Contents

Chapter 1

Sense, Think, Act

When you are given a task to perform, for example, place the chairs in the room into a circular pattern, you might proceed as follows. First, look around the room and assess the situation. How many chairs are there? Where are they located? Second, you might make a plan for putting the chairs into the desired pattern: Go to the first chair, pick it up, and move it to its desired location; then, repeat this for the second chair, and so on. Finally, execute the plan, and arrange the chairs as desired. This basic strategy is often referred to as the Sense-Think-Act paradigm, and it forms the basis for the behavior of most all robotic systems.

In robotic systems, *sensing* might be performed using computer vision, touch sensors, laser range scanners, or any number of currently available sensing modalities. *Thinking* corresponds to the process of manipulating an internal model of the world, reasoning about which actions the robot could perform, and how these actions would change the state of the world. Finally, *acting* is the moment when the robot moves in, and interacts with, its environment.

For real robotic systems, it is almost never the case that a robot succeeds at performing a task by using a single instance of this sense-think-act process. Rather, these steps are typically performed iteratively. At each iteration, sensing updates the robot's understanding of the current situation; thinking is used to update or refine the current plan; actions are then executed to bring things a bit closer to the desired outcome. For this reason, one typically refers to the *sense-think-act loop*. The speed at which this loop is executed varies based on the task to be performed. For a chess playing robot, a single iteration could take seconds or minutes (each iteration corresponding to a single move). For a self-driving car, a single iteration of sense-think-act might take milliseconds (or less). In the limit, as the time associated to a single iteration decreases to zero, we obtain continuous time systems. While the specific behavior of these various systems can be quite different, they are all nicely described in terms of the sense-think-act loop, and for this reason, we use this paradigm as the organizing structure for our development.

In this chapter, we introduce concepts that are fundamental to the study of intelligent robots, particularly those that rely on the sense-think-act paradigm. We introduce the concept of world state, and describe various possible methods that can be used to represent the world state. We then introduce the notion of the robot's state, which is subtly different from the state of objects in the world, due mainly to the fact that the robot can directly

change its own state. The robots actions and how to represent them are the topic of the subsequent section. Finally, we give a brief overview of sensing methodologies that are currently of interest to intelligent robotic systems.

1.1 Representing the Robot's Environment

In order to reason about the world and about its own actions in the world, a robot requires some sort of representations of both itself, and the the world that it inhabits. For any specific robotic system, the system designer must decide what to represent, and which representational scheme should be used. Furthermore, different representation schemes might be used for different aspects of a particular problem. For example, if a robot house keeper is charged with doing the laundry, reasoning about moving from the bedroom to the laundry room versus reasoning about folding the laundry would require fundamentally different types of representations. For the former, a high-level description of the layout of rooms in the house might suffice, while for the latter, the robot might need to model articles of clothing as nonrigid, deformable objects. In the chapters that follow, we will explore a variety of representational schemes, ranging from high-level, discrete abstractions to low-level continuous representations.

The robot's information about its environment is generally referred to as the *world state*. For a chess playing robot, this might include a complete list of the positions of all chess pieces on the board. For a house-keeping robot, the world state might include a map of the house, locations of furniture, and locations of various household objects. In both cases, the world state excludes many details about the world that are not directly relevant to the robot's objectives. For example, a house-keeping robot might not need to know the colors of the walls, the thermostat setting for the bedroom, or the titles of books on a shelf. The key idea of *world state* is that it should include the information necessary for the robot to understand the environment well enough to successfully perform its tasks.

How to represent the world state depends on the kind of information that is required. High-level, symbolic representations are often sufficient for the purpose of constructing general robot plans. A classical example is the STRIPS system for robot planning in a simple *blocks world*. Figure 1.1 shows a simple example. In STRIPS, the state of the world is represented by symbolic relations, such as those shown in Figure 1.1b. This simple set of relations tells us that Blocks A and C are resting on the table, that Block B rests on the top of Block A, and that nothing rests atop either Block B or Block C. Equipped with such a description, STRIPS can create plans, for example, to place Block A atop Block C by picking up Block B and placing it on the table, then picking up Block A and placing it atop block C. This plan excludes any geometric description of how to perform the task, but it provides a high-level description of the sub-tasks that the robot must execute to accomplish its end goal. In later chapters, we will see how this kind of planning works in detail.

While this kind of high-level, qualitative state description may be useful for task-level planning, because it fails to capture any of the geometric aspects of the environment, it would be insufficient when the robot begins to actually move in, and interact with, its environment. For mobile robots, it is often sufficient to use a discrete grid to represent which parts of the environment contain objects, and thus cannot be traversed by the robot.

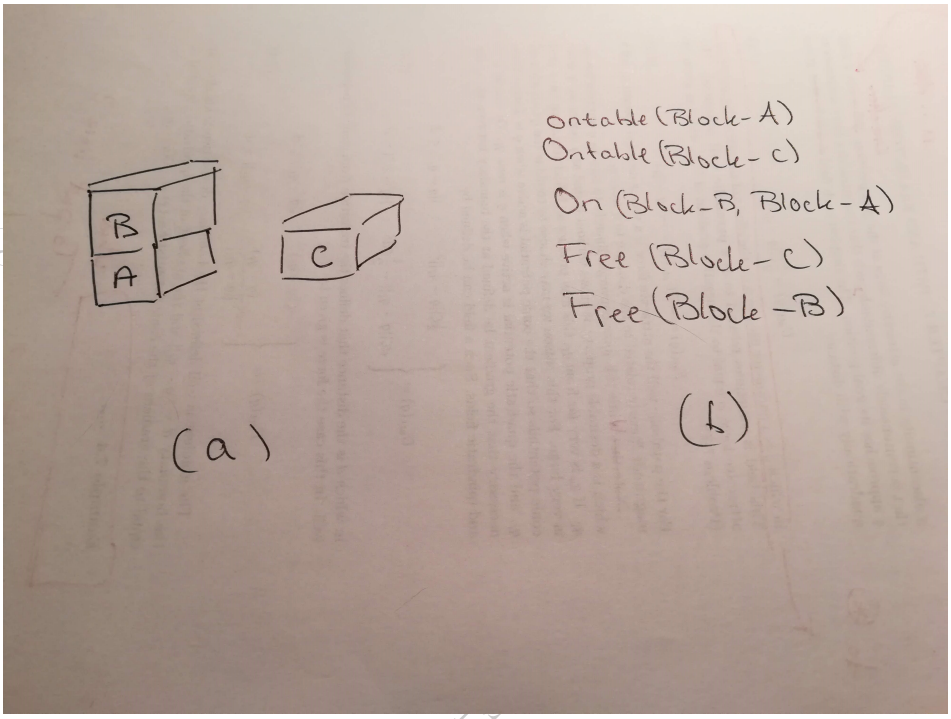
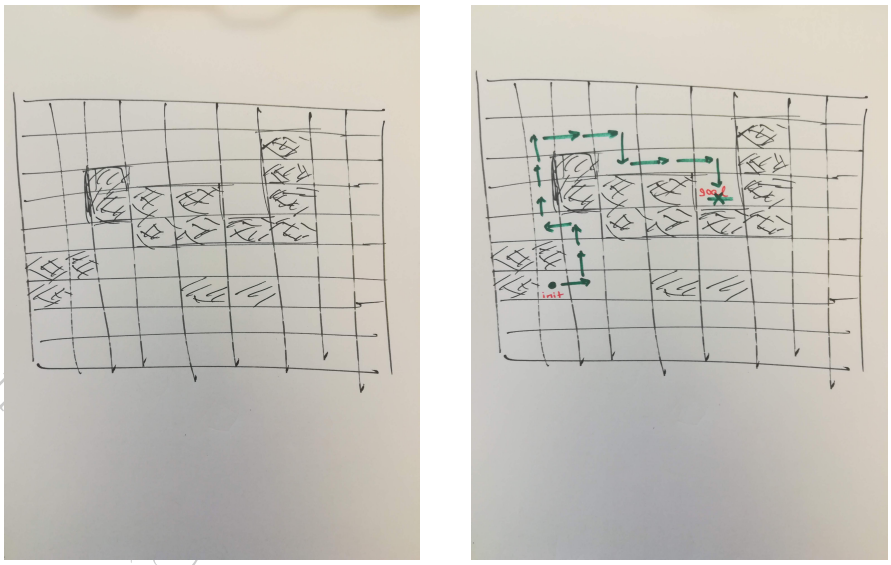


Figure 1.1: An example from the Blocks World. (a) A simple blocks world scene. (b) The symbolic description of the world state.

Because such objects impede the ability of the robot to move freely, we typically refer to these as *obstacles*. The prototypical path planning problem in robotics is to find a path for the robot from its initial location to a specified goal location, while avoiding collision with any obstacles in the environment. Figure 1.2(a) shows an example of an *occupancy grid*, a grid-based map that explicitly indicates which grid cells are occupied by obstacles. If we assume that the robot is able to move to any adjacent empty grid cell that is directly above, below, left, or right of its current location, planning can be accomplished using graph search methods. Figure 1.2(b) illustrates a path in the occupancy grid. In later chapters we will describe how to build an occupancy-grid map from sensor data, how the robot can determine its location in the map, how graph search algorithms can be used to construct a path from start to goal, and how the robot can execute this plan to navigate in its environment.

In some cases, for example if task requires manipulating objects in the environment, a more precise geometric description of the state may be required. Suppose, for example, that a chess playing robot wishes to move its king. In this case, for the robot to grasp the king, its precise location must be known. There are several ways to represent this kind of geometric information, but the most common is to define a Cartesian coordinate frame that is *rigidly attached* to the king. This merely means that the relationship between the king and this coordinate frame is fixed, and does not change when the king is moved. One such possible assignment is shown in Figure 1.3. In order to grasp the king, the robot would perform appropriate geometric computations to bring its fingers to specific positions relative to this coordinate frame. This requires, of course, knowing the precise position and orientation of the king's coordinate frame relative to the robot, information that can be obtained using



(a) (b)

Figure 1.2: Using an occupancy grid to represent the environment of a mobile robot. (a) Shaded cells are occupied by obstacles. (b) A plan to move from the initial to goal cell in the occupancy grid.

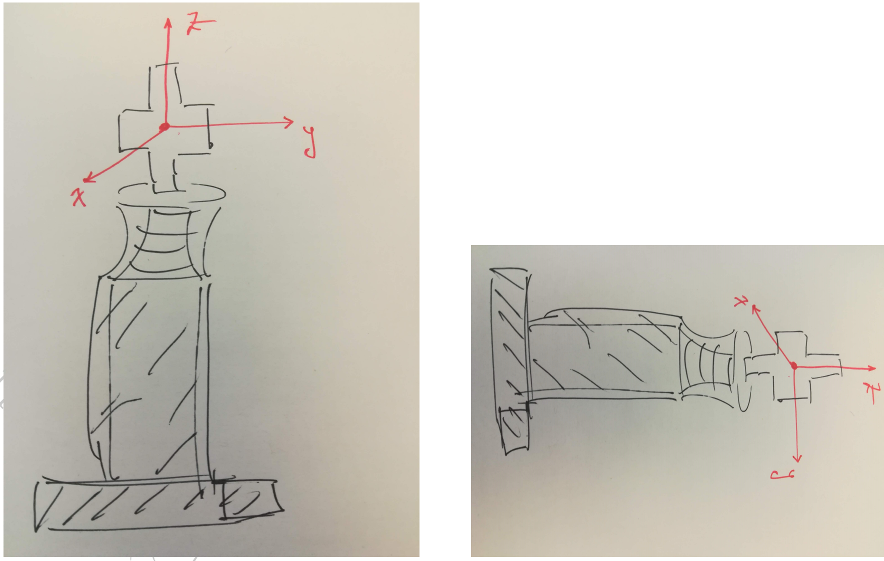
the robot's sensors. In later chapters, we will describe in detail the geometric computations required for this kind of task, as well as how sensors can be used to determine relevant geometric aspects of the world, including the positions and orientations of objects in the robot's work space.

It is often the case that robots must interact with moving objects (e.g., parts on a conveyor belt, other robots). For these situations, it is often advantageous to explicitly include the notion of time in the representation. For example, a robot that plays table tennis should be able to estimate not only the instantaneous position of the ball at any moment in time, but also the trajectory of the ball, thus enabling the prediction of where the ball will be at future moments in time. In this case, if we represent the coordinates of the ball by a vector $x \in \mathbb{R}^3$, we make explicit the dependence on time by writing $x(t)$. Furthermore, if we are interested also in the velocity of the ball, we write

$$\dot{x}(t) = \frac{d}{dt}x(t)$$

to denote the time derivative of the ball's position. Under the laws of Newtonian physics, the position and velocity of the ball at any moment in time completely determine the future trajectory of the ball (assuming no effects of wind, etc.). In the vocabulary of physics, the *state* of a system is a collection of information sufficient to determine the entire future evolution of the system behavior. For this reason, in physics, one refers to the pair $(x, \dot{x})$ as the state of the ball. Our use of the term *state* is somewhat more general than that used to describe physical systems; however the intuition behind both terms is essentially the same.

As an example, Figure 1.4 illustrates basic projectile motion. If the position and velocity



(a) (b)

Figure 1.3: A chess piece with an attached Cartesian coordinate frame. (a) The frame origin is located at the center of the cross, the z-axis is aligned with the main axis of the body, and the y-axis lies in the plane containing the cross. The x-axis completes a right-handed coordinate frame (b) The frame is rigidly attached to the chess piece, and moves when the piece moves.

of the ball are both known at either time t_0 or at time t_1, then it is possible to predict the position and velocity of the ball at any future moment in time. Although this example is fairly simple, in many robotics applications the a system's state is represented using position and velocity. This idea applies not only to various moving objects in the environment, but even to the motion of the robot itself. This is the case for robot arms, whose motion depends on torques generated by motors, and for unmanned air vehicles (UAVs) such as quadrotors, whose motion depends on aerodynamic forces generated by spinning propellers.

It is often the case that discrete time representations are preferable to using continuous time. This is because many algorithms used in robotics rely on numerical methods to compute solutions. This should not be surprising, since computer algorithms are by their nature discrete-time entities. In this case, we use the notation x_t to denote the value of the state x at time instant t. Often, we can compute exact discrete-time system representations by integrating an appropriate description of the system dynamics, such as

$$x_{t+1} = x_t + \int_t^{t+1} \dot{x}(t) dt$$

It is often the case that the robot does not have access to a complete and correct model of its environment. For example, if sensors are used to determine the world state, there will invariably be errors and uncertainties associated to the sensor measurements. There are a variety of ways that one can deal with such uncertainties. We can include uncertainty explicitly in our representations, e.g., using tools from probability theory, or we

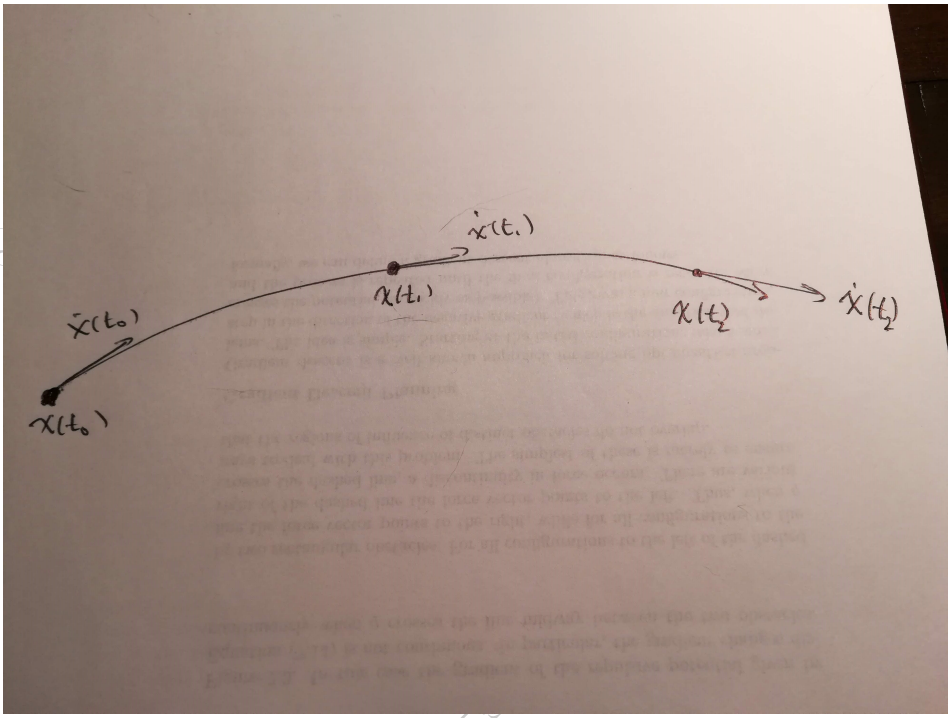


Figure 1.4: The motion of a ball follows the arc of a parabola. Given the position and velocity at any moment in time, it is possible to exactly predict the position and velocity at any future moment in time.

can incorporate uncertainty into our model of the robot's actions in the world. We will consider both of these options in later chapters.

1.2 Representing the State of the Robot

While the robot is, technically, an object within the world, it enjoys the special status of being able to act in the world to effect changes. Furthermore, the robot has direct control over its own actions, unlike obstacles or other actors in the world, over which the robot has, at best, indirect control. Therefore, rather than merely incorporate information about the robot into the world state, we typically represent the robot state separately, using representations that are specifically developed for modeling the robot's geometry, dynamics, and manipulation capabilities.

The most basic information about a robot's state is merely a description of the robot's location in its environment. Four examples are shown in Figure 1.5. For a vacuum cleaning robot, Figure 1.5a, this could be a set of x, y coordinates of the robot's centroid with respect to a floor plan of the house. For a robot arm, Figure 1.5b, we might specify a vector of joint angles. For a UAV, Figure 1.5c, we might specifiy the x, y, z coordinates of the vehicle, along with its orientation. For a humanoid robot, Figure 1.5d, we might specify the position and orientation of the torso's centroid, along with a vector of angles for each of the robot's joints. In robotics, all of these correspond to the general notion of a robot's *configuration*. More precisely, a configuration q of the robot provides a complete specification of the location of

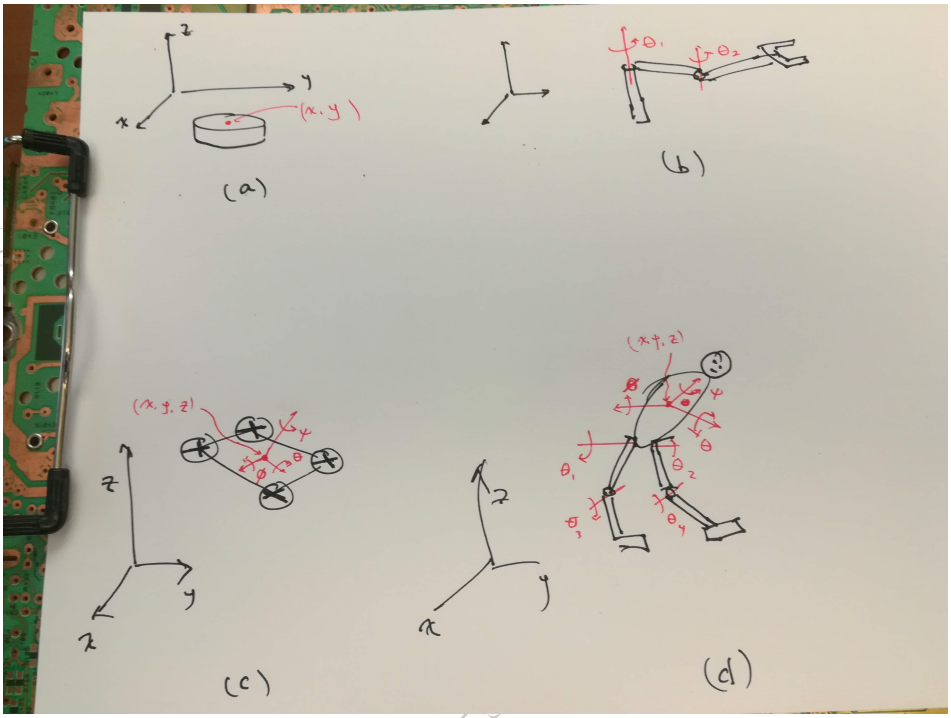


Figure 1.5: Configurations for for different robots. (a) A vacuum cleaning robot whose configuration is specified by x, y coordinates in the world frame. (b) A two-link robot arm, whose configuration is specified by its two joint angles (θ_1, θ_2). (c) A quadrotor, whose configuration is specified by three position parameters x, y, z and three orientation parameters ϕ, θ, ψ. (d) A humanoid robot whose configuration is specified by the position and orientation of the body-attached frame $x, y, z, \phi, \theta, \psi$ and the joint angles in the legs $\theta_1, \theta_2, \theta_3, \theta_4$.

every point on the robot with respect to a reference frame. The set of all configurations is called the *configuration space*, and is denoted by $\mathcal{Q}$.

A common way to define a robot's configuration is to rigidly attach a coordinate frame to each component of the robot that can move, and to then specify the position and orientation of each of these frames. Of the examples given above, we can consider the vacuum cleaning robot and the UAV to be single rigid objects. For the vacuum cleaning robot, assuming that the orientation of the robot is not of concern, we would have $q = (x, y)$, for a UAV we might use $q = (x, y, z, \phi, \theta, \psi)$, where the latter three parameters specify the orientation of the three axes of the UAV's body-attached coordinate frame with respect to a reference frame.

For many robots, motion of the individual components is constrained by the design of the mechanical system. For example, the motion of any link in a robot arm is determined by the rotation of a single motor that connects this link to the previous link. In such cases, a single parameter (in this case a joint angle) is sufficient to specify the configuration of the link, and the configuration of the entire robot arm can be specified by $q = (\theta_1, \cdots, \theta_n)$, in the case of an arm with n revolute joints (i.e., each joint is driven by a rotating motor). More complex mechanisms require a combination of these methods. For example, the configuration of a

humanoid robot might be represented by $q = (x, y, z, \phi, \theta, \psi, \theta_1, \ldots, \theta_n)$, in which the first six parameters specify the position and orientation of a body-attached frame (e.g., located at the centroid of the torso), and $\theta_1, \cdots, \theta_n$ specify the angles for the individual joints of the robot. In later chapters, we will investigate the configuration spaces for a variety of robots, from simple wheeled mobile robots to more complex mobile manipulators comprised of moving platforms with an attached manipulator arm.

The configuration of a robot answers the question of where the robot is at a specific instant in time. If we wish instead to describe the motion of a robot, we must consider the configuration to be time varying, and in this case both the configuration and its time derivative (a velocity) are relevant. As was the case above for moving objects, we often package the configuration and its time derivative into a single vector

$$x(t) = \begin{bmatrix} q(t) \\ \dot{q}(t) \end{bmatrix}$$

In many disciplines related to robotics, x is referred to as the state. This is particularly true in the areas of dynamical systems and control theory. In this text, we will maintain a more general use of the term *state*, but when relevant, we will adopt discipline-appropriate terminology (e.g., when describing how to control a robot's motion).

In many applications, position and velocity provide a sufficiently detailed description of robot motion. This is not true, however, when we must explicitly consider forces that affect the robot's motion. For example, we can essentially regard a vacuum cleaning robot as a device that responds to position and velocity commands: we issue a command to the robot to move to a certain position at a certain velocity, and the robot has no difficulty in executing this command. There are, however, numerous applications in which simple geometric descriptions of robot motion are not adequate. Consider for example the case of a quadrotor that maneuvers by exploiting aerodynamic forces, or a humanoid robot whose locomotion depends on interaction forces between its feet and the ground. In these cases, we typically consider position, velocity, and acceleration, i.e., in terms of x and $\dot{x}$, or, if making the configuration and its derivatives more explicit, in terms of $q, \dot{q}$, and $\ddot{q}$.

1.3 Robot Actions

Robots are of interest to us because they have the ability to move in their environment, manipulate objects, and generally effect changes in the world state. In order for a robot to plan actions that achieve its goals, it must have a representation of its own actions, along with a characterization of how those actions affect the state of the world. The specific choice of representation depends on the task at hand, and the level of abstraction at which the task is represented.

For high-level, task planning, it is often sufficient to use a high-level representation scheme such as that described above for the STRIPS style planners. In this case, the representation of an action includes a set of conditions that must be true for the action to be applicable, and a set of changes that will result if the action is executed. Consider the example of Figure 1.1. We can define an action that causes the robot to grasp a block that is resting on another block as follows:

Action: *UNSTACK(?X, ?Y)*
Preconditions: On(?X, ?Y)
Delete: On(?X, ?Y)
Add: InGripper(?X), Free(?Y)

In this example, the symbols *?X* and *?Y* denote variables that are instantiated based on which objects are the target of the desired action. The *preconditions* provide a set of conditions that must hold in the world state in order for this action to be valid. The *add* and *delete* lists specify the changes to the world state that will occur once the action is applied.

Maybe these Blocks World examples should explicitly use PDDL

As might be expected, for each of the kinds of state representation described above, there is a corresponding action representation. For the case of a robot that moves in a grid (as in Figure 1.2), we might denote the robot state at time t as $x_t = (r_t, c_t)$, in which r, c denote a row and column index into the grid. We could specify a unique action for each of the four possible directions of motion in the grid, but it is more convenient in this case to define actions to be a function of a control input, as $x_{t+1} = f(x_t, u_t)$, in which u_t denotes the control action that is applied at time t. Here there are four possible choices for u, corresponding to moving up, down, left, or right. For example, when moving to the left, we would have

$$x_{t+1} = \begin{bmatrix} r_{t+1} \\ c_{t+1} \end{bmatrix} = f(x_t, \text{left}) = \begin{bmatrix} r_t \\ c_t + 1 \end{bmatrix}$$

For systems described as $x_{t+1} = f(x_t, u_t)$, the control input u provides the connection between planning and acting. Planning can be characterized as the search for a set of control inputs $u_1, u_2, \ldots$ that will transform the world (including the robot itself) from its initial state into the desired goal state. For discrete-time systems that are represented using discrete states, this can often be accomplished using various graph search algorithms. We discuss some of these algorithms in subsequent chapters.

For the case of continuous time robot systems that evolve on continuous state spaces, we typically write the dynamic equations of motion as

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

in which $x(t) = (q(t), \dot{q}(t))$ and $\dot{x}(t) = (\dot{q}(t), \ddot{q}(t))$ and u denotes a control input. The system dynamics for different robots take a variety of special forms. For example, a linear systems whose behavior does not vary with time (so called *linear, time-invariant* or *LTI* systems) can be written as

$$\dot{x}(t) = Ax(t) + Bu(t)$$

This is the most basic kind of continuous time dynamical system, yet such systems can often be used to provide reasonably good approximations to even fairly complex nonlinear systems.

Finding a suitable control input $u(t)$ to drive an arbitrary nonlinear system to its goal state can be a very difficult problem. One way to solve this problem is to discretize the system, and use numerical optimization methods to find a discrete-time control law that will achieve the goal. We discuss these issues, along with other specific forms of system dynamics for a variety of robot systems in subsequent chapters.

1.3.1 Some points to make

- actions have local effects – they affect only small parts of the overall world state. Therefore, we don't need to carry around the entire world state when we write descriptions of actions – we include only those parts of state that are directly relevant to the actions
- anything here about uncertainty? Either about how actions themselves are uncertain, or about how actions can be used to reduce uncertainty.

1.4 Sensing

This section would include a general overview of various sensing modalities, and the kinds of information that they provide.

Chapter 2

Simple Agents in Discrete Domains

2.1 Simple Robots in a Perfect World

- Robot moves in a grid world defined by an occupancy grid
- robot uses sensors to determine its state – lidar? (no noise, no uncertainty)
- Actions: move up, left, down, or right – but no collision allowed
- Planning is the problem of finding a path in the grid from start to goal. Use A* for this?

2.2 Probability and Bayes Nets

Motivation

Bayes nets give us a way to simulate robots. We will view simulation as drawing samples from a structured probability distribution. The probability distribution can be viewed as a stochastic world model, by describing how sensors reflect state, and how actions affect the state.

Our running example will be a robot that exists on a plane, and we will describe the robot's location with the 2D coordinate of a 10 by 10 grid that discretizes the space. The robot has a sensor to figure out where it is, and it's action space is limited to moving left, right, up, or down in the grid.

The random variables we use below can also be used to describe the state of a real robot, and the data structures we build below can be used to control and plan for a real robot. The discrete and coarse nature of our representation will manifest itself in being able to make only probabilistic statements about the robot's state, which is called inference. In this section we will do some of that, but will mostly be content with the modeling part.

Below we take a **Bayesian view of probability**, rather than a frequentist one. This means that we see probabilities as describing our knowledge about events, rather than tallying up frequencies by which they occur. Think of the weather-person talking about the probability of rain tomorrow. Probabilities viewed this way can be used to describe

prior knowledge about the state of the world, how sensors work, and how actions affect the state of an agent and the world.

As we introduce concepts below, we will also introduce a graphical way to represent these concepts, and as such gradually develop the general notion of a Bayes net, which is a graphical model to describe complex probability distributions.

2.2.1 Discrete Distributions

Figure 2.1: Graphical representation of a single random variable.

A **discrete probability distribution** is the probability distribution of a random variable X, taking on one of K discrete values. For example, we can model the action space of the robot with a random variable A, taking on values $a \in \{Left, Right, Up, Down\}$. Note that we denote random variables with a capitalized symbol, e.g., A for action, but when in a formula we talk about the *value* of a random variable, we use a lowercase a.

We can represent the parameters of a discrete probability distribution as a vector of probabilities $p_i \triangleq P(X = x_i)$, called the **probability mass function** or PMF. Probability mass functions obey two basic axioms,

$$p_i \geq 0$$
$$\sum p_i = 1$$

i.e., the probability p_i of an outcome x_i is non-negative, and the probabilities p_i of all outcomes have to sum up to 1.

Graphically, we can represent a PMF over a single random variable X as a single node, with the label X, as shown in Figure 2.1. By convention we use a circular node to represent random variables.

Example

Below is an example PMF describing the distribution over actions for our example robot. This encodes the knowledge that the robot seems more likely to move the right.

a_i	$P(A = a_i)$
Left	0.2
Right	0.6
Up	0.1
Down	0.1

Table 2.1: Probability mass function.

Grid-world example

To describe the location or state S of the robot, let us use 10×10 grid, in which case there are 100 possible outcomes. A PMF for the state S of the robot assigns a probability $p_{i,j}$ to each grid cell (i, j), and can encode our prior knowledge about where the robot could be, given no other information. Some examples for such a PMF include: uniform, a single particular starting position, more probable to be on the left, etc...

				.1					
	.1								
			.1						
.1				.1			.1		
		.1			.1				
						.1			
									.1

Table 2.2: A PMF describing where a robot might be in a grid.

An example is shown in Table 2.2, which models that given no other information, we expect to find the robot in one of 10 different possible locations with equal probability.

2.2.2 Sampling

Given a distribution, we can draw a sample from it. To do so, we use an algorithm called **inverse transform sampling**. For this we need to first assign an order to the outcomes, which we can do by adopting order associated with the (arbitrary) integer indices by which we enumerate outcomes, i.e.: $x_i < x_j$ if $i < j$. Given this order, we can the then compute the **cumulative distribution function** or CDF, which is the probability associated with the subset of outcomes with indices less than or equal to a given index i:

$$F(x_i) = P(X \leq x) = \sum_{j \leq i} P(X = x_i) = \sum_{j \leq i} p_j$$

Given this, the inverse transform algorithm is simple: generate random number $0 \leq u \leq 1$, then return x_i such that i is the smallest index such that $F(x_i) \geq u$.

Example

a_i	$P(A = a_i)$	$F(a_i)$
Left	0.2	0.2
Right	0.6	0.8
Up	0.1	0.9
Down	0.1	1.0

Table 2.3: Cumulative distribution function.

In Table 2.3, we have shown the CDF for the robot action, using the order from top to bottom. To sample a random value

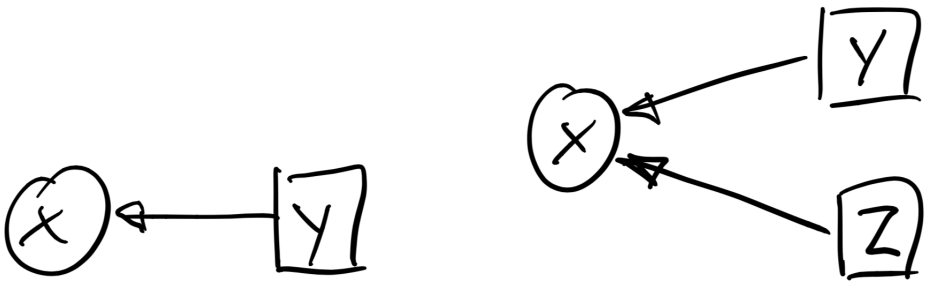
$u = 0.6$ would yield a sample $X = \mathit{Right}$.

Exercises

1. Implement inverse transform sampling for the CDF in Table 2.3, and verify your sampler produces a histogram that approximates the desired PMF.

2. Calculate the CDF for the grid world example in Table 2.2.

2.2.3 Conditional Distribution



(a) $P(X|Y)$ (b) $P(X|Y, Z)$

Figure 2.2: Conditional probability distributions. Above X is a random variable, but Y and Z are known parameters, indicated by the square box.

The probability mass function of a single variable X could be parameterized by a parameter Y, whose value we will assume as known for now. This corresponds to the notion of a conditional probability, which we write as

$$P(X|Y = y).$$

Note that given a particular value of Y, this is just a distribution over X, with parameters given by a PMF, as before. But, because Y can take on several values, we now need a **conditional probability table** or CPT to exhaustively describe our knowledge.

Graphically, we represent a known parameter Y as a square node within a graph, and a random variable governed by a conditional probability as having an incoming edge from the parameter, as shown in Figure 2.2a. Note that it is possible to have multiple parameters, e.g., a conditional probability $P(X|Y, Z)$ is shown in Figure 2.2b.

We can sample from a conditional distribution $p(X|Y)$ by selecting the appropriate PMF, depending on the value of Y, and proceeding as before using the inverse transform sampling method.

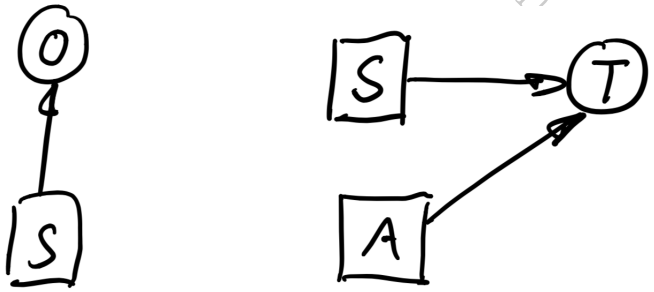
Example

a_i	$P(A = a_i \mid T = Left)$	$P(A = a_i \mid T = Right)$
Left	0.6	0.2
Right	0.2	0.6
Up	0.1	0.1
Down	0.1	0.1

Table 2.4: Conditional probability table (CPT).

Table 2.4 gives an example where we have a variable T to differentiate between robots tending to the left, or tending to the right.

2.2.4 Modeling the World



(a) Sensor Model $P(O|S)$

(b) State transition model $P(T|S, A)$

Figure 2.3: Conditional distributions to model sensing and acting.

Conditional probability distributions are a great way to represent knowledge about the world in robotics. In particular, we will use them to model sensors, as well as how we can affect the state of the robot by actions.

Assuming a robot has a single sensor, or we observe the robot from a different vantage point, we can use a random variable O to model an observation, and specify via a distribution $P(O|S = s)$ how the sensor behaves, given that we are in a particular state s. This is illustrated in Figure 2.3a. A complete **sensor model** specifies this in a (giant) CPT for every possible state. An observation O can be rather impoverished, or very detailed, and one can also envision modeling several different sensors on the robot. In the latter case, we will be able to *fuse* the information from multiple sensors.

Conditional probability tables do not *have* to be specified as giant tables. In case we index the discrete states with semantically meaningful indices, as in the grid-world example, we can often give the CPT in parametric form. For example, a simple sensor would simply report the horizontal coordinate j of the robot faithfully, in which case we have

$$\begin{cases} P(O = k|S = i, j) = 1 & \text{iff } k = j \\ P(O = k|S = i, j) = 0 & \text{otherwise} \end{cases}$$

However, the power of using probability distributions comes from the fact that we can model less accurate/faithful sensing. For example, here is a parametric model for a sensor that reports the vertical coordinate i of the robot, but with 9% probability gives a random faulty reading:

$$\begin{cases} P(O = k|S = i, j) = 0.91 & \text{iff } k = i \\ P(O = k|S = i, j) = 0.01 & \text{otherwise} \end{cases} \tag{2.1}$$

We can model the effects of actions in a similar manner, by a conditional probability distribution $P(T|S = s, A = a)$ on th next state T , given the value s of the current state S, and the value a of the action A, bit viewed as parameters. Because the state space is potentially quite large, such a **state transition model** is almost never explicitly specified, but rather exploits the semantics of the states and actions to provide a more compact representation of the associated CPT.

Exercise

Specify a parametric conditional density for the action models in the grid-world that is somewhat realistic, yet not completely deterministic.

Exercise

It is possible to create models that do not reflect everyday physics. For example, how could we model the game "Portal"?

2.2.5 Joint Distribution

If we consider the parameter Y in a conditional probability distribution not as a known parameter but as a random variable itself, with probability distribution $P(Y)$, we obtain a

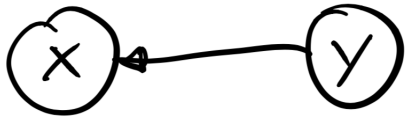


Figure 2.4: Joint probability distribution on two variables X and Y.

joint probability distribution over the pair of variables X and Y, and its PMF is given by the **chain rule**:

$$P(X, Y) = P(X|Y)P(Y)$$

Graphically, we have the graph fragment shown in Figure 2.4.

To sample from a joint distribution $P(X, Y)$ we can take the graph as a guide: we first sample a value y from $P(y)$, as it does not depend on any other variable, and then sample a value x from the conditional distribution $P(x|y)$.

Given a joint distribution $P(X, Y)$ we can ask what the probability is of an outcome x for X, irrespective of the value of Y. We call this the **marginal probability distribution** of X, and it can be calculated as

$$P(X) = \sum_y P(X, Y = y)$$

and of course we can also calculate the marginal distribution of Y, in the same way:

$$P(Y) = \sum_x P(X = x, Y)$$

We can also calculate the conditional distributions when given the joint distribution, by taking the joint probability distribution and dividing by the appropriate marginal:

$$P(X|Y) = P(X, Y)/P(Y)$$
$$P(Y|X) = P(X, Y)/P(X)$$

This is really just the chain rule, re-arranged.

Example

a_i	$T = Left$	$T = Right$	P(A)
Left	0.18	0.14	0.32
Right	0.06	0.42	0.48
Up	0.03	0.07	0.10
Down	0.03	0.07	0.10
P(T)	0.3	0.7	

Table 2.5: Joint probability distribution $P(A, T)$.

Table 2.5 calculates the joint density

$$P(A,T) = P(A|T)P(T)$$

where the CPT for $P(A|T)$ is taken from Table 2.4, and where take $P(T = Right) = 0.7$. The marginals $P(T)$ and $P(A)$ are also shown, respectively as row-wise and column-wise sums of the 4×2 table of joint probabiltites.

Exercises

1. Calculate the conditional distribution $P(T|A)$ given the joint distribution $P(A,T)$ in Table 2.5.
2. Viewing the integer coordinates i and j as separate random variables, use the distribution $P(i,j) = P(S)$ from Table 2.2 and calculate the marginals $P(i)$ and $P(j)$.
3. Similarly, what do the conditional probability tables for $P(i|j)$ and $P(j|i)$ look like? Before you start calculating, ponder the relationship with Table 2.2.

2.2.6 Bayes' Rule

Given the formulas we discussed above, we can now derive Bayes' rule, which allows us to infer knowledge about a variable, say the robot state S, given an observed sensor value o. The rule is named after the reverend Thomas Bayes, an eighteenth century minister who took an interest in probability later in life. He also lends his name to the Bayes nets which we discuss in Section 2.2.7.

Bayes' rule allows us to calculate the **posterior probability distribution** $P(S|O = o)$ on the variable S given an observed value for O. In Bayes' rule we use the term "prior" to indicate knowledge we have about a variable S before seeing evidence for it, and use "posterior" to denote the knowledge after having incorporated evidence. In our case, the evidence is the observed value o. To calculate the posterior, the elements we need are a prior probability distribution $P(S)$, a conditional probability distribution $P(O|S)$ modeling the measurement, and the value o itself. Given these elements, we can calculate the posterior probability distribution on S:

$$P(S|O = o) = \frac{P(O = o|S)P(S)}{P(O = o)} \tag{2.2}$$

The proof is simple and involves applying the chain rule in two ways: $P(O|S)P(S) = P(S,O) = P(S|O)P(O)$.

Because in the above the observation is *known*, statisticians introduce a different quantity called the **likelihood function**,

$$L(S;o) \stackrel{\Delta}{=} P(O = o|S)$$

which is a function of S, and reads as "the likelihood of the state S given the observed measurement o". Strictly speaking, any function proportional to $P(O = o|S)$ will do as the likelihood, but the above definition is simpler.

In addition, note that in Equation 2.2 the quantity $P(O = o)$ simply acts as a normalization constant. Given these two facts, we can state Bayes' rule in a more intuitive way - and easier to remember - as

$$P(S|O = o) \propto L(S; o)P(S) \tag{2.3}$$

or "the posterior is proportional to the likelihood weigthed by the prior."

Finally, when actually computing a posterior there is not even a need to think about anything but the joint distribution. Indeed, because by the chain rule we have $P(O = o|S)P(S) = P(S, O = o)$, and because $P(O = o)$ is just a normalization factor, we obtain a *third* form of Bayes' rule, which is the simplest of all:

$$P(S|O = o) \propto P(S, O = o) \tag{2.4}$$

Hence, if we are given a formula or table of joint probability entries $P(S, O)$, it suffices to just select all of them where $O = o$, normalize, and voila!

Exercises

1. Apply Bayes' rule to calculate the posterior probability $P(S|O = 5)$, using the prior $P(S)$ from Table 2.2 and the sensor model specified in Equation 2.1.

2. Suppose we are interested in estimating the probability of an obstacle in front of the robot, given an "obstacle sensor" that is accurate 90% of the time. Apply Bayes' rule to this example by creating the sensor model, prior, and deriving the posterior.

3. For the previous example, what happens if the sensor is random, i.e., it just outputs "obstacle detected" with 50% probability?

2.2.7 Bayes Nets

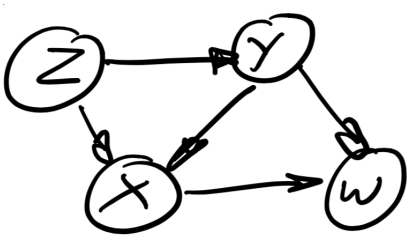


Figure 2.5: A Bayes net representing a more general joint distribution over the random variables W,X,Y, and Z.

A **Bayes net** is a directed acyclic graph (DAG) describing a factored probability distribution a set of random variables. The joint distribution on the set of all variables is given as

$$P(\{X_i\}) = \prod_{i=1}^{n} P(X_i|\Pi_i)$$

where n is the number of variables, and Π_i denotes the set of parents for variable X_i. An example of a Bayes net is shown in Figure 2.5, and it is simply a graphical representation of which random variables's CPT depend on which other variables. In this case, the joint distribution can be read off as

$$P(W, X, Y, Z) = P(W|X, Y)P(X|Y, Z)P(Y|Z)P(Z).$$

Note that the order in which we multiply the conditionals does not matter.

CPT	# entries
$P(Z)$	9
$P(Y\|Z)$	90
$P(X\|Y, Z)$	900
$P(W\|X, Y)$	900

Table 2.6: Number of entries in each CPT for the Bayes net of Figure 2.5, assuming all variables have 10 outcomes.

A Bayes net can be a very efficient representation of complex probability distributions, as they encode the dependence and especially independence relationships between the variables. For example, if we were to construct a full table of probabilities for each possible outcome of the variables W,X,Y, and Z, the table could be quite long. For example, if we assume they all have 10 possible values, then the full joint has 10^4 entries, i.e., $10,000$ unique values. You can save a tiny bit, because they have to sum up to 1, so strictly speaking we need only $9,999$ values. In contrast, we can tally how many entries all four CPT tables have for the Bayes net in Figure 2.5. In Table 2.6 we did just that; for example, $P(X|Y, Z)$ has 900 entries, i.e., 9 (independent) entries for each of 100 possible combinations of Y and Z. Hence, the total number of parameters we need is only $1,899$, which is way less than $9,999$.

2.2.8 Ancestral Sampling

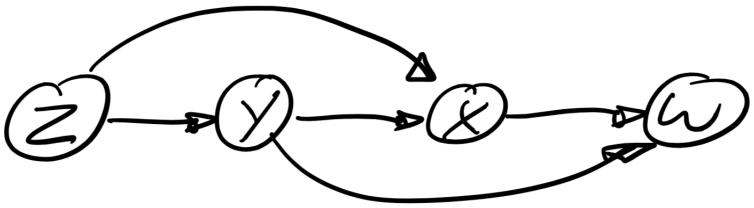


Figure 2.6: The topological sort of the Bayes net in Figure 2.5.

Sampling from the joint distribution given in Bayes net form can be done by sampling each variable in turn, but making sure that we always sample a node's parents first. This can be done through the notion of a **topological sort** of the DAG.

An easy algorithm to obtain a topological sort is Kahn's algorithm, which recursively removes a node from the graph that either has no parents, or whose parents have all been removed already. The order in which nodes were removed constitutes a (non-unique) topological sort order.

Sampling is then done by inverse transform sampling for each variable separately, in topological sort order. An example is shown in Figure 2.6, which shows a topological sort of the Bayes net in Figure 2.5. Hence, in this example we sample first Z, then Y, then X, and then W. Note that in this case the topological sort is unique, but that is an exception rather than the rule.

2.2.9 Dynamic Bayes Nets and Simulation

Note that directed cycles are not allowed in a Bayes net, i.e., the graph is acyclic. Hence, one might wonder how we deal with time: if a robot is all about the sense-think-act cycle, would we not expect a cycle in the graph when describing robots? The answer is to unroll time, as we discuss below.

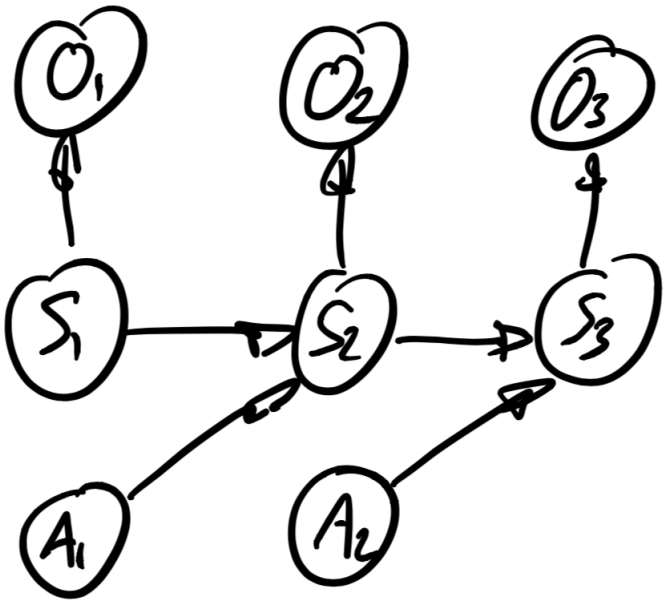


Figure 2.7: A Bayes net modeling our robot example.

When a Bayes net is used to represent the evolution of a system or agent over time, we call it a **dynamic Bayes net** or DBN. A small DBN fragment for a generic robot, modeled using discrete states, observations, and actions, is shown in Figure 2.7. We recognize subgraphs modeling the sensor behavior, and the effects of actions, at each time step. Simulation of the robot is then equivalent to sampling from this DBN, and in this case the topological sort is rather obvious, and hence so is the simulation algorithm:

1. First, sample the initial state s_1 from $P(S_1)$, a prior over the state. Set $k = 1$.

2. Next, simulate the sensor by sampling from the sensor model $P(O_k|S_k = s_k)$, yielding o_k.

3. Then, sample an action a_k from $P(A_k)$.

4. Lastly, simulate the effect of this action by sampling the next state s_k from

$$P(S_{k+1}|S_k = s_k, A_k = a_k).$$

5. Increase k and return to step 2.

Exercise

Simulate two different realizations from the dynamic Bayes net in Figure 2.7.

2.2.10 Inference in Bayes Nets

Inference is the process of obtaining knowledge about a subset of variables given the known values for another subset of variables. In this section we will talk about how to do inference when the joint distribution is specified using a Bayes net, but we will not take advantage of the sparse structure of the network. Hence, the algorithms below are completely general, for any (discrete) joint probability distribution, as long as you can evaluate the joint.

The simplest case occurs when we can *partition* the variables into two sets: the hidden variables $\mathcal{X}$ and the observed values $\mathcal{Z}$. Then we can simply apply Bayes' rule, but now applied to *sets* of variables, to obtain an expression for the posterior over the hidden variables $\mathcal{X}$. Using the easy version of Bayes' rule from equation 2.4 we obtain

$$P(\mathcal{X}|\mathcal{Z} = \mathfrak{z}) \propto P(\mathcal{X}, \mathcal{Z} = \mathfrak{z}),$$

where $\mathfrak{z}$ is the set of observed values for all variables in $\mathcal{Z}$.

x	y	$P(W = 2, X = x, Y = y, Z = 7)$
1	1	$P(W = 2\|X = 1, Y = 1)P(X = 1\|Y = 1, Z = 7)P(Y = 1\|Z = 7)P(Z = 7)$
1	2	$P(W = 2\|X = 1, Y = 2)P(X = 1\|Y = 2, Z = 7)P(Y = 2\|Z = 7)P(Z = 7)$
$\vdots$	$\vdots$	$\vdots$
10	9	$P(W = 2\|X = 10, Y = 9)P(X = 10\|Y = 9, Z = 7)P(Y = 9\|Z = 7)P(Z = 7)$
10	10	$P(W = 2\|X = 10, Y = 10)P(X = 10\|Y = 10, Z = 7)P(Y = 10\|Z = 7)P(Z = 7)$
		$\sum_{x,y} P(W = 2, X = x, Y = y, Z = 7)$

Table 2.7: Computation of the posterior $P(X, Y|W = 2, Z = 7)$ by enumeration. Note that all entries have to be normalized by the sum at the bottom to get the correct, normalized posterior.

There is an easy algorithm to calculate the posterior distribution above: simply enumerate all tuples $\mathcal{X}$ in a table, evaluate $P(\mathcal{X}, \mathcal{Z} = \mathfrak{z})$ for each one, and then normalize. As an example, let us consider the Bayes net from Figure 2.5, and take $\mathcal{X} = (X, Y)$ and $\mathcal{Z} = (W, Z)$.

As before, let us assume that each variable can take on 10 different outcomes, and that $\mathfrak{z} = (2, 7)$. The resulting table for $P(X, Y|W = 2, Z = 7) \propto P(W = 2, X, Y, Z = 7)$ is shown in Table 2.7.

A common inference problem associated with Bayes nets is the **most probable explanation** or MPE for $\mathcal{X}$: given the values $\mathfrak{z}$ for $\mathcal{Z}$, what is the most probable joint assignment to the other variables $\mathcal{X}$? While the posterior gives us the complete picture, the MPE is different in nature: it is a single assignment of values to $\mathcal{X}$. For example, given $\mathfrak{z} = (2, 7)$, the MPE for $\mathcal{X}$ could be $X = 3$ and $Y = 6$. Note that to compute the MPE, we need not bother with normalizing: we can simply find the maximum entry in the unnormalized posterior values.

In both these inference problems, the simple algorithm outlined above is *not* efficient. In the example the table is 100 entries long, and in general the number of entries is exponential in the size of $\mathcal{X}$. However, when inspecting the entries in Table 2.7 there are already some obvious ways to save: for example, $P(Z = 7)$ is a common factor in all entries, so clearly we should not even bother multiplying it in. In the next section, we will discuss methods to fully exploit the structure of the Bayes net to perform efficient inference.

If we had an efficient way to do inference, an MPE estimate would be a great way to estimate the trajectory of a robot over time. For example, using the dynamic Bayes net example from Figure 2.7, assume that we are given the value of all observations O and actions A. Then the MPE would simply be a trajectory of robot states through the grid. This is an example of robot localization over time, and is a key capability of a mobile robot. However, it will have to wait until we can do efficient inference.

Finally, another well known inference problem is the **maximum a posteriori** or MAP estimate: given the values of some variables $\mathcal{Z}$, what is the most probable joint assignment to a *subset* $\mathcal{X}$ of the other variables? In this case the variables are partitioned into three sets: the variables of interest $\mathcal{X}$, the nuisance variables $\mathcal{Y}$, and the observed variables $\mathcal{Z}$. We have

$$P(\mathcal{X}|\mathcal{Z} = \mathfrak{z}) = \sum_{\mathfrak{y}} P(\mathcal{X}, \mathcal{Y} = \mathfrak{y}|\mathcal{Z} = \mathfrak{z}) \propto \sum_{\mathfrak{y}} P(\mathcal{X}, \mathcal{Y} = \mathfrak{y}, \mathcal{Z} = \mathfrak{z}). \tag{2.5}$$

Finding a MAP estimate is more expensive than finding the MPE, as in addition to enumerating all possible combinations of $\mathcal{X}$ and $\mathcal{Y}$ values, we now need to calculate a possibly large number of sums, each exponential in the size of $\mathcal{Y}$. In addition, the *number* of sums is exponential in the size of $\mathcal{X}$. Below we will see that while we can still exploit the Bayes net structure, MAP estimates are fundamentally more expensive even in that case.

Exercises

1. Show that in the example from Figure 2.5, if we condition on known values for $\mathcal{Z} = (X, Y)$, the posterior $P(W, Z|X, Y)$ factors, and as a consequence we only have to enumerate two tables of length 10, instead of a large table of size 100.

2. Calculate the size of the table needed to enumerate the posterior over the states S the Bayes net from Figure 2.7, given the value of all observations O and actions A.

3. Show that if we are given the states, inferring the actions is actually quite efficient, even with the brute force enumeration. Hint: this is similar to the first exercise above.

4. Prove that we need only a single normalization constant in Equation 2.5. This is easiest using the standard form of Bayes' rule, Equation 2.2.

Summary

We briefly summarize what we learned in this section:

1. Discrete distributions model the outcome of a single categorical random variable.
2. Inverse transform sampling allows us to simulate a single variable.
3. Conditional probability distributions allow for dependence on other variables.
4. Models for sensing and acting can be built using parametric conditional distributions.
5. We can compute a joint probability distribution, and marginal and conditionals from it.
6. Bayes' rule allows us to infer knowledge about a state from a given observation.
7. Bayes nets allow us to encode more general joint probability distributions over many variables.
8. Ancestral sampling is a technique to simulate from any Bayes net.
9. Dynamic Bayes nets unroll time and can be used to simulate robots over time.
10. Inference in Bayes nets is a simple matter of enumeration, but this can be expensive.

2.3 Inference in Graphical Models

Motivation

Bayes nets are great for *modeling*, but for inference we need better data structures. Below we start the important special case of filtering, and introduce the Bayes filter. To reason about trajectories we then define hidden Markov models, and highlight their connection with robot localization. We then show how to efficiently perform inference by converting any Bayes net (with evidence) to a factor graph. We show both full posterior inference, MPE, and MAP estimation for HMMs. After that, we generalize to completely general inference in Bayes nets, introducing the elimination algorithm.

2.3.1 Bayes Filtering

We start off with the **Bayes filter**, which is a very simple recursive inference scheme. In this case we present the discrete version, which was used in robotics for localization under the name **Markov localization**. But the same general scheme underlies the venerable Kalman filter, and Monte Carlo Localization, which is based on a particle filter. The latter two are typically used in continuous settings, so we will revisit them later.

In the Bayes filter we are interested in estimating the state of the robot at the current time t, given all the measurements up to and including time t. Assuming that we have a probability distribution $P(S_{t-1})$ over the previous state S_{t-1}, we proceed in two phases:

1. In the **prediction phase** we predict a **predictive distribution** on the current state S_t given the distribution $P(s_{t-1})$ and a given action $A_{t-1} = a_{t-1}$. This is done by marginalizing out the previous state S_{t-1}, by summing over all possible values s_{t-1} for S_{t-1}:
$$P(S_t) = \sum_{s_{t-1}} P(S_t|s_{t-1}, a_{t-1})P(s_{t-1}).$$
2. In the **measurement phase** we upgrade this prior to a posterior via Bayes' rule, given an observation $O_t = o_t$:
$$\begin{aligned} P(S_t|O_t = o_t) &\propto P(o_t|S_t)P(S_t) \\ &\propto L(S_t|o_t)P(S_t). \end{aligned}$$

Above and where it is clear from context we abbreviate $P(S_t|S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1})$ as $P(S_t|s_{t-1}, a_{t-1})$, in the interest of a more succinct notation. But always remember that S_t is a random variable, and s_t is a *value* assigned to the random variable.

To be correct, we should really indicate exactly what information we are conditioning on when giving the formulas for the Bayes filter. To that end, we introduce the notation $\mathcal{O}^t = \{o_1, o_2, \dots o_t\}$, i.e., all measurements o up to and including time t. Similarly, we define the action $\mathcal{A}^t = \{a_1, a_2, \dots a_t\}$. The Bayes filter is then stated succinctly as

$$P(S_t|\mathcal{O}^{t-1}, \mathcal{A}^t) = \sum_{s_{t-1}} P(S_t|s_{t-1}, a_{t-1})P(s_{t-1}|\mathcal{O}^{t-1}, \mathcal{A}^{t-1}) \tag{2.6}$$

$$P(S_t|\mathcal{O}^t, \mathcal{A}^t) \propto L(S_t|o_t)P(S_t|\mathcal{O}^{t-1}, \mathcal{A}^t), \tag{2.7}$$

and it is seen to be perfectly recursive.

Figure 2.8 illustrates the measurement phase for a simple 1D example. In this environment there are two doors, and the robot has a door sensor. The predictive distribution $P(S)$ encodes that we believe the robot to be near the left door. The likelihood $L(S; o)$, where o indicates that we *did* perceive a door, models the fact that we are more likely to be near a door given $O = o$, but also allows for the fact that the door sensor could misfire at any location. Note that the likelihood is unnormalized and there is no need for it to sum up to 1. Finally, the posterior $P(S|o)$ is obtained, via Bayes' rule, as the product of the prediction $P(S)$ and the likelihood $L(S; o)$, and is shown at the bottom as a normalized probability distribution. As you can see, the most probable explanation for the robot state is $S = 5$, but there is a second mode at $S = 17$ due to the bimodal nature of the likelihood. However, that second mode is less probable because of our prior belief over S.

The Bayes filter above is a simple and computationally attractive scheme *if* we are only interested in the posterior distribution over the current state. However, if we are interested in the posterior over the entire state *trajectory*, we need to consider more general inference schemes. To this end, we introduce hidden Markov models in the next section.

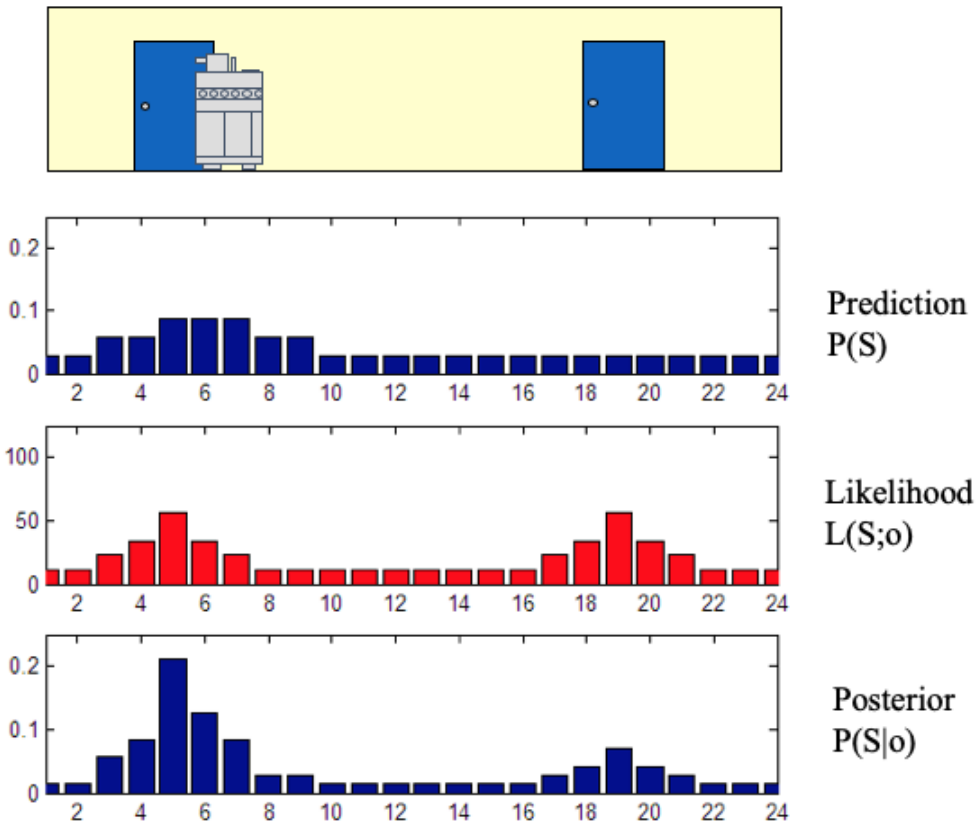


Figure 2.8: One-dimensional example of the measurement phase in Markov localization, the discrete version of the BAyes filter.

2.3.2 Hidden Markov Models

A **hidden Markov model** or HMM is a dynamic Bayes net that has two types of variables: states $\mathcal{X}$ and measurements $\mathcal{Z}$. The states $\mathcal{X}$ are connected sequentially and satisfy the what is called the **Markov property**: the probability of a state x_t is only dependent on the value of the previous state x_{t-1}. We call a sequence of random variables with this property a **Markov chain.** In addition, in an HMM we refer to the states $\mathcal{X}$ as *hidden* states, as typically we cannot directly observe their values. Instead, they are indirectly observed through the measurements $\mathcal{Z}$, where we have one measurement per hidden state. When these two properties are satisfied, we call this probabilistic model a hidden Markov model.

Figure 2.9 shows an example of an HMM for three time steps, i.e.., $\mathcal{X} = \{x_1, x_2, x_3\}$ and $\mathcal{Z} = \{Z_1, Z_2, Z_3\}$. As we discussed, in a Bayes net each node is associated with a conditional distribution: the Markov chain has the prior $P(x_1)$ and transition probabilities $P(x_2|x_1)$ and $P(x_3|x_2)$, whereas the measurements Z_t depend only on the state x_t, modeled by measurement models $P(Z_t|x_t)$. In other words, the Bayes net encodes the following joint distribution $P(\mathcal{X}, \mathcal{Z})$:

$$P(\mathcal{X}, \mathcal{Z}) = P(x_1)P(Z_1|x_1)P(x_2|x_1)P(Z_2|x_2)P(x_3|x_2)P(Z_3|x_3)$$

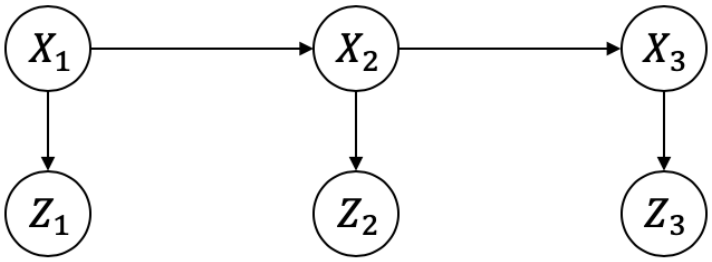


Figure 2.9: An HMM, unrolled over three time-steps, represented by a Bayes net.

Note that we can also write this more succinctly as

$$P(\mathcal{X}, \mathcal{Z}) = P(\mathcal{Z}|\mathcal{X})P(\mathcal{X}) \tag{2.8}$$

where

$$P(\mathcal{X}) = P(x_1, x_2, x_3) = P(x_1)P(x_2|x_1)P(x_3|x_2) \tag{2.9}$$

is the prior over state *trajectories*.

2.3.3 Naive Inference in HMMs

In inference, we might want to infer the maximum probable explanation (MPE) for the states $\mathcal{X}$ given values $\mathcal{z} = \{z_1, z_2, z_3\}$ for $\mathcal{Z}$. As we saw before, one way to perform inference is to apply Bayes' rule to Equation 2.8, and get an expression for the *posterior* probability distribution over the state trajectory $\mathcal{X}$, given the measurements $\mathcal{Z} = \mathcal{z}$:

$$\begin{aligned} P(\mathcal{X}|\mathcal{Z}) &\propto P(\mathcal{Z} = \mathcal{z}|\mathcal{X})P(\mathcal{X}) \\ &= L(\mathcal{X}; \mathcal{Z} = \mathcal{z})P(\mathcal{X}) \end{aligned} \tag{2.10}$$

where $P(\mathcal{X})$ is given above in Equation 2.9, and the **likelihood** $(\mathcal{X}; \mathcal{Z} = \mathcal{z})$ of $\mathcal{X}$ given $\mathcal{Z} = \mathcal{z}$ is defined as before, yielding the following function of $\mathcal{X}$:

$$\begin{aligned} L(\mathcal{X}; \mathcal{Z} = \mathcal{z}) &\triangleq P(\mathcal{Z} = \mathcal{z}|\mathcal{X}) \\ &= P(z_1|x_1)P(z_2|x_2)P(z_3|x_3) \\ &= L(x_1; Z_1)L(x_2; Z_2)L(x_3; Z_3) \end{aligned}$$

As we saw, a naive implementation for finding the most probable explanation (MPE) for $\mathcal{X}$ would tabulate all possible trajectories $\mathcal{X}$ and calculate the posterior (2.10) for each one. Unfortunately the number of entries in this giant table is exponential in the number of states. Not only is this computationally prohibitive for long trajectories, but intuitively it is clear that for many of these trajectories we are computing the same values over and over again. In fact, there are three different approaches to improve on this:

1. Branch & bound
2. Dynamic programming

3. Inference using factor graphs

Branch and bound is a powerful technique but will not generalize to continuous variables, like the other two approaches will. And, we will see that dynamic programming, which underlies the classical inference algorithms in the HMM literature, is just a special case of the last approach. Hence, here we will dive in and immediately go for the most general approach: inference in factor graphs.

2.3.4 Factor Graphs

We first introduce the notion of factors. Again referring to the example from Figure 2.9, let us consider the posterior (2.10). Since the measurements $\mathcal{Z}$ are *known*, the posterior is proportional to the product of six **factors**, three of which derive from the the Markov chain, and three are likelihood factors as defined before:

$$P(\mathcal{X}|\mathcal{Z}) \propto P(x_1)L(x_1; z_1)P(x_2|x_1)L(x_2; z_2)P(x_3|x_2)L(x_3; z_3) \tag{2.11}$$

Some of these factors are unary factors, and some are binary factors. In particular, in (2.11) some of the factors depend on just one hidden variable, for example $L(x_2; z_2)$, whereas others depend on two variables, e.g., the transition model $P(x_3|x_2)$. Measurements are not counted here, because once we are *given* the measurements $\mathcal{Z}$, they merely function as known parameters in the likelihoods $L(x_t; z_t)$, which are seen as functions of *just* the state x_t.

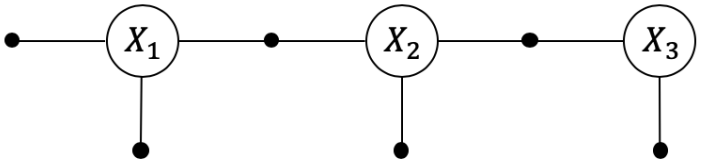


Figure 2.10: An HMM with observed measurements, unrolled over time, represented as a factor graph.

This motivates a different graphical model, a **factor graph**, in which we only represent the *hidden* variables x_1, x_2, and x_3, connected to factors that encode probabilistic information on them. For our example with three hidden states, the corresponding factor graph is shown in Figure 2.10. It should be clear from the figure that the connectivity of a factor graph encodes, for each factor ϕ_i, which subset of variables $\mathcal{X}_i$ it depends on. We write:

$$\phi(\mathcal{X}) = \phi_1(x_1)\phi_2(x_1)\phi_3(x_1, x_2)\phi_4(x_2)\phi_5(x_2, x_3)\phi_6(x_3) \tag{2.12}$$

where the factors in (2.12) are defined to correspond one-to-one to Equation 2.11. For example,

$$\phi_6(x_3) \triangleq L(x_3; z_3).$$

All measurements are associated with unary factors, whereas the Markov chain is associated mostly with binary factors, with the exception of the unary factor $\phi_1(x_1)$. Note that in

defining the factors we can omit any normalization factors, which in many cases results in computational savings.

Formally a factor graph is a bipartite graph $F = (\mathcal{U}, \mathcal{V}, \mathcal{E})$ with two types of nodes: **factors** $\phi_i \in \mathcal{U}$ and **variables** $x_j \in \mathcal{V}$. Edges $e_{ij} \in \mathcal{E}$ are always between factor nodes and variables nodes. The set of random variable nodes adjacent to a factor ϕ_i is written as $\mathcal{X}_i$. With these definitions, a factor graph F defines the factorization of a global function $\phi(\mathcal{X})$ as

$$\phi(\mathcal{X}) = \prod_i \phi_i(\mathcal{X}_i). \tag{2.13}$$

In other words, the independence relationships are encoded by the edges e_{ij} of the factor graph, with each factor ϕ_i a function of *only* the variables $\mathcal{X}_i$ in its adjacency set. As example, for the factor graph in Figure 2.10 we have:

$$\begin{aligned}
\mathcal{X}_1 &= \{X_1\} \\
\mathcal{X}_2 &= \{X_1\} \\
\mathcal{X}_3 &= \{X_1, X_2\} \\
\mathcal{X}_4 &= \{X_2\} \\
\mathcal{X}_5 &= \{X_2, X_3\} \\
\mathcal{X}_6 &= \{X_3\}
\end{aligned}$$

2.3.5 Converting Bayes Nets into Factor Graphs.

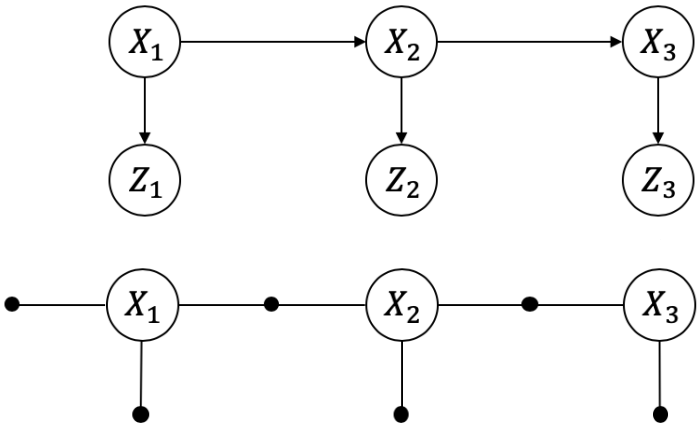


Figure 2.11: Converting a Bayes net into a factor graph, in the case that the variables $\mathcal{Z}$ are known.

Every Bayes net can be trivially converted to a factor graph. Recall that every node in a Bayes net denotes a conditional density on the corresponding variable and its parent nodes. Hence, the conversion is quite simple: every Bayes net node splits in *both* a variable node and a factor node in the corresponding factor graph. The factor is connected to the variable node, as well as the variable nodes corresponding to the parent nodes in the Bayes net. If some nodes in the Bayes net are evidence nodes, i.e., they are given as known variables, we

omit the corresponding variable nodes: the known variable simply becomes a fixed parameter in the corresponding factor.

Once we convert a Bayes net with evidence into a factor graph where the evidence is all implicit in the factors, we can support a number of different computations. First, given any factor graph defining an unnormalized density $\phi(X)$, we can easily **evaluate** it for any given value, by simply evaluating every factor and multiplying the results. The factor graph represents the unnormalized posterior, i.e., $\phi(\mathcal{X}) \propto P(\mathcal{X}|\mathcal{Z})$. Evaluation opens up the way to **optimization**, e.g., finding the most probable explanation or MPE, as we will do below. In the case of discrete variables, graph search methods can be applied, but we will use a different approach.

While local or global maxima of the posterior are often of most interest, **sampling** from a probability density can be used to visualize, explore, and compute statistics and expected values associated with the posterior. However, the ancestral sampling method we discussed earlier only applies to directed acyclic graphs. There are however more general sampling algorithms that can be used for factor graphs, more specifically Markov chain Monte Carlo (MCMC) methods. One such method is Gibbs sampling, which proceeds by sampling one variable at a time from its conditional density given all other variables it is connected to via factors. This assumes that this conditional density can be easily obtained, which is in fact true for discrete variables.

Below we use factor graphs as the organizing principle for probabilistic inference. In later chapters we will expand their use to continuous variables, and will see that factor graphs aptly describe the independence assumptions and sparse nature of the large nonlinear least-squares problems arising in robotics. But their usefulness extends far beyond that: they are at the core of the sparse linear solvers we use as building blocks, they clearly show the nature of filtering and incremental inference, and lead naturally to distributed and/or parallel versions of robotics.

Exercise

1. Convert the dynamic Bayes net from the previous section into a factor graph, assuming no known variables.

1. Finally, do the same again, but now assume the states are given. Reflect on the remarkable phenomenon that happens.

2.3.6 The Max-Product Algorithm for HMMs

Given a factor graph, the max-product algorithm is an $O(n)$ algorithm to find the maximum probable explanation or MPE.

We will use the example from Figure 2.10 to give the intuition. To find the MPE for $\mathcal{X}$ we need to *maximize* the product

$$\phi(x_1, x_2, x_3) = \prod \phi_i(\mathcal{X}_i) \tag{2.14}$$

i.e., the **value** of the factor graph. The **max-product algorithm** proceeds one variable at a time, and in an HMM we will proceed from left to right, i.e. we start with state x_1 and proceed until we processed all states.

Eliminating x_1

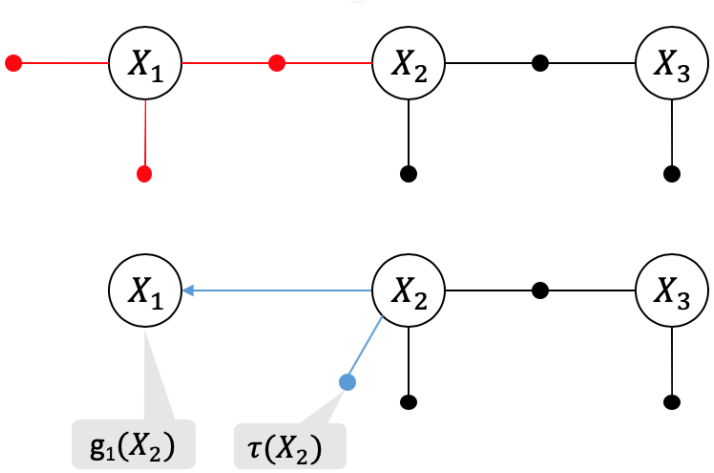


Figure 2.12: Max-product: eliminating x_1.

We start by considering the first state x_1, and we form a **product factor** $\psi(x_1, x_2)$ that collects *only* the factors connected to x_1:

$$\psi(x_1, x_2) = \phi_1(x_1)\phi_2(x_1)\phi_3(x_1, x_2).$$

When we use a factor in a product, we *remove* it from the original factor graph. Note that because one of those factors, the state transition model $\phi_3(x_1, x_2) \triangleq P(x_2|x_1)$, is also connected to the second state x_2, the product factor is a function of *both* x_1 and x_2, i.e., it is a binary factor.

The key observation in the max-product algorithm is that we can now *eliminate* x_1 from the problem, by looking at all possible values x_2 of x_2, and creating a lookup table g_1 for the best possible value of x_1:

$$g_1(x_2) = \arg\max_{x_1} \psi(x_1, x_2).$$

The size of this lookup table is equal to the number of possible outcomes for x_2: in our grid-world example this is 100, as we are using a 10×10 grid. We can of course store this lookup table as a grid, as well. It might help your understanding to think about what this table will look like.

We also record the value of the product factor for that maximum, so we can use it down the line for taking into account the consequence of each choice:

$$\tau(x_2) = \max_{x_1} \psi(x_1, x_2).$$

In practice, of course, both steps can be implemented in a single function. We then put this new factor $\tau(x_2)$ back into the graph, essentially summarizing the result of eliminating x_1 from the problem entirely, obtaining the **reduced graph**

$$\Phi_{2:3} = \tau(x_2)\phi_4(x_2)\phi_5(x_2, x_3)\phi_6(x_3). \tag{2.15}$$

Let us reflect on what happened above, because it is significant: we eliminated x_1 from consideration, and obtained a reduced problem that only depends on the remaining states x_2 and x_3. You can intuitively see that this algorithm will terminate after n steps, and in fact you could prove it by induction. In addition, the lookup table g_1 gives us a way that, once we know what the optimal value for x_2 is, we can just read off the optimal value for x_1. This is what we will do, in *reverse* elimination order, after the algorithm terminates.

Eliminating x_2

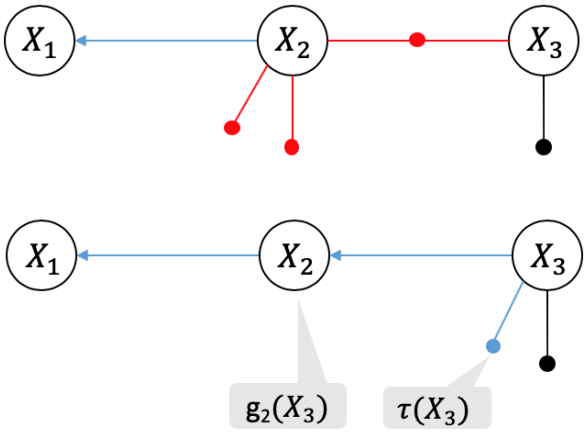


Figure 2.13: Max-product: eliminating x_2.

We now perform exactly the same steps for the state x_2. In this case, the product factor $\psi(x_2, x_3)$ has only factors connected to x_2,

$$\psi(x_2, x_3) = \tau(x_2)\phi_4(x_2)\phi_5(x_2, x_3),$$

but now includes the factor $\tau(x_2)$ from the previous step. Note that since we started from the reduced graph (2.15), the product factor is guaranteed to not depend on the first state x_1: that was eliminated! In fact, we can now in turn eliminate x_2 from the problem, by looking at all possible values x_3 of x_3, and creating a lookup table g_2 for the best possible value of x_2, given x_3,

$$g_2(x_3) = \arg\max_{x_2} \psi(x_2, x_3),$$

and as above we also record the value of the product factor for that maximum in a new factor $\tau(x_3)$:

$$\tau(x_3) = \max_{x_2} \psi(x_2, x_3).$$

We then put this new factor $\tau(x_3)$ back into the graph, which is now reduced even more:

$$\Phi_{3:3} = \tau(x_3)\phi_6(x_3)$$

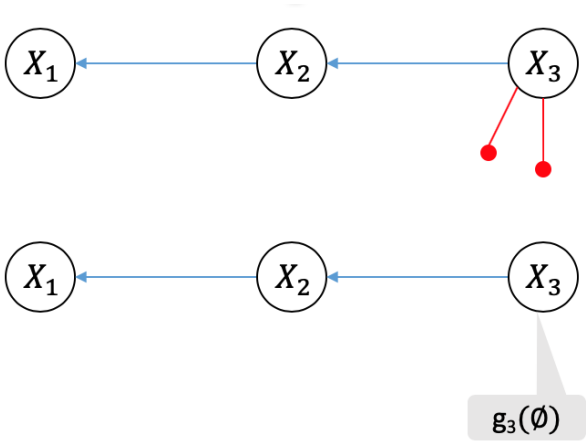


Figure 2.14: Max-product: eliminating x_3.

Eliminating x_3

Finally, we eliminate x_3, where the product factor is now the entire remaining graph and only depends on x_3, as all other states have already been eliminated:

$$\psi(x_3) = \tau(x_3)\phi_6(x_3).$$

We again obtain a lookup table,

$$g_3(\emptyset) = \arg\max_{x_3} \psi(x_3),$$

and a new factor:

$$\tau(\emptyset) = \max_{x_1} \psi(x_3).$$

Note however that now the value does not depend on any arguments! This is indicated by making the argument list equal to the empty set $\emptyset$. Indeed, g_3 just tells us what the best value for x_3 is, and τ tells us the corresponding value. Because it incorporates the factors from the previous elimination steps, this will in fact be exactly the solution to Problem 2.14.

Back-substitution

Once we know the value for x_3, we can simply plug it into the lookup table $g_2(x_3)$ to get the value for x_2, which we can then plug into the lookup table g_1 to get the value for x_1, and we recover the MPE in one single backward pass.

The Entire Algorithm

The HMM max-product algorithm for any value of n is given in Algorithm 1, where we used the shorthand notation $\Phi_{j:n} \triangleq \phi(x_j, \ldots, x_n)$ to denote a reduced factor graph. The algorithm proceeds by eliminating one hidden state x_j at a time, starting with the complete HMM factor graph $\Phi_{1:n}$. As we eliminate each variable x_j, the function EliminateOne

Algorithm 1 The Max-Product Algorithm for HMMs

function MAXPRODUCTHMM($\Phi_{1:n}$) ▷ given an HMM with n states
  for $j = 1...n$ **do** ▷ for all states
    $g_j(x_{j+1}), \Phi_{j+1:n} \leftarrow \text{CreateLookupTable}(\Phi_{j:n}, x_j)$ ▷ eliminate x_j
  return $\{g_1(x_1)g(x_2)\ldots g_n(\emptyset)\}$ ▷ return DAG of lookup tables

Algorithm 2 Create lookup table g_j by eliminating state x_j from a factor graph $\Phi_{j:n}$.

function CREATELOOKUPTABLE($\Phi_{j:n}, x_j$) ▷ given reduced graph $\Phi_{j:n}$
  Remove all factors $\phi_i(\mathcal{X}_i)$ that are contain x_j
  $\psi(x_j, x_{j+1}) \leftarrow \prod_i \phi_i(\mathcal{X}_i)$ ▷ create the product factor ψ
  $g_j(x_{j+1}), \tau(x_{j+1}) \leftarrow \psi(x_j, x_{j+1})$ ▷ **perform argmax, max**
  Add the new factor $\tau(x_{j+1})$ back into the graph
  return $g_j(x_{j+1}), \Phi_{j+1:n}$ ▷ lookup table and reduced graph

produces a single lookup table $g_j(x_{j+1})$, as well as a reduced factor graph $\Phi_{j+1:n}$ on the remaining variables. After all variables have been eliminated, the algorithm returns a chain of lookup tables that can be used to recover the MPE in reverse elimination order.

2.3.7 The Sum-Product Algorithm for HMMs

The sum-product algorithm for HMMs is a slight tweak on the max-product algorithm that instead produces a Bayes net that calculates the posterior probability $P(\mathcal{X}|\mathcal{Z})$. Whereas the max-product produces a DAG of lookup tables, the sum-product produces a DAG of conditionals, i.e., a Bayes net. This is particularly interesting if one is content with a maximum probable explanation or MPE, but instead wants the **full Bayesian probability distribution** of which assignments to the states are more probable than others. The fact that we recover this distribution in the form of a Bayes net again is satisfying, because as we saw that is an economical representation of a probability distribution.

One might wonder about the wisdom of all this: we started with a Bayes net, converted to a factor graph, and now end up with a Bayes net again? Indeed, but there are two important differences: the first Bayes net represents the joint distribution $P(\mathcal{X}, \mathcal{Z})$ and is very useful for modeling. However, the second Bayes represents the posterior $P(\mathcal{X}|\mathcal{Z})$, and only has nodes for the random variables in $\mathcal{X}$, hence it is much smaller. Finally, in many practical cases we do not even bother with the modeling step, but construct the factor graph directly from the measurements.

Algorithm 3 The Sum-Product Algorithm for HMMs

function SUMPRODUCTHMM($\Phi_{1:n}$) ▷ given an HMM with n states
  for $j = 1...n$ **do** ▷ for all states
    $p(x_j|x_{j+1}), \Phi_{j+1:n} \leftarrow \text{ApplyChainRule}(\Phi_{j:n}, x_j)$ ▷ eliminate x_j
  return $p(x_1|x_2)p(x_2|x_3)\ldots p(x_n)$ ▷ return Bayes net

Algorithm 4 Eliminate variable x_j from a factor graph $\Phi_{j:n}$.

function ApplyChainRule($\Phi_{j:n}, x_j$) ▷ given reduced graph $\Phi_{j:n}$
  Remove all factors $\phi_i(\mathcal{X}_i)$ that contain x_j
  $\psi(x_j, x_{j+1}) \leftarrow \prod_i \phi_i(\mathcal{X}_i)$ ▷ create the product factor ψ
  $p(x_j|x_{j+1})\tau(x_{j+1}) \leftarrow \psi(x_j, x_{j+1})$ ▷ **factorize the product** ψ
  Add the new factor $\tau(x_{j+1})$ back into the graph
  return $p(x_j|x_{j+1}), \Phi_{j+1:n}$ ▷ conditional and reduced graph

The only tweak necessary is to replace the maximization and arg max in the elimination step with the chain rule. Indeed, we factor each product factor $\psi(x_j, x_{j+1})$ into a conditional $P(x_j|x_{j+1})$ and an (unnormalized) marginal $\tau(x_{j+1})$:

$$P(x_j|x_{j+1})\tau(x_{j+1}) \leftarrow \psi(x_j, x_{j+1}) \tag{2.16}$$

The algorithm is called the **sum-product algorithm** because the marginal is obtained by summing over all values of the state x_j:

$$\tau(x_{j+1}) = \sum_{x_j} \psi(x_j, x_{j+1}) \tag{2.17}$$

We do not bother normalizing this into a proper distribution, as these marginals are just intermediate steps in the algorithm. However, when computing the conditional, we do normalize, and is it so happens the normalization constant is simply equal to $1/\tau(x_{j+1})$:

$$P(x_j|x_{j+1}) = \frac{\psi(x_j, x_{j+1})}{\tau(x_{j+1})} \tag{2.18}$$

In summary, the chain rule is implemented by 2.17 and 2.18. The entire algorithm is is listed as Algorithm 3.

Note that after we recover the Bayes net the algorithm terminates: there is no back-substitution step. However, one might consider ancestral sampling as a type of back-substitution: the reverse elimination order is always a topological sort of the resulting Bayes net! Hence, after the sum-product algorithm, we can sample as many realizations from the posterior as we want: rather than just one MPE, we now have thousands of plausible explanations, and ancestral sampling will yield them in exactly the correct frequencies.

Sidebar

When we can produce samples $\mathcal{X}^{(s)}$ from a posterior $P(\mathcal{X}|\mathcal{Z})$, we can calculate empirical means of any real-valued function $f(\mathcal{X})$ as follows:

$$E_{P(\mathcal{X}|\mathcal{Z})}[f(x)] \approx \sum f(\mathcal{X}^{(s)})$$

For example, we can calculate the posterior mean of how far the robot traveled, either in Euclidean or Manhattan distance. These estimators will have less variability than just calculating the distance for the MPE, as they average over the entire probability distribution.

2.3.8 The Variable Elimination Algorithm

There exists a general algorithm that, given *any* factor graph, can compute the corresponding posterior distribution $p(\mathcal{X}|\mathcal{Z})$ on the unknown variables $\mathcal{X}$. Above we saw that a factor graph represents the unnormalized posterior $\phi(\mathcal{X}) \propto P(\mathcal{X}|\mathcal{Z})$ as a product of factors, typically generated directly from the measurements. The variable elimination algorithm is a general recipe for converting any factor graph back to a Bayes net, but now *only* on the unknown variables $\mathcal{X}$.

In particular, the **variable elimination** algorithm is a way to factorize any factor graph of the form

$$\phi(\mathcal{X}) = \phi(x_1, \ldots, x_n) \tag{2.19}$$

into a factored Bayes net probability density of the form

$$p(\mathcal{X}) = p(x_1|\mathcal{S}_1)p(x_2|\mathcal{S}_2)\ldots p(x_n) = \prod_j p(x_j|\mathcal{S}_j), \tag{2.20}$$

where the **separator** $\mathcal{S}(x_j)$ is defined as the set of variables on which x_j is conditioned, after elimination. While this factorization is akin to the chain rule, eliminating a sparse factor graph will typically lead to small separators, although this depends on the chosen variable ordering $x_1, \ldots, x_n$.

Algorithm 5 The Variable Elimination Algorithm

function ELIMINATE($\Phi_{1:n}$) ▷ given a factor graph on n variables
  for $j = 1...n$ **do** ▷ for all variables
    $p(x_j|\mathcal{S}_j), \Phi_{j+1:n} \leftarrow$ EliminateOne($\Phi_{j:n}, x_j$) ▷ eliminate x_j
  return $p(x_1|\mathcal{S}_1)p(x_2|\mathcal{S}_2)\ldots p(x_n)$ ▷ return Bayes net

Algorithm 6 Eliminate variable x_j from a factor graph $\Phi_{j:n}$.

function ELIMINATEONE($\Phi_{j:n}, x_j$) ▷ given reduced graph $\Phi_{j:n}$
  Remove all factors $\phi_i(\mathcal{X}_i)$ that are adjacent to x_j
  $\mathcal{S}(x_j) \leftarrow$ all variables involved excluding x_j ▷ the separator
  $\psi(x_j, \mathcal{S}_j) \leftarrow \prod_i \phi_i(\mathcal{X}_i)$ ▷ create the product factor ψ
  $p(x_j|\mathcal{S}_j)\tau(\mathcal{S}_j) \leftarrow \psi(x_j, \mathcal{S}_j)$ ▷ factorize the product ψ
  Add the new factor $\tau(\mathcal{S}_j)$ back into the graph
  return $p(x_j|\mathcal{S}_j), \Phi_{j+1:n}$ ▷ Conditional and reduced graph

The variable elimination algorithm is listed as Algorithm 5, where we again used the shorthand notation $\Phi_{j:n} \triangleq \phi(x_j, \ldots, x_n)$ to denote a partially eliminated factor graph. The algorithm proceeds by eliminating one variable x_j at a time, starting with the complete factor graph $\Phi_{1:n}$. As we eliminate each variable x_j, the function ELIMINATEONE produces a single conditional $p(x_j|\mathcal{S}_j)$, as well as a reduced factor graph $\Phi_{j+1:n}$ on the remaining variables. After all variables have been eliminated, the algorithm returns the resulting Bayes net with the desired factorization.

Above we gave the sum-product version of the variable elimination algorithm. The corresponding max-product version produces a DAG of lookup tables instead, supporting the computation of the MPE. In both cases, the complexity is similar but depends on the chosen variable ordering $x_1, \ldots, x_n$, as we discuss next.

2.3.9 Complexity

The elimination algorithm has exponential complexity in the size of the largest separator. The chosen variable ordering $x_1, \ldots, x_n$ can affect the complexity dramatically. Some orderings lead to smaller separators, and unfortunately it is hard to find an optimal ordering - although it can be done for small graphs. A good heuristic is to greedily eliminate the variables with the smallest separator first. Another is to recursively split the graph, and eliminate starting from the leaves of the binary tree that is formed by the splitting process, but we will not discuss that here.

In the special case of HMMs, and in fact any singly connected graph, the complexity is linear in the number of nodes. The reason is easiest to see for an HMM, as after conversion to a factor graph the graph is just a chain. You can easily prove, by induction, that the size of the separator is always one. It must be, by the way, as the resulting DAG is also singly connected (there is at most one path from any node to any other node). Algorithms 1 and 3 are the max-product and sum-product variants that we obtain when eliminating from left to right. However, it is possible to choose a different ordering, even for HMMs. For example, if we eliminate from right to left, we get an equally efficient algorithm, although all the intermediate product and marginal factors will be different.

Exercises

1. Perform symbolic elimination, i.e., just the graph part without computation, on some factor graphs of interest.

2. Come up with an ordering for an HMM which is neither left-right or right-left which nevertheless results in a singly-connected Bayes net.

3. Come up with an ordering to eliminate an HMM which does not lead to a singly-connected Bayes net.

2.3.10 MAP Estimation

Maximum a posteriori estimation is at least as expensive as MPE or calculating the full posterior. Remember that MAP estimation is only interested in a subset of the variables, and we partition the variables into three sets: the variables of interest $\mathcal{X}$, the nuisance variables $\mathcal{Y}$, and the observed variables $\mathcal{Z}$. The elimination algorithm is easy to modify to do MAP estimation, simply *by making sure that the variables of interest $\mathcal{X}$ are eliminated last.*

Why does this work? We can easily see this if we take a "30,000 feet view" of the elimination algorithm. In MAP estimation, we are interested in maximizing $P(\mathcal{X}|\mathcal{Z} = \mathcal{z})$, but the Bayes net gives us the joint distribution $P(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$. The first step is to instantiate

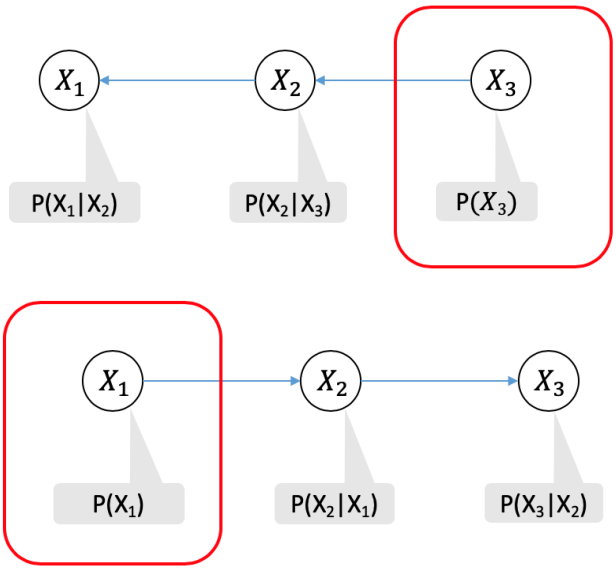


Figure 2.15: MAP estimation for x_3 and x_1, respectively: we simply change the elimination order to make sure the variable(s) of interest are eliminated last.

the evidence $\mathcal{z}$and convert to a factor graph $f(\mathcal{X}, \mathcal{Y}; \mathcal{Z} = \mathcal{z})$. When we eliminate using the sum-product algorithm, using the elimination order $\mathcal{Y}$,$\mathcal{X}$, we obtain a DAG encoding the resulting posterior as

$$P(\mathcal{Y}|\mathcal{X}, \mathcal{Z} = \mathcal{z})P(\mathcal{X}|\mathcal{Z} = \mathcal{z})$$

where $P(\mathcal{X}|\mathcal{Z} = \mathcal{z})$ is the desired marginal distribution on $\mathcal{X}$. Using max-product, the resulting DAG for the MPE is

$$\pi(\mathcal{Y}|\mathcal{X}, \mathcal{Z} = \mathcal{z})\pi(\mathcal{X}|\mathcal{Z} = \mathcal{z})$$

where the lookup table $\pi(\mathcal{X}|\mathcal{Z} = \mathcal{z})$ corresponds to the MAP estimate.

The higher complexity of MAP estimation derives from the fact that not all elimination orderings are allowed anymore. In particular, the optimal ordering, or even approximately optimal orderings, may all be incompatible with eliminating $\mathcal{X}$ last.

Exercises

1. Do MAP estimation for some factor graphs of interest.
2. Construct a small example where MAP estimation is more expensive than MPE, even when using optimal orderings for both.
3. Think about the complexity of MAP estimation in an HMM. When is it not more expensive than MPE?

Summary

We briefly summarize what we learned in this section:

1. Bayes filtering is a recursive state estimation scheme.
2. Hidden Markov models can be used to reason about a sequence of states observed indirectly via sensors.
3. Naive inference in HMMs can be quite expensive.
4. Factor graphs are a new graphical language that make measurements implicit.
5. Any Bayes net (with or without evidence) can be converted to a factor graph.
6. The max-product algorithm for HMMs is an efficient computation for the MPE.
7. The sum-product algorithm returns the full Bayesian posterior as a Bayes net.
8. The variable elimination algorithm is a generalization that works for *any* factor graph.
9. The complexity of variable elimination depends on the elimination order.
10. MAP estimation is always as least as expensive, as it constrains the elimination order.

2.4 Acting Optimally

Motivation

Show how FG also do action.

2.4.1 Finite Horizon Planning

We start with the agent model, but now no observations. Do min-sum, attach costs to actions, reward (negative cost) to goal. Show finite horizon.

2.4.2 Optimal Policy

At one point, time-dependent policy becomes fixed. Can be expensive if state space is large.

2.4.3 Planning through Search

Dijkstra's algorithm. Bidirectional.

2.4.4 Branch and Bound: A*

BB limits computation

2.4.5 A* for MPE Inference

Bring it back: we can also do search to find MPE

Summary

We briefly summarize what we learned in this section:

1. Factor graphs for finite horizon planning
2. Infinite horizon leads to a fixed optimal policy.
3. Search is an alternative to inference and leads to a plan.
4. Branch & bound helps search faster.
5. A* can be used for probabilistic inference, as well.

Chapter 3

A Differential Drive Mobile Robot

3.1 Planar Geometry

We start off by defining and giving some intuitions for geometry in the plane.

3.1.1 Planar Rotations aka SO(2)

I spend a lot of time on 2D rotations below, but bear with it! It is the simplest space to think in, and we can get the most essential concepts across, which will then generalize elegantly to the other transformation groups.

Basic Facts

A point P^b in a rotated coordinate frame B can be expressed in a reference frame S by multiplying with a 2×2 orthonormal **rotation matrix** R_b^s,

$$p^s = R_b^s p^b$$

where the indices on R_b^s indicate the source and destination frames. The set of 2×2 rotation matrices, with matrix multiplication as the composition operator, form a *commutative* group called the **Special Orthogonal group** $SO(2)$. This means it satisfies the following properties, for all $R, R_1, R_2, R_3 \in SO(2)$:

1. Closed: $R_1 R_2 \in SO(2)$.
2. Identity: $I_2 R = R = R I_2$ with I_2 the 2×2 identity matrix.
3. Inverse: $R R^{-1} = I_2 = R^{-1} R$. Note that $R^{-1} = R^T$.
4. Associativity: $(R_1 R_2)\, R_3 = R_1\, (R_2 R_3)$.
5. Commutativity: $R_1 R_2 = R_2 R_1$.

In fact, the 2D rotation matrices form a 1-dimensional subgroup of the general linear group of 2×2 invertible matrices $GL(2)$, i.e., $SO(2) \subset GL(2)$. Informally, $SO(2)$ is also a **manifold**,

because it is a smooth continuous subset of the 2×2 matrices. We say it is a 1-D manifold in the 4-dimensional ambient space of 2×2 matrices. It is clear why this is so: the matrix is constrained to be orthonormal, which provides three non-redundant constraints on its four entries.

However, this is not quite enough to fully define the manifold, as rotations composed with a reflection also satisfy those constraints: together, they form the orthogonal group O(2). The "S" in SO(2) picks one of the two connected components of the O(2) manifold, namely those orthonormal matrices that have unit determinant, i.e., $|R| = 1$.

Finally, since $SO(2)$ is *both* a group and a manifold, we call it - informally, again - a 1-dimensional **Lie group**. An essential property of Lie groups is that the group operation is smooth, i.e., a small change to either R_1 or R_2 yields a small change in their product $R_1 R_2$. Similarly, the group action of $SO(2)$ on a 2D point $p \in \mathbb{R}^2$ is also smooth: a small change in R will yield a small change in Rp.

3.1.2 2D Rigid Transforms aka SE(2)

Basic Facts

In robot manipulators, the pose of the end-effector or tool is computed by concatenating several rigid transforms. Let p^b be the 2D coordinates of a point in the frame B, and p^s the coordinates of the same point in a base frame S. We can transform p^b to p^s by a **2D rigid transform** T_b^s, which is a rotation followed by a 2D translation,

$$p^s = T_b^s \otimes p^b \triangleq R_b^s p^b + t_b^s$$

with $T_b^s \triangleq (R_b^s, t_b^s)$, where $R_b^s \in SO(2)$ and $t_b^s \in \mathbb{R}^2$. In all of the above, a subscript B indicates the frame we are transforming *from*, and the superscript S indicates the frame we are transforming *to*. In well-formed equations, subscripts on one symbol match the superscript of the next symbol.

The set of 2D rigid transforms forms a group, where the group operation corresponding to composition of two rigid 2D transforms is defined as

$$T_s^t = T_s^t \oplus T_b^s = \left(R_s^t,\, t_s^t\right) \oplus \left(R_b^s,\, t_b^s\right) \triangleq \left(R_s^t R_b^s,\, R_s^t t_b^s + t_s^t\right) \tag{3.1}$$

The group of rotation-translation pairs T with this group operation is called the **special Euclidean group** $SE(2)$. It has an identity element $e = (I, 0)$, and the inverse of a transform $T = (R, t)$ is given by $T^{-1} = (R^T, -R^T t)$.

2D rigid transforms can be viewed as a subgroup of a general linear group of degree 3, i.e., $SE(2) \subset GL(3)$. This can be done by embedding the rotation and translation into a 3×3 invertible matrix defined as

$$T_b^s = \begin{bmatrix} R_b^s & t_b^s \\ 0 & 1 \end{bmatrix}$$

With this embedding you can verify that matrix multiplication implements composition, as in Equation 3.1:

$$T_s^t T_b^s = \begin{bmatrix} R_s^t & t_s^t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_b^s & t_b^s \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_s^t R_b^s & R_s^t t_b^s + t_s^t \\ 0 & 1 \end{bmatrix} = T_b^t$$

By similarly embedding 2D points in a three-vector, the so-called **homogeneous coordinates** of the 2D vector, a 2D rigid transform acting on a point can be implemented by matrix-vector multiplication:

$$\begin{bmatrix} R_b^s & t_b^s \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p^b \\ 1 \end{bmatrix} = \begin{bmatrix} R_b^s p^b + t_b^s \\ 1 \end{bmatrix}$$

In what follows we will work with these transform matrices exclusively.

3.2 Continuous Probability Densities

Motivation

In real robots we often deal with continuous variables and continuous state spaces. Hence, we need to extend the notion of probability to continuous variables.

3.2.1 Continuous Probability Densities

In robotics we typically need to model a belief over continuous, multivariate random variables $x \in \mathbb{R}^n$. We do this using **probability density functions** (PDFs) $p(x)$ over the variables x, satisfying

$$\int p(x)dx = 1. \tag{3.2}$$

In terms of notation, for continuous variables we use lowercase letters for random variables, and uppercase letters to denote sets of them. We drop the notational conventions of making distinctions between random variables X and their realized values x, which helped us get used to thinking about probabilities, but will get in the way of clarity below.

A **unimodal** density has a single maximum, its **mode**. In general, however, a density can have multiple modes, in which case we speak of a **multimodal** density. The **mean** of a density is defined as

$$\mu = E_p[x] = \int xp(x)dx$$

irrespective of whether the density is unimodal or multimodal. Above, the notation $E_p[.]$ stands for "the expectation of . with respect to the density p".

3.2.2 Gaussian Densities

A Gaussian probability density on a scalar x given **mean** μ and **variance** σ^2 is given by

$$\mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right\}. \tag{3.3}$$

This density is also known as the "bell curve". This density is very popular in robotics because it is so simple: it is the just the negative exponential of a quadratic around μ. In a Bayesian probability framework we interpret densities as knowledge, and the the **standard deviation** σ indicates the uncertainty we have about the quantity x.

The other reason of the Gaussian's popularity derives from the **central limit theorem**. This theorem says that the probability of the sum of a number of random variables, no matter what the density is on them, will tend to a Gaussian density. And, it does not have to be many random variables either: summing 4 random variables distributed randomly over an interval yields a cubic density, which already matches a Gaussian quite nicely.

A **multivariate Gaussian density** on $x \in \mathbb{R}^n$is obtained by extending the notion of a quadratic to multiple dimensions. We define the squared **Mahalanobis distance,**

$$\|x-\mu\|_{\Sigma}^2 \triangleq (x-\mu)^\top \Sigma^{-1} (x-\mu) \tag{3.4}$$

where $\mu \in \mathbb{R}^n$ is the mean. The quantity Σ is an $n \times n$ **covariance matrix**, a symmetric matrix indicating uncertainty about the mean. The Mahalanobis distance is nothing but a weighted Euclidean distance, and is the multivariate equivalent of the scalar squared distance

$$\|x-\mu\|_{\sigma^2}^2 \triangleq \left(\frac{x-\mu}{\sigma}\right)^2$$

but using the matrix inverse to weight this distance metric in an n-dimensional space. Armed with this, the equation for the multivariate Gaussian is

$$\mathcal{N}(x;\mu,\Sigma) = \frac{1}{\sqrt{|2\pi\Sigma|}} \exp\left\{-\frac{1}{2}\|x-\mu\|_{\Sigma}^2\right\}, \tag{3.5}$$

where the term $|2\pi\Sigma|$ in the normalization factor denotes the determinant of $2\pi\Sigma$.

Gaussian densities, whether scalar or multivariate, have some nice properties. A Gaussian density is unimodal and the mean μ is also its mode. In addition, any scalar marginal of a multivariate Gaussian is also a Gaussian. In fact, the probability density $P(y)$ of *any* linear combination $y = Hx$, with y an m-dimensional vector and H an $m \times n$ matrix, is also Gaussian with mean $H\mu$ and covariance $H\Sigma H^T$.

Exercise

1. Given a 2-dimensional density on (x, y), with mean $\mu = (\bar{x}, \bar{y})$ and covariance matrix

$$\Sigma = \begin{bmatrix} \sigma_x^2 & r \\ r & \sigma_y^2 \end{bmatrix}$$

   what is the variance of the marginals $p(x)$ and $p(y)$? Use the linearity property above.

2. Given the same 2D density, what is the mean and variance of the sum $z = x + y$?

3. Deeper thinking: what happens to the density if we push it through a nonlinear function, e.g., $z = \sqrt{x^2 + y^2}$, the Euclidean norm of the vector (x, y)? A qualitative answer is asked for.

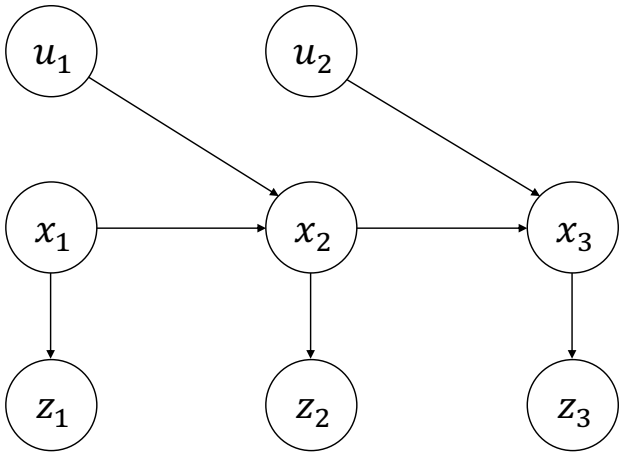


Figure 3.1: Continuous Bayes net modeling a robot with continuous states x_t, continuous measurements z_t, and continuous controls u_t.

3.2.3 Bayes Nets and Mixture Models

Continuous Bayes nets are exactly like discrete Bayes nets, except with continuous variables. Most of the concepts generalize effortlessly, and we will forego very formal definitions where we can.

An example crucial to the robotics domain is shown in Figure 3.1, which is the continuous equivalent of the dynamic Bayes net from Part I. In the figure, the continuous Markov chain backbone represents the evolution of the continuous state x_t over time, conditioned on the **controls** u_t. Notice we use new terminology here: we say *controls* rather than actions in this new, continuous world. For example, for the Duckiebot, the control u might be two-dimensional and represent the wheel speeds of the left and right wheels, respectively. Finally, the continuous measurements z_t at the bottom are conditioned on the states x_t.

We can mix and match discrete and continuous variables. A particularly simple Bayes net is a **Gaussian mixture model**. The joint density is

$$p(x, C) = p(x|C)P(C)$$

where $P(C)$ is a PMF on a discrete variable C that chooses between different Gaussians, , and $p(x|C)$ is the corresponding Gaussian mixture component. Even though each Gaussian density is unimodal, the marginal $p(x)$ is a multimodal density

$$p(x) = \sum_c p(x|C = c)p(C = c)$$

where the sum is over components. An example for a two-component mixture is shown in Figure 3.2.

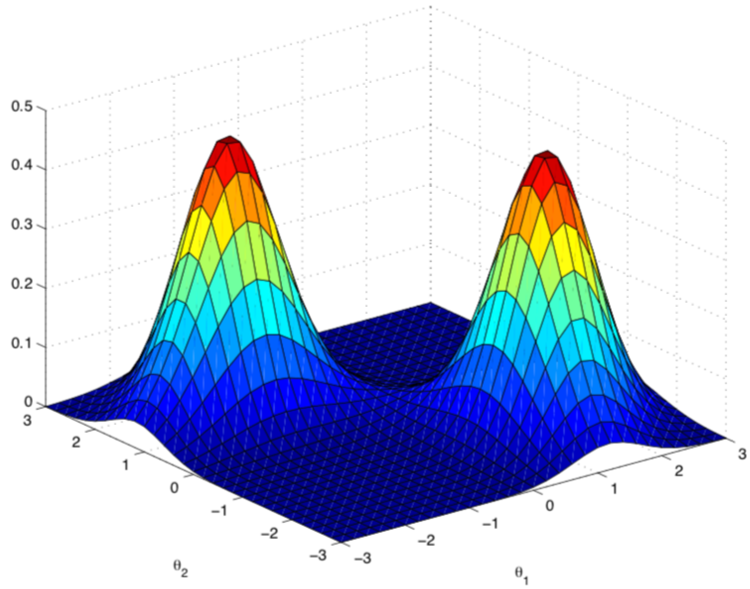


Figure 3.2: Gaussian mixture density with two components of about equal weight.

Continuous probability densities present a representational challenge. We can no longer specify CPTs: either an equation needs to be available, as with the Gaussian density above, or an arbitrary density has to be somehow approximated. One such approximation is exactly using mixture densities: we can "mix" many simpler densities, like Gaussians, to approximate a more complicated density. This is known as **Parzen window density estimation**.

3.2.4 Continuous Measurement Models

In many cases it is both justified and convenient to model measurements as corrupted by zero-mean Gaussian noise. For example, a bearing measurement from a given pose $x \in SE(2)$ to a given 2D landmark l would be modeled as

$$z = h(x, l) + \eta, \tag{3.6}$$

where $h(.)$ is a **measurement prediction function**, and the noise η is drawn from a zero-mean Gaussian density with **measurement covariance** R. This yields the following conditional density $p(z|x, l)$ on the measurement z:

$$p(z|x, l) = \mathcal{N}(z; h(x, l), R) = \frac{1}{\sqrt{|2\pi R|}} \exp\left\{-\frac{1}{2}\|h(x, l) - z\|_R^2\right\}. \tag{3.7}$$

The measurement functions $h(.)$ are often nonlinear in practical robotics applications. Still, while they depend on the actual sensor used, they are typically not difficult to reason about or write down. The measurement function for a 2D bearing measurement is simply

$$h(x, l) = atan2(l_y - x_y, l_x - x_x) - x_\theta, \tag{3.8}$$

where $atan2$ is the well-known two-argument arctangent variant. Hence, the final **probabilistic measurement model** $p(z|x,l)$ is obtained as

$$p(z|x,l) = \frac{1}{\sqrt{|2\pi R|}} \exp\left\{-\frac{1}{2}\left\|atan2(l_y - x_y, l_x - x_x) - x_\theta - z\right\|_R^2\right\}. \tag{3.9}$$

Note that we will not *always* assume Gaussian measurement noise: to cope with the occasional data association mistake, we can use robust measurement densities, with heavier tails than a Gaussian density.

3.2.5 Continuous Motion Models

For a robot operating in the plane, probabilisitic **motion models** are densities of the form $p(x_{t+1}|x_t)$, specifying a **probabilistic motion model** which the robot is assumed to obey. This *could* be derived from odometry measurements, in which case we would proceed exactly as described above. Alternatively, such a motion model could arise from known control inputs u_t. In practice, we often use a conditional Gaussian assumption,

$$p(x_{t+1}|x_t,u_t) = \frac{1}{\sqrt{|2\pi Q|}} \exp\left\{-\frac{1}{2}\left\|g(x_t,u_t) - x_{t+1}\right\|_Q^2\right\}, \tag{3.10}$$

where $g(.)$ is a motion model, and Q a covariance matrix of the appropriate dimensionality, e.g., 3×3 in the case of robots operating in the plane.

Summary

We briefly summarize what we learned in this section:

1. Continuous probability densities generalize the notion of probability distributions to continuous random variables.

2. The scalar and multivariate Gaussian densities are useful and relatively simple.

3. Bayes nets generalize as well, and even allow for discrete and continuous variables in the same Bayes net, as in the case of mixture densities.

4. Continuous measurement models typically have a measurement prediction corrupted by noise, often modeled as Gaussian.

5. Continuous measurement models follow a similar pattern, but are conditioned on controls.

3.3 Monte Carlo Inference

Motivation

With nonlinear dynamics and/or measurement models exact inference is computationally intractable. One way out is to perform approximate inference using sampling. One of the best known such algorithms are particle filters, including Monte Carlo Localization.

3.3.1 Simulating from a Continuous Bayes Net

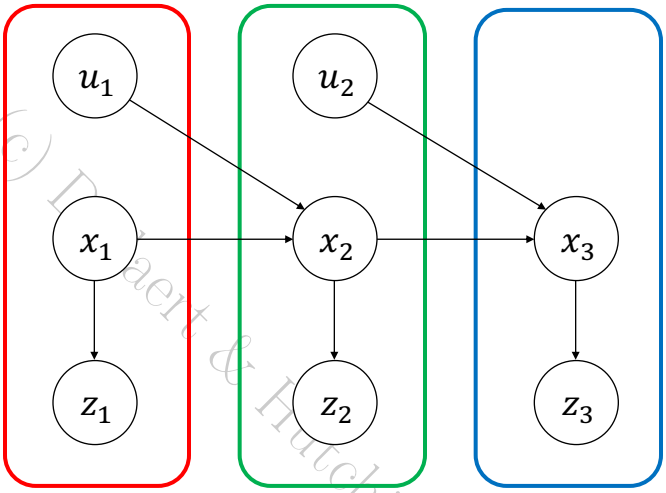


Figure 3.3: Ancestral sampling in a continuous Bayes net.

Sampling from a continuous Bayes net is done using ancestral sampling, as before: we topologically sort the DAG, and then sample the conditional densities in topological sort order. For the dynamic Bayes net from Figure 3.1, the topological sort trivially follows the temporal ordering, and we sample first within the red, then green, and then the blue time-slice, as shown in Figure 3.3.

In each slice, the story is pretty much the same. For example, in the second (green) slice we will

1. Sample the state x_2 from the continuous motion model $p(x_2|x_1, u_1)$, by evaluating $g(x_1, u_1)$, and adding Gaussian noise with mean $\mu = 0$ and covariance Q.

2. Sample the measurement z_2 from the continuous measurement model $p(z_2|x_2)$, and adding zero-mean Gaussian noise with measurement covariance R.

3. Sample a control u_2 that we will use in the next time-slice.

The only exception is the first slice, where the first state x_1 is sampled from the prior $p(x_1)$.

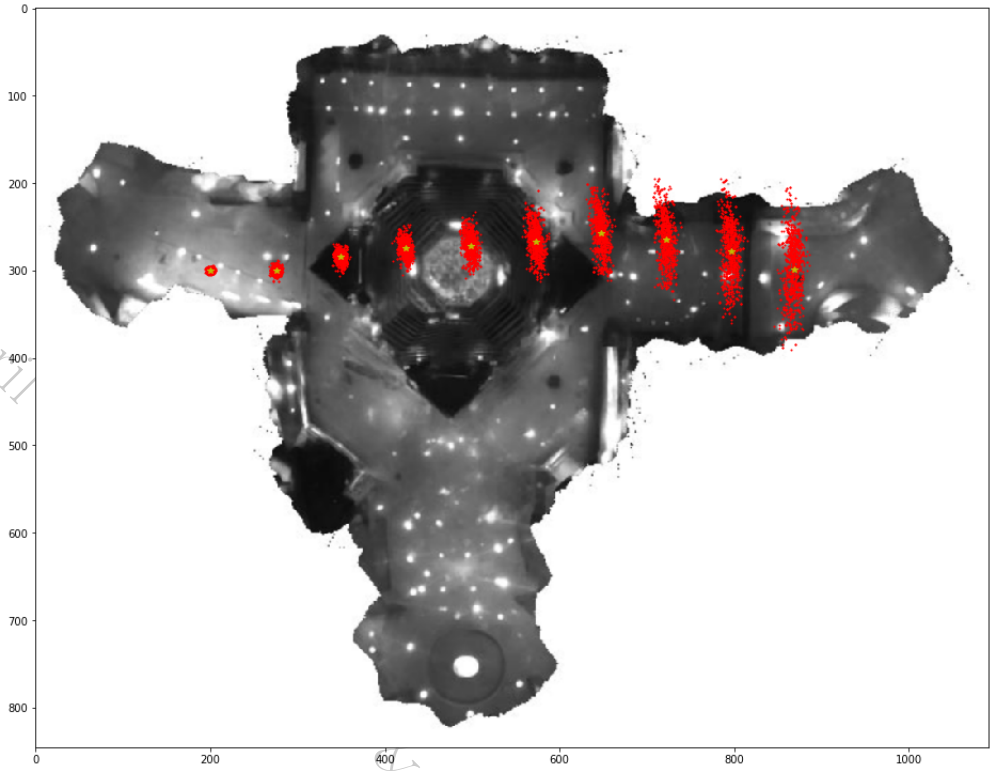


Figure 3.4: Sampling from the motion-model only, giving rise to so-called "banana-shaped" densities.

To illustrate these concepts we use an example where we use a large appearance-based map to localize a robot in a large known environment. The basic idea, which was detailed in a 1999 CVPR paper, was to generate an image mosaic of the ceiling of the environment in which the robot needs to operate, and use that during operation to localize the robot. An example of such a **ceiling mosaic** from our testbed-application is shown in Figure 3.4. The figure shows a large portion (40 m. deep by 60 m. wide) of the first floor ceiling of the Museum for American History (MAH), one of the Smithsonians in Washington, DC. The mosaic was generated from 250 images in a mapping phase.

The figure shows the result of sampling from the motion-model only, giving rise to so-called "banana-shaped" densities, shown as red point clouds. In this example, the states are 2D poses $x = (x_x, x_y, x_\theta)$, and the control has a forward velocity component u_v and an angular velocity component u_θ. The motion model is then:

$$g(x, u) = (x_x + u_v \cos(x_\theta)\Delta t, x_y + u_v \sin(x_\theta)\Delta t, x_\theta + u_\theta \Delta t)$$

What is happening is mysterious at first, until you realize that the red dots in the figure only show the marginal of the robot position (x_x, x_y), not the full 3-dimensional density on the pose. The Gaussian additive noise can make the resulting densities decidedly non-Gaussian after several iterations. The reason is the noise on the heading x_θ: this leads to non-linear effects over time.

Note that the Figure does not yet show the result of incorporating the information from the sensor. That will be discussed below.

3.3.2 Sampling as an Approximate Representation

The solution to the robot localization problem is obtained by running a Bayes filter, alternating between prediction and measurement, as we saw before. Depending on how one chooses to represent the **filtering density** $p(x_t|Z^t)$, one obtains algorithms with vastly different properties.

If both the motion and the measurement model are linear with additive Gaussian noise, and the initial state is also specified as a Gaussian, then the filtering density $p(x_t|Z^t)$ will remain Gaussian at all times. This is the basis of the classical **Kalman filter**. Kalman-filter based techniques are particularly efficient, but in most robotics applications this basic assumption of Gaussian densities is violated. Indeed, for many sensors the likelihood functions are typically complex and multi-modal, as discussed above.

To overcome these disadvantages, different approaches have used increasingly richer schemes to represent uncertainty, moving away from the restricted Gaussian density assumption inherent in the Kalman filter. These different methods can be roughly distinguished by the type of discretization used for the representation of the state space. We have already discussed **Markov localization,** which discretizes the world and can be used in grid worlds and for localizing topological maps. However, in many applications we are interested in a more fine-grained position estimates, e.g., in environments with a simple topology but large open spaces, where accurate placement of the robot is needed. Discretizing the continuous world is powerful, but suffers from the disadvantages of computational overhead and a priori commitment to the size of the state space. For example, the resolution and hence the precision at which they can represent the state has to be fixed in advance.

Finally, one can represent the density by a set of **samples** that are randomly drawn from it. This is the representation we will use in this section. In sampling-based methods one represents the filtering density $p(x_t|Z^t)$ by a set of N random samples or particles $S^t = x_t^{(s)}|s = 1..N$ drawn from it. We are able to do this be-cause of the essential duality between the samples and the density from which they are generated. From the samples we can always approximately reconstruct the density, e.g. using a histogram or a kernel-based density estimation technique.

One nice property of samples as an approximate density is that we can easily compute expected values such as the mean, using a **Monte Carlo approximation**. The name Monte Carlo derives from the city in the south of France, famous for its casinos, where of course a lot of randomness is involved. For the mean, the Monte Carlo approximation is

$$\mu = \int xp(x)dx \approx \frac{1}{N}\sum_r x^{(r)},$$

with N the number of samples. In fact, we can approximate any expectation of any function of x:

$$E_p[f(x)] \approx \frac{1}{N}\sum_r f\left(x^{(r)}\right),$$

where f can be a scalar or multivariate function of the random variable x.

Marginalization of sampled representations is very easy: if we have a set of samples

$$\left\{(x,y)^{(r)}\right\} \sim p(x,y)$$

then the sampling-based approximation of the marginal $p(y)$, for example, is obtained by just taking the y component of each sample:

$$\left\{y^{(r)}\right\} \sim p(y)$$

If we instantiate these ideas within the context of localization, the goal is then to recursively compute at each time- step t the set of samples S^t that is drawn from $p(x_t|Z^t)$. A particularly elegant algorithm to accomplish this is the particle filter, which is a Monte Carlo approximation of the Bayes filtering paradigm. Before we go there, however, we need to introduce one more piece of the puzzle in the next section, which is importance sampling.

Exercise

Prove that the Monte Carlo approximation of $y = Ax$ is equal to $A\hat{\mu}$, where $\hat{\mu}$ is the approximation of the mean.

3.3.3 Importance Sampling

Sampling from arbitrary densities is not easy in general. Gaussian densities, which are relatively easy to sample from, are the exception rather than the rule.

A simple method is **importance sampling**. Given any **target density** $p(x)$, we instead sample from a (hopefully easier) **proposal density** $q(x)$, and then produce *weighted samples* with **importance weights**

$$w^{(r)} = \frac{p(x^{(r)})}{q(x^{(r)})}$$

where $x^{(r)}$ is a sample (with index r) from $q(x)$. After sampling the required number, the importance weights can be normalized to sum up to 1.

Importance sampling can be used as a simple approximate implementation of Bayes rule. In this case, the target density is $p(x|z)$, and we can use the prior $p(x)$ as the proposal density. Hence, the weights can be computed as

$$w^{(r)} = \frac{p(x^{(r)}|z)}{p(x^{(r)})} \propto \frac{l(x^{(r)};z)p(x^{(r)})}{p(x^{(r)})} = l(x^{(r)};z)$$

where we used $p(x|z) \propto l(x;z)p(x)$, with $l(x;z)$ the likelihood of x given z. Hence, when implementing Bayes law by proposing from the prior, the weights are simply the measurement likelihoods $l(x^{(r)};z)$.

3.3.4 Particle Filters and Monte Carlo Localization

Recall that localization in robotics can be done via the Bayes filter. For continuous variables, the (exact) **filtering density** associated with a Bayes filter is given by

$$p(x_t|Z^t) \propto l(x_t;z_t) \int_{x_{t-1}} p(x_t|x_{t-1})p(x_{t-1}|Z^{t-1}), \tag{3.11}$$

where $Z^t \triangleq \{z_1, z_2 \dots z_t\}$ is shorthand for all measurement up to and including time t. In other words, the filtering density$p(x_t|Z^t)$ at time t is recursively obtained from the filtering density $p(x_{t-1}|Z^{t-1})$ at time $t-1$, by first calculating the **predictive density**

$$p(x_t|Z^{t-1}) = \int_{x_{t-1}} p(x_t|x_{t-1})p(x_{t-1}|Z^{t-1}), \tag{3.12}$$

and then weighting with the likelihood function $l(x_t; z_t)$.

There are two ways to intuitively derive particle filtering, which we discuss one by one in the first two sections below.

Resampling-based MCL

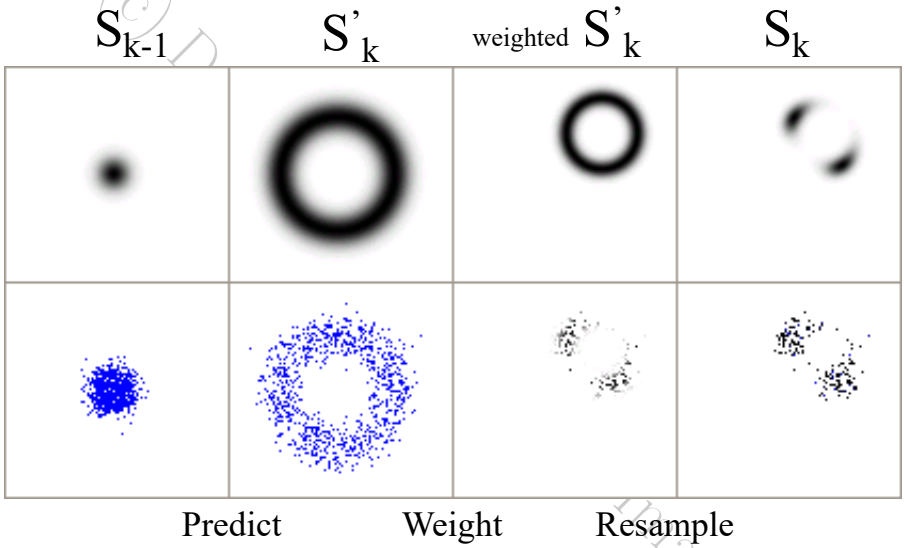


Figure 3.5: Schematic explanation of resampling-based MCL, see text.

Let us assume that we have a set of samples on X_{t-1}, i.e., $\{x_{t-1}^{(s)}\} \sim p(x_{t-1}|Z^{t-1})$. We create a tiny Bayes net, $x_{t-1} \rightarrow x_t$, and extend the sample set as in ancestral sampling, obtaining $\left\{(x_{t-1}, x_t)^{(s)}\right\} \sim p(x_{t-1}, x_t|Z^{t-1})$. We then marginalize to $\{x_t^{(s)}\} \sim p(x_t|Z^{t-1})$, weight by the likelihoods $l(x_t^{(s)}; z_t)$ to obtain $\{x_t^{(s)}, w_t^{(s)}\} \sim p(X_t|Z^t)$. Finally, resample to get back to an unweighted set of particles for the next step. In summary, given the approximate posterior at the previous time step $\{x_{t-1}^{(s)}\} \sim p(x_{t-1}|Z^{t-1})$, we

1. extend to include the next state to obtain $\left\{(x_{t-1}, x_t)^{(s)}\right\} \sim p(x_{t-1}, x_t|Z^{t-1})$;
2. marginalize to obtain the approximate predictive density $\{x_t^{(s)}\} \sim p(x_t|Z^{t-1})$;
3. weight by the likelihoods $w_t^{(s)} = l(x_t^{(s)}; z_t)$ to obtain $\{x_t^{(s)}, w_t^{(s)}\} \sim p(x_t|Z^t)$;
4. resample to get back an unweighted approximation $\{x_t^{(r)}\} \sim p(x_t|Z^t)$.

One iteration of the algorithm is illustrated in 3.5. Each panel in the top row shows the exact density, whereas the panel below shows the particle-based representation of that density. In panel A, we start out with a cloud of samples S_{k-1} representing our uncertainty about the robot position. In the example, the robot is fairly localized, but its orientation is unknown. Panel B shows what happens to our belief state when we are told the robot has moved exactly one meter since the last time-step: we now know the robot to be somewhere on a circle of 1 meter radius around the previous location. Panel C shows what happens when we observe a landmark, half a meter away, somewhere in the top-right corner: the top panel shows the likelihood $p(z_k|x_k)$, and the bottom panel illustrates how each sample is weighted according to this likelihood. Finally, panel D shows the effect of resampling from this weighted set, and this forms the starting point for the next iteration.

Mixture-density MCL

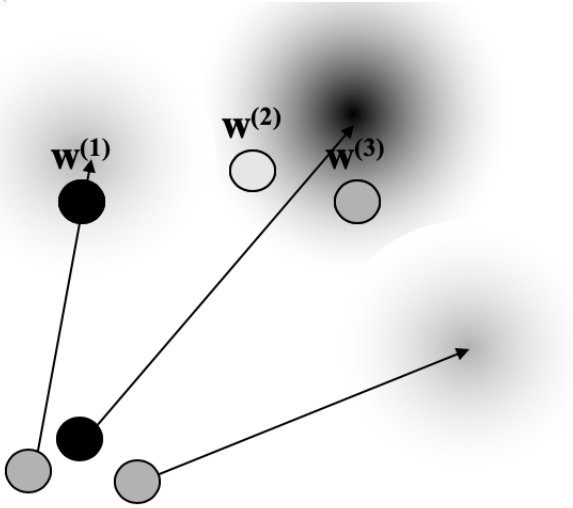


Figure 3.6: Sampling from the predictive density $p(x_t|Z^{t-1})$, approximated by a mixture density $\sum_r w_{t-1}^{(r)} p(x_t|x_{t-1}^{(r)})$ with 3 components. The middle component happens to be picked twice. After that, the three new samples each get a new likelihood weight $w^{(s)}$, as shown.

The second way to view a particle filter is by considering approximating the integral in the predictive density (3.12) with a sum approximation, i.e.,

$$p(x_t|Z^{t-1}) = \int_{x_{t-1}} p(x_t|x_{t-1}) p(x_{t-1}|Z^{t-1}) \approx \sum_r w_{t-1}^{(r)} p(x_t|x_{t-1}^{(r)}) \tag{3.13}$$

Hence, in this view we start out with a weighted sample approximation for the previous filtering density$\{x_{t-1}^{(r)}, w_{t-1}^{(r)}\} \sim p(x_{t-1}|Z^{t-1})$, and then approximate predictive density with a mixture density as in Equation 3.13. We then sample from it to get $\{x_t^{(s)}\} \sim p(x_t|Z^{t-1})$, and

then weight the samples using the likelihood: $\{x_t^{(s)}, w_t^{(s)}\} \sim P(x_t|Z^t)$, where $w_t^{(s)} = l(x_t^{(s)}; z_t)$.
In summary, given the approximate posterior using weighted samples $\{x_{t-1}^{(r)}, w_{t-1}^{(r)}\} \sim p(x_{t-1}|Z^{t-1})$, we

1. approximate the predictive density as a mixture density $p(x_t|Z^{t-1}) \approx \sum_r w_{t-1}^{(r)} p(x_t|x_{t-1}^{(r)})$;
2. sample from it to get $\{x_t^{(s)}\} \sim p(x_t|Z^{t-1})$;
3. weight the samples using the likelihood: $\{x_t^{(s)}, w_t^{(s)}\} \sim P(x_t|Z^t)$, where $w_t^{(s)} = l(x_t^{(s)}; z_t)$.

The former derivation is a straight extension of importance sampling as in the previous section, but the resampling step is somewhat magical. The latter derivation shows that the resampling can be interpreted as approximating the predictive density for X_t. Of course, both derivations lead to the exact same algorithm.

Implementation

Particle filters are very easy to implement. In fact, the inner loop can be done in just three lines of python (provided you have numpy and scipy installed and imported):

```
1  I = numpy.random.choice(N,size=N,p=W)
2  X = numpy.array([np.random.multivariate_normal(g(X[i],u), cov) for i in I])
3  W = scipy.special.softmax([-(h(x)-z)**2 for x in X])
```

Above N is the number of samples, W are the weights for the samples X, and the variable *cov* is the motion model covariance Q. In the code above, the first two lines sample from the predictive mixture density $p(x_t|Z^{t-1}) \approx \sum_r w_{t-1}^{(r)} p(x_t|x_{t-1}^{(r)})$, and the last line uses "softmax" to weight each sample using the likelihood $l(x_t^{(s)}; z_t)$ and then normalize the weights.

An example of this code in action is shown in Figure 3.7, for the ceiling localization example. In this case, the simulated sensor was simply the grayscale value at the simulated location of the robot. This is not a very informative sensor, but it does focus the distributions more on the correct location, indicated in yellow, as compared to the motion-model only particles from Figure (3.7). The likelihood functions $l(x|z)$ associated with this simple sensor model are in general complex and multimodal. To illustrate this, three different likelihood functions are shown in Figure (3.8), respectively when the robot is under a light, at the border of a light, and not under a light. It is clear that these densities do not resemble anything remotely like a Gaussian density. To see the full results with a real robot, operating in a real environment, please refer to the full 1999 paper.

Exercise

1. Show that both algorithms are the same.
2. Examine the python code in detail and try to fully understand it.

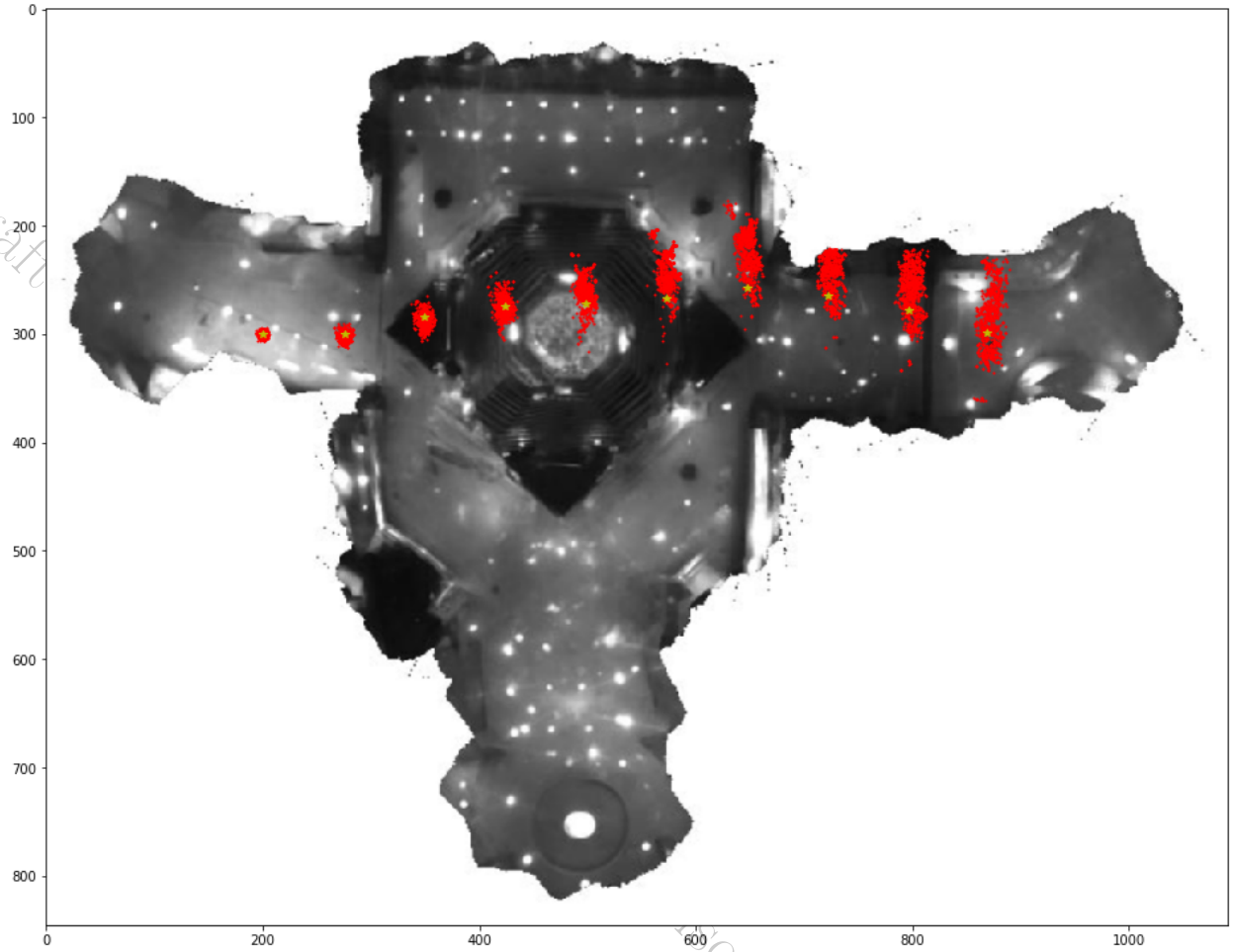


Figure 3.7: Monte Carlo localization resamples the particles based on weights derived from the sensors. In this case the sensor is simply picking up the grayscale value of the ceiling above the robot.

3.3.5 Connection with the Elimination Algorithm*

Factor graphs were not harmed in the making of this filter. But, you can interpret the weighted samples as a factor $f_1(x_{t-1})$ on x_{t-1}, then add a motion model to x_t and a likelihood factor on x_t.

A continuous version of eliminating x_{t-1}, if this was tractable, would form the product factor

$$\psi(x_{t-1}, x_t) = f_1(x_{t-1})p(x_t|x_{t-1})$$

and try to re-write it as $p(x_{t-1}|x_t)\tau(x_t)$. We saw in the discrete sum-product algorithm that this is done by marginalization. Doing this here, we obtain the new factor

$$\tau(x_t) = \sum f_1(x_{t-1})p(x_t|x_{t-1}),$$

which can be recognized as approximating the predictive density $p(x_t|Z^{t-1})$. In the elimination step we also usually calculate the conditional $p(x_{t-1}|x_t)$ but we do not need it here.

Figure 3.8: The likelihood functions $l(x|z)$ of being at a pose x given a brightness measurement z for three values of z. Left: high z indicates the robot is under a light. Middle: intermediate z. Right: low z: not under a light.

Then, eliminating x_t would form the product $\tau(x_t)l(x_t; z_t)$. This is hard if we would like to modify mixture components exactly, but easy if we first sample from them: we just re-weight then with the likelihood, and we are done. Hence, the factor graph interpretation closely aligns with the second derivation above.

3.3.6 Importance Sampling and Gibbs Sampling in Bayes Nets*

One of the downsides of Bayes filtering schemes is that we only get the posterior $p(x_t|Z^t)$ on the current state. But what if we are interested in the posterior $p(X^t|Z^t)$ on the entire trajectory? Below we discuss two schemes: importance sampling and Gibbs sampling.

Importance sampling

Importance sampling is easy. We once again partition the variables into unknown variables X and observed values Z. Rather than converting to a factor graph, we simply sample from the unknown variables in topological sort order, but whenever we encounter a variable $z \in Z$ we of course use the known value rather than sample. Then we give a weight to each sample $X^{(r)}$ equal to $w^{(r)} = p(Z|X) = \prod_z p(z|\Pi_z)$, and normalize the weights. Note that in calculating the weight we can omit any conditionals that do not involve unknown variables x.

As an example, consider again the dynamic Bayes net in Figure 3.1. If the evidence is all actions and measurements, importance sampling comes down to sampling from the motion model Markov chain, as illustrated in Figure 3.4, and then weighting by $\prod p(z_t|x_t)$. We don't need to bother with the probabilities of the actions, as they do not depend on the states $\mathcal{X}$ in any way.

However, the variance associated with producing samples from the posterior makes it only useful for small examples. Indeed: we are multiplying many small numbers together, and if the posterior $p(x|z)$ is very different from the proposal distribution, many of the samples will be wasted. For a large number of variables, almost all samples will be thus wasted.

Gibbs Sampling

A theoretically better way to sample from a posterior is using **Gibbs sampling**. For this, we convert to a factor graph with the method we explained before. Then, we start with a random initial estimate for all variables, and then repeat the following simple procedure over and over: for all variables, sample from the conditional density $p(x|\mathcal{N}_x)$, where $\mathcal{N}_x$ is the set of all neighbors of x, and pretending these neighbors are *known*. We then replace the current value of x with the sampled value. The conditional density $p(x|\mathcal{N}_x)$ can be computed by eliminating x, i.e., forming the product factor. After visiting each variable in turn in each iteration, we produce a large sequence of samples for the set of unknowns X, one for each iteration of Gibbs sampling.

There are pros and cons to Gibbs sampling, as well. The biggest advantage is that it does not has much of a variance problem as importance sampling does. The disadvantages are three-fold: (a) it might not always be easy to calculate the densities needed, or sample from them; (b) the samples of X are correlated over time; (c) it could take a long time for this sample to converge to it stationary distribution $p(X|Z)$.

Summary

We briefly summarize what we learned in this section:

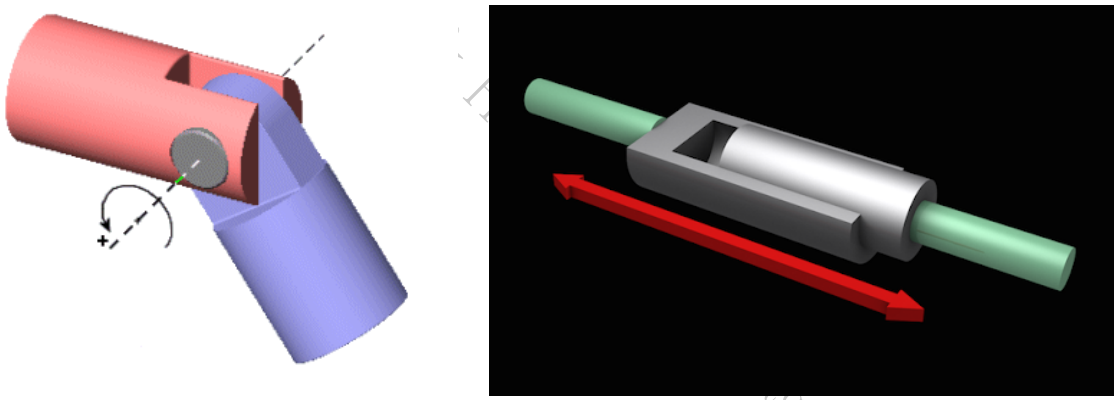
1. Simulating Continuous Bayes Nets
2. Sampling as Approximation
3. Importance Sampling
4. Particle Filters and Monte Carlo Localization
5. Monte Carlo Localization & the Elimination Algorithm

Chapter 4

Planar Manipulators

4.1 Serial Link Manipulators

4.1.1 Basic Definitions



(a) Revolute joint. (b) Prismatic joint.

Figure 4.1: Two main joint types in robot arms.

A **robot arm** (aka **serial link manipulator**) consists of a series of rigid links, connected by joints (motors), each of which has a single degree of freedom.:

- **Revolute** joint: the single degree of freedom is rotation about an axis.
- **Prismatic** joint: the single degree of freedom is translation along an axis.

More complex joints can be described as combinations of these basic joints. There are several types of joint that have more than one degree of freedom – but we do not consider

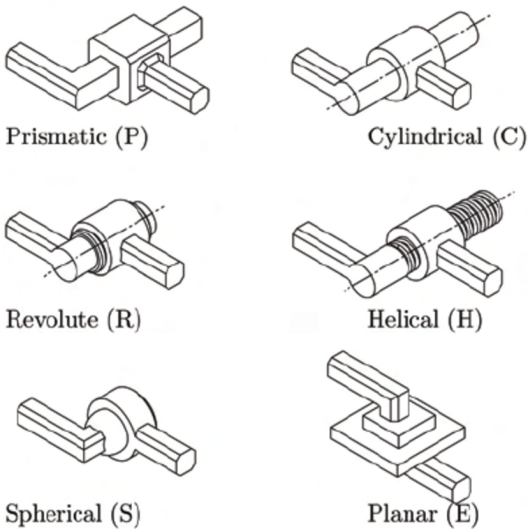


Figure 4.2: Other types of joints.

those in detail. In fact, all of the higher degree-of-freedom joints can be described by combinations of one degree-of-freedom joints, so there is no need to explicitly consider these.

A serial link manipulator has several **links**, numbered 0 to n, connected by **joints**, numbered 1 to n. Joint i connects link $i-1$ to link i. We will consider revolute joints with **joint angle** θ_i, or a prismatic joints with **link offset** d_i.

In summary, for for a robot with n joints:

- the base (which does not move) is Link 0;
- the end-effector or tool is attached to Link n;
- joint i connects Link $i-1$ to Link i.

We can treat both revolute and prismatic joins uniformly by introducing the concept of a **generalized joint coordinate** q_i, and specifying the joint type using a string, e.g., the classical Puma robot is RRRRRR, and the SCARA pick and place robot is RRRP. The vector $q \in Q$ of these generalized joint coordinates is also called the **pose** of the manipulator, where Q is called the **joint space** of the manipulator. In summary, we define the joint variable q_i for joint i as:

$$\begin{cases} \theta_i & \text{if joint is revolute} \\ d_i & \text{if joint is prismatic} \end{cases}$$

4.1.2 An RRR Example

All essential concepts can be easily developed for 2D or **planar manipulators** with revolute joints only. An example is shown in Figure 4.3. The top panel shows the manipulator at rest, along with four 2D coordinate frames: the base frame and one coordinate frame for each of the three links. For this RRR manipulator, the generalized joint coordinates are $q = \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 \end{bmatrix}^T$, and the effect of changing individual joint angles θ_i is shown at the bottom of the figure.

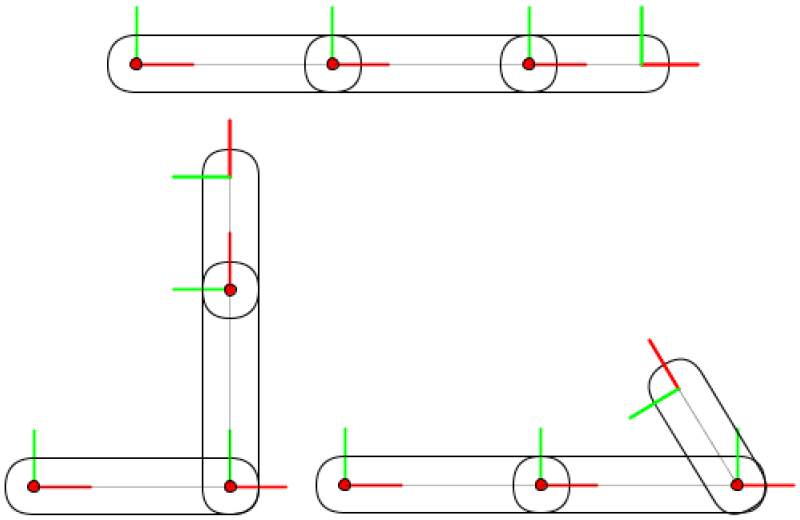


Figure 4.3: Top: rest state of a planar RRR serial manipulator, with the base frame on left. Bottom: actuating two degrees of freedom, respectively rotating θ_2 and θ_3.

4.1.3 Forward Kinematics

Kinematics describes the position and motion of a robot, without considering the forces required to cause the motion. Key questions we will answer are how to determine the pose of the end-effector tool, given joint angles, and the reverse question: what joint angles should we command to get the tool to be at a desired pose? We start with the first question, which is known as forward kinematics.

The **forward kinematics problem** can be stated as follows:

> *Given generalized joint coordinates $q \in Q$, we wish to determine the pose $T_t^0(q)$ of the tool frame T relative to the base frame 0.*

The general approach we will take is this:

- we treat each Link j as a rigid body;
- we attach the reference coordinate frame 0 to Link 0, which is merely the fixed base;
- we describe the pose (position and orientation) of every other Link j by attaching a coordinate frame $T_i^0(q)$, expressed to the base frame 0;
- if two links, say link $j-1$ and link j are connected by a single joint, then the relationship between the two frames can be described by a homogeneous transformation matrix $T_i^{i-1}(q_i)$ which will depend only on the value of the joint variable q_i! Now, the trick is to express $T_i^{i-1}(q_i)$ as a function of q_i.

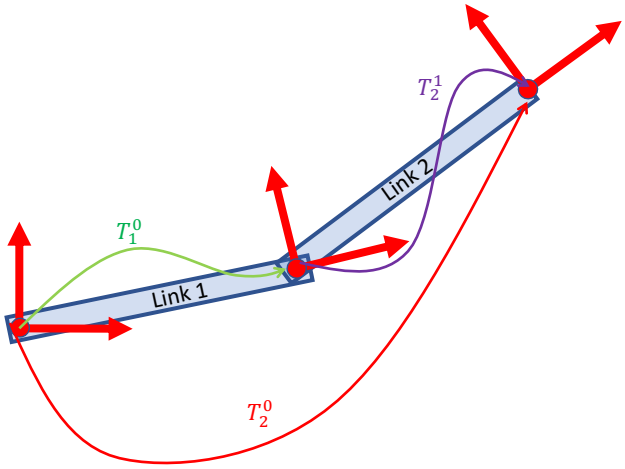


Figure 4.4: Transformations involved in the forward kinematics of a 2-link arm.

Before we do that, let us consider the example of a 2-link arm as shown in Figure 4.4. In the figure, the transform T_1^0 specifies the pose of Link 1 with respect to the base. We attached frame 1 at the end of Link 1, but it could be anywhere you like. In turn, the transform T_2^1 specifies pose of Link 2 *with respect to Link* 1. To obtain the pose of Link 2 with respect to the base frame, we multiply both transforms, using simple matrix multiplication:

$$T_2^0 = T_1^0 T_2^1$$

We now generalize this to n links and an arbitrary end-effector or **Tool frame** in Link n. Since the tool frame T moves with link n, we have

$$T_t^0(q) = T_n^0(q) T_t^n$$

where T_t^n specifies the unchanging pose of the tool T in the frame of link n. Sometimes the tool frame is taken to be identical to frame n, and then X_t^n is simply the identity matrix. The link coordinate frame $T_n^0(q)$ itself can be expressed recursively as

$$T_n^0(q) = T_{n-1}^0(q_1 \dots q_{n-1}) T_n^{n-1}(q_n),$$

finally yielding

- 2- 2

$$T_t^0(q) = T_1^0(q_1) \dots T_i^{i-1}(q_i) \dots T_n^{n-1}(q_n) T_t^n. \quad (4.1)$$

Exercise

Draw a simple two-link RP manipulator and provide the above forward kinematics formula.

4.1.4 Describing Serial Manipulators

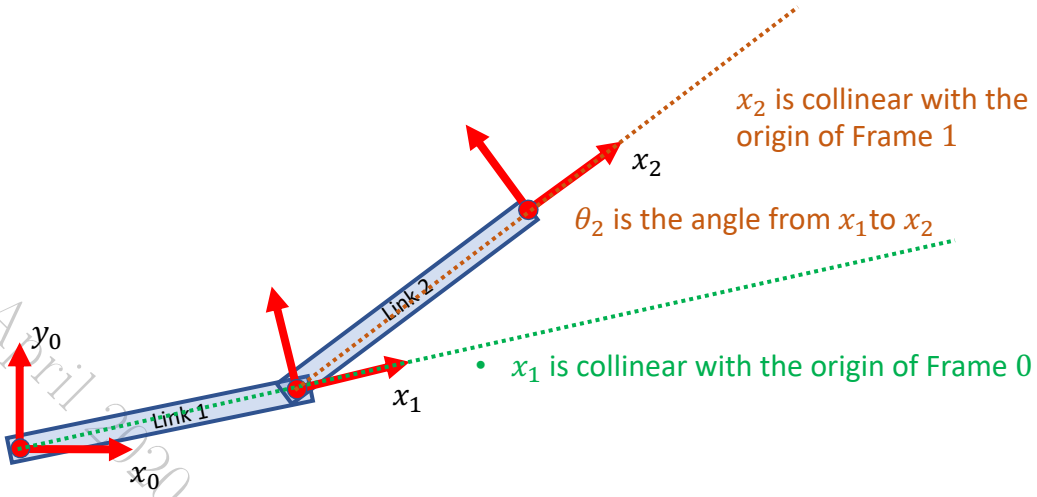


Figure 4.5: Example of assigning link frames for a 2-link arm.

Equation 4.1 is correct but we need to tie it to the robot's geometry. We can make everything easy by adopting the following strategy:

- We take frame 0 (the base frame) to have its origin at the center of Joint 1, i.e., on the axis of rotation.
- We rigidly attach Frame i is to Link i, in such a way that it has its origin at the center of Joint $i+1$.
- The x_i-axis is chosen to be collinear with the origin of Frame $i-1$.
- Define the **link length** a_i as the distance between the origins of Frames i and $i-1$.

This is illustrated for our 2-link RR-arm in Figure 4.5.

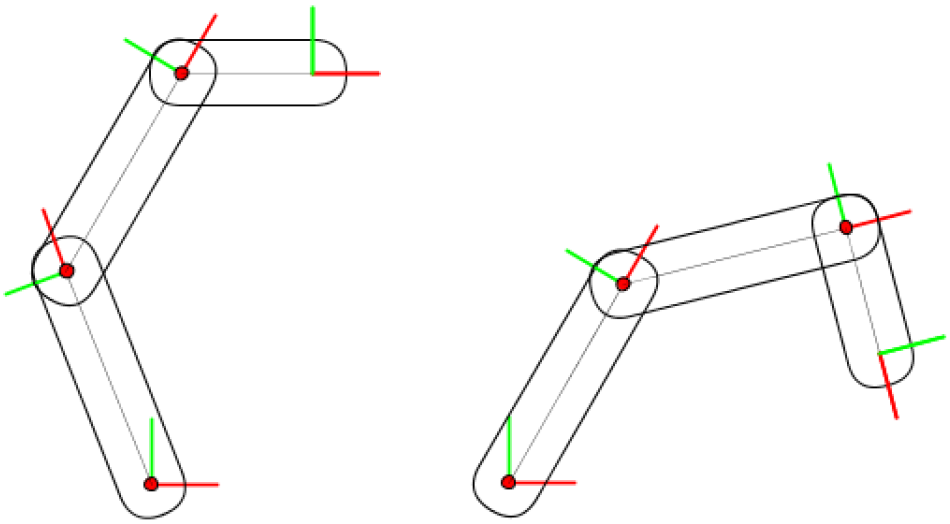
If we do all this, then the homogeneous transformation that relates adjacent frames is given by (for revolute joints):

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

4.1.5 RRR Forward Kinematics: Worked Example

As an example, Figure 4.6 shows a planar RRR manipulator with $a_1 = 3.5$, $a_2 = 3.5$, and $a_3 = 2$, in two different joint configurations. We identified the tool frame with link frame 3, i.e., $X_t^3 = I$. We then have:

$$T_1^0 = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 3.5\cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 3.5\sin\theta_1 \\ 0 & 0 & 1 \end{bmatrix}$$



(a) $\theta_1 = 112^o$, $\theta_2 = -52^o$, and $\theta_3 = -60^o$

(b) $\theta_1 = 60^o$, $\theta_2 = -45^o$, and $\theta_3 = -90^o$

Figure 4.6: Two example configurations for a planar RRR manipulator with all three joints actuated.

$$T_2^1 = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 3.5\cos\theta_2 \\ \sin\theta_2 & \cos\theta_2 & 3.5\sin\theta_2 \\ 0 & 0 & 1 \end{bmatrix}$$

$$T_3^2 = \begin{bmatrix} \cos\theta_3 & -\sin\theta_3 & 2\cos\theta_3 \\ \sin\theta_3 & \cos\theta_3 & 2\sin\theta_3 \\ 0 & 0 & 1 \end{bmatrix}$$

When multiplied out, we obtain

$$T_t^s(q) = \begin{pmatrix} \cos\beta & -\sin\beta & 3.5\cos\theta_1 + 3.5\cos\alpha + 2\cos\beta \\ \sin\beta & \cos\beta & 3.5\sin\theta_1 + 3.5\sin\alpha + 2\sin\beta \\ 0 & 0 & 1 \end{pmatrix} \tag{4.2}$$

with $\alpha = \theta_1 + \theta_2$ and $\beta = \theta_1 + \theta_2 + \theta_3$, the latter being the tool orientation.

Exercise

Provide the multiplied-out forward kinematics formula for your RP manipulator.

4.1.6 Joint-space Motion Control

Trajectory following is an important capability for manipulator robots, and three main approaches are common: (a) trajectory replay, (b) joint space motion control, and (c) cartesian space motion control.

Trajectory replay relies on an operator to perform the motion first, after which the robot simply replays the sequence, and is akin to motion-capture in movies. Even then,

to interpolate between **waypoints** obtained by robot programming, one of the two other methods is needed.

Another scenario where motion control is useful is when we just want to move the end-effector T_t^0 to a particular pose. *Inverse kinematics*, which we will discuss in detail in a future section, allows us to calculate the joint angles q_d for a particular desired pose.

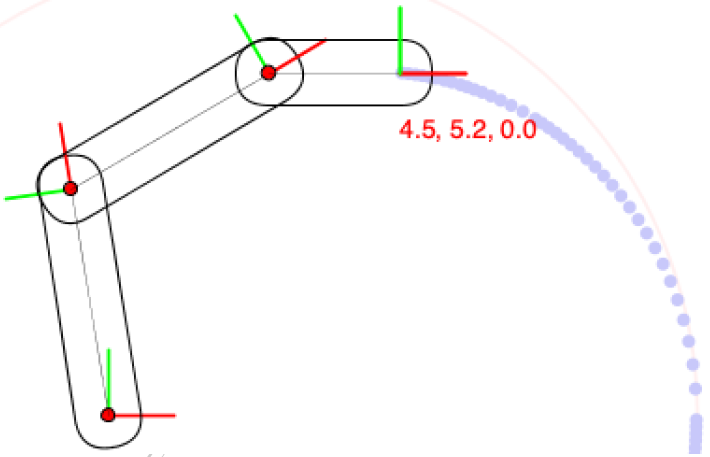


Figure 4.7: In joint-space motion control, the joint angles are linearly interpolated, but this leads to a curved path in Cartesian space.

Joint space motion control is the easiest, and applies linear interpolation or a simple control law in joint space to move from one waypoint to the other, e.g.,

$$q_{t+1} = q_t + K_p(q_d - q_t) \tag{4.3}$$

where q_t and q_d are the current and desired joint angles, and $K_p > 0$ is a gain parameter.

Before fully understanding Equation 4.3, let us consider an example of proportional joint space control in action, as shown in Figure 4.7 for the three-link manipulator. The robot started off in rest, with the end-effector at the bottom-right. The desired end-effector pose T_t^0 in this case is $(4.5, 5.2, 0.0)$ as shown in the figure. Using inverse kinematics the corresponding desired joint angle vector q_d was determined to be $q_d = [99.2, -68.8, -30.4]$, and the figure shows how proportional feedback control executes a curved trajectory upward to achieve the desired pose. The "trail" shows that the movement of the end-effector slows down considerably near the desired goal pose.

Now let us dive into what is happening in Equation 4.3: at any given moment t we calculate the **joint space error** $e_t = q_d - q_t$. Pretending for a moment that q is scalar, then if the error e_t is positive we need to increase q_t to drive the error to zero. We can use a fixed amount Δq, but a better approach is to make the amount by which we increase q_t *proportional* to the error e_t, i.e.,

$$\Delta q = K_p e_t,$$

where K_p is the proportional gain parameter. The entire scheme is called **proportional feedback control**, and has two advantages:

- as the desired value q_d is approached, the adjustment Δq becomes smaller and smaller, slowly approaching q_d so it avoids overshooting.
- it automatically deals with the case in which the error e_t is negative, in which case we then decrease q_t;
- it generalizes directly to multiple dimensions, as is needed for joint angles.

Note that setting the value of the gain K_p needs some care. If we set it too large, we might overshoot the goal, and if we set it too small, convergence to the goal will be very slow. If you are familiar with gradient descent optimization, then K_p plays a similar role as the learning rate.

Other control schemes are possible, and a very popular approach is **PID control**, where in addition to the **P**roportional gain, we also allow feedback terms calculated from the **I**ntegral of the error and **D**erivative of the error, respectively.

Discussion

Joint-space control, while very simple, might seem a sub-optimal way to proceed. Indeed, if we are at a particular pose and want to move to another, desired pose, why not make a beeline for the goal pose? In the **cartesian workspace**, i.e., the regular 2D or 3D space in which the end effector lives, the shortest path should be a line, right? The answer is not so straightforward:

1. If you are talking about the position of the end-effector, then the shortest path is indeed a straight line, in this case in 2D.
2. However, if you take into account the orientation of the end effector, the answer is not so clearcut anymore: what is the shortest path in the space $SE(2)$ of rigid transformations?
3. The shortest path *in joint space* is exactly what joint-space control executes. Only, a straight line in joint space causes a curved path in cartesian space.

In many cases, there is a desire to control the trajectory in cartesian space rather than joint space, however, which leads to the next topic.

4.1.7 The Manipulator Jacobian

Because we eventually do need to control the joint angles q, the key is to derive a relationship between velocities $[\dot{x}, \dot{y}, \dot{\theta}]^T$ in pose space in response to commanded velocities in joint space $\dot{q}$. This relationship is *locally* linear, and hence we have the following expression at a given configuration q:

$$[\dot{x}, \dot{y}, \dot{\theta}]^T = J(q)\dot{q} \tag{4.4}$$

The quantity $J(q)$ above is the **manipulator Jacobian**. For planar manipulators, as $[\dot{x}, \dot{y}, \dot{\theta}]^T \in \mathbb{R}^3$, the Jacobian is a $3 \times n$ matrix, with n is the number of joints. Each column

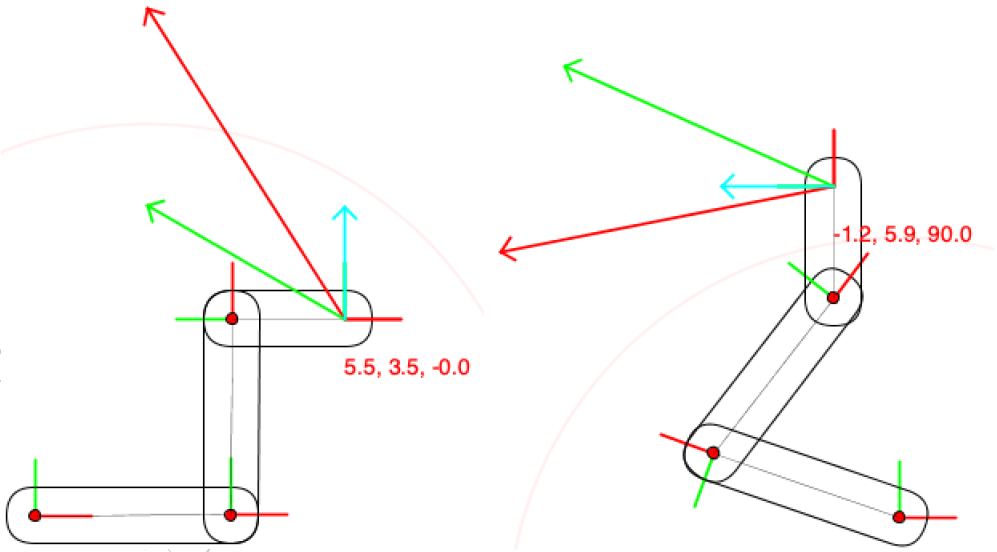


Figure 4.8: The velocities induced by a change in joint angle (red=θ_1, green=θ_2, blue=θ_3), for joint angles $q_{left} = (0°, 90°, 0°)$ and $q_{right} = (161°, -109°, 38°)$.

of the Jacobian $J(q)$ contains the velocity $[\dot{x}, \dot{y}, \dot{\theta}]^T \in \mathbb{R}^3$ corresponding a change in the joint angle q_i only, i.e.,

$$J(q) \triangleq \begin{bmatrix} J_1(q) & J_2(q) & \dots & J_n(q) \end{bmatrix}$$

where each column $J_i(q)$ is the vector of (three) partial derivatives of the pose with respect to joint angle q_i:

$$J_i(q) \triangleq \begin{bmatrix} \frac{\partial x(q)}{\partial q_i} \\ \frac{\partial y(q)}{\partial q_i} \\ \frac{\partial \theta(q)}{\partial q_i} \end{bmatrix}.$$

A graphical way to appreciate what a Jacobian means physically is to draw the 2D velocities in Cartesian space. For the three-link planar manipulator example, Figure 4.8 above shows the Jacobian $J(q)$ as a set of three velocities: red for joint 1, green for joint 2, and blue for joint 3. In other words, the red vector shows the instantaneous velocity vector for the origin of the end effector frame when joint velocity $\dot{\theta}_1 = 1$ and $\dot{\theta}_2 = \dot{\theta}_3 = 0$. These instantaneous velocities depend on the current joint angles q. The pattern is clear: these velocities are always perpendicular to the vector to the joint axis, and proportional to the distance to the joint axis.

Exercise

The Jacobian $J(q)$ varies with the configuration q. Is that always the case? Come up with a robot that has a constant Jacobian.

4.1.8 Cartesian Motion Control using the Inverse Jacobian

Cartesian motion control is a method that allows us to follow a well-defined path in Cartesian space, most often a straight line or some interpolating spline. One approach is to calculate an inverse kinematics solution at many intermediate waypoints and apply joint control again, to get from one to the other. However, there is a method by which we can avoid inverse kinematics altogether.

For a planar manipulator with three joints, i.e., $n = 3$, we can simply invert the 3×3 Jacobian $J(q)$ to calculate the joint space velocities $\dot{q}$ corresponding to a given end-effector velocity $[\dot{x}, \dot{y}, \dot{\theta}]^T$:

$$\dot{q} = J(q)^{-1}[\dot{x}, \dot{y}, \dot{\theta}]^T \tag{4.5}$$

Hence, to achieve a desired trajectory in Cartesian space, we need to calculate the desired direction *in cartesian space* at any given time. If our desired end effector pose is T_d and the current pose is $T(q_t)$ we can do this via

$$E_t(q) = \begin{bmatrix} e_x \\ e_y \\ e_\theta \end{bmatrix} = \begin{bmatrix} x_d - x(q_t) \\ y_d - y(q_t) \\ \theta_d \ominus \theta(q_t) \end{bmatrix}$$

where we extract $[x_d, y_d, \theta_d]^T$ from T_d and $[x(q_t), y(q_t), \theta(q_t)]^T$ from $T(q_t)$, respectively. Note that there are some subtleties around this: extracting a value for the orientation θ is not unique, because e.g., $\cos(\theta) = \cos(\theta + 2\pi)$. In addition, subtracting two θ values is fraught with danger because of this very same issue. To this end, we introduce the notation $\theta_d \ominus \theta(q_t)$ above to signify a "subtraction" that yields the smallest angle between two different orientations, in absolute value. The resulting magnitude $|\theta_d \ominus \theta(q_t)|$ should always be less than or equal to π.

We then calculate the corresponding joint velocities $\dot{q}$ using (4.5), and apply simple proportional control, i.e.,

$$q_{t+1} = q_t + K_p J(q_t)^{-1} E_t(q). \tag{4.6}$$

What is going on in Equation 4.6? At any given moment, the cartesian or **workspace error** $E_t(q_t)$ is evaluated, and this is a three-dimensional error in x, y, and θ. This error gives us a direction in cartesian space that we want to move, and by multiplying it the inverse Jacobian we turn this into an error in joint space. But we already know how to resolve errors in joint space! Using a proportional gain K_p, we make progress in that joint space direction, and we repeat. The secret ingredient is that unlike in pure joint-space control, we re-compute the direction in joint space at each time t to obtain a straight-line trajectory in the workspace.

4.1.9 RRR Jacobian and Cartesian Control: Worked Example

Let us calculate the Jacobian $J(q)$ for the three-link planar manipulator example. To analytically compute the Jacobian in this case, we can read off the pose $T(q)$ components from the forward kinematics equation 4.2, yielding

$$\begin{bmatrix} x(q) \\ y(q) \\ \theta(q) \end{bmatrix} = \begin{bmatrix} 3.5\cos\theta_1 + 3.5\cos\alpha + 2\cos\beta \\ 3.5\sin\theta_1 + 3.5\sin\alpha + 2\sin\beta \\ \beta \end{bmatrix}$$

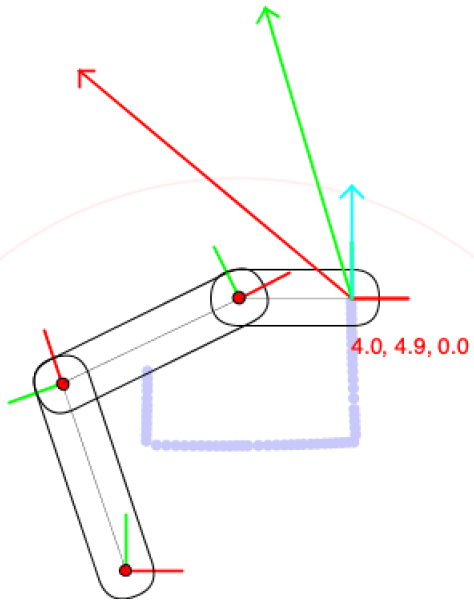


Figure 4.9: Cartesian space motion control, showing the resulting straight trajectories of the end-effector for three successive waypoints.

where $\alpha = \theta_1 + \theta_2$ and $\beta = \theta_1 + \theta_2 + \theta_3$. Hence, the 3×3 Jacobian $J(q)$ can be computed as

$$\begin{pmatrix} -3.5\sin\theta_1 - 3.5\sin\alpha - 2.5\sin\beta & -3.5\sin\alpha - 2.5\sin\beta & -2\sin\beta \\ 3.5\cos\theta_1 + 3.5\cos\alpha + 2.5\cos\beta & 3.5\cos\alpha + 2.5\cos\beta & 2\cos\beta \\ 1 & 1 & 1 \end{pmatrix} \tag{4.7}$$

Note that for a planar manipulator, all entries in the third row will always be 1 the way we defined things: the rotation rates of the joints can just be added up to obtain the rotation rate of the end effector.

An example of Cartesian motion control with proportional control is shown in Figure 4.9 for the three-link planar robot. As shown by the trail in the figure, the inverse Jacobian managed to follow straight lines in cartesian space.

4.2 Three-dimensional Geometry

The story above generalizes almost entirely to three dimensions. We start with some geometry.

4.2.1 Rotations in 3D aka SO(3)

Rotating a point in 3D around the origin from a moving body frame B to a base frame S can be done by multiplying with a 3×3 orthonormal rotation matrix

$$p^b = R^s_b p^b$$

where the indices on R_b^s indicate the source and destination frames. The columns of R_b^s represent the axes of frame B in the S coordinate frame:

$$R_b^s = \begin{bmatrix} \hat{x}_b^s & \hat{y}_b^s & \hat{z}_b^s \end{bmatrix}$$

The 3D rotations together with composition constitute the **special orthogonal group** $SO(3)$. It is made up of all 3×3 orthonormal matrices with determinant 1, with matrix multiplication implementing composition. However, *3D rotations do not commute*, i.e., in general $R_2 R_1 \neq R_1 R_2$.

4.2.2 3D Rigid transforms aka SE(3)

A point p^b on a rigid moving in three-space can be transformed by a 3D rigid transform, which is a 3D rotation followed by a 3D translation, to be expressed in a fixed base frame S,

$$p^s = R_b^s p^b + t_b^s$$

where $R_b^s \in SO(3)$ and $t_b^s \in \mathbb{R}^3$. We denote this transform by $T_b^s \triangleq (R_b^s, t_b^s)$. The **special Euclidean group** $SE(3)$, with the group operation defined similarly as in Equation 3.1. Moreover, the group $SE(3)$ is a subgroup of a general linear group $GL(4)$ of degree 4, by embedding the rotation and translation into a 4×4 invertible matrix defined as

$$T_b^s = \begin{bmatrix} R_b^s & t_b^s \\ 0 & 1 \end{bmatrix}$$

Again, by embedding 3D points in a four-vector, a 3D rigid transform acting on a point can be implemented by matrix-vector multiplication:

$$\begin{bmatrix} R_b^s & t_b^s \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p^b \\ 1 \end{bmatrix} = \begin{bmatrix} R_b^s p^b + t_b^s \\ 1 \end{bmatrix}$$

4.3 Spatial Manipulators

4.3.1 Kinematic Chains in Three Dimensions

In three dimensions the matrices are now 4×4, but exactly the same expressions are use to describe a kinematic chain:

- 2- 2

$$T_t^s(q) = T_1^s(q_1) \dots T_j^{j-1}(q_j) \dots T_n^{n-1}(q_n) X_t^n. \tag{4.8}$$

and to describe any serial manipulator we can again alternate fixed link transforms X_j^{j-1} and parameterized joint transforms $Z_j^j(q_j)$:

- 2- 2

$$T_t^s(q) = X_1^s Z_1^1(q_1) \dots X_j^{j-1} Z_j^j(q_j) \dots X_n^{n-1} Z_n^n(q_n) X_t^n \tag{4.9}$$

In 3D we typically obey the convention that the axis of rotation is chosen to be the Z-axis, hence the suggestive naming of the corresponding matrix parameterized by a joint angle. We can now also introduce **prismatic joints**, which are typically parameterized as translations *along* the Z-axis.

Figure 4.10: An example of a spatial manipulator.

4.3.2 Denavit-Hartenberg Conventions

No introduction to serial manipulators is complete without mentioning the **Denavit-Hartenberg convention**, which is a particular choice of coordinate frames to make equation 4.9 as simple as possible. In particular, as suggested by the alternation of matrices named X and Z, we ensure that

1. all joint axes are aligned with the Z-axis, and the corresponding transform is parameterized by two parameters, a rotation θ around Z and a displacement d along Z;

2. the X-axis is chosen to be the **common perpendicular** between two successive joint axes, and the link geometry is described by two parameters, a rotation α around X and a displacement a along X.

It might be surprising that only four parameters (θ, d, α, a) are needed to specify the location of one frame relative to another. However, these frames are special since they have two independent conditions imposed: the X-axis intersects the next Z-axis, and is perpendicular to it.

Subject to these two constraints, there are two popular variants in use, the proximal and distal variants, that differ on where they put the coordinate frame on the link. In the **distal** variant, the link coordinate frame $T_j^s(q)$ is made to coincide with joint axis $j+1$, and link frame n is defined to be identical to the tool frame. This convention is a bit awkward to work with.

In the simpler, **proximal** variant, also denoted the **modified Denavit-Hartenberg convention**, the link coordinate frame $T_j^s(q)$ is made to coincide with joint axis j, and the transform $T_j^{j-1}(q_j)$ between links is written as:

$$T_j^{j-1}(q_j) = X_j^{j-1}(\alpha_{j-1}, a_{j-1}) Z_j^j(\theta_j, d_j)$$

where q_j is either θ_j for a revolute joint, or to d_j for a prismatic joint, and

$$X_j^{j-1}(\alpha_{j-1}, a_{j-1}) = T_{Rx}(\alpha_{j-1})T_x(a_{j-1}) \qquad Z_j^j(\theta_j, d_j) = T_{Rz}(\theta_j)T_z(d_j)$$

4.4 Inverse Kinematics

Recall that for n links and an arbitrary end-effector pose T_t^n of the tool in Link n, we have the following forward kinematics map

$$T_t^0(q) = T_1^0(q_1) \dots T_i^{i-1}(q_i) \dots T_n^{n-1}(q_n)T_t^n. \tag{4.10}$$

Inverse kinematics (IK) is the process of finding joint angles given a desired end-effector pose $T_{desired}$, i.e., solve Equation 4.10 for q:

$$T_t^0(q) = T_{desired}$$

If $T_{desired}$ is outside the workspace of the robot, there is no solution, otherwise there might be a unique solution or multiple solutions.

The essential concepts can be explained by using a two-link planar manipulator. In this simple case, the forward kinematics are given by

$$\begin{cases} x(q) = l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2) \\ y(q) = l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2) \end{cases} \tag{4.11}$$

The inverse kinematics problem is then to find the joint angles $q = (\theta_1, \theta_2)$ such that $(x(q), y(q)) = (x_d, y_d)$, the desired end-effector position.

4.4.1 Closed-Form Solutions

Many industrial manipulators have **closed-form solutions**, and there are several ways to derive these. In this simple 2-link case above, a closed form IK solution is possible, although generally non-unique, within a radius $(l_1 + l_2)$ of the origin. I adapted a solution from John Hollerbach's 2008 lecture notes, for $l_1 = l_2 = L$: we first compute the second joint angle

$$\theta_2(x_d, y_d) = \pm 2 \arctan \sqrt{\frac{(2L)^2}{x_d^2 + y_d^2} - 1} \tag{4.12}$$

after which we compute the first joint angle

$$\theta_1(x_d, y_d, \theta_2) = \text{atan2}(y_d, x_d) - \text{atan2}(L \sin\theta_2, L(1 + \cos\theta_2)) \tag{4.13}$$

Note that Equation 4.12 is not defined if $x_d^2 + y_d^2 > (2L)^2$, which corresponds to a desired position that is out-of-range.

To extend this to the three-link example, we add a desired tool orientation, i.e., we have a desired 2D pose $T_d = (x'_d, y'_d, \theta'_d)$, where I used primes to distinguish from the 2-link example.

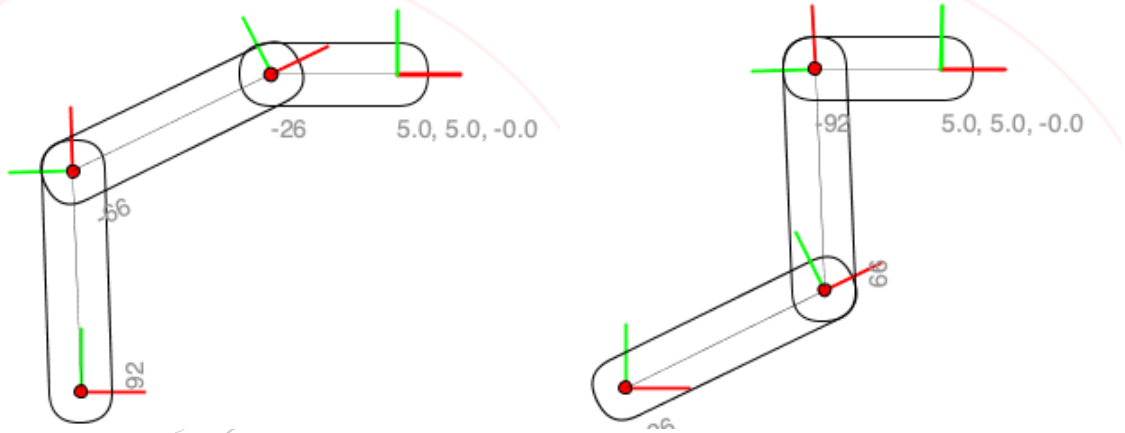


Figure 4.11: Two IK solutions for the planar three-link arm, for tool pose $(x'_d, y'_d, \theta'_d) = (5, 5, 0)$, with joint angles $q = (92^o, -66^o, -26^o)$ and $q = (26^o, 66^o, -92^o)$, corresponding respectively to choosing a positive and negative sign in Equation 4.12.

Note from the forward kinematics Equation that $\beta = \theta'_d$, and we can adapt Equations4.13 and 4.12 to accommodate for the joint 2 axis offset:

$$x_d = x'_d - 2\cos\beta \text{ and } y_d = y'_d - 2\sin\beta$$

$$\theta'_2 = (x_d, y_d) \text{ and } \theta'_1 = (x_d, y_d, \theta'_2)$$

Then, the wrist joint angle is easily computed as

$$\theta'_3 = \theta'_d - \theta'_1 - \theta'_2$$

Figure 4.11 shows two examples corresponding to choosing a different sign in Equation 4.12 for the same desired pose $(x'_d, y'_d, \theta'_d) = (5, 5, 0)$, illustrating the fact that inverse kinematics is not a one-on-one mapping, but that multiple solutions might exist, even for a non-redundant manipulator.

4.4.2 Iterative Methods

As discussed, many industrial manipulators have closed-form solutions, but they are still an area of active research and general solutions are as yet elusive. There exist, however, iterative methods to solve the IK problem.

One approach is to find the joint angles q that minimize the error between desired end-effector pose and computed end-effector pose. For example, for the simple 2-link arm we would try to minimize

$$E(q) \triangleq \frac{1}{2}\|p_d - p(q)\|^2 = \frac{1}{2}\left(x_d - x(q)\right)^2 + \frac{1}{2}\left(y_d - y(q)\right)^2 \tag{4.14}$$

where $p_d \in \mathbb{R}^2$ and $p(q) \in \mathbb{R}^2$ are the desired and computed 2D position. However, this is a non-linear minimization problem, as $x(q)$ and $y(q)$ are typically non-linear functions of the

joint-angles (at least for rotational joints). Linear least-squares problems are easier to solve, which motivates us to start with an initial guess q for the joint angles, and to linearize the forward kinematics around this pose,

$$p(q+\delta q) \approx p(q) + J(q)\delta q \tag{4.15}$$

where $J(q)$ is once again the manipulator Jacobian, derived in Section **??**. For the simple 2-link planar manipulator we can only hope achieve a desired position $p_d \in \mathbb{R}^2$, and hence the Jacobian $J(q)$ is only a 2×2 matrix, easily calculated as

$$\begin{aligned} J(q) &= \begin{bmatrix} \frac{\partial x(q)}{\partial \theta_1} & \frac{\partial x(q)}{\partial \theta_2} \\ \frac{\partial y(q)}{\partial \theta_1} & \frac{\partial y(q)}{\partial \theta_2} \end{bmatrix} \\ &= \begin{bmatrix} -y(q) & -l_2 \sin(\theta_1+\theta_2) \\ x(q) & l_2 \cos(\theta_1+\theta_2) \end{bmatrix} \end{aligned}$$

Substituting the approximation 4.15 into the objective 4.14 we obtain

$$E(q+\delta q) \approx \frac{1}{2} \| (p_d - p(q)) - J\delta q \|^2 \tag{4.16}$$

where the dependance of J on q is implied, for notational simplicity. The above says that we can make the position error $e(q) \stackrel{\Delta}{=} p_d - p(q)$ go to zero by calculating a change δq in joint angles, such that

$$J\delta q = p_d - p(q)$$

For a non-redundant manipulator the Jacobian $J(q)$ is square, and its inverse generally exists (except at the boundaries of the workspace). Hence, we could try to invert it immediately:

$$\delta q = J^{-1} (p_d - p(q)) \tag{4.17}$$

However, because the forward kinematics are non-linear, we might have to iterate this a few times, and the process might in fact diverge.

4.4.3 Damped Least-Squares

A safer approach is to impose some penalty for taking steps δq that are too large, which can be done by adding a term to the objective function 4.16:

$$E(q+\delta q) \approx \frac{1}{2} \| (p_d - p(q)) - J\delta q \|^2 + \frac{1}{2} \| \lambda \delta q \|^2 \tag{4.18}$$

This can be solved for δq by setting the derivative of E in Equation 4.18 to 0,

$$-J^T ((p_d - p(q)) - J\delta q) + \lambda^2 \delta q = 0$$

which, after simply re-arranging, leads to a damped least-squares iteration:

$$\delta q = \left(J^T J + \lambda^2 I \right)^{-1} J^T (p_d - p(q)) \tag{4.19}$$

The value of λ is chosen as to make the iterative process converge, and can even be increased automatically when a low value is seen to lead to divergence. On the other hand, too high a value might lead to slow convergence.

One strategy is to proceed very cautiously, i.e., make λ very large, in which case Equation 4.19 essentially becomes gradient descent:

$$\delta q = \lambda^{-2} J^T \left(p_d - p(q) \right) \tag{4.20}$$

Because it only involves J^T, this is also called the **transpose Jacobian method**, but it has the disadvantage of converging very slowly, see Fig. 4.12.

A method that picks λ automatically is the **Levenberg-Marquardt** method: start with a high λ, and then reduce it by a constant factor at every iteration until the error goes up instead of down: in that case undo the change in λ and try again.

4.4.4 Iterative IK Methods Summary

In summary, all iterative inverse kinematics approaches share the same structure:

Algorithm 7 The basic iterative IK algorithm.

```
1: function IterativeIK(x_d)
2:     q ← q_0                          ▷ Guess an initial value for joint angles
3:     while E(q) ≠ 0 do                ▷ While not converged
4:         e ← p_d − p(q)               ▷ Calculate error between desired and computed pose
5:         Calculate δq = f(J(q), e)    ▷ Using Eqn. (4.17), (4.19), (4.20)
6:         q ← q + δq                   ▷ Update the joint angles q
7:     return q                         ▷ Joint angles yielding p_d
```

4.4.5 Exercise

Provide joint limits for your RP robot and describe the resulting workspace.

4.5 Redundant Manipulators

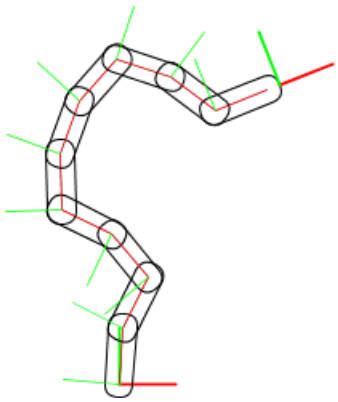


Figure 4.12: A highly redundant planar manipulator.

One advantage of the least-squares approaches above is that they also work for $n > m$, i.e., for **redundant manipulators**. An example of a highly redundant planar manipulator is shown in Figure 4.12. In this case the inverse in Equation 4.17 does not exists (as J is non-square), but we can use the **pseudo-inverse** $J^\dagger$,

$$\delta q = J^\dagger (p_d - p(q)) \tag{4.21}$$

where $J^\dagger \triangleq J^T \left(JJ^T\right)^{-1}$ for $n > m$.

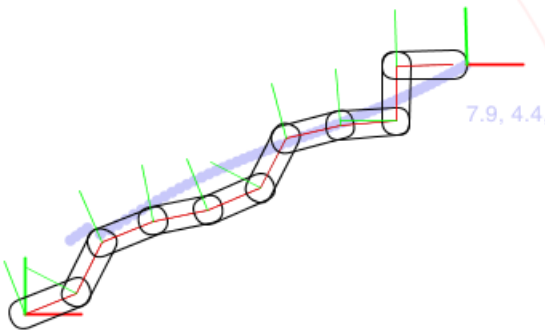


Figure 4.13: Convergence of iterative IK with the transpose Jacobian method.

Note that for redundant manipulators the pseudo-inverse is equivalent to damped least squares with $\lambda = 0$. Of course, even in the redundant case we can take λ to be non-zero and just apply gradient descent (4.19), or make λ large in which case we recover the transpose method (4.20). An example of the latter is shown as a trajectory in Figure 4.13.

In a redundant manipulator there are typically an infinite number of ways to attain a desired end-effector pose. The methods above arbitrarily pick one of the solutions. However, the extra degrees of freedom might be put to other uses, as well: for example, we might want to favor configurations that require less energy to maintain, or avoid singularities, or even - in the case of using IK for animation - follow a certain style [**?**]. This can be done by adding

an additional, user-defined penalty term $E_{user}(q)$ to the error function that penalizes certain joint configurations and favors others:

$$E(q) \stackrel{\Delta^2}{=} \frac{1}{2}\|p_d - p(q)\|^2 + E_{user}(q)$$

As long as the derivative of $E_{user}(q)$ is available, it is easy to incorporate this extra information. A simple example is to make the joint angles as small as possible, i.e., penalizing a deviation from the rest state, leading to

$$E(q) \stackrel{\Delta^2}{=} \frac{1}{2}\|p_d - p(q)\|^2 + \frac{1}{2}\|\beta q\|^2$$

After linearizing, we have

$$E(q + \delta q) \approx \frac{1}{2}\| \left(p_d - p(q)\right) - J\delta q\|^2 + \frac{1}{2}\|\beta(q + \delta q)\|^2$$

which yields the following update,

$$\delta q = \left(J^T J + \beta^2 I\right)^{-1} \left(J^T \left(p_d - p(q)\right) - \beta^2 q\right)$$

which looks very much like the damped least-squares iteration (4.19), except that now there is an extra error term that drives the joint angles to zero.

Finally, a user could also impose hard equality or inequality constraints. One such constraint is that the robot should never self-intersect or collide with objects in its environment. In dynamic environments, then, these constraints define a **configuration space** that can change over time. We then get into the realm of fully fledged **motion planning**, which is a prolific and active area of research.

4.5.1 Exercise

Suppose we modify the RP robot to include an additional R joint, making it RPR. What are the consequences for the workspace of the robot?

Chapter 5

Vision for Robots

5.1 Computer Vision Introduction and Fundamentals

Attribution: much of the text below is inspired by slides widely shared, re-purposed, and re-mixed in the computer vision teaching community. We mention some of these in the preface, but the list is probably not complete. Yet we are grateful to all who contributed to this mindshare.

Motivation

To act sensibly in the world, robots need to infer knowledge about the world from their sensors. One of the most affordable *and* richest sensors are **cameras**. They are actually amazing devices, having the ability to measure precise angles in space. Unfortunately, understanding camera images is not easy: despite the apparent ease by which we (humans) seem to understand what we see, replicating this in a computer has been fiendishly hard. Since the sixties, when "solve computer vision" was famously assigned as an undergraduate summer project, researchers have tried to tackle this problem. However, since 2012 the emergence of **deep learning** has led to incredible progress. Perception for robotics has been following closely behind, and we live in an exciting age where long-standing problems in robot vision are being conquered.

5.1.1 What is Computer Vision?

Computer Vision (CV) is in some sense the inverse of Computer Graphics. Graphics pipelines take 3D models, where they are in space, and their material properties, and then produce pleasing images. This should be very familiar to the gamers among you, and graphics processing units (GPUs) excel at those computations. But Computer Vision goes the other way: it takes images and tries to recover the models, their geometry, and their properties. This is very hard, as in the projection information about the 3D world has been destroyed and "mushed together" from 3D into 2D. As an aside, a third field of interest is *Computational Photography,* which is all about transforming images to images.

As a discipline, CV interacts with a lot of other fields. There are technical sub-disciplines of CV that include Image Processing, Geometric Reasoning, Image and Object Recognition,

Figure 5.1: Crazy eye. TODO: attribution.

and Deep Learning as applied to CV. Then there some fundamental fields that help CV researchers in solving these sub-problems, such as optics, geometry, computational photography, statistics, and machine learning. And CV finds applications in Robotics, HCI, medical fields, neuroscience, etc...

To introduce the problem a bit more, consider the image of an outdoor scene in Figure 5.2, about which we could ask a variety of questions. These questions might include, bur are not limited to:

1. What kind of scene is this?
2. Where are the cars in this image?
3. How far is the building?
4. When was this image taken?
5. Where was this image taken?

In other words, we would like to develop algorithms that can take the raw numerical representation of the image (of which you see a rendered representation in the figure) and make sense of that data, answering all or some of the questions above. Note that in this image the date is overlaid on the image in the bottom right, so these algorithms should include the ability to read! Some of the other questions are much harder to answer, however. Also, images are only one type of visual representation that CV considers: others are video, stereo

Figure 5.2: Outdoor scene about which we could ask a variety of questions. TODO: attribution.

imagery (including stereo video), multi-view imagery in general, multispectral imagery, and even information coming from depth sensors - especially in robot vision.

Yet, despite our own brains seeming to solve this task with *apparent* ease creating computer vision algorithms that work as expected is *really* hard. Introspection is a dangerous activity for computer vision researchers, because we have very little understanding about how our brain solves this problem. It has been said numerous time that "vision [in animals] is an amazing feat of natural intelligence"[1]: neural circuitry devoted to the task of vision in humans takes up a large fraction (approximately a third) of the human brain, and even more in other species. Interestingly, cuttlefish and octopuses also devote large fractions of their brains to the display of visual information. In either case, the large amount of neural machinery devoted to this seems to be roughly proportional to the amount of visual information that is sensed (or displayed) and which has to be digested into actionable information for the animal. And, while we have some sense of the computations that go in the eye and on the information right after the sensing stage, the function of higher-level brain structures is still poorly understood.

While scientific inquiry human and animal vision has al long history, the field of *computer* vision itself started around the sixties, including the venerable summer project assigned by Minsky at MIT. Initially it was very geometry driven, partly because computational resources where scarce. A large amount of progress was made in the nineties and 2000s with the influx of ideas from machine learning, which started to leverage data to do better at some tasks.

[1]This is quoted in so many CV lecture slides it is hard to track down who to attribute it to.

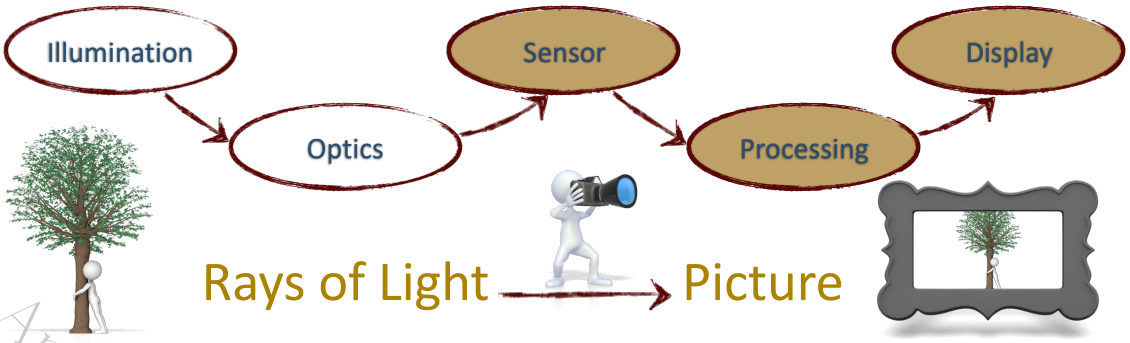


Figure 5.3: The image acquisition pipeline. Image by Irfan Essa.

This took off in a big way in the 2010s with the advent of deep learning, which leveraged the computational graphics pipelines (GPUs!) to really take advantage of large-scale data such as internet image collections.

5.1.2 Applications of CV

Computer vision algorithms have found wide application and have in some cases transformed entire industries.

TBD: please refer to the slides.

5.1.3 Images as 2D arrays

The image acquisition pipeline for a typical camera is illustrated in Figure 5.3. The light reflected of the scene goes through the camera's optics after which is collected by the sensor. After further processing, which typically includes considerable "improvement" by the camera manufacturer, the image is stored in a binary format. The processing these days can involve considerable algorithmic improvements and, on modern smartphones, it could be that the final stored image is derived from a "burst" of images taken by the sensor. Finally, the image can be displayed on a screen or projected, which can involve more tweaking of the actual image.

Grayscale digital images are stored as 2D array of **values**. An example is shown in Figure 5.4. This is a 512 by 512 **resolution** image, making for a quarter megapixel image. The resolution is typically given as $w \times h$, i.e. image width by image height. People use the term "megapixel" or MP as a shorthand for a million pixels. Example sensor resolutions are 640×480 (VGA, 0.3MP), 1280×720 (0.9MP), or 3840×2160 (8.3MP). The **aspect ratio** is the ratio of width to height, e.g., VGA is 4:3, and another important aspect ratio is 16:9.

Each of the elements of the array is a picture element or **pixel** which stores an intensity value, typically between 0 and 255, i.e., one byte or 8 bits. Sometimes, in high-end cameras or when images are stored in RAW format, 10, 12, or 16 bits are used as well. We speak of

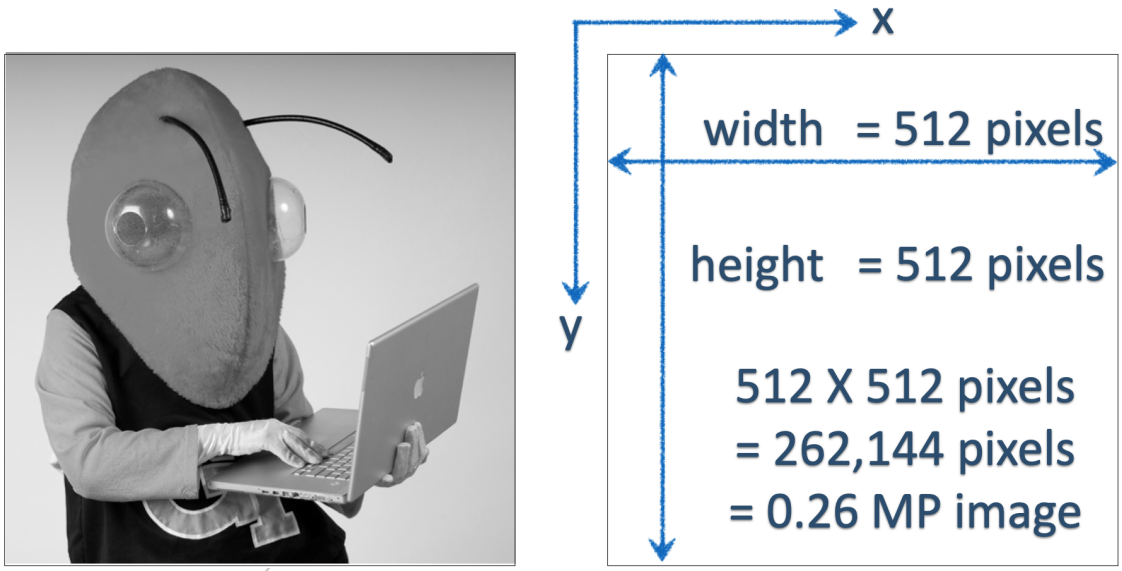


Figure 5.4: Black and white image, stored in a 512 by 512 array of bytes, making for a quarter megapixel image. Image by Irfan Essa.

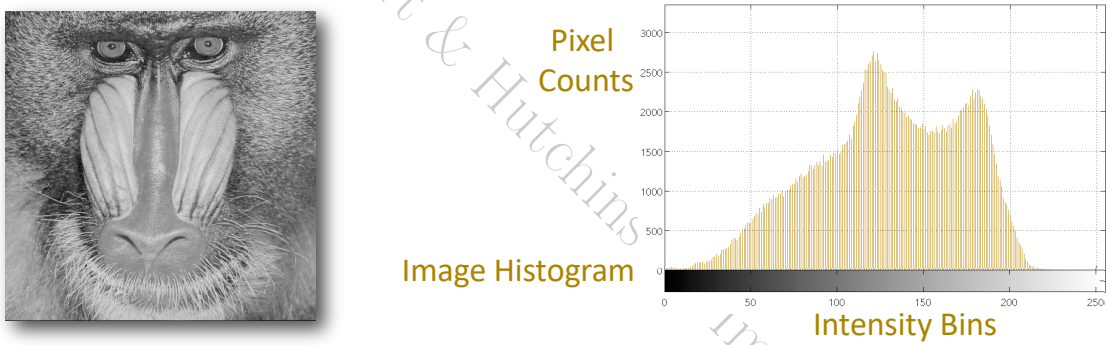


Figure 5.5: An image histogram is one way to display global image statistics.

the **bit-depth**. We say that that the value 0 corresponds to black, whereas 255 corresponds to pure white. In reality, however, the exposure time and aperture of the camera determine the dynamic range of light intensities the sensor will pick up. Hence it is more correct to say that the value 0 corresponds to "the light intensity was not registered by the sensor", and the value 255 corresponds to "the sensor was overexposed in this area".

An image can also be visualized as a 2D function, which is a sampled representation of a continuous intensity field $I(x, y)$. We typically use x, y when referring to continuous coordinates and i, j when referring to discrete coordinates. When using continuous coordinates, typically $x$is left to right and y increases downwards. Note that different conventions are used in different APIs, e.g., in MATLAB the discrete i refers to the row and j refers to the column. And OpenGL uses a y-axis which points upwards, not downwards. Careful attention is thus needed when reading literature around computer vision and/or coding up image processing algorithms.

Figure 5.6: Color images are stored as three channels: R(ed), G(reen), and (B)lue.

One way to analyze images is through image statistics, e.g., the mean or median intensity value. A more fine-grained device is the image histogram, illustrated in Figure 5.5, which tabulates the number of pixel for a given intensity value. The particular example shows that there are no under- or over-exposed pixels, and there are two peaks, probably corresponding to the bright and more grey areas of the animal pictured.

Color images are stored as three separate arrays corresponding to the red (R), green (G), and blue (B) channel. We speak of an **RGB image**. In (very) expensive cameras this can literally correspond to three separate sensors that receive light filtered through a red, green, and blue filter, respectively. However, in almost all consumer cameras on the market today, and certainly in all smartphone cameras, the color is created by an array of color filters glued on top of the sensor, called a **Bayer pattern**. Hence, the color signal is captured at a lower resolution than the actual sensor resolution, and the full resolution color image is estimated from that signal in a process called **debayering** or **demosaicing**.

5.1.4 Basic Image Processing

Image processing refers to algorithms that work on raw pixel data or information that is derived from it. We will not replicate an entire image processing course here, but will discuss two broad categories here, which are

1. Per-pixel image processing
2. Area-based image processing

The latter category will be discussed in the next section. The first category can be expressed in terms of a simple functional relationship, e.g., **contrast** enhancement simply multiplies the pixel values with a constant a,

$$g(x) = af(x)$$

where x refers to a pixel location, $f(x)$ is the source image, and $g(x)$ is the target image. A **brightness** adjustment, on the other hand, is just adding a constant:

$$g(x) = f(x) + b$$

Hence, you can see that the two "master knobs" of any image editing program, brightness and contrast, simply implement a linear transformation of the data:

$$g(x) = af(x) + b$$

One important caveat applies: because the image values are clipped at 0 and 255, respectively, this is not *truly* a linear transformation. However, some image editing software does all the math in signed floating point, and only does the clipping at the end, when saving a file, which gives one temporarily relief of this non-linear clipping. In fact, it is common practice to rescale all intensity values to the domain $[0, 1]$ upon reading, and work in a bit-depth independent manner throughout the entire workflow.

Per-pixel image processing can be non-linear as well. For example, **gamma correction** is an operation frequently used to adapt images for human consumption and/or correct for display non-linearities. It has a single parameter γ, hence its name:

$$g(x) = f(x)^{\gamma}$$

Another example of a nonlinear operation is **histogram equalization**, which computes and then flattens the histogram of pixel values in an image.

Finally, **image arithmetic**, e.g., adding, subtracting, multiplying and dividing images are all per-pixel operations. Again careful attention is required to the clipping of the intensity values. Image arithmetic is frequently used in image compositing, where an **alpha matte** encodes the transparency of objects, and compositing of a foreground F on a background B can be done by some clever mixing. An example is shown in Figure 5.7. Note that in these workflows all images are handled in bit-depth independent floating point intensity, ranging from 0.0 to 1.0, as discussed above. In that context, the alpha matte also ranges from 0.0 (completely transparent) to 1.0 (completely opaque), and if you look carefully you can see that at the boundary of the foreground object, the values are actually somewhere between the two, indicating "mixed pixels".

5.1.5 Image Filtering

TBD. Refer to the slides.

Summary

We briefly summarize what we learned in this section:

1. Computer Vision defined
2. Applications of CV are plentiful! (TBD)
3. Images are 2D arrays of pixel values
4. Basic image processing: contrast, intensity, histogram eq., arithmetic
5. Image filtering: convolution (linear) and non-linear (median) (TBD)

Figure 5.7: Image compositing, frequently used in the special effects industry. Top to bottom, left to right we have the alpha matte a, the pre-multiplied foreground image aF, the pre-multiplied background image $(1-a)B$, and the final composite $aF + (1-a)B$.

5.2 Geometry for Computer Vision

Motivation

We need to model the image formation process. The camera can act as an (angular) measurement device. However, we need a mathematical model for a simple camera. Two cameras are better than one: they can provide metric measurements.

Notation

A word about notation: we will often use lowercase letters for image quantities such as 2D points p and q, and uppercase letters for 3D quantities, such as 3D points P and Q. We also follow this rule when talking about the coordinates of a point, which we will write in two different ways, depending on whether we mention in the text, such as $p = (x, y)$, or in a display formula, such as

$$p = \begin{bmatrix} x \\ y \end{bmatrix} \qquad or \qquad P = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Mathematically, we think of 2D and 3D points as column vectors, but the (.) notation helps us avoid always writing a transpose, i.e., we have $p = \begin{bmatrix} x & y \end{bmatrix}^T = (x, y)$.

5.2.1 Pinhole Camera Model

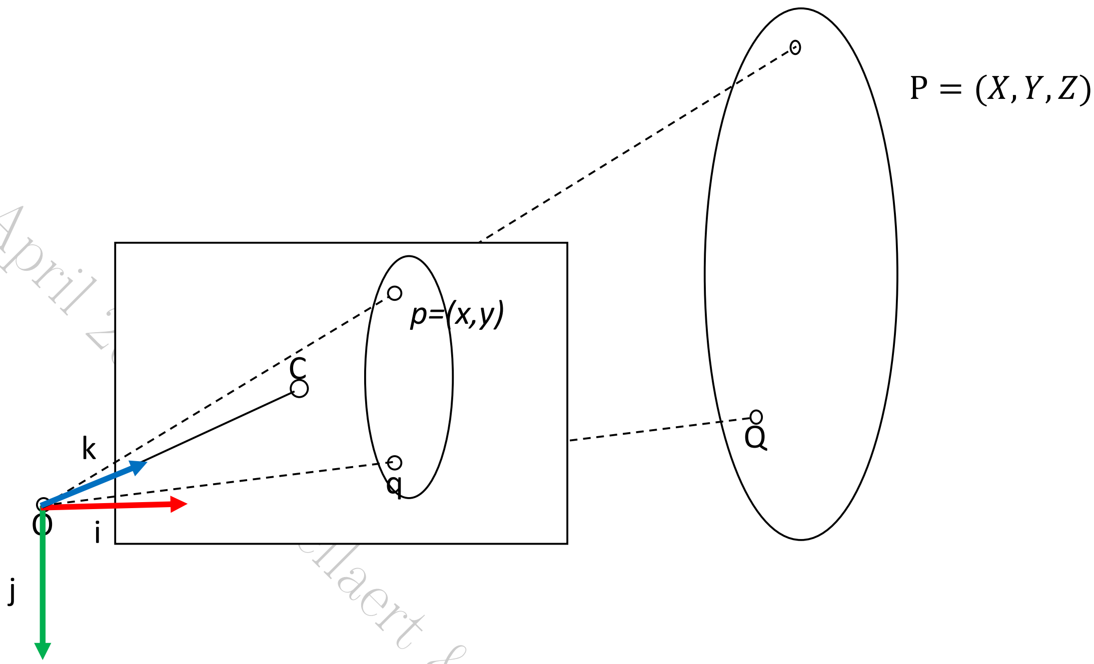


Figure 5.8: Pinhole camera model for projecting 3D points P in the scene to 2D points p in the image.

Geometrically, the simplest and most-used camera model is the **pinhole camera model**. In short, to project a 3D point $P = (X, Y, Z)$ to a 2D image, we use the following equation:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} X/Z \\ Y/Z \end{bmatrix} \tag{5.1}$$

This assumes that the 3D point P is expressed in **camera coordinates**. These are the 3D coordinates of a point, expressed in a coordinate frame rigidly attached to the camera, with its origin at the optical center, and where the Z-axis is the optical axis. Furthermore, it is customary to have the X-axis point right, and the Y-axis *down*. This then yields 2D coordinates (x, y) where x increases to the right and y increases downward. The pinhole camera model is illustrated in Figure 5.8. The camera coordinate frame in which the 3D points are expressed is shown on the left. The point O is the **optical center** and the point c is the **image center**, i.e., where the **optical axis** (the 3D Z-axis, by our convention) pierces the image plane.

The pinhole camera models that fact that 3D space is projected into 2D, and in particular we lose depth information. In camera coordinates, the Z coordinate of a 3D point is called the **depth** of the point. But Equation 5.1 shows that in the 2D projection, only the proportion of X resp. Y with respect to Z is retained: we cannot actually recover the actual 3D location

of the point after projection. Indeed, if we have a point $P = (X, Y, Z)$, and multiply all coordinates by 2, the projection is the same:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2X/2Z \\ 2Y/2Z \end{bmatrix} = \begin{bmatrix} X/Z \\ Y/Z \end{bmatrix} \tag{5.2}$$

In general, all points on the **viewing ray** $\lambda P = (\lambda X, \lambda Y, \lambda Z)$, with λ an arbitrary scalar, will yield the same projection in the image.

5.2.2 Vanishing Points

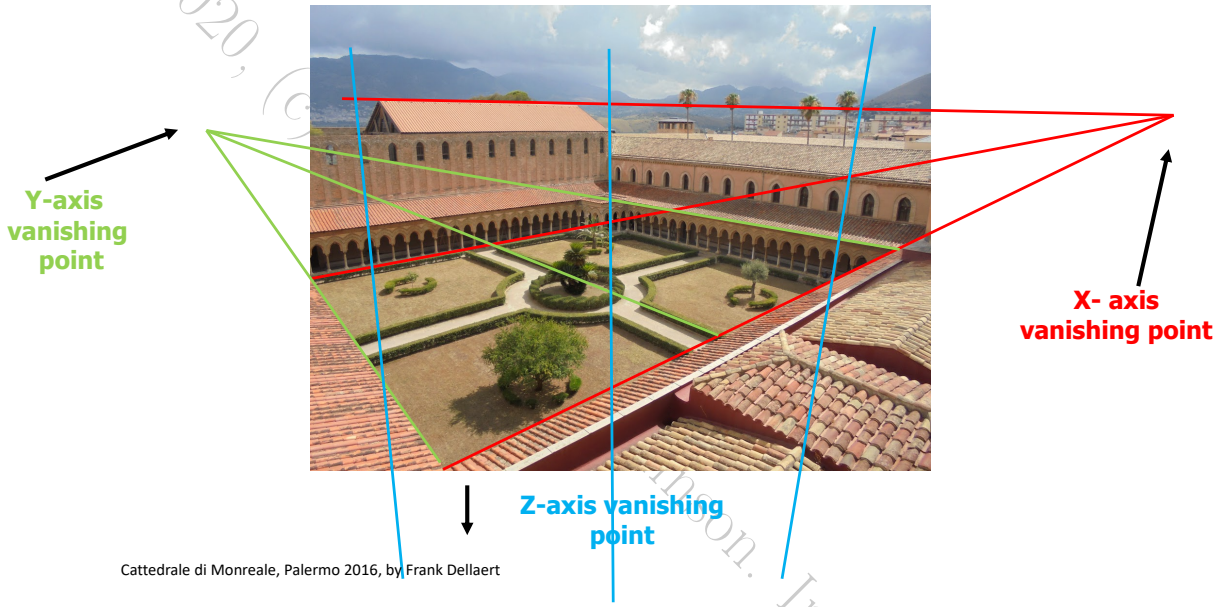


Figure 5.9: This image, taken in Palermo, Sicily, shows that parallel lines in the scene intersect when projected by a camera, although not necessarily within the image bounds.

In addition to losing the notion of depth, as discussed above, another property we lose projecting into an image is parallelism of lines. In other words: parallel lines in the world do not remain parallel in the image. This is illustrated vividly in Figure 5.9, where we have associated the three principal directions in the scene with the world X, Y, and Z axes, respectively.

Let us model this mathematically. Suppose an environment contains a collection of parallel lines such as in Figure 5.9. We can write the equations for these lines in parametric form as

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X_i \\ Y_i \\ Z_i \end{bmatrix} + \gamma \begin{bmatrix} U_i \\ V_i \\ W_i \end{bmatrix}$$

The terms in this equation can be interpreted as follows:

- $P_i = (X_i, Y_i, Z_i)$ gives the coordinates for some point on the i^{th} line, specified with respect to the camera frame.
- $D_i = (U_i, V_i, W_i)$ is the direction of lines i, also specified with respect to the camera frame. Since the lines are parallel, they all have the same direction vector D_i.
- γ is a parameter that determines distance along the line from the point P_i.

Even though the lines are parallel in the world, due to the effects of perspective projection, the images of these lines will intersect, except when the lines are parallel to the image plane. This intersection point is known as the **vanishing point**. We can find the vanishing point by examining the case where $\gamma \to \infty$. For a given line, if we let (u_∞, v_∞) denote the image plane coordinates for the vanishing point, we have

$$\begin{aligned} u_\infty &= \lim_{\gamma\to\infty} \lambda \frac{X}{Z} \\ &= \lim_{\gamma\to\infty} \frac{X + \gamma U}{Z + \gamma W} \\ &= \frac{U}{W} \end{aligned}$$

and by similar reasoning,

$$v_\infty = \frac{V}{W}$$

In other words, the projection of a vanishing point only needs the 3D direction (U, V, W), and all 3D lines along the same direction project to the same vanishing point $(u_\infty, v_\infty) = (U/W, V/W)$.

5.2.3 Camera Calibration Parameters

We now discuss how the dimensionless 2D coordinates x and y give rise to **image coordinates** expressed in units of pixels, which is how we typically access images.

Equation 5.1 assumes that the image plane is located at a distance of 1 of whatever the units we use for the 2D point p. If instead it is placed at a distance F, we get

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = F \begin{bmatrix} X/Z \\ Y/Z \end{bmatrix} \tag{5.3}$$

In Equation 5.3 the 2D point and the **focal length** F are always expressed in the same units. If the focal length F is expressed in mm., which is common in the industry, then x' and y' will be in mm. as well, etc. Note that the units of the 3D point P do not matter, which is significant: we will never know, from a single image alone, whether we are looking at a life-size scene of a dollhouse representation of it. Some might say this is why television works :-)

By Equation 5.3 , the image center c always has the coordinates $c = (0, 0)$. But this is not typically how we address pixels in the image! Instead, we use image coordinates (u, v), which have their origin in the upper left. If the origin is there, then the image center must have

non-zero coordinates, and this depends on the resolution of the image sensor. By convention we call these u_0 and v_0.

The resolution of the sensor also determines how the metric coordinates x and y get converted to pixels: we introduce two constants k and l which express the number of pixels per unit of F (e.g., pixels per mm. if f the focal length F is given in mm.) Hence we have, in image coordinates

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} + F \begin{bmatrix} kX/Z \\ lY/Z \end{bmatrix} = \begin{bmatrix} u_0 + \alpha X/Z \\ v_0 + \beta Y/Z \end{bmatrix} \tag{5.4}$$

where $\alpha \doteq kF$ and $\beta \doteq lF$, both having units of pixels. In cameras with square pixels we have $\alpha = \beta = f$, where f is the **focal length expressed in pixels**.

The set of parameters u_0, v_0, f are called **camera calibration parameters**, and these three (image center and focal length) are actually the most important for most cameras. Other parameters include the aspect ratio $a = \alpha/\beta$, which is 1 if the cameras has square pixels (and most of them do) and the skew s, which we will ignore here. What can be important in wide-angle lenses are additional distortion parameters κ that model the radial distortion that gets increasingly worse near the edges of the image.

5.2.4 Homogeneous Coordinates

We now introduce **homogeneous coordinates**, which allow us to do two cool things: (a) we will be able to talk about points at infinity, and (b) we will be able to write the non-linear projection equation as a *linear* matrix-vector multiply. We do this both for 2D and 3D, and on the surface it just means using a third coordinate equal to 1, i.e.

$$p = (x, y) \rightarrow \tilde{p} = (x, y, 1)$$

and

$$P = (X, Y, Z) \rightarrow \tilde{P} = (X, Y, Z, 1)$$

where we use a tilde to indicate a homogeneous point representation. The cost of using homogenous coordinates is that they are not unique. Homogeneous points that only differ by a multiplicative factor are equivalent, i.e., they represent the same point. Hence, you can always write a finite 2D point as $(x, y, 1)$. For example, the homogeneous point $(4, 6, 2)$ is the same as $(2, 3, 1)$ in homogeneous coordinates, and represent the same 2D point $(x, y) = (2, 3)$. Likewise, all of the points below represent the same 3D point

$$P_0 = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 1 \end{bmatrix} \equiv \begin{bmatrix} 4 \\ 4 \\ 8 \\ 4 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -2 \\ -1 \end{bmatrix}$$

where the symbol $\equiv$ denotes **projective equivalence**.

In particular, any *finite* point in 2D or 3D has a non-zero value in the last slot,

$$p_1 = \begin{bmatrix} 2 \\ 4 \\ 2 \end{bmatrix} \equiv \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \qquad P_1 = \begin{bmatrix} 8 \\ -6 \\ 10 \\ 2 \end{bmatrix} \equiv \begin{bmatrix} 4 \\ -3 \\ 5 \\ 1 \end{bmatrix}$$

and are just a new way to represent the points $(1, 2)$ and $(4, -3, 5)$, respectively in 2D and 3D. We should also mention two special cases: the 2D origin of the image plane is $\tilde{c} = (0, 0, 1)$ in homogeneous coordinates, and the 3D origin of the camera coordinate frame (the optical center) is $\tilde{O} = (0, 0, 0, 1)$.

Points at *infinity* (mind blown!) are obtained by setting the last coordinate equal to zero, e.g.

$$p_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad p_3 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

are 2D points at infinity, in the direction of the u and v axis, respectively. Likewise, in 3D these are all points at infinity:

$$P_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad P_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \qquad P_4 = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 0 \end{bmatrix} \qquad P_5 = \begin{bmatrix} 3 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

Above, P_2 and P_3 are "at the end of the X and Z axis, respectively, whereas P_4 and P_5 are at infinity in two different, arbitrary directions.

Mathematically, what we are doing is representing 2D or 3D Euclidean space with 2D or 3D **projective space**, which extends the usual 3D space with a line or plane at infinity. In 2D, the line at infinity extends the real plane to include points at infinity in a manner analogous to adding $+\infty$ and $-\infty$ to the set of real numbers to obtain the extended real number system. In 3D, we actually have a plane at infinity: indeed, the three first coordinates of all points at infinity are 2D homogeneous coordinates!

The recipe to recover non-homogeneous coordinates is to simply divide by the last coordinate, i.e.

$$\tilde{p} = (u, v, w) \rightarrow p = \left(\frac{u}{w}, \frac{v}{w}\right) \tag{5.5}$$

and

$$\tilde{P} = (X, Y, Z, T) \rightarrow P = \left(\frac{X}{T}, \frac{Y}{T}, \frac{Z}{T}\right). \tag{5.6}$$

This is another way to realize that projectively equivalent points, which only differ up to a scale, represent the same point. And, you can also see that dividing by zero yields infinite points.

5.2.5 Pinhole Model in Homogeneous Coordinates

But what about the promised linear projection equation? Well, Equation 5.5 in turn allows us to write perspective projection in a linear way. Indeed, when we write

$$\tilde{p} = \begin{bmatrix} x \\ y \\ t \end{bmatrix} = \begin{bmatrix} 1 & & & 0 \\ & 1 & & 0 \\ & & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ T \end{bmatrix} = \begin{bmatrix} I & 0 \end{bmatrix} \tilde{P}$$

we get the following simple expression for the 2D homogeneous coordinates of the projection of the 3D point:

$$\tilde{p} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix},$$

In other words, if a 3D point has homogeneous coordinates $(X, Y, Z, 1)$, the homogeneous coordinates of the 2D projection are just $\tilde{p} = (X, Y, Z)$. Converting back to non-homogeneous coordinates via Equation 5.5 we get

$$p = \begin{bmatrix} X/Z \\ Y/Z \end{bmatrix}$$

i.e., this is the pinhole camera model from Equation 5.1!

What if we want to express p in image coordinates? Easy, we multiply on the left by another matrix that does the scaling by f (expressed in pixels) and translation to account for the non-zero image center:

$$\tilde{p} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} f & & u_0 \\ & f & v_0 \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & 0 \\ & 1 & & 0 \\ & & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ T \end{bmatrix} = K \begin{bmatrix} I & 0 \end{bmatrix} \tilde{P} \tag{5.7}$$

Above K is the 3×3 **camera calibration matrix**. When you work through it, you will realize that we obtain

$$\tilde{p} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} u_0 Z + fX \\ v_0 Z + fY \\ Z \end{bmatrix} \rightarrow p = \begin{bmatrix} u_0 + f\frac{X}{Z} \\ v_0 + f\frac{Y}{Z} \end{bmatrix}$$

Remarkably, points at infinity with $T = 0$ also have an image which is typically finite: note that T does not figure in the equation above. In other words: all points on the same viewing ray have the same images, even points at infinity. This agrees with what we found in Section 5.2.1.

5.2.6 Projecting Points in the World

Is it not remarkable in Equation 5.7 how we can use *multiplying* with a 3×3 matrix K to implement scaling and translation in 2D? That brings us to the final transformation: what

if 3D points were not expressed in the camera coordinate frame C but instead in some world frame W? To model this, we introduce the 3×3 **camera rotation matrix** R_c^w which has as columns the axes of the camera frame, expressed in world coordinate frames. Likewise, we also introduce the camera translation t_c^w, expressed in world frame. With this, we can express any point $\tilde{P}^c$ expressed in homogeneous camera coordinates in the world frame by multiplying with the 4×4 matrix below:

$$\tilde{P}^w = \begin{bmatrix} R_c^w & t_c^w \\ 0 & 1 \end{bmatrix} \tilde{P}^c.$$

Substituting this into 5.7 we finally obtain

$$\tilde{p} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} = K \begin{bmatrix} I & 0 \end{bmatrix} \begin{bmatrix} R_c^w & t_c^w \\ 0 & 1 \end{bmatrix}^{-1} \tilde{P}^w \tag{5.8}$$

Because we have (trust us on this for now) that

$$\begin{bmatrix} R_c^w & t_c^w \\ 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} R_c^{wT} & -R_c^{wT} t_c^w \\ 0 & 1 \end{bmatrix}$$

we can also write the entire linear camera projection model as

$$\tilde{p} = K \begin{bmatrix} I & 0 \end{bmatrix} \begin{bmatrix} R_c^{wT} & -R_c^{wT} t_c^w \\ 0 & 1 \end{bmatrix} \tilde{P}^w = K R_c^{wT} \begin{bmatrix} I & -t_c^w \end{bmatrix} \tilde{P}^w \tag{5.9}$$

The last formula on the right can be read, from right to left, as: (a) take $\tilde{P}^w$, (b) subtract the camera center t_c^w, (c) use R_c^{wT} to rotate into camera coordinates, and (d) calibrate to pixels using K, to (e) obtain the homogeneous coordinates $\tilde{p}$ of the projected point.

As one final step, we can combine all these operations using a single 3×4 **camera matrix** $\mathcal{P}$:

$$\tilde{p} = K R_c^{wT} \begin{bmatrix} I & -t_c^w \end{bmatrix} \tilde{P}^w = \mathcal{P} \tilde{P}^w \tag{5.10}$$

Worked Example

If you take a picture with your phone while looking in the direction of the world X axis, and assuming the world Z axis is pointing up (very common convention) then the camera rotation matrix will be given by

$$R_c^w = \begin{bmatrix} & & 1 \\ -1 & & \\ & -1 & \end{bmatrix}$$

In addition, suppose we defined the world coordinate frame such that we are standing at location $(X^w, Y^w) = (20, -5)$, and the camera is at eye height, say $Z^w = 1.5$ (all quantities in meter). Then we also have

$$t_c^w = \begin{bmatrix} 20 \\ -5 \\ 1.5 \end{bmatrix}$$

Finally, if we assume a typical smartphone calibration, e.g., an iPhone 5, we will have $(u_0, v_0) \approx (1600, 1200)$ and $f \approx 3000$, all in pixels. we then have

$$\mathcal{P} \doteq K R_c^{wT} \begin{bmatrix} I & -t_c^w \end{bmatrix} = \begin{bmatrix} 3000 & & 1600 \\ & 3000 & 1200 \\ & & 1 \end{bmatrix} \begin{bmatrix} & -1 & \\ & & -1 \\ 1 & & \end{bmatrix} \begin{bmatrix} 1 & & -20 \\ & 1 & 5 \\ & & 1 & -1.5 \end{bmatrix}$$
$$= \begin{bmatrix} 1600 & -3000 & & -47000 \\ 1200 & & -3000 & -19500 \\ 1 & & & -20 \end{bmatrix}$$

Projecting some points is informative. Looking towards the optical axis, we see the point at infinity in the direction of the axis projects to

$$\mathcal{P} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1600 \\ 1200 \\ 1 \end{bmatrix}$$

i.e., the image center, as expected. The two other main vanishing points, in the Y and Z directions beget

$$\mathcal{P} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -3000 \\ 0 \\ 0 \end{bmatrix} \equiv \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \mathcal{P} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -3000 \\ 0 \end{bmatrix} \equiv \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

This also makes sense: they are points at infinity in the image, respectively in the horizontal and vertical directions. Finally, let's project a point on the ground plane, 10 meters in the X direction:

$$\mathcal{P} \begin{bmatrix} 30 \\ -5 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1600 \\ 1650 \\ 1 \end{bmatrix}$$

which projects below the image center in the image (remember v increases downward).

Exercise

Define a large square on the ground plane, centered around the ground plane point above, and project that in the image. Sketch your solution.

Chapter 6

Self-driving Cars

6.1 SLAM with LIDAR Measurements

Motivation

6.1.1 LIDAR Sensors

TBD See slides.

6.1.2 Localization via ICP

TBD See slides.

6.1.3 PoseSLAM

SLAM is **Simultaneous Localization and Mapping**. In the SLAM problem the goal is to localize a robot using the information coming from the robot's sensors. The additional wrinkle in SLAM is that we do *not* know the map a priori, and hence we have to infer the unknown map simultaneously with localization with respect to the evolving map.

PoseSLAM is a variant of SLAM that uses pose constraints as the basic building block, and where we optimize over the unknown vehicles poses. We do not explicitly optimize over a map: that is reconstructed after the fact.

To represent the pose of a vehicle, recall that 2D poses $T \doteq (x, y, \theta)$ form the Special Euclidean group $SE(2)$, and can be represented by 3×3 matrix of the form

$$T = \left[\begin{array}{cc|c} \cos\theta & -\sin\theta & x \\ sin\theta & \cos\theta & y \\ \hline 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{cc} R & t \\ 0 & 1 \end{array}\right] \tag{6.1}$$

with the 2×1 vector t representing the position of the vehicle, and R the 2×2 rotation matrix representing the vehicle's orientation in the plane.

Note that this representation generalizes equally to three dimensions, but of course t will be a three-vector, and R will be a 3×3 rotation matrix representing the 3DOF attitude of the vehicle. The latter can be decomposed into roll, pitch, and yaw, if so desired.

The PoseSLAM problem is then:

> given a set of noisy relative measurements or **pose constraints** $\tilde{T}_{ij}$, recover the optimal set of poses T_i^* that maximizes the posteriori probability, i.e., recover the MAP solution.

In the case of mapping for autonomous driving, these relative measurements can be derived from performing ICP between overlapping scans. We can use GPS and/or IMU measurements to decide which scans overlap, so that we do not have to compare $O(n^2)$ scans. Depending on the situation, we can optimize for 3D or 2D poses, in the way we will discus below. Afterwards, we can reconstruct a detailed map by transforming the local LIDAR scans into the world frame, using the optimized poses T_i^*.

6.1.4 The PoseSLAM Factor Graph

In our factor-graph-based view of the world, a pose constraint is represented as a factor. As before, the factor graph represent the posterior distribution over the unknown pose variables $\mathcal{T} = \{T_1 \dots T_5\}$ given the known measurements:

$$\phi(\mathcal{T}) = \prod_i \phi_i(\mathcal{T}_i). \tag{6.2}$$

The factor graph encodes which factors are connected to which variables, exposing the sparsity pattern of the corresponding estimation problem.

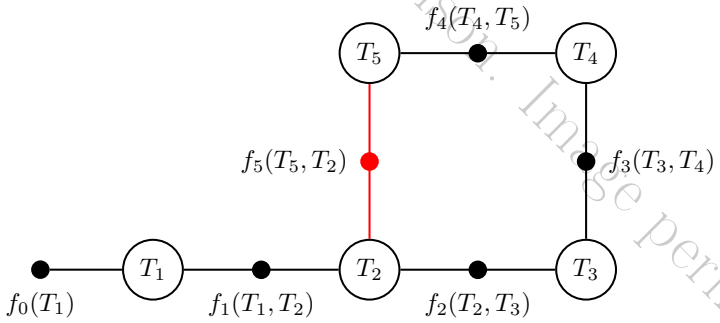


Figure 6.1: PoseSLAM factor graph example.

An example is shown in Figure 6.1. The example represents a vehicle driving around, and taking LIDAR scans at 5 different world poses, represented by T_1 to T_5. The factors f_1 to f_4 are binary factors representing the pose constraints obtained by matching successive LIDAR scans. The factor $f_5(T_5, T_2)$ is a so-called "loop closure" constraint: rather than derived from two successive scans, this one is derived from matching the scan taken at T_5 with the one at T_2. Detecting such loops can be done through a variety of means. The final, unary factor $f_0(T_1)$ is there to "anchor" the solution to the origin: if it is not there, the solution will be undetermined. Another way to anchor the solution is to add unary factors at every time-step, derived from GPS.

Finding the MAP in the case that variables are continuous and measurements are linear combinations of them can be done via least-squares. Above we have discussed MAP inference for discrete variables, and we have discussed probability distributions for continuous variables, but we have never put the two together. In the case of measurements corrupted by zero-mean Gaussian noise, we can recover the MAP solution by minimization. Recall that a multivariate Gaussian density **with mean** μ and **variance** σ^2 is given by

$$\mathcal{N}(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right\}. \tag{6.3}$$

If we focus our attention in PoseSLAM on just the x coordinates, then we predict relative measurements $\tilde{x}_{ij}$ by

$$\tilde{x}_{ij} \approx h(x_i,x_j) = x_j - x_i$$

and each factor in Figure 6.1 could be written as

$$\phi(x_i,x_j) = \frac{1}{\sqrt{2\pi}} \exp\left\{-\frac{1}{2}\left(x_j - x_i - \tilde{x}_{ij}\right)^2\right\}, \tag{6.4}$$

where we assumed $\sigma = 1$ for now. By taking the negative log, maximizing the posterior corresponds to minimizing the following sum of squares, where sum ranges over all (i,j) pairs for which we have a pairwise measurement:

$$\mathcal{X}^* = \arg\min_{\mathcal{X}} \sum_k \frac{1}{2}\left(h(x_i,x_j) - \tilde{x}_{ij}\right)^2 = \arg\min_{\mathcal{X}} \sum_k \frac{1}{2}\left(x_j - x_i - \tilde{x}_{ij}\right)^2.$$

Linear least squares problems like these are easily solved by numerical computing packages like MATLAB or numpy.

Unfortunately, in the PoseSLAM case we cannot use linear least squares, because poses are not simply vectors, and the measurements are not simply linear functions of the poses. Indeed, in PoseSLAM both the prediction $h(T_i,T_j)$ and the measurement $\tilde{T}_{ij}$ are relative poses. The measurement prediction function $h(.)$ is given by

$$h(T_i,T_j) = T_i^{-1}T_j$$

and the measurement error to be minimized is

$$\frac{1}{2}\left\|\log\left(\tilde{T}_{ij}^{-1}T_i^{-1}T_j\right)\right\|^2 \tag{6.5}$$

where $\log : SE(2) \to \mathbb{R}^3$ denotes a map from $SE(2)$ to a three-dimensional local coordinate vector ξ, which will be defined in detail below.

6.1.5 Nonlinear Optimization for PoseSLAM

There are two ways out of the nonlinear quandary. The first is to realize that the only non-linearities stem from the sin and cos terms in the poses, associated with the unknown orientations θ_i. Hence, one solution is to try and solve for the orientations first, and then

solve for the translations using linear least squares, exactly as above. This approach is known as **rotation averaging** followed by linear translation recovery. Unfortunately it is sub-optimal as it does not consider the orientation and translation simultaneously. However, it can serve to provide a (very) good initial estimate for nonlinear optimization, discussed below.

Indeed, we will prefer to take a second route, which is to use **nonlinear optimization**. As discussed, the error expressions (6.5) are *nonlinear*, and we cannot directly optimize over the poses T_i. Instead, we will locally linearize the problem and solve the corresponding linear problem using least-squares, and iterate this until convergence. We do this by, at each iteration, parameterizing a pose T by

$$T \approx \bar{T}\Delta(\xi) \tag{6.6}$$

where ξ are 3D local coordinates $\xi \doteq (\delta x, \delta y, \delta\theta)$ and the incremental pose $\Delta(\xi) \in SE(2)$ is defined as

$$\Delta(\xi) = \left[\begin{array}{cc|c} 1 & -\delta\theta & \delta x \\ \delta\theta & 1 & \delta y \\ \hline 0 & 0 & 1 \end{array}\right]$$

which you can recognize as a small angle approximation of the $SE(2)$ matrix (6.1). In 3D the local coordinates ξ are 6-dimensional, and the small angle approximation is defined as

$$\Delta(\xi) = \left[\begin{array}{ccc|c} 1 & -\delta\theta_z & \delta\theta_y & \delta x \\ \delta\theta_z & 1 & -\delta\theta_x & \delta y \\ -\delta\theta_y & \delta\theta_x & 1 & \delta z \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

With this new notation, we can approximate the nonlinear error (6.5) by a linear approximation:

$$\frac{1}{2}\left\|\log\left(\tilde{T}_{ij}^{-1}T_i^{-1}T_j\right)\right\|^2 \approx \frac{1}{2}\left\|A_i\xi_i + A_j\xi_j - b\right\|^2. \tag{6.7}$$

For $SE(2)$ the matrices A_i and A_j are the 3×3 **or Jacobian matrices** and b is a 3×1 bias term. The above provides a linear approximation of the term within the norm as a function of the incremental local coordinates ξ_i and ξ_j. Deriving the detailed expressions for these Jacobians is beyond the scope of this document, but suffice to say that they exist and not too expensive to compute. In three dimensions, the Jacobian matrices are 6×6 and 16×6, respectively.

The final optimization will—in each iteration—minimize over the local coordinates of all poses by summing over all pose constraints. If we index those constraints by k, we have the following least squares problem:

$$\Xi^* = \arg\min_{\Xi} \sum_k \frac{1}{2}\left\|A_{ki}\xi_i + A_{kj}\xi_j - b_k\right\|^2 \tag{6.8}$$

where $\Xi \doteq \{\xi_i\}$, the set of all incremental pose coordinates.

After solving for the incremental updates Ξ, we update all poses using equation 6.6 and check for convergence. If the error does not decrease significantly we terminate, otherwise we

linearize and solve again, until the error converges. While this is not guaranteed to converge to a global minimum, in practice it does so if there are enough relative measurements and a good initial estimate is available. For example, GPS can provide us with a good initial estimate. However, especially in urban environments GPS can be quite noisy, and it could happen that the map quality suffers by converging to a bad local minimum. Hence, a good quality control process is absolutely necessary in production environments.

For SLAM we typically use specializes packages such as G2O, Ceres, or GTSAM that exploit the sparsity of the factor graphs to dramatically speed up computation. Note that MATLAB and/or numpy can solve sparse least squares problems: the specialized SLAM packages simply provide the translation as well as the calculation of the Jacobian matrices above.

In summary, the algorithm for nonlinear optimization is

- Start with an initial estimate $\mathcal{T}^0$
- Iterate:
  1. Linearize the factors $\frac{1}{2}\left\|\log\left(\tilde{T}_{ij}^{-1}T_i^{-1}T_j\right)\right\|^2 \approx \frac{1}{2}\left\|A_i\xi_i + A_j\xi_j - b\right\|^2$
  2. Solve the least squares problem $\Xi^* = \arg\min_\Xi \sum_k \frac{1}{2}\left\|A_{ki}\xi_i + A_{kj}\xi_j - b_k\right\|^2$
  3. Update $T_i^{t+1} \leftarrow T_j^t \Delta(\xi_i)$
- Until the nonlinear error $J(\mathcal{T}) \doteq \sum_k \frac{1}{2}\left\|\log\left(\tilde{T}_{ij}^{-1}T_i^{-1}T_j\right)\right\|^2$ converges.

6.1.6 Optimization with GTSAM

The GTSAM toolbox (GTSAM stands for "Georgia Tech Smoothing and Mapping") toolbox is a BSD-licensed C++ library based on factor graphs, developed at the Georgia Institute of Technology by myself, many of my students, and collaborators. It provides state of the art solutions to the SLAM and SFM problems, but can also be used to model and solve both simpler and more complex estimation problems. More information is available at `http://gtsam.org`.

GTSAM exploits sparsity to be computationally efficient. Typically measurements only provide information on the relationship between a handful of variables, and hence the resulting factor graph will be sparsely connected. This is exploited by the algorithms implemented in GTSAM to reduce computational complexity. Even when graphs are too dense to be handled efficiently by direct methods, GTSAM provides iterative methods that are quite efficient regardless.

The following C++ code, included in GTSAM as an example, creates the factor graph from Figure 6.1 in code:

```
1  NonlinearFactorGraph graph;
2  auto priorNoise = noiseModel::Diagonal::Sigmas((Vector(3)<< 0.3, 0.3, 0.1));
3  graph.add(PriorFactor<Pose2>(1, Pose2(0,0,0), priorNoise));
4
5  // Add odometry factors
6  auto model = noiseModel::Diagonal::Sigmas((Vector(3)<< 0.2, 0.2, 0.1));
7  graph.add(BetweenFactor<Pose2>(1, 2, Pose2(2, 0, 0     ), model));
8  graph.add(BetweenFactor<Pose2>(2, 3, Pose2(2, 0, M_PI_2), model));
9  graph.add(BetweenFactor<Pose2>(3, 4, Pose2(2, 0, M_PI_2), model));
10 graph.add(BetweenFactor<Pose2>(4, 5, Pose2(2, 0, M_PI_2), model));
11
12 // Add pose constraint
13 graph.add(BetweenFactor<Pose2>(5, 2, Pose2(2, 0, M_PI_2), model));
```

Listing 6.1: Building a graph in C++

Lines 1-4 create a nonlinear factor graph and add the unary factor $f_0(T_1)$. As the vehicle travels through the world, it creates binary factors $f_t(T_t, T_{t+1})$ corresponding to odometry, added to the graph in lines 6-12 (Note that M_PI_2 refers to pi/2). But line 15 models a different event: a **loop closure**. For example, the vehicle might recognize the same location using vision or a laser range finder, and calculate the geometric pose constraint to when it first visited this location. This is illustrated for poses T_5 and T_2, and generates the (red) loop closing factor $f_5(T_5, T_2)$.

6.1.7 Using the python Interface

GTSAM It also provides both a MATLAB and a python interface which allows for rapid prototype development, visualization, and user interaction. The python library can be imported directly into a Google colab via "import gtsam". A large subset of the GTSAM functionality can be accessed through wrapped classes from within python . The following code excerpt is the python equivalent of the C++ code in Listing 6.1:

```
1  graph = gtsam.NonlinearFactorGraph()
2  priorNoise = gtsam.noiseModel_Diagonal.Sigmas(vector3(0.3, 0.3, 0.1))
3  graph.add(gtsam.PriorFactorPose2(1, gtsam.Pose2(0, 0, 0), priorNoise))
4
5  # Create odometry (Between) factors between consecutive poses
6  model = gtsam.noiseModel_Diagonal.Sigmas(vector3(0.2, 0.2, 0.1))
7  graph.add(gtsam.BetweenFactorPose2(1, 2, gtsam.Pose2(2, 0, 0), model))
8  graph.add(gtsam.BetweenFactorPose2(2, 3, gtsam.Pose2(2, 0, pi/2), model))
9  graph.add(gtsam.BetweenFactorPose2(3, 4, gtsam.Pose2(2, 0, pi/2), model))
10 graph.add(gtsam.BetweenFactorPose2(4, 5, gtsam.Pose2(2, 0, pi/2), model))
11
12 # Add the loop closure constraint
13 graph.add(gtsam.BetweenFactorPose2(5, 2, gtsam.Pose2(2, 0, pi/2), model))
```

Listing 6.2: Building a graph in python

Note that the code is almost identical, although there are a few syntax and naming differences:

- Objects are created by calling a constructor instead of allocating them on the heap.
- ***Vector*** and ***Matrix*** classes in C++ are just numpy arrays in python.
- As templated classes do not exist in python, these have been hardcoded in the wrapper, e.g., ***PriorFactorPose2*** corresponds to the C++ class ***PriorFactor<Pose2>***, etc.

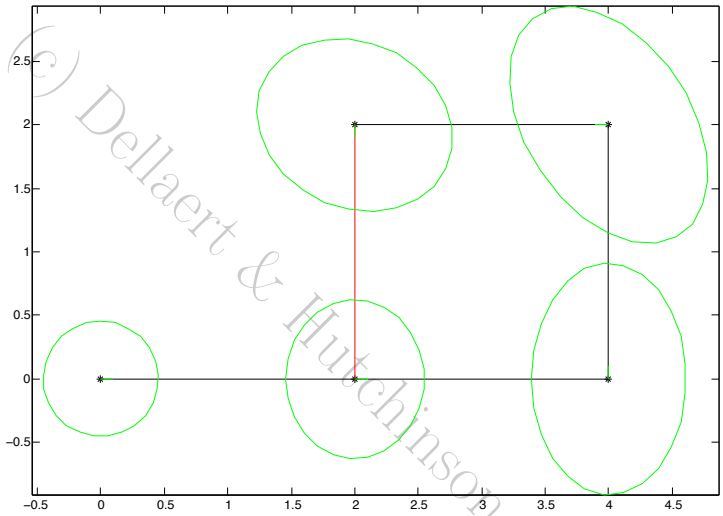


Figure 6.2: The result of running optimize on the factor graph in Figure 6.1.

We can optimize this factor graph, by creating an initial estimate of type ***Values***, and creating and running an optimizer. This is illustrated in the listing below:

```
# Create the initial estimate
initial_estimate = gtsam.Values()
initial_estimate.insert(1, gtsam.Pose2(0.5, 0.0, 0.2))
initial_estimate.insert(2, gtsam.Pose2(2.3, 0.1, -0.2))
initial_estimate.insert(3, gtsam.Pose2(4.1, 0.1, pi/2))
initial_estimate.insert(4, gtsam.Pose2(4.0, 2.0, pi))
initial_estimate.insert(5, gtsam.Pose2(2.1, 2.1, -pi/2))

# Optimize the initial values using a Gauss-Newton nonlinear optimizer
optimizer = gtsam.GaussNewtonOptimizer(graph, initial_estimate)
result = optimizer.optimize()
print("Final Result:\n{}".format(result))
```

Listing 6.3: Optimizing

The result is shown graphically in Figure 6.2, along with covariance ellipses shown in green. These covariance ellipses in 2D indicate the marginal over position, over all possible orientations, and show the area which contain 68.26% of the probability mass (in 1D this would correspond to one standard deviation). The graph shows in a clear manner that the uncertainty on pose T_5 is now much less than if there would be only odometry measurements. The pose with the highest uncertainty, T_4, is the one furthest away from the unary constraint $f_0(T_1)$, which is the only factor tying the graph to a global coordinate frame.

The figure above was created using an interface that allows you to use GTSAM from within MATLAB, which provides some excellent visualization tools. Similar matplotlib-based visualization tools are available in python.

Summary

We briefly summarize what we learned in this section:

1. LIDAR is a key sensor for autonomous driving
2. Localization can be done with LIDAR, or image-based
3. PoseSLAM: a SLAM variant using ICP pose constraints
4. The PoseSLAM factor graph graphically shows the constraints
5. MAP/MAP solution can be done via nonlinear optimization
6. GTSAM is an easy way to optimize over poses in C++/MATLAB/python