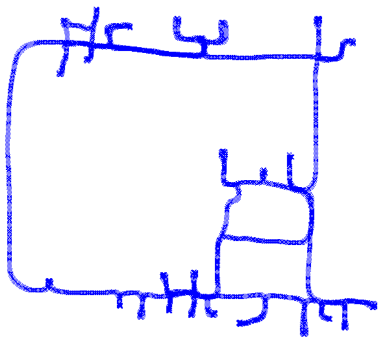
Amazon Warehouse.

Problems in robotics

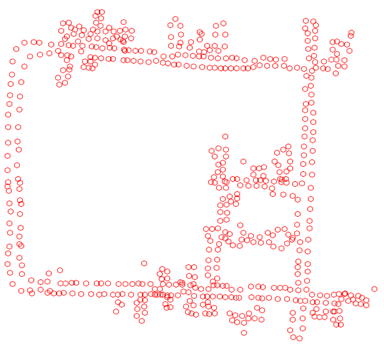
There are several fundamental problems in robotics that we will study in this course.

- **Estimation**
  - **Mapping:** Use measurements to construct a map of the environment.
  - **Localization:** Use measurements to estimate the robot's position relative to an external reference frame.
  - **Simultaneous localization and mapping (SLAM):** The localization and mapping problems are inherently interdependent. To construct a map using sensor data, the robot must know where it is in the environment. But to know where it is, the robot must have a map. In some applications, the robot may be given a map (such as a robotic manipulator knowing the positions of nearby objects) or its location (such as using GPS). When these are both unknown, however, the robot must simultaneously estimate its position while constructing the map.
- **Control**
  - **High-level planning:** Any useful robot must take actions within its environment. In high-level planning, the robot must decide its broad goals and construct a plan to achieve them. For instance, a robot may plan to move to a desired location, pick up a certain object, clean a particular room, etc.
  - **Low-level motor controls:** Once the robot has identified its high-level plan, it must execute that plan using its various motors and actuators. Low-level controls typically involve things like PID motor control and obstacle avoidance.

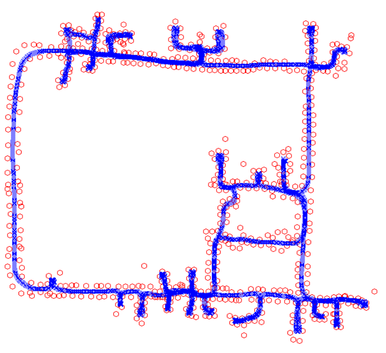
The following example illustrates the localization, mapping, and SLAM problems for a mobile robot in the Institute of Robotics and Mechatronics building at the German Aerospace Center.



Localization



Mappling



SLAM

Discussion questions.

- How do we model the robot's motion?
- How do we model the robot's sensors?
- How do we represent the map?
- Do we solve the problem iteratively while the robot is moving (online) or once all of the data has been collected (offline)?

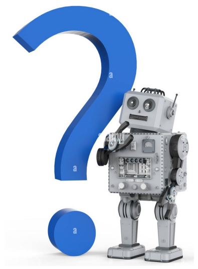
1.2 Sensor fusion

Sensor fusion is combining measurements from multiple sensors such that they jointly give more information than any sensor individually.

Uncertainty in robotics

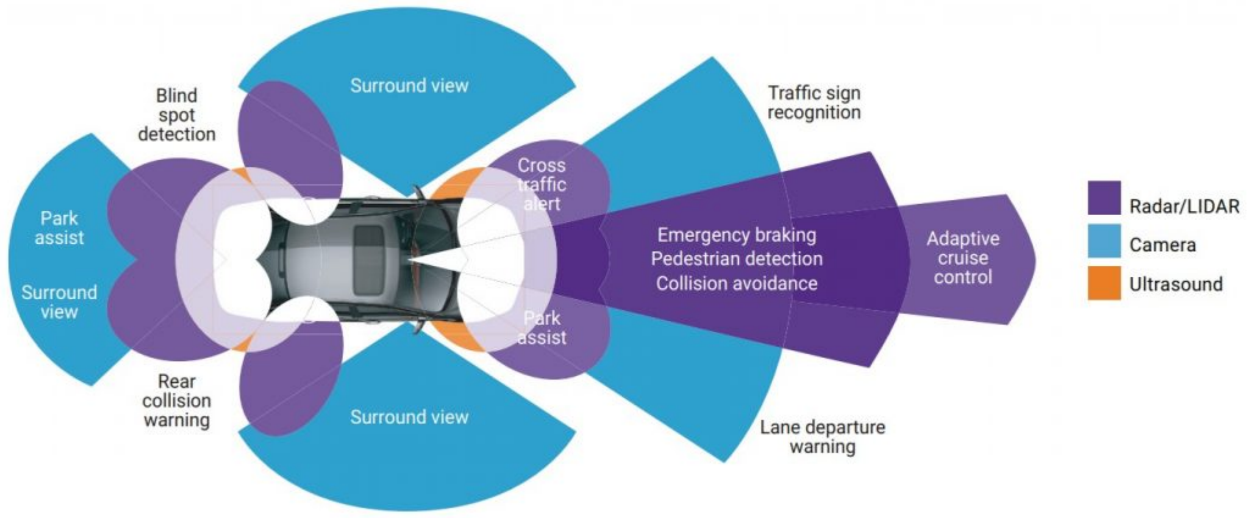
Robots must make decisions in the face of uncertainty. Some causes of uncertainty are:

- sensors are limited in what they can perceive
- actuators produce unpredictable affects on the robot and environment
- robots use imperfect models of reality
- algorithms often use approximations to ease computations
- robots operate in unpredictable environments



For these reasons, it is imperative that robots take uncertainty into account when reasoning about themselves and their environment. We will use probabilitistic models to describe the interactions between a robot and its environment, and the robot will use these probabilistic models to decide which actions to take.

Smart cars, for instance, have numerous sensors to detect nearby vehicles and pedestrians, traffic signs, hazardous conditions, etc. Information from these various sensors must be fused together to construct a cohesive estimate of the surroundings so that the control algorithm can take appropriate actions, such as breaking to avoid a collision.



The following simple example illustrates the main idea of sensor fusion: multiple measurements can be combined to obtain a better estimate than either measurement by itself.

Example (A first example)**.** Suppose we use a sensor to measure a quantity.

- each measurement from the sensor is a random variable
- the measurements are unbiased, so their mean is the true value
- the measurements have variance σ^2

Given two measurements z_1 and z_2, can we construct a better estimate?

$$\hat{z} = \frac{z_1 + z_2}{2} \qquad \text{with variance} \qquad \hat{\sigma}^2 = \frac{\sigma^2}{2}$$

The fused estimate has lower variance than any individual measurement!

Example (A second example)**.** Now suppose we use two different sensors to measure the quantity.

- the measurement from the first sensor is z_1 with variance σ_1^2
- the measurement from the second sensor is z_2 with variance σ_2^2

Now what is a better estimate?

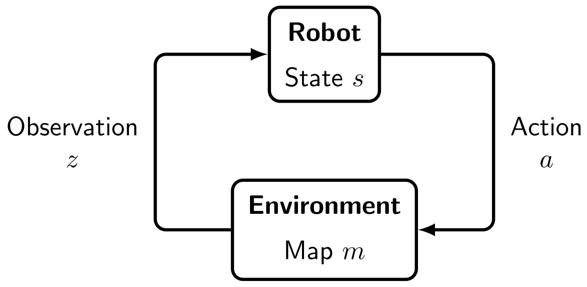
$$\hat{z} = \hat{\sigma}^2 \left(\frac{z_1}{\sigma_1^2} + \frac{z_2}{\sigma_2^2} \right) \qquad \text{with variance} \qquad \hat{\sigma}^2 = \frac{1}{1/\sigma_1^2 + 1/\sigma_2^2}$$

For example, if $\sigma_1 = 0.5$ and $\sigma_2 = 0.2$, then $\hat{\sigma} = 0.19$. Even measurements from a noisy sensor can improve the estimate.

This method of fusing estimates is called *inverse-variance weighting*, and the derivation is provided in the appendix.

1.3 Robot environment interaction

Both the robot and its environment (or world) are dynamical systems that evolve over time. The robot can acquire information about its environment using sensors, and the robot can also influence the environment through its actuators, both of which are uncertain.



Dynamical systems are characterized by their *state*, which is everything that is needed to describe how the system will evolve over time given any input signals. As both the robot and its environment are dynamical systems, they each have a state. We denote time by t, which typically takes positive integer values.

- **Robot:** The state of the robot at time t is denoted s_t. The robot state typically consists of the pose of the robot (position and orientation) and its velocities (translational and rotational). For a robot in three-dimensional space, the pose consists of six variables, three for its position and three for its orientation (pitch, roll, yaw). For a robot in two-dimensional space, the pose consists of three variables, two for its position in the plane and one angle for its orientation, which we denote by

$$s = (x, y, \theta)$$

- **Environment:** The state of the environment at time t is denoted m_t, which we refer to as a *map*. The environment state can be very high-dimensional, making it difficult to represent exactly. Instead, the environment is often approximated by *landmarks*, which we denote by ℓ. A landmark may be a tree, a wall, or a pixel within a larger surface. We denote the position of a landmark in the two-dimensional plane as $\ell = (\ell_x, \ell_y)$. We often assume that the environment is *static* so that its state is constant in time. For a given measurement, c represents the *correspondence* of the landmark. If the robot is able to recognize the landmark and distinguish it from other landmarks (such as seeing a distinct QR code), then the correspondences are known. Otherwise, the robot must reason about which landmark it measured. There are two main types of maps.
  - **Feature-based maps** are indexed by a feature index, where each element in the map is a *feature* that contains the location and properties of the feature.
  - **Location-based maps** are indexed by location, where each element in the map specifies the properties of the environment at that location.
- **Sensor measurements:** Perception is the process by which a robot uses its sensors to obtain information about the environment. Such an interaction produces a *measurement* (or *observation*), which we denote by z. For simplicity, we assume that a robot makes one measurement z_t at each discrete time step t.
- **Control actions:** Robots interact with their environment by applying actions, which may change the state of the robot and/or the environment. Examples of actions include robot motion and manipulation of objects. We denote a control action by a. For simplicity, we assume that a robot takes one action a_t at each discrete time step t.

Time. The robot state s, map of the environment m, sensor measurements z, and control actions s all depend on time t. We denote a collection of these quantities between two times t_1 and t_2 by a subscript $t_1 : t_2$. For instance, we denote the collection of all robot states between two times t_1 and t_2 as

$$s_{t_1:t_2} = (s_{t_1}, s_{t_1+1}, \dots, s_{t_2})$$

To simplify notation, we often omit the explicit dependence on time and instead use a prime to denote a quantity at the next time step. For example, if the current robot state (at some generic time) is s and it takes control action a, the state of the robot after applying this action is denoted by s'.

1.4 Approaches

In probabilistic robots, both the state of the robot and the map of the environment are not known with certainty. Robot algorithms must take this certainty into account when fusing sensor measurements and selecting control actions. There are two main approaches based on the information that is available.

- **Model-based control.** In model-based control, it is assumed that we have access to mathematical models that describe the characteristics of the robot and its environment (or we construct a model from the observations). This includes models for each sensor on the robot along with how its actions affect its own state and that of its environment.
- **Data-driven control.** For large and complex systems, constructing a model may be prohibitively difficult. In data-driven control, the robot reasons about the world directly from the information that it collects through its sensors (without first constructing a model). This approach is also called *reinforcement learning* to emphasize that the robot learns actions by receiving rewards that reinforce its decisions.

In this class, we will focus on model-based control in which the models are stochastic to take into account uncertainty in the world. When models are available, the robot can reason about what actions to take by combining two separate algorithms: a filter and a controller.

$$\begin{tikzcd}\text{Observations} \arrow[r] & \boxed{\text{Filter}} \arrow[r, "\text{Belief}"] & \boxed{\text{Controller}} \arrow[r] & \text{Actions}\end{tikzcd}$$

The filter uses the observations from sensors to construct the robot's *belief* about the state of the world. The belief (also known as its *state of knowledge* or *information state*) captures the robot's knowledge of its own state and/or that of its environment. The controller then chooses the action to take based on this belief. This separates the robot's reasoning into *estimation* and *control*.

The belief is the probability that the robot is in a particular state given all available information (such as past sensor measurements, past control actions, and any prior information). The belief is a conditional probability distribution over states conditioned on the available information. For a quantity x, we denote its belief at time t by $b_t(x)$, which is an abbreviation for the conditional probability distribution

$$b_t(x) = p(x \mid z_{1:t}, a_{1:t})$$

1.5 Notation

For reference, the following table lists some of the notation that will be used throughout the course.

t	time
s	robot state
x	coordinate of the robot state in the x-axis of the two-dimensional plane
y	coordinate of the robot state in the y-axis of the two-dimensional plane
θ	orientation of the robot measured counterclockwise from the x-axis
m	map of the environment
ℓ	landmark
ℓ_x	coordinate of the landmark position in the x-axis of the two-dimensional plane
ℓ_y	coordinate of the landmark position in the y-axis of the two-dimensional plane
c	correspondence of a landmark
z	sensor measurement
a	control action
$p(s' \mid s, a)$	state transition probability
$p(z \mid s)$	measurement probability
$b(s)$	belief
$g(s, a, \varepsilon)$	actuation model
$h(s, m, \delta)$	perception model
ε	actuation noise
δ	perception noise

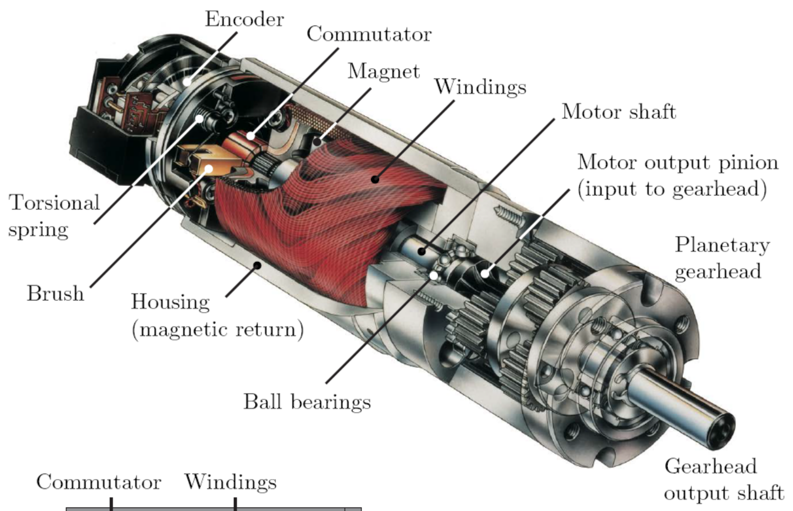
2

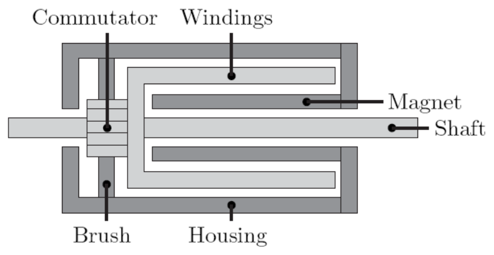
Robot motion

Robot dynamics is the study of robot motions and the forces and torques that cause them. These forces and torques are produced by actuators, and the resulting motion depends on the type of robot (and for mobile robots in particular, what type of wheels they use). In this chapter, we study probabilistic models that describe how a control action influences the motion of a robot.

2.1 Actuators

Actuators are electromechanical components that enable a robot to influence its own state and/or that of its environment. While there are many types of actuators, perhaps the most common is a DC motor, which uses a supplied voltage to apply a torque to a shaft.





2.2 Robot locomotion

While actuators apply forces and torques to the robot, the resulting motion depends on the locomotion mechanism of the robot. Several mechanisms for locomotion are shown below, which are (from left to right): legged locomotion, unicycle, omniwheel, Mecanum (or Swedish) wheel, Ackermann drive.

The type of locomotion has important effects on the way in which the robot can move. For instance, a unicycle may only move in the direction that the wheel is facing (assuming that it does not slip). This results in a constraint on the possible velocities. Omniwheels and mecanum wheels, on the other hand, can be used to obtain an instantaneous velocity in any direction.

2.3 Models

An actuation model is a mathematical relationship between the state of the robot at a future time, the current state of the robot, and the control action. One form of actuation model is the probability that the robot is in some state s' given that it was in state s and took control action a,

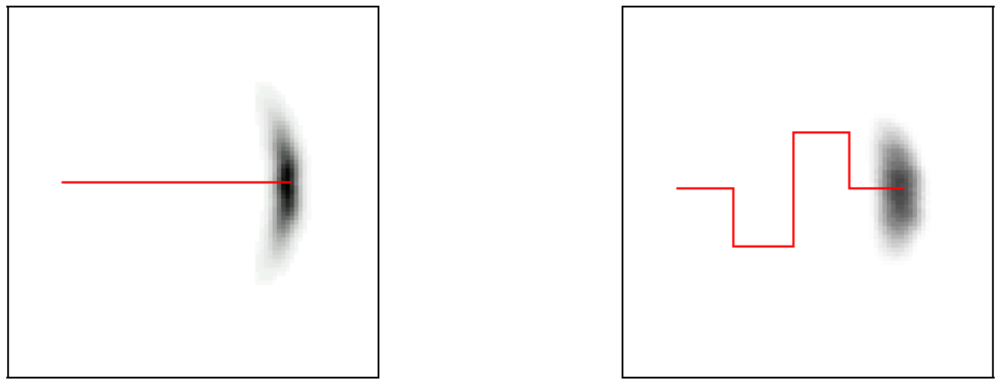
$$p(s' \mid s, a)$$

Another way to represent an actuation model is to specify the next state s' as a function of the current state s and control action a,

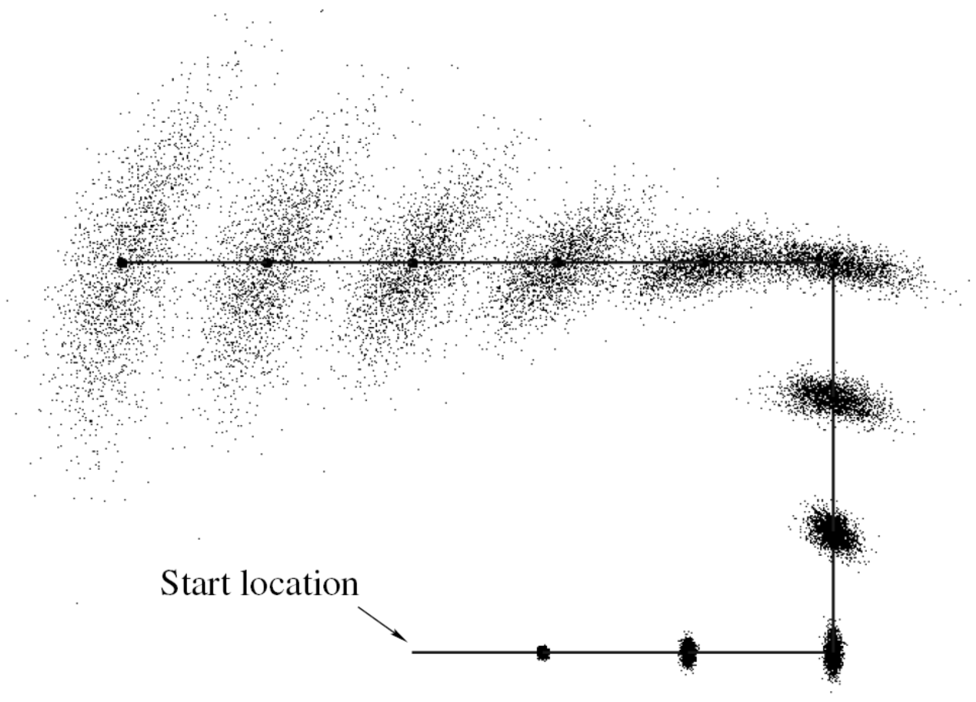
$$s' = g(s, a, \varepsilon)$$

where ε is a random variable that represents the actuation noise. The first form represents the probability of arriving in a given state s', while the second form enables us to sample possible future states by sampling the actuation noise ε and then evaluating the model g. Both forms will be useful representations of the robot motion depending on the context.

The following figures show the probability distribution of the state of a planar mobile robot with state $s = (x, y, \theta)$ after executing the control actions (shown in red). Darker locations indicate higher probabilities. (Instead of the full robot state, the probability distribution is projected onto the two-dimensional space of the robot position (x, y), since it is difficult to also visualize the robot orientation θ.)



A consequence of imperfect actuation is that — without sensing — the state of the robot becomes more uncertain over time. The figure below illustrates this. The robot starts in a known state, and the solid lines indicate the deterministic trajectory for the given control actions. The small dots are samples of the robot state at various points in time. As time progresses, the robot becomes more and more uncertain about its state due to the accumulation of errors in the actuation model. The remedy to this problem is for the robot to sense its environment, which reduces its uncertainty.



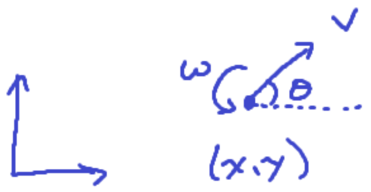
The two previous figures illustrate the two ways in which we will use actuation models. The first figure shows the probability distribution over the future states of the robot given its known starting state and the control actions. Alternatively, the second figure shows samples from this distribution over time. Depending on the algorithm, we may need either the probability of a future state or a sample of the future state from this probability distribution.

2.4 Nonholonomic planar mobile robots

We now consider nonholonomic mobile robots in the plane. The motion of such robots is constrained due to the configuration of the wheels. Examples of such robots are the unicycle, differential drive, and car-like robots.

Deterministic model

To model the motion of a nonholonomic mobile robot, we start with a general model in which the robot controls its instantaneous linear and rotational velocities, v and ω. The pose of the robot consists of its position (x, y) and orientation θ in the plane (represented in some frame).



In terms of the instantaneous velocities, the state of the robot evolves in continuous time according to the following dynamics

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}$$

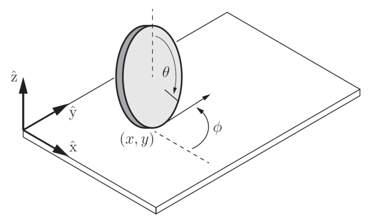
where a dot denotes the derivative with respect to time, for example, $\dot{x}(t) = \frac{\mathrm{d}}{\mathrm{d}t}x(t)$. This is a nonlinear differential equation that describes how the pose of the robot changes over time depending on the input velocities.

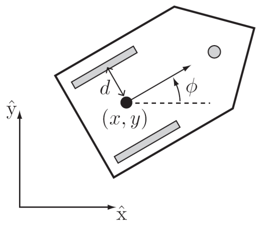
The nonholonomic robots described in this section have constraints on their attainable instantaneous velocities. We can see this from the above equation, as there are three velocities $(\dot{x}, \dot{y}, \dot{\theta})$ and only two control inputs (v, ω). To find the constraint on the velocities, we can solve the first equation for v and substitute this into the second equation to obtain

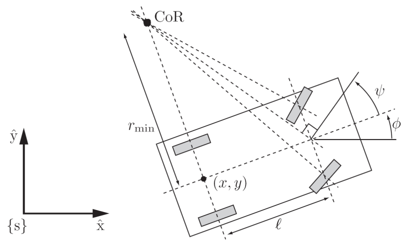
$$\dot{x}\sin\theta - \dot{y}\cos\theta = 0$$

This constraint specifies that the robot must move in the direction of its heading, so its instantaneous linear velocity can have no component orthogonal to its heading.

We now consider three specific types of nonholonomic mobile robots and describe how the instantaneous velocities are related to the control actions (such as the angular speed of the wheels). The diagrams for the three types of robots are shown below (from left to right: unicycle, differential drive, and car-like robots). In each case, all wheels have radius r.







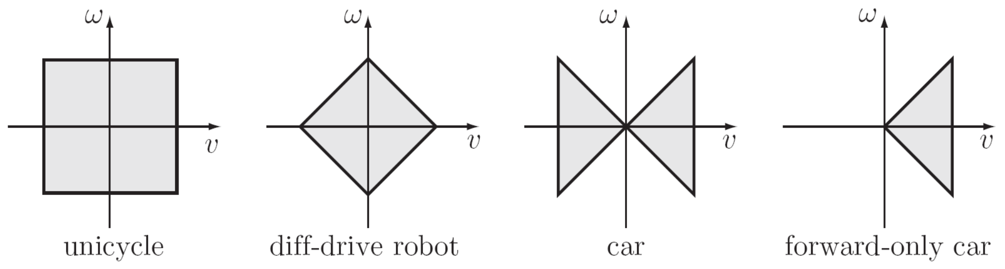
- **Unicycle dynamics**
  For a unicycle with control inputs $u = (u_1, u_2)$ where u_1 is the driving speed of the wheel and u_2 is the angular speed of the change in heading, the instantaneous velocities are related to the control inputs as
$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} v/r \\ \omega \end{bmatrix} \qquad \text{and} \qquad \begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} ru_1 \\ u_2 \end{bmatrix}$$

- **Differential drive robot**
  For a differential drive robot with control inputs $u = (u_l, u_r)$ the angular velocities of the left and right wheels, the instantaneous velocities are related to the control inputs as
$$\begin{bmatrix} u_l \\ u_r \end{bmatrix} = \frac{1}{r}\begin{bmatrix} v - \omega d \\ v + \omega d \end{bmatrix} \qquad \text{and} \qquad \begin{bmatrix} v \\ \omega \end{bmatrix} = \frac{r}{2}\begin{bmatrix} u_r + u_l \\ (u_r - u_l)/d \end{bmatrix}$$
  where $2d$ the distance between the wheels.

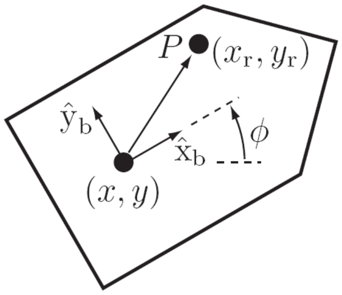
- **Car-like dynamics**
  For a car-like robot, the control inputs are $u = (v, w)$ where v is the forward speed of the car and w is the angular speed of the steering angle. Let ψ denote the steering angle of the car, and let ℓ denote the length of the car between the front and back wheels. Then the motion of the car is described by
$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ (\tan\psi)/\ell & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix}$$
  This is not of the standard form described above. However, suppose we can control the steering angle ψ directly, instead of its rate $w = \dot{\psi}$. This is a valid assumption if we can change the steering angle quickly by a low-level controller. Then ψ is no longer a state variable, and the instantaneous angular velocity ω is related to the heading by
$$\omega = \frac{v}{\ell}\tan\psi \qquad \text{and} \qquad \psi = \tan^{-1}\left(\frac{\ell\omega}{v}\right)$$
  Under the assumption that we can control the steering direction directly, this has the same form as our standard model.

As we have seen, we can represent various types of nonholonomic mobile robots in the standard form, where we take the instantaneous linear and rotational velocities as control inputs. The difference between the various robots, however, is that they have different constraints on the attainable instantaneous velocities. The instantaneous velocity of a car, for instance, may not be backwards (unless it can reverse).

Suppose each of the control inputs u_i is constrained to be in an interval $[u_{i,\min}, u_{i,\max}]$. Then the instantaneous velocities of the various robots are constrained by a polyhedron as shown below.



Suppose the robot has a sensor attached to a point P with coordinates (x_r, y_r) relative to the reference frame of the robot as shown below. We will now derive the instantaneous velocities that move the sensor with some given velocity, which is useful if we want to move the sensor to a desired location.



The coordinates of the point relative to the frame of reference on the center of the robot is

$$\begin{bmatrix} x_P \\ y_P \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + R(\phi) \begin{bmatrix} x_r \\ y_r \end{bmatrix}$$

Differentiating with respect to time yields

$$\begin{bmatrix} \dot{x}_P \\ \dot{y}_P \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} + \dot{\phi} \begin{bmatrix} -\sin\phi & -\cos\phi \\ \cos\phi & -\sin\phi \end{bmatrix} \begin{bmatrix} x_r \\ y_r \end{bmatrix}$$

Substituting the robot dynamics for $(\dot{x}, \dot{y}, \dot{\phi})$ and solving for the instantaneous velocities yields

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \frac{1}{x_r} \begin{bmatrix} x_r \cos\phi - y_r \sin\phi & x_r \sin\phi + y_r \cos\phi \\ -\sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} \dot{x}_P \\ \dot{y}_P \end{bmatrix}$$

This yields the instantaneous velocities needed to control the velocity of a sensor located on a robot.

Discrete-time dynamics

Recall that the state of a nonholonomic mobile robot in the plane evolves in continuous time according to the dynamics

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ \omega \end{bmatrix}$$

where v and ω are the instantaneous linear and rotational velocities. While this is a nonlinear differential equation, in this case we can find a closed-form solution between any two times assuming that the control actions are constant on that interval. This produces a discretized motion model in discrete time.

Let (x, y, θ) denote the robot state at some time t, and suppose the control actions v and ω are constant on the time interval $[t, t']$ for some future time $t' > t$. Denote the length of the time interval as $\Delta t = t' - t$. The orientation of the robot is then a linear function of time,

$$\theta(\tau) = \theta + \int_t^\tau \omega \, \mathrm{d}s = \theta + \omega\,(\tau - t)$$

and the next orientation is

$$\theta' = \theta + \omega\,\Delta t$$

Substituting this into the expression for the horizontal position and integrating yields

$$\begin{aligned} x' &= x + \int_t^{t'} v\cos\theta(\tau)\,\mathrm{d}\tau \\ &= x + v\int_t^{t'} \cos\big(\theta + w\,(\tau - t)\big)\,\mathrm{d}\tau \\ &= x + \frac{v}{\omega}\sin(\theta + \omega\,(\tau - t))\Big|_{\tau=t}^{t'} \\ &= x + \frac{v}{\omega}\Big(\sin(\theta + \omega\,\Delta t) - \sin(\theta)\Big) \end{aligned}$$

Performing a similar computation for the vertical coordinate gives that

$$y' = y - \frac{v}{\omega}\Big(\cos(\theta + \omega\Delta t) - \cos(\theta)\Big)$$

To summarize, the state of the robot (x', y', θ') after applying the (constant) control actions (v, ω) for some time Δt from the previous state (x, y, θ) is described by the velocity-based actuation model as

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} \frac{v}{\omega}\sin(\theta + \omega\Delta t) - \frac{v}{\omega}\sin(\theta) \\ -\frac{v}{\omega}\cos(\theta + \omega\Delta t) + \frac{v}{\omega}\cos(\theta) \\ \omega\Delta t \end{bmatrix} \qquad (\omega \neq 0)$$

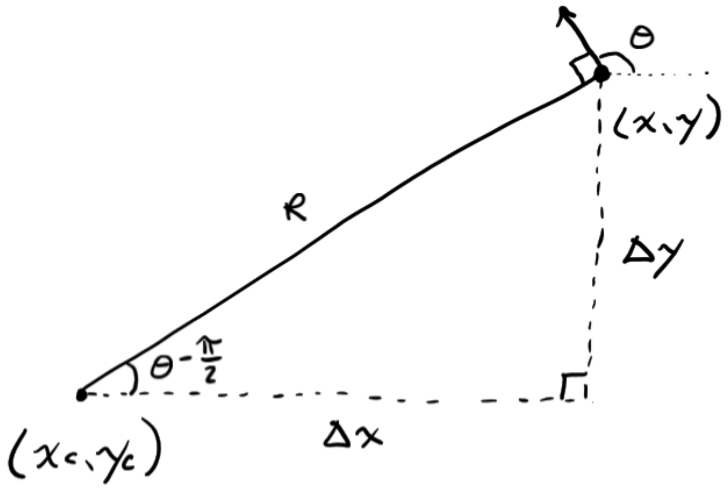
These equations are not valid when the angular velocity is zero since it appears in the denominator of a fraction. But in this case, the orientation of the robot is constant and the robot moves in the straight line motion

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} v\Delta t\cos\theta \\ v\Delta t\sin\theta \\ 0 \end{bmatrix} \qquad (\omega = 0)$$

This can be found by taking the limit as $\omega\Delta t \to 0$, or by simply considering the straight line motion.

Interpretation as circular motion

When moving with constant velocities, the robot moves in a circle. We now find the center of the circle (x_c, y_c) and its radius R in terms of the robot state and instantaneous velocities.



From the above diagram, we have that

$$\begin{aligned}\Delta x &= R\cos\big(\theta - \tfrac{\pi}{2}\big) = R\sin(\theta)\\ \Delta y &= R\sin\big(\theta - \tfrac{\pi}{2}\big) = -R\cos(\theta)\end{aligned}$$

where we used some basic trigonometric identities. From the definition of angular velocity, we have that $v = R\omega$. Therefore, the center of the circle and its radius are given in terms of the instantaneous velocities as

$$\begin{bmatrix} x_c \\ y_c \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \frac{v}{\omega}\begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix} \qquad \text{and} \qquad R = \frac{v}{\omega}$$

Alternative parameterizations

We can use other parameterizations of the robot motion besides the instantaneous velocities v and ω.

- **Circle radius and change in orientation.** The circle radius R and change in orientation $\Delta\theta = \theta' - \theta$ are related to the instantaneous velocities as

  $$R = \frac{v}{\omega} \qquad \text{and} \qquad \Delta\theta = \omega\Delta t$$

  Using this parameterization, the motion is described by

  $$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} R\sin(\theta + \Delta\theta) - R\sin(\theta) \\ -R\cos(\theta + \Delta\theta) + R\cos(\theta) \\ \Delta\theta \end{bmatrix}$$

  This parameterization is convenient in that it does not depend on the time interval Δt.

- **Wheel encoders for differential drive robot.** Consider a differential drive robot equipped with wheel encoders that measure the amount that each wheel rotates. We can use this along with the radius of the wheel to find the distances traveled by the left and right wheels, d_l and d_r. Since each wheel travels along the arc of a circle, its distance traveled is the product of the change in orientation with its distance from the center of the circle. Assuming that the wheels are located a distance $2d$ apart, the distance traveled by each wheel is

  $$d_l = \Delta\theta\,(R - d) \qquad \text{and} \qquad d_r = \Delta\theta\,(R + d)$$

  Solving these equations for the change in orientation and the radius of the circle gives

  $$\Delta\theta = \frac{d_r - d_l}{2d} \qquad \text{and} \qquad R = d \cdot \frac{d_r + d_l}{d_r - d_l}$$

  The circle radius and change in orientation can then be related to the instantaneous velocities as above.

Backward dynamics

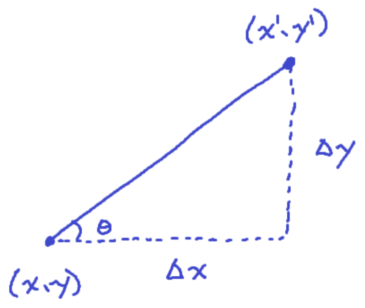
We next construct a backward model for the dynamics of a planar mobile robot. A backward model constructs the control inputs required to achieve a desired motion.

Suppose the robot is in state (x, y, θ). Our goal is to find the instantaneous velocities (v, ω) to drive the robot to a final position (x', y'). Due to the constraint on the robot motion, the final orientation is fixed based on this information. To achieve a desired final orientation as well, we could first move to the desired position and then rotate (for unicycle and differential drive robots, but car-like robots cannot rotate in place).

Suppose the wheel velocities remain constant during the transition, and denote the differences in each coordinate between the initial and final positions as

$$\Delta x = x' - x \qquad \text{and} \qquad \Delta y = y' - y$$

First consider the case in which the robot moves in a straight line from the initial point (x, y) in a direction θ to the final point (x', y') as shown below.



In this case, both wheels travel the same distance

$$d_l = d_r = \sqrt{(\Delta x)^2 + (\Delta y)^2}$$

and the final orientation is the same as the original orientation, $\theta' = \theta$. From the wheel distances, we can use the alternative parameterizations above to obtain the circle radius R and the instantaneous linear velocity v (the instantaneous angular velocity is $\omega = 0$).

Now suppose that the points do not lie on a straight line. We can then solve the forward model for the radius R and change in orientation $\Delta\theta$ to obtain

$$R = \frac{1}{2} \frac{\Delta x^2 + \Delta y^2}{\Delta y \cos\theta - \Delta x \sin\theta} \qquad \text{and} \qquad \Delta\theta = 2 \arctan\left(\frac{\Delta y - \Delta x \tan(\theta/2)}{\Delta x + \Delta y \tan(\theta/2)}\right) - \theta$$

We can then use the alternative parameterizations above to obtain the wheel distances (d_l, d_r) and the instantaneous velocities (v, ω).

Probabilistic model

The previous model is deterministic in that the future pose of the robot is completely determined by its current state and control inputs (or instantaneous velocities). We now show how to extend this to a probabilistic model that takes uncertainty in control actions into account.

A simple way to construct a probabilistic model for a planar mobile robot is perturb the control inputs by random noise:

$$\hat{u} = u + \varepsilon$$

where u is the desired control input, $\hat{u}$ is the perturbed input, and ε is the actuation noise. Since there are typically multiple control inputs, the actuation noise is in general a vector. For a differential drive robot, for instance, we could take the instantaneous linear velocities of the left and right wheels, (v_l, v_r) as the control inputs, in which case the perturbed inputs would be

$$\begin{bmatrix} \hat{v}_l \\ \hat{v}_r \end{bmatrix} = \begin{bmatrix} v_l \\ v_r \end{bmatrix} + \begin{bmatrix} \varepsilon_l \\ \varepsilon_r \end{bmatrix}$$

Alternatively, we could perturb the instantaneous velocities directly:

$$\begin{bmatrix} \hat{v} \\ \hat{\omega} \end{bmatrix} = \begin{bmatrix} v \\ \omega \end{bmatrix} + \begin{bmatrix} \varepsilon_v \\ \varepsilon_\omega \end{bmatrix}$$

With this noise model, however, the motion of the robot is constrained to the circle about the instantaneous center of curvature with radius $R = \hat{v}/\hat{\omega}$. While the realistic motion of the robot has a nonzero probability of being in any three-dimensional state after apply the control action, the robot would be constrained to the two-dimensional subspace lying on this circle. To remedy this, we also perturb the final orientation by an angle ε_θ so that

$$\theta' = \theta + \Delta\theta + \varepsilon_\theta$$

We now use this probabilistic model to sample future states and to determine the probability of being in a given future state given the current state and the control actions.

Sampling states

We can use the forward model to sample future states (x', y', θ') of the robot given its current state (x, y, θ) and the (constant) control actions u applied over a time period of Δt as follows:

a) sample the actuation noise $\varepsilon = (\varepsilon_u, \varepsilon_\theta)$, which perturbs both the control actions u (such as the instantaneous velocities, circle radius and change in orientation, wheel distances, etc.) and the final orientation θ'

b) construct the perturbed control actions $\hat{u}$ and final rotation $\hat{\phi}$

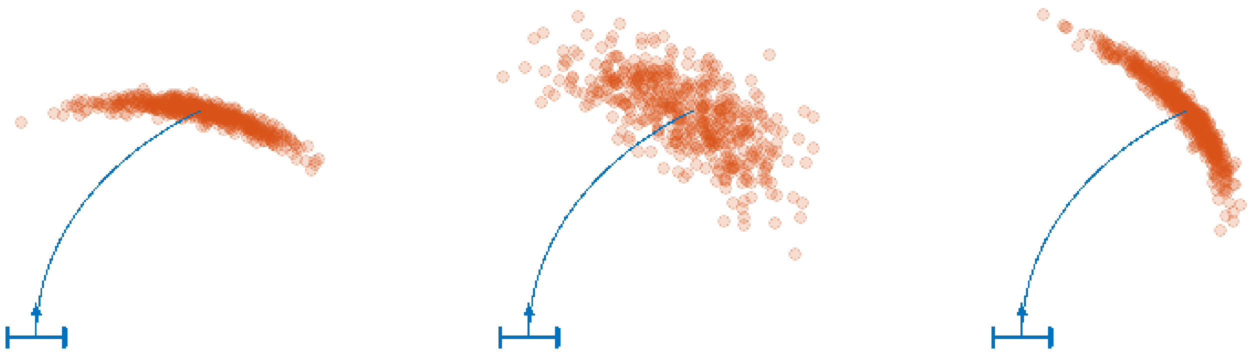
$$\hat{u} = u + \varepsilon_u$$

c) apply the perturbed control actions $\hat{u}$ to the forward model to obtain the next state (x', y', θ')

d) perturb the final orientation

$$\theta' \leftarrow \theta' + \varepsilon_\theta$$

This produces a robot state (x', y', θ') that is sampled from the motion model according to the distribution of the actuation noise.



The above figure shows multiple samples of the foward probabilistic actuation model for a differential drive robot using various distributions for the actuation noise. In each case, the actuation noise has zero mean, so the points are centered about the future state obtained from the deterministic model (shown in blue). The shape of the samples depends on the distribution of the actuation noise.

Probability of states

Instead of sampling the robot state, we can also compute the probability $p(s' \mid s, a)$ of the future robot state s' given the current robot state s and control action a.

To find the probability that the robot is in some state (x', y', θ') given that it is in state (x, y, θ) and applies the control actions u, we can do the following:

a) use the backward model to find the control inputs $\hat{u}$ that move the robot from state (x, y, θ) to position (x', y'), and find the corresponding change in orientation $\Delta\theta$

b) compute the actuation noise, which is the difference between the commanded inputs and the inputs required to move the robot between the two states

$$\varepsilon_u = \hat{u} - u$$

c) compute the final rotation required for the robot to have final orientation θ'

$$\varepsilon_\theta = \theta' - \theta - \Delta\theta$$

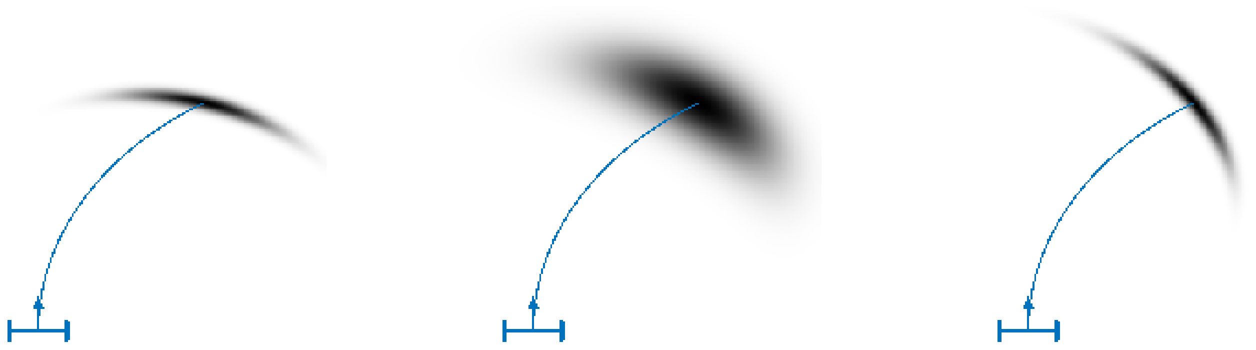
d) use the probability distribution of the actuation noise to compute the probability of the next state given the current state and control action

$$p(s' \mid s, u) \quad = \quad p(\varepsilon \mid s', s, u) \quad = \quad p(\varepsilon_u \mid s', s, u)\, p(\varepsilon_\theta \mid s', s, u)$$

where the last equality assumes that the noise in the control inputs and the final rotation are independent

This produces the probability that the robot is in state (x', y', θ') given that it is in state (x, y, θ) and takes control actions a.

The figure below illustrates the probability distribution of the future robot state given an initial state and control actions (indicated by the circular trajectory) for several different actuation noise distributions. The final robot state after applying this control action is a random variable whose probability distribution is shown in gray. Darker colors indiciate a higher probability of being in that state. In each case, the actuation noise has zero mean, so the points are centered about the future state obtained from the deterministic model. The shape of the distribution depends on the distribution of the actuation noise.



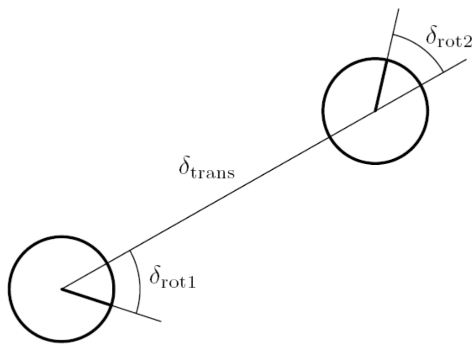
2.5 Odometry

Odometers provide relative pose information as a robot moves. Odometry information may come from wheel encoders that measure the change in angle of each wheel, or from other sensors such as a camera viewing the same object in the environment from multiple poses.

Remark. When using odometry, it is important to remember that odometry information is not actually a control action! Odometry information is obtained by *measuring* relative pose information. We treat odometry information as control actions since it relates the state at two different times; treating odometry as measurements would require increasing the dimension of the state to include velocity information (or state information at two subsequent times). Treating odometry as control actions keeps the state small, which is important for practical implementation of some algorithms whose complexity scales with the state dimension. Since odometry is not a control action, it cannot be used to control the system, but we can still use odometry for estimation, such as in localization and mapping problems. ■

There are various ways to describe relative pose information for planar mobile robots. One such method describes the robot motion in terms of three consecutive actions: a rotation, a translation, and then a second rotation. These three actions are sufficient to describe the motion of a robot between any two poses in the plane.

Suppose a robot is in state (x, y, θ) and measures the relative odometry information $u = (\phi_1, d, \phi_2)$ to its next state (x', y', θ'). This means that the robot moved to its next state by rotating by angle ϕ_1, translating forward by distance d, and then rotating by angle ϕ_2.



Forward model

Given that the robot was in state (x, y, θ) and measured the odometry (ϕ_1, d, ϕ_2), the next state of the robot is

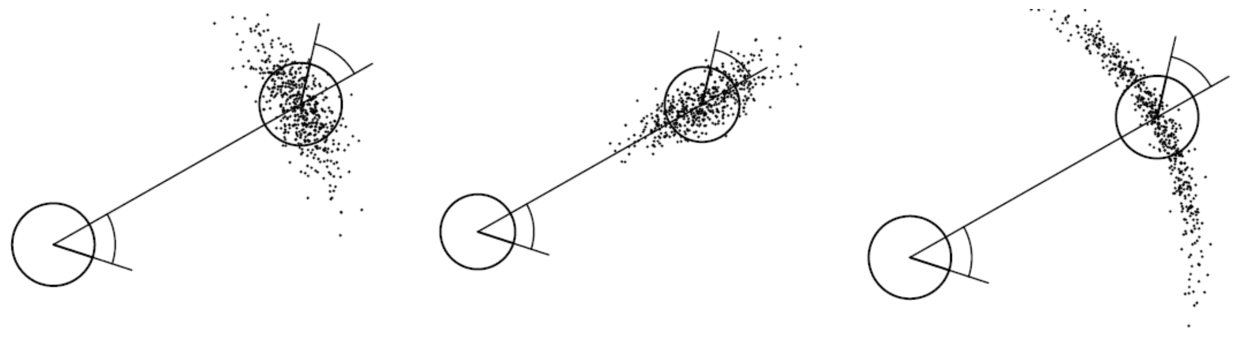
$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} d\cos(\theta + \phi_1) \\ d\sin(\theta + \phi_1) \\ \phi_1 + \phi_2 \end{bmatrix}$$

This model is deterministic in that it exactly specifies the next state of the robot given its current state and odometry. Realistic robotics, however, are uncertain. To more accurately represent the motion of the robot, we can use a probabilistic model. A simple model is to perturb the odometry measurements before applying the deterministic model. For instance, a probabilistic odometry-based model for a planar mobile robot is

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} \hat{d}\cos(\theta + \hat{\phi}_1) \\ \hat{d}\sin(\theta + \hat{\phi}_1) \\ \hat{\phi}_1 + \hat{\phi}_2 \end{bmatrix} \qquad \text{where} \qquad \begin{bmatrix} \hat{\phi}_1 \\ \hat{d} \\ \hat{\phi}_2 \end{bmatrix} = \begin{bmatrix} \phi_1 \\ d \\ \phi_2 \end{bmatrix} + \begin{bmatrix} \varepsilon_{\phi_1} \\ \varepsilon_d \\ \varepsilon_{\phi_2} \end{bmatrix}$$

The parameters $(\varepsilon_{\phi_1}, \varepsilon_d, \varepsilon_{\phi_2})$ are random variables that represent the odometry noise. We first perturb the odometry information by the noise, and then apply this noisy odometry information to obtain the next state of the robot.

We can use this forward probabilistic model to sample future robot states. To do so, just sample the noise parameters (according to some distribution), and then compute the next robot state as above.



The above figure shows multiple samples of the foward probabilistic odometry-based actuation model using various distributions for the odometry noise. In each case, the odometry noise has zero mean, so the points are centered about the future state obtained from the deterministic model. The shape of the samples depends on the distribution of the odometry noise.

Backward model

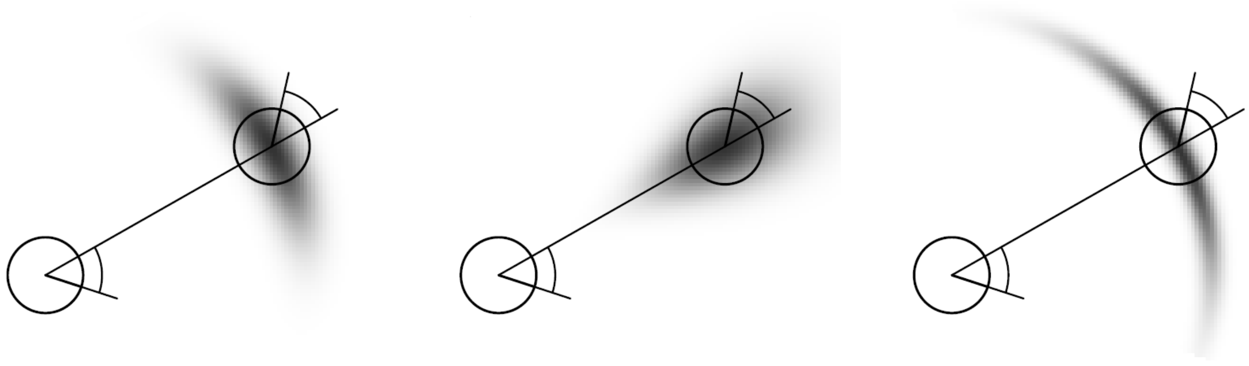
Given two poses, we can also find the odometry information that the robot would measure during the motion. Solving the forward model for the relative position and heading $(\Delta x, \Delta y, \Delta\theta)$ along with the original orientation θ, the corresponding odometry information is

$$\begin{bmatrix}\phi_1\\ d\\ \phi_2\end{bmatrix} = \begin{bmatrix}\operatorname{atan2}(\Delta y, \Delta x) - \theta\\ \sqrt{\Delta x^2 + \Delta y^2}\\ \Delta\theta - \operatorname{atan2}(\Delta y, \Delta x)\end{bmatrix}$$

We can use the backward model to find the probability that the robot is in a given state s' given that it was in state s and measured odometry (ϕ_1, d, ϕ_2). To do so, use the backward model to compute the odometry $(\hat{\phi}_1, \hat{d}, \hat{\phi}_2)$ that describe the relative pose information, then compute the odometry error as

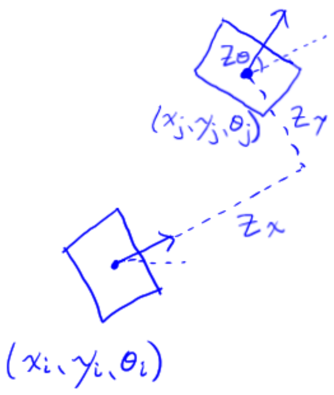
$$\begin{bmatrix}\varepsilon_{\phi_1}\\ \varepsilon_d\\ \varepsilon_{\phi_2}\end{bmatrix} = \begin{bmatrix}\hat{\phi}_1\\ \hat{d}\\ \hat{\phi}_2\end{bmatrix} - \begin{bmatrix}\phi_1\\ d\\ \phi_2\end{bmatrix},$$

and then use the distribution of the odometry error to compute its probability.



Residual

We now construct the residual (or error) between the measured odometry and two given poses for a planar mobile robot. Suppose the robot measures the odometry (z_x, z_y, z_θ) from pose (x_i, y_i, θ_i) to pose (x_j, y_j, θ_j). The odometry describes pose j relative to pose i as shown below.



Let Z_{ij}, X_i, and X_j denote the homogeneous transformation matrices corresponding to the poses. Then the homogeneous transformation matrix $X_i^{-1}X_j$ maps homogeneous coordinates relative to pose j to those relative to pose i, which represents how node i sees node j. The transformation matrix Z_{ij} from the odometry also maps homogeneous coordinates relative to pose j to those relative to pose i. Both $X_i^{-1}X_j$ and Z_{ij} represent the same transformation, but one is constructed from the two poses while the other is measured by the odometry. The error of the measurement is then the difference between these transformations. This is described by the homogeneous transformation $Z_{ij}^{-1}(X_i^{-1}X_j)$, which maps coordinates relative to pose j to pose i and then back to pose j through the odometry measurement. This transformation is given explicitly by

$$Z_{ij}^{-1}(X_i^{-1}X_j) = \begin{bmatrix} \cos(\varepsilon_\theta) & -\sin(\varepsilon_\theta) & (x_j - x_i)\cos(\theta_i + z_\theta) + (y_j - y_i)\sin(\theta_i + z_\theta) - z_x \cos z_\theta - z_y \sin z_\theta \\ \sin(\varepsilon_\theta) & \cos(\varepsilon_\theta) & (x_i - x_j)\sin(\theta_i + z_\theta) - (y_i - y_j)\cos(\theta_i + z_\theta) + z_x \sin z_\theta - z_y \cos z_\theta \\ 0 & 0 & 1 \end{bmatrix}$$

where $\varepsilon_\theta = \theta_j - \theta_i - z_\theta$. To measure the distance between the transformation and the identity, we use the pose corresponding to this homogeneous transformation. If the measurement were exact, then the transformation matrix would be the identity, in which case the pose would be $(0, 0, 0)$. The residual associated with an odometry-based edge is then

$$r_{ij} = \begin{bmatrix} (x_j - x_i)\cos(\theta_i + z_\theta) + (y_j - y_i)\sin(\theta_i + z_\theta) - z_x \cos z_\theta - z_y \sin z_\theta \\ -(x_j - x_i)\sin(\theta_i + z_\theta) + (y_j - y_i)\cos(\theta_i + z_\theta) + z_x \sin z_\theta - z_y \cos z_\theta \\ \theta_j - \theta_i - z_\theta \end{bmatrix}$$

Caution! The angle $\theta_j - \theta_i - z_\theta$ should be mapped to the interval $[-\pi, \pi]$ to prevent it from wrapping around.

The Jacobian of the residual with respect to the first pose x_i is

$$\frac{\partial r_{ij}}{\partial x_i} = \begin{bmatrix} -\cos\phi & -\sin\phi & (x_i - x_j)\sin\phi - (y_i - y_j)\cos\phi \\ \sin\phi & -\cos\phi & (x_i - x_j)\cos\phi + (y_i - y_j)\sin\phi \\ 0 & 0 & -1 \end{bmatrix}$$

where $\phi = \theta_i + z_\theta$, and the Jacobian of the residual with respect to the second pose x_j is

$$\frac{\partial r_{ij}}{\partial x_j} = \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3

Perception

Perception is the method by which a robot gains information about itself and the surrounding environment. Robots perceive their environment using sensors, which are devices that provide measurements related to quantities of interest. In this chapter, we study probabilistic models that describe how a sensor perceives the state of the robot and the environment.

3.1 Sensors

We can characterize sensors based on a variety of criteria.

- **Type of information**
  - proprioceptive: measure information internal to the robot
  - exteroceptive: measure information about the environment
- **Energy source**
  - active: emit energy into the environment and measure the reaction
  - passive: measure ambient environmental energy
- **Type of measurement**
  - range, bearing, speed, acceleration, temperature, color, image, sound, angle, magnetic field, air pressure, force

Example (List of sensors). camera, optical encoder, sonar, laser, potentiometer, compass, contact sensor, GPS, gyroscope, accelerometer, thermometer, magnetometer, barometer, strain gauge

3.2 Models

A sensor model is a mathematical relationship between the quantity of interest, the state of the robot or its environment, and the corresponding measurement. Given a robot with state s and a map m of the environment, a perception model describes the probability that the sensor perceives a measurement z,

$$p(z \mid s, m)$$

Another way of representing a perception model is of the form

$$z = h(s, m, \delta)$$

where δ is a random variable that represents the perception noise.

Measurement noise

The measurement noise represents the uncertainty in the measured value from the sensor. As it is a random variable, it has a probability distribution $\delta \sim p(\delta)$. Note that the measurement noise may be a scalar or a vector. We typically make the following assumptions on the measurement noise.

- We often assume that the sensor is *unbiased*, meaning that the measurement noise has zero mean.

$$\mathcal{E}(\delta) = 0$$

- The variance of the measurement noise represents the amount of uncertainty in the measurement, with a larger variance corresponding to more uncertainty. We often assume that the individual measurements are *uncorrelated*, in which case the covariance matrix is diagonal.

$$\mathbf{cov}(\delta) = \begin{bmatrix} \sigma_1^2 & 0 & \dots & 0 \\ 0 & \sigma_2^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_K^2 \end{bmatrix}$$

- We often assume that the measurements are *uncorrelated* across time, which means that their is no (linear) relationship between them.

$$\mathbf{cov}(\delta_{t_1}, \delta_{t_2}) = 0 \qquad \text{for } t_1 \neq t_2$$

- For vector-valued measurements, we sometimes assume that the individual measurements are conditionally independent given the state of the robot and environment. For a vector of measurements $z = (z^1, z^2, \dots, z^K)$,

$$p(z \mid s, m) = \prod_{k=1}^{K} p(z^k \mid x, m)$$

3.3 Range finder

A range finder is a sensor that measures the distance (in a particular direction) to nearby objects. Range finders are active sensors that emit a signal into the environment and measure the time for the signal to return to the sensor. Range finders can be classified based on the type of signal that they emit; for instance, a laser emits light while an ultrasonic sensor emits sound. The measurements are affected by the type of signal emitted; for instance, light travels in a beam while sound travels in a cone.

Beam model

Consider a range finder in which the emitted signal is modeled as a beam (such as a laser). Range finders typically have a maximum detectable distance, which we denote $z_{\max}$. Given the state s and map m, we model the measurement perceived by a range finder as a random variable z that takes values in the real interval $[0, z_{\max}]$. We construct the probability of a measurement as a weighted sum of four densities, each of which corresponds to a particular type of error:

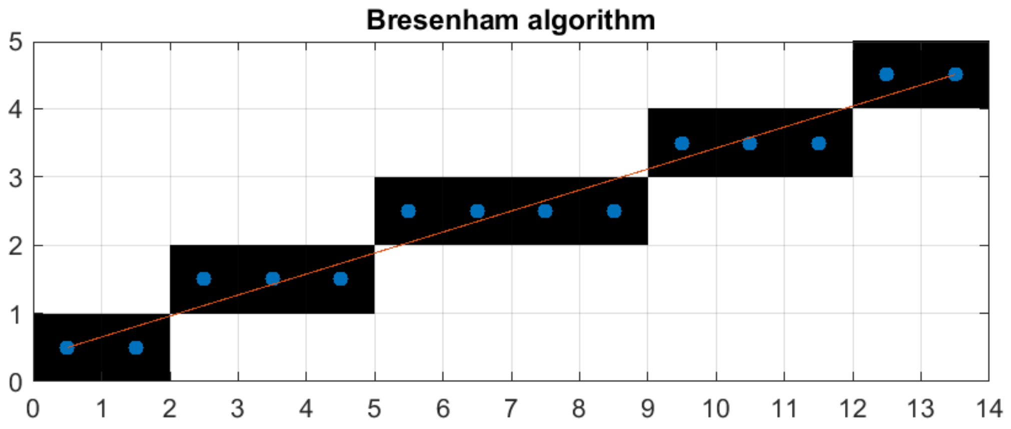
$$p(z \mid s, m) \quad = \sum_{i \in \{\text{hit,short,max,rand}\}} w_i \, p_i(z \mid s, m)$$

The weight w_i describes how much the corresponding error impacts the measurement relative to the other sources of error. For this to be a valid density, the weights must all sum to one,

$$w_{\text{hit}} + w_{\text{short}} + w_{\max} + w_{\text{rand}} = 1$$

Several of these densities depend on the true distance to the nearest object in the direction of the sensor, which we denote $z_\star$. The computation of the true distance depends on the type of map.

- In a location-based map, the true distance is found by *ray casting*. Here, a ray is drawn starting from the sensor in the direction of its orientation, and the ray is extended until it hits the first object. The distance of this ray from the sensor to the first object is the true distance $z_\star$. When the domain of the map is discrete, *Bresenham's algorithm* can be used to compute the points on the grid that the ray passes through.



- In a feature-based map, the true distance is typically taken as the the closest feature within the cone of the measurement.

We now describe each of the four densities that compose the probability of a measurement. These densities depend on the maximum range $z_{\max}$ of the sensor and the range to the nearest object $z_\star$.

- **Correct range with local measurement noise.** The first density represents the correct range with local measurement noise, which is represented by additive Gaussian noise. But simply adding Gaussian noise to the true distance $z_\star$ is not realistic, since the sensor values are limited to the range $[0, z_{\max}]$. So we instead use a Gaussian random variable that is clipped to be in this interval,

$$p_{\text{hit}}(z \mid s, m) = \begin{cases} \eta \, \mathcal{N}(z; z_\star, \sigma^2_{\text{hit}}) & \text{if } z \in [0, z_{\max}] \\ 0 & \text{otherwise} \end{cases}$$

where $\mathcal{N}(x; \mu, \sigma^2)$ is the normal distribution with mean μ and variance σ^2. Since the Guassian is clipped, we need the normalizing constant η so that this is a valid probability distribution. The value of the normalizing constant is

$$\eta = \left(\int_0^{z_{\max}} \mathcal{N}(z; z_\star, \sigma^2_{\text{hit}}) \, \mathrm{d}z \right)^{-1}$$

- **Unexpected objects.** Objects that are not represented in the map can cause the distance measured by the sensor to be smaller than the true distance. For example, a person could move in front of the sensor. We could include such objects in the map, but then the map would be dynamic. To keep things simple, we model such objects as sensor noise. Since objects closer to the sensor always obscure objects further away, the probability of perceiving an unexpected object decreases with distance from the sensor. We therefore model unexpected objects using an exponential distribution, which is parameterized by the exponent λ_{short}. Once again, we clip the density by the maximum range of the sensor,

$$p_{\text{short}}(z \mid s, m) = \begin{cases} \dfrac{\lambda_{\text{short}}}{1 - e^{-\lambda_{\text{short}} z_\star}} \, e^{-\lambda_{\text{short}} z} & \text{if } z \in [0, z_{\max}] \\ 0 & \text{otherwise} \end{cases}$$

The normalizing constant is chosen such that this is a valid probability distribution.

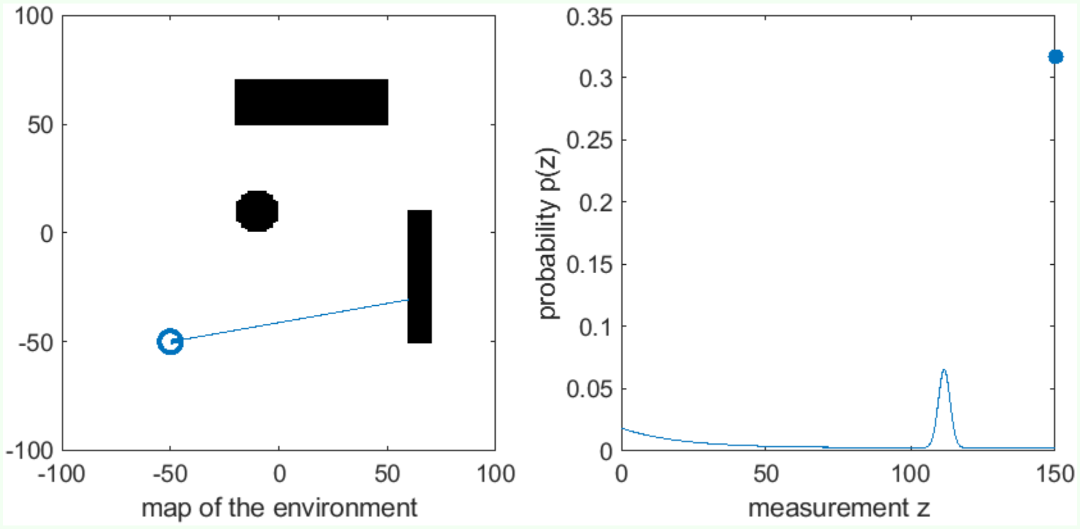
- **Failures.** Failures occur when the emitted signal does not return to the sensor in a reasonable amount of time. Such failures can be due to reflection of the signal, or the signal being absorbed by the object (such as a black object for a laser). In case of failure, the sensor typically returns its maximum value, $z_{\max}$. Therefore, we model failures using the (discrete) probability distribution

$$p_{\max}(z \mid s, m) = \begin{cases} 1 & \text{if } z = z_{\max} \\ 0 & \text{otherwise} \end{cases}$$

- **Random measurements.** The last type of error that we model are random measurements. Such measurements could be do to cross-talk between multiple sensors or phantom readings due to signals bouncing off walls. We model random measurements as a uniform distribution,

$$p_{\text{rand}}(z \mid s, m) = \begin{cases} \frac{1}{z_{\max}} & \text{if } z \in [0, z_{\max}] \\ 0 & \text{otherwise} \end{cases}$$

Example (Beam model). We now illustrate the beam model of a range finder using a discrete location-based map of the environment. The map of the environment is shown on the left, where dark regions correspond to obstacles. The sensor is located at the position $(-50, -50)$ with an orientation of 10 degrees. Given the maximum distance of the sensor $z_{\max} = 150$, we use Bresenham's algorithm to find the grid points between the sensor location and its extension by a distance of $z_{\max}$ in the direction of the sensor. The closest grid point containing an obstacle determines the true distance $z_\star$, which is the length of the blue line from the sensor to the nearest object.

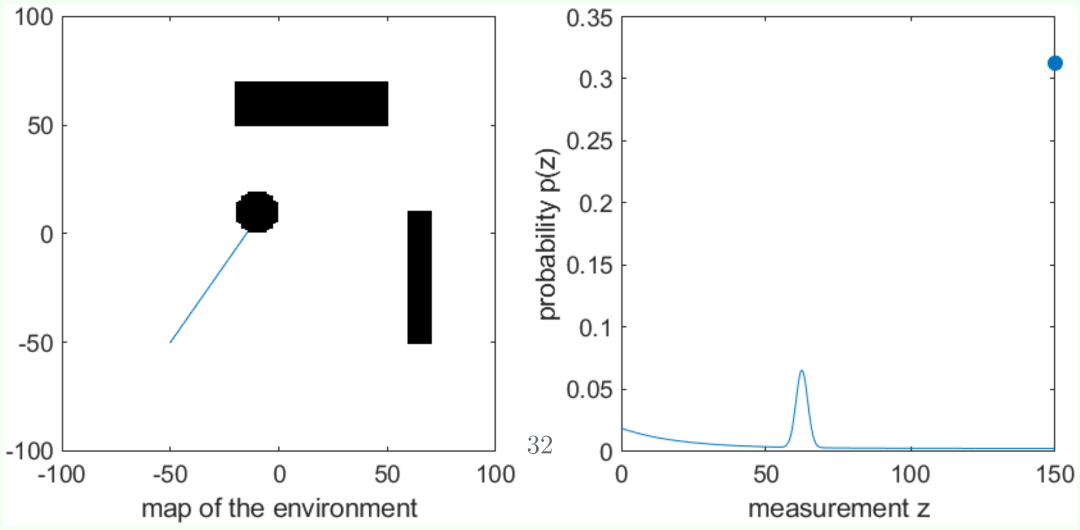


The maximum and true distances are then used to compute the four error densities with the following parameters:

$$\sigma_{\text{hit}} = 2 \quad \lambda_{\text{short}} = 0.05 \quad w_{\text{hit}} = 0.25 \quad w_{\text{short}} = 0.25 \quad w_{\max} = 0.25 \quad w_{\text{rand}} = 0.25$$

The probability that the sensor measures a given distance z using the beam model is shown on the right. There is a normal distribution about the true distance $z_\star$ that represents local measurement noise, an exponential distribution that represents unexpected objects, a discrete distribution at the maximum distance that represents sensor failures, and a uniform distribution that represents random measurements.

If the angle of the sensor is changed to 55 degrees, then the beam hits the circular object at a closer distance. This shifts the mean of the Gaussian distribution to the left since smaller measurements now have higher probability.



There are two main limitations of the beam model:

- **Lack of smoothness.** For the beam model, the probability that the sensor receives a measurement z can be highly discontinuous in the robot pose x and the map m. For instance, suppose that the beam of the sensor narrowly passes by an object. The beam model uses ray casting, which would simply continue until the beam directly reaches an obstacle. But an object near the beam may also reflect the signal causing the sensor to detect that (shorter) distance. Such nonsmooth models can be more difficult to use for estimation in that gradient-based algorithms can get stuck in local optima.
- **Complexity.** The beam model requires ray casting to find $z_\star$, which is a computationally expensive operation. This complexity can be mitigated by pre-computing the range to nearby objects at each pose, but this requires large look-up tables.

Likelihood field

The likelihood field is another sensor model for a range finder that overcomes some of the limitations of the beam model. Similar to the beam model, the likelihood field model of a range finder is composed of three simpler models.

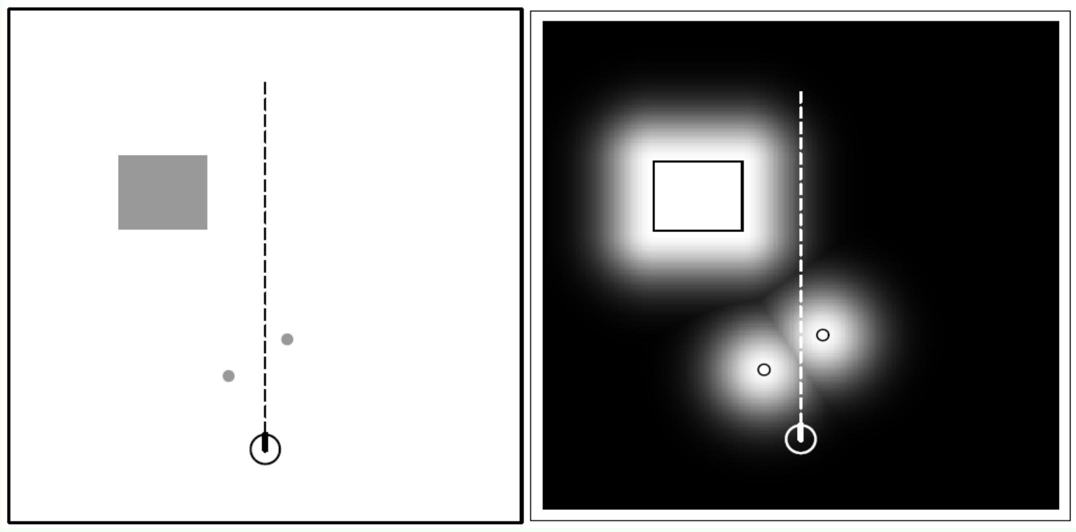
$$p(z \mid s, m) \quad = \sum_{i \in \{\text{field},\text{max},\text{rand}\}} w_i \, p_i(z \mid s, m)$$

The maximum distance readings due to sensor failures and the random mesurements are identical to those from the beam model. Instead of the models for the correct range with local measurement noise and unexpected objects, however, the likelihood field models measurement noise using Gaussians related to the distance to the nearest object. For any given measurement z, let d denote the distance from the measured point to the nearest obstacle in the map. The probability that the sensor receives the measurement z is then modeled as a Gaussian,

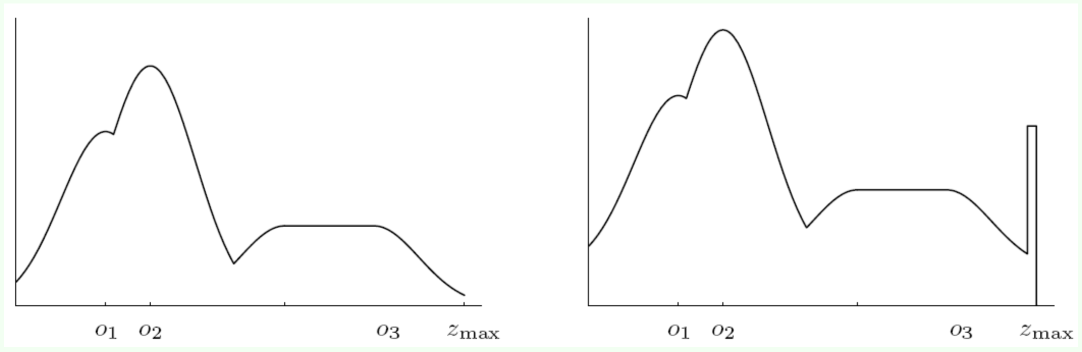
$$p_{\text{field}}(z \mid s, m) = \mathcal{N}(d, 0, \sigma_{\text{field}}^2)$$

where the parameter σ_{field} is the variance of the noise.

Example (Likelihood field)**.** Consider the map of the environment shown on the left, where dark regions indicate obstacles. The sensor is located near the bottom of the map and takes a range measurement along the vertical dashed line. The right plot shows the likelihood field, which is the probability of receiving a measurement at any point. For each (x, y), the probability is calculated by finding the distance to the nearest object in the map and evaluating the density of a Gaussian random variable with zero mean and variance σ^2_{field} at that distance.



The probability that a sensor receives a measurement z given the robot state x and map m is shown below. Even though the beam does not directly hit any objects, it does pass near the objects, denoted o_1, o_2, and o_3. Each of these objects results in an increased probability, depending on how close the beam passes to the object.



The likelihood field model has several advantages over the beam model. First, the likelihood field is smooth in the robot state x and map m. Small deviations in either of these lead to small deviations in the perception model. As with the beam model, we can pre-compute the likelihood field to increase the efficiency of the algorithm. While the beam model requires storing the distance to the closest object for every three-dimensional state (x, y, θ), the likelihood field model only needs to store the likelihood for each two-dimensional position (x, y).

While the likelihood field model has advantages, some disadvantages are that it does not explicitly model short sensor measurements, it treats sensors as if they can see through obstacles (since ray casting was replaced by finding the nearest neighbor, which may be behind another obstacle), and it does not take any uncertainty in the map into account which makes it unsuitable for unexplored areas.

3.4 Feature-based models

In contrast to using raw sensor measurements, another way to construct a perception model is to extract *features* from the measurements. A feature is a function of the measurement, which we denote $f(z)$. Examples of features include lines, corners, objects, distinct patterns, hallways, intersections, etc. We refer to features that represent physical objects (such as the corner of a building or a tree) as *landmarks*. The landmark is completely described by its location, which we denote in global coordinates as $\ell = (\ell_x, \ell_y)$. The map is then the collection of all landmarks,

$$m = \{\ell_1, \ell_2, \ell_3, \ldots\}$$

Range–bearing sensor

Consider a robot at state (x, y, θ) that uses a sensor to measure the range r and bearing ϕ to a known landmark ℓ. A simple probabilistic model for the range and bearing measurements is given by

$$z = \begin{bmatrix} r \\ \phi \end{bmatrix} = \begin{bmatrix} \sqrt{(\ell_x - x)^2 + (\ell_y - y)^2} \\ \operatorname{atan2}(\ell_y - y, \ell_x - x) - \theta \end{bmatrix} + \begin{bmatrix} \varepsilon_r \\ \varepsilon_\phi \end{bmatrix}$$

where the parameters $(\varepsilon_r, \varepsilon_\phi)$ represent perception noise. This models the sensor measurements as the true values perturbed by additive noise.

Probability of a measurement

We now use the range–bearing model to find the probability that the sensor perceives a given measurement. Given the robot state (x, y, θ) and the landmark position (ℓ_x, ℓ_y), the probability that the sensor perceives a measurement $z = (r, \phi)$ is

$$p(z \mid s, m) = p(\varepsilon \mid z, s, m)$$

where the noise of the measurement is

$$\begin{bmatrix} \varepsilon_r \\ \varepsilon_\phi \end{bmatrix} = \begin{bmatrix} r \\ \phi \end{bmatrix} - \begin{bmatrix} \sqrt{(\ell_x - x)^2 + (\ell_y - y)^2} \\ \operatorname{atan2}(\ell_y - y, \ell_x - x) - \theta \end{bmatrix}$$

Typically, the noise parameters are independent, in which case their joint distribution is the product of the two individual distributions,

$$p(\varepsilon \mid z, s, m) = p(\varepsilon_r \mid z, s, m)\, p(\varepsilon_\phi \mid z, s, m)$$

Note that this model requires the robot to know which landmark it observed, meaning it has *known correspondence* between the measurements and the landmarks.

Probability and sampling robot states

To find the probability that the robot is in a state s given the measurement z and map m, we can use Bayes rule to obtain

$$p(s \mid z, m) = \frac{p(z \mid s, m)\, p(s \mid m)}{p(z \mid m)}$$

Suppose that, given the map, the robot has the same probability of being in any state (which is often not the case!). Then the conditional probability $p(s \mid m)$ is constant. In this case, the probability of being in some state s is the same as the probability of perceiving some measurement z,

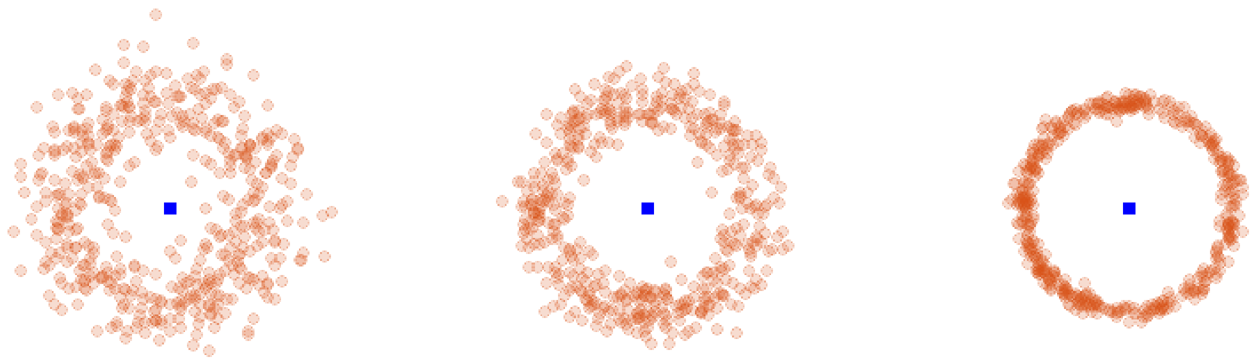
$$p(s \mid z, m) \propto p(z \mid s, m) = p(\varepsilon \mid z, s, m)$$

where the perception noise is given above in terms of the measurement z, state s, and map m.

It is also sometimes useful to sample robot states that correspond to a given measurement. To sample robot states, we can solve the perception model for the robot position

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \ell_x \\ \ell_y \end{bmatrix} + (r - \varepsilon_r) \begin{bmatrix} -\cos(\theta + \phi - \varepsilon_\phi) \\ \sin(\theta + \phi - \varepsilon_\phi) \end{bmatrix}$$

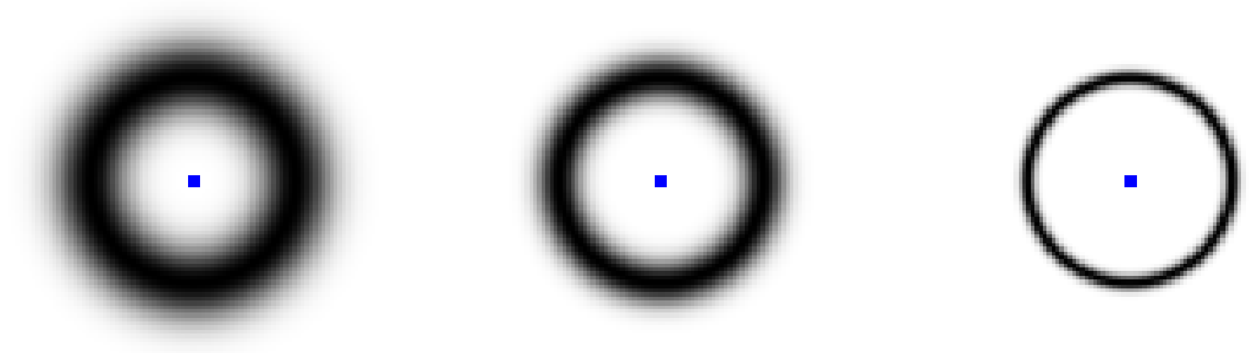
Since there is a position (x, y) that corresponds to the measurement (r, ϕ) for any orientation θ, we sample θ uniformly in the interval $[0, 2\pi)$ and then compute the corresponding position (x, y) to sample robot states. The following figure shows the positions (x, y) of sampled states (shown in orange) given a measurement of the landmark (shown in blue) for several noise distributions.



We can also compute the probability of a state given a measurement. Once again, since any orienation θ can correspond to a given position (x, y) and measurement (r, ϕ), the probability of a state (x, y, θ) given a measurement (r, ϕ) is

$$p(s \mid z, m) = p(\varepsilon_r \mid x, y, \ell_x, \ell_y) \qquad \text{where} \qquad \varepsilon_r = r - \sqrt{(\ell_x - x)^2 + (\ell_y - y)^2}$$

The following figure shows the probability that the robot is at position (x, y) given a measurement (r, ϕ) of the landmark for several noise distributions.



Perception-based residual

We now construct the residuals and their Jacobians for several perception models for feature-based maps.

Two distance sensors

Suppose the robot is at pose (x, y, θ) and perceives the measurement (z_x, z_y) of the relative distances in both the x and y directions to a landmark with global coordinates (ℓ_x, ℓ_y). Assume that each measurement is corrupted by additive noise (δ_x, δ_y). In homogeneous coordinates, the perception model is

$$\underbrace{\begin{bmatrix} z_x \\ z_y \\ 1 \end{bmatrix}}_{\substack{\text{sensor} \\ \text{measurement in} \\ \text{robot coordinates}}} = \underbrace{\begin{bmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1 \end{bmatrix}^{-1}}_{\substack{\text{transformation from} \\ \text{global frame to robot} \\ \text{frame}}} \underbrace{\begin{bmatrix} \ell_x \\ \ell_y \\ 1 \end{bmatrix}}_{\substack{\text{landmark position} \\ \text{in global} \\ \text{coordinates}}} + \underbrace{\begin{bmatrix} \delta_x \\ \delta_y \\ 1 \end{bmatrix}}_{\text{perception noise}}$$

Using the expression for the inverse of a homogeneous transformation, the sensor measurement is

$$\begin{bmatrix} z_x \\ z_y \end{bmatrix} = R(\theta)^\mathsf{T} \begin{bmatrix} \ell_x - x \\ \ell_y - y \end{bmatrix} + \begin{bmatrix} \delta_x \\ \delta_y \end{bmatrix}$$

where $R(\theta)$ is the rotation matrix for the angle θ. The residual for this measurement is the perception noise (δ_x, δ_y), which is the following function of the robot pose, landmark location, and measurement:

$$r = \begin{bmatrix} \delta_x \\ \delta_y \end{bmatrix} = \begin{bmatrix} z_x \\ z_y \end{bmatrix} - R(\theta)^\mathsf{T} \begin{bmatrix} \ell_x - x \\ \ell_y - y \end{bmatrix}$$

The residual is affine (linear plus constant) in the robot position (x, y), so this would lead to a linear least squares problem if we know the orientation θ of the robot and the position (ℓ_x, ℓ_y) of the landmark. But if we do not know these quantities, then the residual is nonlinear in the robot pose and landmark location $(x, y, \theta, \ell_x, \ell_y)$, so this requires solving a nonlinear least squares problem.

The Jacobian of the residual with respect to the robot pose is

$$\frac{\partial r}{\partial x} = \begin{bmatrix} \cos\theta & \sin\theta & (\ell_x - x)\sin\theta - (\ell_y - y)\cos\theta \\ -\sin\theta & \cos\theta & (\ell_y - y)\sin\theta + (\ell_x - x)\cos\theta \end{bmatrix}$$

and the Jacobian of the residual with respect to the landmark position is

$$\frac{\partial r}{\partial \ell} = \begin{bmatrix} -\cos\theta & -\sin\theta \\ \sin\theta & -\cos\theta \end{bmatrix}$$

Range–bearing sensor

Suppose the robot is at pose (x, y, θ) and perceives the measurement (z_r, z_ϕ) of the relative range and bearing to a landmark with global coordinates (ℓ_x, ℓ_y). Assume that each measurement is corrupted by additive noise (δ_x, δ_y). The perception model for this measurement is then

$$\begin{bmatrix} z_r \\ z_\phi \end{bmatrix} = \begin{bmatrix} \sqrt{(\ell_x - x)^2 + (\ell_y - y)^2} \\ \operatorname{atan2}(\ell_y - y, \ell_x - x) - \theta \end{bmatrix} + \begin{bmatrix} \delta_r \\ \delta_\phi \end{bmatrix}$$

The residual for this measurement is the perception noise, which is the following function of the robot pose, landmark location, and measurement:

$$r = \begin{bmatrix} z_r \\ z_\phi \end{bmatrix} - \begin{bmatrix} \sqrt{(\ell_x - x)^2 + (\ell_y - y)^2} \\ \operatorname{atan2}(\ell_y - y, \ell_x - x) - \theta \end{bmatrix}$$

The residual is nonlinear in the robot pose and landmark location $(x, y, \theta, \ell_x, \ell_y)$, so this requires solving a nonlinear least squares problem.

The Jacobian of the residual with respect to the robot pose is

$$\frac{\partial r}{\partial x} = \begin{bmatrix} \dfrac{\ell_x - x}{\sqrt{(\ell_x - x)^2 + (\ell_y - y)^2}} & \dfrac{\ell_y - y}{\sqrt{(\ell_x - x)^2 + (\ell_y - y)^2}} & 0 \\ \dfrac{-(\ell_y - y)}{\sqrt{(\ell_x - x)^2 + (\ell_y - y)^2}} & \dfrac{\ell_x - x}{\sqrt{(\ell_x - x)^2 + (\ell_y - y)^2}} & 1 \end{bmatrix}$$

and the Jacobian of the residual with respect to the landmark location is

$$\frac{\partial r}{\partial \ell} = \frac{1}{\sqrt{(\ell_x - x)^2 + (\ell_y - y)^2}} \begin{bmatrix} -(\ell_x - x) & -(\ell_y - y) \\ (\ell_y - y) & -(\ell_x - x) \end{bmatrix}$$

Part II

Filtering

4

Bayes Filter

The Bayes filter is a general algorithm for recursively updating an estimate for the state of a dynamical system given the control actions and measurements. The state of the algorithm is the *belief*, which is the probability that the system is in a particular state given all known information. In the Bayes filter, this belief is updated whenever the system applies a control action or receives a measurement. The Bayes filter can only be implemented exactly in several particular cases. When it cannot be implemented exactly, the belief is often approximated in various ways, each of which leads to a particular instance of the Bayes filter. Some particular instances (or approximations) of the Bayes filter are the Kalman filter and its extended version, the unscented Kalman filter, the particle filter, the histogram filter, and many more. This chapter introduces the generic Bayes filter, while the next chapter describes many of its specific instances.

4.1 Belief

In robotics, the belief reflects the robot's internal knowledge of its own state and that of the environment at any given time. Robots are not typically able to directly measure their pose. Even with a global positioning system (GPS), measurements are still noisy. To take this uncertainty into account, we model the belief of the robot's state and the map of the environment as the probability of each quantity given all previous information available. We denote the belief at time t as b_t. The quantity to be estimated depends on the particular problem.

- **Localization.** In localization, the robot constructs its belief about its state, which is the conditional probability distribution
$$b_t(s_t) = p(s_t \mid z_{1:t}, a_{1:t})$$
where s_t is the state of the robot at time t, and $z_{1:t}$ and $a_{1:t}$ are the sets of all measurements and control actions up to time t, respectively.
- **Mapping.** To construct a map, the robot would estimate the belief of the map,
$$b_t(m_t) = p(m_t \mid z_{1:t}, a_{1:t})$$
- **SLAM.** To simultaneously construct a map and localize itself within the map (the SLAM problem), the robot would estimate the joint belief of both states,
$$b_t(s_t, m_t) = p(s_t, m_t \mid z_{1:t}, a_{1:t})$$

Since the following discussion applies to each of these scenarios, we will study the Bayes filter to estimate the belief of a generic state s that may refer to the robot state, the map, or both.

Notation. To simplify the notation, we often drop the subscripts denoting time. Let s denote the state (at some generic time), and let i denote all information available at that time (which includes all previous measurements and control actions as well as any prior knowledge about the state). Then belief is then

$$b(s) = p(s \mid i)$$

Now suppose that we gain some new information, such as observing a measurement or applying a control action. Let i' denote all available information after gaining this new information (which includes all previous information i). The posterior belief after incorporating the new information is then denoted

$$b'(s) = p(s \mid i')$$

To summarize, $b(s)$ denotes the *prior* belief before obtaining the new information, and $b'(s)$ denotes the *posterior* belief after fusing the new information. ■

4.2 Bayes rule

The Bayes filter is based on Bayes rule which states that any conditional probability is equivalent to

$$p(x \mid y) = \frac{p(y \mid x)\, p(x)}{p(y)} = \frac{\text{likelihood} \cdot \text{prior}}{\text{evidence}}$$

This follows directly from the definition of conditional probability since

$$p(x, y) = p(x \mid y)\, p(y) = p(y \mid x)\, p(x)$$

Typically, the random variable x is the quantity that we are trying to estimate, and the random variable y is the available information. Bayes rule describes the probability that the quantity of interest has some value x given the available information y using three terms:

- **Likelihood.** The likelihood $p(y \mid x)$ is the probability of measuring y given x. Since the measurement y is known, we can think of this as a function of the unknown x, although it is not a probability distribution in x.
- **Prior.** The prior $p(x)$ describes the probability of an estimate before fusing the information.
- **Evidence.** The evidence $p(y)$ is the probability of the measurement independent of the unknown x. As a function of x, the evidence is constant so we typically absorb it into a normalizing constant:

$$p(x \mid y) = \eta\, p(y \mid x)\, p(x) \qquad \text{where} \qquad \eta = \frac{1}{p(y)}$$

If we have other background knowledge (such as previous measurements), then we can condition all probabilities in Bayes rule by this extra information:

$$p(x \mid y, z) = \frac{p(y \mid x, z)\, p(x \mid z)}{p(y \mid z)}$$

Example (COVID testing). At the height of the COVID-19 pandemic, roughly 2% of the population was infected with the virus. Suppose the probability that a test is positive given that the person is healthy (a false positive) is 0.01, and the probability that the test is positive given that the person has the virus is 0.95. From the law of total probability, the probability of a positive test is

$$\begin{aligned} p(\text{positive}) &= p(\text{positive} \mid \text{virus})\, p(\text{virus}) + p(\text{positive} \mid \text{healthy})\, p(\text{healthy}) \\ &= (0.95)(0.02) + (0.01)(0.98) \\ &= 0.03 \end{aligned}$$

We can then use Bayes rule to find the probability of having the virus given a positive test as

$$p(\text{virus} \mid \text{positive}) \quad = \quad \frac{p(\text{positive} \mid \text{virus})\, p(\text{virus})}{p(\text{positive})} \quad = \quad \frac{(0.95)\,(0.02)}{(0.03)} \quad = \quad 0.66$$

In other words, the probability of *not* having the virus given a positive test is 0.34.

4.3 Description

The Bayes filter iteratively updates the belief as the robot applies control actions and receives measurements. We describe the perception and actuation updates separately.

Perception update

Given the current belief $b(s)$, the belief after observing a measurement z is

$$b'(s) = \eta\, p(z \mid s)\, b(s)$$

where η is a normalizing constant. The measurement probability $p(z \mid s)$ describes the probability of measuring z given state s. The normalizing constant η is chosen such that the belief is a probability distribution (and therefore has total probability one).

Proof. Let s denote the state before observing the measurement, and let i denote all information that is available before observing the measurement — such as all past control actions and measurements — so that the belief is $b(s) = p(s \mid i)$. Now suppose that we observe the measurement z. Using Bayes rule and the fact that the system is in the same state before and after the measurement, the belief after observing the measurement is

$$b'(s) = p(s \mid z, i) = \frac{p(z \mid s, i)\, p(s \mid i)}{p(z \mid i)}$$

Since the state s completely describes the system (by definition), knowing both the state and the past information is the same as knowing only the state, so $p(z \mid s, i) = p(z \mid s)$. Making these simplifications in the above equation produces the perception update of the Bayes filter, where the constant is $\eta = p(z \mid i)$. ■

Actuation update

Given the current belief $b(s)$, the belief after applying a control action a is

$$b'(s') \quad = \quad \int p(s' \mid s, a)\, b(s)\, \mathrm{d}s \qquad \text{or} \qquad \sum_s p(s' \mid s, a)\, b(s)$$

where the integral is used if the state s is continuous and the sum if it is discrete. The transition probability $p(s' \mid s, a)$ describes the probability that the robot is in state s' given that it was in state s and applied the control action a.

Proof. As in the perception update, let s denote the state before applying the control action and let i denote all of the information that is available before applying the control action so that the belief is $b(s) = p(s \mid i)$. Now suppose that we apply the control action a, and let s' denote the state after applying the control action. Using the law of total probability, the belief after applying the control action (assuming that the state is continuous) is

$$b'(s') = p(s' \mid a, i) = \int p(s' \mid s, a, i)\, p(s \mid a, i)\, \mathrm{d}s$$

Since i denotes all information available before applying the control action, knowing the previous state, the control action, and all previous information is equivalent to knowing only the previous state and the control action, so $p(s' \mid s, a, i) = p(s' \mid s, a)$. Also, knowing the control action tells us nothing about the state before the action was applied, so $p(s \mid a, i) = p(s \mid i) = b(s)$. Making these simplifications in the above equation produces the actuation update of the Bayes filter. ■

Initialization

The Bayes filter iteratively updates the belief, which is a probability distribution. To implement the filter, the belief must be initialized as some probability distribution. There are several typical cases:

- *Full information.* If we know the initial state exactly, then the initial belief is a point mass at this state.
- *No information.* If we know nothing about the initial state, then the initial belief is a uniform distribution over the states.

If we know some information about the initial state, then this information can be encoded in an appropriate probability distribution (such as a Gaussian).

Markov assumption

The Bayes filter assumes that the state s completely describes the system in that past and future states are independent conditioned on the current state. This is known as the Markov assumption. There are several reasons why this assumption may not be valid:

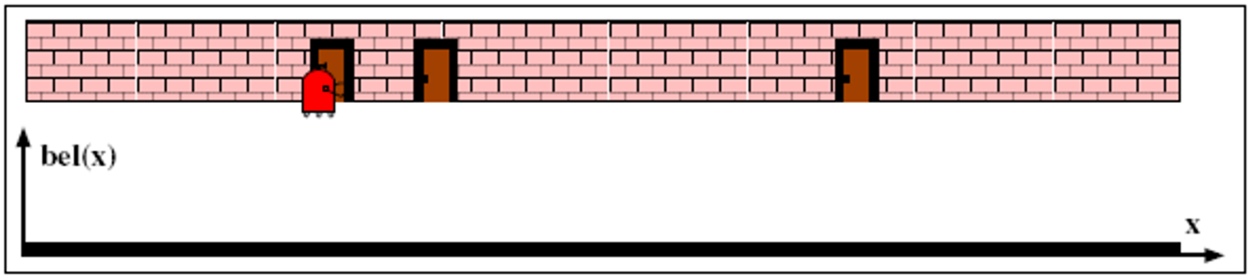
- The robot and/or environment may have unmodeled dynamics. Possible sources of unmodeled dynamics are moving people in the environment, and only using the pose (without velocity information) as the state of the robot.
- The probabilistic models $p(s' \mid s, a)$ and $p(z \mid s)$ may be inaccurate.
- The belief may be approximated which leads to approximation errors.

While the Markov assumption does not always hold, the Bayes filter is typically quite robust to such violations.

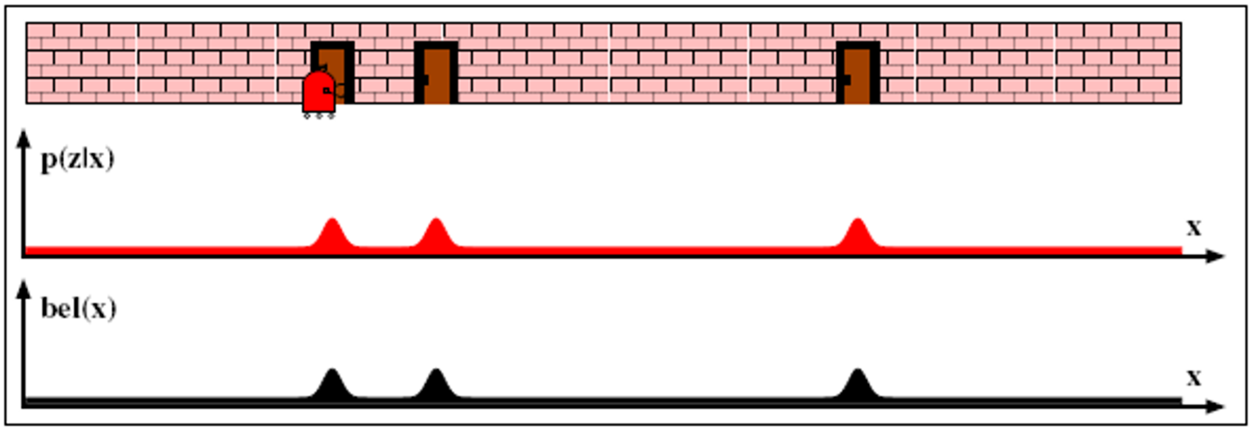
Examples

Qualitative example. The following example illustrates the basic concepts of the Bayes filter. Here, a robot uses a Bayes filter to localize its position in a one-dimensional environment with three doors. The robot has a sensor (such as a camera) that is able to detect a door, and the robot knows the locations of the doors.

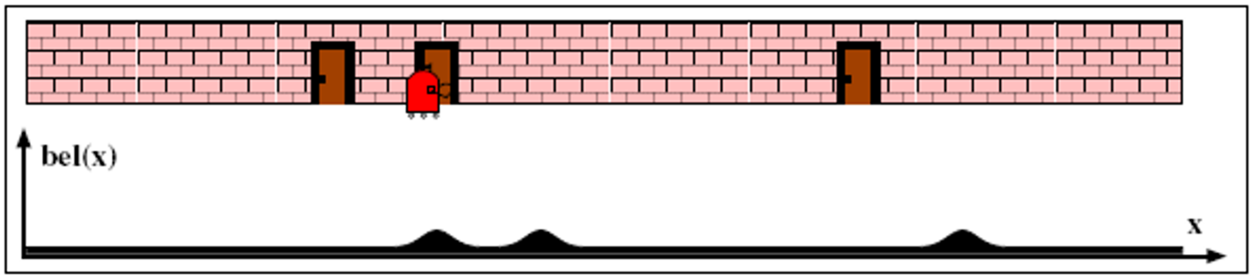
Initially, the robot does not know anything about its location, so the prior belief is uniformly distributed across the environment.



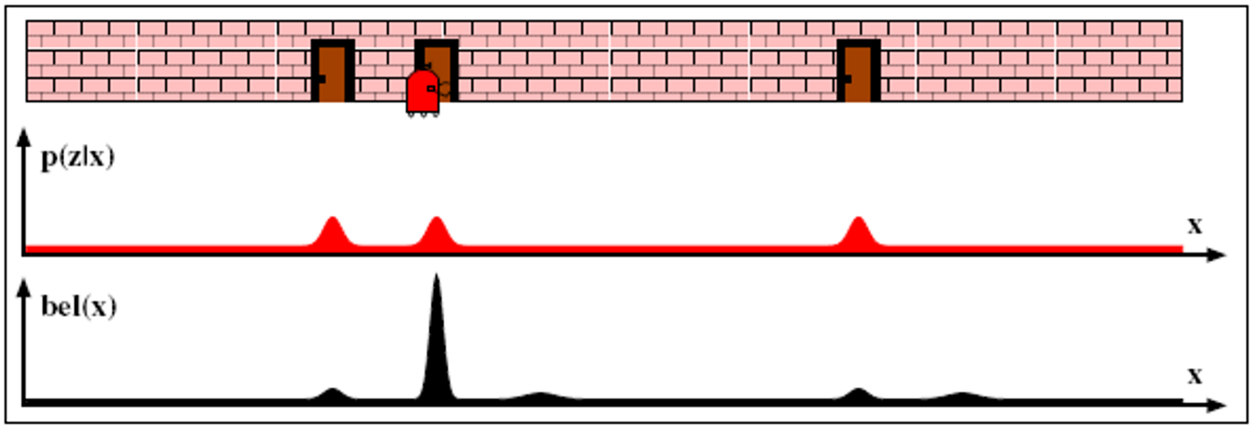
The robot then detects a door, and the probability of detecting a door is represented by three normal distributions centered about each door. After fusing this information using the perception update of the Bayes filter, the belief is now the same as the sensor model.



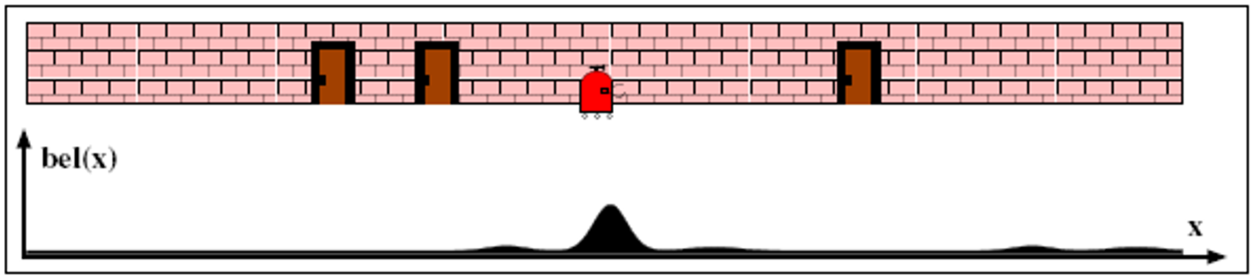
The robot then moves to the right. Because actuation diffuses the robot's information about its location, the belief after applying the actuation update in the Bayes filter is still three normal distributions but with larger variance.



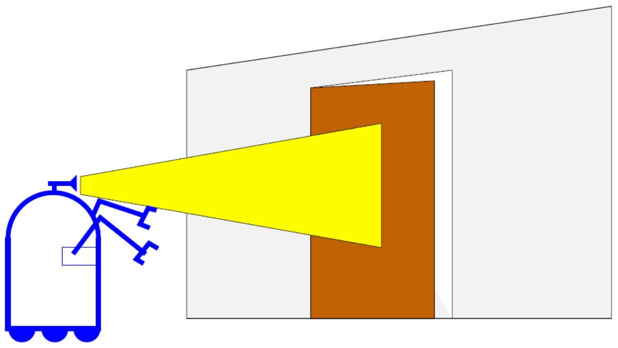
After observing another measurement, however, the robot is now much more certain that it is at the second door since this is where the sensor model best matches the prior belief.



And finally, the robot continues moving to the right, in which case its belief continues to spread out due to the noise in the actuation.



Quantitative example. We now illustrate the Bayes filter on a concrete example. Consider a robot that uses a sensor (such as a camera) to estimate whether a door is open or closed. The robot is also equipped with a manipulator that it can use to open or close the door.



The state s in this case is a binary variable that indicates whether the door is open or closed. To implement the Bayes filter, we need probabilistic models for the robot's perception and actuation.

- *Perception model.* Suppose the camera system is used in multiple experiments to detect whether the door is open or closed, and it is found that its measurement z of the state of the door satisfies the following probabilities:

$$p(z = \text{open} \mid s = \text{open}) = 0.6 \qquad p(z = \text{open} \mid s = \text{closed}) = 0.2$$
$$p(z = \text{closed} \mid s = \text{open}) = 0.4 \qquad p(z = \text{closed} \mid s = \text{closed}) = 0.8$$

These probabilities indicate that the sensor more accurately identifies a closed door than an open door.

- *Actuation model.* The robot can also use its manipulator to push the door open (or do nothing), although the action is not always successful. After applying the manipulator in multiple experiments to push the door open, it is found that the control action a satisfies the following probabilities:

$$\begin{aligned} p(s' = \text{open} \mid s = \text{open}, a = \text{push}) &= 1 & \qquad p(s' = \text{open} \mid s = \text{closed}, a = \text{push}) &= 0.8 \\ p(s' = \text{closed} \mid s = \text{open}, a = \text{push}) &= 0 & \qquad p(s' = \text{closed} \mid s = \text{closed}, a = \text{push}) &= 0.2 \end{aligned}$$

Let s_0 denote the initial state of the door. Suppose that the robot initially knows nothing about the state, so its initial belief has equal probability of being open and closed.

$$b(s_0 = \text{open}) = 0.5 \qquad \text{and} \qquad b(s_0 = \text{closed}) = 0.5$$

Now suppose the robot senses that the door is open, and let s_1 denote the state of the door after observing this measurement. Using the perception update of the Bayes filter, the belief of the two scenarios is

$$\begin{aligned} b(s_1 = \text{open}) &= \eta\, p(z = \text{open} \mid s_0 = \text{open})\, b(s_0 = \text{open}) = \eta\,(0.6)(0.5) = 0.3\eta \\ b(s_1 = \text{closed}) &= \eta\, p(z = \text{open} \mid s_0 = \text{closed})\, b(s_0 = \text{closed}) = \eta\,(0.2)(0.5) = 0.1\eta \end{aligned}$$

Since the belief must sum to one, the normalizing coefficient is $\eta = 1/(0.3 + 0.1) = 2.5$. The belief after taking into account the measurement is then

$$b(s_1 = \text{open}) = 0.75 \qquad \text{and} \qquad b(s_1 = \text{closed}) = 0.25$$

Now suppose the robot uses its manipulator to apply the control action $a = \text{push}$, and let s_2 denote the state of the door after applying the control action. Using the actuation update of the Bayes filter, the belief after applying the control action is

$$\begin{aligned} b(s_2 = \text{open}) = {} & p(s_2 = \text{open} \mid s_1 = \text{open}, a = \text{push})\, b(s_1 = \text{open}) \\ & + p(s_2 = \text{open} \mid s_1 = \text{closed}, a = \text{push})\, b(s_1 = \text{closed}) = (1)(0.75) + (0.8)(0.25) = 0.95 \end{aligned}$$

and

$$\begin{aligned} b(s_2 = \text{closed}) = {} & p(s_2 = \text{closed} \mid s_1 = \text{open}, a = \text{push})\, b(s_1 = \text{open}) \\ & + p(s_2 = \text{closed} \mid s_1 = \text{closed}, a = \text{push})\, b(s_1 = \text{closed}) = (0)(0.75) + (0.2)(0.25) = 0.05 \end{aligned}$$

Therefore, the belief after taking into account the control action is

$$b(s_2 = \text{open}) = 0.95 \qquad \text{and} \qquad b(s_2 = \text{closed}) = 0.05$$

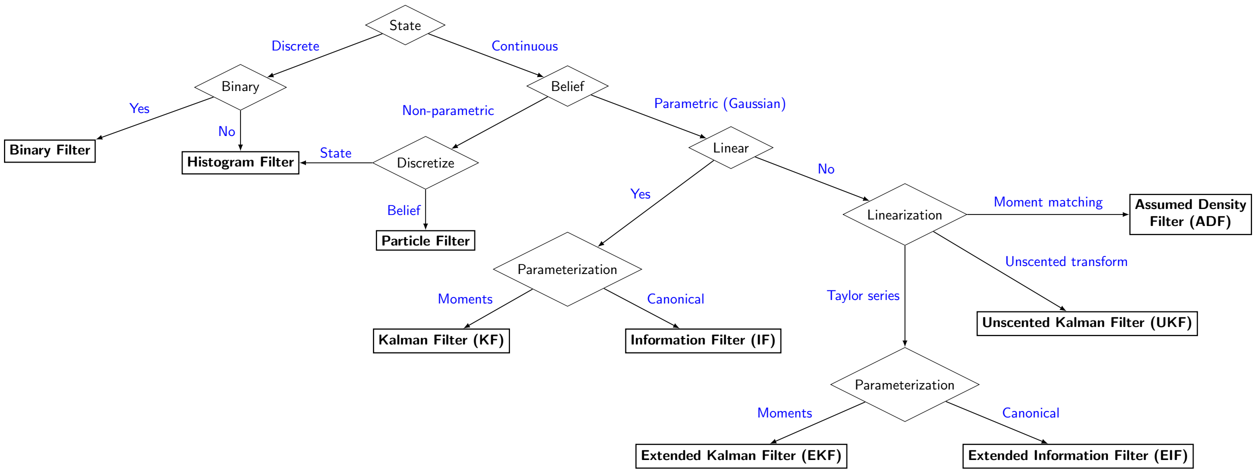
At this point, the robot believes with 95% probability that the door is open. Even with such high certainty, however, the robot should weigh the probability that it is correct with the cost of being incorrect. Acting on an a belief that does not accurately reflect the robot and its environment may result in the robot crashing or causing harm to humans.

5

Bayesian Filters

The Bayes filter describes how the belief of the state changes over time due to measurements and control actions. The Bayes filter takes various forms depending on whether the state is continuous or discrete, the actuation and perception noise is Gaussian or non-Gaussian, the actuation and perception models are linear or nonlinear, etc.

In the following diagram, specific cases of the Bayes filter are shown in boxes and decisions based on the scenario are shown in diamonds with the choices for each decision shown in blue.



The Bayes filter can only be implemented exactly in several scenarios. The difficulty is in representing the belief, which assigns a probability to every state. One case in which the Bayes filter can be implemented exactly are when the state is discrete and finite (the discrete and binary filter), since the belief is then represented by a finite set of numbers. The other case is when the state is continuous, the actuation and perception models are both linear, the noise is Gaussian, and the initial belief is Gaussian. In this scenario, the belief remains Gaussian and can therefore be parameterized by its mean and covariance (Kalman filter) or by its information vector and information matrix (information filter). All other cases are approximations of the Bayes filter, with different approximations resulting in different filters.

In this chapter, we describe each of these specific instances of the Bayes filter.

5.1 Discrete filter

Suppose the statespace is discrete (that is, the state s is a discrete random variable) with a finite number of values. Label the states as $s_1, s_2, \ldots, s_n$ and the corresponding belief probabilities as $p_1, p_2, \ldots, p_n$, that is, $b(s_i) = p_i$. Then the Bayes filter can be implemented directly from its general form. In particular, the discrete Bayes filter computes the updated belief probabilities $p'_1, p'_2, \ldots, p'_n$ after observing a measurement z as

$$p'_i = \eta\, p(z \mid s_i)\, p_i$$

where the coefficient η is chosen such that the belief after the measurement sums to one, $p'_1 + p'_2 + \ldots + p'_n = 1$. Similarly, the belief after applying a control action a is

$$p'_i = \sum_{j=1}^{n} p(s_i \mid s_j, a)\, p_j$$

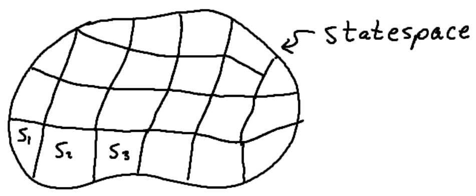
When the statespace is discrete and finite, the belief is represented by the n scalars $p_1, p_2, \ldots, p_n$. This is one of the few cases in which we can implement the Bayes filter exactly.

5.2 Histogram filter

When the statespace is continuous (that is, the state is a continuous random variable), the actuation update of the Bayes filter requires computing an integral which cannot typically be computed in closed form. Instead, one approach is to discretize the statespace and then use the discrete filter. When the discrete filter is applied to the discretization of a continuous statespace, it is called the histogram filter.

To apply the histogram filter, we approximate the domain of the continuous state s with finitely many non-intersecting regions that collectively cover the entire space:

$$\operatorname{dom}(s) = \boldsymbol{s}_1 \cup \boldsymbol{s}_2 \cup \cdots \cup \boldsymbol{s}_n \qquad \text{where} \qquad \boldsymbol{s}_i \cap \boldsymbol{s}_j \text{ is empty for all } i \neq j$$



Here, n is the number of regions used to represent the statespace. A straightforward decomposition of a continuous statespace is a multi-dimensional grid, where each region $\boldsymbol{s}_i$ is a cell in the grid. We can trade-off accuracy with computational efficiency through the granularity of the grid.

For each region index i, let p_i denote the belief that the continuous state s is in the region $\boldsymbol{s}_i$. The histogram filter approximates the belief of the continuous state with a piecewise uniform distribution that is constant over each region,

$$b(s) \approx \frac{p_i}{|\boldsymbol{s}_i|} \qquad \text{for } s \in \boldsymbol{s}_i$$

where $|\boldsymbol{s}_i|$ denotes the volume of the region. The belief of the histogram filter is then represented by the discrete probabilities $p_1, p_2, \ldots, p_n$ (the scaling by the volume of the region makes it so that these values sum to one).

Implementing the Bayes filter requires the actuation and perception models. When the state is continuous, these models are typically given in terms of the continuous state and are therefore probability *densities*. To implement the histogram filter, we instead need probability mass functions over the discretized state. One way is to define the discrete probability mass functions as the continuous probability densities evaluated at a representative point in the region,

$$p(z \mid \boldsymbol{s}_i) \approx p(z \mid \hat{s}_i) \qquad \text{and} \qquad p(\boldsymbol{s}_i \mid \boldsymbol{s}_j, a) \approx \eta \, |\boldsymbol{s}_i| \, p(\hat{s}_i \mid \hat{s}_j, a)$$

(The probabilistic justification of these formulas is described in Section 4.1.3 of *Probabilistic Robotics*. We omit the derivation here as it is quite lengthy and not particularly insightful.) For instance, we could take $\hat{s}_i$ to be the average state in the region,

$$\hat{s}_i = \frac{1}{|\boldsymbol{s}_i|} \int_{\boldsymbol{s}_i} s \, \mathrm{d}s$$

The histogram filter is then the discrete filter applied to the discretization of the statespace using these discrete actuation and perception models as approximations of the continuous ones. In particular, the perception and actuation updates of the histogram filter are

$$p_i' = \eta \, p(z \mid \hat{s}_i) \, p_i \qquad \text{and} \qquad p_i' = \eta \sum_{j=1}^{n} |\boldsymbol{s}_i| \, p(\hat{s}_i \mid \hat{s}_j, a) \, p_j$$

where the normalizer η is chosen such that the belief sums to one in each case (the normalizers are typically different for each update).

Decomposition techniques

There are various ways to decompose a continuous statespace into a discrete set of points. Decompositions may be *static* or *dynamic*, depending on whether the discretization changes over time. Static decompositions are the easiest to implement but are often wasteful (or less accurate) than dynamic ones since they do not adapt to the specific shape of the belief. One dynamic decomposition technique is the famly of density trees.

An alternative to using a dynamic decomposition is to use selective updating. Instead of updating the belief at every point in the discretization, we can save computational resources by only updating points that change significantly from the prior belief.

Decompositions may also be *topological* or *metric*. Topological decompositions discretize the statespace by significant places (or features) in the environment, such as doors, trees, etc. Metric decompositions, on the other hand, discretize the statespace based on distance, such as every 10cm. Metric decompositions often have higher resolution than topological decompositions, although this need not be the case.

5.3 Binary filter with static state

The binary filter can be used to estimate a binary state (one with only two possible values) that does not change over time. A robot may use a binary filter to estimate whether or not a door is open or closed using multiple measurements, where the state of the door is static. While this could be implemented using a discrete filter, we will see that the binary filter has a particularly simple form. Later, we will use the binary filter to estimate an occupancy grid map of the environment.

For a binary state, we denote the two values as s and $\neg s$. Since the belief is a probability distribution, we have that $b(\neg s) = 1 - b(s)$. The state is also static in this case, so there are no control actions and the belief only depends on the past measurements.

The binary filter represents the belief as a log odds ratio. The odds of a binary state is the ratio of the probability of this event divided by the probability of the other event,

$$\text{odds of } s = \frac{p(s)}{p(\neg s)} = \frac{p(s)}{1 - p(s)}$$

The log odds of the belief and its inverse are then

$$\ell(s) = \log \frac{b(s)}{1 - b(s)} \qquad \text{and} \qquad b(s) = 1 - \frac{1}{1 + \exp\{\ell(s)\}}$$

While discrete probabilities must be in the closed interval $[0, 1]$, the log odds can be any real number.

Given a measurement z, the binary filter updates the log odds of the belief as

$$\ell'(s) = \ell(s) + \log \frac{p(s \mid z)}{1 - p(s \mid z)} - \log \frac{p(s)}{1 - p(s)}$$

The binary filter uses the inverse sensor model $p(s \mid z)$ instead of the usual forward model $p(z \mid s)$. Since the state is binary, it is typically easier to find this inverse model. For instance, suppose that the state s describes whether or a door is open or closed, and the measurement z consists of a camera image of the door. It is much easier to find a function that describes the probability of the door being open or closed given the image than to assign a probability over all images.

Proof. Recall that the perception update for the Bayes filter is

$$p(s \mid z, i) = \frac{p(z \mid s)\, p(s \mid i)}{p(z \mid i)}$$

where i is all information that is available before observing the measurement z. Applying Bayes rule to the perception model gives

$$p(z \mid s) = \frac{p(s \mid z)\, p(z)}{p(s)}$$

Substituting this into the filter update and doing similarly for the opposite event gives

$$p(s \mid z, i) = \frac{p(s \mid z)\, p(z)\, p(s \mid i)}{p(s)\, p(z \mid i)} \qquad \text{and} \qquad p(\neg s \mid z, i) = \frac{p(\neg s \mid z)\, p(z)\, p(\neg s \mid i)}{p(\neg s)\, p(z \mid i)}$$

Using the Markov assumption, the odds of the belief is then

$$\frac{p(s \mid z, i)}{p(\neg s \mid z, i)} = \frac{p(s \mid z)}{1 - p(s \mid z)} \cdot \frac{p(s \mid i)}{1 - p(s \mid i)} \cdot \frac{1 - p(s)}{p(s)}$$

Taking the logarithm of the odds of the belief yields the binary filter. ■

5.4 Linear–Gaussian filters

In this section, we study the Bayes filter in the case where the models are *linear* and both the noise and the initial belief are *Gaussian*. Together, these assumptions enable us to efficiently compute the exact Bayes filter for a continuous statespace. There are two forms of the Bayes filter in the linear–Gaussian case: the Kalman filter uses the moments parameterization of a Gaussian random variable (mean and covariance) while the information filter represents the belief in information form (information vector and information matrix). When the models are nonlinear, we can approximate the Bayes filter by linearizing the models about the mean of the current belief and updating the belief using this linearization; this results in the extended Kalman filter and extended information filter.

Suppose the actuation and perception are described by the linear models

$$\begin{aligned} s' &= As + Ba + \varepsilon \qquad & \varepsilon &\sim \mathcal{N}(0, Q) \\ z &= Cs + \delta \qquad & \delta &\sim \mathcal{N}(0, R) \end{aligned}$$

where the actuation noise ε and perception noise δ are Gaussian random variables with zero mean and covariance Q and R, respectively. We also assume that the initial belief is a Gaussian random variable with mean μ and covariance Σ,

$$s_0 \sim \mathcal{N}(\mu, \Sigma)$$

Under these assumptions, the belief is a Gaussian random variable and is therefore parameterized by its mean and covariance.

Instead of describing the models in terms of the noise parameters, we can also express them in terms of their probability densities. The actuation and perception models are the Gaussian random variables

$$\begin{aligned} p(s' \mid s, a) &= \mathcal{N}(s'; As + Ba, Q) \\ p(z \mid s) &= \mathcal{N}(z; Cs, R) \end{aligned}$$

Notation. We use $\mathcal{N}(x; \mu, \Sigma)$ to denote the density of a Gaussian random variable with mean μ and covaraince Σ as a function of x,

$$\mathcal{N}(x; \mu, \Sigma) = \det(2\pi\Sigma)^{-1/2} \exp\big(-\tfrac{1}{2}(x-\mu)^\mathsf{T}\Sigma^{-1}(x-\mu)\big)$$

and we use $x \sim \mathcal{N}(\mu, \Sigma)$ to denote that x is a Gaussian random variable with mean μ and covariance Σ. ■

Kalman filter

The Kalman filter is the linear–Gaussian filter where the belief is represented in moment form by its mean and covariance. At each time, the belief is a Gaussian random variable with mean μ and covariance Σ,

$$b(s) = \mathcal{N}(s; \mu, \Sigma)$$

While this is a continuous probability distribution, it is parameterized by the finite-dimensional variables μ and Σ. To describe how the belief propagates over time, we will describe how the mean and covariance are updated based on control actions and measurements.

Before describing the Kalman filter updates, we list a few useful properties of Gaussian random variables.

Proposition. If $x \sim \mathcal{N}(\mu, \Sigma)$, then $Ax + b \sim \mathcal{N}(A\mu + b, A\Sigma A^\mathsf{T})$. ■

Proposition. If $\begin{bmatrix} x \\ y \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix}, \begin{bmatrix} \Sigma_x & \Sigma_{xy} \\ \Sigma_{yx} & \Sigma_y \end{bmatrix}\right)$, then $x \mid y \sim \mathcal{N}(\mu_x + \Sigma_{xy}\Sigma_y^{-1}(y - \mu_y), \Sigma_x - \Sigma_{xy}\Sigma_y^{-1}\Sigma_{yx})$. ■

Proposition. If $x \sim \mathcal{N}(\mu_x, \Sigma_x)$ and $y \sim \mathcal{N}(\mu_y, \Sigma_y)$ are independent, then $x + y \sim \mathcal{N}(\mu_x + \mu_y, \Sigma_x + \Sigma_y)$. ■

Actuation update

Suppose the belief is a Gaussian random variable with mean μ and covariance Σ. After applying a control action a in which the covariance of the actuation noise is Q, the belief is still a Gaussian random variable with mean and covariance

$$\mu' = A\mu + Ba \qquad \text{and} \qquad \Sigma' = A\Sigma A^\mathsf{T} + Q$$

The updated mean just applies the deterministic actuation model with zero noise to the previous mean. The updated covariance only depends on the matrix A and covariance Q and is typically larger than the previous covariance since actuation makes us less certain about the state. The posterior belief is the Gaussian random variable

$$b'(s') = \mathcal{N}(s'; A\mu + Ba, A\Sigma A^\mathsf{T} + Q)$$

Proof. Using the formula for an affine transformation applied to a Gaussian random variable, the distribution of the next state without noise is

$$As + Ba \sim \mathcal{N}(A\mu + Ba, A\Sigma A^\mathsf{T})$$

The actual state update also has additive noise. But the noise is assumed to be independent across time and independent of the initial state, so we can use the result about the distribution of the sum of two independent Gaussian random variables to obtain the posterior belief. ■

Perception update

Now suppose that the belief is a Gaussian random variable with mean μ and covariance Σ and we observe a measurement z in which the covariance of the perception noise is R. After the measurement, the belief is still a Gaussian random variable with mean and covariance

$$\mu' = \mu + K\,(z - C\mu) \qquad \text{and} \qquad \Sigma' = (I - KC)\Sigma$$

where the matrix K, called the *Kalman gain*, is given by

$$K = \Sigma C^\mathsf{T}(C\Sigma C^\mathsf{T} + R)^{-1}$$

In other words, the posterior belief is

$$b'(s) = \mathcal{N}(s; \mu + K\,(z - C\mu), (I - KC)\Sigma)$$

Since the measurement is correlated with the true state, the posterior belief has less uncertainty than the prior belief before applying the perception update.

Proof. Let b denote the random variable corresponding to the prior belief, which is the random variable for the state before the measurement conditioned on all previous information. Since the measurement noise is independent of the previous actuation noise, perception noise, and initial state (by assumption), the belief and perception noise have the joint distribution

$$\begin{bmatrix} b \\ \varepsilon \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu \\ 0 \end{bmatrix}, \begin{bmatrix} \Sigma & 0 \\ 0 & R \end{bmatrix}\right)$$

The belief and measurement are linear functions of the belief and perception noise,

$$\begin{bmatrix} b \\ z \end{bmatrix} = \begin{bmatrix} I & 0 \\ C & I \end{bmatrix} \begin{bmatrix} b \\ \varepsilon \end{bmatrix}$$

Using the result stating that an affine transformation applied to a Gaussian random variable is also Gaussian, we have that the joint distribution of the belief and the measurement is

$$\begin{bmatrix} b \\ z \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu \\ C\mu \end{bmatrix}, \begin{bmatrix} \Sigma & \Sigma C^\mathsf{T} \\ C\Sigma & C\Sigma C^\mathsf{T} + R \end{bmatrix}\right)$$

The belief after the measurement is the previous belief conditioned on the measurement, $b' = b \mid z$. Using the result about the conditional expectation of two jointly Gaussian random variables, the belief after the measurement update is

$$b' \sim \mathcal{N}\big(\mu + \Sigma C^\mathsf{T}(C\Sigma C^\mathsf{T} + R)^{-1}(z - C\mu), \Sigma - \Sigma C^\mathsf{T}(C\Sigma C^\mathsf{T} + R)^{-1}C\Sigma\big)$$

Rewriting this in terms of the Kalman gain gives the update to the mean and covariance. ■

Information filter

Instead of representing the belief by its moments (mean and covariance), we can represent it in information form using its information matrix and information vector. This is called the canonical parameterization. The information filter is equivalent to the Kalman filter; it just uses a different parameterization of the Gaussian belief. However, the Kalman and information filters have different computational complexities that make each filter faster in certain scenarios.

Canonical parameterization

Before describing the information filter, we first describe the canonical parameterization of a Gaussian random variable. Recall that a Gaussian random variable has the probability density function

$$p(x) = \det(2\pi\Sigma)^{-\frac{1}{2}} \exp\big(-\tfrac{1}{2}(x-\mu)^\mathsf{T}\Sigma^{-1}(x-\mu)\big)$$

Expanding this out, we have

$$p(x) = \underbrace{\det(2\pi\Sigma)^{-\frac{1}{2}} \exp\big(-\tfrac{1}{2}\mu^\mathsf{T}\Sigma^{-1}\mu\big)}_{\text{constant}} \exp\big(-\tfrac{1}{2}x^\mathsf{T}\Sigma^{-1}x + x^\mathsf{T}\Sigma^{-1}\mu\big)$$

This motivates the canonical parameterization of a Gaussian random variable, which is

$$p(x) = \eta \exp\big(-\tfrac{1}{2}x^\mathsf{T}\Omega x + x^\mathsf{T}\xi\big)$$

where $\Omega = \Sigma^{-1}$ is the *information matrix* and $\xi = \Sigma^{-1}\mu$ is the *information vector*. This is an equivalent parameterization of a Gaussian, and we can recover the mean and covariance from the information matrix and information vector as $\mu = \Omega^{-1}\xi$ and $\Sigma = \Omega^{-1}$.

We now describe the actuation and perception updates of the information filter, which describe how the information matrix and information vector are updated based on actions and observations.

Actuation update

Suppose the belief is a Gaussian random variable with information matrix Ω and information vector ξ. After applying a control action u in which the covariance of the actuation noise is R, the belief is still a Gaussian random variable with information matrix and vector

$$\Omega' = (A\Omega^{-1}A^\mathsf{T} + Q)^{-1} \qquad \text{and} \qquad \xi' = \Omega'\,(A\Omega^{-1}\xi + Bu)$$

The actuation update requires inverting two $n \times n$ matrices, where n is the dimension of the statespace. While this operation is quite expensive, it can be performed more efficiently in some cases.

Proof. Since we have already found the updates for the Kalman filter, we can construct the updates for the information filter by simplying applying the transformation between the two parameterizations. Recall that the actuation update for the Kalman filter is

$$\mu' = A\mu + Bu \qquad \text{and} \qquad \Sigma' = A\Sigma A^\mathsf{T} + Q$$

The information filter update follows directly from the transformation $\Omega = \Sigma^{-1}$ and $\xi = \Sigma^{-1}\mu$, or equivalently, $\mu = \Omega^{-1}\xi$ and $\Sigma = \Omega^{-1}$. ■

Perception update

Now suppose we observe a measurement z in which the covariance of the perception noise is R. After the measurement, the belief is still a Gaussian random variable with information matrix and vector

$$\Omega' = \Omega + C^\mathsf{T} R^{-1} C \qquad \text{and} \qquad \xi' = \xi + C^\mathsf{T} R^{-1} z$$

In information form, the update to the belief due to a measurement is additive. A consequence of this is that measurements affecting only some states require updates to only the corresponding components of the information form.

Proof. From the Bayes filter, the perception update for the information filter is $b'(s) = \eta\, p(z \mid x)\, b(s)$. The perception model and prior belief are both Gaussian random variables, and multiplying two exponentials sums their exponents, so the density of their product is the exponential

$$b'(s) = \eta \exp\bigl(-\tfrac{1}{2}(z - Cs)^\mathsf{T} R^{-1}(z - Cs) - \tfrac{1}{2}(s-\mu)^\mathsf{T}\Sigma^{-1}(s-\mu)\bigr)$$

Converting the parameterization of the prior belief to canonical form gives

$$b'(s) = \eta \exp\bigl(-\tfrac{1}{2}s^\mathsf{T} C^\mathsf{T} R^{-1} C s + s^\mathsf{T} C^\mathsf{T} R^{-1} z - \tfrac{1}{2}s^\mathsf{T}\Omega s + s^\mathsf{T}\xi\bigr)$$

Reordering terms in the exponent, we have that

$$b'(s) = \eta \exp\bigl(-\tfrac{1}{2}s^\mathsf{T}(\Omega + C^\mathsf{T} R^{-1} C)s + s^\mathsf{T}(\xi + C^\mathsf{T} R^{-1} z)\bigr)$$

This is a Gaussian random variable in information form, so we can read off the information matrix and information vector, which gives the perception update of the information filter. ∎

Comparison

	Kalman filter	**Information filter**
Action	$\mu' = A\mu + Ba$ $\Sigma' = A\Sigma A^\mathsf{T} + Q$	$\Omega' = (A\Omega^{-1}A^\mathsf{T} + Q)^{-1}$ $\xi' = \Omega'\,(A\Omega^{-1}\xi + Bu)$
Measurement	$\mu' = \mu + K\,(z - C\mu)$ $\Sigma' = (I - KC)\Sigma$	$\Omega' = \Omega + C^\mathsf{T} R^{-1} C$ $\xi' = \xi + C^\mathsf{T} R^{-1} z$

Kalman gain: $K = \Sigma C^\mathsf{T}(C\Sigma C^\mathsf{T} + R)^{-1}$

- **No information.** In information form, we can represent having no information as $\Omega = 0$ and $\xi = 0$. We can use this to initialize the filter when we have no prior knowledge of the state. In the moments parameterization, however, we cannot represent having no information since this corresponds to infinite covariance.
- **Computational complexity.** The computations required to implement the actuation and perception updates of the Kalman and information filters can be vastly different. Let n, m, and p denote the dimensions of the state, action, and measurements spaces, respectively. For the Kalman filter, the actuation update only requires matrix multiplication, while the measurement update requires inverting a matrix of size $p \times p$ to compute the Kalman gain. For the information filter, the measurement update only requires matrix multiplication (R^{-1} may be precomputed), while the actuation update requires inverting two matrices of size $n \times n$. Therefore, the most computationally efficient filter depends on the relative sizes of the state, action, and measurements spaces, as well as the relative frequency of actions and measurements.

Extended versions

The linear–Gaussian filters are exact instances of the Bayes filter when the models are linear, and the noise and initial belief are Gaussian random variables. These assumptions imply that the belief is always Gaussian, since linear functions of Gaussian are also Gaussian. Robots, however, are nonlinear systems. One way to approximate the Bayes filter for nonlinear systems is to linearize the models about the mean of the current belief and apply a linear–Gaussian filter to this linearization.

Taylor series expansion

Recall that the Taylor series expansion of a scalar function $f : \mathbb{R} \to \mathbb{R}$ about a point $\tilde{x} \in \mathbb{R}$ is

$$f(x) \quad = \quad \sum_{n=0}^{\infty} \frac{f^{(n)}(\tilde{x})}{n!}(x-\tilde{x})^n \quad = \quad f(\tilde{x}) + \frac{f'(\tilde{x})}{1!}(x-\tilde{x}) + \frac{f''(\tilde{x})}{2!}(x-\tilde{x})^2 + \frac{f'''(\tilde{x})}{3!}(x-\tilde{x})^3 + \dots$$

For multi-dimensional functions $f : \mathbb{R}^n \to \mathbb{R}$, the first-order Taylor series expansion about a point $\tilde{x} \in \mathbb{R}^n$ is

$$f(x) \approx f(\tilde{x}) + \frac{\partial f(\tilde{x})}{\partial x}(x-\tilde{x})$$

where the *Jacobian* of the function f is the matrix-valued function

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

Example.

$$f(x_1, x_2) = \begin{bmatrix} -4x_2 \\ 4x_1 - x_1^2 - 0.5x_2 \end{bmatrix} \qquad \frac{\partial f}{\partial x}(x_1, x_2) = \begin{bmatrix} 0 & -4 \\ 4 - 2x_1 & -0.5 \end{bmatrix}$$

Linearization of the models

Suppose the actuation and perception are described by the nonlinear models

$$s' = g(s, a) + \varepsilon$$
$$z = h(s) + \delta$$

where the actuation and perception noise are Gaussian random variables as before. The issue with applying the Bayes filter directly to this system is that, even if the initial belief is Gaussian, it will not be Gaussian after the nonlinear actuation or perception updates. We can then no longer represent the belief by its mean and covariance (or in information form), but must instead represent this (arbitrarily complex) probability distribution.

The extended linear–Gaussian filters approximate the nonlinear models with their linear approximations

$$g(s,a) \approx g(\mu,a) + G\,(s-\mu)$$
$$h(s) \approx h(\mu) + H\,(s-\mu)$$

where the matrices G and H are the Jacobians of the models with respect to the state evaluated at the current mean of the state and the control action,

$$G = \frac{\partial g}{\partial x}(\mu,a) \qquad \text{and} \qquad H = \frac{\partial h}{\partial x}(\mu)$$

The extended filters are identical to the standard versions except that they use these Jacobians in place of the linear model matrices. Since the Jacobians are functions of the state and the control action, they must be evaluated at the current iterates for each update.

For the extended Kalman filter, the actuation update is

$$\mu' = g(\mu,a) \qquad \text{and} \qquad \Sigma' = G\Sigma G^{\mathsf{T}} + Q \qquad \text{where} \qquad G = \frac{\partial g}{\partial x}(\mu,a)$$

and the perception update is

$$\mu' = \mu + K\,(z - h(\mu)) \quad \text{and} \quad \Sigma' = (I - KH)\Sigma \quad \text{where} \quad K = \Sigma H^{\mathsf{T}}(H\Sigma H^{\mathsf{T}} + R)^{-1} \quad \text{and} \quad H = \frac{\partial h}{\partial x}(\mu)$$

When using the extended linear–Gaussian filters, it is important to remember that the belief is being represented by a Gaussian random variable even though the true belief is not Gaussian. Gaussians represent random variables that have a single mean and some amount of deviation that decays with the distance from the mean. Gaussians are not good approximations of random variables with multiple modes, such as the belief that a robot is in front of one of two doors with high certainty but low probability of being in between. It is therefore important to observe enough measurements relative to the number of control actions to keep the uncertainty in the belief small.

Model Jacobians

Odometry-based actuation model

Consider the odometry-based actuation model

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} a_t\cos(\theta + a_{r_1}) \\ a_t\sin(\theta + a_{r_1}) \\ a_{r_1} + a_{r_2} \end{bmatrix} + \begin{bmatrix} \delta_x \\ \delta_y \\ \delta_\theta \end{bmatrix}$$

where (x, y, θ) is the robot pose before applying the control action, (x', y', θ') is the pose after the control action, and $(\delta_x, \delta_y, \delta_\theta)$ is the actuation noise. The control action is (a_{r_1}, a_t, a_{r_2}), which describes a transformation between the two robot poses in which the robot first rotates by a_{r_1}, then translates in the direction of its heading by a_t, and then rotates again by a_{r_2}.

The Jacobian of the odometry-based actuation model is

$$G = \begin{bmatrix} \frac{\partial g_1}{\partial x} & \frac{\partial g_1}{\partial y} & \frac{\partial g_1}{\partial \theta} \\ \frac{\partial g_2}{\partial x} & \frac{\partial g_2}{\partial y} & \frac{\partial g_2}{\partial \theta} \\ \frac{\partial g_3}{\partial x} & \frac{\partial g_3}{\partial y} & \frac{\partial g_3}{\partial \theta} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -a_t\sin(\theta + a_{r_1}) \\ 0 & 1 & a_t\cos(\theta + a_{r_1}) \\ 0 & 0 & 1 \end{bmatrix}$$

In general, the Jacobian is a function of the pose (x, y, θ), but in this case it depends only on the orientation θ.

Range-bearing perception model

Consider a robot at state (x, y, θ) that uses a sensor to measure the range z_r and bearing z_ϕ to a landmark at known position (ℓ_x, ℓ_y). A simple probabilistic model for the range and bearing measurements is given by

$$\begin{bmatrix} z_r \\ z_\phi \end{bmatrix} = \begin{bmatrix} \sqrt{(\ell_x - x)^2 + (\ell_y - y)^2} \\ \operatorname{atan2}(\ell_y - y, \ell_x - x) - \theta \end{bmatrix} + \begin{bmatrix} \varepsilon_r \\ \varepsilon_\phi \end{bmatrix}$$

where the parameters $(\varepsilon_r, \varepsilon_\phi)$ represent perception noise. This models the sensor measurements as the true values perturbed by additive noise.

The Jacobian of the range-bearing perception model is

$$H = \begin{bmatrix} \dfrac{\partial h_1}{\partial x} & \dfrac{\partial h_1}{\partial y} & \dfrac{\partial h_1}{\partial \theta} \\ \dfrac{\partial h_2}{\partial x} & \dfrac{\partial h_2}{\partial y} & \dfrac{\partial h_2}{\partial \theta} \end{bmatrix} = \begin{bmatrix} -\dfrac{\ell_x - x}{r} & -\dfrac{\ell_y - y}{r} & 0 \\ \dfrac{\ell_y - y}{r^2} & -\dfrac{\ell_x - x}{r^2} & -1 \end{bmatrix}$$

where $r = \sqrt{(\ell_x - x)^2 + (\ell_y - y)^2}$ is the distance from the robot position (x, y) to the landmark position (ℓ_x, ℓ_y). In general, the Jacobian is a function of the pose (x, y, θ), but in this case it depends only on the position (x, y). Here, we used the partial derivatives

$$\frac{\partial}{\partial x}\operatorname{atan2}(y, x) = \frac{-y}{x^2 + y^2} \qquad \text{and} \qquad \frac{\partial}{\partial y}\operatorname{atan2}(y, x) = \frac{x}{x^2 + y^2}$$

Extensions

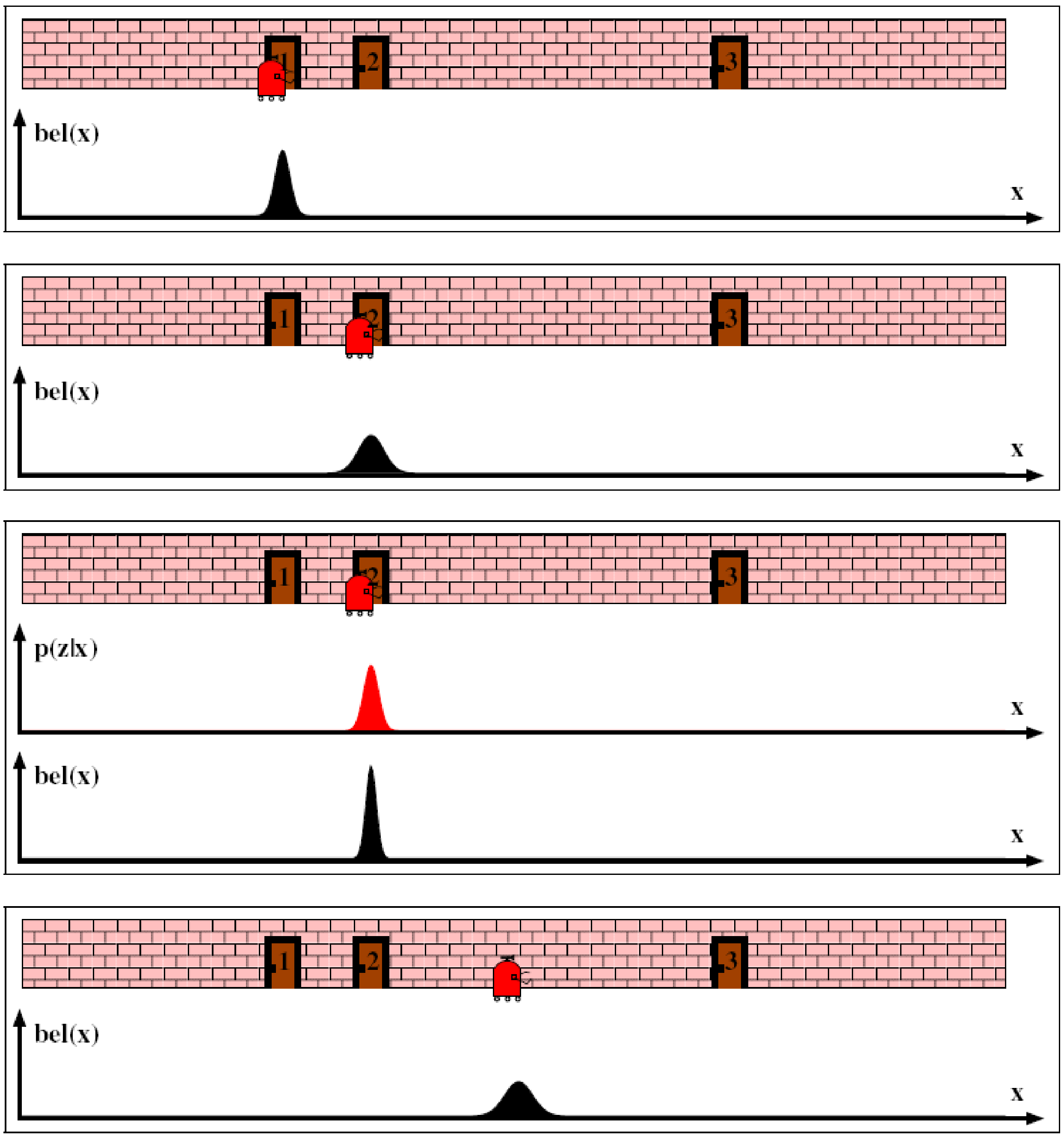
- **Unknown correspondences.** If we do not know the correspondence of a measurement, we can choose the correspondence that is most consistent with the measurement according to the model,

$$c \quad = \quad \arg\max_c \; p(z \mid s, m, c)$$

- **Multi-hypothesis tracking.** One of the main limitations of the extended Kalman filter is that it models the belief as a Gaussian random variable, which has only a single mean. This makes the EKF unsuited for situations in which there are multiple distinct poses that the robot could be in, such as when there is ambiguity due to symmetry in the environment. One extension to the EKF that addresses this problem is to approximate the belief as a *mixture of Gaussians*,

$$b(s) = \frac{1}{\sum_i a_i} \sum_i a_i \, \det(2\pi\Sigma_i)^{-1/2} \exp\!\left(-\tfrac{1}{2}(s - \mu_i)^\mathsf{T} \Sigma_i^{-1} (s - \mu_i)\right)$$

  Here, the belief is a convex combination of a set of Gaussian random variables with means μ_i and covariances Σ_i, and a_i is the weight of the i^{th} Gaussian. The belief still has a finite parameterization (although it requires more memory than a single Gaussian) and is able to represent multiple distinct hypotheses of the state.



5.5 Unscented Kalman filter

Like the extended Kalman filter, the unscented Kalman filter (UKF) linearizes the models to represent the belief as a Gaussian random variable. Instead of using a Taylor series approximation, however, the UKF uses the unscented transform, which is a different way of linearizing the models to obtain a Gaussian posterior. The unscented transform is in general a better linearization than the series expansion, so the UKF performs (slightly) better than the EKF. Also, computation of the unscented transform does not require the Jacobians of the models and is therefore a derivative-free filter.

Unscented transform

The unscented transform estimates the distribution of a nonlinear function of a Gaussian random variable. Instead of linearizing the function about the mean as in the EKF, the UKF propagates a set of points through the nonlinear function and then sets the mean and covariance as a weighted sum of the empirical mean and covariance.

Let x be a Gaussian random variable with mean μ and covariance Σ, and suppose we want to approximate $f(x)$ by a Gaussian random variable where f is some nonlinear function. The UKF selects a set of sigma points $\boldsymbol{X}_i$ and constructs the mean and covariance from the images $f(\boldsymbol{X}_i)$. In particular, the mean and covariance of $f(x)$ are approximated as

$$\mu' = \sum_i w_i^m \, f(\boldsymbol{X}_i) \qquad \text{and} \qquad \Sigma' = \sum_i w_i^c \left(f(\boldsymbol{X}_i) - \mu'\right)\left(f(\boldsymbol{X}_i) - \mu'\right)^{\mathsf{T}}$$

where the weights w_i^m and w_i^c each sum to one. There are various ways to select the sigma points and the weights. We now discuss one common method.

Parameters

The unscented transform uses the following parameters: $\kappa \geq 0$ and $\alpha \in (0, 1]$ influence how far the sigma points are from the mean, $\lambda = \alpha^2(n + \kappa) - n$, and $\beta = 2$ is the optimal choice for Gaussians.

Sigma points

For an n-dimensional Gaussian random variable, one way to choose the sigma points is

$$\boldsymbol{X}_i = \begin{cases} \mu & \text{if } i = 0 \\ \mu + \left(\sqrt{(n+\lambda)\,\Sigma}\right)_i & \text{if } i = 1, \ldots, n \\ \mu - \left(\sqrt{(n+\lambda)\,\Sigma}\right)_{i-n} & \text{if } i = n+1, \ldots, 2n \end{cases}$$

Here, the subscript i denotes the i^{th} column of the matrix and the square root is the matrix square root. Since the covariance matrix has dimensions $n \times n$, there are $2n + 1$ sigma points.

Weights

Each sigma point has two weights associated with it, one for computing the mean and the other for the covariance. One way to choose the weights is

$$w_0^m = \frac{\lambda}{n+\lambda} \qquad \text{and} \qquad w_0^c = \frac{\lambda}{n+\lambda} + (1 - \alpha^2 + \beta) \qquad \text{and} \qquad w_i^m = w_i^c = \frac{1}{2\,(n+\lambda)} \qquad \text{for } i = 1, \ldots, 2n$$

Actuation and perception updates

The unscented Kalman filter is similar to the extended Kalman filter except that it uses the unscented transform to approximate the mean and covariance after applying the nonlinear models.

Actuation update

Suppose that the belief is a Gaussian random variable with mean μ and covariance Σ. After applying a control action a with actuation noise covariance Q, the updated belief of the UKF is also Gaussian with mean and covariance

$$\mu' = \sum_i w_i^m \, g(\boldsymbol{X}_i, a) \qquad \text{and} \qquad \Sigma' = \sum_i w_i^c \left(g(\boldsymbol{X}_i, a) - \mu'\right)\left(g(\boldsymbol{X}_i, a) - \mu'\right)^\mathsf{T} + Q$$

where $\boldsymbol{X}_i$ are the sigma points and w_i^m and w_i^c are the weights of the unscented transform.

Perception update

Similarly, suppose we observe a measurement z with perception noise covariance R. Propagate the sigma points through the perception model to obtain $\mathcal{Z}_i = h(\boldsymbol{X}_i)$, and denote the empirical mean of the measurement as

$$\hat{z} = \sum_i w_i^m \, \mathcal{Z}_i$$

Define the empirical covariance between the state x and measurement z as

$$\Sigma_{x,z} = \sum_i w_i^c \, (\boldsymbol{X}_i - \mu)(\mathcal{Z}_i - \hat{z})^\mathsf{T}$$

and define the empirical covariance of the measurement as

$$S = \sum_i w_i^c \, (\mathcal{Z}_i - \hat{z})(\mathcal{Z}_i - \hat{z})^\mathsf{T} + R$$

The empirical Kalman gain is then $K = \Sigma_{x,z} S^{-1}$, and the updated belief of the UKF is a Gaussian random variable with mean and covariance

$$\mu' = \mu + K\,(z - \hat{z}) \qquad \text{and} \qquad \Sigma' = \Sigma - KSK^\mathsf{T}$$

5.6 Particle filter

In contrast to the linear–Gaussian filters and the UKF that parameterize the belief as a Gaussian random variable, the particle filter is a nonparametric filter that approximates the belief by a set of particles, where each particle is a concrete instantiation of the state. Since the particle filter does not assume that the belief is Gaussian, it is capable of representing complex belief distributions.

The histogram filter also approximates the belief by a finite set of numbers, but the particle filter differs in the way the in which these parameters are generated and how they populate the statespace. The particle filter approximates the belief by a set of random samples drawn from the posterior.

In the particle filter, the belief is approximated by a set of samples of the posterior, called particles. We denote the particles by

$$s = \{s_1, s_2, \ldots, s_n\}$$

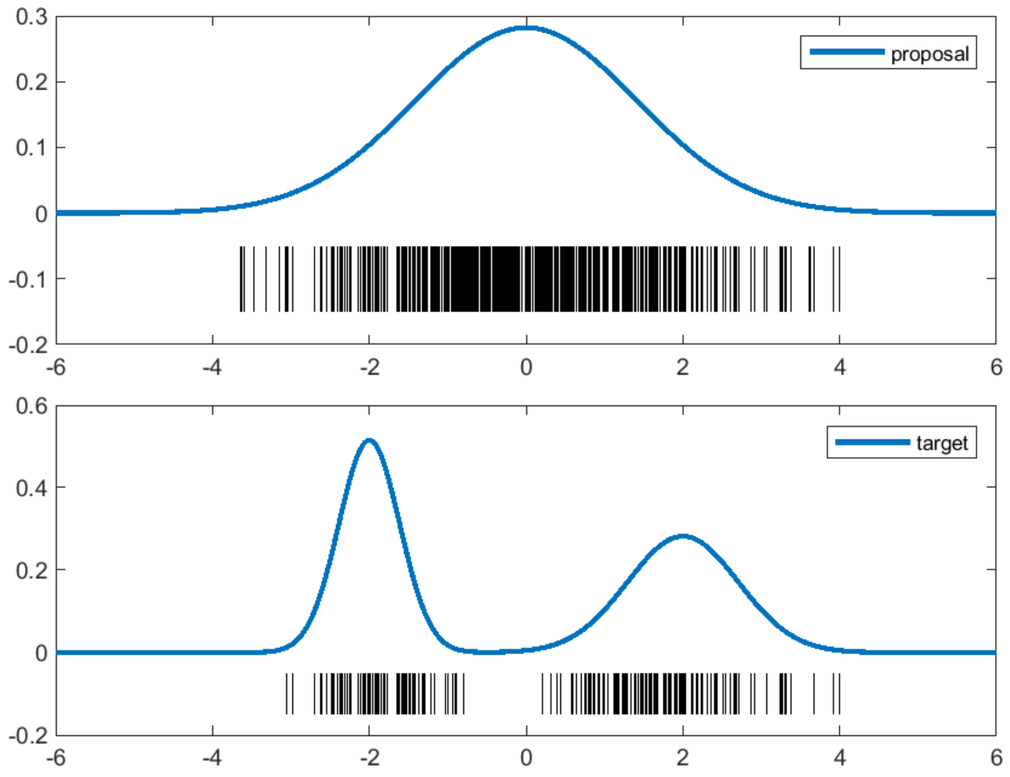
where n is the number of particles. Each particle is a sample of the belief, so it represents a hypothesis of the state. The particle filter approximates the belief by a set of impulses located at each of the particles,

$$b(s) \approx \sum_{i=1}^{n} \delta(s - s_i)$$

where δ is the Dirac delta function. The infinite-dimensional belief is approximated by a finite number of impulses. Using more particles results in a better approximation of the belief at the cost of higher computational complexity.

Approximating continuous probability distributions by their samples

The particle filter approximates a continuous probability distribution (the belief) by a set of particles that are sampled from the probability distribution. The following figure illustrates this idea. The figure shows two probability distributions along with a set of samples from the distribution. A key benefit of using samples is that this parameterization is able to approximate any probability distribution in the limit as the number of samples tends to infinity.



For the particle filter, the samples are used to approximate the belief which evolves over time. To implement the Bayes filter, we need to know how to update the particles when the robot applies an action and makes a measurement:

$$b'(s) = \eta\, p(z \mid s)\, b(s) \qquad \text{and} \qquad b'(s') = \int p(s' \mid s, a)\, b(s)\, \mathrm{d}s$$

Given a set of particles sampled from the prior belief $b(s)$, how do we obtain samples of the posterior belief $b'(s)$ for both actuation and perception updates? We do not know the prior belief distribution $b(s)$ (we only know a set of its samples), so we cannot sample from the posterior distribution directly. The actuation update is quite simple, while the perception update is significantly more complicated.

Actuation update

Let $s = \{s_1, s_2, \ldots, s_n\}$ denote the set of particles that represent the prior belief at some point in time. After applying an action a, the posterior belief is described by a new set of particles $s' = \{s'_1, s'_2, \ldots, s'_n\}$, where each updated particle is sampled from the actuation model conditioned on the sample of the prior belief and the action,

$$s'_i \sim p(s' \mid s_i, a)$$

Applying the update requires sampling from the actuation model.

Importance sampling

To implement the perception update of the Bayes filter, we need to sample from the posterior belief given samples of the prior belief. One way to sample from one probability distribution given samples from another distribution is *importance sampling*. Importance sampling generates samples from a probability distribution given samples of another distribution.

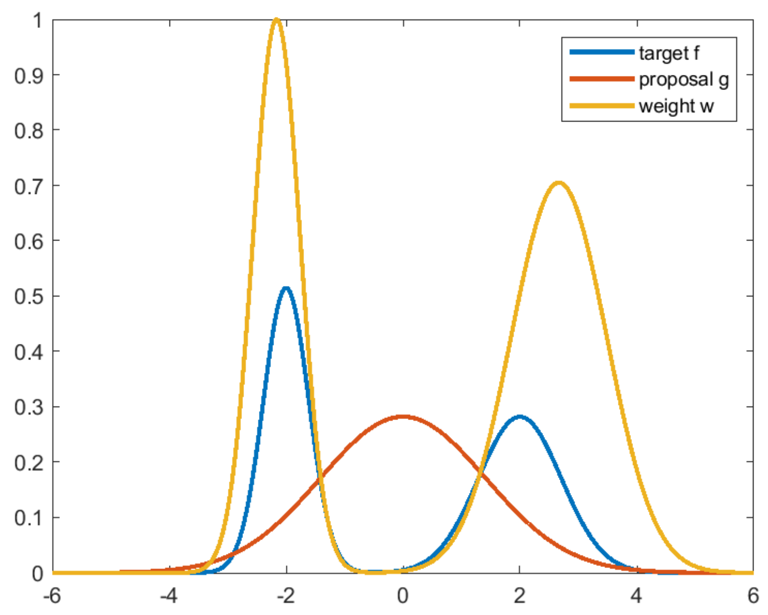
- The *target distribution* is the distribution that we want to sample and has density f.
- The *proposal distribution* is the distribution that we are able to sample and has density g.

In importance sampling, we first sample from the proposal distribution and then resample in a particular way so that the samples are drawn from the target distribution. Let the set $\{x_1, \ldots, x_n\}$ be samples drawn from the proposal distribution $g(x)$. We associate with each sample an *importance weight*

$$w_i = \frac{f(x_i)}{g(x_i)}$$

To be well-defined, this requires that $g(x) > 0$ whenever $f(x) > 0$. The weight describes the importance of each sample. Important samples occur when the proposal density is small and the target density is large, meaning that they rarely occur in the set of samples of the proposal distribution but they should often occur in the samples of the target distribution. To generate a set of samples $\{y_1, \ldots, y_n\}$ from the target distribution, each sample y_i from the target distribution is obtained by drawing the samples $\{x_1, \ldots, x_n\}$ of the proposal distribution with replacement with probability proportional to the importance weight $\{w_1, \ldots, w_n\}$.

The above figure illustrates importance sampling. Here, the samples from the proposal distribution were used to generate the samples from the target distribution. The figure below shows the target and proposal distributions along with the importance weight.



Perception update

We now use importance sampling to update the particles after a measurement. Here, the proposal distribution is the prior belief and the target distribution is the posterior belief.

Let s denote the current belief, and suppose we observe a measurement z. Let $g(s) = b(s)$ denote the proposal distribution and $f(s) = b'(s)$ denote the target distribution. According to the Bayes filter, the posterior belief is $b'(s) = \eta\, p(z \mid s)\, b(s)$. Therefore, the importance weights are proportional to the perception model,

$$w(s) = \frac{f(s)}{g(s)} \propto p(z \mid s)$$

The importance weight w_i for each particle i is then proportional to the perception model $p(z \mid s_i)$, and the weights are normalized to sum to one,

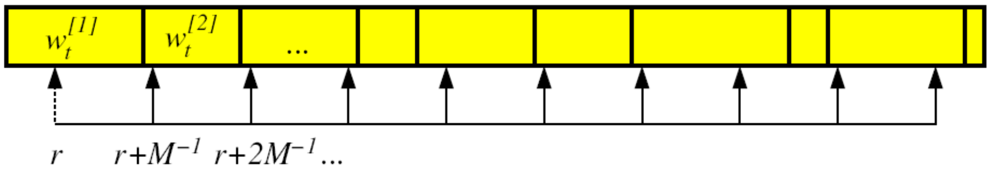
$$w_i \propto p(z \mid s_i) \qquad \text{and} \qquad w_1 + w_2 + \ldots + w_n = 1$$

There are several ways to resample the particles based on the importance weights. Each sampling method results in some variance of the samples from that of the desired distribution.

- **Independent sampling.** Draw with replacement particle s'_i with probability w_i from the prior samples $s = \{s_1, \ldots, s_n\}$ to obtain the posterior samples $s' = \{s'_1, \ldots, s'_n\}$. Since the particles are drawn with replacement, some of them will be duplicated in the posterior while others will never get sampled. This pruning process promotes survival of particles that most likely represent the true state.
- **Low-variance sampling.** Choose a random number $r \in [0, 1/n]$, where n is the number of particles. Then for each index $i = 1, \ldots, n$, select the particle i' such that

$$i' = \arg\min_{i'} \left\{ \sum_{k=1}^{i'} w_k \geq r + \frac{i-1}{n} \right\}$$

  The diagram below illustrates the selection process. The particles are lined up with size equal to their weight. Then each number $r + (i-1)/n$ for $i = 1, \ldots, n$ points to a single particle that is selected.



This is a principled approach to achieve low variance sampling that has multiple benefits.

- While the previous approach samples particles independently of each other, low-variance sampling selects evenly-spaced particles which covers the space of samples in a more systematic fashion.
- Any particle with imporance weight w_i at least $1/n$ is *always* sampled using this approach, so particles with large weights are guaranteed to survive. Moreover, if all particles have the same weight $1/n$, then the updated particles are identical to the original set.
- Low-variance sampling has a lower computational complexity than independent sampling, since it only needs to sample a single random number and then loop through the particles once.

Low-variance sampling can be implemented using the following pseudocode.

```
Algorithm  Low-variance sampling
 1: $s' = \{\}$                         ▷ Initialize the posterior as the empty set
 2: $r = \text{rand}(0, 1/n)$           ▷ Randomly sample the offset
 3: $i' = 1$
 4: $c = w_1$                           ▷ $c = \sum_{k=1}^{i'} w_k$
 5: for $i = 1, \dots, n$ do
 6:     $U = r + (i-1)/n$
 7:     while $U > c$ do                ▷ Increment $i'$ until $c \leq U$
 8:         $i' = i' + 1$
 9:         $c = c + w_{i'}$
10:     add particle $s_{i'}$ to the posterior $s'$
11: return $s'$
```

Comments

- **Particle deprivation.** Even when using a large number of particles, there may be no particles near the true state. While this may be due to not using enough particles, it may also be the case that the resampling procedure wiped out all particles near the true state. One solution to this is to add random samples back into the set of particles to improve particle diversity.
- **Sampling bias.** If $n = 1$, the perception update has no effect! Also, non-normalized weights are drawn from n-dimensional space, but normalization reduces the dimension by one.
- **Number of particles.** There is no general method to determine the number of particles, n. The amount needed highly depends on the particular application and the structure of the true belief. For instance, we would need many particles to accurately represent a uniform belief, while we could use a smaller number of particles to represent a belief that is highly localized to a small region of the statespace.

Part III

Localization and Mapping

6

Localization

Bayes filters can be used to localize a robot in a known environment. In this case, the state of the filter is the pose of the robot. The only modification to the Bayes filter is that the actuation and perception models are also conditioned on the map (which we assume is known). This is a low-dimensional estimation problem since the pose of the robot is typically only a few variables (three in two-dimensional space and six in three-dimensional space).

6.1 Properties

- local vs global (position tracking, kidnapped robot problem)
- static vs dynamic environments
- passive vs active
- single-robot vs multi-robot

6.2 Markov localization

Bayes filter applied to localization.

6.3 EKF localization

Known correspondences

Unknown correspondences

6.4 Multi-hypothesis tracking

6.5 UKF localization

6.6 Grid localization

Given a map of the environment, grid localization uses the histogram filter to estimate the pose of a mobile robot within the map based on the applied control actions and the observed measurements.

The grid localization algorithm is identical to the histogram filter except that the perception and actuation models are also conditioned on the map. Explicitly, the perception and actuation updates are

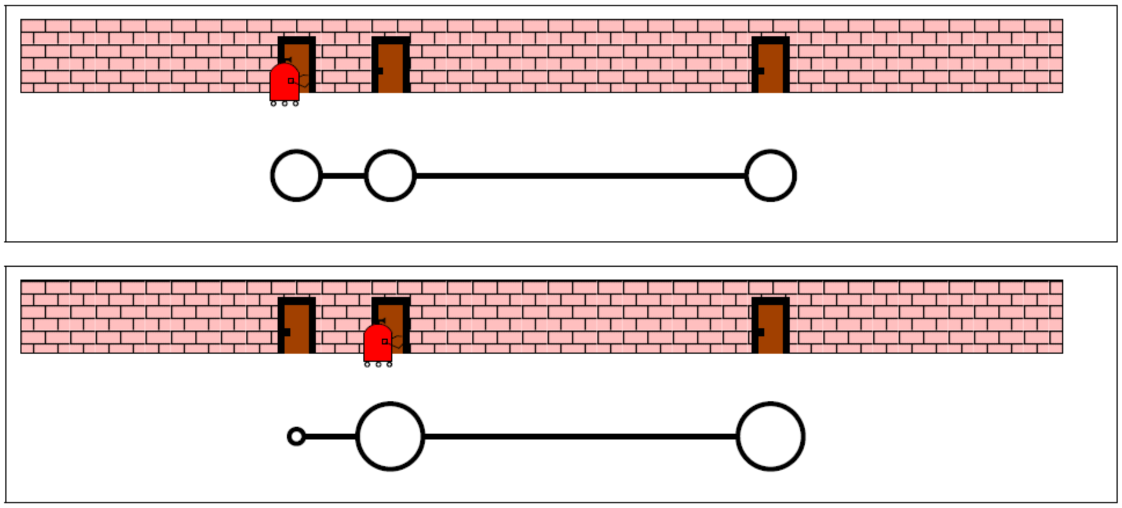
$$p_i' = \eta\, p(z \mid \hat{s}_i, m)\, p_i \qquad \text{and} \qquad p_i' = \eta \sum_{j=1}^{n} |\boldsymbol{s}_i|\, p(\hat{s}_i \mid \hat{s}_j, a, m)\, p_j$$

where the normalizer η is chosen such that the belief sums to one in each case (the normalizers are typically different for each update).

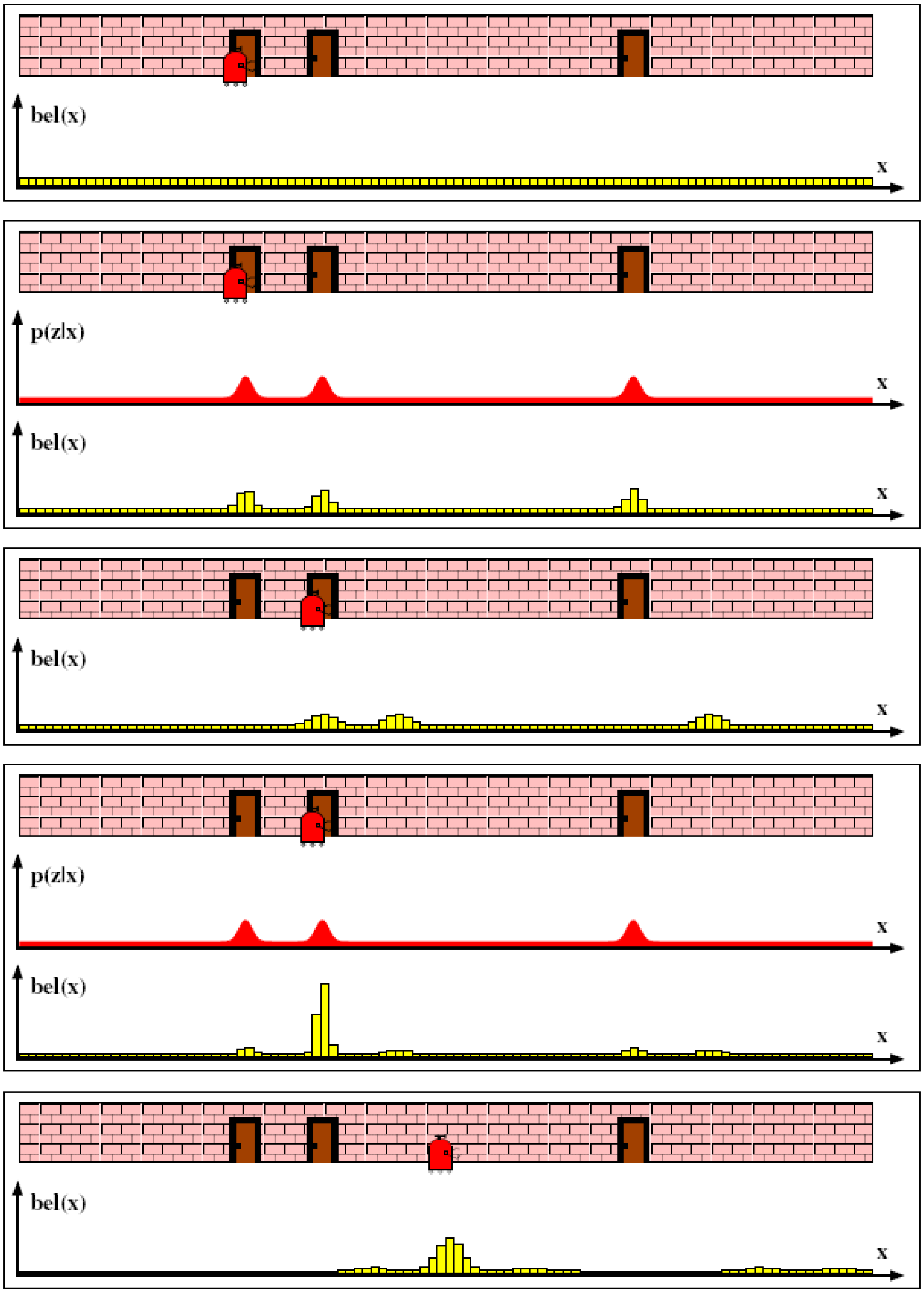
Grid types

The grid of the statespace may be topological or metric. Metric grids discretize the statespace based on distance, while topological grids discretize the statespace based on significant features in the environment.

The following figure shows the familiar scenario in which a robot localizes its position in a one-dimensional hallway. Here, the topological grid has only three points corresponding to each of the three doors. The belief at each point is represented by the size of the circle.

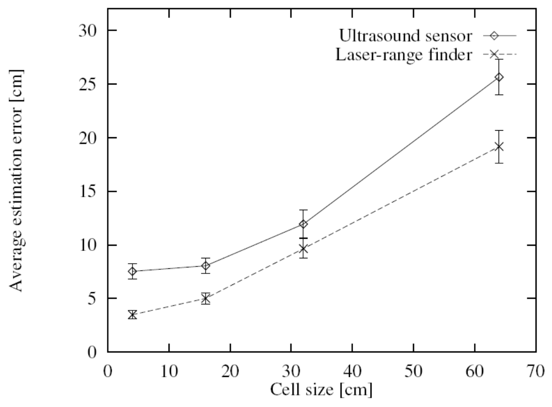


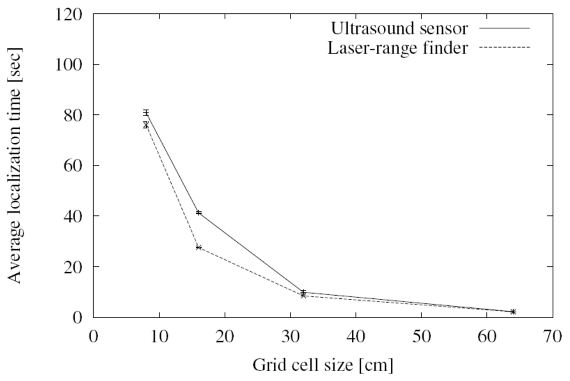
In contrast to the topological grid, the following figure illustrates the same scenario with a metric grid. Here, the grid consists of evenly-spaced points along the hallway. The metric grid uses many more points to represent the belief than the topological grid, although it does not require any significant features in the environment.


bel(x)
x
p(z|x)
x
bel(x)
x
bel(x)
x
p(z|x)
x
bel(x)
x
bel(x)
x

Computational complexity

For metric grids, the resolution of the grid trades-off accuracy with computational complexity. The following figure shows the average estimation error and the average localization time as a function of the grid size for two types of sensors. In general, the estimation error increases and the computational time decreases as the resolution increases.





The following methods improve the computational complexity so that a smaller grid resolution may be used.

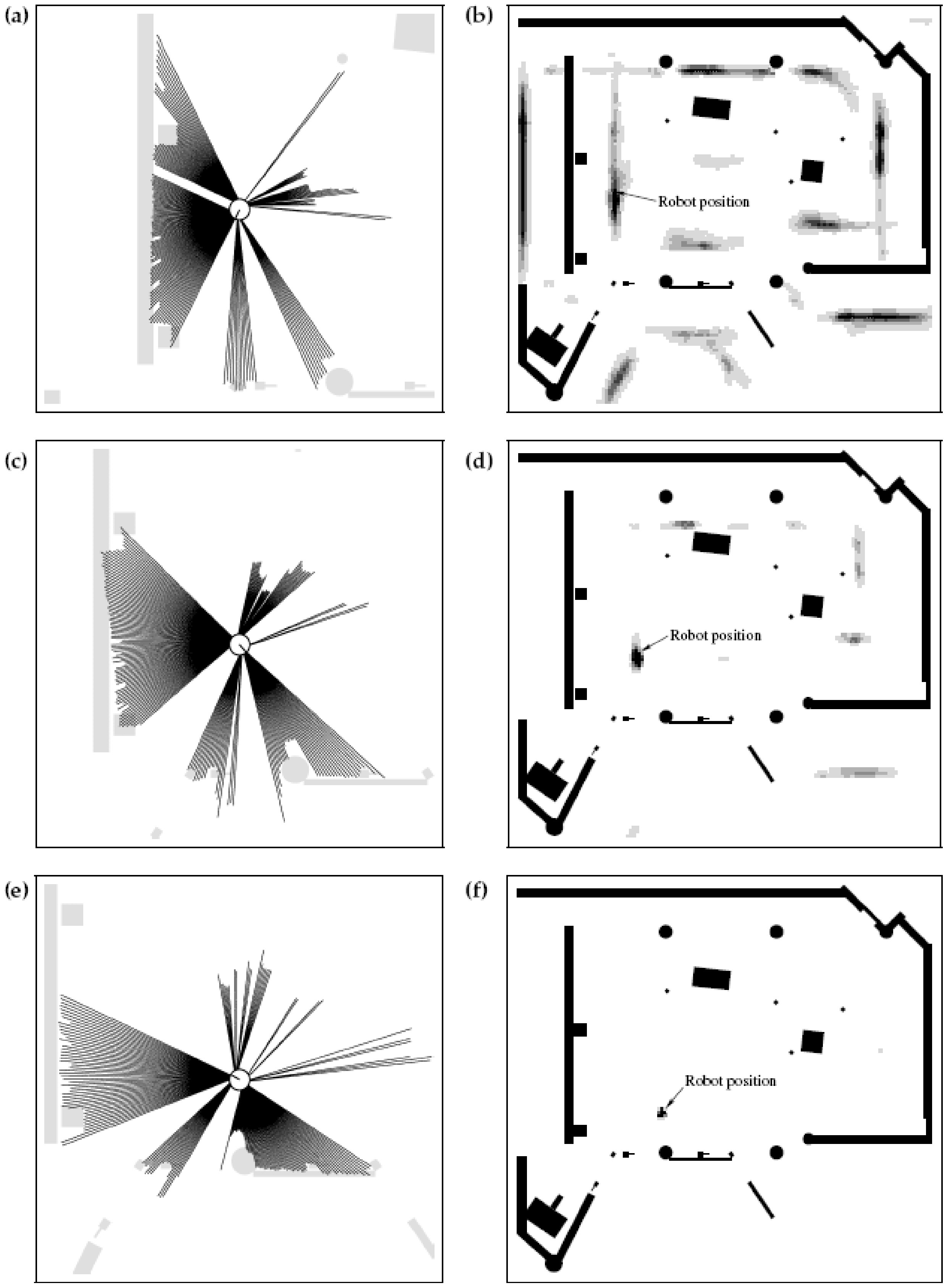
- **Model pre-caching.** Some sensor models require computationally intensive operations to compute, such as ray casting or the computation of likelihood fields for distance sensors. Pre-caching the expensive computations can enable use of a smaller resolution.
- **Sensor subsampling.** Instead of updating the belief every time a measurement is made, we could ignore some measurements and only update the belief on a subset of the measurements. We can subsample in space and/or time.
- **Delayed motion updates.** Instead of updating the belief every time a control action is applied, we could integrate the motion model and only update the belief after the location has changed by a significant amount. This reduces the accuracy since new measurements are fused into the belief using slightly outdated information, but it also reduces the computational complexity.
- **Selective updating.** Instead of updating the belief at all points in the grid, we may selectively update only a subset of grid cells. For instance, we may only update grid cells whose belief changes by more than some threshold.

Examples

Room

The following figure illustrates grid localization in a room environment. The robot knows the map and localizes itself within the map using two laser range-finders with the beam model. The environment is a metric grid with spatial resolution of 15 centimeters and angular resolution of 5 degrees. The robot initially has no knowledge of its location, so the prior belief is uniformly distributed over the domain.

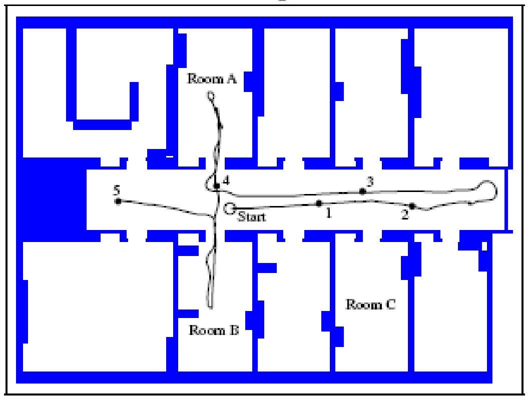
The plots on the left show the measured distances from the laser range-finder at an array of angles, where maximum distance readings are not shown. The plots on the right show the belief of the robot in grayscale after observing the measurement, with objects shown in black and the arrow pointing to the true robot position. After three scans, the robot is highly certain about its location within the map.


(a)
(b)
Robot position
(c)
(d)
Robot position
(e)
(f)
Robot position

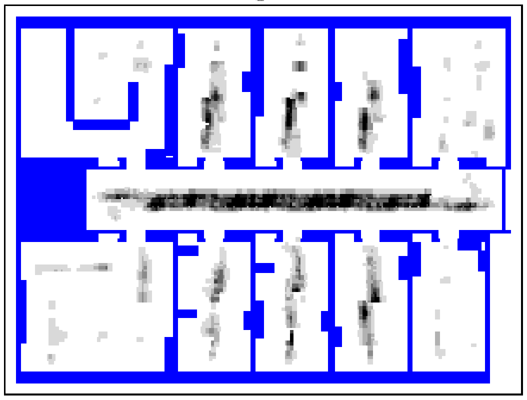
Office

The next example illustrates mobile robot localization in an office environment using a sonar sensor. Since the environment is symmetric, the robot must enter a room before it is able to know its pose with certainty. At reference pose 1, the robot is highly uncertain about its location due to the symmetric nature of the environment. At reference pose 2, it has traveled far enough that it knows that it is in a hallway instead of a room. At reference pose 3, it has narrowed its location down to two points that are symmetric with respect to each other. It is only after the robot enters a room that it is able to eliminate one of the poses to correctly localize itself within the map.

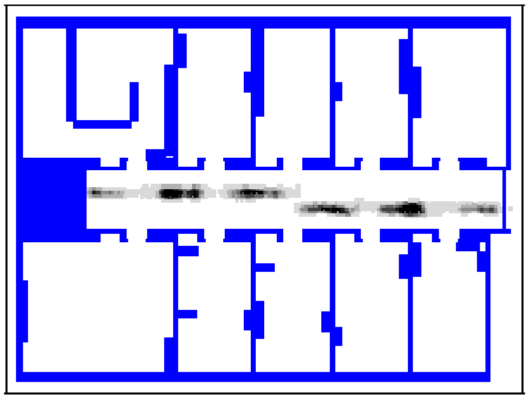
(a) Path and reference poses



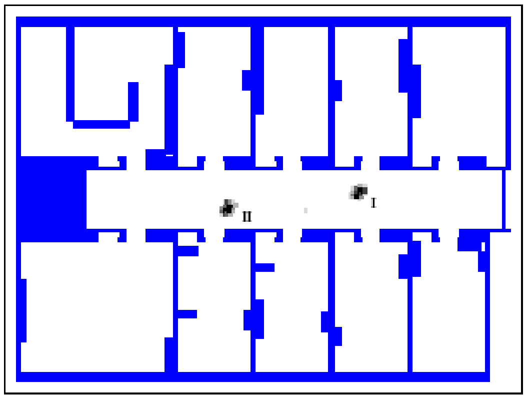
(b) Belief at reference pose 1



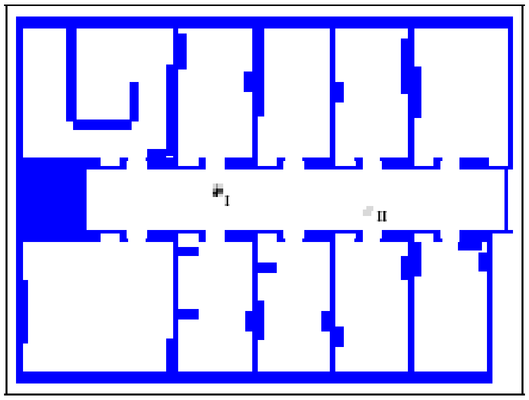
(c) Belief at reference pose 2



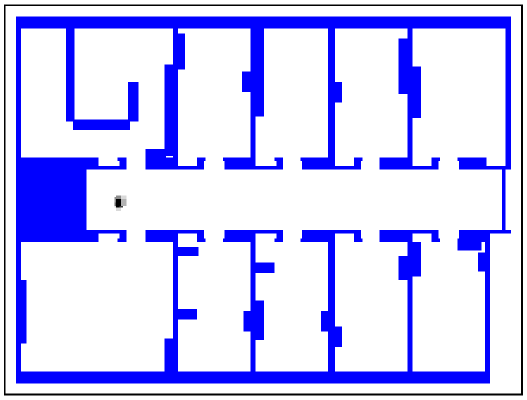
(d) Belief at reference pose 3



(e) Belief at reference pose 4

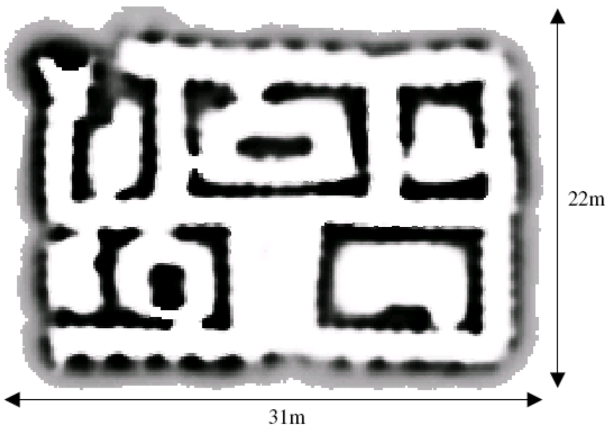


(f) Belief at reference pose 5

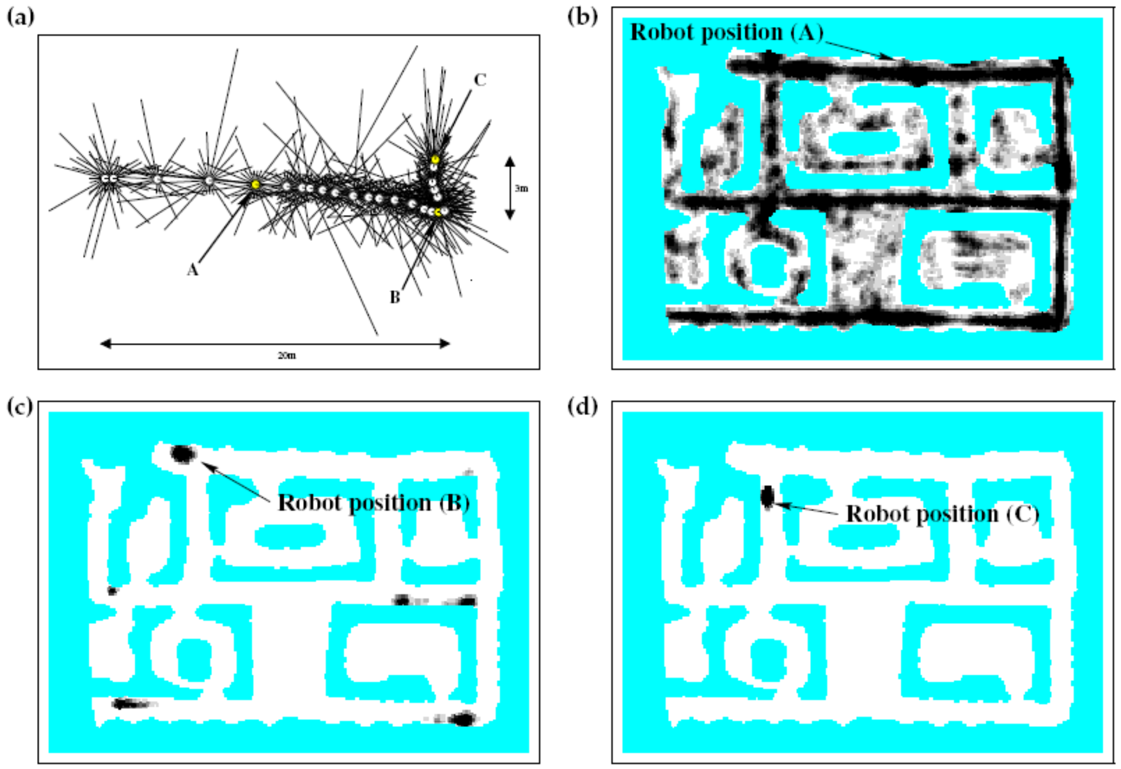


Association for the Advancement of Artificial Intelligence (AAAI) competition

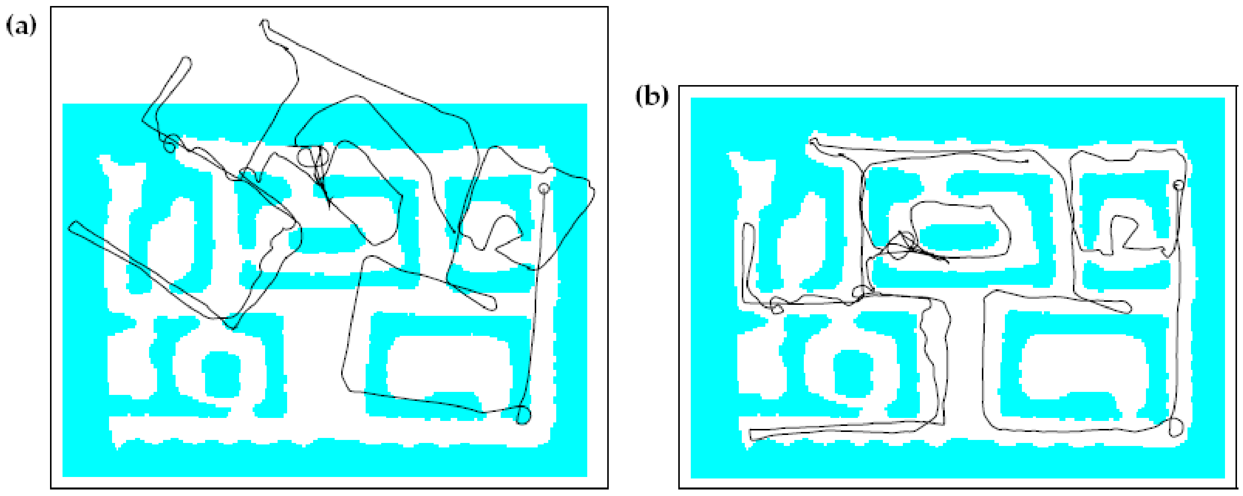
The next example illustrates mobile robot localization in a highly-symmetric environment using sonar sensors. The map of the environment is shown below.



The data set, which consists of odometry and sonar range scans, is shown in (a). The odometry indicates that the robot moves forward and then turns to the left. The walls of the environment are smooth, which causes many of the sonar measurements to be corrupted. The belief is updated using the beam model for the sonar sensors. When the robot is at position A after three sonar scans, the belief is almost uniformly distributed throughout the environment. After traveling further to position B, the belief is concentrated around several positions. By the time the robot is at position C, it has correctly localized itself within the environment.



Using sensor data to localize itself within the environment, the robot is much more certain about its position than if it had only relied on odometry data. The figure on the left shows the trajectory of the robot obtained from only the odometry information, while the figure on the right shows the corrected trajectory using the measurement data.



6.7 Monte-Carlo localization

Monte-Carlo localization uses a particle filter to localize the pose of a robot within a known map. Each particle in the particle filter is a hypothesis of the pose of the robot. The proposal distribution is the actuation model,

$$x^i_{t+1} \sim p(x_{t+1} \mid x_t, u_t)$$

and the correction uses the perception model,

$$w^i_t = \frac{\text{target}}{\text{proposal}} \propto p(z_t \mid x_t, m)$$

7

Mapping

Bayes filters can be used to construct a map given measurements of the environment at known poses of the robot. This is called *mapping with known poses*. The state of the Bayes filter is the map of the environment. The only modification to the Bayes filter is that the actuation and perception models are also conditioned on the (known) pose of the robot. This is a high-dimensional estimation problem since the map of the environment typically requires many variables to describe, such as the probability of each space being occupied or the locations of each landmark.

In this chapter, we make the unrealistic assumption that the pose of the robot is known. One application of such algorithms is in post-processing map estimates after running a SLAM algorithm. Many SLAM techniques do not generate maps fit for path planning and navigation. After obtaining an approximate trajectory from a SLAM algorithm, however, we can assume that the trajectory is correct and use it with the algorithms in this chapter to generate a more accurate map of the environment.

7.1 Problem setup

The posterior probability distribution for the problem of mapping with known poses is the conditional probability of the map given the entire history of robot poses and corresponding measurements,

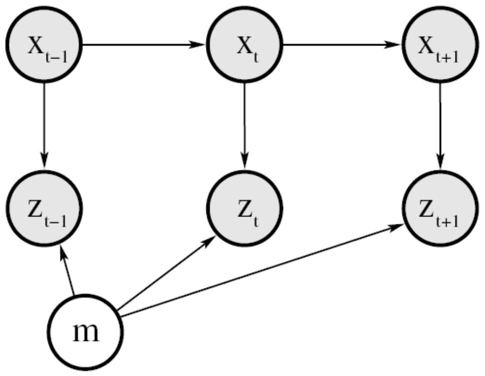
$$p(m \mid s_{1:t}, z_{1:t})$$

The control actions $a_{1:t}$ are not needed since we are assuming that we have access to the robot poses $s_{1:t}$.

Estimating a map using a mobile robot is challenging for several reasons.

- **Size.** The space of all possible maps is huge. Maps are often defined over a continuous space, so there are an infinite number of maps. Even in discretized spaces, a small resolution is required to obtain a good map which results in a large number of variables.
- **Noise.** The more noise in the robot's sensors and actuators, the more difficult it is to construct a map.
- **Correspondences.** When different places look alike, it is difficult for the robot to establish the correct correspondence between the measurement and the location in the map. Using incorrect correspondences can lead to catastrophic failure, so it is important for the algorithm to be robust to incorrect correspondences.

The graphical model for the problem of mapping with known poses is shown below. The known variables are the robot poses and the measurements shown in gray circles. The unknown quantity to be estimated is the map of the environment shown in a white circle. The arrows show the direction of causality: the measurement z_t is influenced by the known pose s_t of the robot and the unknown map m of the environment.



7.2 Occupancy grid mapping

In occupancy grid mapping, the map of the environment is approximated by a discretization over the space of locations into a finite set of grid cells. The map is represented by

$$m = \{m_1, m_2, \dots, m_K\}$$

where m_i represents the grid cell with index i and K is the number of grid cells. Each grid cell has an associated binary value that specifies whether or not the cell is occupied. We let $p(m_i)$ denote the probability that the grid cell with index i is occupied, with $1 - p(m_i)$ the probability that it is free.

The space of all occupancy grid maps is huge. For a map with K grid cells, there are 2^K possible maps. (For reference, 2^K overflows IEEE double precision when $K = 1024$.) It is therefore not computationally tractable to compute the full posterior. Instead, occupancy grid mapping make the following assumption.

Assumption. The occupancy of each grid cell is independent from the occupancy of all other grid cells. ■

Under this (restrictive) assumption, the posterior factors as the product of its marginal distributions,

$$p(m \mid s_{1:t}, z_{1:t}) \quad = \quad \prod_{i=1}^{K} p(m_i \mid s_{1:t}, z_{1:t})$$

where $p(m_i \mid s_{1:t}, z_{1:t})$ is the probability that grid cell i is occupied given the robot poses and measurements. The problem of estimating the posterior over the entire map now reduces to the problem of estimating the posterior for each individual grid cell independently. The occupancy value of each grid cell is a binary random variable that (we assume) does not change over time. Therefore, we can estimate this quantity using a binary filter with a static state.

The binary filter represents the posterior in log-odds form. For the occupancy grid mapping problem, the log odds of the posterior for the grid cell with index i at time t is the real number $\ell_{t,i}$ given by

$$\ell_{t,i} = \log \frac{p(m_i \mid s_{1:t}, z_{1:t})}{1 - p(m_i \mid s_{1:t}, z_{1:t})}$$

Given a measurement z_t, the binary filter updates the posterior for each grid cell i in the perceptual field of z_t by

$$\ell_{t+1,i} = \ell_{t,i} + \log \frac{p(m_i \mid s_t, z_t)}{1 - p(m_i \mid s_t, z_t)} - \ell_{0,i}$$

where $\ell_{0,i}$ is the prior probability of occupancy for grid cell i represented as a log odds ratio,

$$\ell_{0,i} = \log \frac{p(m_i = \text{occupied})}{p(m_i = \text{free})} = \log \frac{p(m_i)}{1 - p(m_i)}$$

The posterior can be recovered from its log-odd representation as

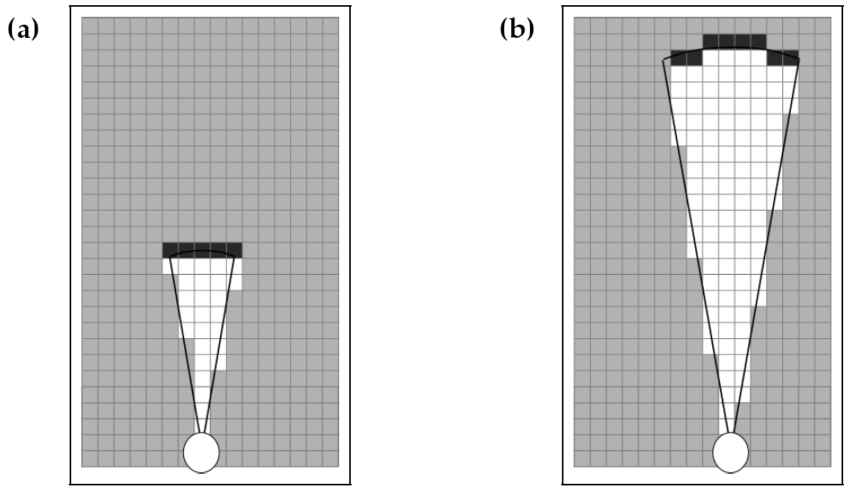
$$p(m_i \mid s_{1:t}, z_{1:t}) = 1 - \frac{1}{1 + \exp(\ell_{t,i})}$$

Implementation of the occupancy grid mapping algorithm requires the inverse sensor model $p(m_i \mid s_t, z_t)$, which is the probability that grid cell i is occupied given the measurement z_t taken from pose s_t.

Inverse range finder model

We now discuss an inverse sensor model for a range finder. This model first constructs the sensor cone, which are all grid cells in the range of the sensor. All grid cells within the sensor cone whose range is close to the measured range are assumed to be occupied and are assigned an occupancy value ℓ_{occ}. All other grid cells in the sensor cone are assumped to be unoccupied and are assigned an occupancy value ℓ_{free}. All grid cells outside of the sensor cone are not updated since they do not influence the measurement.

Two examples of the inverse range finder model are shown below. The robot is located at the white circle at the bottom of the figure and is oriented pointing up. The map is discretized into grid cells as shown. The black lines indicate the sides of the sensor cone. Black indicates cells that are assumed to be occupied given the range measurement, white cells are assumed to be free, and gray cells are not updated since they fall outside of the sensor cone.



A particular instance of the inverse range finder model designed for ultrasonic sensors is as follows. Suppose the robot is at the pose $s_t = (x, y, \theta)$ and observes the measurement $z_t = (z_r, z_\phi)$. Let $z_{\max}$ denote the maximum distance of the sensor, α denote the width of an obstacle, and β the angle of the sensor cone. Let (x_i, y_i) denote the center of mass of the grid cell with index i. The distance and angle from the robot to the grid cell is then

$$r_i = \sqrt{(x_i - x)^2 + (y_i - y)^2} \qquad \text{and} \qquad \phi_i = \text{atan2}(y_i - y, x_i - x) - \theta$$

Then the probability that a grid cell is occupied given the robot pose and measurement is

$$p(m_i \mid s_t, z_t) = \begin{cases} \ell_{\text{occ}} & \text{if } z_r < z_{\text{max}} \text{ and } |\phi_i - z_\phi| \leq \frac{\beta}{2} \text{ and } |r_i - z_r| \leq \frac{\alpha}{2} \\ \ell_{\text{free}} & \text{if } z_r < z_{\text{max}} \text{ and } |\phi_i - z_\phi| \leq \frac{\beta}{2} \text{ and } z_r < r_i - \frac{\alpha}{2} \\ \ell_{0,i} & \text{otherwise} \end{cases}$$

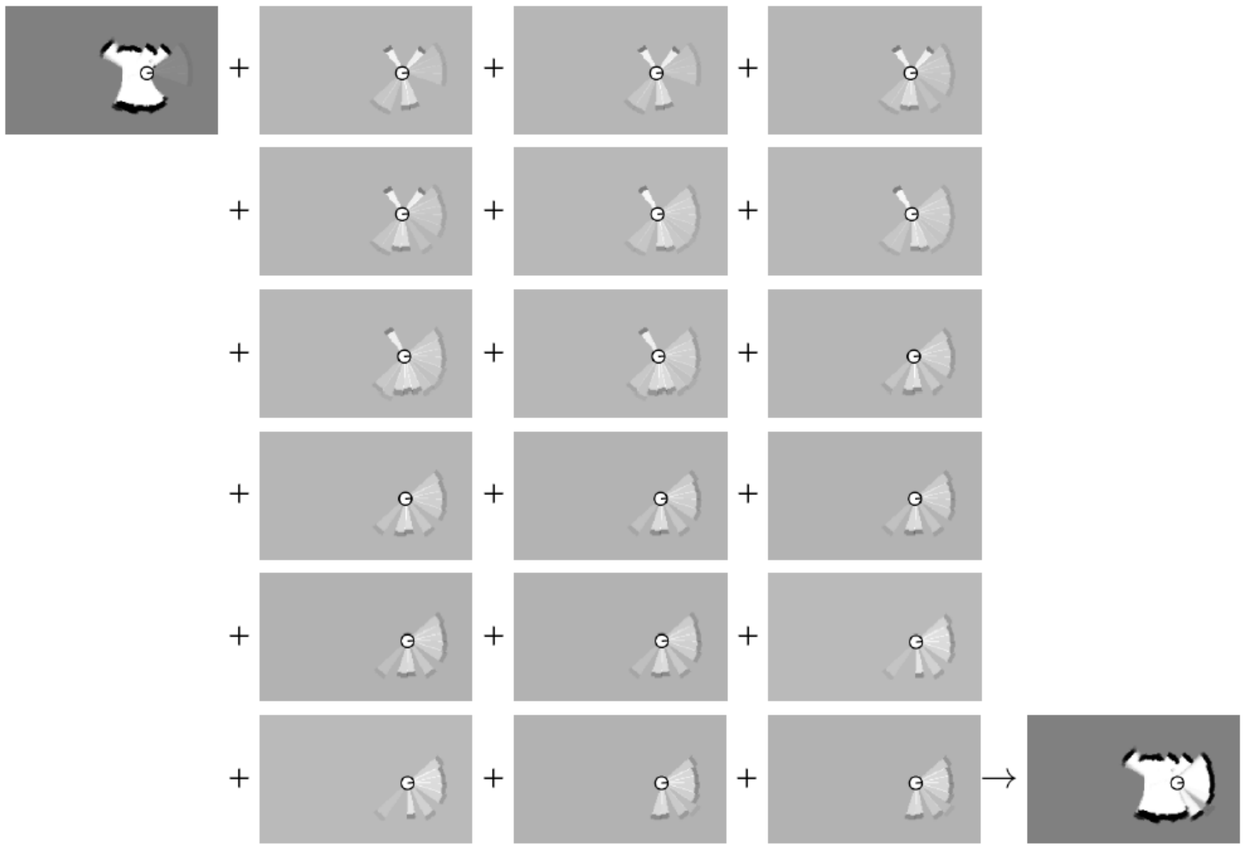
The model returns the prior $\ell_{0,i}$ whenever the measurement is outside of the sensor cone or the maximum distance is returned by the sensor. Inside of the sensor cone, the log odds probability ℓ_{occ} is used if the distance between the measured distance and the center-of-mass of the cell is less than half the width of an object, and the log odds probability ℓ_{free} is used for all other grid cells.

This inverse sensor model uses a cone to describe the spread of the measurement. This model is designed for ultrasonic sensors, since sound disperses as it travels. A similar model could be constructed for a laser range finder using a beam instead of a cone to more accurately model the characteristics of light.

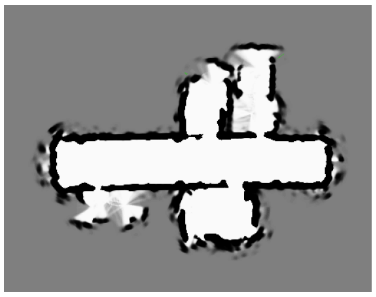
This model also only uses the sensor measurement to update the map. An alternative source of information is the pose of the robot itself. If we know the dimensions of the robot and that the robot is at a certain pose, then there is very low probability that the space occupied by the robot is also occupied by another object. Such information can easily be included in the inverse sensor model.

Examples

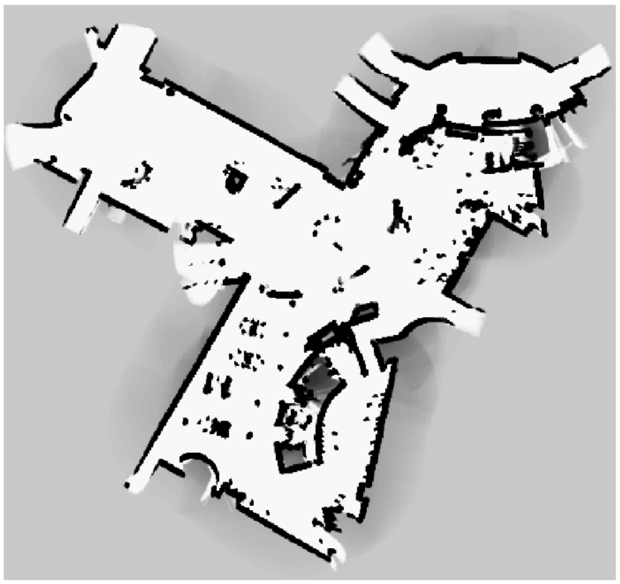
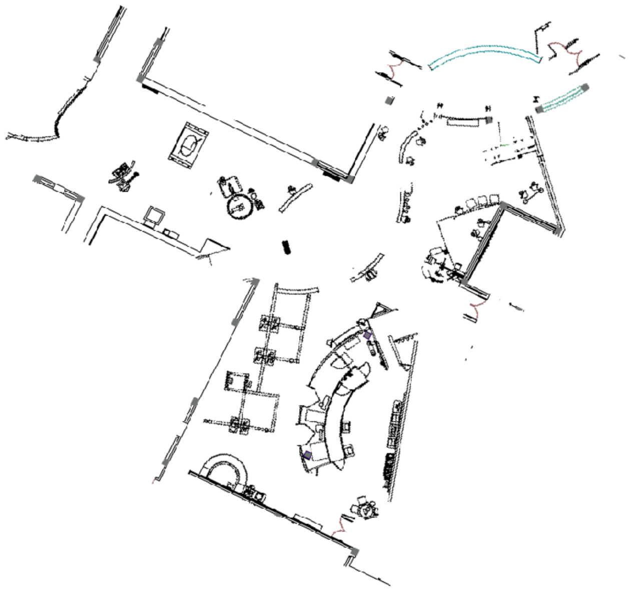
We now illustrate occupancy grid mapping through several examples. The first example constructs an occupancy map using an ultrasonic sensor. The figure below shows the prior map obtained after running the algorithm for a while on the top left. The middle images show local maps generated using the inverse sensor model described above. The information from these maps is fused together with the prior map to produce the posterior map shown on the bottom right.



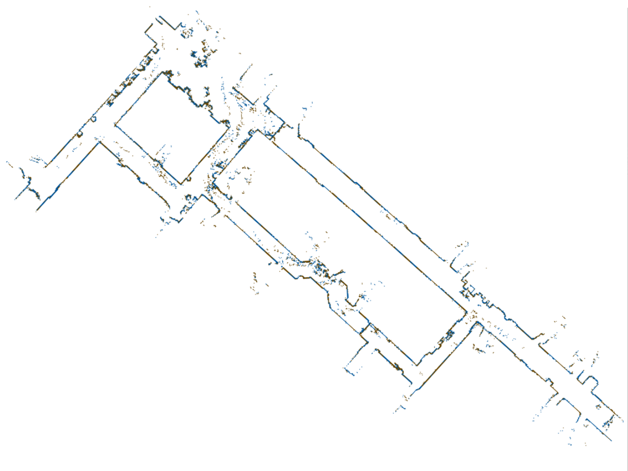
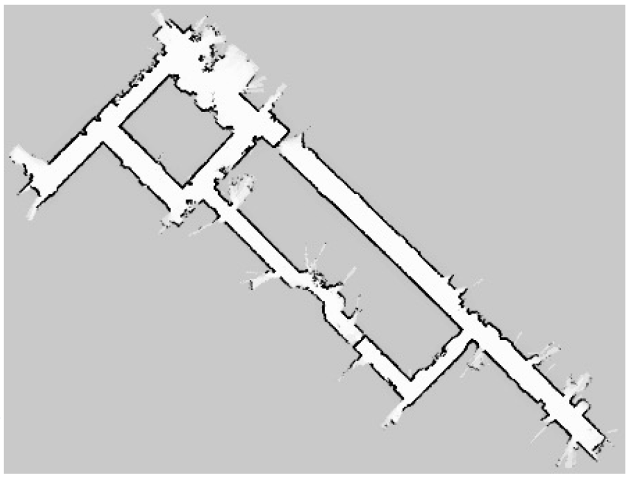
After running the algorithm for a while, the robot has the more complete occupancy map shown below.



The next example shows the occupancy map obtained from a robot with a laser range finder along with the corresponding blueprint of the building. While each cell in the occupancy map has an associated probability of occupancy, the map is almost entirely black (occupied) or white (free) indicating that the robot knows the map with high certainty. Comparing the occupancy map to the blueprint, the map shows all major structural elements along with dynamic elements (such as people) that were present during the mapping. Notice that there are some small discrepancies between the blueprint and the map!

As pointed out in the beginning of this chapter, one of the main motivations for occupancy grid maps are to post-process maps obtained from a SLAM algorithm. The following example illustrates the importance of this post processing. The image on the left shows the raw sensor data after using a SLAM algorithm to correct the pose information. Dots indicate the detection of an obstacle, such as a person walking in front of the robot. The image on the right is the occupancy map constructed from the sensor data with the poses obtained from the SLAM algorithm. By fusing the measurements, the occupancy map is dark near walls (indicating occupied) and white inside the hallways (indicating free) without the presence of all the dynamic obstacles. Having such a noise-free map is important during path planning, since a control algorithm using the raw sensor data would try to avoid all of the many obstacles in the map.

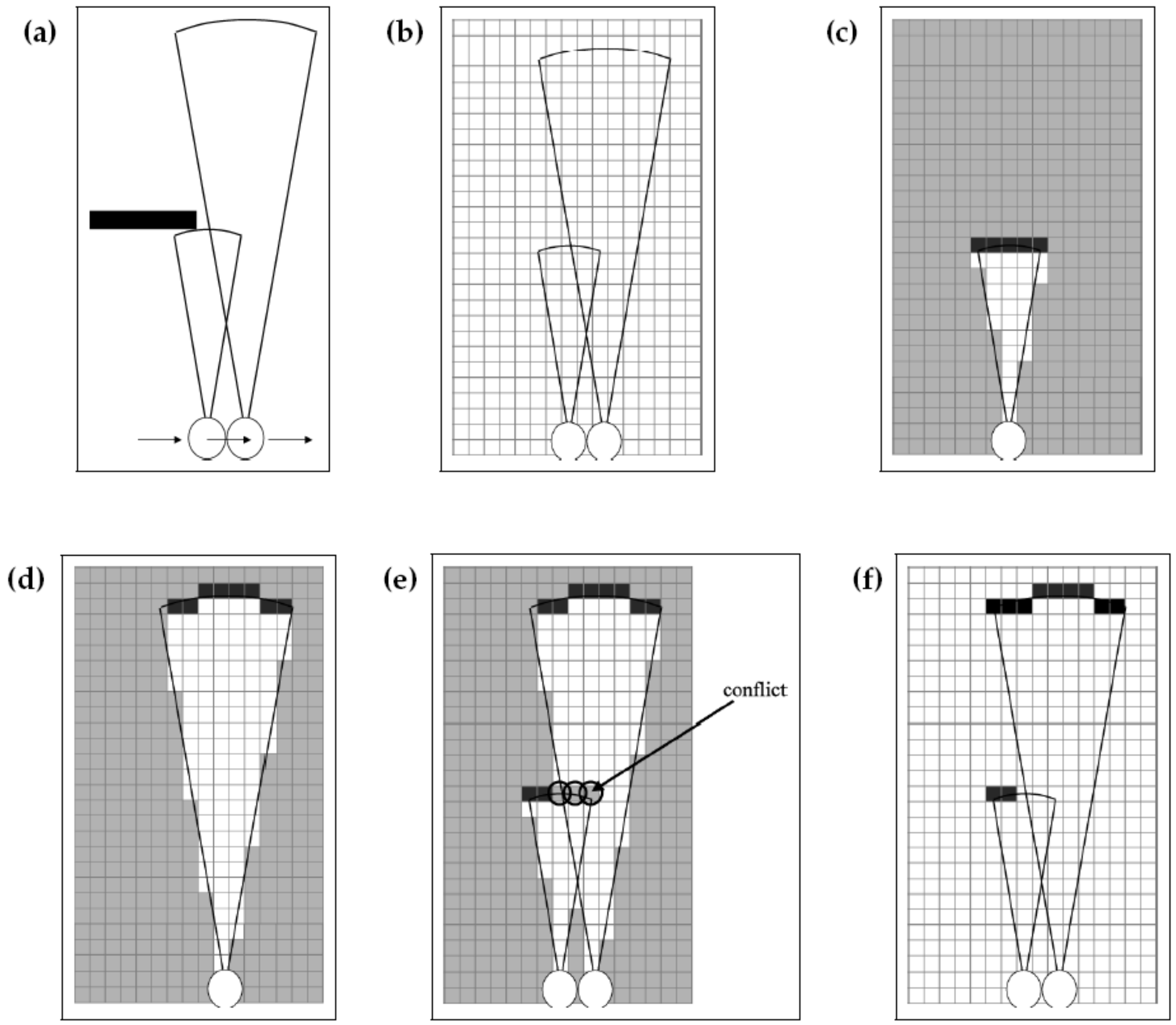



Maximum a posteriori estimation

The occupancy grid mapping algorithm assumes that the occupancy of each grid cell is independent from that of all other grid cells so that the posterior factors as the product of its marginal distributions over all grid cells,

$$p(m \mid s_{1:t}, z_{1:t}) \quad = \quad \prod_{i=1}^{K} p(m_i \mid s_{1:t}, z_{1:t})$$

A possible issue with this assumption is shown in the figure below. Here, the robot is facing a wall and receives two noise-free sonar range measurements. Because of our assumption, the occupancy map is updated with a high probability of occupancy along the entire arc of its measured range. The occupancy of these grid cells cannot be coupled since we assumed that they are independent. Figures (c) and (d) show the updates due to the two sensor readings, and the results are combined in (e). The overlapping region, however, results in an inconsistency in the map: the occupancy map suggests that several points extending from the wall may be occupied, even though such a map is not consistent with the two measurements. A more accurate map that is consistent with both measurements is shown in (f). This map assumes there is an object somewhere in the measurement cone, but not necessarily across the entire cone. Because of our assumption that grid cells are independent, we lose the ability to consider such dependencies when constructing the map.



The standard occupancy grid mapping algorithm sums up positive and negative evidence for occupancy, which partially resolves this problem. However, the produced occupancy map depends on the relative frequencies of the two types of measurements which is undesirable.

Instead of taking the occupancy map as the full posterior, we can reduce the negative effects of our independence assumption by choosing the map that has the highest probability. In maximum a priori occupancy grid mapping, we choose the map as the mode of the posterior.

Definition (MAP estimate)**.** The maximum a posteriori (MAP) estimate of the map given the measurements and poses is the mode of the posterior,

$$\underset{m}{\text{maximize}} \; \log p(m \mid s_{1:t}, z_{1:t})$$

We choose to maximize the log of the posterior for convenience, as this has the same optimizer as maximizing the posterior itself (since the logarithm is an increasing function). While we do not directly have access to the posterior distribution, we can find the MAP estimate as follows.

Theorem. Suppose the prior occupancy $p(m_i)$ of each grid cell i is the same, and let

$$\ell_0 = \log \frac{p(m_i)}{1 - p(m_i)}$$

denote this probability in log-odds form. Then, the MAP estimate is the solution to the optimization problem

$$\underset{m}{\text{maximize}} \; \sum_t \log p(z_t \mid s_t, m) + \ell_0 \sum_i m_i$$

whose variables are the occupancy values m_i of each grid cell i.

The MAP estimate is the solution to an optimization problem that depends on the (forward) measurement model $p(z \mid s, m)$ and the prior occupancy of a grid cell in log-odds form ℓ_0.

Proof. From Bayes rule, the posterior is

$$p(m \mid s_{1:t}, z_{1:t}) = \frac{p(z_{1:t} \mid s_{1:t}, m)\, p(m \mid s_{1:t})}{p(z_{1:t} \mid s_{1:t})}$$

Since the states $s_{1:t}$ do not affect the map and the denominator is constant with respect to the map, the log of the posterior factors into a map prior and a measurement likelihood,

$$\log p(m \mid s_{1:t}, z_{1:t}) = \log(z_{1:t} \mid s_{1:t}, m) + \log p(m) + \text{constant}$$

Since the measurements are independent when conditioned on the pose and the map (by assumption), the log-likelihood of the measurements decomposes into the summation

$$\log(z_{1:t} \mid s_{1:t}, m) = \sum_t \log p(z_t \mid s_t, m)$$

This yields the first term in the MAP optimization problem. To obtain the second term, we assume that the prior occupancy of grid cells are independent from each other. Then, since the occupancy of each grid cell is a binary random variable, the prior of any map is the product

$$p(m) = \prod_{i=1}^{K} p(m_i)^{m_i} (1 - p(m_i))^{1 - m_i}$$

where $p(m)$ is the prior for the occupancy of the entire map, $m_i \in \{0, 1\}$ is the occupancy of grid cell i, and $p(m_i)$ is the prior probability of occupancy of grid cell i. Assuming that the prior probability of occupancy of each grid cell is the same (that is, $p(m_i)$ is constant with respect to i), we can split the second term into one that is constant with respect to i, which we can then replace with our generic constant η to obtain

$$p(m) = \eta \prod_{i=1}^{K} p(m_i)^{m_i}(1-p(m_i))^{-m_i}$$

The logarithm of the prior is then

$$\log p(m) = \ell_0 \sum_{i=1}^{K} m_i + \text{constant}$$

The constant term is constant with respect to the map m and therefore does not affect the MAP estimate, so we can omit it from the optimization problem without changing the optimal map. Substituting these expressions for the log of the posterior produces the given expression. ■

Algorithm

Finding the maximum a posteriori occupancy grid map requires solving a nonconvex optimization problem, which is difficult in general. A simple heuristic approach is as follows. We initialize the map with no obstacles by setting $m_i = 0$ for all i. We then repeatedly iterate over the grid cells and "flip" the occupancy of the grid cell if it increases the posterior. For grid cell i, we set

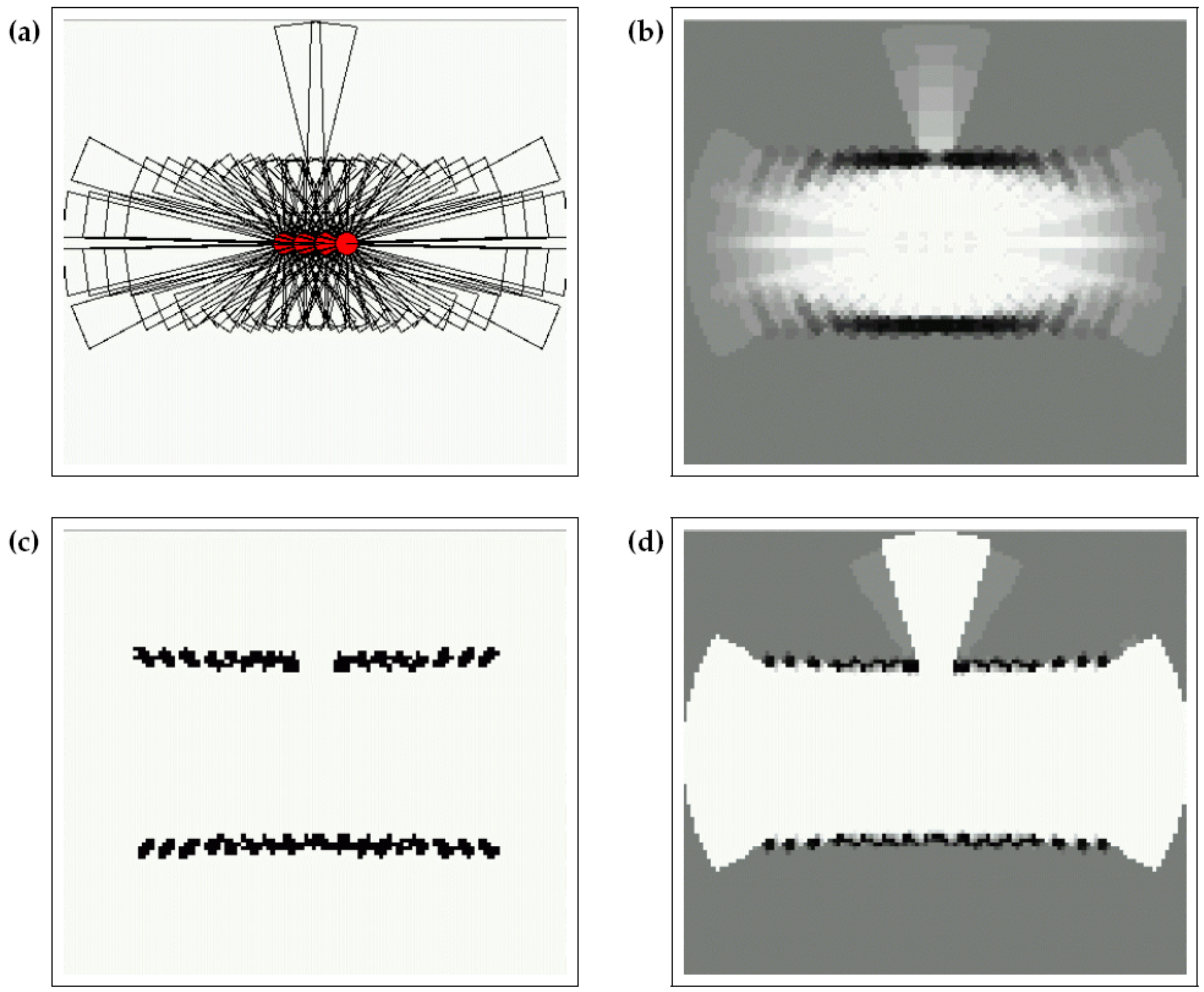
$$m_i = \operatorname*{arg\,max}_{k\in\{0,1\}} \; k\,\ell_0 + \sum_t \log p(z_t \mid s_t, m \text{ with } m_i = k)$$

Each grid cell in the resulting map is either occupied ($m_i = 1$) or unoccupied ($m_i = 0$). At each iteration of this algorithm, the value of the posterior cannot get smaller, but could possibly get bigger (if flipping the occupancy of grid cell i increases the posterior). While this algorithm is not guaranteed to find the maximum a posteriori estimate of the map, it typically works well in practice.

In contrast to the standard occupancy grid mapping algorithm, the MAP occupancy grid mapping algorithm uses the *forward* measurement model $p(z \mid s, m)$.

Example

An example of maximum a posteriori occupancy grid mapping is shown below, where a robot travels in a hallway past a door. Figure (a) shows the noise-free range measurements, some of which detect the open door while others reflect off the wall. The results of standard occupancy grid mapping are shown in (b), which fails to detect the open door. The maximum a posteriori map is shown in (c), which clearly shows the walls with the open door in the top wall. This map is clearly better for robot navigation, as it allows the robot to travel through the open door. The map in (d) shows the residual uncertainty, which is the amount by which flipping the occupancy of the grid cell decreases the log-likelihood. This map clearly indicates the uncertainty of objects behind obstacles.



Limitations

The proposed algorithm for computing the maximum a posteriori occupancy grid map has several limitations.

- The MAP estimate only returns the mode of the posterior, which provides no information as to the certainty in the map. We can approximate the uncertainty by computing the sensitivity of the log-likelihood with respect to each individual grid cell.
- The MAP estimate is computed using all of the pose and measurement information, and cannot be updated incrementally as in the Bayes filter. One way to improve convergence of the algorithm is to initialize the map with the result of the standard occupancy grid mapping algorithm.
- Flipping a single grid cell does not affect all measurements. A more computationally efficient algorithm takes this into account to avoid redundant computations.

8

Simultaneous Localization and Mapping

The localization and mapping problems in robotics are inherently interdependent. To construct a map using sensor data, the robot must know where it is in the environment. But to know where it is, the robot must have a map. In some applications, the robot may be given a map (such as a robotic manipulator knowing the positions of nearby objects) or its location (such as using GPS). When these are both unknown, however, the robot must simultaneously estimate its position while constructing the map. In this chapter, we introduce simultaneous localization and mapping (SLAM), one of the fundamental problems in robotics.

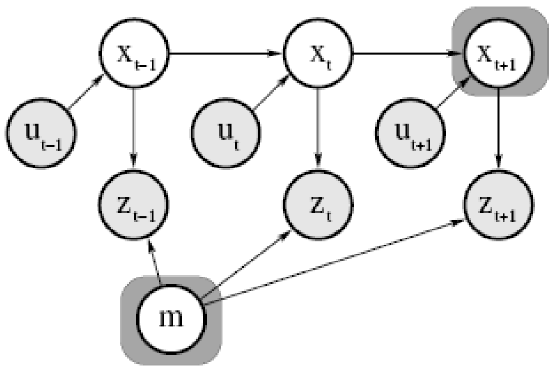
8.1 Overview

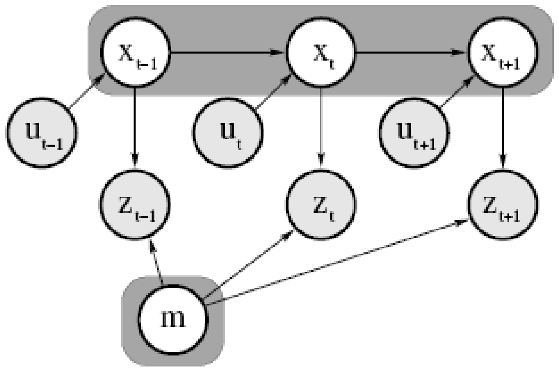
There are two main types of SLAM problems depending on what the robot seeks to estimate. The online SLAM problem is to estimate the posterior probability distribution of the current robot pose and map given the measurements and control actions:

$$p(x_t, m \mid z_{1:t}, u_{1:t})$$

Algorithms for online SLAM are often incremental in that they process measurements and control actions as they occur and then discard the measurements after they are used to update the estimate. Alternatively, the full SLAM problem is to estimate the posterior probability distribution of the entire trajectory of robot poses and map given the measurements and control actions:

$$p(x_{1:t}, m \mid z_{1:t}, u_{1:t})$$





Since $p(x) = \int p(x, y)\,\mathrm{d}y$, the online SLAM posterior can be obtained from the full SLAM posterior by integrating out the past states,

$$p(x_t, m \mid z_{1:t}, u_{1:t}) = \int \cdots \int \int p(x_{1:t}, m \mid z_{1:t}, u_{1:t}) \, \mathrm{d}x_1 \, \mathrm{d}x_2 \ldots \mathrm{d}x_{t-1}$$

We can view the information in the SLAM problem as a graph, where the nodes in the graph are variables (poses, measurements, control actions, or the map) and edges indicate causal relationships between the variables. This graphical model for the online and full SLAM problems is shown above. White circles indicate unknown variables (robot poses and the map), while shaded circles indicate observed variables (measurements and control actions). The dark regions indicate the unknown variables that are estimated in each problem. The edges indicate that the robot pose is influenced by the control actions and previous pose, while the measurements are influenced by the robot pose and the map.

8.2 Properties

Beyond that of localization and mapping separately, simultaneous localization and mapping is difficult for several reasons.

- **Cycles.** Cycles occur when the robot returns to a location where it has already been. Depending on the length of the trajectory, the actuation noise may be very large making it difficult to realize that the two places are in fact the same.

8.3 Algorithms

The basic idea behind SLAM algorithms is for the robot to simultaneously estimate its belief of the robot state and the map of the environment using a Bayes filter:

$$b_t(s_t, m_t) = p(s_t, m_t \mid z_{1:t}, a_{1:t})$$

Some straighforward SLAM algorithms that use this approach are the following.

EKF SLAM

SLAM via the extended Kalman filter.

Known correspondences

Unknown correspondences

Sparse extended information filter

SLAM via the sparse extended information filter

Fast SLAM

SLAM via the particle filter

Grid-based fast SLAM

Graph SLAM

Graph-based SLAM is an algorithm for the full simultaneous localization and mapping problem. A graph is a mathematical object that consists of nodes and edges. In graph-based SLAM, the nodes of the graph correspond to poses of the robot and features in the map. The robot creates an edge in the graph between two poses if they are adjacent in time (so that the robot has odometry measurements relating the two poses), or if the robot observes the same part of the environment from both poses. The poses of the robot and the landmark locations are then obtained by solving a nonlinear weighted least squares problem.

Problem setup

Suppose we are given a set of measurements $z_{1:t}$ and a set of control variables $a_{1:t}$, where the notation $z_{1:t}$ denotes the set of variables $\{z_1, z_2, \ldots, z_t\}$ and similarly for $a_{1:t}$. The ultimate goal for the full SLAM problem would be to compute the posterior $p(s_{0:t}, m \mid z_{1:t}, a_{1:t})$. This is a high-dimensional probability distribution, and it is typically too difficult to obtain the entire posterior. Instead, we will find the robot poses and map that are most consistent with the data. In particular, we will find the robot poses and map with the highest probability given the control actions and measurements. This is known as the maximum a posteriori (MAP) estimate since it maximizes the posterior after taking into account the available information. To find the MAP estimate, we first factor the full SLAM posterior as the product of a prior and the actuation and perception models at each time.

Theorem (Full SLAM posterior)**.** The full SLAM posterior is the product of the prior with the actuation and perception models at each time,

$$p(s_{0:t}, m \mid z_{1:t}, a_{1:t}) \quad = \quad \eta\, p(s_0, m) \prod_{\tau=0}^{t-1} p(s_{\tau+1} \mid s_\tau, a_\tau)\, p(z_\tau \mid s_\tau, m)$$

where the parameter η is a normalizing constant.

Proof. Using Bayes rule, the posterior factors as

$$p(s_{0:t}, m \mid z_{1:t}, a_{1:t}) = \eta\, p(z_t \mid s_{0:t}, m, z_{1:t-1}, a_{1:t})\, p(s_{0:t}, m \mid z_{1:t-1}, a_{1:t})$$

By definition of the state, the first probability simplifies to the measurement model

$$p(z_t \mid s_{0:t}, m, z_{1:t-1}, a_{1:t}) = p(z_t \mid s_t, m)$$

Similarly, we can factor the second probability as

$$\begin{aligned} p(s_{0:t}, m \mid z_{1:t-1}, a_{1:t}) &= \eta\, p(s_t \mid s_{0:t-1}, m, z_{1:t-1}, a_{1:t})\, p(s_{0:t-1}, m \mid z_{1:t-1}, a_{1:t}) \\ &= \eta\, p(s_t \mid s_{t-1}, a_t)\, p(s_{0:t-1}, m \mid z_{1:t-1}, a_{1:t-1}) \end{aligned}$$

Substituting these back into the expression for the full SLAM posterior yields the recursive update

$$p(s_{0:t}, m \mid z_{1:t}, a_{1:t}) = \eta\, p(z_t \mid s_t, m)\, p(s_t \mid s_{t-1}, a_t)\, p(s_{0:t-1}, m \mid z_{1:t-1}, a_{1:t-1})$$

The final expression is then found by iterating over time t. ∎

Instead of maximizing the posterior in this form, it will be more convenient to represent it in information form, which is the negative logarithm of the posterior. In information form, the full SLAM posterior is the summation

$$I(s_{0:t}, m \mid z_{1:t}, a_{1:t}) = I(s_0, m) + \sum_{\tau=0}^{t-1} I(s_{\tau+1} \mid s_\tau, a_\tau) + \sum_{\tau=0}^{t-1} I(z_\tau \mid s_\tau, m) + \text{constant}$$

Assumption (Gaussian noise)**.** We now assume that the prior, actuation model, and perception model are all normally distributed with known distribution.

- The initial robot pose and map are independent random variables, so the prior $p(s_0, m)$ factors as $p(s_0)\, p(m)$. We assume that we have no prior knowledge of the map, so $p(m)$ is constant. We also assume that the prior robot pose is normally distributed with zero mean and information matrix Ω_0, which we typically take as a large multiple of the identity matrix. This corresponds to a high certainty that the robot starts at the origin of the global coordinate system[1].
- The actuation model $p(s_{t+1} \mid s_t, a_t)$ is normally distributed with mean $g(s_t, a_t)$ and covariance matrix R_t.
- The perception model $p(z_t \mid s_t, m)$ is normally distributed with mean $h(s_t, m)$ and covariance matrix Q_t.

■

Recall that the information form of a Gaussian random variable is a quadratic function centered about the mean with curvature given by the information matrix. Therefore, under these assumptions, the posterior in information form is

$$I(s_{0:t}, m \mid z_{1:t}, a_{1:t}) = \|s_0\|^2_{\Omega_0} + \sum_{\tau=0}^{t-1} \|s_{\tau+1} - g(s_\tau, a_\tau)\|^2_{R_\tau^{-1}} + \sum_{\tau=0}^{t-1} \|z_\tau - h(s_\tau, m)\|^2_{Q_\tau^{-1}} + \text{constant}$$

Since the negative logarithm is a monotonically decreasing function for nonnegative numbers (which probabilities are), the poses and map that maximize the posterior are those that minimize the information of the posterior. We can also ignore the constant in the optimization since it does not change the solution. Therefore, the robot poses and map that are most consistent with the control actions and measurements under the Gaussian assumption on the noise are the solution to the optimization problem

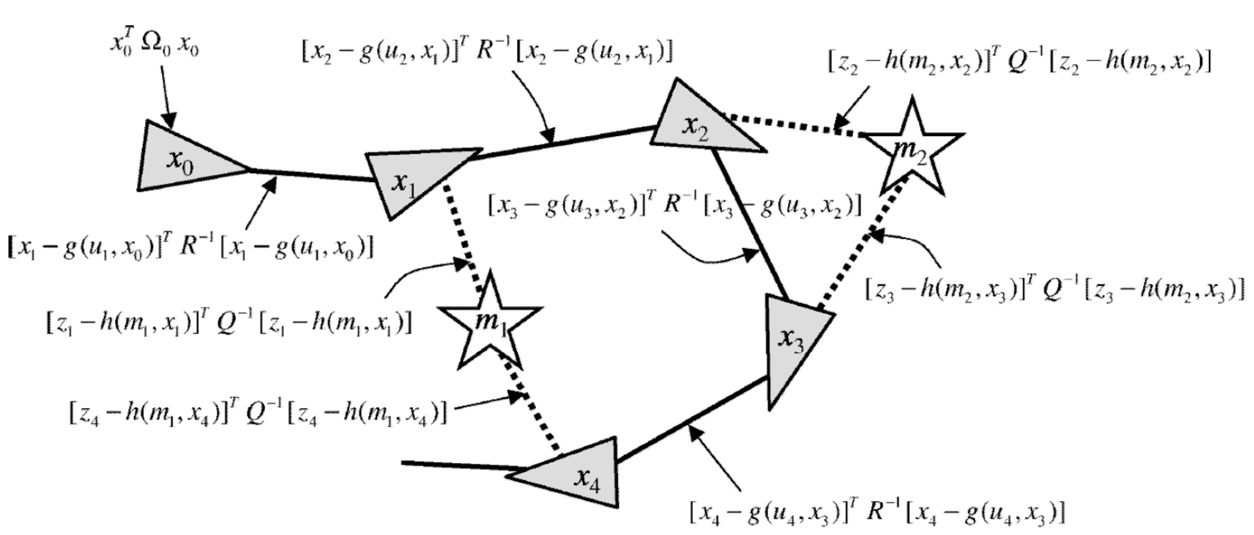
$$\underset{s_{0:t},\, m}{\text{minimize}} \quad \underbrace{\|s_0\|^2_{\Omega_0}}_{\text{anchor}} \quad + \quad \underbrace{\sum_\tau \|s_{\tau+1} - g(s_\tau, a_\tau)\|^2_{R_\tau^{-1}}}_{\text{actuation residual}} \quad + \quad \underbrace{\sum_\tau \|z_\tau - h(s_\tau, m)\|^2_{Q_\tau^{-1}}}_{\text{perception residual}}$$

This is a least squares problem in which the cost is the sum of three terms: the anchor, the actuation residual, and the perception residual. The prior anchors the solution to make it unique (without the prior, we could shift all of the robot poses and the map by any amount to obtain another solution with the same least squares error).

Information graph

We can visualize the SLAM information using a graph, where nodes in the graph are robot poses (triangles) and landmark locations (stars). Edges in the graph indicate available information that relates two nodes. Edges between robot poses (solid lines) represent information from the actuation model, while edges between a robot pose and landmark (dashed lines) represent information from the perception model. Associated with each edge is a residual and an information matrix of the associated noise. The graph SLAM algorithm seeks to find the robot poses and landmark locations that are most consistent with the available information by solving a weighted least squares problem, where the weights are the information matrices of the actuation and perception noise.

[1] We could instead enforce that the initial robot pose is *exactly* the origin, but this would lead to a constrained optimization problem, which is in general more difficult to solve.


$x_0^T \Omega_0 x_0$
$[x_2 - g(u_2, x_1)]^T R^{-1} [x_2 - g(u_2, x_1)]$
$[z_2 - h(m_2, x_2)]^T Q^{-1} [z_2 - h(m_2, x_2)]$
x_0
x_1
x_2
m_2
$[x_3 - g(u_3, x_2)]^T R^{-1} [x_3 - g(u_3, x_2)]$
$[x_1 - g(u_1, x_0)]^T R^{-1} [x_1 - g(u_1, x_0)]$
$[z_3 - h(m_2, x_3)]^T Q^{-1} [z_3 - h(m_2, x_3)]$
$[z_1 - h(m_1, x_1)]^T Q^{-1} [z_1 - h(m_1, x_1)]$
m_1
x_3
$[z_4 - h(m_1, x_4)]^T Q^{-1} [z_4 - h(m_1, x_4)]$
x_4
$[x_4 - g(u_4, x_3)]^T R^{-1} [x_4 - g(u_4, x_3)]$

Linearization. Denote the set of all robot poses and all landmarks in the map for the full SLAM problem (which is typically a very high-dimensional vector!) as

$$x = (s_0, s_1, \dots, s_t, \ell_1, \ell_2, \dots, \ell_N)$$

Let $r(x)$ denote the residual of the least squares problem, which consists of all the actuation and perception residuals along with the anchor constraint. If the actuation and perception models are nonlinear (as they typically are), then the posterior is not Gaussian, or equivalently, the posterior in information form is not quadratic. By linearizing the residual about a point x_k, however, the posterior in information form is approximately quadratic,

$$I(s_{0:t}, m \mid z_{1:t}, a_{1:t}) \quad \approx \quad \tfrac{1}{2} x^{\mathsf{T}} \Omega(x_k)\, x - x^{\mathsf{T}} \xi(x_k) + \text{constant}$$

where $\Omega(x_k)$ and $\xi(x_k)$ are the information matrix and information vector linearized about the point x_k. Since the information form of the posterior is the sum of many terms, the information matrix and information vector of the posterior linearized about a point x have the additive structure

$$\Omega(x) = \sum_{i,j} \Omega_{ij}(x) \qquad \text{and} \qquad \xi(x) = \sum_{i,j} \xi_{ij}(x)$$

where i and j are any two nodes in the graph that are connected by an edge. These nodes may both be poses for an actuation-based edge, or they may be a pose and a landmark for an observation-based edge. This illustrates that information is an additive quantity, so the total information is the sum of all the individual information resulting from the prior and the actuation and perception models at each time.

The matrix $\Omega_{ij}(x)$ and vector $\xi_{ij}(x)$ are the information matrix and information vector of the linearized posterior for the edge connecting nodes i and j. From studying the least squares problem, we know that these are given in terms of the residual $r_{ij}(x)$, its Jacobian $J_{ij}(x)$, and the weight matrix W_{ij} as

$$\Omega_{ij}(x) = J_{ij}(x)^{\mathsf{T}} W_{ij} J_{ij}(x) \qquad \text{and} \qquad \xi_{ij}(x) = J_{ij}(x)^{\mathsf{T}} W_{ij} r_{ij}(x)$$

Later in this chapter, we will see how to compute the residual and its Jacobian for various types of actuation and perception models. In each case, the weight matrix is the information matrix of the noise.

Sparsity of the information matrix. The information matrix and information vector due to a single edge have a particular structure. In particular, they are *sparse*, meaning that only a few of the elements are nonzero. Understanding this structure will guide us in how to update the full information matrix and vector from those for a single edge.

Since the residual between two nodes i and j only involves those nodes, the Jacobian is zero with respect to all other nodes in the graph. For instance, if the nodes are both robot poses, then the Jacobian has the form

$$J_{ij} = \begin{bmatrix} 0 & \dots & 0 & \frac{\partial r_{ij}}{\partial s_i} & 0 & \dots & 0 & \frac{\partial r_{ij}}{\partial s_j} & 0 & \dots & 0 \end{bmatrix}$$

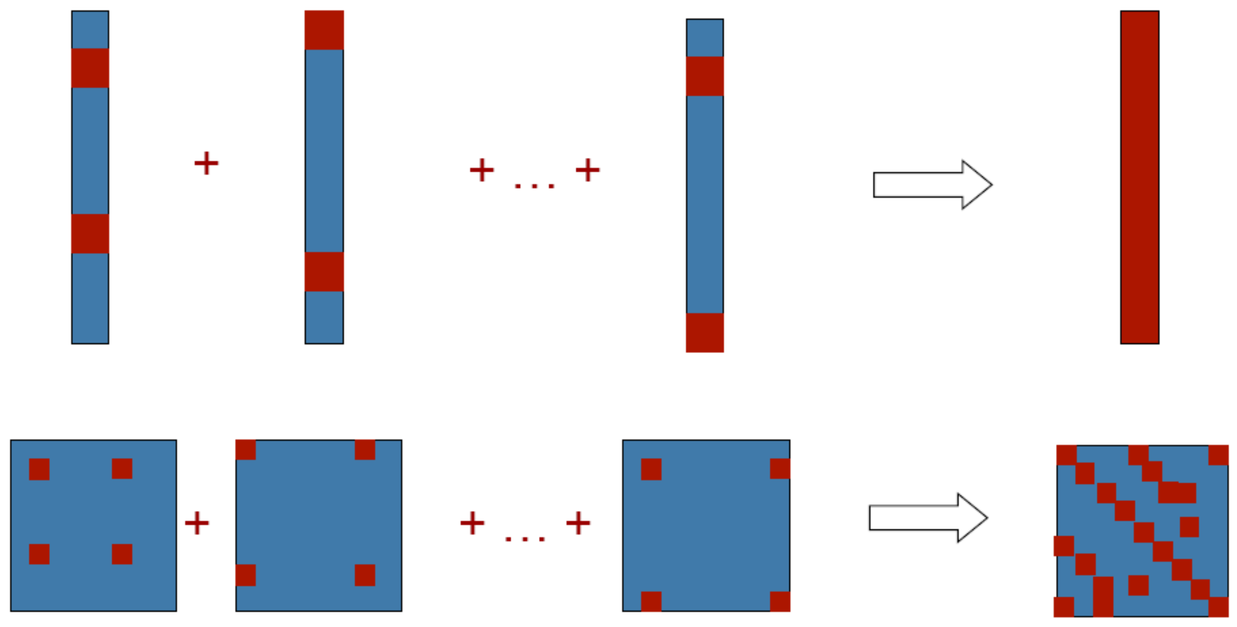
where the first term corresponds to the position of s_i in the global decision vector x, and similarly for the second term. Likewise, if node i is a pose and node j is a landmark, then the Jacobian is

$$J_{ij} = \begin{bmatrix} 0 & \dots & 0 & \frac{\partial r_{ij}}{\partial s_i} & 0 & \dots & 0 & \frac{\partial r_{ij}}{\partial \ell_j} & 0 & \dots & 0 \end{bmatrix}$$

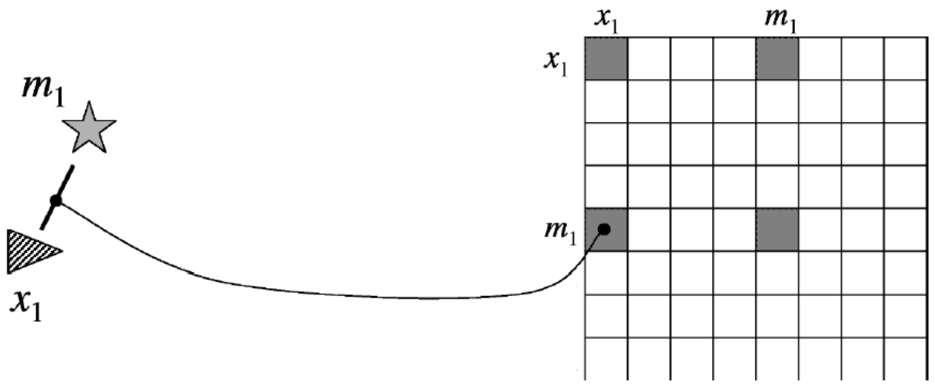
Due to this structure of the Jacobian, the information matrix and information vector for an edge connecting nodes i and j are only nonzero in the corresponding elements. In particular, the information is all zero except for the rows and columns corresponding to the i^{th} and j^{th} nodes,

$$\Omega_{ij} = \begin{bmatrix} \left(\frac{\partial r_{ij}}{\partial s_i}\right)^{\mathsf{T}} W_{ij} \frac{\partial r_{ij}}{\partial s_i} & \left(\frac{\partial r_{ij}}{\partial s_i}\right)^{\mathsf{T}} W_{ij} \frac{\partial r_{ij}}{\partial s_j} \\ \left(\frac{\partial r_{ij}}{\partial s_j}\right)^{\mathsf{T}} W_{ij} \frac{\partial r_{ij}}{\partial s_i} & \left(\frac{\partial r_{ij}}{\partial s_j}\right)^{\mathsf{T}} W_{ij} \frac{\partial r_{ij}}{\partial s_j} \end{bmatrix} \quad \text{and} \quad \xi_{ij} = \begin{bmatrix} \left(\frac{\partial r_{ij}}{\partial s_i}\right)^{\mathsf{T}} W_{ij} r_{ij} \\ \left(\frac{\partial r_{ij}}{\partial s_j}\right)^{\mathsf{T}} W_{ij} r_{ij} \end{bmatrix}$$

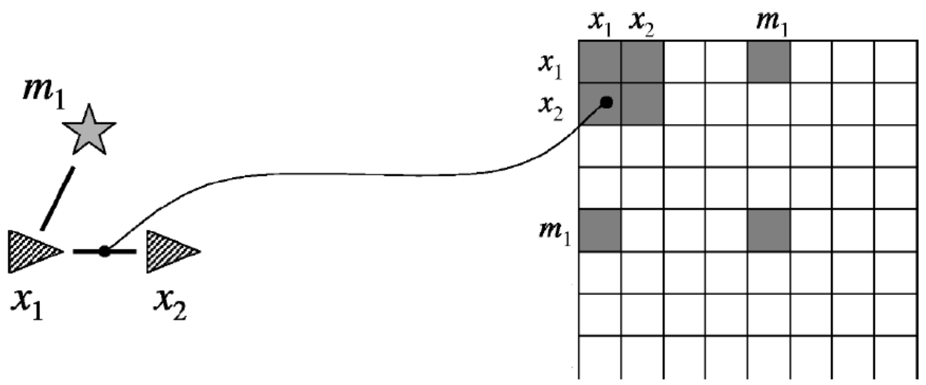
The structure of information is illustrated in the following figure. The top shows the information vector, and the bottom shows the information matrix. Each term in the sum represents the local information corresponding to a single edge, where red blocks indicate information between the two edges while blue indicates zero values with no information. Since information is additive, these individual pieces of information get summed to produce the total information vector and matrix. The information vector is in general full, since there is some information about every robot pose and landmark location. The information matrix, however, is sparse. This is due to the fact that the robot does not have information connecting every two nodes. Actuation connects consecutive poses, but not poses that are far away from each other. Likewise, the robot does not typically measure every landmark while at every pose. Also, the bottom right block of the information matrix is always diagonal, because measurements only connect landmark locations to robot poses, not landmarks to other landmarks.



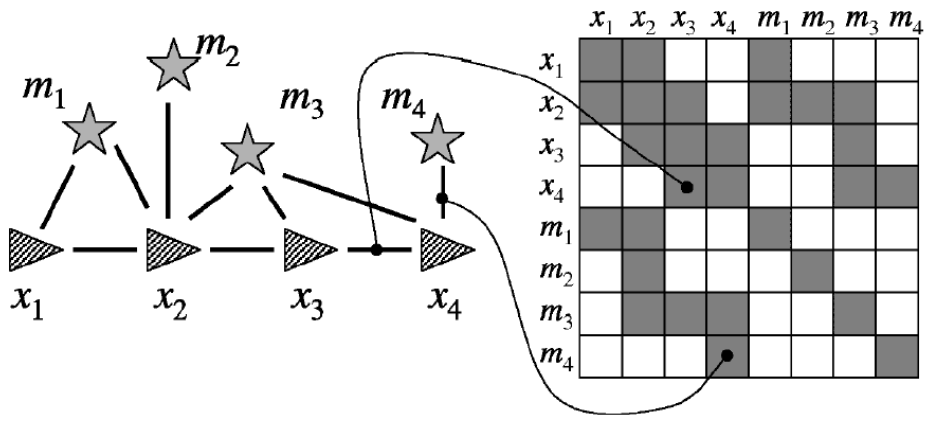
The following illustrates how the information matrix evolves over time as more edges are added to the graph. The first figure shows an edge connecting the robot pose x_1 with landmark m_1 and the resulting sparse information matrix. The only nonzero entries in the information matrix corresponding to this constraint are the entries between the pose x_1 and landmark m_1.



As the robot moves in the environment and observes measurements, it gains information that fills in the information matrix. If the robot moves to a new pose x_2, for instance, then there is now an actuation-based edge from pose x_1 to pose x_2, which results in terms connecting the two poses that are added to the information matrix.



After applying several control actions and making several measurements, the information matrix gets filled with the available information connecting the robot poses and landmark locations.



Part IV

Planning and Control

9

Planning and control

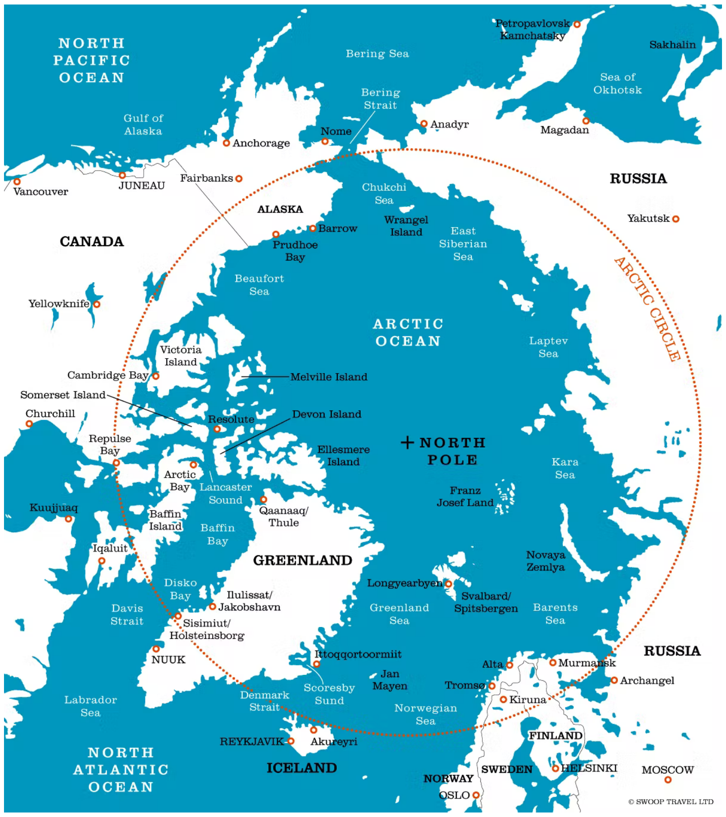
A robot must ultimately decide what actions to take. So far, we have studied robot perception, which is the problem of estimating certain quantities — such as the robot pose and map of the environment — from sensor measurements. We now consider how to use this information to choose appropriate actions.

9.1 Motivation

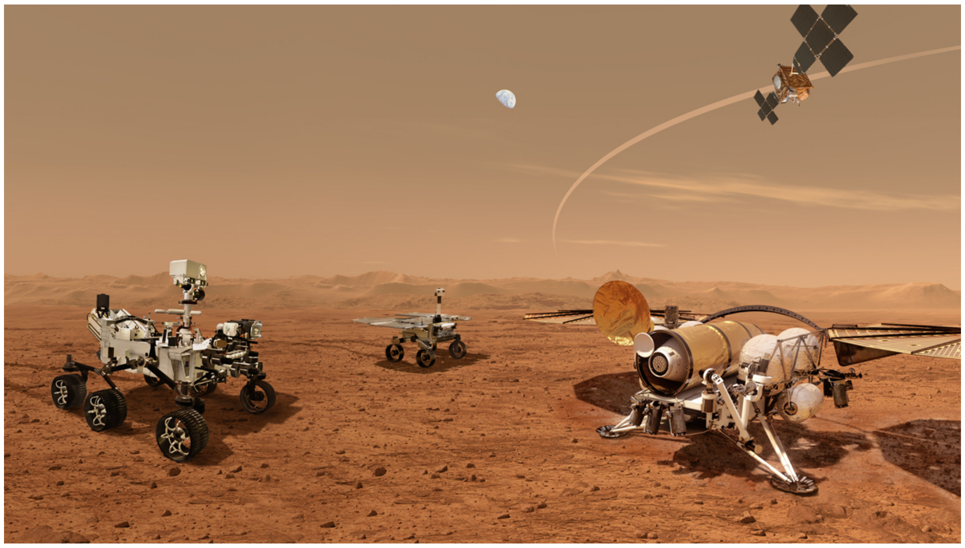
Planning and control must take into account uncertainty. Consider the following scenarios:

- A a robotic manipulator grasps and assembles parts arriving in random configuration on a conveyer belt. The configuration of a part is unknown at the time it arrives, yet the optimal manipulation strategy requires knowledge of the configuration. How can a robot manipulate such pieces? Will it be necessary to sense? If so, are all sensing strategies equally good? Are there manipulation strategies that result in a well-defined configuration without sensing?

- An underwater vehicle shall travel from Canada to the Barents Sea. Shall it take the shortest route through the North Pole, running risk of loosing orientation under the ice? Or sould it take the longer route through open waters, where it can regularly localize using GPS, the satellite-based global positioning system? To what extent do such decisions depend on the accuracy of the submarine's inertial sensors?

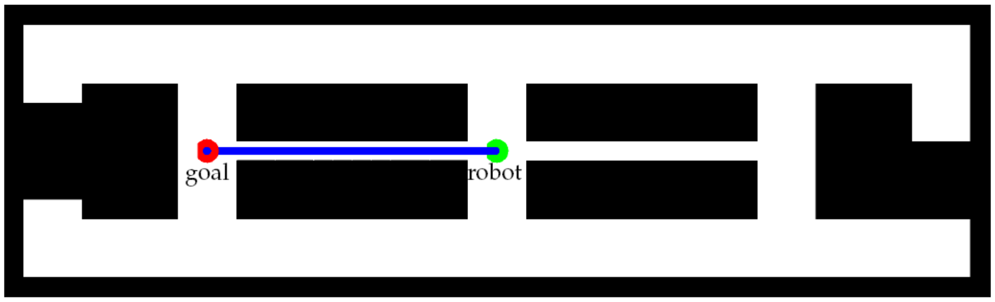


- A team of robots explores an unknown planet, seeking to acquire a joint map. Shall the robots seek each other to determine their relative location to each other? Or shall they instead avoid each other so that they can cover more unknown terrain in shorter amounts of time? How does the optimal exploration strategy change when the relative starting locations of the robots are unknown?

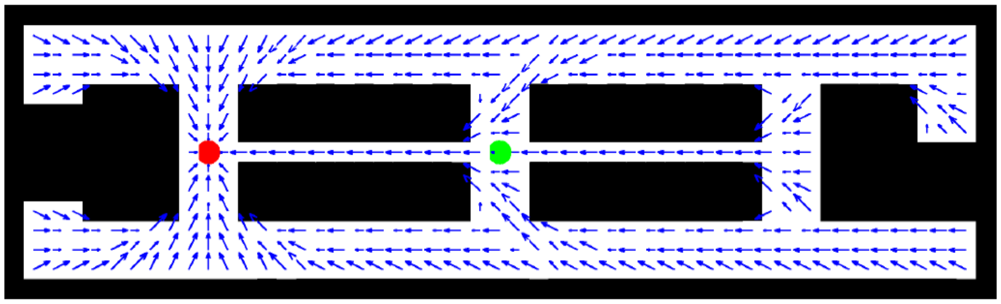


9.2 Example

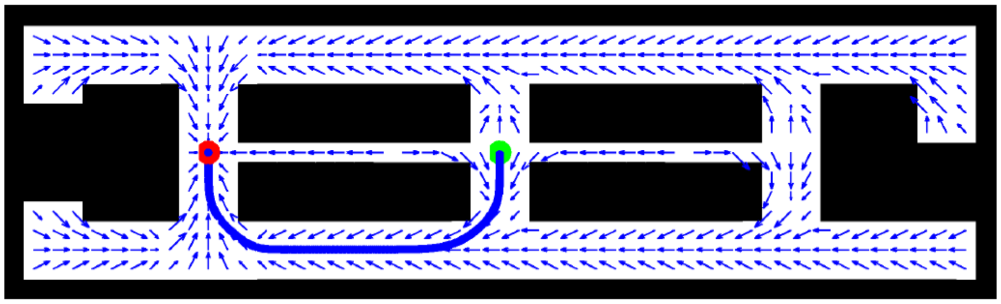
To illustrate the effects of uncertainty on motion planning, consider the following simple scenario in which a robot wants to move from a starting location to a goal location. If control actions are deterministic, then the robot does not need to sense its environment but can simply move straight through the narrow corridor to the goal location.



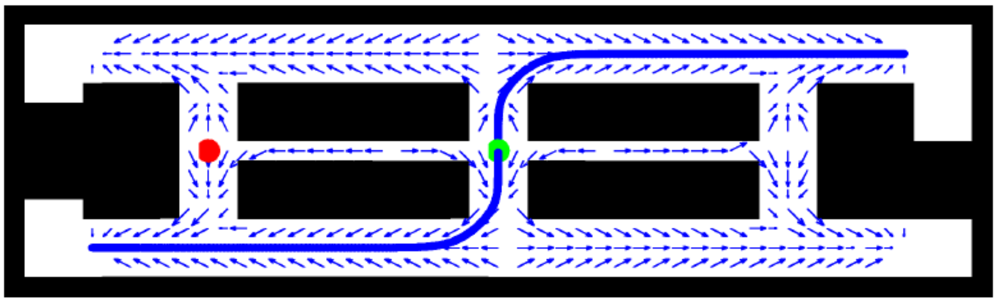
The previous trajectory indicated only how to get from the starting position to the goal position. We can also construct a plan that indicates how to get from *any* starting position to the goal position. This is illustrated as follows, where the blue arrows indicate the direction the robot should move from that position.



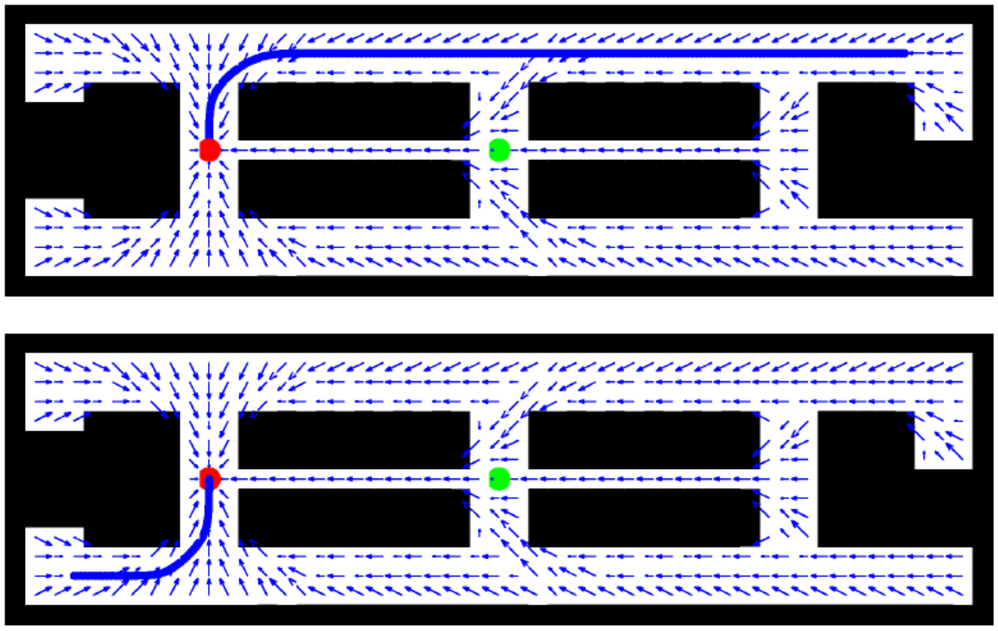
Both of the previous scenarios assumed that the robot knows its full state exactly and can apply deterministic control actions. In a slightly more realistic scenario, now suppose that control actions are non-deterministic, so there is some uncertainty about how a given control action will change the state of the robot. In this case, it is not enough to have a single desired trajectory from start to goal since the non-deterministic control actions may move the robot off the desired trajectory. We now need to specify what the robot should do no matter what state it is in. The optimal plan is indicated below, where it is now safer for the robot to take the longer path through the wider corridor to the goal location.



We now relax the assumption that the robot knows its exact position. Suppose the robot only knows its starting location but not orientation, and it has no sensor to detect that it has arrived at the goal location. The robot must now rely on sensor measurements to estimate its location within the map. Due to the symmetry of the environment, there is a large amount of uncertainty in the robot's precise location until it sees either of the ends of the environment from which it can determine its location. The following control policy may be used to move the robot from any state to the nearest corner so that it can localize itself in the map.



Once the robot knows its pose within the map, it can then apply a different control policy to navigate to the goal location. The following figure illustrates two possible trajectories, depending on which corner it used for localization.

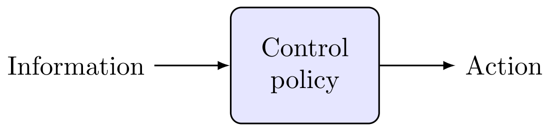


Some key aspects from this example:

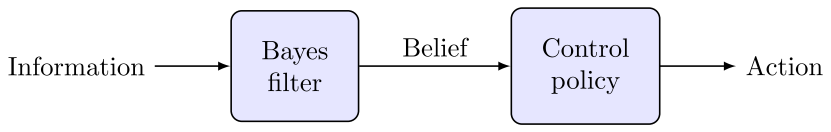
- The robot must actively gather information, and to do so it may have to take a suboptimal route (compared to a robot that has full information).
- The optimal plan depends on the amount of uncertainty in both actuation and perception.

9.3 Separation and controller structure

Robots must decide how to choose actions in the face of uncertainty. A control policy specifies what action the robot should take given its current information. The general structure of a control policy is as follows.

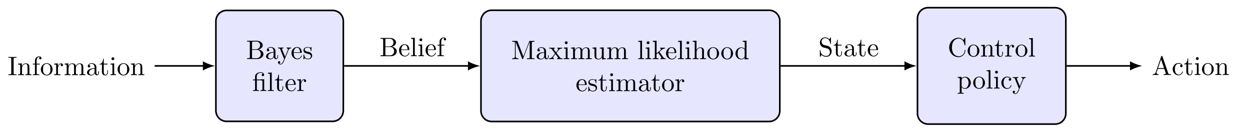


Here, the control policy depends on *all* known information, so the robot plans in *information space*. As the robot gains information, the policy must determine how to use this new information as well as all of the previous information to choose an action. Needing to store all known information can be computationally challenging for robots with a finite amount of memory. Instead, we could split the control policy into two components: estimation and control. This structure is shown below, were a Bayes filter is used to estimate the belief, and the control policy chooses actions based on this belief.



Here, the robot plans under uncertainty by generating plans in *belief space*. The belief space is the space of all possible beliefs that the robot may have. A control policy then maps any belief to an appropriate control action. In this way, the control policy chooses the best action based on the robot's current belief.

As we have seen, the belief is often a high-dimensional probability distribution (in fact, infinite dimensional for continuous state spaces). To simplify the design of the control policy, another possible controller structure is to choose actions based only on the most likely state. This structure is shown below, where the maximum likelihood estimator chooses the state with the highest probability in the belief.



As we impose more structure on the control policy, designing the controller becomes more simple, but the performance that the controller achieves may diminish. For instance, choosing actions based on the most-likely state enables the use of simple control policies that map states to control actions, but is highly suboptimal in many scenarios. For instance, such a controller will never choose to explore the environment to gain more information. To illustrate this idea, let's go back to the previous example. In that case, there are only two different beliefs: either the robot has localized itself and therefore knows the goal location, or it has no clue. This leads to the two distinct control policies shown, one for each belief. While there are only two beliefs in this simple example, more realistic scenarios often have a large number of beliefs.

The belief space typically has high dimensionality. This is because there are typically many more beliefs than states. For the simplest case in which the state space is discrete and finite, the belief space is usually continuous with finitely many dimensions. As we have seen before in estimation, working with continuous spaces is often problematic. It is even worse when the state space is continuous, since the belief is then usually continuous with infinitely many dimensions!

10

Discrete Control

We first consider the simplest scenario in planning and control in which the state space is discrete, the robot knows the state exactly, and the control actions are deterministic. This formulation is also known as graph search, which is a mathematical framework that models decision making over graphs.

10.1 Problem setup

The components of the discrete control problem are as follows:

- A discrete state space S, where each element s of S is a state.
- For each state s, an action space $A(s)$ of possible actions a available from state s.
- A state transition function f that constructs the next state s' given that the robot was in state s and applied action a according to the state transition equation $s' = f(s, a)$.
- A stage cost $\ell(s, a)$ associated with applying action a from state s.
- An initial state s_I.
- A set of goal states S_G.

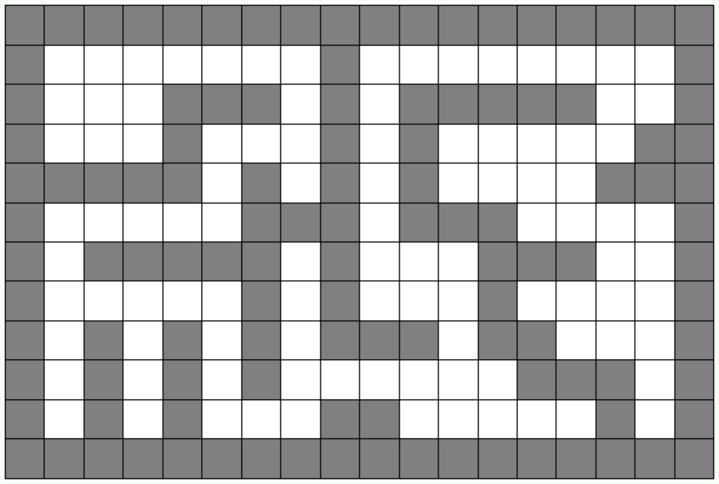
Since everything is deterministic, we only need to find a single trajectory from the initial state to the goal set (as opposed to a function that specifies what action to take from any state). In the terminology of control, we are finding an *open-loop* controller. Our goal is summarized as follows.

Choose a sequence of control actions that minimize the cumulative cost over the trajectory from the initial state to a goal state.

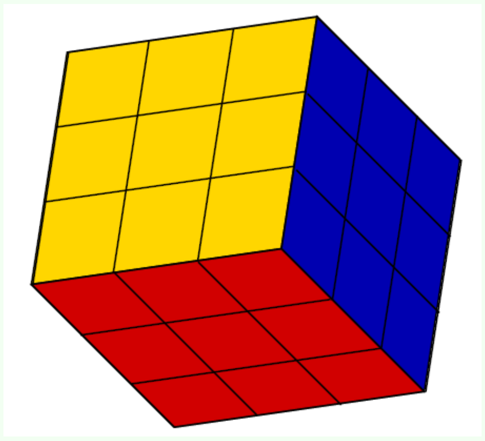
More precisely, we seek to find a sequence of actions $a_0, a_1, \ldots, a_{K-1}$ and states $s_0, s_1, \ldots, s_K$ that move the system from the initial state to a goal state with minimum cumulative cost:

$$\begin{aligned}
&\text{minimize} && \sum_{k=0}^{K-1} \ell(s_k, a_k) \\
&\text{subject to} && s_{k+1} = f(s_k, a_k) \quad \text{for } k = 0, 1, \ldots, K-1 \\
&\quad a_k \in A(s_k) \quad \text{for } k = 0, 1, \ldots, K-1 \\
&\quad s_0 = s_I \\
&\quad s_K \in S_G
\end{aligned}$$

Example (Grid world)**.** A canonical example of discrete control is a robot moving in a grid world as shown below. Here, the domain is discretized into a grid in which dark cells are occupied (walls) and white cells are open (free space). The state space is the set of all white cells, the set of possible actions from a given state are the set of adjacent white cells, the state transition function is $f(s, a) = s + a$ in which the state s and action a are interpreted as two-dimensional integer vectors, and the stage cost is one to move to an adjacent cell and zero to remain stationary. For a particular initial and goal state, the objective is to find the shortest path through the maze.

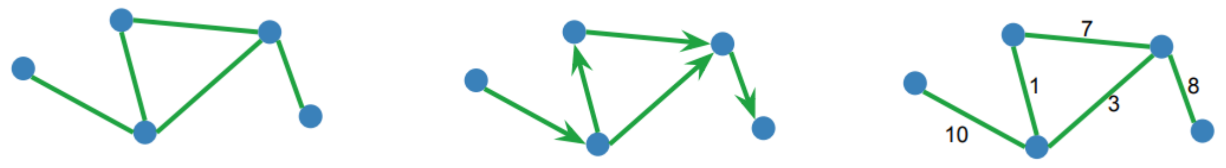


Example (Rubik's cube)**.** A Rubik's cube is an array of $3 \times 3 \times 3$ smaller cubes, where each face of the smaller cubes one one of six colors. The state space consists of all configurations of the cube, and an action consists of rotating a 3×3 slice by a quarter turn (so there are 12 possible actions from any state). The stage cost is one per move. Given an initial configuration, the goal is to find the shortest sequence of actions that returns in to the configuration in which each face is a single color (the goal state).

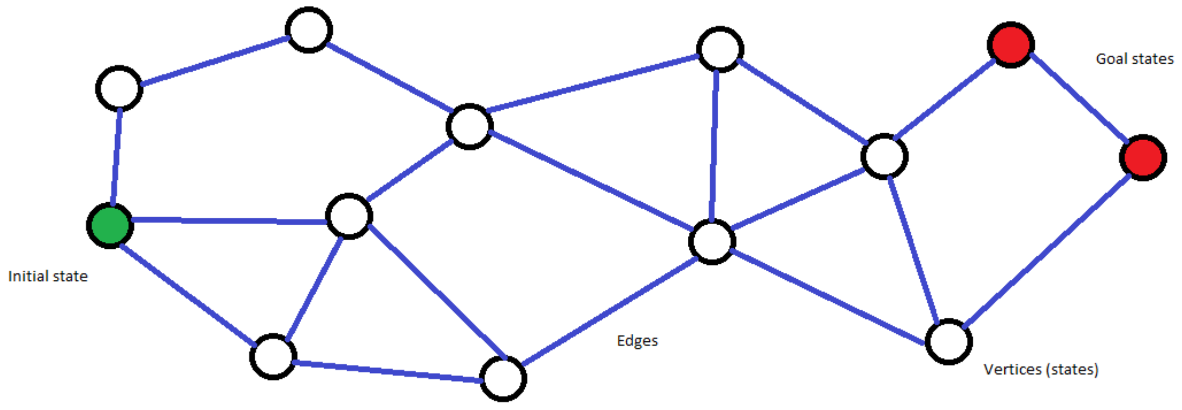


State transition graph

The discrete control problem can be posed in the context of searching over a graph. A graph is a mathematical object that consists of nodes (or vertices) and edges. Each edge connects two nodes together. A graph is undirected if all edges go both directions, that is, if there is an edge from node v to w, then there is also an edge from w to v. Otherwise, the graph is directed and edges point from one node to the other, which is drawn as an arrow. A weighted graph also associates a weight to each edge.



The state transition graph is a weighted directed graph whose nodes are states. There is a directed edge from state s to s' if and only if there exists an action $a \in A(s)$ such that $s' = f(s,a)$, in which case we say that s' is a neighbor of s. The weight of the edge is the cost $\ell(s,a)$ of applying action a from state s. Each state has a set of neighboring states. The initial and goal sets are designated as special nodes in the graph.



Edge costs for occupancy grid map. Suppose we are given an occupancy grid map, where each grid cell has an associated proabability of occupancy. One way to define the weight for an edge connecting nodes s and s' is the distance between the nodes (which is constant for an evenly-spaced grid) plus a constant multiple of the occupancy probability. For instance, the weight associated with an edge from s to s' by taking action a could be

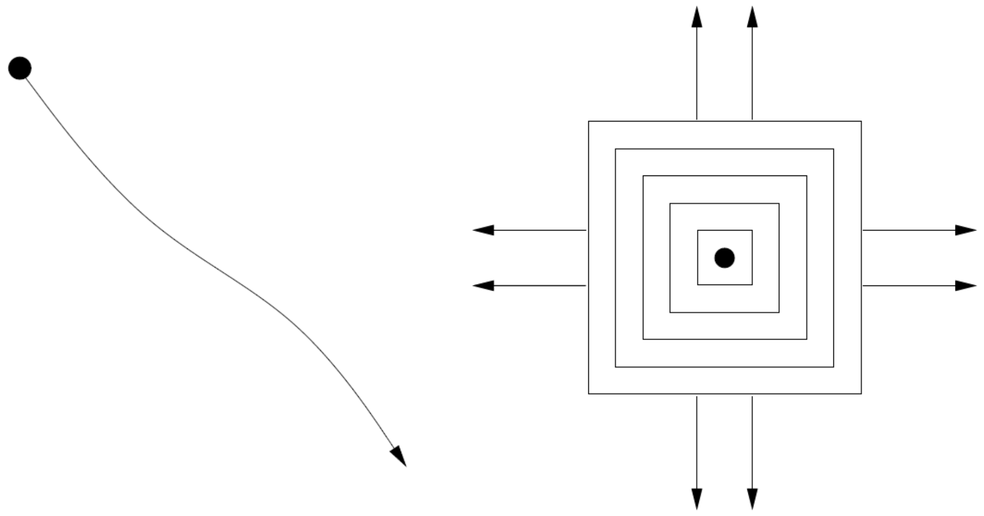
$$\ell(s,a) = \begin{cases} \|s - s'\| + \eta\, p(s') & \text{if } p(s') \leq 0.5 \\ \infty & \text{otherwise} \end{cases}$$

where $p(s')$ is the probability that grid cell s' is occupied and $\eta > 0$ is a positive constant. This will prohibit the trajectory from passing through grid cells that are most-likely occupied, and will penalize moving through grid cells with higher probability of occupancy.

Properties

An algorithm is a systematic search procedure that is used to find a solution to the discrete control problem. Algorithms can have various properties.

- **Systematic.** An algorithm is systematic if, for a finite graph, every vertex in the graph is eventually searched, so the algorithm will correctly determine whether or not the goal set is reachable in a finite amount of time. For countably infinite graphs, we cannot search all states in a finite amount of time. Instead, an algorithm is systematic if it detects a feasible path in finite time if one exists (and otherwise may run forever). This requires that the algorithm explores every reachable vertex in the limit as time goes to infinity. To be systematic, the algorithm should keep track of which states have been explored to avoid revisiting the same states over and over again.

  The difference between a systematic and non-systematic search is illustrated above. On the left, the algorithm only searches in a single direction, which may prevent the algorithm from finding the goal state even when it is close by. On the right, the search expands in wavefronts to ensure eventually finding a goal state.

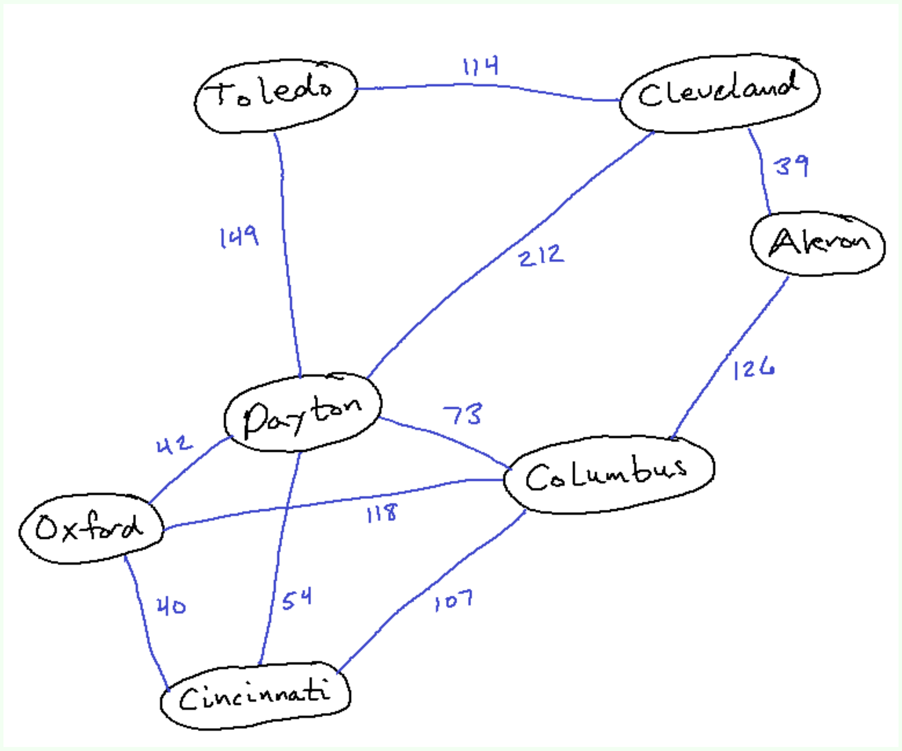
- **Optimal.** An algorithm is optimal if it finds a sequence of control actions to transfer the system from the initial state to a goal state with minimal cumulative cost, which is the sum of the stage costs over the trajectory.

Cumulative costs

Many algorithms use two cumulative costs to guide the search for a trajectory from the initial to goal states. Each of the following cumulative costs is a function that maps states to real numbers.

- **Cost-to-come.** For each state s, the cost-to-come $C(s)$ is the sum of the stage costs over a trajectory from the initial state to s. In general, this depends on the trajectory taken from the initial state to s. When this is the optimal route (in that it has the minimal cost-to-come), we denote it as $C^*(s)$.
- **Cost-to-go.** For each state s, the cost-to-go $G(s)$ is the sum of the stage costs over a trajectory from state s to a goal state. In general, this depends on the trajectory taken from state s to a goal state. When this is the optimal route (in that it has the minimal cost-to-go), we denote it as $G^*(s)$.

Example (Traveling in Ohio)**.** To illustrate the problem setup, consider traveling between major cities in Ohio. In this scenario nodes in the graph are cities, and edges connecting two cities indicate the distance in miles to drive between the cities. The edges are undirected since we can always travel either direction between two cities. We could have used other costs as well, such as the amount of time that it takes to travel between two cities or the amount of emissions produced by the travel.



Suppose that the initial node is Oxford. The cost-to-come to Cleveland if we first travel to Columbus and then through Akron is the sum of the costs along the route:

$$C(\text{Cleveland}) = 118 + 126 + 39 = 283$$

The optimal cost-to-come, however, is obtained by taking the route through Dayton, in which case the optimal cost-to-come is

$$C^*(\text{Cleveland}) = 42 + 212 = 254$$

Now suppose that Toledo is the goal node. The cost-to-go from Cincinnati traveling through Oxford and then Dayton is

$$G(\text{Toledo}) = 40 + 42 + 149 = 231$$

The optimal cost-to-go is obtained by traveling straight from Cincinnati to Dayton,

$$G^*(\text{Toledo}) = 54 + 149 = 203$$

10.2 General forward search

One way to explore the state space is to start from the initial state and work outward until a goal state is found. This technique is called forward search, and there are many instances of this general idea. At each point in time, there are three kinds of states:

- **Unvisited:** States that have not yet been explored.
- **Alive:** States that have been visited and have unvisited neighbors.
- **Dead:** States that have been visited and for which every neighbor has also been visited. Such states have been completely explored and cannot contribute any more information to the search.

These sets of states form a partition of the state space in that every state is always in exactly one category.

$$S \quad = \quad \text{unvisited} \quad \oplus \quad \text{alive} \quad \oplus \quad \text{dead}$$

Initially, all states are unvisited. To search for a path from the initial state to the set of goal states, the first step is to explore the initial state, in which case it becomes alive. The set of alive states is stored in a priority queue, Q. These are the set of states that we still need to explore. We then loop until all states are dead (in which case the queue is empty) and select the first state from the queue according to its priority function. If this state is in the goal set, then we have succeeded in finding a path from the initial state to a goal state so we return success. Otherwise, we add all unvisited neighbors of the current state to the queue and mark them as visited. This general algorithm is described below.

Algorithm General forward search

```
1: Insert $s_I$ to $Q$ and mark $s_I$ as visited                    ▷ States in $Q$ are alive
2: while $Q$ is not empty do
3:     Let $s$ be the first state in $Q$                               ▷ Explore state $s$
4:     if $s \in S_G$ then
5:         return SUCCESS
6:     for all $a \in A(s)$ do
7:         $s' \leftarrow f(s, a)$
8:         Set state $s$ with action $a$ as the parent of $s'$         ▷ Needed to reconstruct the path
9:         if $s'$ is unvisited then
10:            Insert $s'$ to $Q$ and mark $s'$ as visited
11:        else
12:            Resolve duplicate $s'$
13: return FAILURE
```

States that have never been added to the queue are unvisited, states that are in the queue are alive, and after states are removed from the queue they are visited but not alive and therefore dead.

The *frontier* is the set of states that we have seen but have not yet explored, which are the alive states (those that are in the queue). For each particular implementation, it will be insighful to see how the frontier expands as the algorithm searches the state space.

In general, it can be quite costly to check whether or not a state has been visited since this requires looping over both alive and dead states.

If the algorithm terminates with success, then we can obtain the sequence of control actions that transfer the system from the initial state to the goal set as follows. Start with the goal state that caused the algorithm to terminate. In line 8 of the algorithm, we marked its parent as another state s and action a, meaning that we need to apply action a from state s to move to the goal state. Now lookup the parent of state s to find its parent and the corresponding action. Continuing to do so, we obtain the sequence of states and actions that lead from the initial state to a goal state.

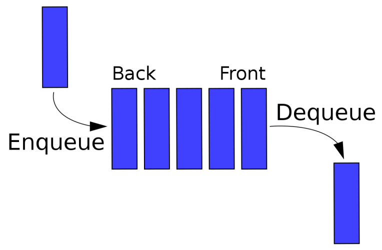
Line 12 resolves duplicate states s' that have already been visited. For instance, when a state is reached multiple times, the algorithm may update the cost function used to sort the priority queue.

10.3 Particular forward search algorithms

While the previous section described the general structure of forward search, we now describe several specific implementations of this general procedure. Each algorithm is obtained by defining a particular priority function for the queue.

Breadth-first search

Breath-first search (BFS) uses a First-In-First-Out (FIFO) queue. The priority function for this queue prioritizes states based on when they arrived at the queue, with the first state to arrive having the highest priority. This causes the frontier to expand uniformly away from the initial state, which is why the algorithm is called breadth-first search. Since the frontier expands outward from the initial state, we do not need to do anything to resolve duplicate states.



Properties

- The first feasible trajectory to the goal found by breadth-first search uses the smallest number of steps. This is because all plans that have k steps are exhausted before investigating plans with $k+1$ steps. This trajectory may not be optimal, however, since the smallest number of state transitions does not necessarily have the smallest cost (it is optimal if all edge weights are equal).
- Since the frontier expands uniformly from the initial state, breadth-first search is systematic.
- The asymptotic running time of BFS is $O(|V| + |E|)$, where $|V|$ is the number of vertices (or states) and $|E|$ is the number of edges (or state transitions).

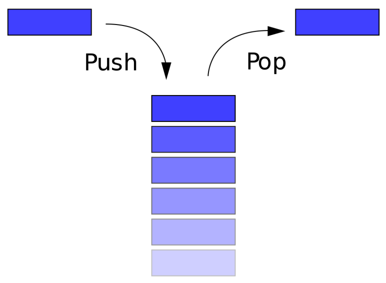
Example (BFS on traveling Ohio). To illustrate breadth-first search, consider applying it to the traveling Ohio example. Suppose that we start in Oxford and want to travel to Akron. The algorithm keeps track of states that are visited and alive. We assume that nodes are added to the queue in alphabetical order. The explored node and sets of visited and alive nodes at each iteration are as follows.

Explored	Visited	Alive
	Oxford	Oxford
Oxford	Oxford, Cincinnati, Columbus, Dayton	Cincinnati, Columbus, Dayton
Cincinnati	Oxford, Cincinnati, Columbus, Dayton	Columbus, Dayton
Columbus	Oxford, Cincinnati, Columbus, Dayton, Akron	Dayton, Akron
Dayton	Oxford, Cincinnati, Columbus, Dayton, Akron, Cleveland, Toledo	Akron, Cleveland, Toledo
Akron	Oxford, Cincinnati, Columbus, Dayton, Akron, Cleveland, Toledo	Cleveland, Toledo

At this point, Akron has been expanded, so we have found a path from Oxford to Akron and the algorithm terminates.

Depth-first search

Depth-first search (DFS) uses a Last-In-First-Out (LIFO) queue — or stack — in which the last state to enter the queue is the first to be expanded. This results in an aggressive exploration of the state space that dives quickly into the graph, in contrast to the uniform expansion of BFS. As before, there is nothing extra to do to resolve duplicate states.



Properties

- Depth-first search is suboptimal in that the first feasible trajectory to the goal set may not use the smallest number of steps or have the smallest cost.
- Depth-first search is systematic for finite state spaces but not for countably infinite state spaces since it may expand a single direction and never search large portions of the state space, even in the limit as the number of iterations tends to infinity.
- The asymptotic running time of depth-first search is also $O(|V| + |E|)$.

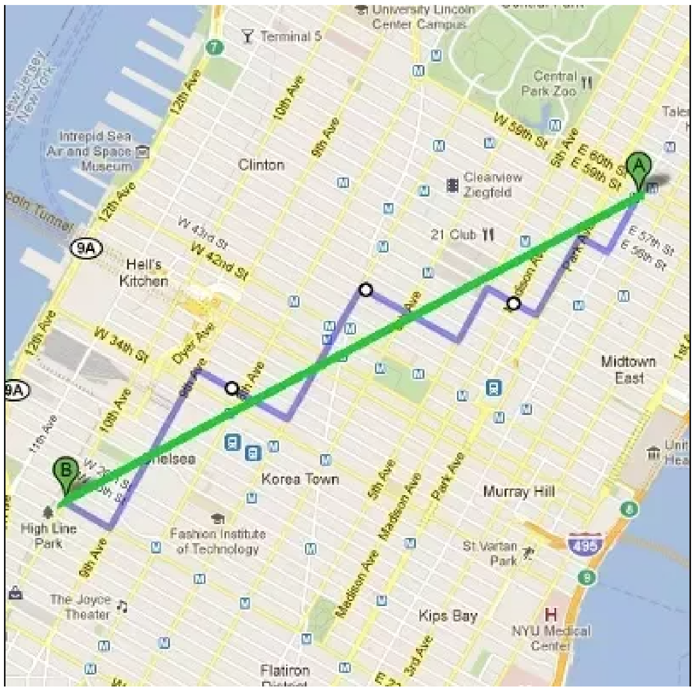
Example (DFS on traveling Ohio). To illustrate depth-first search, consider applying it to the traveling Ohio example. Suppose that we start in Oxford and want to travel to Akron. The algorithm keeps track of states that are visited and alive. We assume that nodes are added to the queue in alphabetical order. The expanded node and sets of visited and alive nodes at each iteration are as follows.

Expanded	Visited	Alive
	Oxford	Oxford
Oxford	Oxford, Cinci, Columbus, Dayton	Cinci, Columbus, Dayton
Dayton	Oxford, Cinci, Columbus, Dayton, Cleveland, Toledo	Cinci, Columbus, Cleveland, Toledo
Toledo	Oxford, Cinci, Columbus, Dayton, Cleveland, Toledo	Cinci, Columbus, Cleveland
Cleveland	Oxford, Cinci, Columbus, Dayton, Cleveland, Toledo, Akron	Cinci, Columbus, Akron
Akron	Oxford, Cinci, Columbus, Dayton, Cleveland, Toledo, Akron	Cinci, Columbus

At this point, Akron has been expanded, so we have found a path from Oxford to Akron and the algorithm terminates.

Greedy best-first search

Instead of using a first-in-first-out or last-in-first-out queue, greedy best-first search uses a *heuristic* as the priority function used to select the next state to explore. A heuristic is an approximate measure of how close a state is to the goal, which is used to guide the algorithm in the right direction. Ideally, the heuristic would be the true distance from the state to the goal (the cost-to-go), but we do not know this ahead of time since it is what we are trying to compute! Instead, a heuristic should approximate the optimal cost-to-go while being easy to compute. Some examples of heuristics in a two-dimensional environment are the Euclidean and Manhattan distances.



Greedy best-first search is a forward search algorithm in which the priority function used to sort the queue is the heuristic.

Properties

- Greedy best-first search is suboptimal in that the first feasible trajectory to the goal set will not necessarily use the smallest number of steps.
- Greedy best-first search is not systematic, since it may leave large portions of the state space unexplored in infinite state spaces.
- The asymptotic running time of greedy best-first search is also $O(|V| + |E|)$.

Dijkstra's algorithm

Dijkstra's algorithm uses the best available cost-to-come as the priority function for the queue. The cost-to-come is computed incrementally during the execution of the algorithm and is the best known cost-to-come given the nodes that have been explored. The cost-to-come is updated when a better cost-to-come is found. Initially, $C(s_I) = 0$ since there is no cost to move from the initial state to itself, and all other states have an infinite cost-to-come. When the algorithm explores a state s' from a previous state s with action a, we have found a path to s' through s. We do not know, however, that we arrived at s' in the optimal way (there may be a better route that we have not yet explored). If the newly explored state s' already exists in the queue, then this new path *may* be more efficient than the previous path, or it may not. When resolving duplicate states, if the cost-to-come using the new path through node s to s' is less than the previously stored cost-to-come for node s', then the algorithm updates the cost-to-come and also updates the parent of node s' as state s with action a. In particular, the cost-to-come is updated as the minimum of the current cost-to-come (which is infinite if a path has not yet been found to the state) and the cost-to-come through state s (which is the cost-to-come to state s plus the cost of transitioning from state s to state s' by applying action a),

$$C(s') \quad \leftarrow \quad \min\{C(s'),\ C(s) + \ell(s,a)\}$$

Once a state s is removed from the queue, it is then dead and therefore cannot be reached at a lower cost. At this point, its value $C(s)$ is the optimal cost-to-come, $C^*(s)$.

Properties

- Dijkstra's algorithm is optimal in that it always find the trajectory from the initial state to the set of goal states with minimal cumulative cost.
- Dijkstra's algorithm is systematic.
- When the priority queue is implemented using a Fibonacci heap, the asymptotic running time of Dijkstra's algorithm is $O(|V| \log |V| + |E|)$.
- If all edge costs are equal, then Dijkstra's algorithm is equivalent to breadth-first search.

In addition to finding the optimal trajectory to the goal set, Dijkstra's algorithm finds the optimal trajectory from the initial node to *all* other nodes.

A*

While Dijkstra's algorithm is optimal, it may be slow since it places no priority on searching for the goal (it simply terminates when the goal is found). The algorithm A^* (pronounced "A-star") combines the benefits of Dijkstra's algorithm and greedy best-first search by using a heuristic to guide its search toward the goal while remaining optimal.

A^* is exactly like Dijkstra's algorithm, except that the priority function used to sort the queue is the cost-to-come plus a heuristic for the cost-to-go,

$$C(s) + h(s)$$

If the heuristic $h(s)$ underestimates the optimal cost-to-go $G^*(s)$ (which is unknown), then A^* is optimal. We can typically find a heuristic by relaxing some of the problem constraints, as this leads to a lowerbound on the optimal cost-to-go. For navigating in a two-dimensional space, for instance, simple heuristics are the Euclidean and Manhattan distances, which ignore the constraints caused by obstacles.

It is useful to consider the two extreme cases:

- If the heuristic is the optimal cost-to-go, then A^* will directly take the optimal path to the goal. This is the best scenario, but it is unrealistic since we do not know the optimal cost-to-go (or else the problem would be solved).
- If the heuristic is zero, then A^* is equivalent to Dijkstra's algorithm, which is still optimal but makes no preference in searching towards the goal and is therefore slow.

Properties

- A^* is both optimal and systematic if the heuristic underestimates the optimal cost-to-go, meaning that $0 \leq h(s) \leq G^*(s)$ for all states s.
- When the priority queue is implemented using a Fibonacci heap, the worst-case asymptotic running time of $A^\star$ is $O(|V| \log |V| + |E|)$.

The following table summarizes the forward search algorithms and their properties.

Algorithm	optimal	systematic	complexity	priority function
Breadth-first search	no[1]	yes	$O(\|V\| + \|E\|)$	arrival time (first-in-first-out)
Depth-first search	no	no	$O(\|V\| + \|E\|)$	arrival time (last-in-first-out)
Greedy best-first search	no	no	$O(\|V\| + \|E\|)$	$h(s)$
Dijkstra's algorithm	yes	yes	$O(\|V\| \log \|V\| + \|E\|)$	$C(s)$
A^* search	yes[2]	yes[2]	$O(\|V\| \log \|V\| + \|E\|)$	$C(s) + h(s)$

[1] BFS is optimal in terms of the number of state transitions (steps), but not necessarily the cost
[2] A^* is both optimal and systematic if the heuristic underestimates the optimal cost-to-go

10.4 Value iteration

Value iteration iteratively computes the cost-to-go — also known as the *value function* — from each state over trajectories of increasing length. The previous algorithms only expanded a single state at each iteration and iteratively expanded the search space. In contrast, value iteration updates the cost-to-go for all states at each iteration.

Let G_k^* denote the optimal cost-to-go over trajectories of length k. Our goal is then to compute G_∞^*, the optimal cost-to-go over trajectories of any length. Value iteration iteratively computes G_k^*. The cost-to-go is suboptimal until the number of iterations k is equal to the length of the optimal trajectory (if it exists). At this point the cost-to-go cannot be improved from any state and is therefore stationary, meaning that $G_{k+1}^* = G_k^*$. If this occurs, then the algorithm has found the optimal cost-to-go.

To start, let's consider the (very) simple case of planning over a trajectory of length zero. There are no actions to take, so the optimal cost-to-go over trajectories of length zero is zero for states in the goal set and infinite otherwise,

$$G_0^*(s) = \begin{cases} 0 & \text{if } s \in S_G \\ \infty & \text{otherwise} \end{cases}$$

Let's now construct the optimal cost-to-go over trajectories of length one. For a state s for which there exists an action $a \in A(s)$ such that $s' = f(s,a)$ is in the goal set, the cost-to-go is the stage cost $\ell(s,a)$. If there does not exist such an action, then the cost-to-go is infinite.

$$G_1^*(s) = \begin{cases} \ell(s,a) & \text{if there exists } a \in A(s) \text{ such that } f(s,a) \in S_G \\ \infty & \text{otherwise} \end{cases}$$

We can express this in terms of the optimal cost-to-go over zero-length trajectories as

$$G_1^*(s) = \min_{s \in A(s)} \; \ell(s,a) + G_0^*(f(s,a))$$

This says that the optimal cost-to-go from state s is the minimum of the optimal cost-to-go over all neighboring states s' plus the cost to get from state s to s'. Likewise, for each state s, the optimal cost-to-go for trajectories of length two is the minimum of the optimal cost-to-go for trajectories of length one over all neighboring states plus the cost to get to the neighbor. Continuing this recursion, the optimal cost-to-go over trajectories of length $k+1$ can be written in terms of the optimal cost-to-go over trajectories of length k as

$$G_{k+1}^*(s) = \min_{a \in A(s)} \; \ell(s,a) + G_k^*(f(s,a))$$

Since the cost-to-go is a function of the state, this formula must be used to compute $G_{k+1}^*(s)$ for all s. Value iteration uses this formula to recursively compute the cost-to-go along trajectories of increasing length.

Termination. If the stage costs are all nonnegative, then the optimal cost-to-go over trajectories of length k becomes stationary after a finite number of iterations. At this point, G_k^* is the optimal cost-to-go G^* and the algorithm terminates. We can construct an upper bound on the number of iterations as follows. For every state s, there either exists a trajectory that reaches the goal with finite cost or there is no solution. For each state for which there exists a plan that reaches the goal, consider the number of stages in the optimal trajectory. The maximum number of stages taken from all states that can reach the goal is an upper bound on the number of iterations before the cost-to-go becomes stationary.

Principle of optimality. Value iteration is based on the principle of optimality. This general principle states that *portions of optimal plans are also optimal.* To see this, consider an optimal trajectory $s_1, s_2, \dots, s_n$ from state s_1 to s_n. For any i and j, the portion of the trajectory from s_i to s_j must also be the optimal path between s_i and s_j. This general principle allows us to break up the problem of finding an optimal trajectory into smaller subproblems of finding optimal trajectories between closer states. The value iteration recursion states that the optimal cost-to-go from state s is the optimal cost-to-go from state s' plus the cost to get from s to s' (by applying action a).

Steady state. If there exists a trajectory from the initial state to a goal state, then G_k^* becomes stationary after a finite number of iterations and is equal to the optimal cost-to-go G^*. In this case, the optimal cost-to-go satisfies the *Bellman equation*

$$G^*(s) = \min_{a \in A(s)} \; \ell(s,a) + G^*(f(s,a))$$

This is a functional equation whose variable is the optimal cost-to-go G^* (which is a function of the state).

Optimal controller. Value iteration constructs the optimal cost-to-go, but does not directly construct the optimal trajectory. We can recover the optimal action (from any state) using the optimal cost-to-go. For each state s, the optimal action a is the action that minimizes the immediate cost $\ell(s, a)$ of taking action a from state s plus the optimal cost-to-go from the next state $f(s, a)$,

$$\arg\min_{a \in A(s)} \; \ell(s,a) + G^*\big(f(s,a)\big)$$

This is a state feedback controller in that it chooses an action based on the state.

Value iteration consists of the following iterations:

$$G^*_{k+1}(s) = \min_{a \in A(s)} \; \ell(s,a) + G^*_k(f(s,a)), \qquad G^*_0(s) = \begin{cases} 0 & \text{if } s \in S_G \\ \infty & \text{otherwise} \end{cases}$$

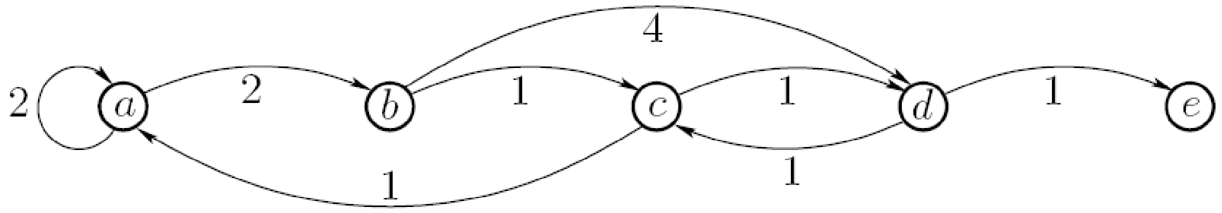
- G^*_k is the optimal cost-to-go over trajectories of length k.
- If there exists a trajectory from the initial state to a goal state, then value iteration converges after a finite number of iterations, in which case G^*_k is the optimal cost-to-go G^*.
- The optimal cost-to-go satisfies the stationary Bellman equation

$$G^*(s) = \min_{a \in A(s)} \; \ell(s,a) + G^*(f(s,a))$$

- The optimal state-feedback controller chooses the action $a \in A(s)$ that achieves the minimum in the Bellman equation.
- Value iteration is optimal, systematic, and has $O(|S|\,|A|)$ time complexity.

Example

Consider the following five-state graph, where $s_I = a$ and $S_G = \{d\}$.



The cost-to-go at each iteration of the algorithm is shown below. The cost-to-go is the same at iterations three and four, so at this point the algorithm terminates and this is the optimal cost-to-go.

	a	b	c	d	e
G^*_0	∞	∞	∞	0	∞
G^*_1	∞	4	1	0	∞
G^*_2	6	2	1	0	∞
G^*_3	4	2	1	0	∞
G^*_4	4	2	1	0	∞
G^*	4	2	1	0	∞

To recover the optimal actions, consider starting in state a. The action that leads to b is chosen since $2 + G^*(b) = 4$ is better than $2 + G^*(a) = 6$ (the 2 comes from the action cost). From state b, the optimal action leads to c, which produces a total cost $1 + G^*(c) = 1$. Similarly, the next action leads to d which is a goal state.

11

Markov Decision Processes

We now extend our discrete control problem formulation to include uncertainty in the control actions. We still assume that the state is known. This problem formulation is called a Markov decision process (MDP), which is a mathematical framework that models decision making under partial uncertainty.

11.1 Problem formulation

The components of a Markov decision process are as follows:

- A state space S, where each element s of S is a state.
- For each state s, an action space $A(s)$ of possible actions a available from state s.
- A state transition probability $p(s' \mid s, a)$ that describes the probability of being in state s' given that the robot was in state s and applied action a.
- A trajectory length T that may be finite or infinite.
- A stage reward $r(s, a)$ associated with applying action a from state s, and a terminal reward $r_T(s)$ associated with being in state s at the end of the trajectory (only applies to trajectories of finite length).
- A discount factor γ that indicates how much to discount future rewards (we need $\gamma < 1$ for infinite-length trajectories to ensure that the expected cumulative cost is finite, but allow $\gamma = 1$ for finite-length trajectories).

Since state transitions are now stochastic, we need to find a *controller* or *policy* that specifies what action to take from any state, which we denote as

$$\pi : S \to A(s)$$

For any state s, the controller specifies that the robot should take action $a = \pi(s)$. In the terminology of control, this is a *closed-loop* controller. Furthermore, the trajectory is now a stochastic process, and so the cumulative reward is a random process that depends on the particular state transitions. We therefore consider the *expected* cumulative reward over the trajectory.

Our goal is to find a controller that maximizes the expected cumulative reward over the trajectory from any initial state.

The reason for the terminal action and the discount factor is so that we can search over trajectories of finite or infinite length. When searching over trajectories of a finite length, we can use the terminal reward to reward the system for ending in certain states. For instance, we could set

$$r_T(s) = \begin{cases} \infty & \text{if } s \in S_G \\ 0 & \text{otherwise} \end{cases}$$

to reward being in a set of goal states S_G at the last iteration. When searching over trajectories of infinite length, we need a discount factor to ensure that the expected cumulative reward is finite (since it is an infinite sum). In this case the terminal reward has no effect.

The expected cumulative reward over a trajectory of length T from state s and following policy π is

$$V_T^\pi(s) = \mathcal{E}\left[\sum_{t=0}^{T-1} r(s_t, a_t) + r_T(s_T)\right] \qquad \text{or} \qquad V^\pi(s) = \mathcal{E}\left[\sum_{t=0}^{\infty} \gamma^t \, r(s_t, a_t)\right]$$

where the expectation is over the trajectory obtained from the initial state $s_0 = s$ and policy π.

Remark. Here we maximize reward while we previously minimized cost. These formulations are equivalent when the cost is the negative reward, $\ell(x, u) = -r(x, u)$. The field of reinforcement learning is optimistic and maximizes reward, while the field of controls is pessimistic and minimizes cost. ■

11.2 Value iteration

We previously studied various algorithms for the case in which state transitions are deterministic. Many of these algorithms (such as A^*), however, do not easily generalize to the stochastic setting. The algorithm that does naturally generalize is value iteration. As we will see, value iteration may also be used when the control actions are stochastic and the state cannot be directly measured.

Before, we used G_t to denote the cost-to-go over trajectories of length t. Since we are now maximizing reward, we use V_t for the value function, which is the expected cumulative reward over trajectories starting from time t, and we use V for the value function over the entire planning horizon. For example, the optimal value function from time t is

$$V_t^*(s) = \max_{a_t, \ldots, a_{T-1}} \mathcal{E}\left[\sum_{k=t}^{T-1} \gamma^k \, r(s_k, a_k) + r_T(s_T)\right]$$

where the expectation is taken over the state transitions $s_t, \ldots, s_T$ beginning from state $s_t = s$ and applying the sequence of actions $a_t, \ldots, a_{T-1}$. We want to compute the optimal value function over the entire planning horizon, which is V_0^*.

As before, let's consider planning over trajectories of length zero, that is, starting at the end of the planning horizon. There are no actions to take at this point, so the optimal value function from time T is just the terminal reward,

$$V_T^*(s) = r_T(s)$$

To construct the value iteration, let's write the optimal value function starting at time $t-1$,

$$V_{t-1}^*(s) = \max_{a_{t-1}, a_t, \ldots, a_T} \mathcal{E}\left[\sum_{k=t-1}^{T-1} \gamma^k \, r(s_k, a_k) + r_T(s_T)\right]$$

where $s_{t-1} = s$ and the expectation is over the sequence of states $s_t, \ldots, s_T$. We will see that we can construct this recursively in terms of the optimal value function from time t. To do so, we first separate the maximization over the first action a_{t-1} and the expectation over the next state s_t to obtain

$$V^*_{t-1}(s) = \max_{a_{t-1}} \max_{a_t,\dots,a_T} \mathcal{E}_{s_t}\mathcal{E}_{s_{t+1},\dots,s_T}\left[\sum_{k=t-1}^{T-1} \gamma^k\, r(s_k,a_k) + r_T(s_T)\right]$$

The actions $a_{t+1},\dots,a_T$ do not affect the state s_t, so we can swap the order of the maximization and expectation to obtain

$$V^*_{t-1}(s) = \max_{a_{t-1}} \mathcal{E}_{s_t}\left[\max_{a_t,\dots,a_T} \mathcal{E}_{s_{t+1},\dots,s_{T+1}}\left[\sum_{k=t-1}^{T-1} \gamma^k\, r(s_k,a_k) + r_T(s_T)\right]\right]$$

The first term $r(s_{t-1},a_{t-1})$ is now independent of the inner maximization and expectation, so we can pull it out.

$$V^*_{t-1}(s) = \max_{a_{t-1}} \mathcal{E}_{s_t}\left[r(s_t,a_t) + \gamma \max_{a_t,\dots,a_T} \mathcal{E}_{s_{t+1},\dots,s_{T+1}}\left[\sum_{k=t}^{T-1} \gamma^{k-1}\, r(s_k,a_k) + r_T(s_T)\right]\right]$$

The inner term is the maximum expected cumulative reward over a trajectory of length T starting from state s_1, which is precisely $V^*_T(s_1)$. Therefore, we have the recursion

$$V^*_{t-1}(s) = \max_{a_0} \mathcal{E}_{s_1}\left[r(s_0,a_0) + \gamma\, V^*_T(s_1)\right]$$

where the expectation is over the next state s_1 obtained from being in state $s = s_0$ and applying action a_0. To summarize, value iteration consists of the following recursion:

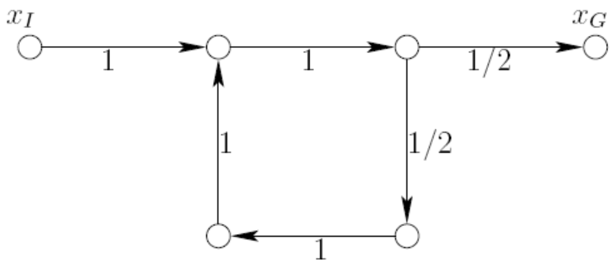
$$V^*_{t-1}(s) = \max_{a\in A(s)}\left\{r(s,a) + \gamma\,\mathcal{E}_{s'\in S}\left[V^*_t(s')\right]\right\}, \qquad V^*_0(s) = r_T(s)$$

where the expectation is over all next states s' given the current state s and control action a. Since the reward is deterministic, the state space is discrete, and the next state s' has the probability density $p(s' \mid s,a)$, we can write the expectation explicitly as follows.

$$V^*_{t-1}(s) = \max_{a\in A(s)}\left\{r(s,a) + \gamma\sum_{s'\in S} V^*_t(s')\,p(s' \mid s,a)\right\}, \qquad V^*_0(s) = r_T(s)$$

To construct a controller over trajectories of finite length, we can set the discount factor to $\gamma = 1$. To construct a controller over trajectories of infinite length, the discount factor must be strictly less than one $\gamma < 1$ and the terminal reward is irrelevant.

Convergence. Unlike the deterministic case, value iteration for stochastic problems may only converge in the limit as the number of iterations tends to infinity. This happens where there are cycles in the graph with probabilities in the open interval $(0,1)$. As the algorithm iterates, the value function considers longer and longer trajectories that may traverse the cycle more and more times, each time with a reduced probability.



To illustrate the asymptotic convergence in the stochastic setting, consider the following problem, where each edge has a reward of 1 with the transition probability shown on the graph. With probability 1/2, the reward is 3. With probability 1/4, the reward is 7. With probability 1/8, the reward is 11. Each time another cycle is taken, the reward increases by 4 and the probability is cut in half. The expected cumulative reward is then the infinite summation

$$V(x_I) = 3\left(\tfrac{1}{2}\right) + 7\left(\tfrac{1}{4}\right) + 11\left(\tfrac{1}{8}\right) + \ldots = \sum_{i=0}^{\infty} \frac{3+4i}{2^{i+1}} = 7$$

The value function converges to a finite value, but only in the limit as the length of the path tends to infinity.

Stochastic value iteration may only converge asymptotically as the number of iterations tends to infinity.

Optimal controller. The value function implicitly describes a controller, which specifies which action to take from any state. The optimal controller is the argument that achieves the minimum in the value function recursion. This is a state feedback controller in that it chooses an action based on the state. If the value function is optimal, then so is the controller.

The value function implicity defines the state-feedback controller

$$\pi(s) \quad = \quad \underset{a\in A(s)}{\arg\max}\left\{ r(s,a) + \gamma \sum_{s'\in S} V(s')\, p(s' \mid s,a)\right\}$$

11.3 Applications

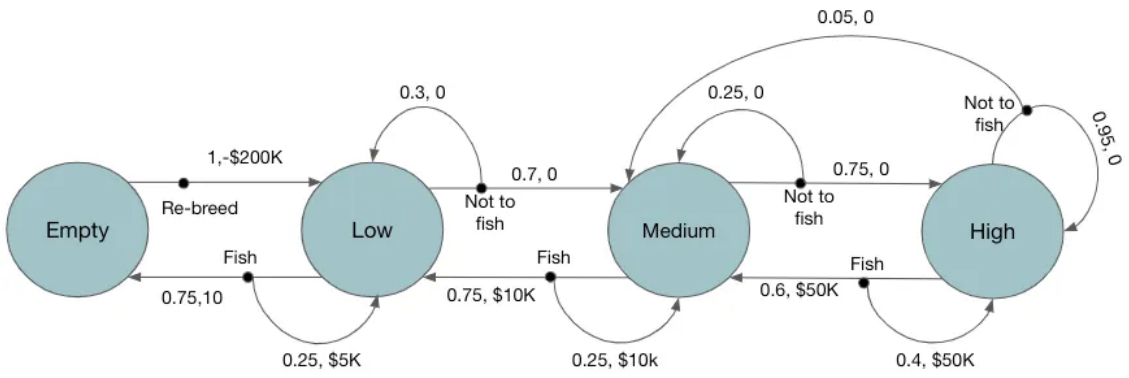
We now provide several applications of Markov decision processes[1].

Fishing

Consider fishing in a specific area over time. Our goal is to maximize the expected cumulative reward from fishing the area, where we get paid each year based on the number of fish that are caught and sold. Fishing too much results in a large immediate reward, but this will result in a low fish population that limits the future reward.

The search graph for this problem is shown below. In contrast to the deterministic case, each edge of the search graph now has three associated quantities: the control action a, the stage reward $r(s,a)$, and the transition probability $p(s' \mid s,a)$.

[1] `https://towardsdatascience.com/real-world-applications-of-markov-decision-process-mdp-a39685546026`



The states are the possible fish populations, which for simplicity are empty, low, medium, and high. From any non-empty fish population, the possible actions are to fish or not to fish. From the empty state, the only option is to re-breed the fish. There is no reward for choosing not to fish, while the reward for fishing increases with the size of the population. Re-breeding on the other hand has a large negative reward since it costs money. Fishing has a high probability of reducing the population, no probability of increasing the population, and a small probability that the population remains the same.

Value iteration. We can apply value iteration to find the policy that maximizes the expected cumulative discounted reward. The value function for the low state is updated as follows:

$$V^*_{t+1}(\text{Low}) = \max\big\{\$5 + \gamma\left[0.75\,V_t(\text{Empty}) + 0.25\,V_t(\text{Low})\right], \gamma\left[0.3\,V_t(\text{Low}) + 0.7\,V_t(\text{Medium})\right]\big\}$$

With $\gamma = 0.95$, the value function (rounded to whole numbers) over multiple iterations is shown in the following table, where the last row is the optimal value function (to within a small tolerance).

Iteration	Empty	Low	Medium	High
0	0	0	0	0
1	-200	5	10	50
2	-195	8	38	75
⋮	⋮	⋮	⋮	⋮
439	272	497	535	572

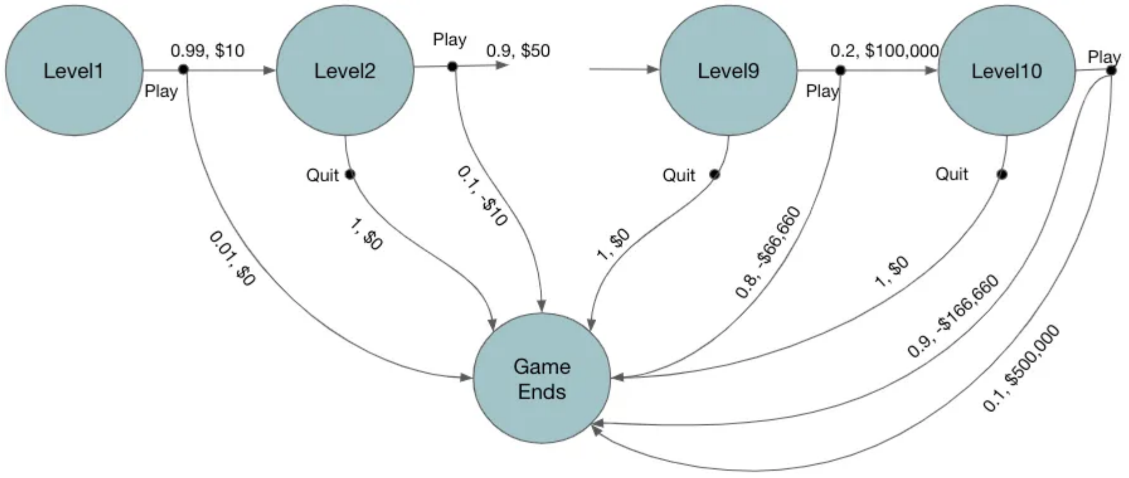
The optimal policy is as follows:

Empty	Low	Medium	High
Re-breed	Do not fish	Do not fish	Fish

The optimal controller indicates that the best strategy is to wait until the population is high and then fish. The reason to always fish when the population is high is that future rewards are discounted, so it is better to fish now and get the immediate reward than to not fish since that cannot lead to a better state.

Game show

Consider playing a game show that consists of 10 rounds of questions of increasing difficulty. In each round, the contestant chooses whether to answer the question or quit. Correctly answering the question earns a reward and moves the contestant to the next round, while answering incorrectly results in losing everything. At any point, the contestant may quit and keep the current reward.



Here, the reward for taking an action from a state is stochastic, as it also depends on whether the contestant answers the question correctly or not. The generalization of our problem formulation to this setting is straightforward.

11.4 Linear–Quadratic–Regulator

The linear–quadratic–regulator (LQR) problem is to find a control policy for a *linear* dynamical system that optimizes a *quadratic* cost function. In particular, consider the system dynamics

$$x_{t+1} = A_t x_t + B_t u_t + w_t$$

where $w_t \sim N(0, \sigma^2 I)$ is Gaussian noise. Given matrices Q_t and R_t, consider the optimal value function that is the minimum expected cumulative cost,

$$V_T^*(x_0) = \min_{u_0,\dots,u_{T-1}} \mathcal{E}\left[x_T^\top Q_T x_T + \sum_{t=0}^{T-1} x_t^\top Q_t x_t + u_t^\top R_t u_t\right]$$

where the expectation is over the process noise. We will show that the optimal value function is a quadratic function of the state, meaning that there exists a matrix P_t such that

$$V_t^*(x) = x^\top P_t x$$

Over trajectories of length zero, the optimal value function is the terminal cost, so $P_0 = Q_T$. Now suppose that the optimal value function over trajectories of length t is quadratic. We will show that the optimal value function over trajectories of length $t+1$ is also quadratic. From value iteration,

$$V_{t+1}^*(x) = \min_u \, (x^\top Q_{t+1} x + u^\top R_{t+1} u) + \mathcal{E}_w\left[V_t^*(A_t x + B_t u + w)\right]$$

Since the optimal value function over iterations of length t is quadratic (by assumption), the last term is

$$\mathcal{E}_w\left[V_t^*(A_t x + B_t u + w)\right] = \mathcal{E}_w\left[(A_t x + B_t u + w)^\top P_t (A_t x + B_t u + w)\right]$$

Expanding the quadratic and using that the noise w is zero mean with covariance $\sigma^2 I$, the expectation becomes

$$\mathcal{E}_w\big[V_t^*(A_t x + B_t u + w)\big] = (A_t x + B_t u)^\mathsf{T} P_t (A_t x + B_t u) + \sigma^2 \operatorname{tr}(P_t)$$

Therefore, we can rewrite value iteration as

$$V_{t+1}^*(x) = \min_u \begin{bmatrix} x \\ u \end{bmatrix}^\mathsf{T} \begin{bmatrix} A_t^\mathsf{T} P_t A_t + Q_{t+1} & A_t^\mathsf{T} P_t B_t \\ B_t^\mathsf{T} P_t A_t & B_t^\mathsf{T} P_t B_t + R_{t+1} \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} + \sigma^2 \operatorname{tr}(P_t)$$

The last term does not depend on the action u. In general, the quadratic optimization problem

$$\min_u \begin{bmatrix} x \\ u \end{bmatrix}^\mathsf{T} \begin{bmatrix} Q & S \\ S^\mathsf{T} & R \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix}$$

has optimal value $x^\mathsf{T}(Q - SR^{-1}S^\mathsf{T})x$ with optimizer $u = -R^{-1}S^\mathsf{T}x$. Applying this result to the above problem, we have that the optimal value function is quadratic with

$$P_{t+1} = A_t^\mathsf{T} P_t A_t + Q_{t+1} - A_t^\mathsf{T} P_t B_t \big(B_t^\mathsf{T} P_t B_t + R_{t+1}\big)^{-1} B_t^\mathsf{T} P_t A_t$$

and the optimal policy is the linear state-feedback controller $\pi_t(x) = K_t x$ where

$$K_t = -\big(B_t^\mathsf{T} P_t B_t + R_{t+1}\big)^{-1} B_t^\mathsf{T} P_t A_t$$

12

Partially Observable Markov Decision Processes

We now consider the case in which the state is not directly observable and both actuation and perception are corrupted by noise. We still assume that the state space is discrete. This problem formulation is called a partially observable Markov decision process (POMDP), which is a mathematical framework that models decision making under uncertainty in both control actions and measurements.

12.1 Problem formulation

The components of the problem are as follows:

- A finite state space S, where each element s of S is a state.
- An action space A, where each element a of A is an action.
- An observation space O, where each element o of O is an observation.
- A state transition probability $p(s' \mid s, a)$ that describes the probability of being in state s' given that the system was in state s and applied action a.
- An observation probability $p(o \mid s)$ that describes the probability of observing o from state s.
- A (possibly infinite) planning horizon T.
- A stage reward $r_t(s, a)$ associated with applying action a from state s at time t, and a terminal reward $r_T(s)$ associated with being in state s at the end of the planning horizon.

At each time $t \in \{0, 1, \ldots, T-1\}$, the system is in state $s_t \in S$, the decision-maker observes an observation $o_t \in O$ and then applies an action $a_t \in A$ which causes the system to transition to the next state $s_{t+1} \in S$. We assume that the state completely describes the system at each point in time. This is sometimes called the *Markov assumption*, which means that future states and past states are conditionally independent given the present state and decision,

$$\begin{aligned} p(s_{t+1} \mid s_t, a_t, \ldots, s_0, a_0) &= p(s_{t+1} \mid s_t, a_t) \\ p(o_t \mid s_t, a_{t-1}, s_{t-1}, \ldots, a_0, s_0) &= p(o_t \mid s_t) \end{aligned}$$

When choosing the action a_t, the decision-maker has access to the *information history*, which is the collection of all measurements and actions up to time t,

$$h_t = (o_0, a_0, \ldots, o_{t-1}, a_{t-1}, o_t)$$

In general, the policy that the agent uses to choose actions may depend on the entire information history, and may be deterministic or stochastic. Since a deterministic policy is a special case of a stochastic policy, we assume a stochastic policy. The action a_t is then sampled from the policy

$$a_t \sim \pi_t(\cdot \mid h_t)$$

where the policy is the probability distribution over actions given the information history.

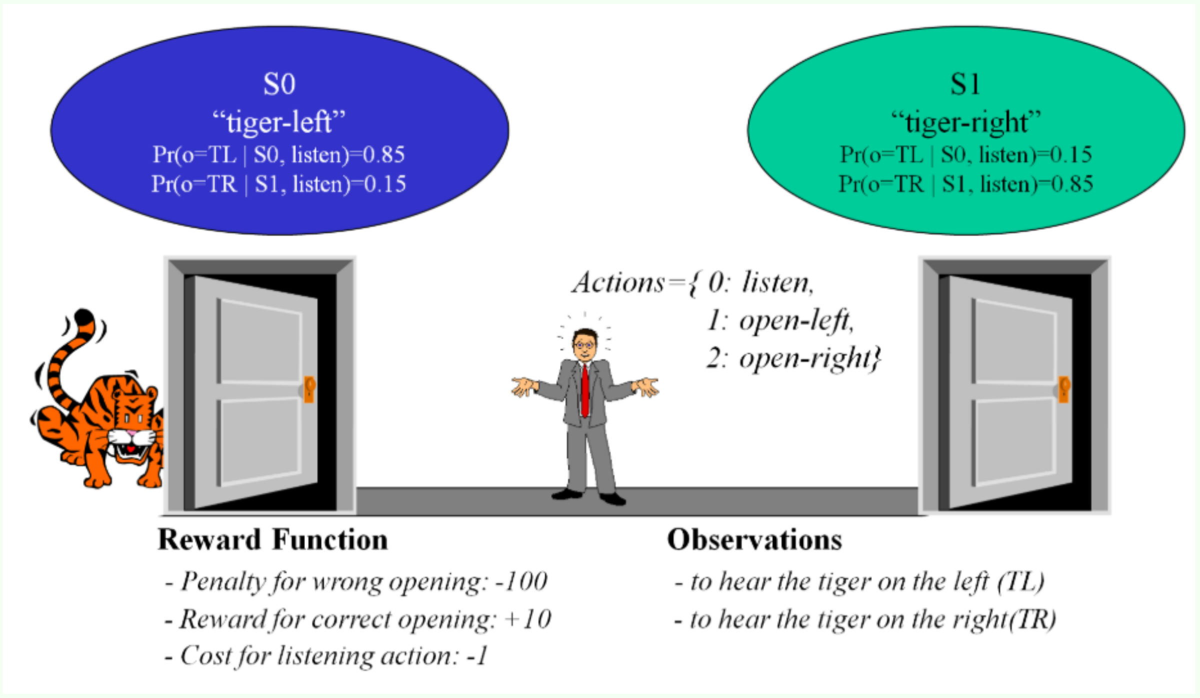
In contrast to the MDP case, the action space A does not depend on the state since we do not know the state (otherwise we could gain information by observing what actions are allowable!). Instead, we allow for any action $a \in A$ at each point in time, so the state transition probability must take this into account (if an action is not allowable, then its transition probability must be zero). Also, we only consider planning over a finite horizon since this case is tractable; we can approximate planning over an infinite horizon by making the terminal time T large.

Problem: Find a policy that maximizes the expected cumulative reward,

$$\underset{\pi_0,\ldots,\pi_{T-1}}{\text{maximize}} \ \mathbb{E}\left[\sum_{t=0}^{T-1} r_t(s_t, a_t) + r_T(s_T)\right]$$

where the expectation is taken over the observations $o_t \sim p(\cdot \mid s_t)$, actions $a_t \sim \pi_t(\cdot \mid h)$, and state transitions $s_{t+1} \sim p(\cdot \mid s_t, a_t)$. We denote the optimal policy at time t as π_t^*.

Example (Tiger POMDP)**.** A classic example of a POMDP is the following. Consider a game in which a tiger is hidden behind one of two doors, while a person seeks to open the door without the tiger. The person may choose to listen for the tiger or open either door. There is a small cost to listen, while there is a large cost to choosing the door with the tiger (because it will attack you). The goal is for the person to choose the actions that maximize the expected reward.


S0
"tiger-left"
Pr(o=TL | S0, listen)=0.85
Pr(o=TR | S1, listen)=0.15
S1
"tiger-right"
Pr(o=TL | S0, listen)=0.15
Pr(o=TR | S1, listen)=0.85
Actions={ 0: listen,
1: open-left,
2: open-right}
Reward Function
- Penalty for wrong opening: -100
- Reward for correct opening: +10
- Cost for listening action: -1
Observations
- to hear the tiger on the left (TL)
- to hear the tiger on the right(TR)

12.2 Value function

In the MDP case, the value function was a function of the state, and the optimal controller was constructed using the optimal value function. While we could use the same value function in this case, we do not know the state and therefore cannot construct the optimal controller from that value function. Instead, we would like the value function to depend on the information that we do know, which is precisely the information history. We now define the value function and associated controller as a function of the information history.

The (suboptimal) value function associated with a policy π from time t to the final time T is defined as the expected cumulative reward associated with an information history,

$$V_t^\pi(h) = \mathbb{E}\left[\sum_{k=t}^{T-1} r_k(s_k, a_k) + r_T(s_T) \;\middle|\; h_t = h\right]$$

The optimal value function is the maximum value function over all policies,

$$V_t^*(h) = \max_\pi \; V_t^\pi(h)$$

Our goal is to compute the optimal value function V_t^* at each time t, in which case the optimal policy is

$$\pi_t(a_t \mid h_t) = \arg\max_{a_t \in A} \; \mathbb{E}\big[r_t(s_t, a_t) + V_{t+1}^*(h_{t+1}) \mid h_t, a_t\big]$$

12.3 Principle of optimality

The principle of optimality states that the optimal value function can be computed recursively backward in time starting from the terminal time T.

Theorem (Principle of optimality)**.** The optimal value function satisfies the backward recursion

$$\begin{aligned} V_T^*(h_T) &= \mathbb{E}[r_T(s_T) \mid h_T] \\ V_t^*(h_t) &= \max_{a_t \in U} \; \mathbb{E}[r_t(s_t, a_t) + V_{t+1}^*(h_{t+1}) \mid h_t, a_t] \quad \text{for } t = 0, \dots, T-1 \end{aligned}$$

where $h_{t+1} = (h_t, z_{t+1}, a_t)$ is the information available at time t with $h_0 = (z_0)$.

Remark. The proof of the principle of optimality is considerably more complicated than in the MDP case. In particular, the same reasoning as before gives an *incorrect* proof. This line of reasoning is as follows. By definition, the optimal value function at time t is

$$V_t^*(h_t) = \max_K \; \mathbb{E}_K\left[\sum_{k=t}^{T-1} r_k(x_k, u_k) + r_T(x_T) \;\middle|\; h_t\right]$$

We can write this in terms of the value function at time $t+1$ as

$$V_t^*(h_t) = \max_K \; \mathbb{E}_K\big[r_t(s_t, a_t) + V_{t+1}^K(h_{t+1}) \mid h_t\big]$$

We can separate the maximization and expectation over the current controller K_t and the future controllers as

$$V_t^*(h_t) = \max_{K_T} \max_{K_{t+1:T-1}} \; \mathbb{E}_{K_t} \mathbb{E}_{K_{t+1:T-1}}\big[r_t(s_t, a_t) + V_{t+1}^K(h_{t+1}) \mid h_t\big]$$

But in general, the maximum of the expectation is less than or equal to the expectation of the maximum,

$$\max_x \mathbb{E}_y f(x,y) \le \mathbb{E}_y \max_x f(x,y)$$

So when we swap the order of the maximization and expectation, we get the inequality

$$V_t^*(h_t) \le \max_{K_T} \mathbb{E}_{K_t} \max_{K_{t+1:T-1}} \mathbb{E}_{K_{t+1:T-1}} \big[r_t(s_t,a_t) + V_{t+1}^K(h_{t+1}) \mid h_t \big]$$

The immediate reward is independent of the future variables, so we can bring in the inner maximization and expectation to obtain

$$V_t^*(h_t) \le \max_{K_T} \mathbb{E}_{K_t} \big[r_t(s_t,a_t) + V_{t+1}^*(h_{t+1}) \mid h_t \big]$$

Replacing the controller K_t with the action a_t, we obtain an inequality version of the principle of optimality,

$$V_t^*(h_t) \le \max_{a_t \in U} \mathbb{E}\big[r_t(s_t,a_t) + V_{t+1}^*(h_{t+1}) \mid h_t, a_t \big]$$

While this argument only shows that the above inequality holds, it can be shown that it also holds with equality, which is the principle of optimality. ■

Proof. We now provide a correct proof of the principle of optimality. At each time t, define the Bellman operator $\mathcal{T}_t$, which maps a function V of the information history at time $t+1$ to a function of the information history at time t,

$$\mathcal{T}_t(V)(h_t) = \min_u \ \mathbb{E}(r_t(s_t,a_t) + V(h_{t+1}) \mid h_t, a_t)$$

The principle of optimality then states that the optimal value function is obtained by iterating the Bellman operator,

$$V_T^*(h_T) = \mathbb{E}[r_T(x_T) \mid h_T], \qquad V_t^*(h_t) = \mathcal{T}_t(V_{t+1}^*)(h_t) \quad \text{for } t = 0, \dots, T-1$$

We write $f \ge g$ to denote that $f(x) \ge g(x)$ for all x.

a) *Bellman iteration is one-step optimal (even if the current value function is suboptimal).*

$$\begin{aligned} V_t^\pi(h_t) &= \mathbb{E}((r_t(s_t,a_t) + V_{t+1}^\pi(h_{t+1}))\pi_t(a_t,h_t) \mid h_t, a_t) \\ &\ge \min_{\hat\pi} \ \mathbb{E}\Big(\big(r_t(s_t,a_t) + V_{t+1}\pi(h_{t+1})\big)\, \hat\pi_t(s_t,a_t) \mid h_t, a_t \Big) \\ &= \min_a \ \mathbb{E}\big(r_t(s_t,a_t) + V_{t+1}^\pi(h_{t+1}) \mid h_t, a_t\big) \\ &= T_t(V_{t+1}^\pi)(h_t) \end{aligned}$$

The second step changes optimization over policies to optimization over actions, which is possible because we're optimizing a linear function of the pdf $\hat{K}$, so an optimal policy can be found by picking the pointwise maximum at each h_t.

b) *The Bellman operator is contractive.* That is, if $f \le g$, then $\mathcal{T}_t(f) \le \mathcal{T}_t(g)$. To prove this, suppose a pair of value functions satisfy $V_{t+1}^K \le V_{t+1}^{\hat{K}}$ for $t = 0, \dots, N-1$. Then for any tuple (s_t, a_t, h_{t+1}),

$$r_t(s_t,a_t) + V_{t+1}^K(h_{t+1}) \le r_t(s_t,a_t) + V_{t+1}^{\hat{K}}(h_{t+1})$$

Now use the fact that if $f \le g$, then $\min_x f(x) \le \min_x g(x)$. Minimizing both sizes with respect to a_t, we obtain $\mathcal{T}_t(V_{t+1}^K) \le T_t(V_{t+1}^{\hat{K}})$ as required.

We can now prove optimality of the recursive Bellman policy. For any policy K,

$$\begin{aligned}
V_t^K &\geq \mathcal{T}_t(V_{t+1}^K) && \text{(one-step optimality)} \\
&\geq \mathcal{T}_t \mathcal{T}_{t+1}(V_{t+2}^K) && \text{(monotonicity of } \mathcal{T}_t \text{ applied to } V_{t+1}^K \geq \mathcal{T}_{t+1}(V_{t+2}^K)) \\
&\geq \mathcal{T}_t \mathcal{T}_{t+1} \mathcal{T}_{t+2}(V_{t+3}^K) && \text{(monotonicity of } \mathcal{T}_{t+1} \text{ and } \mathcal{T}_t \text{ applied to } V_{t+2}^K \geq \mathcal{T}_{t+2}(V_{t+3}^K)) \\
&\vdots \\
&\geq \mathcal{T}_t \mathcal{T}_{t+1} \cdots \mathcal{T}_{N-1}(V_N^K) && \text{(iterating above steps)} \\
&\geq \mathcal{T}_t \mathcal{T}_{t+1} \cdots \mathcal{T}_{N-1}(V_N^*) && \text{(all policies have same final value function)} \\
&= V_t^* && \text{(recursion from the principle of optimality)}
\end{aligned}$$

To summarize, we have shown that

$$V_t^K \geq (V_t^* \text{ from principle of optimality}) = V_t^{K^*} = (V_t^* \text{ optimal value function})$$

Therefore, V_t^* from the principle of optimality is indeed an optimal policy, and it is also deterministic. ■

12.4 Belief

The belief is defined as the probability distribution over the state given the information history,

$$b_t(s \mid h) = p(s_t = s \mid h_t = h)$$

Notation. We previously wrote the belief as a function of the state only. Here, we emphasize that the belief depends also on the information history. ■

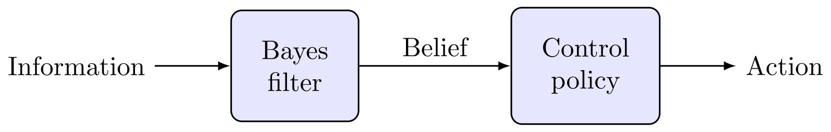
The Bayes filter describes how the belief changes based on applying actions and observing measurements. While we previously decomposed the Bayes filter into actuation and perception updates, we here describe a full update consisting of both actuation and perception. After applying an action a and observing o, the belief b' at the next time step is

$$b'(s', h') = \eta \sum_{s \in S} p(s' \mid s, a)\, p(o' \mid s)\, b(s, h)$$

where the constant η is chosen such that the belief sums to one and the information history at the next time step is $h' = (h, o', a)$.

12.5 Separation principle

The value function may in general depend on the entire information history whose size grows with time. Instead of keeping track of all past information, however, it is sufficient to keep track of the belief, which is of fixed size. For this reason, we say that the belief is a *sufficient statistic* for the problem.



Since we can construct the optimal policy from the optimal value function, and the optimal value function depends on the belief, we can separate the control problem into two steps: estimation and control. We first use the Bayes filter to construct the belief, which is the estimate of the underlying state, and then construct the optimal value function (for example, using value iteration) to obtain the optimal policy. This fundamental result is known as the separation principle.

Theorem (Separation principle)**.** The belief is a sufficient statistic in that the optimal value function is a function of the belief. In terms of the belief, value iteration is the backward recursion

$$\begin{aligned} V_T^*(b) &= \sum_{s\in S} r_T(s)\, b(s) \\ V_t^*(b) &= \max_{a\in A} \sum_{s\in S}\left(r_t(s,a) + \sum_{o\in O} V_{t+1}(b') \sum_{s'\in S} p(o \mid s')\, p(s' \mid s,a)\right) b(s) \quad \text{for } t = 0,\dots,T-1 \end{aligned}$$

where b' is the belief obtained from applying the Bayes filter from belief b with action a and observation o.

Proof. From the principle of optimality, the optimal value function at time T is

$$V_T^*(h_T) = \mathbb{E}[r_T(s) \mid h_T] = \sum_{s\in S} r_T(s)\, b_T(s)$$

where the expectation is over the state s. This is a function of the belief, b_T. Therefore, there exists a function J_T such that $V_T(h) = J_T(b_T(h))$. We will now use induction to show that this holds for all times, that is, there exists J_t such that $V_t(h) = J_t(b_t(h))$.

Suppose that $V_{t+1}(h) = J_{t+1}(b_{t+1}(h))$ for some function J_{t+1}, and apply the principle of optimality:

$$\begin{aligned} V_t(h_t) &= \min_{a_t}\ \mathbb{E}(r_t(s_t,a_t) + V_{t+1}(h_{t+1}) \mid h_t, a_t) \\ &= \min_{a_t}\ \mathbb{E}(r_t(s_t,a_t) + J_{t+1}(b_{t+1}(h_{t+1})) \mid h_t, a_t) \\ &= \min_{a_t}\ \mathbb{E}(r_t(s_t,a_t) + J_{t+1}(F_t(b_t(h_t), o_{t+1}, a_t)) \mid h_t, a_t) \\ &= \min_{a_t}\left(\sum_{s\in S} r_t(s_t,a_t)\, b_t(s,h_t) + \sum_{y\in Y} J_{t+1}(F_t(b_t(h_t), y, a_t))\, p(o_{t+1} = y \mid h_t, a_t)\right) \end{aligned}$$

The final probability above can be evaluated as:

$$\begin{aligned} p(o_{t+1} = o \mid h_t, a_t) &= \sum_{o,s\in S} p(o_{t+1} = o, s_{t+1} = s', s_t = s \mid h_t, a_t) \\ &= \sum_{o,s\in S} p(o_{t+1} = o \mid o_{t+1}, s_t = s, h_t, a_t)\, p(s_{t+1} \mid s_t, h_t, a_t)\, p(s_t = s \mid h_t, a_t) \\ &= \sum_{o,s\in S} p(y_{t+1} = y \mid s_{t+1}, a_t)\, p(x_{t+1} = z \mid s_t = s, a_t)\, p(s_t = s \mid h_t) \\ &= \sum_{o,s\in S} C_{t+1}(y,z,a_t)\, A_t(o,s,a_t)\, b_t(s,h_t) \end{aligned}$$

Subsituting this back into the above expression for the value function, we obtain

$$\begin{aligned} V_t(h_t) &= \min_{a_t\in A} \sum_{s\in S} b_t(s)\left(r_t(s_t,a_t) + \sum_{y\in Y} J_{t+1}(F_t(b_t(h_t), y, a_t)) \sum_{s',s\in S} C_{t+1}(y,s',a)\, A_t(s',s,a)\right) \\ &= J_t(b_t) \end{aligned}$$

for some J_t, since the value function depends only on the belief b_t. So by induction, we have that $V_t(h_t) = J_t(b_t)$ for all t. The optimal decision also depends only on the belief,

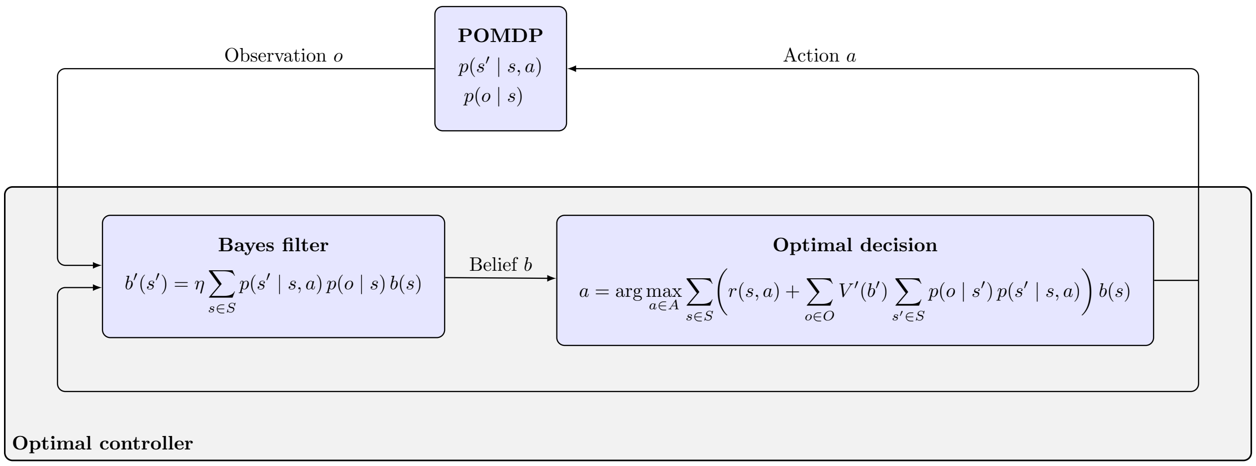
$$\begin{aligned} a_t &= \arg\min_{a_t \in A} \sum_{s \in S} b_t(s) \left(r_t(s_t, a_t) + \sum_{y \in Y} J_{t+1}(F_t(b_t(s), y, a_t)) \sum_{z, x \in X} C_{t+1}(y, z, u)\, A_t(z, x, u) \right) \\ &= k_t(b_t) \end{aligned}$$

Therefore, we do not need to remember the entire information history h_t (which is growing with time). It suffices to remember the belief b_t, which is of fixed size. For this reason, we say that the belief state is a sufficient statistic for the problem. ∎

Given the optimal value function V_t^* in terms of the belief, the optimal controller is the action that attains the maximum value. In particular, we can express the optimal controller in terms of the belief as

$$\pi_t(b) = \arg\max_{a \in A} \sum_{s \in S} \left(r_t(s, a) + \sum_{o \in O} V_{t+1}(b') \sum_{s' \in S} p(o \mid s')\, p(s' \mid s, a) \right) b(s)$$

The optimal controller separates into an optimal state estimator (the Bayes filter) and the above optimal controller based on the belief. While we could have used a stochastic controller, we have shown that there always exists an optimal controller that is deterministic, so it is not necessary to use a stochastic controller.


POMDP
$p(s' \mid s, a)$
$p(o \mid s)$
Observation o
Action a
Bayes filter
$b'(s') = \eta \sum_{s \in S} p(s' \mid s, a)\, p(o \mid s)\, b(s)$
Belief b
Optimal decision
$a = \arg\max_{a \in A} \sum_{s \in S} \left(r(s, a) + \sum_{o \in O} V'(b') \sum_{s' \in S} p(o \mid s')\, p(s' \mid s, a) \right) b(s)$
Optimal controller

12.6 LQG scenario

The problem of making sequential decision under uncertainty has a particularly simple solution when the models are *linear*, the reward is *quadratic*, and the noise is *Gaussian*. This is known as the LQG case. In the LQG scenario, the Bayes filter is the Kalman filter, and the optimal decision is the solution to the linear–quadratic regulator (LQR) problem.

When the system dynamics are linear, the state transition function has the form

$$x_{t+1} = As_t + Ba_t + w_t$$

where w_t is the process noise. Similarly, the measurement function has the form

$$z_t = Cs_t + v_t$$

where v_t is the measurement noise. Here, we assume that both the process and measurement noise are Gaussian random variables that are independent across time,

$$w_t \sim \mathcal{N}(0, W) \qquad \text{and} \qquad v_t \sim \mathcal{N}(0, V)$$

where W and V are the process and measurement covariances. We also assume that the initial state is a Gaussian random variable with $x_0 \sim \mathcal{N}(\hat{x}_0, \Sigma_0)$. We also assume that both the stage reward and terminal reward are quadratic,

$$r(x, u) = x^\mathsf{T} Q x + u^\mathsf{T} R u \qquad \text{and} \qquad r_T(x) = x^\mathsf{T} Q_T x$$

with $Q \succeq 0$, $Q_T \succeq 0$, and $R \succ 0$.

Under these assumptions, the Bayes filter is the Kalman filter. The belief at each time t is a Gaussian random variable with mean $\hat{s}_t$ and covariance Σ_t. We can use value iteration to recursively compute the belief backward in time as follows:

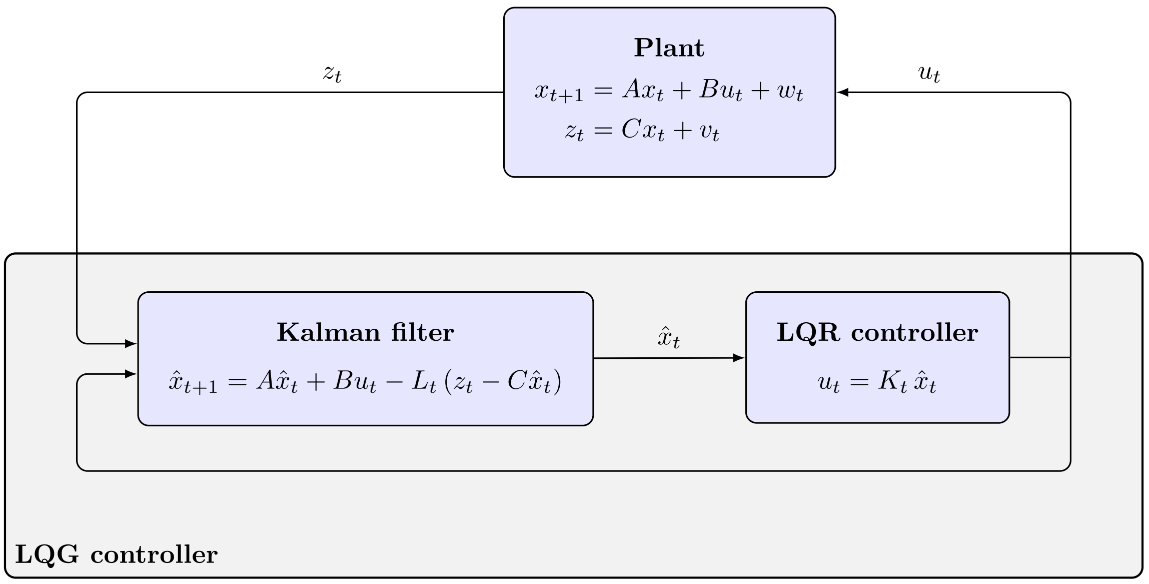
$$\begin{aligned} L_t &= -A\Sigma_t C^\mathsf{T}(C\Sigma_t C^\mathsf{T} + V)^{-1} \\ \Sigma_{t+1} &= A\Sigma_t A^\mathsf{T} + W + L_t C \Sigma_t A^\mathsf{T} \\ \hat{x}_{t+1} &= A\hat{s}_t + Ba_t - L_t\,(z_t - C\hat{s}_t) \end{aligned}$$

Using the optimal value function, the optimal decision is the LQR controller given by

$$\begin{aligned} K_t &= -(B^\mathsf{T} P_{t+1} B + R)^{-1} B^\mathsf{T} P_{t+1} A \\ P_t &= A^\mathsf{T} P_{t+1} A + Q - A^\mathsf{T} P_{t+1} B\,(B^\mathsf{T} P_{t+1} B + R)^{-1} B^\mathsf{T} P_{t+1} A \\ a_t &= K_t\,\hat{s}_t \end{aligned}$$

with terminal condition $P_T = Q_T$. Note that the Kalman filter propagates the belief forward in time, while value iteration recursively constructs the belief backward in time.

In this case, there is an additional separation in the structure of the optimal controller. For a general POMDP the optimal decision depends on the entire belief b_t. But here the optimal controller only depends on the mean $\hat{s}_t$ of the belief. The Kalman filter constructs the Gaussian belief which consists of a mean and covariance, but the optimal controller only depends on the mean of the belief. The optimal controller structure in this case is to produce the best estimate of the state possible, and then use the optimal *state-feedback* controller on this estimate as if it were the true state.



12.7 Belief MDP

As we have seen, the optimal controller separates into an optimal estimator (the Bayes filter) followed by an optimal decision based on the belief. While the state is unobservable, the belief is formed from the information history which is known, so the decision-maker can make decisions based on its belief. We now show that the optimal decision is equivalent to applying value iteration in the MDP case to the belief directly.

Recall that value iteration on the belief is given by

$$V_t^*(b_t) = \max_{u\in U} \sum_{x\in X} \left(r_t(x,u) + \sum_{z\in Z} V_{t+1}(b') \sum_{x'\in X} p(z \mid x')\, p(x' \mid x,u) \right) b_t(x)$$

To make this look like value iteration applied directly to the belief, we need to formulate the stage reward for a given belief and the transition function for the belief. The expected reward of having belief b and taking action u is

$$R_t(b,u) = \mathbb{E}[r_t(x,u) \mid h_t] = \sum_{x\in X} r_t(x,u)\, b(x)$$

The probability of the next belief b' given the current belief b and action u is

$$p(b' \mid b,u) = \sum_{x\in X} \sum_{z\in Z} \sum_{x'\in X} p(b' \mid b,u,z)\, p(z \mid x',u)\, p(x' \mid x,u)\, b(x)$$

where $p(b' \mid b,u,z)$ is a point-mass distribution centered on the single belief b' produced by the Bayes filter from the belief b with action u and measurement z. Using the reward and state transition for the belief, value iteration is equivalent to the recursion

$$V_t^*(b) = \max_{u\in U} \left\{ R_t(b,u) + \int V_{t+1}^*(b')\, p(b' \mid b,u)\, \mathrm{d}b' \right\}$$

where the integral is over all future beliefs b'. This is precisely value iteration applied directly to the (observable) belief, where the iteration involves an integral since the belief space is continuous (even though the state space is discrete). The only difference between this and the case in which the state space is discrete is that the summation became an integral over all beliefs. There are several difficulties:

- The belief space is continuous, so it is uncertain how to represent the belief in a tractable way.
- To do value iteration, we must compute an integral over belief space, which is not trivial.

12.8 Value iteration

As we have seen, we can use value iteration to recursively compute the optimal value function backward in time, and then use the optimal value function to construct the optimal controller. The difficulty is that, even when the state space is discrete, the belief space consists of probability distributions over states and is therefore continuous. While value iteration is conceptually very similar in both discrete and continuous spaces, it is much more difficult to implement in continuous spaces since we must somehow represent the value function on a computer.

Despite these difficulties, it is possible to find an exact solution when everything is finite (the state space, action space, observation space, and planning horizon). Let n denote the number of states, and denote the belief as a vector of probabilities $b = (p_1, \dots, p_n)$. In this case, the optimal value function over trajectories of any finite length is piecewise linear convex (PWLC), which is a function of the form

$$V_t^*(b) \quad = \quad \max_{v\in\mathcal{V}_t} v^\mathsf{T} b \quad = \quad \max_{v\in\mathcal{V}_t} \sum_{x\in X} v(x)\, b(x)$$

for some finite set $\mathcal{V}_t$ of n-dimensional vectors.

13

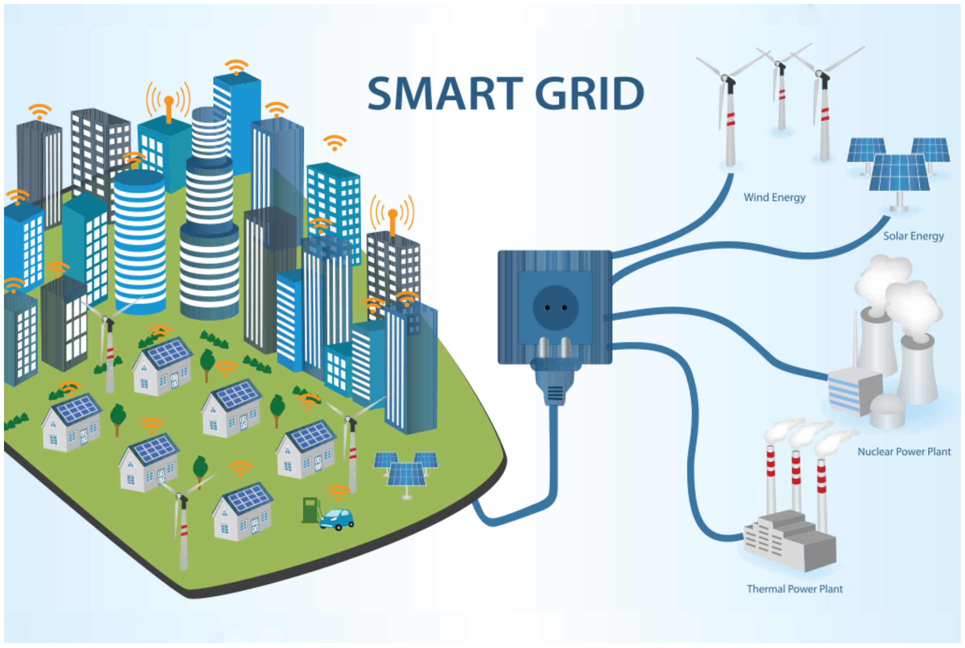
Multi-Agent Sensor Fusion

Sensor data is often collected using multiple sensors that are connected together in a network. These estimates are then fused together to form a cohesive view of the world. We now introduce decentralized algorithms for sensor fusion.

13.1 Motivation

Applications

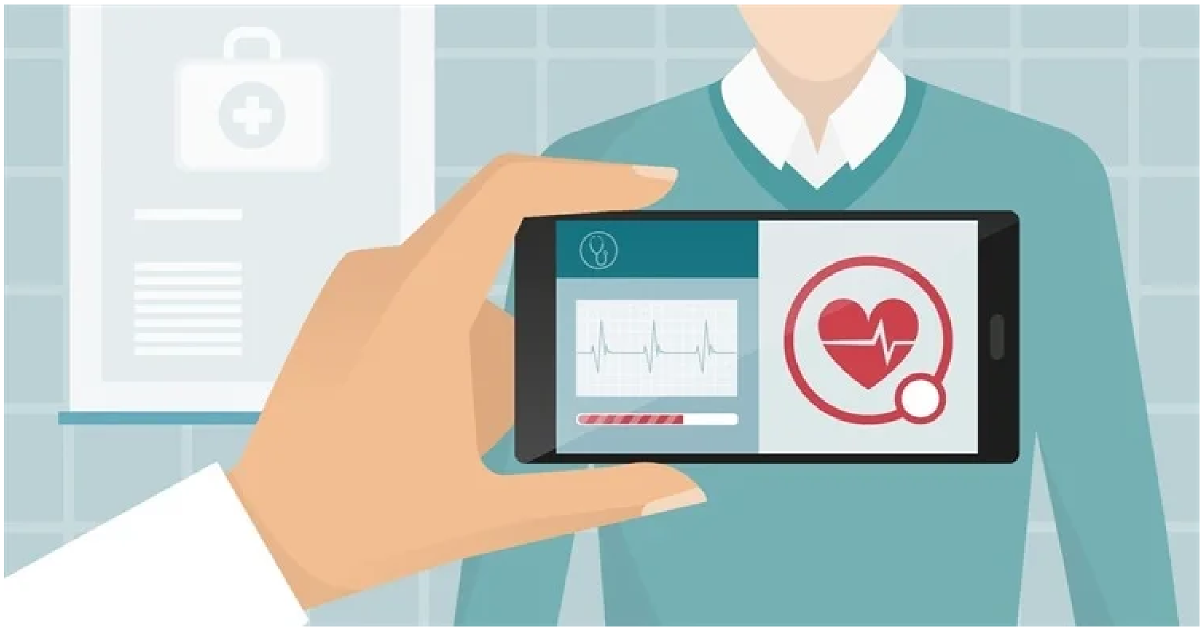
- **Smart grid.** The electric grid is a network of transmission lines, substations, transformers, and other devices that deliver electricity from the power plant to homes and businesses[1]. The smart grid uses communication between the utility and its consumers and sensing along transmission lines to increase efficiency, reduce outages, integrate renewable energy systems, and lower costs by reducing peak demand.



[1]`https://www.smartgrid.gov/the_smart_grid/smart_grid.html`

- **Smart transportation.** The transportation network consists of roads, train tracks, bike lanes, airports, and other infrastructure that enable transit. Smart transportation uses a variety of technologies to monitor, evaluate, and manage transportation systems to enhance efficiency and safety[2].

- **Smart healthcare.** Healthcare is the organized provision of medical care to individuals or a community. Smart healthcare is the integration of patients and doctors on a common platform for intelligent health monitoring by analyzing day-to-day human activities.



- **Environmental monitoring.** Environmental monitoring uses networks of sensors to analyze various environments, from farming crops to the ocean. Monitoring is used to build models of the environment (such as currents in the ocean), to track oil spills in the ocean, and to protect the public and the environment from toxic contaminants and pathogens.

[2]https://www.digi.com/blog/post/introduction-to-smart-transportation-benefits

Centralized solutions

In each of these applications, information is collected by sensors connected in a network. The main difficulty in fusing the information is that the information is spread across the network at various locations. One approach is to gather all the information in a single place, fuse the information (for instance, using a Bayes filter), and then send the information back to each agent. While straightforward, this solution has various drawbacks:

- the algorithm is not robust to failures of the centralized agent because, if the centralized agent fails, then the entire computation fails
- the method is not scalable because the amount of communication and memory required on each agent scales with the size of the network
- each agent must have a unique identifier so that the centralized agent counts each information only once
- the fused information is delayed by an amount that grows with the size of the network
- the information from each agent is exposed over the entire network, which is unacceptable in applications involving sensitive data

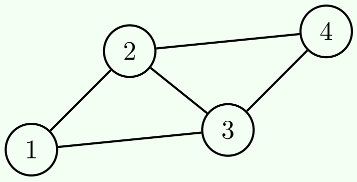
Another approach is called *flooding the network*. Instead of using a single central processor, here all agents act as the central processor. All sensor data is communicated across the network until every agent has all the information. Each agent can then implement a centralized data fusion algorithm. While this approach is more robust, it still has many of the drawbacks of the single centralized processor solution. In addition, the computational demands on each agent scales with the size of the network, so each agent must be a powerful processor.

To overcome these limitations, decentralized algorithms use the computation and communication abilities of each agent to fuse the information without using a central processor. Agents communicate relatively small amounts of information with their local neighbors and process the results using simple computations. This results in algorithms that are robust, scalable, safe, and efficient.

13.2 Graph theory

In multi-agent systems, it is common to describe the communication network among the agents using a graph. A graph is a collection of nodes (or vertices) that are connected by edges (or arcs). Nodes in the graph represent agents (such as sensors or robots), and edges in the graph indicate the flow of information among the agents.

Example. The following graph consists of the four nodes $\{1, 2, 3, 4\}$. There is an edge connecting nodes 2 and 4, which indicates that agent 2 is able to communicate information with agent 4. The adjacency and Laplacian matrices are as shown.

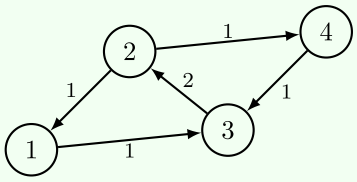


$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \qquad L = \begin{bmatrix} \neg 2 & -1 & -1 & \neg 0 \\ -1 & \neg 3 & -1 & -1 \\ -1 & -1 & \neg 3 & -1 \\ \neg 0 & -1 & -1 & \neg 2 \end{bmatrix}$$

In the above graph, edges do not have an associated direction. In terms of the communication network among the agents, this describes the scenario in which the flow of information is always bidirectional between two agents, meaning that if one agent can send information to another agent then it can also receive information from that agent. To describe directed communication, we can use a directed graph (or digraph), where edges have an associated direction. The direction is typically denoted by an arrow in the diagram. We use the convention that the direction of the arrow indicates the flow of information, so an arrow from node i to node j indicates that agent j can receive information from agent i.

In addition to direction, edges may also have weights associated with them. Directed graphs with edge weights are called *weighted digraphs*. The edge weights are typically shown next to the edge in the diagram. We can always interpret an unweighted graph as a weighted graph in which the edge weights are all one. Likewise, we can interpret an undirected graph as a directed graph in which edges always come in pairs.

Example. The following is a weighted digraph. The set of nodes is $\{1, 2, 3, 4, 5\}$, where each node represents an agent (or robot) in the system. There is a directed edge from node 2 to node 3 with weight a_{23}, which represents that agent 2 is able to communicate information to agent 3, and the information is weighted by a_{23}.



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \qquad L = \begin{bmatrix} \neg 1 & -1 & \neg 0 & \neg 0 \\ \neg 0 & \neg 2 & -2 & \neg 0 \\ -1 & \neg 0 & \neg 2 & -1 \\ \neg 0 & -1 & \neg 0 & \neg 1 \end{bmatrix}$$

There are several matrices that are often used to represent graphs. The *adjacency matrix* is an $n \times n$ matrix where n is the number of nodes in the graph. The ij^{th} component of the matrix is equal to the weight a_{ij} if there is an edge between nodes i and j and zero otherwise. The adjacency matrices for each of the previous examples is shown above.

Another common matrix is the *Laplacian matrix.* The Laplacian matrix is defined in terms of the adjacency matrix as $L = \operatorname{diag}(A\mathbf{1}) - A$, where $\mathbf{1}$ is an n-dimensional vector of all ones. In other words, the off-diagonal elements of the Laplacian matrix are $-a_{ij}$ for $i \neq j$, and the diagonal elements are $\sum_{j=1}^{n} a_{ij}$. Multiplication by the Laplacian matrix represents an averaging operation over the graph. To see this, suppose that each node i has an associated variable x_i, and stack all of these variables into the concatenated vector. When we multiply this vector by the Laplacian matrix, the i^{th} entry of the resulting vector is

$$(Lx)_i = \sum_{j=1}^{n} a_{ij}\,(x_i - x_j) \qquad \text{where} \qquad x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

This is a weighted sum of the differences between the variable on agent i and that of agent j. If all agents have the same value meaning that $x_1 = x_2 = \ldots = x_n$, then the product Lx is the vector of all zeros. For connected graphs, $Lx = 0$ if and only if all elements of x are the same, meaning that all agents agree on the same value. Agreement for multi-agent systems is often called *consensus*. This is quite useful as we often want all agents to have a common view of the world. The Laplacian matrices for each of the previous examples is shown above.

Connected. A graph is *connected* if there exists a path between any two nodes, and a directed graph is *strongly connected* if there exists a directed path from any node to any other node. Multi-agent algorithms require the communication graph to be connected since otherwise there would be some agents that do not receive information (either directly or indirectly) from some other agents which makes data fusion impossible.

Balanced. A weighted graph is *balanced* if the sum of the weights along all incoming edges is equal to the sum of the weights on all outgoing edges. Notice that the diagonal elements of the Laplacian matrix are the sums of the incoming weights, so each row of the Laplacian matrix sums to zero. In terms of the Laplacian matrix, a graph is balanced if and only if all columns of the Laplacian matrix also sum to zero. Undirected graphs are always balanced. The previous examples are both balanced. In multi-agent algorithms, using a balanced graph is important since this preserves the amount of incoming and outgoing information.

13.3 Static average consensus

We now consider one of the simplest problems for multi-agent systems: computing the average of static quantities over a network. While computing an average may seem trivial, it can be quite challenging depending on the scenario (such as if the quantities vary in time, the agents are mobile so the communication network changes in time, or agents enter/leave the network during the computation).

Consider a group of n agents, where each agent i has a static reference variable u_i. The static average consensus problem is for all agents to compute the average of the reference variables over all agents:

$$u_{\text{avg}} = \frac{1}{n}\sum_{i=1}^{n} u_i$$

Perhaps the simplest decentralized algorithm to solve this problem is as follows:

$$x_i(k+1) = x_i(k) - \sum_{j=1}^{n} a_{ij}\,\big(x_i(k) - x_j(k)\big), \qquad x_i(0) = u_i$$

where a_{ij} are the weights of the adjacency matrix of the communication graph and k is the iteration index. Each agent updates its estimate of the average using a weighted sum of the difference between its estimate and those of its neighbors ($a_{ij} = 0$ if there is no edge from node i to j, so the summation only involves quantities of neighboring agents). At each iteration k, this requires each agent i to communicate its current estimate $x_i(k)$ with its neighbors and then perform the above update. By stacking the agent variables into vectors, we can write the algorithm in terms of the graph Laplacian matrix as

$$x(k+1) = (I - L)\,x(k), \qquad x(0) = u$$

The following result states that, under reasonable assumptions on the communication network, the iterates on each agent converge asymptotically to the correct average u_{avg}.

Theorem. Suppose that the communication graph is a constant, strongly connected, and weight-balanced digraph and that the reference signal u_i at each agent $i \in \{1, \dots, n\}$ is a constant scalar. Then there exists a positive constant $c > 0$ such that

$$\left\| x_i(k) - \frac{1}{n} \sum_{i=1}^{n} u_i \right\| \leq c\, \rho^k$$

for all agents $i \in \{1, \dots, n\}$, where the convergence factor is $\rho = \|I - L - \frac{1}{n}\mathbf{1}\mathbf{1}^\mathsf{T}\|_2$.

The above algorithm solves the *static* average consensus problem in that each agent is able to compute the average of the reference variables over the network using only local communications with neighboring agents and local computations that do not scale with the number of agents. Notice, however, that the reference variables only enter the algorithm as the initial conditions. This algorithm therefore cannot adapt to changes in the reference signals or track time-varying references.

Choice of weights

The above algorithm converges if $\rho = \|I - L - \frac{1}{n}\mathbf{1}\mathbf{1}^\mathsf{T}\| < 1$ which depends on the edge weights in the graph (through the Laplacian matrix). For the directed graph above using the given edge weights, we have $\rho = 2.56$, so the bound does not guarantee convergence. If we instead choose the weights so that the adjacency matrix is

$$A = \begin{bmatrix} 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 \\ \frac{1}{3} & 0 & 0 & \frac{1}{3} \\ 0 & \frac{1}{2} & 0 & 0 \end{bmatrix}$$

then we have $\rho = 0.58$ and the iterates converge quickly to the average of the reference variables. Given the structure of the communication network, we can efficiently solve an optimization problem to find the edge weights that result in the fastest convergence rate[3].

Analysis

To analyze the convergence properties of the algorithm, we first show that the sum of the iterates is preserved over time. Multiplying the recursive update of the algorithm on the left by the row vector $\mathbf{1}^\mathsf{T}$ and using that the Laplacian is balanced, we have that

$$\mathbf{1}^\mathsf{T} x(k+1) = \mathbf{1}^\mathsf{T}(I - L)\, x(k) = \mathbf{1}^\mathsf{T} x(k)$$

so the average of the estimates is the same at each iteration. Applying this recursively and using the initialization, we have that $\mathbf{1}^\mathsf{T} x(k) = \mathbf{1}^\mathsf{T} u$ for all k. Now define the error of each agent i as the difference between the estimate of agent i and the exact average of the reference variables,

$$e_i(k) = x_i(k) - \frac{1}{n} \sum_{j=1}^{n} u_j$$

Let $e(k)$ denote the concatenated error vector over all the agents. Since the average is preseved over time, we can write the error as

$$e(k) = \left(I - \tfrac{1}{n}\mathbf{1}\mathbf{1}^\mathsf{T}\right) x(k)$$

[3]`https://web.stanford.edu/~boyd/papers/pdf/fastavg_cdc03.pdf`

The matrix $\frac{1}{n}\mathbf{1}\mathbf{1}^\mathsf{T}$ can be interpreted as a perfect averaging matrix, since multiplying this matrix by a vector results in a vector where each element is the average of the original vector. Now consider how the error evolves as a function of time. Using the recursive update of the algorithm, the error at the next time step is

$$e(k+1) = \left(I - \tfrac{1}{n}\mathbf{1}\mathbf{1}^\mathsf{T}\right)(I - L)\,x(k) = \left(I - L - \tfrac{1}{n}\mathbf{1}\mathbf{1}^\mathsf{T}\right)x(k) = \left(I - L - \tfrac{1}{n}\mathbf{1}\mathbf{1}^\mathsf{T}\right)e(k)$$

where we used that the Laplacian satisfied both $L\mathbf{1} = \mathbf{0}$ and $\mathbf{1}^\mathsf{T}L = \mathbf{0}^\mathsf{T}$. Taking the norm of both sides gives the bound $\|e(k+1)\| \leq \rho\,\|e(k)\|$. Applying this recursively, we obtain $\|e(k+1)\| \leq \rho^k\,\|e(0)\|$ which implies the above result. ■

13.4 Dynamic average consensus

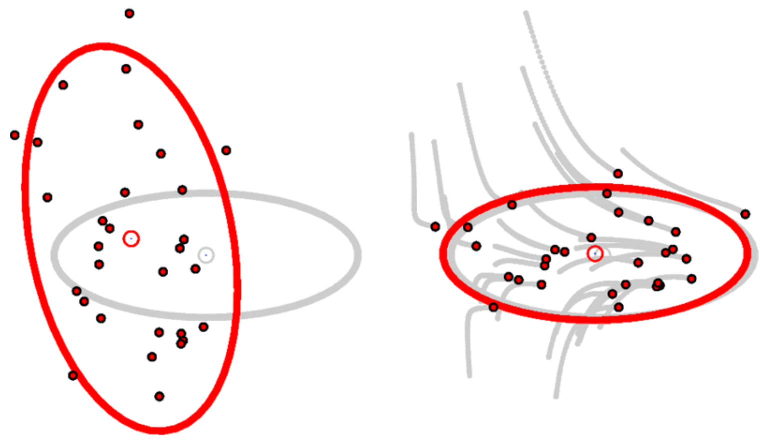
Applications

Formation control

Consider a group of n agents that seek to position themselves to form a particular formation. One way to represent a formation is as a vector of moments. For instance, if the agents are in the two-dimensional plane and we let (p_{ix}, p_{iy}) denote the position of agent i with $p = (p_1, \ldots, p_n)$ the positions of all agents, then a vector of first- and second-order moments is of the form

$$f(p) = \frac{1}{n}\sum_{i=1}^{n}\begin{bmatrix} p_{ix} \\ p_{iy} \\ p_{ix}^2 \\ p_{iy}^2 \end{bmatrix}$$

If each agent can measure its position, then the agents can move in such a way that their positions minimize the cost $\|f(p) - f_\star\|$ where $f_\star$ is a given a vector of desired moments[4].



Environmental monitoring

Consider a group of n agents where each agent is able to measure the environment at its current location. The goal is for each agent to construct a cohesive model of the global environment in a decentralized

[4]https://ieeexplore.ieee.org/document/4700861

manner through communicating with its local neighbors. Suppose that the environmental is described by the parameters x that evolve according to the linear dynamics

$$x(k+1) = Ax(k) + Bu(k) + w(k)$$

where $w(k)$ is zero-mean Gaussian noise with covariance matrix $Q(k)$. We will assume for simplicity that A and B are constant, but the following also works when they vary in time. In the absense of a good model for the environment dynamics, we can set A to the identity and B to zero to indicate that the environment is (approximately) static.

At each time k, each agent i measures the environment at its location $p_i(k)$ to obtain the measurement $z_i(k)$. We assume that measurements are linear functions of some given basis functions for the environment, that is,

$$z_i(k) = C_i(k)\, x(k) + v_i(k)$$

where $v_i(k)$ is zero-mean Gaussian noise with covariance matrix $R_i(k)$ and

$$C_i(k) = \begin{bmatrix} \psi^{(1)}\big(p_i(k)\big) & \psi^{(1)}\big(p_i(k)\big) & \dots & \psi^{(1)}\big(p_i(k)\big) \end{bmatrix}$$

The functions $\psi^{(j)}$ are local basis functions, such as sinusoids in Fourier series, wavelets, piecewise polynomials, or splines. Stacking all of the quantities into vectors, this is equivalent to

$$z(k) = C(k)\, x(k) + v(k)$$

The optimal state estimator for this system is the Kalman filter. In information form, the actuation update is

$$\Omega' = (A\Omega^{-1}A + Q)^{-1}$$
$$\xi' = \Omega'\,(A\Omega^{-1}\xi + Bu)$$

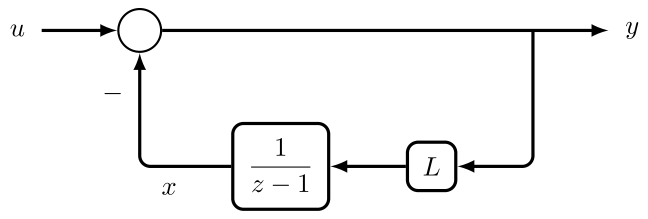
Each agent can implement these equations to update its model of the environment based on the control action $u(k)$. The perception update is

$$\Omega' = \Omega + C^{\mathsf{T}} R^{-1} C = \Omega + \sum_{i=1}^{n} C_i^{\mathsf{T}} R_i^{-1} C_i$$
$$\xi' = \xi + C^{\mathsf{T}} R^{-1} z = \xi + \sum_{i=1}^{n} C_i^{\mathsf{T}} R_i^{-1} z_i$$

Updating the estimate of the environmental parameters based on observations requires the measurements of *all* agents. Since information is additive, this appears as a sum over all agents. One way for the agents to implement the information filter in a decentralized manner is to use a dynamic average consensus estimator to track the average of each element of the summations, multiply by the number of agents (which may be known or can be estimated), and then added to the current information matrix and information vector.[5]

Dynamic average consensus estimator

All of the above examples require a group of agents to cooperatively track the average of time-varying signals. We now describe a simple algorithm to do this, known as a *dynamic average consensus estimator*.



[5]https://ieeexplore.ieee.org/document/4542870

One particular estimator is given by the above block diagram, where u is the (possibly time-varying) vector of reference signals, y is the vector of estimates of the average of the reference signals, and L is the graph Laplacian. This is a discrete-time linear time-invariant system, which is described by the following difference equations:

$$\begin{aligned} x(k+1) &= x(k) + Ly(k) \\ y(k) &= u(k) - x(k) \end{aligned}$$

While this is a compact way of representing the algorithm, keep in mind that multiplication by the Laplacian L is an operation that can be performed in a decentralized manner using only communication with neighboring agents. For each agent i, the algorithm is given by

$$\begin{aligned} x_i(k+1) &= x_i(k) + \sum_{j\in\mathcal{N}_i} a_{ij}\left(y_i(k) - y_j(k)\right) \\ y_i(k) &= u_i(k) - x_i(k) \end{aligned}$$

where $\mathcal{N}_i$ is the set of agents from which agent i receives information (that is, its neighbors). Each iteration of the algorithm requires agents to perform a round of communication with neighboring agents and simple local computations. The algorithm only requires each agent to store a single scalar state variable $x_i(k)$.

Theorem. At each iteration k, the error of the estimator is upper bounded by

$$\|y(k) - \operatorname{avg}(u(k))\| \le \|\operatorname{avg}(x(0))\| + \sum_{m=0}^{k} \|I - L - \tfrac{1}{n}\mathbf{1}\mathbf{1}^\mathsf{T}\|^{k-m}\,\|\operatorname{dis}(u_m - u_{m-1})\|$$

where $u(-1) := x(0)$ is the initial condition.

From this bound, we make a few observations:

- The first term depends on the average initial condition $x(0)$. This suggests that we must initialize the state of the estimator to have an average of zero. A simple way to do this is to set $x_i(0) = 0$ for each agent i.
- The second term depends on the disagreement between the difference of the reference signals at two consecutive time steps. If the reference signals are constant, then these terms are all zero and the error converges asymptotically to zero, just as with the static average consensus estimator. When the reference signals are time varying, the error depends on how quickly the signals change.
- The graph Laplacian appears in the term $\|I - L - \frac{1}{n}\mathbf{1}\mathbf{1}^\mathsf{T}\|$. To improve the convergence properties of the algorithm, we can choose the edge weights to minimize this quantity, just as we did for the static average consensus estimator.

Analysis

The algorithm is a discrete-time linear time-invariant system, so we can write down the closed-form solution for the error signal as a function of time and use that to construct the upper bound. First, define the error as the difference between the output and the average input,

$$e(k) = y(k) - \operatorname{avg}(u(k))$$

Using the algorithm dynamics, the error satisfies the recursion

$$e(k+1) = (I-L)\,e(k) + \operatorname{dis}\big(u(k+1) - u(k)\big), \qquad e(0) = \operatorname{dis}(u(0)) - x(0)$$

This is a linear time-invariant system which has the solution

$$e(k) = (I - L)^k \big(\mathrm{dis}(u(0)) - x(0)\big) + \sum_{m=0}^{k-1} (I - L)^{k-m-1} \, \mathrm{dis}(u(m+1) - u(m))$$

Shifting the index in the summation and using that multiplication by $\frac{1}{n}\mathbf{1}\mathbf{1}^\mathsf{T}$ does not affect the disagreement directions, we have that

$$e(k) = (I - L)^k \big(\mathrm{dis}(u(0)) - x(0)\big) + \sum_{m=1}^{k} (I - L - \tfrac{1}{n}\mathbf{1}\mathbf{1}^\mathsf{T})^{k-m} \, \mathrm{dis}(u(m) - u(m-1))$$

Now define $u(-1) = x(0)$ and absorb the disagreement part of the first term into the summation to obtain the simplified expression

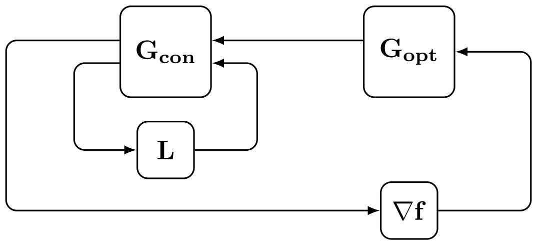
$$e(k) = -\mathrm{avg}(x(0)) + \sum_{m=0}^{k} (I - L - \tfrac{1}{n}\mathbf{1}\mathbf{1}^\mathsf{T})^{k-m} \, \mathrm{dis}(u(m) - u(m-1))$$

Taking the norm of both sides and using that $\|a+b\| \le \|a\| + \|b\|$ and $\|ab\| = \|a\|\,\|b\|$ gives the upper bound in the result. ■

Extensions

The above algorithm has the limitation that it must be initialized correctly in order to have zero steady-state error. While this is straightforward to do when initializing the algorithm, it makes the algorithm non-robust to agents entering or leaving the network in the middle of the computation (for instance, if an agent's battery dies or it wanders away from the rest of the agents). We can interpret such a scenario as restarting the algorithm from the current iterates of all the agents except for the addition or subtraction of the agent that entered or exited. If $x_i(k)$ is nonzero for the transitioning agent i, then this results in an error that will persist in the algorithm. There are algorithm designs that automatically adjust to agents entering and leaving the network; such algorithms are called *robust* dynamic average consensus estimators.

Here, we only considered computing a simple average over a network. To solve complex problems, agents may need to fuse their information in other ways. For instance, the agents may want to cooperatively solve an optimization problem. This is known as *decentralized optimization.* It turns out that decentralized optimization algorithms decompose into a centralized optimization algorithm and a dynamic average consensus estimator[6].



[6]https://ieeexplore.ieee.org/document/9794703

Part V

Appendices

A

Probability

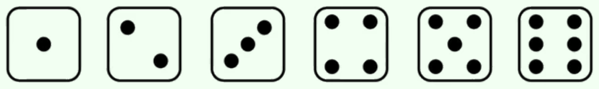
Probability is a branch of mathematics that describes the likelihood of an event. In probabilistic robotics, uncertainty in sensor measurements, control actions, the robot state, and the map of the environment are taken into account by modeling such quantities as random variables. This chapter describes the basic probabilistic concepts and notation that are used throughout the book.

A.1 Probability space

A *probability space* is a mathematical tool used to model a random experiment in which various *outcomes* may occur. A probability space consists of the following three elements:

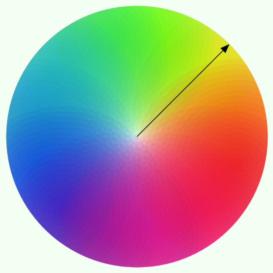
- The **sample space** is the set of all possible outcomes of the experiment.
- The **event space** is the set of all events, where each event is a set of outcomes.
- The **probability function** that assigns a probability to each event.

Example (Rolling a die). We can use probability to model the number facing up after rolling a die.



- In this example, the experiment is rolling the die.
- The possible outcomes are that any one face of the die is facing up; we label each of these outcomes with the numbers one through six.
- The sample space is the set of all outcomes, which is the set $\{1, 2, 3, 4, 5, 6\}$.
- Events are sets of outcomes (or subsets of the sample space), such as the event of rolling the number one $\{1\}$ and the event of rolling an even number $\{2, 4, 6\}$.
- The probability function assigns a probability to each event. If the die is fair, for instance, then we would assign the same probability to each outcome.

Example (Spinner)**.** We can use probability to model the direction (or color) of a spinner after being spun.



- In this example, the experiment is spinning the spinner.
- The possible outcomes are the directions that the spinner may face; we label each of these outcomes with their angle from the horizontal axis.
- The sample space is the set of all outcomes, which is the set $[0, 2\pi)$.
- Events are sets of outcomes (or subsets of the sample space), such as the event of facing exactly north $\{\pi/2\}$ and the event of being on the right-hand side $[-\pi/2, \pi/2]$.
- The probability function assigns a probability to each event. If the spinner is fair, for instance, then we would assign the same probability to each angle.

The probability of an event is a real number between zero and one that describes the likelihood of the event. The probability of an event A is denoted $P(A)$.

Probability functions must satisfy certain rules. For any events A and B, the following properties hold.

- $0 \leq P(A) \leq 1$
- if $A \subset B$, then $P(A) \leq P(B)$
- $P(A^c) = 1 - P(A)$ where A^c is the complement of A
- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$

If A and B are disjoint (their intersection is empty), then the last property simplifies to $P(A \cup B) = P(A) + P(B)$.

Furthermore, the probability that any event occurs is one.

Example (Die, continued)**.** Going back to our example, suppose the die is fair, meaning that each side has the same probability of landing face up. Then each probability must be $P(\{i\}) = 1/6$ for $i \in \{1, 2, 3, 4, 5, 6\}$ so that the total probability is one. From this, we can construct probabilities of other sets by using the above properties. For instance, we can compute the probability of rolling an even number as

$$P(\{2,4,6\}) = P(\{2\} \cup \{4\} \cup \{6\}) = P(\{2\}) + P(\{4\}) + P(\{6\}) = \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2}$$

A.2 Random variables

A *random variable* is a variable over the outcomes of an experiment.

Remark. Technically, a random variable is neither random nor a variable. Instead, a random variable is a function from an outcome to a number (such as a real number or integer). In this class, however, we will ignore this detail and think of random variables as quantities that can be manipulated (for example, using addition and multiplication) and have other associated quantities (such as probability densities, expected values, etc.). ■

Notation. We typically use a capital letter (such as X) to refer to a random variable and the corresponding lower case letter (such as x) to refer to its possible values. ■

Continuous vs discrete

A random variable may be *discrete* or *continuous*, depending on the size of the sample space of the experiment. A discrete random variable is a random variable over a finite (or countably infinite, such as the set of integers) set of outcomes. An example of a discrete random variable is rolling a die, since there are only six possible outcomes. A continuous random variable is a random variable over an uncountably infinite (such as the set of real numbers) set of outcomes. An example of a continuous random variable is measuring the distance to an object, since the outcome may be any positive real number (possibly up to a maximum reading of the sensor).

Probability mass and density functions

For a discrete random variable X, its associated probability mass function (pmf) is the function p_X that maps outcomes x to probabilities by

$$p_X(x) = P(X = x)$$

For a continuous random variable X, its associated probability density function (pdf) is the function p_X that maps outcomes x to probabilities such that

$$\int_a^b p_X(x)\,\mathrm{d}x = P(a \leq X \leq b)$$

Notation. For simplicity, we often omit the reference to the random variable and just write $p(x)$. ■

Total probability

Since the probability that any event occurs is one, summing a discrete random variable over all outcomes must be one. Likewise, integrating a continuous random variable over all outcomes must be one.

$$\sum_x p(x) = 1 \qquad \text{or} \qquad \int_x p(x)\,\mathrm{d}x = 1$$

The sum and integral are over the entire sample space.

Joint probability

The joint probability that a random variable X has value x and another random variable Y has value y is

$$p(x, y) = p(X = x \text{ and } Y = y)$$

The random variables are called *independent* if the joint probability is the product of the individual probabilities,

$$p(x, y) = p(x)\, p(y) \qquad \text{(independent)}$$

Conditional probability

Random variables often provide information about other random variables. Suppose that we know the value y of a random variable Y and we would like to know the value x of another random variable X. This is the conditional probability $p(X = x \mid Y = y)$ that is defined as

$$p(x \mid y) = \frac{p(x, y)}{p(y)}$$

If X and Y are independent, then Y provides no information about X and their conditional probability simplifies to

$$p(x \mid y) = p(x) \qquad \text{(independent)}$$

Conditional independence

Two random variables X and Y are *conditionally independent* from another random variable Z if the random variable Z carries no information about the conditional distribution of X given Y. Conditional independence can be expressed in the following equivalent forms:

$$p(x \mid y, z) = p(x \mid z) \qquad \text{or} \qquad p(x, y \mid z) = p(x \mid z)\, p(y \mid z)$$

Remark. In general, conditional independence does *not* imply independence, and independence does *not* imply conditional independence! ■

Law of total probability

The law of total probability involves decomposing a probability function as the sum (or integral) over the entire sample space.

$$p(x) = \sum_y p(x \mid y)\, p(y) \qquad \text{or} \qquad p(x) = \int_y p(x \mid y)\, p(y)\, \mathrm{d}y$$

This is often useful to decompose a probability that is unknown, $p(x)$, into probabilities that are known, $p(x \mid y)$ and $p(y)$.

Bayes rule

Bayes rule provides another way of rewriting a random variable in terms of other random variables (that may be easier to compute). For events A and B, Bayes rule states that the conditional probability of A

given B is

$$p(A \mid B) = \frac{p(B \mid A)\, p(A)}{p(B)}$$

In terms of probability functions, Bayes rule states that

$$p(x \mid y) = \frac{p(y \mid x)\, p(x)}{p(y)} = \frac{\text{likelihood} \cdot \text{prior}}{\text{evidence}}$$

This follows directly from the definition of conditional probability since

$$p(x, y) = p(x \mid y)\, p(y) = p(y \mid x)\, p(x)$$

Typically, the random variable x is the quantity that we are trying to estimate, and the random variable y is the available information. The likelihood $p(y \mid x)$ is the probability of measuring y given x. Since we have measured y, we can think of this as a function of the unknown x, although it is not a probability distribution in x. The prior $p(x)$ describes the probability of an estimate before fusing the information, and the evidence $p(y)$ is the probability of the measurement independent of the unknown x. As a function of x, the evidence is constant so we typically absorb it into a normalizing constant:

$$p(x \mid y) = \eta\, p(y \mid x)\, p(x) \qquad \text{where} \qquad \eta = \frac{1}{p(y)}$$

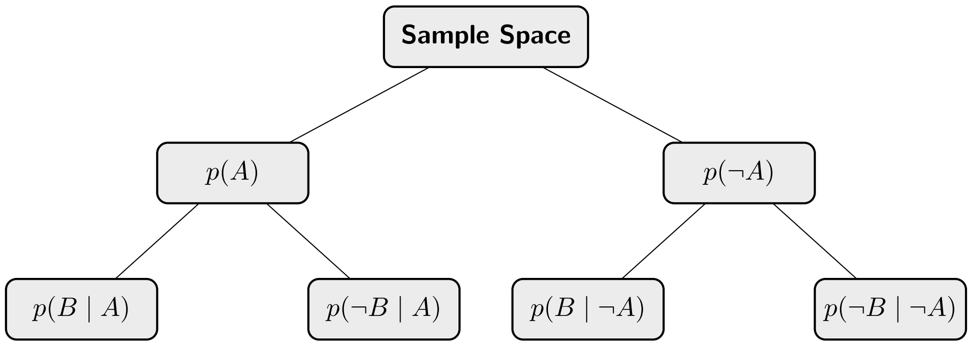
If we have other background knowledge (such as previous measurements), then we can condition all probabilities in Bayes rule by this extra information:

$$p(x \mid y, z) = \frac{p(y \mid x, z)\, p(x \mid z)}{p(y \mid z)}$$

Notation (Normalizing constant)**.** We often use η to refer to a normalizing constant that can be inferred by the fact that the resulting expression must be a valid probability distribution and therefore sum (or integrate) to one. The same symbol η may be used in multiple expressions, even though the value of the constant is different. ∎

Tree diagram

We can use a tree diagram to visualize events and their associated probabilities. Consider two events A and B. The following two (equivalent) trees illustrate all possible scenarios.



Given any three independent probabilities, it is possible to compute all remaining probabilities in the tree. For instance, suppose we know the probabilities $p(A)$ and $p(B)$ of each event along with the conditional probability $p(B \mid A)$. We can then solve for the remaining probabilities as follows. First, from the law of total probability, the probability of not A is $p(\neg A) = 1 - p(A)$ and the probability of not B is $p(\neg B) = 1 - p(B)$. We can also use Bayes rule to obtain the conditional probability of A given B as

$$p(A \mid B) = \frac{p(B \mid A)\, p(A)}{p(B)}$$

Applying the law of total probability again, the probability of not A given B is $p(\neg A \mid B) = 1 - p(A \mid B)$. Then applying Bayes rule again, the probability of not A given B is

$$p(\neg A \mid B) = \frac{p(B \mid \neg A)\, p(\neg A)}{p(B)}$$

A.3 Gaussian random variable

A Gaussian random variable is a continuous random variable whose probability density function is the normal distribution. For a one-dimensional Gaussian random variable with mean μ and variance σ^2, the density is

$$p(x) \quad = \quad \mathcal{N}(x; \mu, \sigma^2) \quad = \quad \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right)$$

For a multi-dimensional random variable with mean μ and covariance matrix Σ, the density is

$$p(x) \quad = \quad \mathcal{N}(x; \mu, \Sigma) \quad = \quad \det(2\pi\Sigma)^{-\frac{1}{2}} \exp\left(-\tfrac{1}{2}(x-\mu)^\mathsf{T}\Sigma^{-1}(x-\mu)\right)$$

This recovers the one-dimensional case when x is a scalar and $\Sigma = \sigma^2$.

The cumulative distribution function of a one-dimensional Gaussian random variable is

$$\frac{1}{2}\left[1 - \operatorname{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right]$$

where erf is the error function.

Proposition. If $x \sim \mathcal{N}(\mu, \Sigma)$, then $Ax + b \sim \mathcal{N}(A\mu + b, A\Sigma A^\mathsf{T})$. ■

Proposition. If $\begin{bmatrix} x \\ y \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix}, \begin{bmatrix} \Sigma_x & \Sigma_{xy} \\ \Sigma_{yx} & \Sigma_y \end{bmatrix}\right)$, then $x \mid y \sim \mathcal{N}(\mu_x + \Sigma_{xy}\Sigma_y^{-1}(y - \mu_y), \Sigma_x - \Sigma_{xy}\Sigma_y^{-1}\Sigma_{yx})$. ■

Proposition. If $x \sim \mathcal{N}(\mu_x, \Sigma_x)$ and $y \sim \mathcal{N}(\mu_y, \Sigma_y)$ are independent, then $x + y \sim \mathcal{N}(\mu_x + \mu_y, \Sigma_x + \Sigma_y)$. ■

A.4 Statistics

There are various statistics that we can use to describe a random variable.

Expectation

The *expectation* (or expected value) of a random variable X is

$$\mathcal{E}(X) = \sum_x x\, p(x) \qquad \text{or} \qquad \int_x x\, p(x)\, \mathrm{d}x$$

The expectation is linear, so for any a and b,

$$\mathcal{E}(aX+b) = a\,\mathcal{E}(X) + b$$

For a vector or matrix-valued random variable, the expectation is computed componentwise.

Notation. We often use the symbol μ to denote the expectation of a random variable. ■

Covariance

The *covariance* is a measure of the joint variability between two random variables. The covariance is defined as

$$\mathbf{cov}(X,Y) = \mathcal{E}[(X - \mathcal{E}\,X)(Y - \mathcal{E}\,Y)^\mathsf{T}] = \mathcal{E}(XY^\mathsf{T}) - \mathcal{E}(X)\,\mathcal{E}(Y^\mathsf{T})$$

A Guassian random variable is parameterized by its covariance matrix Σ.

Variance

The *variance* of a random variable is its covariance with itself,

$$\mathrm{var}(X) = \mathbf{cov}(X,X) = \mathcal{E}[(X-\mu)(X-\mu)^\mathsf{T}]$$

Notation. We often use the symbol σ^2 to denote the variance of a scalar random variable, or the symbol Σ for a matrix random variable. ■

A.5 Entropy

The entropy of a probability distribution is a function that describes the expected information of a given outcome. For a distribution p, the entropy is

$$H_p(x) = \mathcal{E}[-\log_2 p(x)]$$

where $-\log_2 p(x)$ is the number of bits needed to encode the outcome x using an optimal encoding. The entropy describes where a robot may want to explore to gain more information (by moving to areas with high entropy).

A.6 Density of a transformation

Given an n-dimensional random variable x with density f, the density g of $y = h(x)$ is

$$g(y) = f(h^{-1}(y)) \left| \det \frac{\partial h^{-1}(y)}{\partial y} \right|$$

A.7 Inverse-Variance Weighting

Two scalar measurements

Suppose we want to measure a scalar quantity whose true value is μ. To approximate the quantity, we take two measurements z_1 and z_2, which are random variables. Assume that

- the measurements are unbiased, meaning that $\mathcal{E}(z_i) = \mu$
- the measurements are uncorrelated, meaning that $\mathbf{cov}(z_1, z_2) = 0$
- the variance of each measurement is $\mathrm{var}(z_i) = \sigma_i^2$

To obtain a better estimate, we could take a weighted linear combination of the measurements:

$$\hat{z} = w_1 z_1 + w_2 z_2$$

But how should we choose the weights? The fused estimate $\hat{z}$ is also a random variable. Let's first compute its expected value. Using that the expectation is linear and that the estimates are unbiased,

$$\mathcal{E}(\hat{z}) = \mathcal{E}(w_1 z_1 + w_2 z_2) = (w_1 + w_2)\mu$$

For the fused estimate to also be unbiased, the weights must sum to one, $w_1 + w_2 = 1$. This gives one equation that the weights must satisfy, but we need another equation since there are two weights. We want to choose the weights such that they minimize the variance of the estimate, which is given by

$$\begin{aligned}
\mathrm{var}(\hat{z}) &= \mathcal{E}[(\hat{z} - \mu)^2] \\
&= \mathcal{E}[(w_1(z_1 - \mu) + w_2(z_2 - \mu))^2] \\
&= w_1^2\,\mathcal{E}[(z_1 - \mu)^2] + 2w_1 w_2\,\mathcal{E}[(z_1 - \mu)(z_2 - \mu)] + w_2^2\,\mathcal{E}[(z_2 - \mu)^2] \\
&= w_1^2\sigma_1^2 + w_2^2\sigma_2^2
\end{aligned}$$

To write this as a function of a single variable, we can use that the weights must sum to one.

$$\mathrm{var}(\hat{z}) = w_1^2\sigma_1^2 + (1 - w_1)^2\sigma_2^2$$

To minimize the variance, we take the derivative and set it equal to zero:

$$0 = \frac{\mathrm{d}}{\mathrm{d}w_1}\mathrm{var}(\hat{z}) = 2w_1\sigma_1^2 - 2\,(1 - w_1)\sigma_2^2$$

which implies that the optimal weights are

$$w_1 = \frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2} \qquad \text{and} \qquad w_2 = \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2}$$

Using these optimal weights, the fused estimate is

$$\hat{z} = \frac{\sigma_2^2\, z_1 + \sigma_1^2\, z_2}{\sigma_1^2 + \sigma_2^2}$$

The variance of the optimal estimate is

$$\mathrm{var}(\hat{z}) = \frac{\sigma_1^2\,\sigma_2^2}{\sigma_1^2 + \sigma_2^2}$$

which is better than either of the individual measurements.

A.8 Estimation

Suppose we are given the value of a random variable z and we want to estimate the value of another random variable s. The likelihood is the conditional probability distribution of z given s, viewed as a function of s,

$$s \mapsto p(z \mid s)$$

The maximum likelihood estimate (MLE) is the value of s for which the likelihood is maximized,

$$\hat{s}_{\text{MLE}} = \arg\max_s \; p(z \mid s)$$

Now suppose we have prior knowledge about the distribution of s. We can then use Bayes rule to write the probability of s given z as

$$p(s \mid z) = \frac{p(z \mid s)\, p(s)}{p(z)}$$

The maximum a posteriori (MAP) estimate is the mode of this conditional probability distribution, which is the state s that is most likely given the measurement z. Since the denominator $p(z)$ is constant with respect to s, the MAP estimate is

$$\hat{s}_{\text{MAP}} = \arg\max_s \; p(z \mid s)\, p(s)$$

Comments

- The MAP estimate is the same as the MLE when the prior is uniform.
- Since the logarithm is a monotonically increasing function, we can take the log of the objective without changing the optimal solution. Therefore,

$$\hat{s}_{\text{MAP}} \quad = \quad \arg\max_s \; \underbrace{\log p(z \mid s)}_{\substack{\text{measurement}\\ \text{model}}} \quad + \quad \underbrace{\log p(s)}_{\text{prior}}$$

A.9 Information theory

The *information* associated with a random event is a quantity that describes the amount of information associated with an observation of the event.

Definition

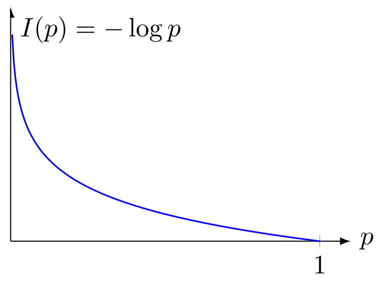
For an event with probability p, the information $I(p)$ is characterized by following three properties:

a) $I(p)$ is monotonically decreasing in the probability p, so an increase in the probability of an event decreases the information from observing the event and vice versa.

b) $I(1) = 0$, so events that always occur provide no information.

c) $I(p_1 p_2) = I(p_1) + I(p_2)$, so the information from independent events is the sum of the information from each event.

The only choice for the information function that satisfies these properties is the negative logarithm of the probability, whose graph is the following:



Example (Information of a coin flip)**.** Consider flipping a coin with probability p of landing on heads and probability $1 - p$ of landing on tails. The information associated with each event is

$$\begin{aligned} I(\text{heads}) &= -\log p \\ I(\text{tails}) &= -\log(1-p) \end{aligned}$$

When $p = 0.5$, both events have the same probability and therefore provide the same amount of information. On the other extreme, suppose that $p = 0$. Then the information associated with observing heads is infinite (since this event has zero probability of occuring) while observing tails has information zero (since this event occurs with probability one). In general, events with low probability provide high information, and vice versa.

Remark. While we could use any base for the logarithm in the definition of the information, base two is often used since the information is then the number of bits required to encode the random variable. ■

We can view the information as a function of the probability p, or of the underlying random variable x whose probability is $p(x)$, that is,

$$I(x) = -\log p(x)$$

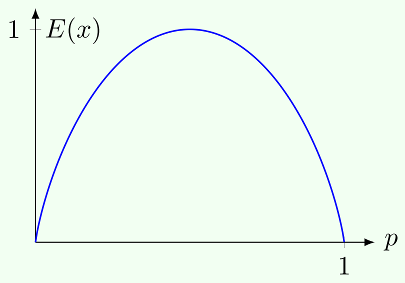
The *entropy* associated with a random variable is the expected information,

$$E(x) = \mathcal{E}[I(x)] = \mathcal{E}[-\log p(x)]$$

Example (Entropy of a coin flip). Going back to our previous example, the entropy associated with the random variable x of flipping the coin is

$$E(x) = -p \log p - (1-p)\log(1-p)$$

where p is the probability of heads and $1-p$ that of tails. The maximum entropy is one bit, which occurs when $p = 0.5$. At the extreme when $p = 0$ or $p = 1$, the entropy is zero since there is no expected information obtained from flipping the coin.



Gaussian random variable

We now describe the information associated with a Gaussian random variable. Recall that the probability density function of a Gaussian random variable with mean μ and covariance Σ is

$$p(x) = \det(2\pi\Sigma)^{-\frac{1}{2}} \exp\left(-\tfrac{1}{2}(x-\mu)^\mathsf{T}\Sigma^{-1}(x-\mu)\right)$$

Expanding the exponential, we have

$$p(x) = \underbrace{\det(2\pi\Sigma)^{-\frac{1}{2}} \exp\left(-\tfrac{1}{2}\mu^\mathsf{T}\Sigma^{-1}\mu\right)}_{\text{constant}} \exp\left(-\tfrac{1}{2}x^\mathsf{T}\Sigma^{-1}x + x^\mathsf{T}\Sigma^{-1}\mu\right)$$

This motivates the canonical parameterization of a Gaussian random variable, which is

$$p(x) = \eta \exp\left(-\tfrac{1}{2}x^\mathsf{T}\Omega x + x^\mathsf{T}\xi\right)$$

where $\Omega = \Sigma^{-1}$ is the *information matrix* and $\xi = \Sigma^{-1}\mu$ is the *information vector*. This is an equivalent parameterization of a Gaussian, and we can recover the mean and covariance from the information matrix and information vector as $\mu = \Omega^{-1}\xi$ and $\Sigma = \Omega^{-1}$.

Using the definition of information, the information of a Gaussian random variable (in information form) is

$$I(x) = \tfrac{1}{2}x^\mathsf{T}\Omega x - x^\mathsf{T}\xi + \text{constant}$$

where the information matrix Ω is the quadratic term and the information vector ξ is the (negative of the) linear term. We can also interpret this as a quadratic centered at the mean μ with curvature Ω,

$$I(x) = \tfrac{1}{2}(x-\mu)^\mathsf{T}\Omega(x-\mu) + \text{constant}$$

Both the covariance matrix Σ and information matrix Ω are positive definite, meaning that all of their eigenvalues are positive. Therefore, the information is an upward-facing quadratic function of the random variable x, and the quadratic is centered about the mean μ. So observing the mean contains the least amount of information, while observing measurements further from the mean have more information.

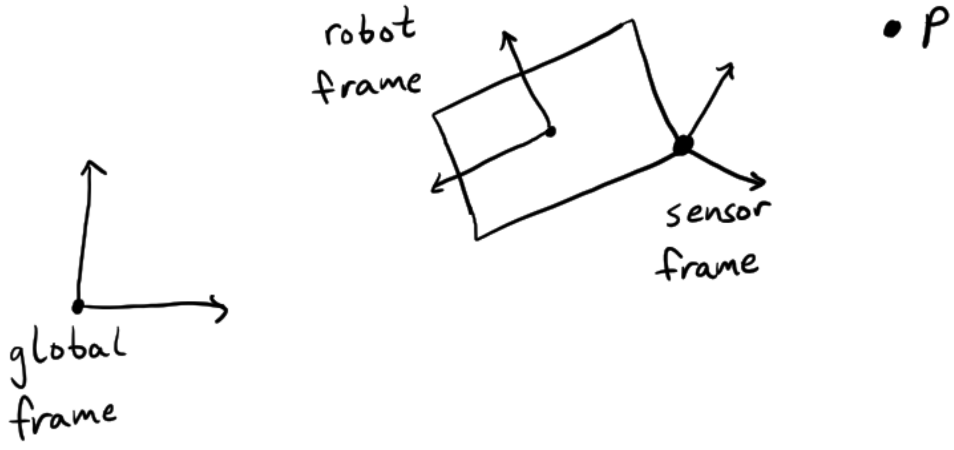
B

Coordinates and Transformations

Mobile robots take measurements relative to their current pose while moving in their environment. To fuse measurements from various locations, we must transform these local measurements to a global reference frame.

B.1 Motivation: Sensor measurements in global coordinates

Consider a robot located at some known position and orientation with respect to a global reference frame. Suppose there is a sensor mounted on the robot at some location and orientation with respect to the robot. If the sensor measures a point p relative to itself (the sensor does not know where it is in the world), then what is the coordinate of the point in the global reference frame?



Let (r_x, r_y, r_θ) denote the pose of the robot with respect to the global reference frame, and let (s_x, s_y, s_θ) denote the pose of the sensor with respect to the robot reference frame. Then a point $p = (p_x, p_y)$ in the sensor reference frame has coordinates $z = (z_x, z_y)$ in the global reference frame, where

$$\begin{bmatrix} z_x \\ z_y \end{bmatrix} = \begin{bmatrix} r_x \\ r_y \end{bmatrix} + \begin{bmatrix} \cos r_\theta & -\sin r_\theta \\ \sin r_\theta & \cos r_\theta \end{bmatrix} \begin{bmatrix} s_x \\ s_y \end{bmatrix} + \begin{bmatrix} \cos(r_\theta + s_\theta) & -\sin(r_\theta + s_\theta) \\ \sin(r_\theta + s_\theta) & \cos(r_\theta + s_\theta) \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix}$$

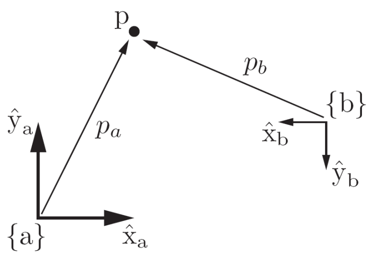
In this chapter, we will learn how to derive this expression as well as more general coordinate transformations.

B.2 Points, vectors, and reference frames

In Euclidean space, a reference frame is a coordinate system defined by a reference point at the origin of the frame and a reference point at unit distance along each of the coordinate axes. Typically reference frames used in robotics are a global reference frame (such as the corner of the room or the location of a fixed AprilTag), a body frame attached to the robot, and sensor frames attached to individual sensors on the robot.

Remark. All frames that we use are *stationary* and *intertial*. When we refer to a reference frame attached to a moving rigid body, what we actually mean is a motionless frame that is instantaneously coincident with a frame that is fixed to the (possibly moving) rigid body. This is in contrast to *dynamics* where a non-inertial reference frame is attached to a rotating rigid body. ■

A point p in physical space can be represented in relation to a reference frame by specifying it as a linear combination of the coordinate axes from the origin of the frame. Keep in mind that this is just a representation of the point, but this representation depends on the particular reference frame. The same point p has different representations in different frames.


p
p_b
{b}
$\hat{\text{y}}_\text{a}$
p_a
$\hat{\text{x}}_\text{b}$
$\hat{\text{y}}_\text{b}$
{a}
$\hat{\text{x}}_\text{a}$

In this example, the point p is represented as $p_a = (1, 2)$ in frame a while it has representation $p_b = (4, -2)$ in frame b.

We can use a point in some reference frame to describe a vector, which is an arrow from the origin of the reference frame to the point. To describe a vector without relying on explicit coordinates, we can use the difference between two points in which case the vector $v = p_2 - p_1$ points from p_1 to p_2.

B.3 Rotations

Consider two reference frames a and b. We can represent the origin of b with respect to a by a point p. But how do we represent the orientation of the frame? We can specify the orientation of a rigid body in the plane by a single angle θ. While this is sufficient in two dimensions, more angles are needed to describe orientation in three-dimensional space. To see how this generalizes, consider representing the directions of the coordinate axes $(\hat{x}_b, \hat{y}_b)$ of frame b in terms of the coordinate axes $(\hat{x}_a, \hat{y}_a)$ of frame a.

$$\begin{bmatrix} \hat{x}_b \\ \hat{y}_b \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \hat{x}_a \\ \hat{y}_a \end{bmatrix}$$

Therefore, the orientation of a frame in two dimensions can be represented by the above matrix, which is called a rotation matrix. The columns of a rotation matrix are unit vectors (since the coordinate axes $(\hat{x}_b, \hat{y}_b)$ are unit vectors), are orthogonal (since the coordiante axes are orthogonal), and have unit determinant (since the reference frame is assumed to be right-handed). These conditions yield the following general definition of a rotation matrix.

Definition (Special orthogonal group)**.** The special orthogonal group $SO(n)$, also known as the group of rotation matrices, is the set of all $n \times n$ real orthogonal matrices R with unit determinant.

$$SO(n) = \{R \in \mathbb{R}^{n\times n} \mid R^{\mathsf{T}} R = I \text{ and } \det R = 1\}$$

Planar rotations

In two dimensions, a rotation matrix always has the form

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

for some angle θ, which represents a counterclockwise rotation by θ radians in the plane. Planar rotations have the following properties:

- Rotation by an angle θ_1 followed by a rotation θ_2 is equivalent to a rotation by the sum $\theta_1 + \theta_2$,

$$R(\theta_1)\, R(\theta_2) = R(\theta_1 + \theta_2)$$

- Rotation by an angle of zero leaves points unchanged:

$$R(0) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- The inverse of a rotation represents a *clockwise* rotation in the plane. Since a counterclockwise rotation of θ is equivalent to a clockwise rotation of $-\theta$, we have that

$$R(-\theta) = R(\theta)^{-1} = R(\theta)^{\mathsf{T}} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

All of these relationships may be easily verified by directly substituting the definition of the rotation matrix and using standard trigonometric identities. The last property also follows from the fact that a rotation matrix is orthogonal, which means that its inverse is equal to its transpose.

Group structure

As their name suggests, the set of rotation matrices form a mathematical group (the special orthogonal group). A group is a set of elements along with a group operation (denoted here by $\cdot$) such that, for any elements A and B in the group, the following properties are satisfied:

- **closure:** $A \cdot B$ is also in the group
- **associativity:** $(A \cdot B) \cdot C = A \cdot (B \cdot C)$
- **identity element:** there exists an identity element I in the group such that $A \cdot I = I \cdot A = A$
- **inverse element:** there exists an element A^{-1} in the group such that $A^{-1} \cdot A = A \cdot A^{-1} = I$

For the special orthogonal group, elements of the group are rotations and the group operation is matrix multiplication.

For rotations in the plane, closure is due to the fact that two consecutive rotations is equivalent to a single rotation by the sum of the angles, associativity is due to this along with associativity of addition, the identity element is a rotation by angle zero (in which case R is the identity matrix), and the inverse of a rotation is a rotation in the opposite direction.

Uses

Rotation matrices may be used in various contexts. A rotation matrix may be used for the following:

a) To represent the orientation of a reference frame relative to another frame

A rotation R_{ab} may be used to represent the orientation of reference frame b with respect to frame a. We may also simply write R_b when a is the global reference frame. The orientation of frame a with respect to frame b is given by the inverse of the rotation,

$$R_{ba} = R_{ab}^{-1}$$

b) To change the orientation of a reference frame

We may also interpret a rotation R_{ab} as an operator that changes the orientation R_{bc} of frame c with respect to b to the orientation R_{ac} of frame c with respect to a. This change of orientation is described by

$$R_{ac} = R_{ab} R_{bc}$$

where the middle frame b "cancels out".

c) To rotate a vector or a frame

We can also use a rotation to describe the rotation of a vector or a frame. To emphasize this description as a rotation instead of an orientation, we use the notation

$$R = \mathrm{Rot}(\hat{\omega}, \theta)$$

to describe a rotation about the unit vector $\hat{\omega}$ by an amount θ (any rotation in two and three dimensions can be represented in this way). Such a rotation is given as follows, depending on the frame in which the unit vector is represented:

$$\mathrm{Rot}(\hat{\omega}_a, \theta) R_{ab} \qquad \text{or} \qquad R_{ab}\, \mathrm{Rot}(\hat{\omega}_b, \theta)$$

We can also rotate a vector v about a unit vector $\hat{\omega}$ to obtain another vector v', all represented in the same reference frame, as

$$v' = \mathrm{Rot}(\hat{\omega}, \theta)\, v$$

B.4 Rigid body motions

We now consider both the orientation and position (that is, the pose or configuration) of a rigid body. We could use a rotation matrix $R \in SO(n)$ to represent the orientation and a point $p \in \mathbb{R}^n$ to represent the position of a rigid body with respect to some reference frame. To simplify the notation, we combine both of these into the following single object.

Definition (Special Euclidean group). The special Euclidean group $SE(n)$, also known as the group of rigid body motions or homogeneous transformation matrices in $\mathbb{R}^n$, is the set of pairs of rotations and points,

$$SE(n) = SO(n) \times \mathbb{R}^n = \{(R, p) \mid R \in SO(n),\ p \in \mathbb{R}^n\}$$

We use $T = (R, p)$ to denote rigid body motion in $SE(n)$, where R is a rotation in $SO(n)$ and p is a point in $\mathbb{R}^n$.

Homogeneous coordinates and transformations

Homogeneous coordinates are a convenient way to represent points and vectors, and homogeneous matrices represent their transformations.

Consider a point $p \in \mathbb{R}^n$ in n-dimensional space. In homogeneous coordinates, this point is represented by the $(n+1)$-dimensional vector $\left[\begin{smallmatrix} p \\ 1 \end{smallmatrix}\right]$. Homogeneous coordinates are always one dimension larger than the standard coordinates. This extra dimension will allow us to represent transformations in a compact and simple manner.

Since a vector $v \in \mathbb{R}^n$ is the difference between two points, its homogeneous representation is $\left[\begin{smallmatrix} v \\ 0 \end{smallmatrix}\right]$.

The homogeneous form of vectors and points is different: points append a one while vectors append a zero. This convention will reinforce several rules:

- sums and differences of vectors are vectors
- the sum of a vector and a point is a point
- the difference between two points is a vector
- the sum of two points is meaningless

To transform a homogeneous coordinate back to a normal coordinate, first scale it so that the last component is one, and then extract the top part to obtain (x, y). For example, the homogeneous coordinate (x, y, z) corresponds to the standard coordinate $(x/z, y/z)$. However, all transformations that we will do preserve the one in the last component, so this extra scaling is not necessary; you can simply read off the x and y values as the first two components of the homogeneous coordinate.

We can associate an element $T \in SE(n)$ with its homogeneous transformation matrix, which is the $(n+1) \times (n+1)$ real matrix

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

For planar rigid body motion, a homogeneous transformation matrix has the form

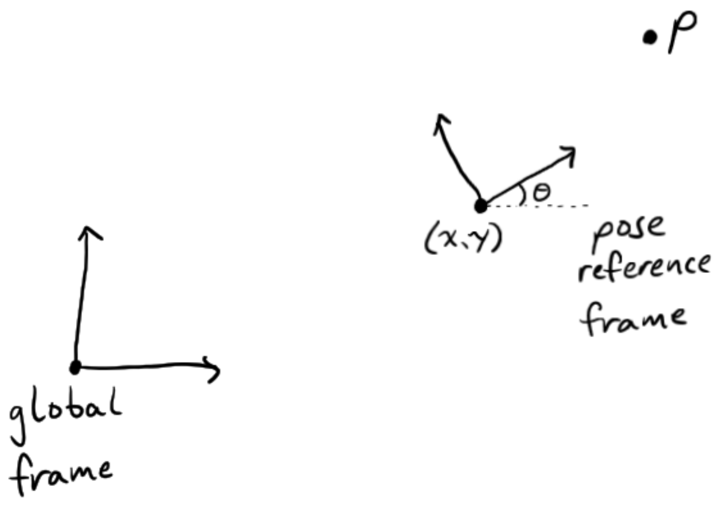
$$T = \begin{bmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1 \end{bmatrix}$$

where (x, y) is the position and θ the orientation of the rigid body.

The homogeneous transformation matrix T maps homogeneous coordinates in the pose frame to homogeneous coordinates in the global frame.

$$T: \quad \text{pose frame} \quad \mapsto \quad \text{global frame} \quad \text{(homogeneous coordinates)}$$

Given a point p in the pose frame, its corresponding homogeneous coordinate in the global frame is $T \left[\begin{smallmatrix} p \\ 1 \end{smallmatrix}\right]$.



Properties

- The inverse of a transformation matrix is also a transformation matrix.

$$T^{-1} = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} R^\top & -R^\top p \\ 0 & 1 \end{bmatrix} \in SE(n)$$

- The product of two transformation matrices is also a transformation matrix.

$$T_1 T_2 = \begin{bmatrix} R_1 & p_1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_2 & p_2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_1 R_2 & R_1 p_2 + p_1 \\ 0 & 1 \end{bmatrix} \in SE(n)$$

- Multiplication of tranformation matrices is associative, meaning that $(T_1T_2)T_3 = T_1(T_2T_3)$, but generally not commutative, meaning that $T_1T_2 \neq T_2T_1$.
- For any transformation matrix $T \in SE(n)$ and any points $x, y, z \in \mathbb{R}^n$,
  - **preservation of distances between points:** $\|Tx - Ty\| = \|x - y\|$
  - **preservation of angles between vectors:** $\langle Tx - Tz, Ty - Tz\rangle = \langle x - z, y - z\rangle$

  where $\langle \cdot, \cdot \rangle$ and $\| \cdot \|$ denote the standard Euclidean inner product and norm in $\mathbb{R}^n$.

This last point establishes that elements of $SE(n)$ do in fact represent rigid body motions in that they preserve angles between vectors and distances between points.

Uses

Just as rotation matrices may be used in various contexts, we can use an element $T \in SE(n)$ in the following ways:

a) To represent the pose of a reference frame relative to another frame
A rigid body motion $T_{ab} = (R_{ab}, p_{ab})$ may be used to represent the pose of reference frame b with respect to frame a, where R_{ab} and p_{ab} are the orientation and position of frame b represented in frame a. We may also simply write T_b when a is the global reference frame. The pose of frame a with respect to frame b is given by the inverse of the motion,

$$T_{ba} = T_{ab}^{-1}$$

b) To change the frame in which a vector or frame is represented
Similar to the subscript cancellation rule for rotations, a rigid body motion T_{ab} may be used to change the frame in which a frame is represented:

$$T_{ac} = T_{ab}T_{bc}$$

Similarly, for a vector v_b represented in b, its representation in a is

$$v_a = T_{ab}v_b$$

c) To displace (rotate and translate) a vector or frame
We can also use a rigid body motion to describe the displacement of a vector or a frame. To emphasize this description as a rotation and translation, we use the notation

$$T = \mathrm{Trans}(p)\,\mathrm{Rot}(\hat{\omega},\theta) \qquad \text{where} \qquad \mathrm{Rot}(\hat{\omega},\theta) = \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix} \qquad \text{and} \qquad \mathrm{Trans}(p) = \begin{bmatrix} I & p \\ 0 & 1 \end{bmatrix}$$

Such a displacement is given as follows, depending on the frame in which the unit vector $\hat{\omega}$ and point p are represented:

$$TT_{ab} = \mathrm{Trans}(p_a)\,\mathrm{Rot}(\hat{\omega}_a,\theta)T_{ab} \qquad \text{or} \qquad T_{ab}T = T_{ab}\,\mathrm{Trans}(p_b)\,\mathrm{Rot}(\hat{\omega}_b,\theta)$$

We can also displace a vector v about a unit vector $\hat{\omega}$ and point p to obtain another vector v', all represented in the same reference frame, as

$$v' = Tv = \mathrm{Trans}(p)\,\mathrm{Rot}(\hat{\omega},\theta)\,v$$

In the first case, T represents the position and orientation of a reference frame (such as one placed on a rigid body), while in the last two cases T represents an operator that either changes the reference frame or moves a vector or frame.

Example (relative odometry measurements). Suppose the robot measures the odometry z_{ij} from pose x_i to pose x_j with information matrix Ω_{ij}. Let Z_{ij}, X_i, and X_j denote the corresponding homogeneous transformation matrices. The frames that these matrices map between is as follows:

$$\begin{aligned} X_i &: \text{pose } i \mapsto \text{global} \\ X_j &: \text{pose } j \mapsto \text{global} \\ Z_{ij} &: \text{pose } i \mapsto \text{pose } j \end{aligned}$$

Then the homogeneous transformation matrix $X_i^{-1}X_j$ represents how node i sees node j. The error associated with an odometry-based edge is given by the transformation matrix

$$Z_{ij}^{-1}(X_i^{-1}X_j) : \text{pose } j \mapsto \text{pose } j$$

Sensor measurements in global coordinates

We now show how to use homogeneous coordinates and transformations to map a sensor measurement p taken from a sensor on a robot to the global reference frame. Using the same notation as previously, the sensor measurement in the global frame is given by

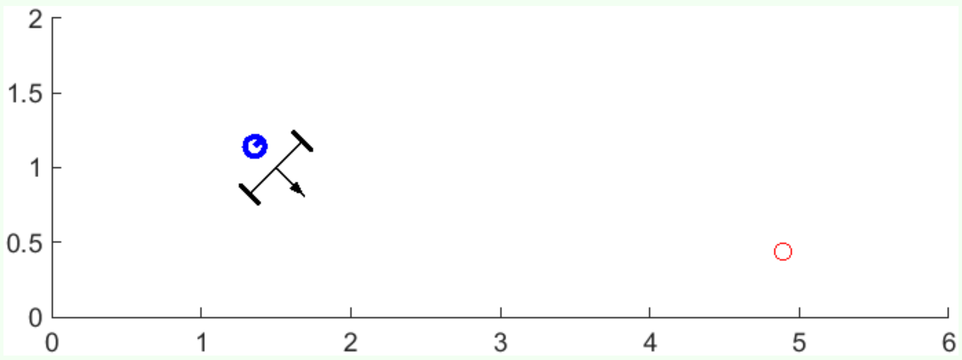
$$\underbrace{\begin{bmatrix} z_x \\ z_y \\ 1 \end{bmatrix}}_{\substack{\text{sensor} \\ \text{measurement in} \\ \text{homogeneous} \\ \text{global coordinates}}} = \underbrace{\begin{bmatrix} \cos r_\theta & -\sin r_\theta & r_x \\ \sin r_\theta & \cos r_\theta & r_y \\ 0 & 0 & 1 \end{bmatrix}}_{\substack{\text{transformation} \\ \text{from robot frame} \\ \text{to global frame}}} \underbrace{\begin{bmatrix} \cos s_\theta & -\sin s_\theta & s_x \\ \sin s_\theta & \cos s_\theta & s_y \\ 0 & 0 & 1 \end{bmatrix}}_{\substack{\text{transformation} \\ \text{from sensor frame} \\ \text{to robot frame}}} \underbrace{\begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix}}_{\substack{\text{sensor} \\ \text{measurement in} \\ \text{homogeneous} \\ \text{sensor coordinates}}}$$

Expanding this result and simplifying yields the expression shown previously.

Example. Consider the following values:

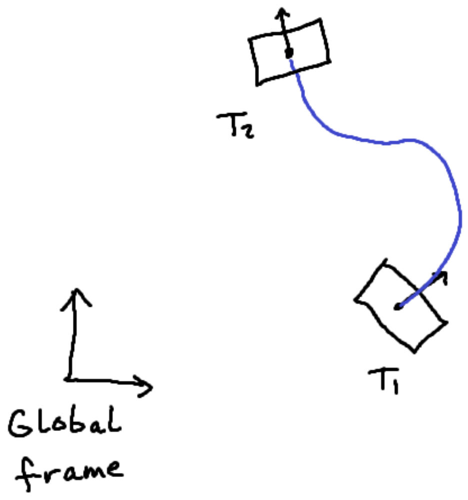
$$\begin{bmatrix} r_x \\ r_y \\ r_\theta \end{bmatrix} = \begin{bmatrix} 1.5 \\ 1 \\ -\pi/4 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} s_x \\ s_y \\ s_\theta \end{bmatrix} = \begin{bmatrix} -0.2 \\ 0 \\ \pi/2 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} 2 \\ -3 \end{bmatrix}$$

The following figure shows the robot (black), mounted sensor (blue), and measurement all in the world frame.



Relative poses

Suppose a rigid body moves from pose T_1 to T_2 (both with respect to a global reference frame) as shown below.



We can interpret T_2 as the second pose relative to the global frame and T_1^{-1} as the change of reference frame from the global frame to that of the first pose. Therefore, the second pose relative to the first pose is given by

$$T_{12} = T_1^{-1} T_2$$

Using their representation as homogeneous transformation matrices we obtain

$$T_{12} = T_1^{-1} T_2 = \begin{bmatrix} R_1^\top R_2 & R_1^\top (p_2 - p_1) \\ 0 & 1 \end{bmatrix} = \left(R_1^\top R_2,\, R_1^\top (p_2 - p_1) \right)$$

For a planar rigid body moving from pose (x_1, y_1, θ_1) to pose (x_2, y_2, θ_2), the homogeneous transformation matrix representing the second pose with respect to the first pose is

$$\begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & x_1 \\ \sin\theta_1 & \cos\theta_1 & y_1 \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & x_2 \\ \sin\theta_2 & \cos\theta_2 & y_2 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\Delta\theta & -\sin\Delta\theta & \Delta x\cos\theta_1 + \Delta y \sin\theta_1 \\ \sin\Delta\theta & \cos\Delta\theta & \Delta y\cos\theta_1 - \Delta x\sin\theta_1 \\ 0 & 0 & 1 \end{bmatrix},$$

so, the second pose with respect to the first pose is

$$\begin{bmatrix} \Delta x\cos\theta_1 + \Delta y\sin\theta_1 \\ \Delta y\cos\theta_1 - \Delta x\sin\theta_1 \\ \Delta\theta \end{bmatrix} \qquad \text{where} \qquad \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\theta \end{bmatrix} = \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \\ \theta_2 - \theta_1 \end{bmatrix}$$

B.5 Perspective-n-point problem

The perspective-n-point (PnP) problem is to estimate the relative pose between an object and a camera, given a set of correspondences between points in space and their projections on the image plane.

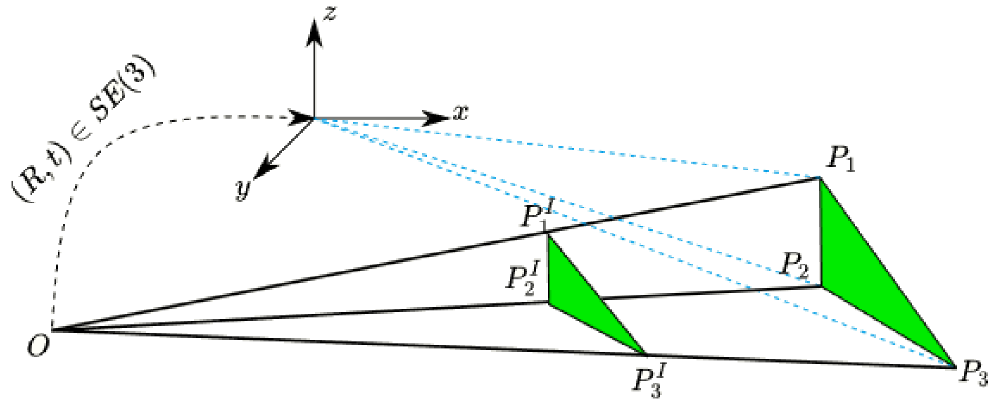
Suppose we are given a set of n known points $P_1, P_2, \ldots, P_n$ in a fixed world frame, and we use a camera to measure their projections $P_1^I, P_2^I, \ldots, P_n^I$ on the image plane. The goal is to find the coordinates $P_1^C, P_2^C, \ldots, P_n^C$ of the points in the camera frame. The coordinates in the camera frame are related to those of the world frame by a transformation:

$$P_i^C = RP_i + t$$

Our goal is to find the rotation matrix R and translation vector t so that we can map points in the world frame to points in the camera frame (or vice-versa).

For there to be a unique solution for this transformation, we need at least $n = 3$ points. This is known as the P3P problem, whereas the more general case with n points is called PnP.

The P3P problem setup is illustrated below, where O is the optical center (or center of projection) of the camera, the points P_1, P_2, P_3 are the points of interest that we know in the world frame, P_1^I, P_2^I, P_3^I are their projections onto the camera frame, and the rotation matrix R and tranlation vector t are the unknowns that map the world frame to the camera frame.[1]



[1] https://jingnanshi.com/blog/pnp_minimal.html

C

Least Squares

Least squares (LS) is an approach to approximate the solution to an over-determined system of equations by minimizing the sum of the squares of the residuals (or errors) from each individual equation.

C.1 Problem statement

Given a set of n decision variables $x_1, \ldots, x_n$ and a set of m residuals $r_1, \ldots, r_m$, each of which maps a set of decision variables to a scalar, consider the set of equations

$$\begin{aligned} r_1(x_1, \ldots, x_n) &= 0 \\ r_2(x_1, \ldots, x_n) &= 0 \\ &\vdots \\ r_m(x_1, \ldots, x_n) &= 0 \end{aligned}$$

This system of equations may have one solution, many solutions, or no solution. If the equations have no solution, then the best that we can do is find a set of variables that make the residuals as small as possible (since they cannot all be made zero). The least squares approach to this problem is to minimize the sum of the squared residuals,

$$\underset{x_1,\ldots,x_n}{\text{minimize}} \quad \sum_{j=1}^{m} r_j(x_1, \ldots, x_n)^2$$

This is called the least squares problem because it finds the values that make the squared errors as small as possible. Instead of minimizing the sum of the squared errors, we could also minimize a weighted sum of the squared errors:

$$\underset{x_1,\ldots,x_n}{\text{minimize}} \quad \sum_{j=1}^{m} w_j \, r_j(x_1, \ldots, x_n)^2$$

where w_j is the weight for the j^{th} residual. To simplify the notation, we can write the least squares problem in vector form by defining

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad \text{and} \qquad r(x) = \begin{bmatrix} r_1(x) \\ \vdots \\ r_m(x) \end{bmatrix} \qquad \text{and} \qquad W = \begin{bmatrix} w_1 & \ldots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & w_m \end{bmatrix}$$

The decision variable x is now an n-dimensional vector, the residual r is a function that maps a decision variable to an m-dimensional vector of residuals, and the weight W is an $m \times m$ diagonal matrix. The least squares problem can now equivalently be written as

$$\underset{x}{\text{minimize}}\ \ r(x)^\mathsf{T} W r(x)$$

We could also define the weighted norm $\|y\|_W = \sqrt{y^\mathsf{T} W y}$, in which case the weighted least squares problem is

$$\underset{x}{\text{minimize}}\ \ \|r(x)\|_W^2$$

The solution to this optimization problem is called the *least squares estimator* (LSE), which we denote by $\hat{x}$.

C.2 Linear least squares

Consider the case in which the residuals are each affine functions of the decision variables,

$$\begin{aligned} r_1(x_1, \ldots, x_m) &= a_{11}x_1 + a_{12}x_2 + \ldots a_{1n}x_n - b_1 \\ r_2(x_1, \ldots, x_m) &= a_{21}x_1 + a_{22}x_2 + \ldots a_{2n}x_n - b_2 \\ &\vdots \\ r_m(x_1, \ldots, x_m) &= a_{m1}x_1 + a_{m2}x_2 + \ldots a_{mn}x_n - b_m \end{aligned}$$

The scalar a_{ij} is the coefficient in the residual r_i of the decision variable x_j, and the scalar b_i is the constant coefficient in the residual r_i. The vectorized residual is then

$$r(x) = Ax - b \qquad \text{where} \qquad A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \qquad \text{and} \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

If there exists a solution x to the linear equation $Ax = b$, then the residual for this solution is zero. Even when this system of equations does not have a solution, the least squares solution minimizes the sum of the squared residuals to find an approximate solution. The least squares problem is then to find the "best" solution by solving the optimization problem

$$\underset{x}{\text{minimize}}\ \ \|Ax - b\|_W^2$$

While there is no closed-form expression for the solution to a general least squares problem, we can obtain a simple expression in the linear case. A closed-form expression for the linear least squares estimator (LLSE) is

$$\hat{x} = (A^\mathsf{T} W A)^{-1} A^\mathsf{T} W b$$

To derive this expression, we can use the following rules for vector calculus,

$$\frac{\partial}{\partial x}(x^\mathsf{T} A x) = 2Ax \qquad \text{and} \qquad \frac{\partial}{\partial x}(Ax) = A^\mathsf{T}$$

Then setting the derivative of the objective function equal to zero and applying the chain rule, we have that

$$0 = \frac{\partial}{\partial x}(Ax - b)^\mathsf{T} W (Ax - b) = 2A^\mathsf{T} W (Ax - b)$$

whose solution is the linear least squares estimator. When the weights are all one, the weight matrix is the identity and the linear least squares estimator simplifies to

$$\hat{x} = (A^\mathsf{T} A)^{-1} A^\mathsf{T} b$$

If A has linearly independent columns, then the matrix $(A^\mathsf{T} A)^{-1} A^\mathsf{T}$ is the *pseudoinverse* of A, which is denoted $A^\dagger$. This is a left inverse in that $A^\dagger A = I$, but it is not necessarily a right inverse in that $AA^\dagger \neq I$ in general. Instead of computing the inverse explicitly, a more computationally efficient way is to use the command `x = A\b` in MATLAB.

Statistical interpretation

In estimation problems, the residual is typically of the form $r(x) = z - g(x)$, where z is the measurement and $g(x)$ is the sensor model. If the model is exact, then the residuals are all zero. But sensor measurements are often noisy, which we model as additive noise,

$$z = g(x) + \delta$$

where δ is the measurement noise. The residual is then equal to the measurement noise, that is, $r(x) = \delta$. We now study the statistical properties of the least squares estimator in terms of the statistical properties of the measurement noise.

Using that the residual is the measurement noise $\delta = Ax - b$, the mean of the least squares estimator is

$$\begin{aligned}\mathcal{E}(\hat{x}) &= \mathcal{E}\big((A^\mathsf{T} W A)^{-1} A^\mathsf{T} W b\big) \\ &= \mathcal{E}\big((A^\mathsf{T} W A)^{-1} A^\mathsf{T} W (Ax - \delta)\big) \\ &= x - (A^\mathsf{T} W A)^{-1} A^\mathsf{T} W \, \mathcal{E}(\delta)\end{aligned}$$

The mean of the least squares estimator is equal to the true value if and only if the measurement noise has zero mean (so the sensor is unbiased),

$$\mathcal{E}(\hat{x}) = x \qquad \text{if and only if} \qquad \mathcal{E}(\delta) = 0$$

Assuming that the sensor is unbaised, the covariance of the linear least squares estimator is

$$\begin{aligned}\mathbf{cov}(\hat{x}) &= \mathcal{E}\big((\hat{x} - \mathcal{E}(\hat{x}))(\hat{x} - \mathcal{E}(\hat{x}))^\mathsf{T}\big) \\ &= \mathcal{E}\Big(\big((A^\mathsf{T} W A)^{-1} A^\mathsf{T} W (Ax - \delta) - x\big)\big((A^\mathsf{T} W A)^{-1} A^\mathsf{T} W (Ax - \delta) - x\big)^\mathsf{T}\Big) \\ &= \mathcal{E}\Big(\big((A^\mathsf{T} W A)^{-1} A^\mathsf{T} W \delta\big)\big((A^\mathsf{T} W A)^{-1} A^\mathsf{T} W \delta\big)^\mathsf{T}\Big) \\ &= (A^\mathsf{T} W A)^{-1} A^\mathsf{T} W \, \mathcal{E}(\delta\delta^\mathsf{T}) W^\mathsf{T} A (A^\mathsf{T} W A)^{-1}\end{aligned}$$

Therefore, the linear least squares estimate covariance for an unbiased sensor is

$$\mathbf{cov}(\hat{x}) = (A^\mathsf{T} W A)^{-1} A^\mathsf{T} W \mathbf{cov}(\delta) W^\mathsf{T} A (A^\mathsf{T} W A)^{-1}$$

It can be shown that the choice of weight matrix W that minimizes the covariance of the least squares estimate is $W = \mathbf{cov}(\delta)^{-1}$, in which case the covariance of the estimate reduces to

$$\mathbf{cov}(\hat{x}) = (A^\mathsf{T} W A)^{-1}$$

Based on this analysis, a principled choice for the weights is the inverse of the noise variance, that is, $w_j = 1/\sigma_j^2$, where σ_j^2 is the variance of the j^{th} sensor. If the sensor has a lot of noise, then its variance is large, which produces a small weight, so the measurement does not have a large effect in the optimization problem.

Example (Inverse-variance weighting). Suppose we directly take m measurements of a quantity x that we want to estimate, and each estimate z_j is perturbed by zero-mean additive Gaussian noise with variance σ_j^2. The residual is then

$$\begin{bmatrix} r_1 \\ \vdots \\ r_m \end{bmatrix} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} x - \begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix}$$

The decision variable is a scalar, while the residual is a vector. Choosing the weights as the inverse of the measurement noise variance, we have

$$A = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} 1/\sigma_1^2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 1/\sigma_m^2 \end{bmatrix}$$

The least squares estimator is then

$$\hat{x} = (A^\mathsf{T} W A)^{-1} A^\mathsf{T} W b = \left(\sum_{j=1}^{m} \frac{1}{\sigma_j^2} \right)^{-1} \sum_{j=1}^{m} \frac{z_j}{\sigma_j^2}$$

which is precisely the inverse-variance weighting solution. For instance, if there are $m = 2$ measurements, then this simplifies to

$$\hat{x} = \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 + \sigma_2^2} \left(\frac{z_1}{\sigma_1^2} + \frac{z_2}{\sigma_2^2} \right)$$

C.3 Nonlinear least squares

When the residuals in the least squares problem are not affine functions of the decision variable, there is in general no closed-form solution for the least squares estimator. Instead, we can use numerical algorithms that recursively estimate the solution.

The objective function in the least squares problem is the weighted squared norm of the residuals,

$$f(x) = \tfrac{1}{2} r(x)^\mathsf{T} W r(x) = \tfrac{1}{2} \|r(x)\|_W^2$$

which maps a decision vector x to a scalar, where we include the factor of one half for convenience (it does not change the optimal solution). To describe such numerical algorithms, we will use the gradient of the objective function, which is the vector of partial derivatives of f with respect to each of the decision variables x_i. In addition, we will use the Jacobian of the residual, which is the $m \times n$ matrix of partial derivatives of each residual function r_j with respect to each decision variable x_i.

$$J = \begin{bmatrix} \dfrac{\partial r_1}{\partial x_1} & \dfrac{\partial r_1}{\partial x_2} & \dots & \dfrac{\partial r_1}{\partial x_n} \\ \dfrac{\partial r_2}{\partial x_1} & \dfrac{\partial r_2}{\partial x_2} & \dots & \dfrac{\partial r_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial r_m}{\partial x_1} & \dfrac{\partial r_m}{\partial x_2} & \dots & \dfrac{\partial r_m}{\partial x_n} \end{bmatrix} \quad \text{and} \quad \nabla f = \begin{bmatrix} \dfrac{\partial f}{\partial x_1} \\ \dfrac{\partial f}{\partial x_2} \\ \vdots \\ \dfrac{\partial f}{\partial x_m} \end{bmatrix}$$

Each of these partial derivatives are in general functions of the decision variable, so both the gradient ∇f and the Jacobian J are functions of x. We can write the gradient of the objective function in terms of the residual and its Jacobian as

$$\nabla f(x) = J(x)^{\mathsf{T}} W\, r(x)$$

While we could apply any generic unconstrained optimization algorithm to the objective function f, there are more specialized algorithms that take into account the specific structure of the least squares problem.

Gauss–Newton algorithm

Given an initial condition x_0 for the decision variable, the Gauss–Newton algorithm solves a general (non-linear) least squares problem by iterating the recursion

$$x_{k+1} = x_k - \big(J(x_k)^{\mathsf{T}} W J(x_k)\big)^{-1} J(x_k)^{\mathsf{T}} W r(x_k)$$

The matrix J_k the Jacobian of the residual r evaluated at the current iterate x_k. Instead of inverting the matrix, it is more computationally efficient to perform the update $x_{k+1} = x_k - \Delta_k$, where the update Δ_k is given by the solution to the linear system of equations

$$\Omega_k\, \Delta_k = \xi_k$$

where

$$\Omega_k = J(x_k)^{\mathsf{T}} W J(x_k) \qquad \text{and} \qquad \xi_k = J(x_k)^{\mathsf{T}} W r(x_k)$$

Note that ξ_k is the gradient of the objective and Ω_k an approximation to the Hessian evaluated at the current iterate. A typical stopping criteria is when the update is small, that is, $\|\Delta_k\|$ is less than some tolerance.

Successive linearization

The Gauss–Newton algorithm can be derived by applying the linear least squares solution recursively to the linearization of the residual about the current iterate:

$$r(x_{k+1}) = r(x_k - \Delta_k) \approx r(x_k) - J(x_k)\Delta_k$$

To minimize the residual at the next iterate x_{k+1}, the Gauss–Newton algorithm solves the *linear* least squares problem with $A = J(x_k)$ and $b = r(x_k)$ to find the update Δ_k and then sets $x_{k+1} = x_k - \Delta_k$.

Information interpretation

We can interpret the quantities Ω_k and ξ_k in the Gauss-Newton algorithm as the information matrix and information vector of the residual linearized about the current iterate x_k. Suppose the residual $r(x)$ is a zero-mean Gaussian random variable, and recall that the information of a Gaussian random variable is a quadratic centered about the mean with curvature given by the information matrix. Therefore, if the weight matrix W is chosen optimally as the information matrix, the information of the residual is

$$I(r(x)) = \tfrac{1}{2} r(x)^{\mathsf{T}} W r(x) + \text{constant}$$

Using the linearization of the residual about the current iterate x_k, the information at the residual at the next iterate is approximately

$$\begin{aligned} I(r(x_{k+1})) &\approx I(r(x_k) - J(x_k)\Delta_k) \\ &= \tfrac{1}{2}(r(x_k) - J(x_k)\Delta_k)^{\mathsf{T}} W (r(x_k) - J(x_k)\Delta_k) + \text{constant} \\ &= \tfrac{1}{2}\Delta_k^{\mathsf{T}} \Omega_k \Delta_k - \Delta_k^{\mathsf{T}} \xi_k + \text{constant} \end{aligned}$$

Therefore, Ω_k, ξ_k, and the update $\Delta_k = \Omega_k^{-1}\xi_k$ are the information matrix, information vector, and mean of the residual at the next iterate linearized about the current iterate x_k.

Levenberg–Marquardt algorithm

The Levenberg–Marquardt algorithm is a regularized version of the Gauss–Newton algorithm. The regularization term is of the form $\lambda\,\|x - x_k\|^2$ for some nonnegative scalar λ. The only change to the Gauss–Newton algorithm is that the matrix in the linear system has an additional term due to the regularization:

$$\Omega_k = J(x_k)^\mathsf{T} W J(x_k) + \lambda I$$

where I is the identity matrix. Let's consider what happens to the algorithm in the two extreme cases for the choice of parameter λ.

- When $\lambda = 0$, the Levenberg–Marquardt algorithm reduces to the standard Gauss–Newton algorithm since the regularization term is zero.
- For large values of λ, the term λI is much larger than $J(x_k)^\mathsf{T} W J(x_k)$, so the update is approximately a scaled gradient,
$$\Delta_k \approx \frac{1}{\lambda}\nabla f(x_k)$$

To summarize, small values of λ approximate the Gauss–Newton algorithm, while large values of λ approximate gradient descent.

Alternative regularization

One disadvantage of using the regularization term $\lambda\,\|x - x_k\|^2$ is that the regularization is not scale invariant: all components of x are weighted the same no matter how they are scaled or weighted. An alternative regularization is to use the diagonal of the non-regularized matrix,

$$\Omega_k = J(x_k)^\mathsf{T} W J(x_k) + \lambda\,\operatorname{diag}\big(J(x_k)^\mathsf{T} W J(x_k)\big)$$

For this algorithm, the size of each term in the regularization is scaled based on the Jacobian and weight matrix.

Adaptive stepsize

In general, it is difficult to choose the regularization parameter λ. If the function is approximately linear so that its linearization is a good approximation globally, then we should choose λ small to approximate the Gauss–Newton algorithm. But if the function is highly nonlinear, then we cannot trust the linearization far from the current iterate so we should choose λ large to force the next iterate to be close to the current one, in which case the algorithm is approximately gradient descent.

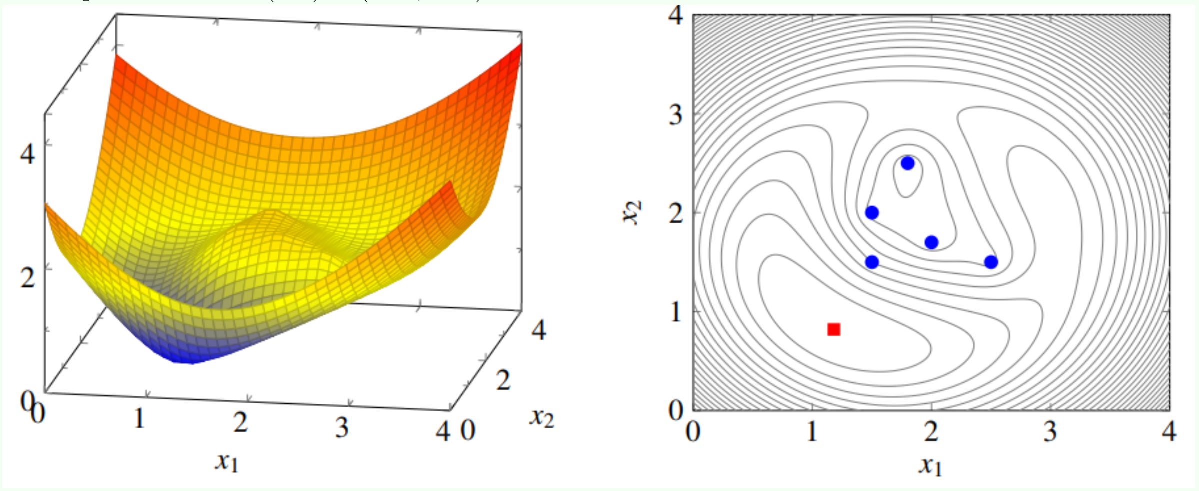
One way to choose the parameter λ is to select it adaptively as the algorithm iterates. One such scheme is as follows. Select an initial value λ_0 and a factor $\alpha \in (0, 1)$. Then at iteration k of the algorithm, first compute x_{k+1} using the current value of the parameter λ_k.

- If the value of the cost at x_{k+1} is less than the value at x_k, then accept x_{k+1} as the new iterate and decrease the parameter as $\lambda_{k+1} = \alpha\,\lambda_k$.
- Otherwise, set keep the previous iterate $x_{k+1} = x_k$ and increase the regularization parameter $\lambda_{k+1} = \lambda_k/\alpha$.

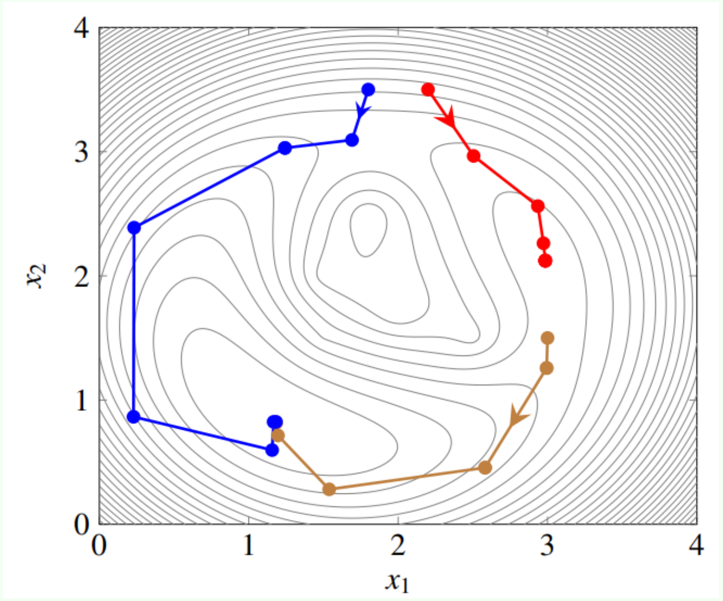
Example (location from range measurements)**.** Let (x, y) represent the unknown position of a robot in the two-dimensional plane. Suppose the robot makes range measurements $z_1, \dots, z_m$ to landmarks at known locations (ℓ_x^i, ℓ_y^i) for $i = 1, \dots, m$. The position of the robot can be estimated by solving the nonlinear least squares problem

$$\underset{x,y}{\text{minimize}} \quad \sum_{i=1}^{m} \left(z_i - \left\| \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} \ell_x^i \\ \ell_y^i \end{bmatrix} \right\| \right)^2$$

For instance, the following figure shows the norm of the residual (left) and contour lines of the squared residual (right) using five measurements (blue). The true robot position is $(1, 1)$, and the nonlinear least squares estimate (red) is $(1.18, 0.82)$.



Unlike linear least squares, nonlinear least squares problems may have multiple local minima, and the algorithm is not guaranteed to find the globally optimal solution. The figure below shows three trajectories of the algorithm starting from various initial conditions. While the blue and gold trajectories reach the global optimum, the red trajectory gets "stuck" in a local optimum.



C.4 Iterative linear least squares

Measurements are often taken sequentially in time. Instead of recomputing the least squares estimator from scratch each time a new measurement arrives, we can instead update our estimate to take into account the new measurement.

Consider the least squares estimator $\hat{x}$ with covariance matrix $\mathbf{cov}(\hat{x}) = \Sigma$ that minimizes the weighted least squares cost $\|Ax - b\|_W^2$ for some decision variable x. Now suppose that we obtain a new measurement $z = Cx + \delta$, where the measurement noise δ has zero mean $\mathcal{E}(\delta) = 0$, covariance matrix $\mathbf{cov}(\delta) = R$, and is independent of the previous measurement noise $\mathbf{cov}(\hat{x} - x, \delta) = 0$. Then the updated weighted linear least squares estimator, using the inverse covariance for the weight of the measurement, is

$$\hat{x}' = \hat{x} + K\left(z - C\hat{x}\right)$$

where the matrix K, known as the *Kalman gain*, is given by

$$K = \Sigma C^\mathsf{T}(C\Sigma C^\mathsf{T} + R)^{-1}$$

The covariance Σ' of the new estimate can be expressed in terms of the covariance Σ of the previous estimate as

$$\Sigma' = \Sigma - K\left(C\Sigma C^\mathsf{T} + R\right)K^\mathsf{T}$$

Interestingly, the covariance of the new estimate does not depend on the value z of the measurement, only its covariance R. Also, the updated covariance Σ' is “smaller” than the previous covariance Σ, indicating that the updated estimator is better than the previous one since it takes into account the additional measurement.

Derivation

Weighting the measurement by the inverse of its covariance, the weighted least squares cost of the previous residuals and the new measurement is

$$f(x) = (Ax - b)^\mathsf{T} W (Ax - b) + (z - Cx)^\mathsf{T} R^{-1}(z - Cx)$$

To find the optimal estimate, we take the derivative of the cost and set it equal to zero,

$$0 = \frac{\partial f(x)}{\partial x} = 2A^\mathsf{T} W\left(Ax - b\right) - 2C^\mathsf{T} R^{-1}\left(z - Cx\right)$$

The solution to this equation is the updated linear least squares estimator, which we denote $\hat{x}'$,

$$\hat{x}' = (C^\mathsf{T} R^{-1} C + A^\mathsf{T} W A)^{-1}(C^\mathsf{T} R^{-1} z + A^\mathsf{T} W b)$$

Since $\hat{x}$ is the linear least squares estimator for the original problem and its covariance is Σ, we have that

$$\hat{x} = (A^\mathsf{T} W A)^{-1} A^\mathsf{T} W b \qquad \text{and} \qquad \Sigma = (A^\mathsf{T} W A)^{-1}$$

Substituting these into the expression for the updated linear least squares estimate, we have that

$$\hat{x}' = \left(C^\mathsf{T} R^{-1} C + \Sigma^{-1}\right)^{-1}\left(C^\mathsf{T} R^{-1} z + \Sigma^{-1}\hat{x}\right)$$

To simplify the inverse on the left, we apply the matrix inversion lemma — also called the Woodbury matrix identity — which states that

$$\left(C^\mathsf{T} R^{-1} C + \Sigma^{-1}\right)^{-1} = \Sigma - \Sigma C^\mathsf{T}(C\Sigma C^\mathsf{T} + R)^{-1} C\Sigma$$

The updated linear least squares estimator is then

$$\hat{x}' = (\Sigma - KC\Sigma)\left(C^\mathsf{T} R^{-1} z + \Sigma^{-1}\hat{x}\right)$$

It can be shown that

$$(\Sigma - KC\Sigma)C^{\mathsf{T}}R^{-1} = K$$

which gives the final expression for the updated estimate.

Since the original estimator is unbiased and the measurement noise has zero mean, the updated estimate is also unbiased,

$$\mathcal{E}(\hat{x}') = \mathcal{E}(\hat{x}) + K\left(\mathcal{E}(z) - C\,\mathcal{E}(\hat{x})\right) = x$$

The difference between the updated estimate and its mean is then

$$\hat{x}' - x = (I - KC)(\hat{x} - x) + K\delta$$

Since the measurement noise is uncorrelated with the error of the previous estimate, $\mathbf{cov}(\hat{x} - x, \delta) = 0$. Then the covariance of the updated estimate is

$$\Sigma' = \mathbf{cov}(\hat{x}') = (I - KC)\Sigma(I - KC)^{\mathsf{T}} + KRK^{\mathsf{T}}$$

It can be shown that this is equivalent to the expression above.