Course Reader Principles of Safe Autonomy (ECE484)

Contents

Preface

Course Reader for Principles of Safe Autonomy prepared 2022-23. Eternally grateful to Pranav Sriram, Akshunna Vaishnav, Edward Guo, Arjun Ray, Chenhui Zhang, John Pohovey for editing and carefully reading the notes.

Sayan Mitra, Urbana
April 2022

1 End-to-end safety

1.1 Data and models

How to check safety of an autonomous car? The natural approach for checking any system, not just safety of autonomous cars, is through *testing*.

A single *test* for a system $\mathcal{A}$ runs the system and observes whether the resulting *execution* passes or fails some requirements. For an autonomous car, testing or generating an execution means either test driving the car on the road or running the car's software in a simulation tool. Requirements here could be, for example, that the "car does not collide" or "car does not run a stop signs".

What we can and cannot learn from tests? If a test violates a requirement $\mathcal{R}$, that can give useful information. For example, it can pinpoint the road and traffic conditions that led to the collision. It can also help identify the bug in the code or the design of the software that needs to be fixed. Such a test or an execution of the system $\mathcal{A}$ is called *a counter-example* for the requirement $\mathcal{R}$.

While a finite set of tests can be used to show that a requirement is *not* satisfied, they cannot prove that such a requirement $\mathcal{R}$ is satisfied for all executions of $\mathcal{A}$. This is because $\mathcal{A}$ can have infinitely many executions, even if it is a finite state machine. Later on we will see when and how finite number of tests can be used to say something *statistical* about the safety of $\mathcal{A}$, but for now, let us train our eyes on *absolute safety.*

"Testing can be used to show the presence of bugs, but never to show their absence!"

— Edsger W. Dijkstra

Why bother with models? To learn or extrapolate about all—infinitely many—executions of $\mathcal{A}$ form a finite sampling of executions, we need to make some *assumptions* about $\mathcal{A}$. A collection of these assumptions defines the *model* $\hat{A}$ for $\mathcal{A}$.

In this class, we will learn about many different types of models and learn how finite amount of data or tests can be used to make precise statements about safety of $\hat{A}$. We will also learn how different components of an autonomous vehicle—vision, localization, control, planning, decision making—are naturally described by wildly different types of models. In this lecture, we will start with a simple but very powerful class of models called *automata,* also known as *discrete transition systems* and *state machines.*

- A model $\hat{\mathcal{A}}$ for $\mathcal{A}$ can be *proved* to satisfy a given set of the requirements $\mathcal{R}$. This proof does not "prove" anything for $\mathcal{A}$ (because of the "model to reality gap"), but it can serve as evidence, explanation, certificate, assurance case for the safety of $\mathcal{A}$.
- The process of certification of $\hat{A}$ invariably leads to more effective testing or usage of execution data.

Going forward, when we are discussing a single model we will not distinguish between the system $\mathcal{A}$ and its model $\hat{A}$. But, in using the results from a model, it is important to mind this gap.

Roadmap

- A simple class of models: *automata*.
- What are executions of automata: sequence of states
- What are requirements?
- *Reachable states*, why we want to compute them, and why that can be hard
- *Invariants* as approximations of reachable states

1.2 Automata or state machines

Definition 1.1. A *state machine* $\mathcal{A}$ is defined by (1) a set of states Q, (2) a set of *start states* $Q_0 \subseteq Q$, and (2) a set of transitions $\mathcal{D} \subseteq Q \times Q$.

Example 1.1 nice FSA example with brake, cruise, accel □

Explicitly drawing the states and transitions can become unwieldy very quickly. So, we will use programs to describe state machines. Actually, programs *are* state machines.

Example 1.2 [Collision avoidance in 1d] Consider two cars moving on a straight road segment with initial positions $x_{20} > x_{10} > 0$, and velocities $v_{10} > v_{20} \geq 0$. Car 1 has a sensor with range $d_s > 0$ and it brakes with a deceleration $a_b > 0$. Under what assumptions on the model parameters can we show that there is no collision?

Let us proceed by first creating a (simple) model of this system. The important state variables here are the positions and the velocities of the two cars $x_1, x_2, v_1, v_2 \in \mathbb{R}$. Actually, v_2 never changes and we can make it into a constant parameter of the model instead of a state variable. So, the state space Q for this automaton $\mathcal{A}$ is $Q = \mathbb{R}^3$. For any vector $\mathbf{x} \in Q$, we can stack the three state components to be in some fixed order, say $\mathbf{x} = \langle x_1, x_2, v_1 \rangle$. Notice the state space is uncountably infinite.

Next, the initial states Q_0 of $\mathcal{A}$ is defined simply as

$$Q_0 := \{\mathbf{x} \in Q \mid \mathbf{x}.x_1 = x_{10}, \mathbf{x}.x_2 = x_{20}, \mathbf{x}.v_1 = v_{10}\}.$$

Here $|Q_0| = 1$, but in general it need not be finite or countable.

Finally, the discrete transitions $\mathcal{D} \subseteq Q \times Q$ can be described by this program:

```
 **if** $x_2 - x_1 < d_s$
  $v_1 := \max(0, v_1 - a_b)$
 **else** $v_1 := v_1$
$x_2 := x_2 + v_2$
$x_1 := x_1 + v_1$
```

What this means is that the set of transitions is

$$\begin{aligned}\mathcal{D} := \{(\mathbf{x}, \mathbf{x}') \in Q \times Q \mid & \mathbf{x}'.x_2 = \mathbf{x}.x_2 + \mathbf{x}.v_2 \wedge \mathbf{x}'.x_1 = \mathbf{x}.x_1 + \mathbf{x}'.v_1 \\ & (\mathbf{x}.x_2 - \mathbf{x}.x_1 < d_s) \Rightarrow (\mathbf{x}'.v_1 := \max(0, \mathbf{x}.v_1 - a_b)) \\ & (\mathbf{x}.x_2 - \mathbf{x}.x_1 \geq d_s) \Rightarrow (\mathbf{x}'.v_1 := \mathbf{x}.v_1)\}.\end{aligned}$$

Now we have completely defined the automaton $\mathcal{A}$ for this example. By the way, although this model is very simple, one-dimensional scenarios are commonly used for safety assurance arguments for cars and even aircraft landing ISO (2011); Fabris (2012); Perry et al. (2013). □

Determinism An automaton $\mathcal{A}$ is *deterministic* if for every state $\mathbf{x} \in Q$ there is at most one state $\mathbf{x}'$ such that $(\mathbf{x}, \mathbf{x}') \in \mathcal{D}$ is a transition. If $\mathcal{A}$ is not deterministic then it is *nondeterministic*.

The automaton in Example 1.2 is deterministic. Do you see why? For any state $\mathbf{x}$ the unique next state can be written as $\mathbf{x}' = f(\mathbf{x})$, were f the function in the code snippet shown above. This is not that realistic.

Exercise 1.1. Rewrite the code snippet to make the automaton in Example 1.2 nondeterministic. For example, you can rewrite the assignment of v_1 as to account for a range of deceleration values.

$$v_1 := \textbf{choose}\ [\max(0, v_1 - a_b - \varepsilon), \max(0, v_1 - a_b)]$$

How can we accommodate a sensor that sometimes fails? Or one that works only in some range of relative velocities? A situation where Car 2 changes its velocity arbitrarily from a range? For each of these cases, write the new $\mathcal{D}$.

1.3 Executions of automata

An *execution* for an automaton $\mathcal{A}$ is a particular run of that automaton. Mathematically, an execution is a (possibly infinite) sequence of states $q_0, q_1, \ldots,$ such that $q_0 \in Q_0$ is a start state and $(q_i, q_{i+1}) \in \mathcal{D}$ is a valid transition of $\mathcal{A}$.

Exercise 1.2. Write out example executions of all the automata introduced earlier. Write a program to plot all the four variables of the nondeterministic automaton in Exercise **??**.

A deterministic automaton with a single initial state $|Q_0| = 1$ has a "single execution". Technically, this is not exactly right because for any execution $\alpha = q_0, q_1, \ldots, q_k$ of $\mathcal{A}$, its prefix $\alpha' = q_0, q_1, \ldots, q_{k-1}$ is a also a distinct execution.

Given an execution α we may refer to the i^{th} state in α as $\alpha[i] = q_i$.

Safety and requirements. At the beginning of this lecture, and throughout this course, we use the term "safety" as a catch-all for *requirements of the system*. More precisely, a *requirement* $\mathcal{R}$ for a system $\mathcal{A}$ is a property that all executions of $\mathcal{A}$ must meet or satisfy.

Example 1.3 Some examples of requirements written in English are:

- *A car should never come within 0.5m of another car.*
- *A car should never exceed the speed-limit.*
- *A car entering an intersection should exit it within 20 seconds.*
- *A car should drive at least 28 miles per gallon (mpg) over any interval in its operating life.*

The first three are related to driving safety. The last one is about performance and environmental impact. All are requirements of the system. □

Once again, we will say that an automaton $\mathcal{A}$ (or a vehicle) meets a requirement $\mathcal{R}$ if *all its executions* satisfies the requirement. The first requirement above for our example with two cars can be written as:

$$\forall\ \alpha, \forall\ k, \alpha[k].x_2 - \alpha[k].x_1 > 0.5.$$

Notice the quantification over all executions α and over all states $\alpha[k]$ in α.

Perspective change. *To reason about all executions from a finite set of executions (or tests) we need to represent and manipulate* sets *of states and executions.*

1.4 Reachable states, safety, and invariance

To discuss sets of executions, first let us define one step transitions for automaton $\mathcal{A}$ over sets of states. For any set of states $S \subseteq Q$

$$Post(S) := \{\mathbf{x}' \in Q \mid \exists \mathbf{x} \in S, (\mathbf{x}, \mathbf{x}') \in \mathcal{D}\}.$$

Post function defines how a set of states S changes after every individual state in the set S performs 1-step transition.

Exercise 1.3. (a) Write the *Post* function for the automaton in Example 1.2 as a logical formula (as in the definition of $\mathcal{D}$).

(b) Write the *Post* function for the most general automaton you created in Exercise 1.1 as a logical formula (as in the definition of $\mathcal{D}$).

Exercise 1.4. Show that *Post*() is a monotonic function. That is, if $S_1 \subseteq S_2$ then $Post(S_1) \subseteq Post(S_2)$.

Thought experiment. For a deterministic automaton, given a single state $\mathbf{x}$, computing the next state generated by $\mathcal{D}$ is straight forward. Now that you have written a math formula for *Post*, think about how you could compute $Post(\{\mathbf{x}\})$ for any single state $\mathbf{x} \in Q$? How would you compute $Post(S)$ for a set $S \subseteq Q$? Are you assuming a particular shape for S and a particular representation? What if S is a complicated irregular set?

The above thought experiment should convince you that computing $Post(S)$ can become difficult when the S, $\mathcal{D}$ are complex. Computing the set of all executions of length k essentially involves computing $Post^k(Q_0)$ which is inductively defined as:

$$Post^k(S) = \begin{cases} S & k = 0 \\ Post(Post^{k-1}(S)) & k > 0 \end{cases} \tag{1.1}$$

Exercise 1.5. Prove that for any automaton $\mathcal{A} = \langle Q, Q_0, \mathcal{D} \rangle$ the set of states reached by any execution at the end of k transitions *equals* $Post^k(Q_0)$. *Hint. (1) To prove equality of sets $A = B$, you'd want to show $A \subseteq B$ and $B \subseteq A$. (2) Try induction on the length of the execution.*

Reachable states. A state $\mathbf{x} \in Q$ is said to be *reachable* if it is the last state of any execution. Equivalently, $Post^k(Q_0)$ is the set of states reachable after k step. Why is this important? Well, if you wanted to check that the system is safe with respect to an unsafe set $U \subseteq Q$ (Starting from an initial set Q_0) then all you would have to check is that all reachable are not in U. That is,

$$\forall\, k \geq 0, Post^k(Q_0) \cap U = \emptyset.$$

In general, computing the set of all reachable states can be hard.

We will see this in more detail later. Informally, the transition relation $\mathcal{D}$ can be complicated with lots of nondeterminism and computing $Post^k(\cdot)$ large k may not terminate.

The trick we will see next can help bypassing the problem of computing $Post^k(\cdot)$.

Invariance. The idea of invariant property or invariance is common in physics and computer science. Any property or quantity related to a system that *remains unchanged* is an invariant. For example, the total energy of a lossless system in an invariant. Conserved quantities correspond to some invariant. The sum of the angles of a polygon is an invariant under scaling and linear transformations. For an automaton $\mathcal{A}$ an invariant I is any set of states that contains all the reachable states. This is useful because, if $I \cap U = \emptyset$ then we can infer safety of $\mathcal{A}$.

We now give a simple method for checking that a set of states $I \subseteq Q$ is an invariant of an automaton $\mathcal{A}$.

Proposition 1.1. *If we can find a set $I \subseteq Q$ such that (i) $Q_0 \subseteq I$ and (ii) $Post(I) \subseteq I$, then $\forall\, k$, $Post^k(Q_0) \subseteq I$. That is, I is an invariant of $\mathcal{A}$.*

Proof. We prove this by induction on k.

For $k = 0$ (base case) $Post^0(Q_0) = Q_0$ [by Definition of $Post^k(\cdot)$]. And, $Q_0 \subseteq I$ [by (i)]. Therefore, $Post^0(Q_0) \subseteq I$.

For $k > 0$ (inductive step), we assume $Post^k(Q_0) \subseteq I$. By applying $Post(\cdot)$ to both sides and using monotonicity of $Post(\cdot)$, we get $Post(Post^k(Q_0)) \subseteq Post(I)$. That is $Post^{k+1}(Q_0) \subseteq Post(I)$. Using (ii) it follows $Post^{k+1}(Q_0) \subseteq I$. □

Proposition 1.1 implies that if we can somehow find a set (an invariant) satisfying conditions (i) and (ii), then we do not have to compute $Post^k(\cdot)$ and in addition if $I \cap U = \emptyset$ then we can happily conclude that the system is safe.

Exercise 1.6. Let us return to Example 1.2 with the candidate invariant requirement: $d := x_2 - x_1 > 0$. Is this really an invariant? Can we prove this using Proposition 1.1? What additional assumptions do we need?

Let us assume $v_{20} = 0$ for the rest of the discussion. It is easy to see that $d := x_2 - x_1 > 0$ is not an invariant. We have not said enough about the initial values of x_{10}, x_{20}, v_{10}. What if v_{10} is so large that in one step x_1 becomes $\geq x_2$.

Now, let us take a different approach to find an inductive invariant. If we can upper-bound the time that Car 1 spends *after* it detects Car 2, then we should be able to bound the total distance it travels while braking. To do this, we introduce a *timer* into our model.

```
 if $x_2 - x_1 < d_s$
   if $v_1 > a_b$
     $v_1 := v_1 - a_b$
     $timer := timer + 1$
   else $v_1 := 0$
 else $v_1 := v_1$
$x_1 := x_1 + v_1$
```

Exercise 1.7. Show that the following is an invariant using Proposition 1.1:

$$I_2 : timer + v_1/a_b \leq v_{10}/a_b.$$

Invariant I_2 is indeed an inductive invariant and it implies that $timer \leq v_0/a_b$. This implies that the total distance traveled by Car 1 after detection is at most v_{10}^2/a_b. Now we see that if we assume that the sensing distance $d_s > v_{10}^2/a_b$, then the in all executions always $x_2 > x_1$. That is, in all reachable states there is no collision.

Identifying assumptions under which the system is guaranteed to work, is a key benefit of (absolute) safety analysis. These assumptions can be used to define what are called *operating design domains (ODD).*

Summary

- Absolute safety checking boils down to showing that none of the executions of the automaton reach an unsafe set U.
- To reason about all executions of $\mathcal{A}$ we have to work for infinite sets of states.

- One way to compute infinite sets is using the *Post* operator. But, computing all executions for unbounded time can be hard.
- If we can guess an invariant $I \subseteq Q$ and if it satisfies the two conditions of Proposition 1.1, (i) $Q_0 \subseteq I$ and (ii) $Post(I) \subseteq I$, then indeed all reachable states are contained in I, i.e., $Post^k(Q_0) \subseteq I$. This give a shortcut for proving safety by checking $I \cap U = \emptyset$.
- The inavariant may contain important information about conserved quantities, and thus, may tell us *why* the system is safe.
- If our guessed set I fails the two conditions, then we can try to modify it by adding more information in it about other state variables.

1.5 Minding the gap

We conclude this module by discussing the gap—some of the assumptions in our models that may break the safety analysis.

1. *Perception.*

   (a) Sensor s detects the obstacle **iff** distance $d \leq D_{sense}$. No false-positives, no false-negatives, no probabilities. This is a very idealized model of a sensor. More realistic sensor specifications will give distances and probabilities specific to detecting particular objects like people, cars, bicycles; detection zone will be directional.

   (b) Car 2 is known to be moving with constant velocity $v_0 = 0$ from initial position x_{20}. This assumption was heavily used in the safety analysis, but it is not used the vehicle's automatic braking algorithm. What if the sensor can detect the speed of Car 2, and applies an appropriate braking force? How will the safety analysis change?

2. *No sensing, computation, actuation delay.* The time step in which $d \leq D_{sense}$ becomes smaller is exactly when the velocity starts to decrease. What if there is a delay or a reaction time?
3. *Mechanics.* Cars moving in 1-D lane with perfect discrete kinematic model for velocity and acceleration.
4. *Inherent assumptions in any automaton model.*

   (a) *Discrete time.* Each transition of $\mathcal{A}$ models advancement of time by *some discrete amount*. You can think of 1 transition step = Δ seconds. Then, $x_1(t+1) = x_1(t) + v_1(t)\Delta$. Either way, we are not allowed to talk about what happens between $[t, t+1]$.

   (b) *Atomic steps**. We consider 1 step to be the complete (atomic) execution of the program. We cannot directly talk about the states that the program visits as it executes the individual lines/statements in the code. This may become important when considering many concurrent programs in the car, e.g., *race conditions*.

2 Basic Perception: Edge Detection

Sensors convert electromagnetic radiation from the physical environment into bits. The perception subsystem converts these bits into bits that hold *meaningful information about the environment*. These bits are then used by the downstream decision and control modules to generate the actuation, for example, steering and torque.

For example, consider a lane tracking control system as in *autopilot.* A front-facing camera generates an image from the visible light reflecting off the road. The key meaningful information in this image are the lanes, which are defined by edges of the lane markers. An edge-detection algorithm can detect the lanes in that image. Using perspective transforms, some geometry, and the focal length of the camera, a perception subsystem can estimate the position and the orientation of the camera, and therefore the car, relative to the lanes. The relative position and orientation can then be used to steer and center the car between the lanes by a lane-tracking control system.

Roadmap

- Images and filters.
- Kernels and convolution for implementing filters
- Common kernels: Gaussian, Sobel, blur, etc.
- Edges, implementing edge detection with convolution

2.1 Filtering

An *image* is a two dimensional array of pixels. Here we will consider grayscale images. For a grayscale image *img*, we will denote the $(i, j)^{th}$ pixel by $img[i][j]$. For 8-bit images $img[i][j] \in \{0, 255\}$.

Filtering is an operation in which each pixel is modified as a function of its neighboring pixels. Filtering is essential for denoising and enhancing details in images. *Linear filters* are a special class where the function applied is linear.

Example 2.1 Intensity scaling by k $\texttt{Int}(k, Img)$: $\forall i, j,\ img'[i][j] = k \times img[i][j]$.
$\texttt{Shift}(s, Img)$: $\forall i,\ img'[i][j] = img[i][j-s]$, $img'[i][0] = \ldots = img'[i][s-1] = 0$.
Average over 3-neighborhood $\texttt{Avg}(3, Img)$:

$$\forall i, j,\ img'[i][j] = \frac{1}{9} \sum_{\substack{k \in \{i-1, i, i+1\} \\ \ell \in \{j-1, j, j+1\}}} img[k][\ell]$$

Notice that this function can be conveniently written as a matrix multiplication:

$$\forall i, j,\ img'[i][j] = \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix} \cdot \begin{bmatrix} img[i-1][j-1] & img[i-1][j] & img[i-1][j+1] \\ img[i][j-1] & img[i][j] & img[i][j+1] \\ img[i+1][j-1] & img[i+1][j] & img[i+1][j+1] \end{bmatrix}$$

The constant matrix $\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$ is called the *kernel* of this filter. □

2.2 Convolution

The application of a linear function (kernel) to all the pixels can be compactly represented by the *convolution* operation.

Definition 2.1. Let f be an image and g be a kernel. The output of *convolving* f with g is denoted by $f * g$,

$$(f * g)[i][j] = \sum_{k,\ell} f[i-k, j-\ell]g[k,\ell].$$

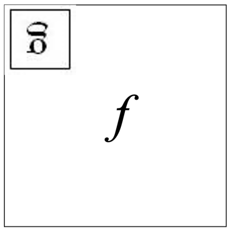


Figure 2.1
In image processing literature, the convention is to flip the convolution kernel.

For gaining intuition, we will often work with one-dimensional "images". This can also be useful for signals coming from range-finders. The output of *convolving* 1-dimensional

f with g is

$$(f * g)[i] = \sum_k f[i-k]g[k].$$

2.2.1 Properties of convolution

The convolution operation has several nice properties.

Exercise 2.1. Show that convolution has the following properties. For simplicity, assume 1-dimensional kernels and images that extend to infinity $img : [-\infty, \infty] \to \mathbb{N}$. If some parameters in the functions are missing, then show that the property holds for any choice of those parameters.

(a) *Shift invariance.* $\texttt{Shift}(Img) * g = \texttt{Shift}(Img * g)$
(b) *Linear.* $g * (Img_1 + Img_2) = g * Img_1 + g * Img_2$
(c) *Commutative.* $g * f = f * g$
(d) *Associative.* $a * (b * c) = (a * b) * c$
(e) *Distributive.* $a * (b + c) = (a * b) + (a * c)$
(f) *Identity.* For unit impulse $e = [\dots, 0, 0, 1, 0, 0, \dots]$, $a * e = a$

Proof. We provide a sample proof for (a). Without loss of generality, let the shift amount be s:

$$\texttt{Shift}(Img)[i] = Img[i+s].$$

Therefore, we have:

$$\begin{aligned}(\texttt{Shift}(Img) * g)[i] &= \sum_k \texttt{Shift}(Img)[i-k]g[k] \\ &= \sum_k Img[i+s-k]g[k] \\ &= (Img * g)[i+s] \\ &= \texttt{Shift}(Img * g)[i],\end{aligned}$$

which completes the proof. □

Any linear shift invariant operation can be written as a convolution.

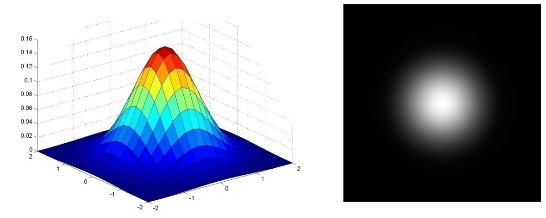
2.2.2 Kernels

Exercise 2.2. Write some example kernels for blurring, sharpening, and shifting, for two dimensional images.

The Gaussian kernel is defined by the 2-dimensional Gaussian probability distribution:

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}},$$

where $\sigma > 0$ is the standard deviation of the distribution. As a rule of thumb, the filter width is set as 3σ. The kernel is shown in Figure 2.2.



0.003	0.013	0.022	0.013	0.003
0.013	0.059	0.097	0.059	0.013
0.022	0.097	0.159	0.097	0.022
0.013	0.059	0.097	0.059	0.013
0.003	0.013	0.022	0.013	0.003

5 x 5, σ = 1

Figure 2.2
The 5×5 Gaussian kernel with $\sigma = 1$. Constant factor at front makes volume sum to 1 (can be ignored when computing the filter values, as we should renormalize weights to sum to 1 in any case).

Exercise 2.3. Show that convolving a Gaussian kernel with another Gaussian, results in a Gaussian. Show that convolving two times with Gaussian kernel with standard deviation σ is same as convolving once with $\sigma\sqrt{2}$.

The 2D Gaussian filter can be written as the application of two 1D Gaussian filters because of the *separability property* of Gaussian.

$$\begin{aligned} G_\sigma(x, y) &= \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} && (2.1)\\ &= \left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}\right)\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{y^2}{2\sigma^2}}\right) && (2.2)\end{aligned}$$

This has an immediate efficiency implication. A 2D kernel

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

The complexity of convolving an $n \times n$ image with an $m \times m$ kernel is $O(n^2 m^2)$. For separable kernels this becomes $O(n^2 m)$.

Larger Gaussian filters (i.e., filters with larger σ) will reduce more noise, but they will also blur features.

Median filter The median filter provides an alternative to the averaging and the Gaussian filter: Median over 3-neighborhood $\texttt{Median}(3, Img)$:

$$\forall i, j,\ img'[i][j] = Median(img[k][\ell] \mid k \in \{i-1, i, i+1\} \ell \in \{j-1, j, j, j+1\})$$

Exercise 2.4. Is the median filter linear? If yes, write the kernel. If no, then show why not with an example.

2.3 Edge detection

The goal of perception is to extract meaningful information about the environment, from the bits generated by sensors. The "Edges" in an image carry most of the semantic information and shape information in an image. E.g., Lanes, traffic signs, cars.

What is an *edge?* Edges are the lines along which there are sudden changes or *discontinuities* in an image. Edge detection is the problem of identifying sudden changes (discontinuities) in an image. Edges in an image may correspond to sudden changes in color, depth, surface normal, illumination, etc.

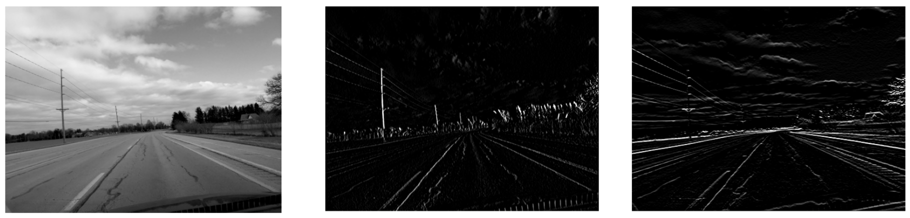


Figure 2.3
Lane detection from edges. Center image is the $\frac{\partial f}{\partial x}$ and right image is $\frac{\partial f}{\partial y}$.

2.3.1 Derivatives of images

Speaking of discontinuities, we have the definition of 1-d derivative:

$$\frac{\partial f(x)}{\partial x} = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

For a 2d image f we can define (at least) two derivatives:

$$\frac{\partial f(x,y)}{\partial x} = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon, y) - f(x,y)}{\varepsilon} \approx \frac{f(x+1,y) - f(x,y)}{1} \tag{2.3}$$

$$\frac{\partial f(x,y)}{\partial y} = \lim_{\varepsilon \to 0} \frac{f(x, y+\varepsilon) - f(x,y)}{\varepsilon} \approx \frac{f(x,y+1) - f(x,y)}{1} \tag{2.4}$$

How can we implement the approximate version of these derivatives using kernels and convolution?

$$f' := g_x * f,$$

where $g = [-1\ 1]$. Similarly, $g_y := \begin{bmatrix} -1 \\ 1 \end{bmatrix}$. There are other kernels that can compute other derivatives. See for example, the Sobel Prewitt, and Roberts filters.

2.3.2 Gradient of an image

In general, the edges may not be perfectly aligned with the axes of an image and we would be interested in the *gradient* which is defined as:

$$\nabla f := [\frac{\partial f(x,y)}{\partial x} \frac{\partial f(x,y)}{\partial y}]$$

The gradient points in the direction of most rapid increase in intensity. The gradient direction is given by $\theta = \tan^{-1}(\frac{\partial f}{\partial y} / \frac{\partial f}{\partial x})$. The direction of the edge is normal to the direction of the gradient. The edge strength is given by the gradient's magnitude (norm): $||\nabla f|| = \sqrt{(\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial y})^2}$

2.4 Differentiation and de-noising

In order to perform edge detection on a noisy image f, we would first want to apply a Gaussian filter on f and then take the derivative of the result, i.e., compute $\frac{d}{dx}(f * g)$. Since differentiation is convolution, and convolution is associative, we can instead compute $f * \frac{d}{dx}(g)$. This saves one filtering operation!

Canny Edge Detector The gradient gives a point-wise estimate of sharp changes in intensity. How can we convert these point-wise estimates to curves? For each pixel location q with the $||\nabla f(q)||$ above some threshold, check that the gradient magnitude is higher than at neighbors p and r along the direction of the gradient. Sometimes pixels along an edge may not survive this thresholding. Solution: use *hysteresis*. All of this together constitutes a very well-known edge detection algorithm called the Canny edge detector Canny (1986). With the OpenCV library you can use it as: `canny(image,th1,th2)`.

```
Input  image $f$
  Compute $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}$
  Compute $||\nabla f||$, orientation $\theta$ of $\nabla f$
```

Thin out wide ``ridges'' down **to** *single pixel width*
(Non-maximum suppression)
Define two thresholds on $||\nabla f||$: $s_1 < s_2$
Use s_2 **to** *start* **and** s_1 **to continue** *edges*

Roadmap

- Convolution is a translation invariant linear operations on signals and images.
- Kernels for different filters: blur, sharpen, differentiation
- Median filter is not linear
- Gaussian kernel is separable which makes is efficient to implement.
- Edges contain important semantic information. Edge detection uses differentiation, denoising—both implemented using convolution, and a few tricks.

3 Classification and object recognition

3.1 The classification problem

Image recognition is an example of a classification problem. Recognizing road signs, pedestrians, lanes, traffic cones, from images can be seen as a classification problem. The *classification problem* is the following.

Given a *training set* of N labeled examples $T = \{(x_1, y_1), \ldots, (x_N, y_N)\}$, we would like to learn a *prediction function* or the *classifier* $f(x) = y$. Such that the prediction error on never-before-seen examples $\{(x_s, y_s)\}_{s \in S}$, which is, $e_S = \sum_s |f(x_s) - y_s|$ is small. The set over which the trained classifier f is evaluated is called the *test set.*

For each i, x_i is the input and y_i is the label or the output. For example, x_i could be a vector representing the features in an image (defined below), or it could be the whole image, or a histogram of words appearing in a document. The y_i is the label for that input image, such as “car”, “pedestrian”, or “traffic cone”. For text documents, classification is a building block for sentiment analysis and the labels could be e.g., “politics”, “sports”, “music”. The training process typically proceeds by minimizing the error over the training set $\sum_{i \in T} |y_i - f(x_i)|$. This is called the *training error*. Of course, making training error 0 (e.g., by making f memorize all the samples in the training set) may not be such a great idea, because, then the classifier may fail miserably on the tests.

This type of classification is also called *supervised learning* as the training data set has labels. .

Roadmap

- Representing high-dimensional vectors (images) with features
- Histogram of visual words
- Clustering (k-means algorithm); see Chapter 4

- Classification (k-nearest neighbors)

3.1.1 Object recognition with image features

The "classical" (before deep neural networks became the *de facto* method) recognition pipeline required hand-crafted *features*. That is, we will have a feature map $\phi : \mathcal{X} \to \mathcal{Z}$ which would map each high-dimensional image $x \in \mathcal{X}$ to a low-dimensional vector $\phi(x) \in \mathcal{Z}$, then we would proceed with training of the classifier with the training set $\{(\phi(x_i), y_i)\}_{i\in[N]}$.

Examples of feature maps:

- Histogram of text words: Given a document x, the feature vector $\phi(x)$ is the histogram of the most popular k words in the document
- "Textons" or Texture building blocks
- Histogram of visual words
- Scale-invariant feature transforms (SIFT) Lowe (1999)
- Histogram of gradients (HOG) Dalal and Triggs (2005)

Visual words To create a histogram as a feature vector for an image, we need to (a) identify visual words and then (b) count the number of occurrences of these words. Both of these steps present some challenges.

(a) How to identify interesting image patches that can serve as visual words?
(b) How can we count the number of occurrences of the visual words? No two occurrence is going to match exactly.

To address the first problem, we sample patches of the image based on characteristic scale that are covariant with the image transformations. One possible way to do this is to convolve the image with the blob filter (derivative) at different scales and to look for the extrema of the filter response. This is the main idea behind SIFT Lowe (1999).

From all the images $\{x_i\}$ in in the entire training set T, extract all the image patches. For example, for image x_i the patches may be $\{p_{i1}, p_{i2}, \ldots, p_{ik_i}\}$, and so on. We consider each patch p_{ij} is a vector in $\mathbb{R}^n$, and then perform clustering on all of these patches (see Chapter 4 for Clustering). The resulting cluster heads or centroids are used as the visual words. . The collection of all the cluster heads define the *visual vocabulary*. Suppose the resulting cluster heads, or visual words be $\hat{p}_1, \ldots, \hat{p}_k$. The number of cluster heads (or visual words) to be chosen in the vocabulary, is a design parameter and we shall return to this later.

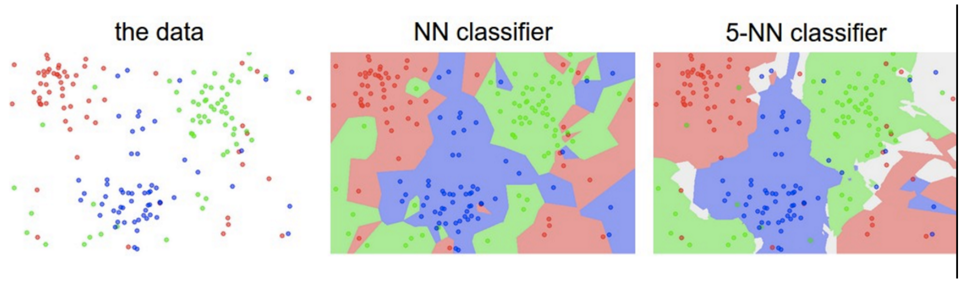


Figure 3.1
Nearest neighbor classification. Credit: Andrej Karpathy, see more at this link.

Exercise 3.1. Explain in a few sentences how visual words are obtained from a collection of images.

Quantization To address the second problem, of counting the number of occurrences of visual works in an image x_i, we have to quantize the image patches. That is, associate each patch $\{p_{i1}, p_{i2}, \ldots, p_{ik_i}\}$ of image x_i with the nearest cluster center (visual word) $\hat{p}_j$. Then we accumulate the visual word frequencies over the image for each $\hat{p}_j$. This histograms of visual words is the (lower-dimensional) feature vector $\phi(x_i)$ corresponding to the image x_i and now we can apply classification on this vector.

3.2 K-Nearest neighbor classifier

Nearest neighbor is a simple classification method. For any new data x to be classified, we simply look-up the set N_K of the K nearest training data points of x; then output the label y^* is chosen as the one corresponding to the *majority* of the labels among the points in N_K. To find the K nearest neighbors, we have to choose a distance function (discussed below). If we are using feature vectors, then, instead of computing distances on x_i we have to do so on $\phi(x_i)$.

```
Require: $T = \{(x_i, y_i\}_{i=1}^N$ training set, $K > 0$, input data $x$, $\phi$ feature map, $dist$ function
  $N_K = \min_{i \in [N]} dist(\phi(x_i), \phi(x), K)$
  for $y \in T$ do
      $C_y := \{x_i \mid x_i \in N_K, (x_i, y) \in T\}$
  end for
  $y^* := argmax_y |C_y|$
  output $y^*$
```

MITPress NewMath.cls LaTeX Book Style Size: 7x9 September 21, 2022 7:18pm

Remarks (1) The different possible choices of distance functions were discussed in the lecture. See slides.

(2) Components of the vector x_i (or $\phi(x_i)$) with large values may influence the classification more than others. To mitigate this problem, normalize vector: $z_i[j] = \frac{x_i[j]-\mu[j]}{\sigma[j]}$, where $\mu[j]$ is the mean and $\sigma[j]$ is the standard deviation of $z_\cdot[j]$ across all input samples $x_i[j]$'s.

Exercise 3.2. What are some of the choices in the design of a classifier? (Hyperparameters). For example, the number of visual words k, ... Discuss the effects of each of these parameters on efficiency and precision.

4 Clustering

4.1 Clustering problem

Clustering is a problem about discovering patterns in data. Specifically, given a data set $\{x_1, \ldots, x_N\}$ clustering requires us to find how these points are organized in groups or *clusters*. For example, clustering is used to build groups of genes with related expression patterns; in the study of social networks, is used to recognize communities within large groups of people.

Unlike the labeled training data we used in the classification problem of Chapter 3, here the data set is unlabeled. Indeed, clustering is an *unsupervised learning* technique.

Clustering problem (informal). Given $x_1, \ldots, x_N \in \mathbb{R}^n$, and $k > 1$, partition the N vectors into k groups such that the vectors in the same group are "close".

Norms. To talk about closeness ofvectors in $\mathbb{R}^n$ we need some notion of distance. Distance between vectors is defined by *norms*. The choice of the norm or the distance function will have a big impact on the shape of the clusters. Generally, a any function $f : \mathbb{R}^n \to \mathbb{R}_{\geq 0}$ satisfying these properties would be an acceptable norm:

- *homogeneous*: For any $a \in \mathbb{R}_{\geq 0}, x \in \mathbb{R}^n$, $f(ax) = af(x)$
- *triangle inequality*: For any $x, y \in \mathbb{R}$, $f(x + y) \leq f(x) + f(y)$
- *definite*: $f(x) = 0 \iff x = 0$.

Exercise 4.1. Show that the following functions are norms: (a) $f(x) := |x| = \sum_{i=1}^{k} |x[i]|$; (b) $f(x) := \|x\| = \sqrt{\sum_{i=1}^{k} x[i]^2}$; (c) $f(x) := |x|_\infty = \max_i |x[i]|$. Here we are denoting the j^{th} component of the vector $x_i \in \mathbb{R}^n$ as $x_i[j]$. That is, $x_i = (x_i[1], \ldots, x_i[j], \ldots, x_i[n])$.

Given any norm on $\mathbb{R}^n$, we can measure the distance between two vectors x_1 and $x_2 \in \mathbb{R}^n$ as $|x_1 - x_2|$.

We setup some notation to discuss the clustering problem more precisely. Consider a given set of points $\{x_1, \ldots, x_N\}$, each $x_i \in \mathbb{R}^n$. Suppose we want to group these points into

k clusters. If x_i is assigned to some cluster $j \in \{1, \ldots, k\}$ then we will set the variable c_i to be j. That is, each c_i, $i \in \{1, \ldots, N\}$ is a variable that can take values in $\{1, \ldots, k\}$. Let G_j, for $j \in \{1, \ldots, k\}$ be the set of x_i's in cluster j. That is,

$$G_j = \{x_i \mid c_i = j\}.$$

Finally, let $z_j \in \mathbb{R}^n$, $j \in \{1, \ldots, k\}$ be a representative vector for cluster j. That is, z_{c_i} is the representative vector for the cluster that x_i belongs to.

Clustering problem. Given $x_1, \ldots, x_N \in \mathbb{R}^n$, and $k > 1$, choose the cluster assignments $c_1, \ldots, c_N$ for all the points and the representatives $z_1, \ldots, z_k$, such that the following clustering cost is minimized:

$$J_{clust}(c, z) = \frac{1}{N} \sum_{i=1}^{n} |x_i - z_{c_i}|^2$$

As you can see, clustering is an optimization problem over the choice of assigning N vectors to k groups to minimize J_{clust}. There are k^N choices and in the worst case. Further, comparisons involving the points will involve computing norms which can be expensive for high-dimensional vectors (with large n). J_{clust} is a non-convex function.

4.2 K-means clustering

We will discuss one of the simplest and most well-known clustering algorithms. This k-means clustering algorithm uses the familiar Euclidean norm (2-norm).

```
Require: $\{x_i\}_{i=1}^N$, $k > 1$
  initialize $z_j$, $j \in [k]$ randomly
  while Clusters do not change do
    for $i \in \{1, \ldots N\}$ do
      $c_i := argmin_{j \in [k]} |x_i - z_j|^2$
    end for
    for $j \in \{1, \ldots k\}$ do
      $G_j := \{x_i \mid c_i = j\}$
      $z_j := \frac{1}{|G_j|} \sum_{i \in G_j} x_i$
    end for
  end while
```

This algorithm has an outer while-loop which keeps repeating until the clusters stop changing. Inside this while loop there are two steps: (1) First, each x_i is assigned to the

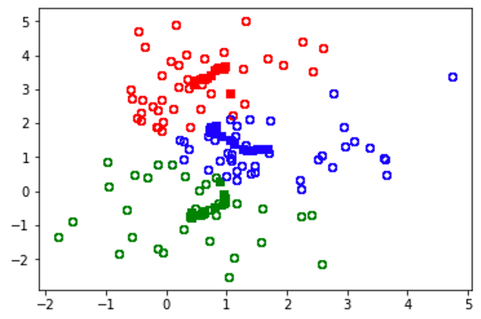


Figure 4.1
K-means clustering.

nearest z_j. That is, c_i is set to the j that minimizes $|x_i - z_j|^2$. In step (2) the representative z_j is updated to be the centroid of the cluster j, i.e., the mean of all the x_i's assigned to j. To see an online demo of k-means in action click here. Why do these two steps make sense?

It can be shown that in the inner-loop, k-means repeatedly minimizes J_{clust} with respect to x_i while holding z_j fixed, and then minimizes J_{clust} with respect to z_j while holding x_i assignments fixed. Thus, J_{clust} must monotonically decrease, and the value of J_{clust} must converge.

Since J_{clust} is nonconvex, k-means can get stuck at a local minima. If you are worried about getting stuck in a bad local minima, one common thing to do is run k-means many times (using different random initial assignments for z_j's, and then pick the clustering that gives the lowest distortion J_{clust}.

Exercise 4.2. Show an example where k-means terminates at a local minima. That is, (a) give/draw a set of points (small set $N \leq 5$, low dimension $n \leq 2$); (b) give the clustering c_i, z_j; (c) show that k-means terminates with this clustering and the corresponding cost J_{clust}. (d) Then show the globally minimal clustering and the corresponding cost J^*_{clust}.

Expand on this discussion

Exercise 4.3. Implement KNN-classification on different datasets, different values of K. Share the code with us using co-lab.

Summary

- What is the clustering problem
- k-means algorithm as coordinate decent
- Termination, local minima

5 Control

5.1 The control problem

In this chapter, we will study the control module, which sits between higher-level decisions (e.g., change lanes, slow down) and the lower-level control and actuation (e.g., steering and throttle). *Control theory* is the *art* of making *things* do what *you want* them to do. In this class, each of these words have a specific meaning. Art describes the creation of parameterized controllers or algorithms and ways of tuning them. By things we mean phenomena that can be represented using differential equations. And, we want them to follow some desired behavior, track some set-point, or follow a reference trajectory.

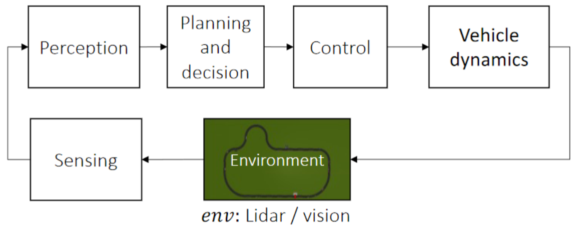


Figure 5.1
Control in the autonomy pipeline.

Examples of control systems:

- Thermostat-heating system: the temperature is kept within a *desirable range*, despite the changes in occupancy and weather conditions.
- A toilet flush: water-level maintained after use.
- Cruise control: The car maintains a set *reference speed*, despite changes in slope, road conditions.
- Autosteer: The car steers to track the center of the lane markings.

MITPress NewMath.cls LaTeX Book Style Size: 7x9 September 21, 2022 7:18pm

Figure 5.2
Many applications of control theory.

Roadmap

- Modeling the control problem using Differential Equations
  - Solutions and their properties
- Control design
  - Open vs. closed loop control
  - PID control design
  - State feedback
- Requirements
  - Stability
  - Lyapunov theory and its relation to invariance

5.2 Differential Equation Models

A control system can be modeled using two components: a *plant* and a *controller*. The plant is the thing or physical process being controlled, and the controller is the algorithm or the piece of software doing the controlling (Figure 5.3). Here, $x \in \mathbb{R}^n$ is the n-dimensional state vector, $u \in \mathbb{R}^m$ is the control input, and $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ is the function describing how the state evolves with control input. We write x_t, u_t, etc. to denote the state and control at time $t \in \mathbb{N}$.

$$x_{t+1} = f(x_t, u_t).$$

The control u is typically computed as a function of the state, $u_t = g(x_t)$, in which case, the *closed loop system* is written as

$$x_{t+1} = f(x_t, g(x_t)).$$

The continuous time version of this model is written using *ordinary differential equations (ODE)*:

$$\dot{x}(t) = f(x(t), u(t)), or \tag{5.1}$$

$$\dot{x}(t) = f'(x(t)) \tag{5.2}$$

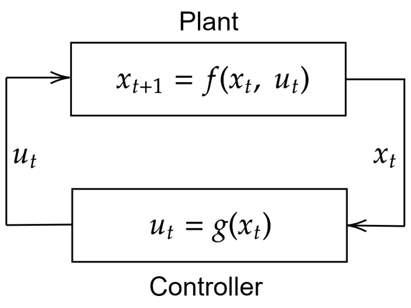


Figure 5.3
A discrete time model of a control system.

Here $\dot{x}$ is a shorthand for $\frac{dx(t)}{dt}$, i.e., the derivative of the signal $x(t)$ taken with respect to time t.

Equation (5.2) combines f and g into a new function $f' : \mathbb{R}^n \to \mathbb{R}^n$ which does not have the dependence on inputs. Such ODEs are called *autonomous system*[1].

A *solution* to the ordinary differential Equation (5.2), is a any function $x : \mathbb{R} \to \mathbb{R}^n$ such that:

$$\frac{dx(t)}{dt} = f(x(t)).$$

For this definition of solution to make sense, the function x should be differentiable with respect to time.

If the control input $u(t)$ is not continuous, then the solution $x(t)$ will not be differentiable (at the points of discontinuity). See Figure 5.4. We have to be careful about the definition of solution.

When do solutions exist? Are the solutions unique? The above discussion tells us that we have to be careful about existence and uniqueness of solutions of ODEs if we are to use them as the basis for discussing control design.

Example 5.1 (Pendulum) Consider the equations describing the motion of a pendulum with mass m and length l. There are two state variables, the angular position θ and the angular velocity $\dot{\theta}$ (see Figure 5.5). Let us name the state variables $x_1 = \theta, x_2 = \dot{\theta}$; then the

[1] This term should not be confused with autonomous as in autonomous cars. Here autonomous just means that the model has no external inputs.

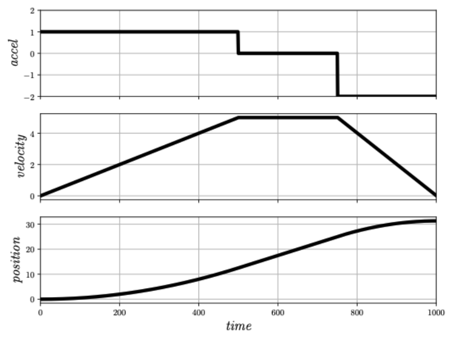


Figure 5.4
The position, velocity, and acceleration of a point moving from point A to B. Since the acceleration is not continuous, the velocity is not differentiable.

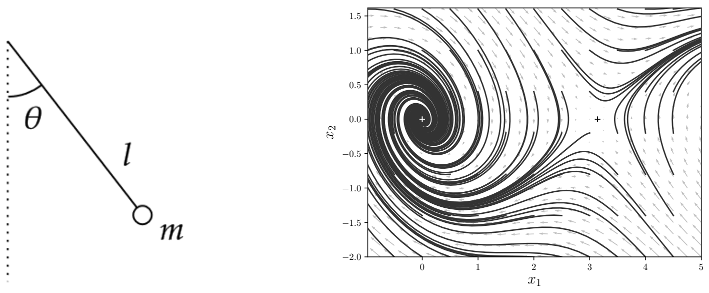


Figure 5.5
A pendulum with angular position θ and length l (*left*). Trajectories of the pendulum and two equilibrium points (*right*)

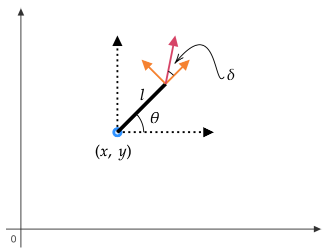
ODE describing the system is the following:

$$\begin{bmatrix} \dot{x}_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ \frac{g}{l}\sin(x_1) - \frac{k}{m}x_2 \end{bmatrix} \tag{5.3}$$

Here g is the acceleration due to gravity, and k is a coefficient of friction and air-resistance. Check that the RHS of the ODE is Lipschitz continuous (Definition 5.1). How do we find out the states where the pendulum does not move? We simply set the RHS to 0, which gives us $x_1 = 0, \pi, 2\pi, \ldots$, and $x_2 = 0$. These are the *equilibria* of the system. □

Example 5.2 (Bicycle model) Consider the kinematic vehicle model of a car with length ℓ. For the state variables, we have the car's position and orientation $\mathbf{p} = (x, y, \theta) \in \mathbb{R}^3$. For simplicity, we consider the longitudinal speed v to be constant. The control input u is the steering angle which is denoted by δ. Then, we can describe the first time derivative for each of the components of $\boldsymbol{p}$ using the inputs, as described below:

$$\boldsymbol{p} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \qquad \begin{aligned} \dot{x} &= v\cos\theta \\ \dot{y} &= v\sin\theta \\ \dot{\theta} &= \frac{v}{l}\tan\delta \end{aligned}$$



You can read more about vehicle models for autonomous driving and their derivations from this excellent survey article Paden et al. (2016). □

Example 5.3 Consider the 1-dimensional ODE:

$$\dot{x} = x^2$$

with initial state $x(0) = 1$. Check by taking derivatives that the following function of time

$$x(t) = \frac{1}{1-t}$$

is a solution. However, notice that as $t \to 1$, $x(t) \to \infty$. Therefore, the solution blows-up and cannot be extended beyond $t = 1$. □

Example 5.4 Consider $\dot{x} = \sqrt{x}$. Again, chec that both

$$x(t) = \frac{t^2}{4} \text{ and } x(t) = 0$$

are solutions of this ODE. That is, the model does not have a unique solution even from the same starting state. □

How can we impose conditions on f such that a unique solution always exists? This is where the notion of *Lipschitz continuity* becomes useful.

Definition 5.1. A function $f : \mathbb{R}^n \to \mathbb{R}^n$ is *Lipschitz continuous* if $\exists L > 0$ such that for any pair $x, x' \in \mathbb{R}^n$,

$$\|f(x) - f(x')\| \leq L\|x - x'\|.$$

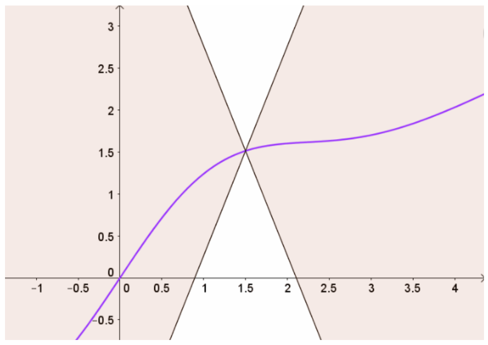


Figure 5.6
The slopes of the black lines marking the red areas are L and $-L$. A function is Lipschitz continuous with L if it lies inside the cones defined by L at the neighborhood of every point.

Exercise 5.1. (a) Show that the functions $6x + 4$; $|x|$ are Lipschitz continuous. (b) Show that all differentiable functions with bounded derivatives are also Lipschitz. Note that a function does not necessarily have to be differentiable to be Lipschitz (as seen in the case of $|x|$). (c) Show that $\sqrt{x}$ and x^2 are not Lipschitz continuous.

Theorem 5.1. *If $f(\cdot)$ is Lipschitz continuous, then $\dot{x} = f(x(t))$ has unique solutions.*

Definition 5.2. Given an ODE $\dot{x} = f(x)$, a state $x^* \in \mathbb{R}^n$ at which $f(x^*) = 0$ is called an *equilibria* of the ODE.

Model-reality gap As in automata models of Chapter 1, the model-reality gap exists for ODEs as well. More detailed models add complexity and could make the model less tractable. Less accurate models will represent reality more coarsely, and therefore, the inferences drawn may lead to false positivies in testing and verification and more conservative design.

In this class we will often focus on linear ODEs of the form:

$$\dot{x}(t) = f(x(t), u(t)) = A(t)x(t) + B(t)u(t). \tag{5.4}$$

Any linear function f can be represented in this form, where $A(t) \in \mathbb{R}^{n\times n}$ and $B(t) \in \mathbb{R}^{m\times n}$ are matrices with entries which can be functions of time t. This is called a Linear Time-Varying system (LTV). If $A(t)$ and $B(t)$ are independent of time, we call that a Linear Time-Invariant system (LTI).

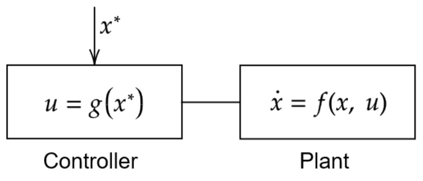


Figure 5.7
The open loop control model.

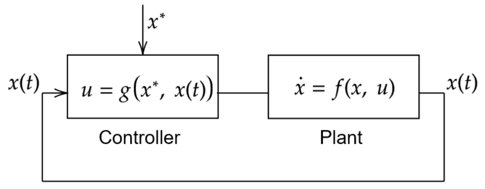


Figure 5.8
The feedback or closed-loop control model.

5.3 Control Design

How would we design a control system? A simple strategy to consider is known as *open loop control*: given an input reference (or target) position x^*, the control input $u(t)$ is calculated simply as a function g of x^* and time t.

This approach does not use the state of the plant $x(t)$ to determine what the controller has to do. So, the controller does not respond to the state or the environment.

Exercise 5.2. (a) Name some examples of household appliances where open loop control is used. What is the state x, the reference x^* and what function do you think is used to compute the control signal? (b) Where would open-loop control work fine and in what situations do you think it would fail?

5.3.1 PID control

A way to incorporate the state of the plant is to include a *feedback* mechanism. *Feedback control*, also known as *closed-loop* control, can be described as follows: Here, we incorporate the difference between the reference x^* and the current output $x(t)$ from the plant through an *error* term, for example $u(t) = g(e(t))$ where the error $e(t) = x^* - x(t)$. More generally, the actual plant state $x(t)$ may not be available, and we have to work with some output $y(t) = s(x(t))$ that is generated by a sensor $s(\cdot)$. More on this and the role of state estimation in Chapter 6.

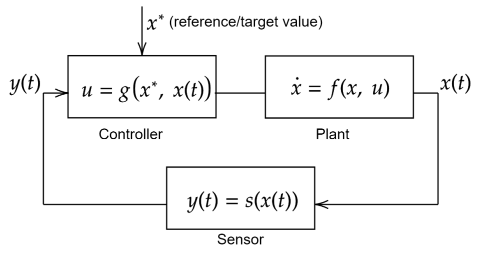


Figure 5.9
A model where a sensor is used for proportion control.

One example is where a car driving down a lane on the road. If the car is very close to the left edge of the lane, we would want a considerable turn in the right direction. If the car is slightly to the left, we would want to move just a little bit to the right. Here the error is the deviation of the car's position $x(t)$ from the center of the lane x^*.

This motivates the incorporation of *proportional* control, where we can include a proportional gain term as so: $u(t) = g(x^*, x(t)) = K_P(x^* - x(t))$. This is also known as *error feedback control*, and to show this in our model, we add a sensor, as seen in Figure 5.9.

There are more ways to fine-tune the function g:

- g not only depends on the error but also the rate of change of error (derivative)
- g also depends on the history of the error (integral)

This gives the general form of the PID controller:

$$\begin{aligned} u(t) &= K_P e(t) + K_I \int_0^t e(\tau) d\tau + K_D \frac{de(t)}{dt} \\ &= K_P[e(t) + \frac{1}{T_I}\int_0^t e(\tau) d\tau + T_D \frac{de(t)}{dt}] \end{aligned} \tag{5.5}$$

Example 5.5 Consider a simple 1-dimensional system in which the state (say, velocity) $y(t)$ is directly controlled by the controller input (say, acceleration) $u(t)$ plus some disturbance $d(t)$. That is,

$$\dot{y}(t) = u(t) + d(t).$$

Suppose the goal is to track a target speed y^*. We can consider the error signal $e(t) = y(t) - y^*$ and use this as negative feedback. That is, we define

$$u(t) = -K_P e(t) = -K_P(y(t) - y^*)$$
$$\dot{y}(t) = -K_P(y(t) - y^*) + d(t),$$

where K_P is the design parameter called *proportional gain.* Now, suppose that in the steady-state the disturbance signal becomes constant, $d(t) = d_{ss}$. What is the corresponding steady state output? We use the usual method of setting the RHS of the ODE to 0 and solving for $y(t)$. We get

$$y_{ss} = y(t) = \frac{d_{ss}}{K_P} + y^*.$$

Note that increasing the proportional gain K_P value makes the error value smaller. Can we make it arbitrarily large? What about the transient behavior of this function? Rewriting the ODE in terms of y_{ss} we get: $\dot{y}(t) = -K_P y(t) + y_{ss}$. Being an ODE of the form $\dot{x} = -ax + b$, we know the form of the solution:

$$y(t) = y(0)e^{-\frac{t}{T}} + y_{ss}(1 - e^{-\frac{t}{T}}),$$

where the time constant $T = 1/K_P$. At $t = 0$, $y(t) = y(0)$ and as $t \to \infty$ $y(t) = y_{ss}$, and the system converges to y_{ss} faster with a smaller K_P. □

We can reduce the steady-state errors (by increasing K_p but this also increases the amplitude of the control signal $u(t)$ and the RHS of $y(y)$. If K_p is very large this may lead to responses that are too quick leading to oscillations and instability. This illustrates one of many trade-off in choosing the PID gains; there are others.

5.3.2 Path following controller

Recall the bicycle model for our vehicle: we represent the direction that the vehicle is moving by using a single wheel at the center of the model. The state of the model can be defined by a vector $p_B(t) = [x_B, y_B, \theta_B, v_b] \in \mathbb{R}^4$. Now, consider a target (reference) position p^* defined by a higher-level planner.

The error function is now a vector: $e(t) = [\delta_s(t), \delta_n(t), \delta_\theta(t), \delta_v(t)]$. The first two terms are known as the *along track* and *cross-track errors*, respectively.

Along track error is the distance ahead or behind the target $p*$ at any instantaneous direction of motion at time t:

$$\delta_s = (x^*(t) - x_B(t)) \cos \theta_B(t) + (y^*(t) - y_B(t)) \sin \theta_B(t)$$

The cross-track error is orthogonal to the intended direction of motion:

$$\delta_n = -(x^*(t) - x_B(t)) \sin \theta_B(t) + (y^*(t) - y_B(t)) \cos \theta_B(t).$$

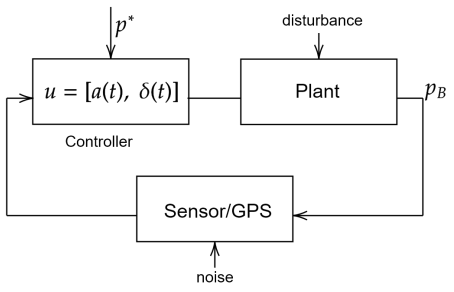


Figure 5.10
A model of our path following system. Note the various sources of error introduced for a more complex model. $a(t)$ is throttle and $\delta(t)$ is steering.

The heading error and velocity error are defined as follows: $\delta_\theta = \theta^*(t) - \theta_B(t)$ $\delta_v = v^*(t) - v_B(t)$. A variant of the path follower controller is the pure pursuit controller, which performs proportion-differential PD-control to correct against the along-track and cross-track error terms:

$$u(t) = K \begin{bmatrix} \delta_s \\ \delta_n \\ \delta_\theta \\ \delta_v \end{bmatrix} \qquad [\text{where } K = \begin{bmatrix} K_s & 0 & 0 & K_v \\ 0 & K_n & K_\theta & 0 \end{bmatrix}]$$

5.4 State-feedback control design

State feedback control is another control design technique. The idea here is to design the control $u(t) = g(x(t))$ signal as a full function of the state. Notice that for a state variable x_1 its higher derivatives $\dot{x}_1, \ddot{x}_1$, etc. can always be included as part of the state vector x for the purpose of control design.

Often the system we study will model the "error" relative to some target, for instance, it could be the difference between the current speed and the target speed in a cruise control system, or the difference between the current position and the target waypoint in a path tracking control system. We would like the error to go to 0 as $t \to \infty$, that is, $\forall x(0) \in \mathbb{R}^n, \lim_{t\to\infty} x(t) = 0$.

5.4.1 Linear state feedback control

Consider a linear time invariant (LTI) system: $\dot{x} = Ax, x \in \mathbb{R}^n, A \in \mathbb{R}^{n\times n}$, and the initial state $x(0) \in \mathbb{R}^n$. The solution of this LTI system is given by:

$$x(t) = x(0)e^{At} \tag{5.6}$$

For discrete time model $x(t+1) = Ax(t)$ the solution is $x(t) = A^t x(0)$.

Review the definition of the matrix exponential e^{At} appearing in Equation (5.6). If this linear system models how the error evolves, and we would like the error to go to 0 as $t \to \infty$.

$$\forall x(0) \in \mathbb{R}^n, \lim_{t\to\infty} x(t) = \lim_{t\to\infty} x(0)e^{At} \overset{?}{=} 0 \tag{5.7}$$

Theorem 5.2. *An LTI system $\dot{x} = Ax$ is asymptotically stable iff all eigenvalues of A have strictly negative real parts. Such matrices are called* Hurwitz.

Exercise 5.3. Visit this this website. Explore different matrices, observe how the stability of the trajectories is related to the eigenvalues.

Exercise 5.4. Consider a two-dimensional system with two states x (national income) and y (rate of consumer spending) and another variable g (rate of government expenditure) such that g is a function of the state x. Let the states x and y and g be defined as follows,

$$\dot{x} = x - \alpha y \tag{5.8}$$
$$\dot{y} = \beta(x - y - g) \tag{5.9}$$
$$g = g_0 + kx \tag{5.10}$$

Applying g into $\dot{y}$ will yield $\dot{y} = \beta[(1-k)x - y - g_0]$. (a) What are equilibrium point of this linear system? (b) Can perform a linear coordinate transform so that in the new coordinate system the equilibrium point is at the origin? (c) Write a program to simulate the system with different values of g_0, k, etc., with different initial conditions.

Vector fields ODEs can be viewed as vector fields. Simply plot $(\dot{x}, \dot{y})$ as a vector at (x, y) and this vector shows how the point is forced to move under the field defined by the ODE (see Figure 5.11 as an example).

For a given point, the trajectory can be drawn as a curve as it travels along the defined vector field. For Figure 5.11, you can see that the curve does not end at a given point. It just loops back onto itself. This means that, for the given point that was picked to start from, the system does not stabilize. The values of x and y oscillate.

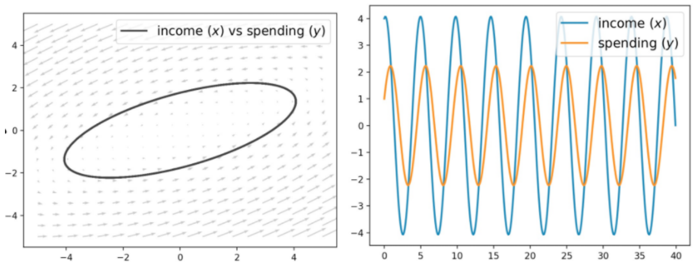


Figure 5.11
Left: Lyapunov stable (but not asymptotically stable) trajectory in phase plane. Right: time plot of the same trajectory.

5.4.2 Eigenvalues and eigenvectors

Recall $x \in \mathbb{R}^n$ and λ are eigenvalues and eigenvectors of A if $Ax = \lambda x$. In other words, $(A - \lambda I)x = 0$. The eigenvalues of the linear system is then determined by solving the characteristic equation $det(\lambda I - A) = 0$. The question then becomes, how can this be used for control design?

Suppose there is a plant that is a linear system and a controller set in a configuration shown in Figure 5.12 that is defined as

$$\dot{x} = Ax + Bu \tag{5.11}$$

$$u = -Kx \tag{5.12}$$

u can be substituted into $\dot{x}$ which would yield

$$\dot{x} = (A - BK)x \tag{5.13}$$

Let's define a matrix $A' = (A - Bk)$. This means that we can choose some matrix for K (gain matrix) to make A' is Hurwitz.

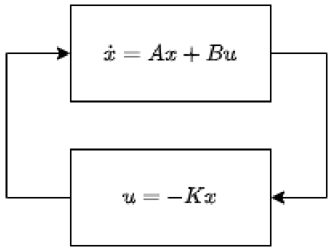


Figure 5.12
Plant and controller for a linear system

Example 5.6 Given the previous model of the path tracking problem, a linearized model of the system is given by

$$\begin{bmatrix} \dot{\delta}_s \\ \dot{\delta}_n \\ \dot{\delta}_\theta \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & v \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \delta_s \\ \delta_n \\ \delta_\theta \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \delta_v \\ \delta_k \end{bmatrix} \tag{5.14}$$

$$\dot{x} = Ax + Bu \tag{5.15}$$

Applying state feedback control will define $u = KBx$ and this results in:

$$\begin{bmatrix} \delta_v \\ \delta_k \end{bmatrix} = \begin{bmatrix} K_s & 0 & 0 \\ 0 & K_n & K_\theta \end{bmatrix} \begin{bmatrix} \delta_s \\ \delta_n \\ \delta_\theta \end{bmatrix} \tag{5.16}$$

Substituted into the right hand side of the above equation yields $\dot{x} = (A - BK)x$, where

$$A - BK = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & v \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} K_s & 0 & 0 \\ 0 & 0 & 0 \\ 0 & K_n & K_\theta \end{bmatrix} = - \begin{bmatrix} K_s & 0 & 0 \\ 0 & 0 & -v \\ 0 & K_n & K_\theta \end{bmatrix} \tag{5.17}$$

K can then be determined such that the matrix $A - BK$ is Hurwitz by solving for K in the characteristic equation $det(A - BK - \lambda I)$. Calculating the determinant of the matrix $A - BK - \lambda I$ results in a polynomial in λ in terms of the values within K (K_θ, K_n, K_w). In order to make the matrix Hurwitz, the real parts of the roots of this polynomial has to be negative, so we choose values for K_θ, K_n, and K_w such that the real parts of the roots are negative. □

Example 5.7 Given some plant that is a linear function $\dot{x} = Ax + Bu$ in which $x \in \mathbb{R}^2$, choose a controller $u = -Kx$ that is some linear function of the state.

The closed loop system is then given to be

$$\dot{x} = Ax - BKx = (A - BK)x \tag{5.18}$$

Let $A' = A - BK$, we want to choose K such that A' is Hurwitz.

Suppose $A = \begin{bmatrix} 0 & v \\ 1 & \frac{1}{2} \end{bmatrix}$, $B = \begin{bmatrix} 0 & 1 \end{bmatrix}$, and $-K = -\begin{bmatrix} K_{11} & 0 \\ 0 & K_{22} \end{bmatrix}$.

$$A - BK = \begin{bmatrix} 0 & v \\ 1 & \frac{1}{2} \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 0 & K_{22} \end{bmatrix} = \begin{bmatrix} 0 & v \\ 1 & \frac{1}{2} - K_{22} \end{bmatrix}$$

$$\lambda I - (A - BK) = \begin{bmatrix} \lambda & -v \\ -1 & \lambda - \frac{1}{2} + K_{22} \end{bmatrix} \tag{5.19}$$

The characteristic equation of the matrix $\lambda I - (A - BK)$ is obtained by setting the determinant to 0.

$$det(\lambda I - (A - BK)) = 0 \tag{5.20}$$

$$\lambda^2 + \lambda(K_{22} - \frac{1}{2}) - v = 0 \tag{5.21}$$

Solving this equation for the roots yield

$$\lambda_1, \lambda_2 = \frac{-(K_{22} - \frac{1}{2}) \pm \sqrt{(K_{22} - \frac{1}{2})^2 + 4v}}{2} \tag{5.22}$$

In order to make the matrix A' Hurwitz, the real components of roots of the characteristic equation have to be negative. This means that $Re(\lambda_1) < 0$ and $Re(\lambda_2) < 0$.

This creates two cases in to make A' be Hurwitz.

1. If $(K_{22} - \frac{1}{2})^2 + 4v < 0$, then the real component of the root is defined by $\frac{-(K_{22} - \frac{1}{2})}{2}$. This means that K_{22} must be greater than $\frac{1}{2}$ in order for the real component to be negative.
2. Otherwise $(K_{22} - \frac{1}{2})^2 + 4v \geq 0$. The real component of the root is defined by the entire root. This means that $(K_{22} - \frac{1}{2})^2 + 4v < (K_{22} - \frac{1}{2})^2$ or $v < 0$ in order for the entire root to be negative.

□

Summary so far

- PID control
  - Detailed models are not necessary
  - Heuristics are used for tuning parameters
- State-feedback control
  - Uses a linearized model
  - Algorithm for finding parameters (gain matrix) using the Hurwitz condition

5.5 Requirements for control systems: Stability

What are the typical performance requirements of a control system?

- Convergence to an equilibrium. The system under study often models the error (see previous section), relative to some reference point or trajectory. The convergence to equilibrium then corresponds to tracking the reference point.

- State variables stay bounded; this is related to invariance (how?)
- Bounded-input, bounded-output (BIBO) stability (not covered)

Convergence to an equilibrium set point is the main objective of any control system and can be implemented as driving the system to zero as mentioned above. In a continuous state space, we would like our system to exhibit some type of bounded behavior. This is important to be able to guarantee anything about the system, in regards to both safety and general proper functionality of the system. In previous chapters, we describe invariance and defined an invariant set I to be any set of states that contains all the reachable states. More formally, for any possible choice of the initial state of the system, and for all time, the state of the system stays within some nice set I:

$$\forall\ x(0) \in \theta, \forall t, x(t) \in I_\theta$$

By proving some ground truths about the behavior of our system, we can make the assumptions that are necessary to bridge the gap between model and reality. BIBO stability, in short, is the property of a system where the output of the system is proportional to its inputs. In other words, a small disturbance to the system will not result in unbounded behavior.

For the purpose of discussion and in accordance with some of the plots to come, let's consider the origin $\vec{0} \in \mathbb{R}^2$ as an equilibrium (i.e. $f(\vec{0}) = 0$). Previous sections describe the conditions that must hold for asymptotically stability (to be defined). Let's consider the non-linear system where

$$\dot{x} = f(x)$$

5.5.1 Lyapunov stability

Russian mathematician and physicist Aleksandr M. Lyapunov defines stability of ordinary differential equations (ODEs). In the theory of probability, he generalized the works of Chebyshev and Markov, and proved the Central Limit Theorem under more general conditions that his predecessors.

For the next definition we use the following notation. A *ball* of radius $r > 0$ centered at $x \in \mathbb{R}^n$, is defined as the set $B_{r,x} := \{x' \in \mathbb{R}^n \mid |x - x'| \leq r\}$. For balls centered around the origin $x = 0$, we write in brief B_r instead of $B_{r,0}$.

Definition 5.3. (Lyapunov stability) The system $\dot{x} = f(x)$ is *stable (Lyapunov stable)* if

$$\forall \epsilon > 0, \exists \delta_\epsilon > 0 \text{ such that if} x(0) \in B_\delta \text{ then } \forall t \geq 0, x(t) \in B_\epsilon.$$

Otherwise the system is *unstable*.

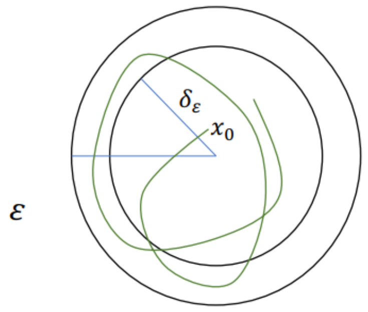


Figure 5.13
Shown are a pair of δ_ε and ε balls in definition of Lyapunov stability. The orange trajectory starts within the δ_ε-ball and never leaves the ε-ball.

Notice that each δ_ϵ may depend on the corresponding ϵ.

How is this notion of stability related to invariants and reachable states?

According to this definition of stability, for any $\epsilon > 0$, if the system starts anywhere inside the $x(0) \in B_{\delta_\epsilon}$, $x(t)$ may evolve outside of the B_{δ_ϵ}, but $x(t)$ will always remain B_ϵ. This captures the idea that the states in a stable system do not “blow up”. We can think of the B_ϵ as an invariant set, provided that the initial state $x(0) \in B_{\delta_\epsilon}$. Thus, Lyapunov stability is strong property; it gives a whole family of B_ϵ-balls as invariants (provided the system starts from a small enough δ-ball).

5.5.2 Asymptotic stability

Definition 5.4. (Asymptotic Stability) The system is (globally) asymptotically stable if it is Lyapunov stable and $\lim_{t\to\infty} x(t) = 0$.

Asymptotic stability is by definition a stronger notion that Lyapunov stability. See Figure **??**

Exercise 5.5. What can we say about the stability/asymptotic stability of the following systems?

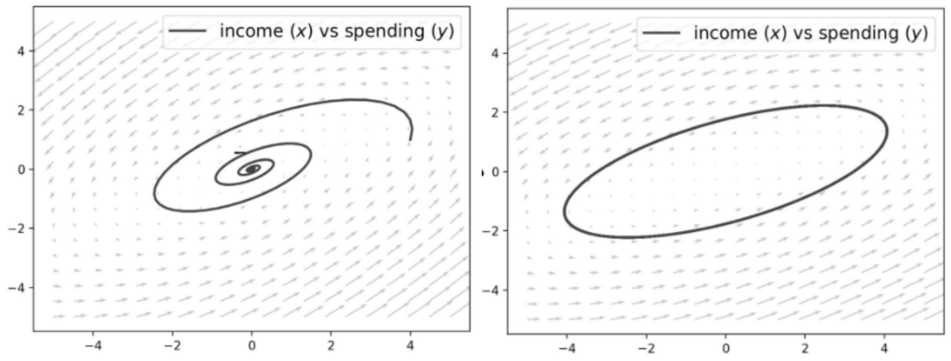
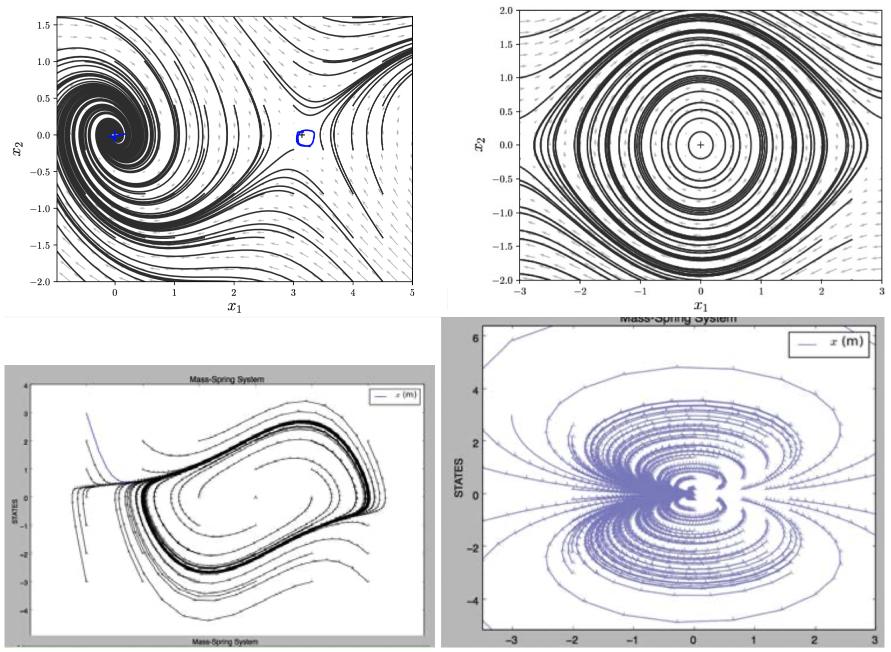


Figure 5.14
Left: Trajectory eventually converges to the origin. Right: Trajectory does not converge to zero, but also does not blow up.



5.5.3 Invariance & stability verification

In Chapter 1, we used the following method to prove invariants of a system. This is a restatement of Proposition 1.1: $I \subset \mathbb{R}^n$ is an invariant if:

$$x(0) \in I \text{ and } f(I) \subseteq I.$$

That is, a set I is an invariant set if (i) any initial state is in I, and (ii) for any state in I, the function f (that models our system) applied to that state results in a state that is also in I. We will see a similar one-step method for proving stability and asymptotic stability of nonlinear models $\dot{x} = f(x)$.

First let's build some intuition by asking a slightly different question. Given any initial state of the system, can we determine whether the system will reach the equilibrium or not? How do we prove that a program or an automaton terminates?

Intuition: Suppose there something about the system, let's call it "energy", that has some sort of a lower bound that cannot drop below some floor, and in every step of the program, this "energy" decreases. If we can come up with such an energy function, the can show that the system always goes down to the zero-energy states.

To make that mathematically precise, we define the ranking function.

Ranking function: $R : X \to \mathbb{N}$

$$\forall x, R(f(x)) < R(x)$$

For any state, the rank of a function f applied to that state is strictly less than the rank of the state. Note that the range of the function being over all natural numbers $\forall x, R(x) \geq 0$.

Coming up with the ranking function is not straightforward but if we were able to come up with this ranking function, we could show that program terminates.

More generally, we ask if we could allow for a more general ranking $R : X \to \mathbb{R}$? The answer is yes, and the solution lies among differential equations and their solutions given by the *Lyapunov function*.

Theorem 5.3. *Suppose their exists a positive definite, radially unbounded continuous function $V : \mathbb{R}^n \to \mathbb{R}_{\geq 0}$ such that*

1. *if $\dot{V} \leq 0$, then the system is Lyapunov stable.*
2. *if $\forall x \neq 0$ and $\dot{V} < 0$, then the system is asymptotically stable.*

What is $\dot{V}$? It might seem odd that V is a function of state and yet, we are taking derivative of V w.r.t. time. What is going on here? $\dot{V} = \frac{\partial V(x(t))}{\partial t}$. Using the chain rule of differentiation, we can write $\dot{V}$ as $\frac{\partial v}{\partial x} \cdot \frac{\partial x(t)}{\partial t} = \frac{\partial v}{\partial x} f(x)$. The last step is using the fact that $x(t)$ is a solution of the ODE $\dot{x} = f(x(t))$ and replcing $\frac{\partial x(t)}{\partial t}$ with $f(x)$.

Notice that Theorem 5.3 does not require us to solve the differential equations and yet we get to prove the strong property of (asymptotic) stability.

Example 5.8 Consider the nonlinear system described by

$$\dot{x} = -x + y^2$$
$$\dot{y} = -2y + 3x^2$$

and the corresponding Lyapunov function

$$V(x, y) = \frac{x^2}{2} + \frac{y^2}{4}. \tag{5.23}$$

Clearly V is positive definite and radially unbounded. Now calculate

$$\begin{aligned}\dot{V}(x, y) &= \frac{\partial V}{\partial x} f_x(x, y) + \frac{\partial V}{\partial y} f_y(x, y)\\ &= x(-x + y^2) + y/2(-2y + 3x^2)\\ &= -x^2 + xy^2 - y^2 + \frac{3}{2}yx^2\\ &= -x^2(1 - \frac{3}{2}y) - y^2(1 - x),\end{aligned}$$

which is negative for all x, y satisfying $x < 1$ and $y < 2/3$. This indicates that the origin is (locally) asymptotically stable. □

We relate the idea of idea of a trajectory being contained within a certain ball to the idea of a Lyapunov function via level sets.

Definition 5.5. For any function $f : \mathbb{R}^n \to \mathbb{R}$ and an $a \in \mathbb{R}$, the *a-level set* of f is the set

$$L_a = \{x \in \mathbb{R}^n \mid f(x) = a\}.$$

The *a-sublevel set* of f is the set

$$S_a = \{x \in \mathbb{R}^n \mid f(x) \leq a\}.$$

Corollary 5.4. *Consider a nonlinear system $\dot{x} = f(x)$ with an initial set of states $\Theta \subseteq \mathbb{R}^n$. Suppose $V : \mathbb{R}^n \to \mathbb{R}_{\geq 0}$ is a Lyapunov function. Then for any $a \geq 0$, with the sub-level set S_a containing Θ, S_a is an invariant of the system.*

Exercise 5.6. Prove Corollary 5.4.

Finding a Lyapunov function may be challenging in general. For stable linear systems, there is always a quadratic Lyapunov function. Moreover, such a quadratic Lyapunov function can always be computed by solving what is called Lyapunov equations, which is a linear matrix inequality. For nonlinear systems, one typically guesses the form of the Lyapunov function then solves an optimization problem to make $\dot{V}$ negative. Once we have found a Lyapunov function, it give us (asymptotic) stability and a whole family of sublevel sets which are candidate invariants.

Summary

- Stability Requirements
- Lyapunov stability and asymptotic stability
- Lyapunov method for proving stability

6 State estimation and localization

The notes in this chapter heavily borrow from Thrun et al. (2005).

6.1 The state estimation problem (informal)

Recall from Chapter 5 that a closed-loop control system can be represented as:

$$x(t+1) = f(x(t), u(t))$$

$$u(t) = g(x(t))$$

In that chapter, we assumed that the complete and accurate state $x(t)$ is available to the controller g at all times. In real autonomous system, the complete state information may not be available. For example, the speed of the car may be available from a vehicle wheel speed sensor, but the absolute position and heading angles may not be available. This leads to the *state estimation problem:* How do we find $x(t)$? Usually we have observations or measurements of some quantities related to $x(t)$, say, $z(t) = h(x(t))$, from which we would like to infer the actual state $x(t)$ or the relevant parts of state. The *estimated state* is usually denoted by $\hat{x}(t)$. *Localization* is a special case of estimation where we determine the pose of a robot relative to the given map of the environment.

6.1.1 Different types of localization

There are many different approaches we can take to the localization problem:

Problem approaches:	**Global Localization Problem** The initial position is unknown	**Local Localization Problem** The initial position is known; the uncertainty in this problem comes from motion
Type of environment:	**Static Environment** All obstacles in the environment are stationary	**Dynamic Environment** obstacles in the environment can move
Number of robots:	**Single robot**	**Multiple robots**
Control Approach:	**Passive Approach** The localization module does not influence control input u_t	**Active Approach** The localization module can influence u_t to gain more information about the environment

6.1.2 Setup

The discrete time model we will use is as follows:

$$\begin{aligned} x_t &= f(x_{t-1}, u_t) + w_t \\ z_t &= g(x_t, m) \end{aligned} \tag{6.1}$$

where x_t is the state at time t (which may not be fully observable), u_t is the control input for time t, w_t represents noise, and z_t is the observed measurement of the state x_t at time t.

In this chapter, we will work with sequences of states, inputs, and observations, and for that we introduce the following notations. For any two time points $t_2 > t_1$:

- $x_{t_1:t_2}$ is the sequence of states $x_{t_1}, x_{t_1+1}, \ldots, x_{t_2-1}, x_{t_2}$.
- $u_{t_1:t_2} = u_{t_1}, \ldots, u_{t_2}$
- $z_{t_1:t_2}$ is defined similarly.

We will review basic concepts in probability before stating the estimation problem precisely.

6.1.3 Review of probability

Random variables are written using capital letters X, Y, Z, etc. The values that a random variable X can take are written as x_1, x_2, etc. $P(X = x_i)$ is the probability that X takes the value x_i. For example, if X is the integer-valued random variable modeling the result of pair of dice rolls, then we would write $P(X = 12) = \frac{1}{36}$. The notation is extended to multiple random variables: $P(X = x_i, Y = y_j)$ is the probability that $X = x_i$ *and* $Y = y_i$. We will write $P(x_i, y_j)$ as an abbreviation for $P(X = x_i, Y = y_j)$, when the random variables X, Y are clear from context.

The probability that $X = x_i$ *given* that $Y = y_j$ is called *conditional probability,* and is written as $P(X = x_i|Y = y_j)$ or as $P(x_i|y_j)$ in brief. Conditional probability is defined as

$$P(x_i|y_j) = \frac{P(x_i, y_j)}{P(y_j)},$$

provided where $P(y_j) > 0$. Therefore, $P(x_i, y_j) = P(x_i|y_j)P(y_j) = P(y_j|x_i)P(x_i)$. After substitution (and dropping the indices), we get:

Bayes Rule.

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)}$$

Here $P(x)$ is called the prior distribution over x, the conditional probability $P(x|y)$ is called *posterior* distribution, and $P(y|x)$ is called the *inverse conditional distribution.* Often the denominator $P(y)$, which is independent of x, is not that important. This is because, we'd want $P(x|y)$ to be a probability distribution over x and therefore the quantity $P(y|x)P(x)$ is normalized. In terms of the state estimation problem, we are interested in the posterior distribution of x given the measurement z, that is $P(x \mid z)$, starting from the prior knowledge $P(x)$.

Exercise 6.1. Show the following

$$P(x \mid y, z) = \frac{P(y \mid x, z)P(x \mid z)}{P(y \mid z)}, \text{where } P(z) > 0 \tag{6.2}$$

Exercise 6.2. Show the following fact which is called the *law of total probability.*

$$P(x) = \Sigma_y P(x \mid y)P(y). \tag{6.3}$$

6.2 Estimation problem (formal)

Let $X_0, X_1, X_2, \ldots$ be a sequence of random variables representing the states of the system at $t = 0, 1, 2, \ldots$. That is, $X_0 = x_0, X_1 = x_1, X_2 = x_2, \ldots$ represents a particular trajectory of the discrete time system of Equation (6.1). Similarly, let $U_1, U_2, \ldots$ be the random variables representing the control inputs, and let $Z_1, Z_2, \ldots$ be the random variables for the measurements. The state estimation problem is to estimate the state X_t at time t, given the history of the previous states, inputs, and measurements. Thus, the problem is to compute or estimate the following probability distribution:

$$\begin{aligned} P(X_t = x_t \mid \quad & X_0 = x_0, \ldots, X_{t-1} = x_{t-1}, U_1 = u_1, \ldots, U_t = u_t \\ & Z_1 = z_1, \ldots, Z_{t-1} = z_{t-1}) \end{aligned} \tag{6.4}$$

Which we write in brief as:

$$P(x_t \mid x_{0:t-1}, u_{1:t}, z_{1:t-1}). \tag{6.5}$$

To solve this problem, we will use probabilistic versions of the system model described in Equation 6.1.

We assume that the state is *complete*, that is, it is a sufficient summary of all that happened in previous time steps. That is, x_{t-1} has all the relevant information from the previous states, controls, and measurements to define the next state with the given input u. This can be written as

$$P(x_t \mid x_{0:t-1}, u_{1:t}, z_{1:t-1}) = P(x_t \mid x_{t-1}, u_t).$$

This is called the *Markovian property* and this distribution P is called the generative model of state evolution. In the extreme case of a purely deterministic system, $P(x_t = f(x_{t-1}, u_t) \mid x_t, u_t) = 1$.

The second model we need is related to the measurement function g. We assume that the measurement model is *complete* with respect to the state. That is, $P(z_t \mid x_{0:t}, u_{1:t}, z_{0:t-1}) = P(z_t \mid x_t)$.

The final notion we need to introduce is the notion of *belief.*

Definition 6.1. The *belief* at a given time t is the posterior distribution over state, given all the past measurements and control inputs. Belief at time t is denoted by

$$bel(x_t) = P(x_t \mid z_{1:t}, u_{1:t}). \tag{6.6}$$

6.3 Bayes filter

Applying Equation (6.2) to the definition of $bel(x_t)$ (with $x = x_t$, $y = z_t$, $z = z_{1:t-1}; u_{1:t}$), we get:

$$\begin{aligned} bel(x_t) &= \frac{P(z_t \mid x_t, z_{1:t-1}, u_{1:t})P(x_t \mid z_{1:t-1}, u_{1:t})}{P(z_t \mid z_{1:t-1}, u_{1:t})} \\ &= \eta P(z_t \mid x_t, z_{1:t-1}, u_{1:t})P(x_t \mid z_{1:t-1}, u_{1:t}) \\ &= \eta P(z_t \mid x_t)P(x_t \mid z_{1:t-1}, u_{1:t}) \quad \text{[Using completeness of measurement]} \\ bel(x_t) &= P(z_t \mid x_t)\overline{bel}(x_t) \end{aligned} \tag{6.7}$$

The distribution over the state $\overline{bel}(x_t)$ is very similar to $bel(x_t)$ except that it does not use the most recent measurement z_t. Now we expand the definition of $\overline{bel}(x_t)$ using the law of totla probability:

$$\begin{aligned}
\overline{bel}(x_t) &= P(x_t \mid z_{1:t-1}, u_{1:t}) \\
&= \sum_{x_{t-1}} P(x_t \mid x_{t-1}, z_{1:t-1}, u_{1:t}) P(x_{t-1} \mid z_{1:t-1}, u_{1:t}) \\
&= \sum_{x_{t-1}} P(x_t \mid x_{t-1}, u_t) P(x_{t-1} \mid z_{1:t-1}, u_{1:t}) \quad \text{[Using Markov state transitions]} \\
\overline{bel}(x_t) &= \sum_{x_{t-1}} P(x_t \mid x_{t-1}, u_t) P(x_{t-1} \mid z_{1:t-1}, u_{1:t-1}) \quad \text{[Assuming } u_t \text{ randomly]} \\
\overline{bel}(x_t) &= \sum_{x_{t-1}} P(x_t \mid x_{t-1}, u_t) bel(x_{t-1}) \qquad (6.8)
\end{aligned}$$

This derivation is the basis for the Bayes filtering algorithm which is presented below:

```
1: Bayes_filter($bel(x_{t-1}), u_t, z_t$):
2:
3: for $x_t$ do
4:     $\overline{bel}(x_t) = \int P(x_t \mid x_{t-1}, u_t)\, bel(x_{t-1}) dx_{t-1}$
5:     $bel(x_t) = \eta P(z_t \mid x_t)\, \overline{bel}(x_t)$
6: end for
7: return $bel(x_t)$
```

The `Bayes_filter` algorithm takes as input the belief at time $t-1$, the control input u_t at time t, and the observed state z_t at time t, and returns the belief at time t. Since $bel(x_t)$ is a distribution over x_t, for a discrete state system, the algorithm computes the probability $bel(x_t)$ for each state. Line 4 implements a continuous version of Equation (6.8), which updates the belief based on the control input u_t. This is called the *prediction step,* because it uses the motion model $P(x_t \mid x_{t-1}, u_t)$. Line 5 implements Equation (6.7), which updates the belief based on the measurement z_t. This is called the *correction step,* and it uses the measurement model $P(z_t \mid x_t)$.

Exercise 6.3 (From Thurn)**.** A robot uses a range sensor that can measure ranges from 0m to 3m. For simplicity, assume that actual ranges are distributed uniformly in this interval. Unfortunately, the sensor may be faulty When the sensor is faulty, it consistently outputs a range below 1m, regardless of the actual output range in the sensor's measurement cone. We know that the prior probability for a sensor to be faulty is $p = 0.01$.

Suppose the robot queries its sensor N times and every time the measurement value is below 1m. What is the posterior probability of a sensor fault for $N = 1, 2, ..., 10$. Formulate the corresponding probabilistic model.

6.4 Histogram filter

The histogram filter is a finite state version of the Bayes filter where the random variable X_t can take finitely many values $x_1, \ldots, x_k$. Examples include occupancy grids.

This algorithm essentially represents the beliefs by histograms $\{p_{k,t}\}$, and replaces the integral with a sum.

```
1: Histogram_filter($\{p_{k,t-1}\}, u_t, z_t$):
2:
3: for $k$ do
4:     $\bar{p}_{k,t} = \sum_i P(X_t = x_k \mid X_{t-1} = x_i, u_t) p_{i,t-1}$
5:     $p_{k,t} = \eta P(z_t \mid X_t = x_k)\, \overline{p}_{k,t}$
6: end for
7: return $\{p_{k,t}\}$
```

6.5 Particle filter

The particle filter is a powerful filtering algorithm which uses the same underlying principles of the Bayes filter, but represents the beliefs in a completely different way. In a particle filter, the beliefs $bel(x_t)$ are represented by a finite number of *particles* or samples of the state. This representation brings a few advantages:

- The representation is *non-parametric*, and therefore, can represent a broader set of distributions. The Kalman filter, in contrast, relies heavily on Gaussian distributions represented by their means and variances.
- Particle filters can handle nonlinear transformations of the distributions. For example, the operations needed for the prediction and correction stages of the filter.
- Particle filters scale to higher-dimensional models than grid/histogram filter because the entire state space does not have to be discretized.

Before giving the concrete algorithm, let's first define the key notion of particles more precisely.

Definition 6.2. Samples of $bel(x_t)$ distribution are called *particles*. We denote M particles by the set $\mathcal{X}_t = \{x_t^{[1]}, x_t^{[2]}, \ldots, x_t^{[M]}\}$, where each $x_t^{[m]} \in \mathbb{R}^n$ for an n-dimensional real-valued state space.

Example 6.1 For the rear wheel vehicle model, $x_t^{[m]} = \langle \text{pos}_x, \text{pos}_y, \theta \rangle$ and typically using $M \approx 100$ particles will be adequate. □

In representing $bel(x_t)$ with $\mathcal{X}_t$, ideally, the $x_t^{[m]}$ should be included in $\mathcal{X}_t$ with probability proportional to $bel(x_t) = P(x_z|\cdots)$. This will hold asymptotically as $M \to \infty$. The particle filtering algorithm is shown below.

```
**Require:** $X_{t-1}, u_t, z_t$
  $X_t = \{x_t^{[1]}, \ldots, x_t^{[M]}\}$ particles
  $\bar{X}_{t-1} = X_t = \emptyset$
  **for** all $m$ in $[M]$ **do**
    sample $x_t^{[m]} \sim p(x_t|u_t, x_{t-1}^{[m]})$ // sampling from motion model
    $w_t^{[m]} = p(z_t|x_t^{[m]})$ // calculates importance factor $w_t$ or weight
    $\hat{X}_t = \hat{X}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$ // intermediate weighted particles.
  **end for**
  **for** all $m$ in $[M]$ **do**
    draw $m$ with probability $\propto w_t^{[m]}$ // importance sampling based on weights
    add $x_t^{[m]}$ to $X_t$
  **end for**
  Return $X_t$
```

Importance sampling The key new idea in the particle filter is *importance sampling* which is implemented in the second for loop. Suppose we want to compute $\mathbb{E}_f[I(x \in A)]$ but we can only sample from density $g(\cdot)$. For reasons that will become clear, we will have to assume that for any x, $f(x) > 0 \implies g(x) > 0$. We can write $\mathbb{E}_f[I(x \in A)]$ as follows:

$$\begin{aligned}\mathbb{E}_f[I(x \in A)] &= \int f(x) I(x \in A)\, dx \\ &= \int \frac{f(x)}{g(x)} g(x) I(x \in A)\, dx \\ &= \int w(x) g(x) I(x \in A)\, dx \\ &= \mathbb{E}_g[w(x) I(x \in A)]\end{aligned}$$

The multiplication and division in step 2 is allowed only if the above assumption holds. The fraction $\frac{f(x)}{g(x)}$ is precisely the weight $w(x)$ associated with the value (particle) x which accounts for the mismatch between f and g. Importance sampling utilizes this transformation.

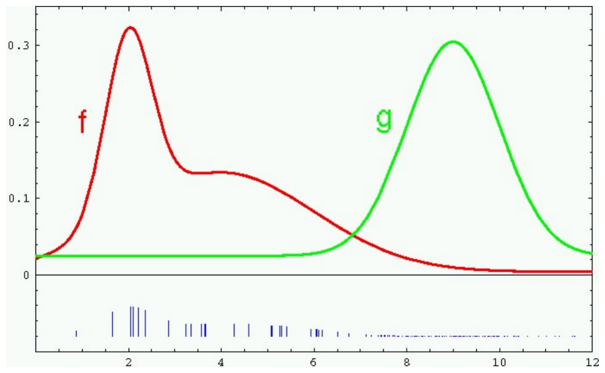


Figure 6.1
Importance Sampling Figure from Thrun et al. (2005).

The resampling step in the particle filtering algorithm, is a special case of importance sampling: we want to sample from $bel(x_t)$, but we can only sample from $\overline{bel}(x_t)$, and therefore, the weighting trick is used.

6.5.1 Monte-Carlo Localization

The *Monte Carlo Localization* (MCL) algorithm is a direct applicaiton of particle filtering to the robot localization problem. Given a map of the environment, this algorithm estimates the location and orientation by plugging in motion and measurement models in the particle filter.

```
Require: $X_{t-1}, u_t, z_t, m$
  $X_t = \{x_t^{[1]}, \dots, x_t^{[M]}\}$ particles
  $\bar{X}_{t-1} = X_t = \emptyset$
  for all $m$ in $[M]$ do
    sample $x_t^{[m]}$ = sample_motion_model($u_t, x_{t-1}^{[m]}$)
    $w_t^{[m]}$ = sample_measurement_model($z_t, x_t^{[m]}$)
    $\hat{X}_t = \hat{X}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
  end for
  for all $m$ in $[M]$ do
    draw $i$ with probability $\propto w_t^{[t]}$
    add $x_t^{[i]}$ to $X_t$
  end for
  Return $X_t$
```

Although the pipeline we described so far is able to track the pose of a mobile robot and to globally localize the robot, mitigating localization errors/failures (like the kidnapped robot problem) remains a problem, which is particularly serious when the number of particles is small. To further address this problem, we can randomly insert samples according to localization performance. We can monitor the probability of sensor measurements $p(z_t|z_{1:t-1}, u_{1:t}, m)$. Concretely, for particle filters, such probability is approximately $\frac{1}{M}\sum_{i=1}^{M} w_t$. Through inserting random samples proportional to the average likelihood of the particles, the robot will be teleported with higher probability when the likelihood of its observations drops.

Summary

- Particle filters are an implementation of recursive Bayesian filtering
- They represent the posterior by a set of weighted samples.
- In the context of localization, the particles are propagated according to the motion model.
- They are then weighted according to the likelihood of the observations.
- In a re-sampling step, new particles are drawn with a probability proportional to the likelihood of the observation.

6.6 SLAM Overview

6.6.1 Problem statement

Rest of this Chapter is not yet reviewed. In the localization problems we have dealt with so far, we are always assuming that the map is known. On the other hand, Simultaneous Localization and Mapping (SLAM) is task of building a map while estimating the pose of the robot relative to this map. SLAM is particular hard because it is a chicken-egg problem: a map is needed to localize the robot but a pose estimate is needed to build a map. More concretely, imagine a robot moving though an unknown, static environment, given knowledge about the robot's controls and its observations of nearby features, we need a map of features and a path of the robot.

SLAM problem also have different formulations depending on the available state/history and continuous/discrete correspondence variables.

- State / history
  - Online SLAM: Estimates most recent pose and map $p(x_t, m|z_{1:t}, u_{1:t})$

    Given control inputs (u), measurements(z), and white nodes to be determined (x, m), we want to calculate $p(x_t, m|z_{1:t}, u_{1:t})$.
  - Full SLAM: Estimates entire path and map $p(x_{1:t}, m|z_{1:t}, u_{1:t})$

Given control inputs (u), measurements(z), and white nodes to be determined (x, m), we want to calculate $p(x_{1:t}, m|z_{1:t}, u_{1:t})$.

- Continuous or discrete correspondence variables: $p(x_t, m_t, c_t|z_{1:t}, u_{1:t})$ x_t and m are continuous while the relationship of detected objects to new objects are discrete.

In reality, SLAM is a hard problem for the following reasons.

- In the real world, the mapping between observations and landmarks is unknown.
- Picking wrong data associations can have catastrophic consequences.
- Pose error correlates data associations.

6.6.2 Data Association Problem

Definition 6.3. A data association is an assignment of observations to landmarks.

In general, there are more than $\binom{n}{m}$ possible associations where n is the number of observations and m is the number of landmarks. Although particle filter could technically be used to solve SLAM, the number of particles needed to represent a posterior grows exponentially with the dimension of the state space! Therefore, a naïve implementation of particle filters to SLAM will be crushed by the curse of dimensionality.

However, we use the dependency between the dimensions of the state space to solve the problem more efficiently. Concretely, in the context of SLAM, the map depends on the poses of the robot while know how to build a map given the position of the sensor is known. To further illustrate this point, we need to recall some tools from statistics.

Definition 6.4. Random variables A and B are conditionally independent given C if

$$P(A \cap B|C) = P(A|C) \cdot P(B|C).$$

For example, height and vocabulary are not independent, but they are conditionally independent given age.

By utilizing conditional independence, we can factor the SLAM posterior into a robot path posterior and a conditionally independent landmark position posterior: the knowledge of the robot's true path renders landmark positions conditionally independent.

$$p(x_{1:t}, l_{1:t}|z_{1:t}, u_{0:t-1}) = p(x_{1:t}|z_{1:t}, u_{0:t-1}) \cdot p(l_{1:t}|z_{1:t}, x_{1:t}) = p(x_{1:t}|z_{1:t}, u_{0:t-1}) \cdot \prod_{i=1}^{M} p(l_i|z_{1:t}, x_{1:t}).$$

This factorization is also called Rao-Blackwellization. Given that the second term can be computed efficiently, particle filtering becomes possible!

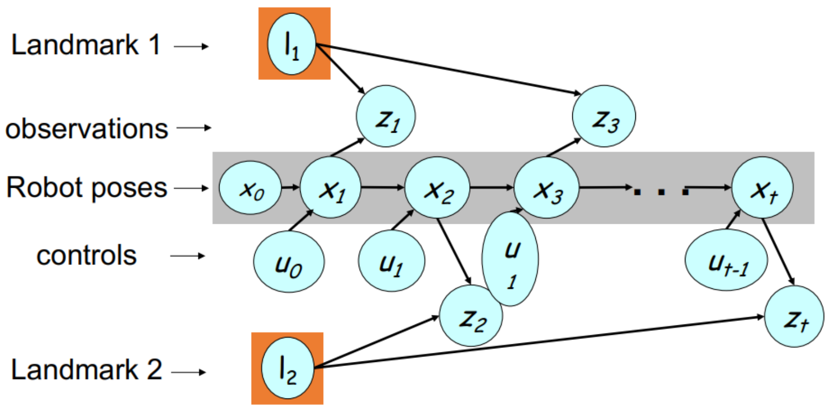


Figure 6.2
Mapping using Landmarks

Enabled by Rao-Blackwellization, **FastSLAM** algorithm was developed, where each landmark is represented by a 2x2 Extended Kalman Filter (EKF) and each particle therefore has to maintain M EKFs. As a result, FastSLAM achieved $O(N \cdot \log(M))$ overall complexity where N is the number of particles and M is the number of map features.

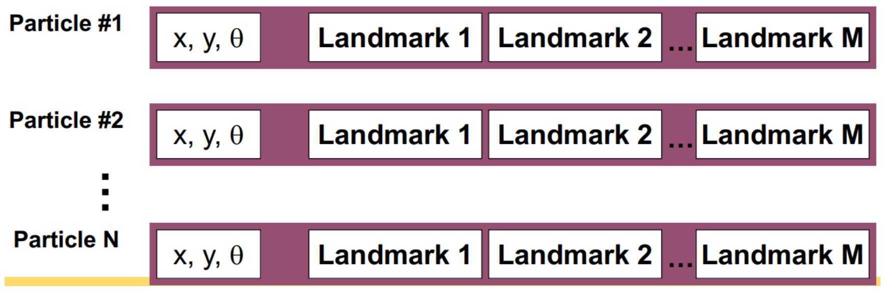


Figure 6.3
FastSLAM

Now, let's come back to the data association problem we defined previously with our efficient SLAM particle filter in mind. A robust SLAM must consider possible data associations while potential data associations depend also on the pose of the robot. With efficient particle filters, we can tackle the data association problem on a per-particle basis where robot pose error is factored out of data association decisions. Concretely, there are two options for per-particle data association: we can pick the most probable match or we can pick an random association weighted by the observation likelihoods. If the probability is too low, we will generate a new landmark.

FastSLAM

- Maintain set of particles
  - Each particle contains s sampled robot path and a map
  - Each feature in the map represented by local Gaussian
  - Result linear is size of map and number of particles
- Trick is to represent map as a set of separate Gaussians instead of a giant joint distribution, which is possible because of conditional independence given a path
- In the context of localization, the particles are propagated according to the motion model.
- Update rule similar to conventional particle filter
- Each particle can be based on a different data association

7 Searching and planning

7.1 The planning problem

This chapter will cover the concepts of search and planning. There are many different motivations and levels of navigation that search and planning are used at, from global path planners to compute routes (i.e. Dijkstra's), to figuring out how to navigate areas such as parking lots (variation of A*). We can begin to address the planning problem through using an e.g. four dimensional (using x position, y position, heading θ , direction dir) discrete uniform grid, but this path may not actually be realistic for vehicle dynamics to perform. We can smooth this discrete path to make it more feasible to traverse, which we will cover later in this chapter.

Roadmap

- Performance Metrics
- Deterministic Search and planning
  - Uniform-Cost Search
  - Greedy (Best-First) Search
  - A search
  - A* Search
  - Hybrid A* Search
- Sampling-based planning
  - Probabilistic Roadmaps (PRMs)
  - Rapidly-expanding Random Trees (RRT)
  - Rapidly-expanding Random Graphs (RRG)

Before any specifics, consider the shortest path problem. It takes in an input of $\langle V, E, w, start, goal\rangle$, and outputs a path (sequence of vertices $v \in V$) $\langle P\rangle$. V is a finite set of vertices, $E \subseteq V \times V$ is a finite set of the edges, w is a function that gives a positive weight to each edge e, and both of $\{start, goal\} \in V$. The objective is for $w(P)$ to be minimal.

Exercise 7.1: What is the minimum path from s to g in figure 7.1?

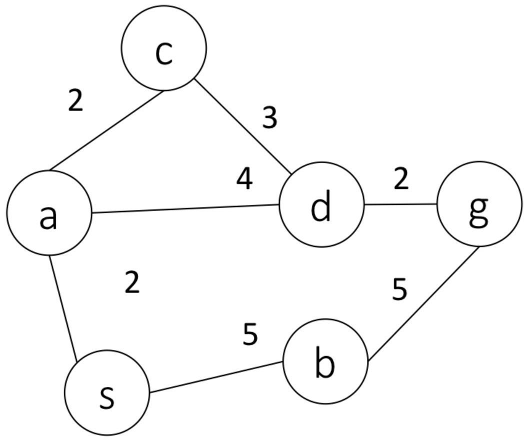


Figure 7.1
Small sample graph

7.2 Performance Metrics

When evaluating a search and planning algorithm, there are a few criterion that we can use to describe its behavior and performance.

- Soundness
- Completeness
- Optimality
- Space complexity
- Time complexity

Soundness asks the question ”Is the solution that is returned guaranteed to be correct?”. *Completeness* means that if there is a solution to be found, the search algorithm will find that solution. *Optimality* revolves around whether the solution that is found is the best solution to the problem. Space and time complexity discusses the amount of memory needed and how fast does the algorithm run respectively.

7.3 Deterministic Search

A deterministic search algorithm is simply an algorithm that if you run it multiple times on the same data set, each time the same solution will be returned.

7.3.1 Uniform Cost Search

Properties

- Complete
- Optimal
- Time complexity: $O(b^{W^*/\epsilon})$, W^* is optimal cost, ϵ is is a value that no edge weights are less than
- Space complexity: $O(b^{W^*/\epsilon})$

Uniform Cost Search (UCS) is a deterministic search algorithm that effectively is like BFS, but for actions (read: edges) that have different costs or weights. Something to note about UCS is that it does not store a list of nodes that it has already visited. UCS is generally implemented with a priority queue, by which the item with (in this case) the smallest cost will be placed at the front of the priority queue. UCS will expand the current shortest path (the path at the head of the queue), and has no bias towards the goal. Psuedocode for UCS is below:

```
1:  $Q \leftarrow \langle start \rangle$
2:  while $Q \neq \emptyset$ do
3:      from $Q$, pick the path $P$ with the lowest cost $g = w(P)$
4:      if $head(P) = goal$ then
5:          return $P$
6:      end if
7:      for each vertex $v$ such that $(head(P), v) \in E$ do
8:          add $\langle v, P \rangle$ to $Q$
9:      end for
10: end while
11: return $Fail$
```

Exercise 7.2: Recall figure 7.1. Running UCS on this graph will give us a path $P = \langle g, d, a, s \rangle$ with cost = 8. Verifying this claim is left as an exercise to the reader.

We can realize that UCS is both complete and optimal, as it will always find the shortest path and will not miss any (like BFS).

Exercise 7.3: Prove that UCS is both complete and optimal.

Downsides of UCS include that due to its nature of expanding the current shortest path, in a graph where there are many vertices each connected by a small cost edge, UCS will expand this long string of short edges, only to find out in the end that there is a shorter way to get to the goal.

What if we were able to avoid this? What if we were to have some sort of magic function that tells us not to expand all of those short edges?

7.3.2 Best-first or Greedy Search

Properties

- Not complete
- Not optimal
- Time and Space complexity: $O(b^m)$, m

Greedy or Best-First Search uses the concept of a heuristic h. This function provides an estimate of the distance to the goal, such as by using Manhattan or Euclidean distance objectives. A greedy search algorithm works by expanding the path with the lowest heuristic cost first. Greedy Search is similar to DFS in that it keeps exploring until it has to go back due to a dead-end.

Psuedocode for Greedy Search is below:

```
1:  $Q \leftarrow \langle start \rangle$
2:  **while** $Q \neq \emptyset$ **do**
3:      from $Q$, pick the path $P$ with the lowest *heuristic* cost $h(head(P))$
4:      **if** $head(P) = goal$ **then**
5:          return $P$
6:      **end if**
7:      **for each** vertex $v$ such that $(head(P), v) \in E$ **do**
8:          add $\langle v, P \rangle$ to $Q$
9:      **end for**
10: **end while**
11: return $Fail$
```

Exercise7.4: Guided work-through of greedy search algorithm on figure 7.2:

Start

Path	Cost	Heuristic
$\langle s \rangle$	0	10

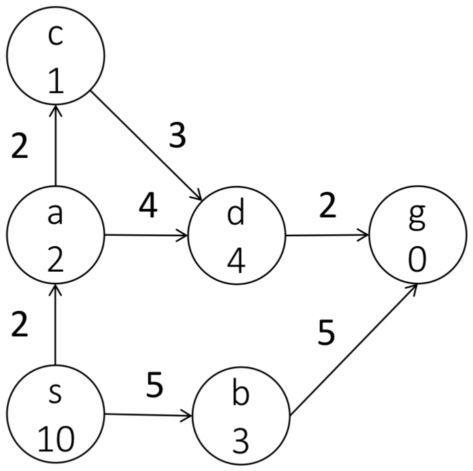


Figure 7.2
Simple graph, now with heuristic

Adding neighbors and removing $\langle s \rangle$:

Path	Cost	Heuristic
$\langle a, s \rangle$	2	2
$\langle b, s \rangle$	5	3

Adding neighbors and removing $\langle a, s \rangle$:

Path	Cost	Heuristic
$\langle c, a, s \rangle$	4	1
$\langle b, s \rangle$	5	3
$\langle d, a, s \rangle$	6	4

Adding neighbors and removing $\langle c, a, s \rangle$:

Path	Cost	Heuristic
$\langle b, s \rangle$	5	3
$\langle d, a, s \rangle$	6	4
$\langle d, c, a, s \rangle$	7	4

Adding neighbors and removing $\langle b, s\rangle$:

Path	Cost	Heuristic
$\langle g, b, s\rangle$	10	0
$\langle d, a, s\rangle$	6	4
$\langle d, c, a, s\rangle$	7	4

And now we are done, with returning the path $\langle g, b, s\rangle$. Notice that while this is a valid path, it is not the right path for the shortest path from s to g.

Therefore, although greedy search can be quite fast, it is not complete nor optimal. **Exercise7.4**: Create an example where greedy search gets stuck

7.3.3 A & A* search (and heuristics)

7.3.3.1 A search *A search* is different from both greedy and uniform cost search in that it keeps track of both the cost of the path so far $g(v)$, as well as the *heuristic* (AKA: estimate of cost to go) $h(v)$. From these two functions, we form an path cost estimation $f(v) = g(v) + h(v)$.

```
$Q \leftarrow \langle start, goal, h, V, E, w\rangle$
while $Q \neq \emptyset$ do
    from $Q$, pick the path $P$ with the lowest estimated cost $f(P)$, where the cost $f(P) = g(P) + h(head(P))$
    if $head(P) = goal$ then
        return $P$
    end if
    for each vertex $v$ such that $(head(P), v) \in E$ do
        add $\langle v, P\rangle$ to $Q$
    end for
end while
return $Fail$
```

Exercise7.4: Guided work-through of A search algorithm on figure 7.3:

Start

Path	g	h	f
$\langle s\rangle$	0	10	10

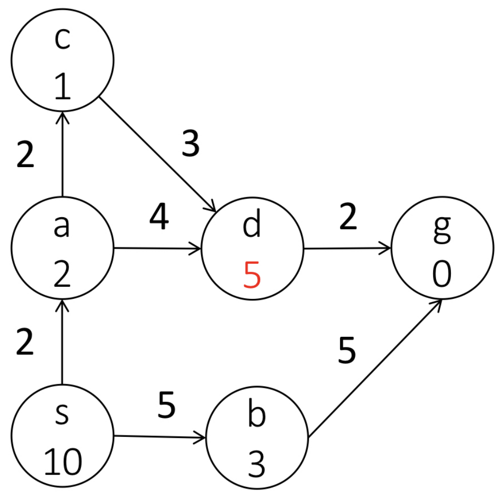


Figure 7.3
Simple graph, now with heuristic

Adding neighbors and removing $\langle s \rangle$:

Path	g	h	f
$\langle a, s \rangle$	2	2	4
$\langle b, s \rangle$	5	3	8

Adding neighbors and removing $\langle a, s \rangle$

Path	g	h	f
$\langle c, a, s \rangle$	4	1	5
$\langle b, s \rangle$	5	3	8
$\langle c, a, s \rangle$	6	5	11

Adding neighbors and removing $\langle c, a, s \rangle$

Path	g	h	f
$\langle b, s \rangle$	5	3	8
$\langle c, a, s \rangle$	6	5	11
$\langle d, c, a, s \rangle$	7	5	12

Adding neighbors and removing $\langle b, s\rangle$

Path	g	h	f
$\langle g, b, s\rangle$	10	0	10
$\langle c, a, s\rangle$	6	5	11
$\langle d, c, a, s\rangle$	7	5	12

And now we are done. So, A search is complete, but is the path $\langle g, b, s\rangle$ optimal? No it is not. Why? Remember, that if $h(v) = 0$, then A search is effectively the same as UCS. Look at the heuristic at vertex d in figure 7.3.
The heuristic estimates a cost of 5, whereas the true cost is 2. This leads into the concept of heuristic admissibility.

7.3.3.2 A^* search Let us define a $h^*(v)$ as the exact cost remaining to go. Then, we want our heuristic function $h(v) \leq h^*(v)$. In other words, the heuristic should not overestimate the actual remaining cost. If a heuristic $h(v)$ satisfies this condition, it is said to be *admissible*. If a heuristic is admissible, **A search becomes A^* *search***, and is now guaranteed to be optimal.

Suppose we change the heuristic of vertex s to 6 and the of vertex d to 1 in figure 7.3. What do you notice now? The heuristic is now admissible.
Exercise 7.6: Prove the optimality of A^*.

What are examples of admissible heuristic functions? $h(v) = 0$ is an admissible heuristic, but this particular heuristic changes A^* into UCS. We can do better. Better examples could be $h(v) = distance(v, g)$, where the vertices are physical locations is an admissible heuristic, as well as is the $h(v) = ||v - g||_p$ in a normed vector space. **Exercise 7.7**: What is another example of an admissible heuristic? An inadmissible heuristic?

7.3.3.3 A word on heuristics We have already discussed admissibility in the previous section, particularly that an admissible heuristic will not overestimate the true remaining cost to the goal.
Partial order of heuristics:

We say that a heuristic h_1 is dominated by another heuristic h_2 if $\forall v \in V$, $h_1(v) < h_2(v)$. The heuristic h^* dominates all admissible heuristics, and the 0 heuristic is dominated by all admissible heuristics. As a general rule, we want to chose a heuristic h such that is as close as possible to h^*.
Consistency:

A heuristic h is consistent if $\forall$edges $(u, v) \in E$, for some vertices u and v, $h(u) \leq h(v) + w(u, v)$. It follows that the total path cost estimation function $f(v) \geq f(u)$.

7.3.4 Hybrid A*

At the beginning of this chapter we alluded to a way to ”smooth” a discrete path in order to make it more feasible for the dynamics of a vehicle to traverse, as in figure 7.5. This is the idea of A^* search. Read the paper by Sebastian Thurn, et. al on hybrid A* *here*.

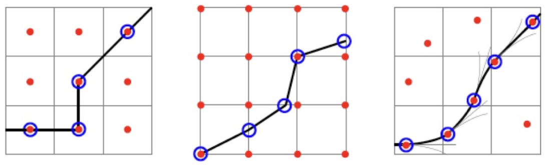


Figure 7.4
A* to D* to Hybrid A*

A^* search associates a cost with a cell center. However, this result sin a discrete planned path. D^* goes a step farther and allows for linear paths between cells by associating a cost with cell corners, instead of cell centers. Finally, *hybrid* A^* associates a continuous state with each cell.

Lets bring back our 4-D coordinate state space that we mentioned in the first section, $\langle x, y, \theta, dir \rangle$. As seen in figure 7.5, the current state is associated with cell c_i. Upon a

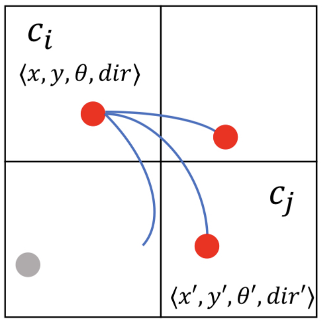

c_i
$\langle x, y, \theta, dir \rangle$
c_j
$\langle x', y', \theta', dir' \rangle$

Figure 7.5
hybrid A*

control input u, the resulting state is $\langle x', y', \theta', dir' \rangle$ in c_j. While Hybrid A^* will always give a realizable path, it is not complete. A coarser discretization makes hybrid A^* more likely to fail.

7.4 Sampling-based Planning - Non-deterministic Planning

A sample-based planner has solutions that are based on samples from a distribution. They can have probabilistic completeness, can work with general dynamics, and don't have explicit constraints.

7.4.1 The Motion Planning Problem

The motion planning problem gets our vehicle from some point A to another point B whilst avoiding obstacles. These sampling- based algorithms also incorporate the dynamical constraints of the vehicle. Consider a standard ODE

$$\frac{dx}{dt} = f(x, u), x(0) = x_{init}$$

which represents a control system, where x is a state and u is the control input. With an obstacle set X_{obs} and goal set X_{goal} (both $\subset R^d$), the goal is to find a u such that

- $\forall t \in R_{\geq 0}, x(t) \notin X_{obs}$
- for some finite $T \geq 0, \forall t > T, x(t) \in X_{goal}, 0$
- Fail if there is no u that exists.

There are many types of planners. Discretization and graph search planners, such as the search algorithms we have already discussed, are not easily scale-able to higher dimensions. Algebraic planners provides an explicit representation of obstacles, and is complete, but not practical in most case. Potential fields / navigation functions have attractive forces toward the goal and repulsive forces away from obstacles. These do not have a completeness guarantee unless there are navigation functions involved. High computational difficulty.

7.4.2 Probabilistic Road Maps (PRM)

Properties

- Multi-query
- Probabilistic Completeness
- Not asymptotically optimal
- Complexity for N samples $\Theta(N^2)$. Can be faster with e.g. k-Nearest Neighbor.

PRM is a multi query algorithm - meaning that it can find a path from any point to any other point. PRM is split up into a reprocessing and a run time phase. In the prepossessing phase, we build a graph by sampling n points from the free space $X_{free} = [0, 1]^d \mathcal{X}_{obs}$ (basically anywhere in the space that isn't inside an obstacle). Check out this animation of the prepossessing phase on *Wikipedia*.

The psuedocode for a simple PRM construction:

```
1: $V \leftarrow \{x_{\text{init}}\} \cup \{\text{SampleFree}\}_{i=1...N-1}$
2: $E \leftarrow \emptyset$
3: for each vertex $v \in V$ do
4:     $U \leftarrow Near(G = (V, E), v, r) \setminus \{v\}$
5:     for each vertex $u \in U$ do
6:         if $CollisionFree(v, u)$ then
7:             $E \leftarrow E \cup \{(v, u), (u, v)\}$
8:         end if
9:     end for
10: end for
11: return $G = (V, E)$
```

The function $Near(G, v, r)$ finds the subset of vertices in G that are within r distance of v. The $CollisionFree(v, u)$ function checks whether or not there is a path from u to v that doesn't collide with any obstacles, taking into account physics and the vehicle dynamics.

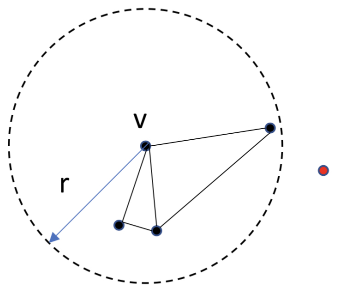


Figure 7.6
Points within the circle returned by $Near(\cdot)$

PRM is guaranteed to be sound. If you sample enough, e.g. N is large enough, then PRM is probabilistically complete. PRM isn't guaranteed to find the optimal solution because it is randomized depending on sampling.

7.4.2.1 Probabilistic completeness A motion planning problem $P = (X_{\text{free}}, x_{\text{init}}, X_{\text{goal}})$ is *robustly feasible* if $\exists$ some small $\delta > 0$ such that a solution remains a solution if obstacles are dilated by δ.

Consider figure 7.7. It is not robust because if the X_{obs} space is dilated by a small non-negative *delta*, the path to the goal state will be cut off. It is important to note that the

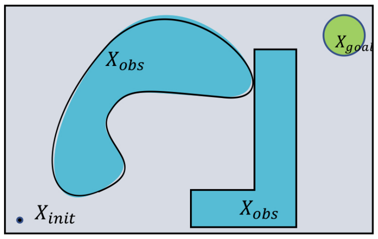


Figure 7.7
A non-robust planning problem

key to robustness is a small δ. An arbitrary size δ would make all planning problems non robust. Instead it is intended that δ is a tiny value, where dilating the $X_{textobs}$ space by even the tiniest amount makes the solution we had no longer the solution. A good example of this is an asymptote. An asymptote itself does not touch whatever it is an asymptote of, but adding the smallest of values to it would make make it touch.

We say that an algorithm *ALG* is *probabilistic completeness* if, for any robustly feasible motion planning problem defined by $P = (X_{\text{free}}, x_{\text{init}}, X_{\text{goal}})$, $\lim_{N\longrightarrow\infty} Pr(ALG$ returns a solution to $P) = 1$. Here the probability is coming from the sampling that is being done.

7.4.2.2 Asymptotic optimality An algorithm *ALG* is *asymptotically optimal* if, for any motion planning problem $P = (X_{\text{free}}, x_{\text{init}}, X_{\text{goal}})$ and cost function c that admit a robust optimal solution with finite cost c^*,

$$P\Big(\{\lim_{i\longrightarrow\infty} c(Y_i^{ALG}) = c^*\}\Big) = 1$$

.

Exercise 7.1. Explore why PRM is not asymptotically optimal.

7.4.3 Rapidly Expanding Random Trees (RRT)

Properties

- Single-query
- Not asymptotically optimal

The idea of *RRT* is to build in real time a tree, exploring the region of the state space that can be reached from the initial condition. At each step of the algorithm, one point is sampled from X_{free} (all space outside the obstacles), and connect the closest vertex to this sampled point. This is what enables the tree to grow.

```
1:  $V \leftarrow \{x_{\text{init}}\}$
2:  $E \leftarrow \emptyset$
3:  for $i = 1, ..., N$ do
4:      $x_{\text{rand}} \leftarrow SampleFree_i$
5:      $x_{\text{nearest}} \leftarrow Nearest(G = (V, E), x_{\text{rand}})$
6:      $x_{\text{new}} \leftarrow Steer(x_{\text{nearest}}, x_{\text{rand}})$
7:      if $ObstacleFree(x_{\text{nearest}}, x_{\text{new}}$ then
8:          $V \leftarrow V \cup \{x_{\text{new}}\}$
9:          $E \leftarrow E \cup \{x_{\text{nearest}}, x_{\text{new}}\}$
10:     end if
11: end for
12: return $G = (V, E)$
```

The function $Obstaclefree(\cdot)$ is analogous to $CollisionFree(\cdot)$ in PRM. The Steer Function takes into account the vehicle dynamics to determine whether it is feasible to go from one point to another. There is a bias towards sampling points that are closer to X_{goal}, called *Voronoi Bias*.

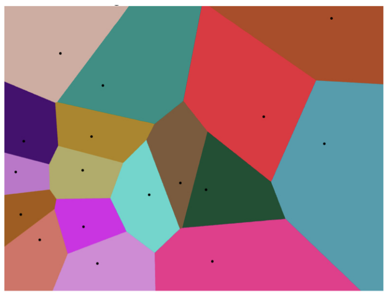


Figure 7.8
A Voronoi Diagram

7.4.3.1 Voronoi Diagram and Bias Given $n \in \mathbb{R}^d$ points, the *Voronoi Diagram* of the points ia partition of $\mathbb{R}^d$ into regions of one point each, such that the point inside a region is closer to that entire region than any other point. The vertices of RRT that are more ”isolated” have larger Voronoi regions, and are more likely to be selected for extension, hence the bias of expansion towards the goal state.

7.4.4 Rapidly Exploring Random Graphs

Rapidly Exploring Random Graphs (RRG) tries to connect the new sample x_{new} to all vertices in a ball of radius r_n centered at it.

```
$V \leftarrow \{x_{\text{init}}\}$
$E \leftarrow \emptyset$
for $i = 1, ..., N$ do
    $x_{\text{rand}} \leftarrow SampleFree_i$
    $x_{\text{nearest}} \leftarrow Nearest(G = (V, E), x_{\text{rand}})$
    $x_{\text{new}} \leftarrow Steer(x_{\text{nearest}}, x_{\text{rand}})$
    if $ObstacleFree(x_{\text{nearest}}, x_{\text{new}}$ then
        $X_{\text{near}} \leftarrow Near(G = (V, E), x_{\text{new}}, \min\{\gamma_{RRG}(\log(|V|)/|V|)^{1/d}, \eta\})$
        $V \leftarrow V \cup \{x_{\text{new}}\}$
        $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}}), (x_{\text{new}}, x_{\text{nearest}})\}$
        for each$x_{\text{near}} \in X_{\text{near}}$ do
            if $CollisionFree(x_{\text{near}}, x_{\text{new}})$ then
                $E \leftarrow E \cup \{(x_{\text{near}}, x_{\text{new}}), (x_{\text{new}}, x_{\text{near}})\}$
            end if
        end for
    end if
end for
return $G = (V, E)$
```

References

Canny, John F. 1986. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 8 (6): 679–698. http://dblp.uni-trier.de/db/journals/pami/pami8.htmlCanny86a.

Dalal, Navneet, and Bill Triggs. 2005. Histograms of oriented gradients for human detection. In *2005 ieee computer society conference on computer vision and pattern recognition (cvpr'05)*, Vol. 1, 886–893. Ieee. Ieee.

Fabris, Simone. 2012. Method for hazard severity assessment for the case of undemanded deceleration, Technical report, TRW Automotive.

ISO. 2011. Road vehicles—functional safety, Technical Report ISO 26262, International Organization for Standardization (ISO).

Lowe, David G. 1999. Object recognition from local scale-invariant features. In *Proceedings of the seventh ieee international conference on computer vision*, Vol. 2, 1150–1157. Ieee. Ieee.

Paden, Brian, Michal Cáp, Sze Zheng Yong, Dmitry S. Yershov, and Emilio Frazzoli. 2016. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on Intelligent Vehicles* 1 (1): 33–55. doi:10.1109/TIV.2016.2578706.

Perry, Raleigh B., Michael M. Madden, Wilfredo Torres-Pomales, and Ricky W. Butler. 2013. *The simplified aircraft-based paired approach with the ALAS alerting algorithm*, Technical Report NASA/TM-2013-217804, NASA, Langley Research Center.

Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. 2005. *Probabilistic robotics*. *Intelligent robotics and autonomous agents*. MIT Press.

Index