CHAPTER 26

ROBOTICS

In which agents are endowed with sensors and physical effectors with which to move about and make mischief in the real world.

26.1 Robots

Robot
Effector

Robots are physical agents that perform tasks by manipulating the physical world. To do so, they are equipped with **effectors** such as legs, wheels, joints, and grippers. Effectors are designed to assert physical forces on the environment. When they do this, a few things may happen: the robot's state might change (e.g., a car spins its wheels and makes progress on the road as a result), the state of the environment might change (e.g., a robot arm uses its gripper to push a mug across the counter), and even the state of the people around the robot might change (e.g., an exoskeleton moves and that changes the configuration of a person's leg; or a mobile robot makes progress toward the elevator doors, and a person notices and is nice enough to move out of the way, or even push the button for the robot).

Sensor

Robots are also equipped with **sensors**, which enable them to perceive their environment. Present-day robotics employs a diverse set of sensors, including cameras, radars, lasers, and microphones to measure the state of the environment and of the people around it; and gyroscopes, strain and torque sensors, and accelerometers to measure the robot's own state.

Maximizing expected utility for a robot means choosing how to actuate its effectors to assert the *right* physical forces—the ones that will lead to changes in state that accumulate as much expected reward as possible. Ultimately, robots are trying to accomplish some task in the physical world.

Robots operate in environments that are partially observable and stochastic: cameras cannot see around corners, and gears can slip. Moreover, the people acting in that same environment are unpredictable, so the robot needs to make predictions about them.

Robots usually model their environment with a continuous state space (the robot's position has continuous coordinates) and a continuous action space (the amount of current a robot sends to its motor is also measured in continuous units). Some robots operate in high-dimensional spaces: cars need to know the position, orientation, and velocity of themselves and the nearby agents; robot arms have six or seven joints that can each be independently moved; and robots that mimic the human body have hundreds of joints.

Robotic learning is constrained because the real world stubbornly refuses to operate faster than real time. In a simulated environment, it is possible to use learning algorithms (such as the Q-learning algorithm described in Chapter 23) to learn in a few hours from millions of trials. In a real environment, it might take years to run these trials, and the robot cannot risk (and thus cannot learn from) a trial that might cause harm. Thus, transferring what has been

(a)

(b)

Figure 26.1 (a) An industrial robotic arm with a custom end-effector. Image credit: Macor/123RF. (b) A Kinova® JACO® Assistive Robot arm mounted on a wheelchair. Kinova and JACO are trademarks of Kinova, Inc.

learned in simulation to a real robot in the real world—the **sim-to-real** problem—is an active area of research. Practical robotic systems need to embody prior knowledge about the robot, the physical environment, and the tasks to be performed so that the robot can learn quickly and perform safely.

Robotics brings together many of the concepts we have seen in this book, including probabilistic state estimation, perception, planning, unsupervised learning, reinforcement learning, and game theory. For some of these concepts robotics serves as a challenging example application. For other concepts this chapter breaks new ground, for instance in introducing the continuous version of techniques that we previously saw only in the discrete case.

26.2 Robot Hardware

So far in this book, we have taken the agent architecture—sensors, effectors, and processors—as given, and have concentrated on the agent program. But the success of real robots depends at least as much on the design of sensors and effectors that are appropriate for the task.

26.2.1 Types of robots from the hardware perspective

When you think of a robot, you might imagine something with a head and two arms, moving around on legs or wheels. Such **anthropomorphic robots** have been popularized in fiction such as the movie *The Terminator* and the cartoon *The Jetsons*. But real robots come in many shapes and sizes.

Anthropomorphic robot

Manipulators are just robot arms. They do not necessarily have to be attached to a robot body; they might simply be bolted onto a table or a floor, as they are in factories (Figure 26.1 (a)). Some have a large payload, like those assembling cars, while others, like wheelchair-mountable arms that assist people with motor impairments (Figure 26.1(b)), can carry less but are safer in human environments.

Manipulator

(a) (b)

Figure 26.2 (a) NASA's Curiosity rover taking a selfie on Mars. Image courtesy of NASA. (b) A Skydio drone accompanying a family on a bike ride. Image courtesy of Skydio.

Mobile robot
Quadcopter drone
UAV
AUV
Autonomous car
Rover
Legged robot

Mobile robots are those that use wheels, legs, or rotors to move about the environment. **Quadcopter drones** are a type of **unmanned aerial vehicle (UAV)**; **autonomous underwater vehicles (AUVs)** roam the oceans. But many mobile robots stay indoors and move on wheels, like a vacuum cleaner or a towel delivery robot in a hotel. Their outdoor counterparts include **autonomous cars** or **rovers** that explore new terrain, even on the surface of Mars (Figure 26.2). Finally, **legged robots** are meant to traverse rough terrain that is inaccessible with wheels. The downside is that controlling legs to do the right thing is more challenging than spinning wheels.

Other kinds of robots include prostheses, exoskeletons, robots with wings, swarms, and intelligent environments in which the robot is the entire room.

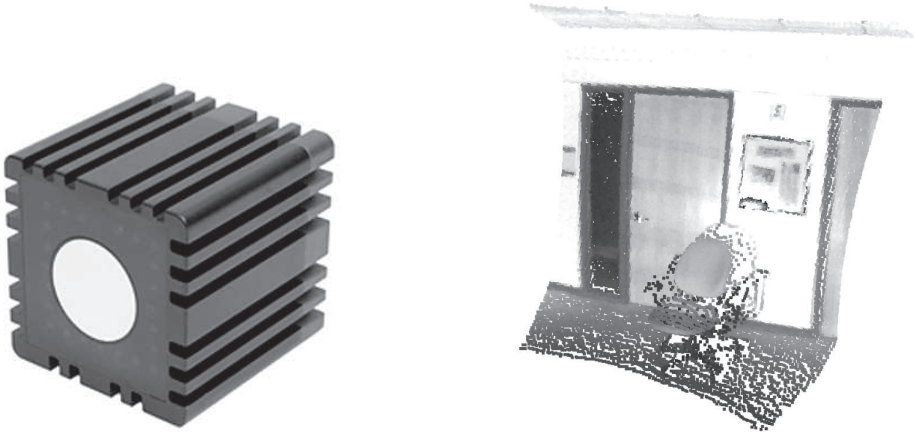
26.2.2 Sensing the world

Passive sensor
Active sensor

Sensors are the perceptual interface between robot and environment. **Passive sensors**, such as cameras, are true observers of the environment: they capture signals that are generated by other sources in the environment. **Active sensors**, such as sonar, send energy into the environment. They rely on the fact that this energy is reflected back to the sensor. Active sensors tend to provide more information than passive sensors, but at the expense of increased power consumption and with a danger of interference when multiple active sensors are used at the same time. We also distinguish whether a sensor is directed at sensing the environment, the robot's location, or the robot's internal configuration.

Range finder
Sonar
Stereo vision

Range finders are sensors that measure the distance to nearby objects. **Sonar** sensors are active range finders that emit directional sound waves, which are reflected by objects, with some of the sound making it back to the sensor. The time and intensity of the returning signal indicates the distance to nearby objects. Sonar is the technology of choice for autonomous underwater vehicles, and was popular in the early days of indoor robotics. **Stereo vision** (see Section 27.6) relies on multiple cameras to image the environment from slightly



(a) (b)

Figure 26.3 (a) Time-of-flight camera; image courtesy of Mesa Imaging GmbH. (b) 3D range image obtained with this camera. The range image makes it possible to detect obstacles and objects in a robot's vicinity. Image courtesy of Willow Garage, LLC.

different viewpoints, analyzing the resulting parallax in these images to compute the range of surrounding objects.

For mobile ground robots, sonar and stereo vision are now rarely used, because they are not reliably accurate. The Kinect is a popular low-cost sensor that combines a camera and a **structured light** projector, which projects a pattern of grid lines onto a scene. The camera sees how the grid lines bend, giving the robot information about the shape of the objects in the scene. If desired, the projection can be infrared light, so as not to interfere with other sensors (such as human eyes).

Structured light

Most ground robots are now equipped with active optical range finders. Just like sonar sensors, optical range sensors emit active signals (light) and measure the time until a reflection of this signal arrives back at the sensor. Figure 26.3(a) shows a **time-of-flight camera**. This camera acquires range images like the one shown in Figure 26.3(b) at up to 60 frames per second. Autonomous cars often use **scanning lidars** (short for *light detection and ranging*)—active sensors that emit laser beams and sense the reflected beam, giving range measurements accurate to within a centimeter at a range of 100 meters. They use complex arrangements of mirrors or rotating elements to sweep the beam across the environment and build a map. Scanning lidars tend to work better than time-of-flight cameras at longer ranges, and tend to perform better in bright daylight.

Time-of-flight camera

Scanning lidar

Radar is often the range finding sensor of choice for air vehicles (autonomous or not). Radar sensors can measure distances up to kilometers, and have an advantage over optical sensors in that they can see through fog. On the close end of range sensing are **tactile sensors** such as whiskers, bump panels, and touch-sensitive skin. These sensors measure range based on physical contact, and can be deployed only for sensing objects very close to the robot.

Radar

Tactile sensor

A second important class is **location sensors**. Most location sensors use range sensing as a primary component to determine location. Outdoors, the **Global Positioning System** (GPS) is the most common solution to the localization problem. GPS measures the distance to

Location sensor

Global Positioning System

satellites that emit pulsed signals. At present, there are 31 operational GPS satellites in orbit, and 24 GLONASS satellites, the Russian counterpart. GPS receivers can recover the distance to a satellite by analyzing phase shifts. By triangulating signals from multiple satellites, GPS receivers can determine their absolute location on Earth to within a few meters. **Differential GPS** involves a second ground receiver with known location, providing millimeter accuracy under ideal conditions.

Differential GPS

Unfortunately, GPS does not work indoors or underwater. Indoors, localization is often achieved by attaching beacons in the environment at known locations. Many indoor environments are full of wireless base stations, which can help robots localize through the analysis of the wireless signal. Underwater, active sonar beacons can provide a sense of location, using sound to inform AUVs of their relative distances to those beacons.

The third important class is **proprioceptive sensors**, which inform the robot of its own motion. To measure the exact configuration of a robotic joint, motors are often equipped with **shaft decoders** that accurately measure the angular motion of a shaft. On robot arms, shaft decoders help track the position of joints. On mobile robots, shaft decoders report wheel revolutions for **odometry**—the measurement of distance traveled. Unfortunately, wheels tend to drift and slip, so odometry is accurate only over short distances. External forces, such as wind and ocean currents, increase positional uncertainty. **Inertial sensors**, such as gyroscopes, reduce uncertainty by relying on the resistance of mass to the change of velocity.

Proprioceptive sensor

Shaft decoder

Odometry

Inertial sensor

Other important aspects of robot state are measured by **force sensors** and **torque sensors**. These are indispensable when robots handle fragile objects or objects whose exact size and shape are unknown. Imagine a one-ton robotic manipulator screwing in a light bulb. It would be all too easy to apply too much force and break the bulb. Force sensors allow the robot to sense how hard it is gripping the bulb, and torque sensors allow it to sense how hard it is turning. High-quality sensors can measure forces in all three translational and three rotational directions. They do this at a frequency of several hundred times a second so that a robot can quickly detect unexpected forces and correct its actions before it breaks a light bulb. However, it can be a challenge to outfit a robot with high-end sensors and the computational power to monitor them.

Force sensor

Torque sensor

26.2.3 Producing motion

The mechanism that initiates the motion of an effector is called an **actuator**; examples include transmissions, gears, cables, and linkages. The most common type of actuator is the **electric actuator**, which uses electricity to spin up a motor. These are predominantly used in systems that need rotational motion, like joints on a robot arm. **Hydraulic actuators** use pressurized hydraulic fluid (like oil or water) and **pneumatic actuators** use compressed air to generate mechanical motion.

Actuator

Hydraulic actuator

Pneumatic actuator

Actuators are often used to move joints, which connect rigid bodies (links). Arms and legs have such joints. In **revolute joints**, one link rotates with respect to the other. In **prismatic joints**, one link slides along the other. Both of these are single-axis joints (one axis of motion). Other kinds of joints include spherical, cylindrical, and planar joints, which are multi-axis joints.

Revolute joint

Prismatic joint

To interact with objects in the environment, robots use grippers. The most basic type of gripper is the **parallel jaw gripper**, with two fingers and a single actuator that moves the fingers together to grasp objects. This effector is both loved and hated for its simplicity.

Parallel jaw gripper

Three-fingered grippers offer slightly more flexibility while maintaining simplicity. At the other end of the spectrum are humanoid (anthropomorphic) hands. For instance, the Shadow Dexterous Hand has a total of 20 actuators. This offers a lot more flexibility for complex manipulation, including in-hand manipulator maneuvers (think of picking up your cell phone and rotating it in-hand to orient it right-side up), but this flexibility comes at a price—learning to control these complex grippers is more challenging.

26.3 What kind of problem is robotics solving?

Now that we know what the robot hardware might be, we're ready to consider the agent software that drives the hardware to achieve our goals. We first need to decide the computational framework for this agent. We have talked about search in deterministic environments, MDPs for stochastic but fully observable environments, POMDPs for partial observability, and games for situations in which the agent is not acting in isolation. Given a computational framework, we need to instantiate its ingredients: reward or utility functions, states, actions, observation spaces, etc.

We have already noted that robotics problems are nondeterministic, partially observable, and multiagent. Using the game-theoretic notions from Chapter 17, we can see that sometimes the agents are cooperative and sometimes they are competitive. In a narrow corridor where only one agent can go first, a robot and a person collaborate because they both want to make sure they don't bump into each other. But in some cases they might compete a bit to reach their destination quickly. If the robot is too polite and always makes room, it might get stuck in crowded situations and never reach its goal.

Therefore, when robots act in isolation and know their environment, the problem they are solving can be formulated as an MDP; when they are missing information it becomes a POMDP; and when they act around people it can often be formulated as a game.

What is the robot's reward function in this formulation? Usually the robot is acting in service of a human—for example delivering a meal to a hospital patient for the patient's reward, not its own. For most robotics settings, even though robot designers might try to specify a good enough proxy reward function, the true reward function lies with the user whom the robot is supposed to help. The robot will either need to decipher the user's desires, or rely on an engineer to specify an approximation of the user's desires.

As for the robot's action, state, and observation spaces, the most general form is that observations are raw sensor feeds (e.g., the images coming in from cameras, or the laser hits coming in from lidar); actions are raw electric currents being sent to the motors; and state is what the robot needs to know for its decision making. This means there is a huge gap between the low-level percepts and motor controls, and the high-level plans the robot needs to make. To bridge the gap, roboticists decouple aspects of the problem to simplify it.

For instance, we know that when we solve POMDPs properly, perception and action interact: perception informs which actions make sense, but action also informs perception, with agents taking actions to gather information when that information has value in later time steps. However, robots often separate perception from action, consuming the outputs of perception and pretending they will not get any more information in the future. Further, hierarchical planning is called for, because a high-level goal like "get to the cafeteria" is far removed from a motor command like "rotate the main axle 1°,"

Task planning

In robotics we often use a three-level hierarchy. The **task planning** level decides a plan or policy for high-level actions, sometimes called action primitives or subgoals: move to the door, open it, go to the elevator, press the button, etc. Then **motion planning** is in charge of finding a path that gets the robot from one point to another, achieving each subgoal. Finally, **control** is used to achieve the planned motion using the robot's actuators. Since the task planning level is typically defined over discrete states and actions, in this chapter we will focus primarily on motion planning and control.

Control

Preference learning

People prediction

Separately, **preference learning** is in charge of estimating an end user's objective, and **people prediction** is used to forecast the actions of other people in the robot's environment. All these combine to determine the robot's behavior.

Whenever we split a problem into separate pieces we reduce complexity, but we give up opportunities for the pieces to help each other. Action can help improve perception, and also determine what kind of perception is useful. Similarly, decisions at the motion level might not be the best when accounting for how that motion will be tracked; or decisions at the task level might render the task plan uninstantiatable at the motion level. So, with progress in these separate areas comes the push to reintegrate them: to do motion planning and control together, to do task and motion planning together, and to reintegrate perception, prediction, and action—closing the feedback loop. Robotics today is about continuing progress in each area while also building on this progress to achieve better integration.

26.4 Robotic Perception

Perception is the process by which robots map sensor measurements into internal representations of the environment. Much of it uses the computer vision techniques from the previous chapter. But perception for robotics must deal with additional sensors like lidar and tactile sensors.

Perception is difficult because sensors are noisy and the environment is partially observable, unpredictable, and often dynamic. In other words, robots have all the problems of **state estimation** (or **filtering**) that we discussed in Section 14.2. As a rule of thumb, good internal representations for robots have three properties:

1. They contain enough information for the robot to make good decisions.
2. They are structured so that they can be updated efficiently.
3. They are natural in the sense that internal variables correspond to natural state variables in the physical world.

In Chapter 14, we saw that Kalman filters, HMMs, and dynamic Bayes nets can represent the transition and sensor models of a partially observable environment, and we described both exact and approximate algorithms for updating the **belief state**—the posterior probability distribution over the environment state variables. Several dynamic Bayes net models for this process were shown in Chapter 14. For robotics problems, we include the robot's own past actions as observed variables in the model. Figure 26.4 shows the notation used in this chapter: $\mathbf{X}_t$ is the state of the environment (including the robot) at time t, $\mathbf{Z}_t$ is the observation received at time t, and A_t is the action taken after the observation is received.

We would like to compute the new belief state, $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{z}_{1:t+1}, a_{1:t})$, from the current belief state, $\mathbf{P}(\mathbf{X}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})$, and the new observation $\mathbf{z}_{t+1}$. We did this in Section 14.2,

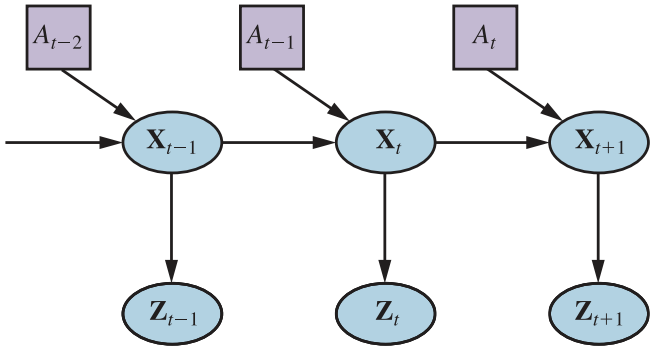


Figure 26.4 Robot perception can be viewed as temporal inference from sequences of actions and measurements, as illustrated by this dynamic decision network.

but there are two differences here: we condition on the actions as well as the observations, and we deal with *continuous* rather than *discrete* variables. Thus, we modify the recursive filtering equation (14.5 on page 485) to use integration rather than summation:

$$\begin{aligned}&\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{z}_{1:t+1}, a_{1:t}) \\ &= \alpha \mathbf{P}(\mathbf{z}_{t+1} \mid \mathbf{X}_{t+1}) \int \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t, a_t)\, P(\mathbf{x}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})\, d\mathbf{x}_t\,. \end{aligned} \tag{26.1}$$

This equation states that the posterior over the state variables $\mathbf{X}$ at time $t+1$ is calculated recursively from the corresponding estimate one time step earlier. This calculation involves the previous action a_t and the current sensor measurement $\mathbf{z}_{t+1}$. For example, if our goal is to develop a soccer-playing robot, $\mathbf{X}_{t+1}$ might include the location of the soccer ball relative to the robot. The posterior $\mathbf{P}(\mathbf{X}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})$ is a probability distribution over all states that captures what we know from past sensor measurements and controls. Equation (26.1) tells us how to recursively estimate this location, by incrementally folding in sensor measurements (e.g., camera images) and robot motion commands. The probability $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t, a_t)$ is called the **transition model** or **motion model**, and $\mathbf{P}(\mathbf{z}_{t+1} \mid \mathbf{X}_{t+1})$ is the **sensor model**.

Motion model

26.4.1 Localization and mapping

Localization is the problem of finding out where things are—including the robot itself. To keep things simple, let us consider a mobile robot that moves slowly in a flat two-dimensional world. Let us also assume the robot is given an exact map of the environment. (An example of such a map appears in Figure 26.7.) The pose of such a mobile robot is defined by its two Cartesian coordinates with values x and y and its heading with value θ, as illustrated in Figure 26.5(a). If we arrange those three values in a vector, then any particular state is given by $\mathbf{X}_t = (x_t, y_t, \theta_t)^\top$. So far so good.

Localization

In the kinematic approximation, each action consists of the "instantaneous" specification of two velocities—a translational velocity v_t and a rotational velocity ω_t. For small time intervals Δt, a crude deterministic model of the motion of such robots is given by

$$\hat{\mathbf{X}}_{t+1} = f(\mathbf{X}_t, \underbrace{v_t, \omega_t}_{a_t}) = \mathbf{X}_t + \begin{pmatrix} v_t \Delta t \cos\theta_t \\ v_t \Delta t \sin\theta_t \\ \omega_t \Delta t \end{pmatrix}.$$

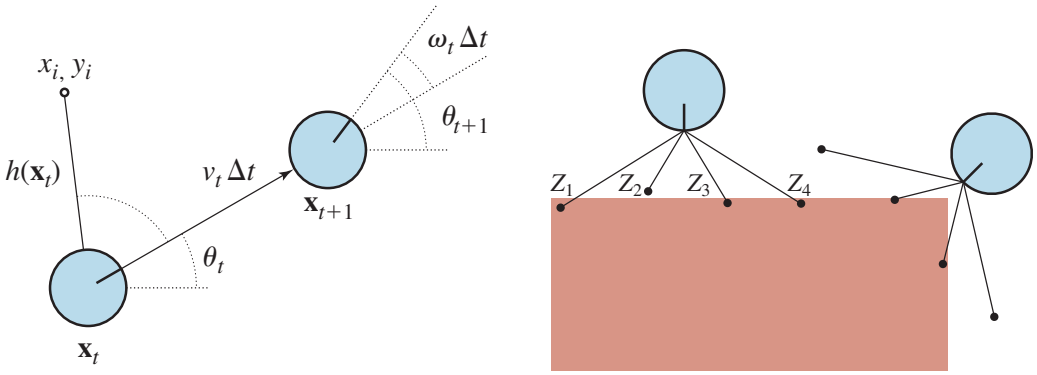


Figure 26.5 (a) A simplified kinematic model of a mobile robot. The robot is shown as a circle with an interior radius line marking the forward direction. The state $\mathbf{x}_t$ consists of the (x_t, y_t) position (shown implicitly) and the orientation θ_t. The new state $\mathbf{x}_{t+1}$ is obtained by an update in position of $v_t\Delta_t$ and in orientation of $\omega_t\Delta_t$. Also shown is a landmark at (x_i, y_i) observed at time t. (b) The range-scan sensor model. Two possible robot poses are shown for a given range scan (z_1, z_2, z_3, z_4). It is much more likely that the pose on the left generated the range scan than the pose on the right.

The notation $\hat{\mathbf{X}}$ refers to a deterministic state prediction. Of course, physical robots are somewhat unpredictable. This is commonly modeled by a Gaussian distribution with mean $f(\mathbf{X}_t, v_t, \omega_t)$ and covariance Σ_x. (See Appendix A for a mathematical definition.)

$$\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{X}_t, v_t, \omega_t) \;=\; \mathcal{N}(\hat{\mathbf{X}}_{t+1}, \Sigma_x)\,.$$

This probability distribution is the robot's motion model. It models the effects of the motion a_t on the location of the robot.

Next, we need a sensor model. We will consider two kinds of sensor models. The first assumes that the sensors detect *stable*, *recognizable* features of the environment called **landmarks**. For each landmark, the range and bearing are reported. Suppose the robot's state is $\mathbf{x}_t = (x_t, y_t, \theta_t)^\top$ and it senses a landmark whose location is known to be $(x_i, y_i)^\top$. Without noise, a prediction of the range and bearing can be calculated by simple geometry (see Figure 26.5(a)):

Landmark

$$\hat{\mathbf{z}}_t = h(\mathbf{x}_t) = \begin{pmatrix} \sqrt{(x_t - x_i)^2 + (y_t - y_i)^2} \\ \arctan\frac{y_i - y_t}{x_i - x_t} - \theta_t \end{pmatrix}.$$

Again, noise distorts our measurements. To keep things simple, assume Gaussian noise with covariance Σ_z, giving us the sensor model

$$P(\mathbf{z}_t \mid \mathbf{x}_t) = \mathcal{N}(\hat{\mathbf{z}}_t, \Sigma_z)\,.$$

Sensor array

A somewhat different sensor model is used for a **sensor array** of range sensors, each of which has a fixed bearing relative to the robot. Such sensors produce a vector of range values $\mathbf{z}_t = (z_1, \ldots, z_M)^\top$.

Given a pose $\mathbf{x}_t$, let $\hat{z}_j$ be the computed range along the jth beam direction from $\mathbf{x}_t$ to the nearest obstacle. As before, this will be corrupted by Gaussian noise. Typically, we assume

```
function MONTE-CARLO-LOCALIZATION(a, z, N, P(X'|X, v, ω), P(z|z*), map)
  returns a set of samples, S, for the next time step
  inputs: a, robot velocities v and ω
          z, a vector of M range scan data points
          P(X'|X, v, ω), motion model
          P(z|z*), a range sensor noise model
          map, a 2D map of the environment
  persistent: S, a vector of N samples
  local variables: W, a vector of N weights
                   S', a temporary vector of N samples

  if S is empty then
    for i = 1 to N do        // initialization phase
      S[i] ← sample from P(X_0)
  for i = 1 to N do          // update cycle
    S'[i] ← sample from P(X'|X = S[i], v, ω)
    W[i] ← 1
    for j = 1 to M do
      z* ← RAYCAST(j, X = S'[i], map)
      W[i] ← W[i] · P(z_j | z*)
    S ← WEIGHTED-SAMPLE-WITH-REPLACEMENT(N, S', W)
  return S
```

Figure 26.6 A Monte Carlo localization algorithm using a range-scan sensor model with independent noise.

that the errors for the different beam directions are independent and identically distributed, so we have

$$P(\mathbf{z}_t \mid \mathbf{x}_t) = \alpha \prod_{j=1}^{M} e^{-(z_j - \hat{z}_j)/2\sigma^2}.$$

Figure 26.5(b) shows an example of a four-beam range scan and two possible robot poses, one of which is reasonably likely to have produced the observed scan and one of which is not. Comparing the range-scan model to the landmark model, we see that the range-scan model has the advantage that there is no need to *identify* a landmark before the range scan can be interpreted; indeed, in Figure 26.5(b), the robot faces a featureless wall. On the other hand, if there *are* visible, identifiable landmarks, they may provide instant localization.

Section 14.4 described the Kalman filter, which represents the belief state as a single multivariate Gaussian, and the particle filter, which represents the belief state by a collection of particles that correspond to states. Most modern localization algorithms use one of these two representations of the robot's belief $\mathbf{P}(\mathbf{X}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})$.

Monte Carlo localization

Localization using particle filtering is called **Monte Carlo localization**, or MCL. The MCL algorithm is an instance of the particle-filtering algorithm of Figure 14.17 (page 510). All we need to do is supply the appropriate motion model and sensor model. Figure 26.6 shows one version using the range-scan sensor model. The operation of the algorithm is illustrated in Figure 26.7 as the robot finds out where it is inside an office building. In the first image, the particles are uniformly distributed based on the prior, indicating global uncertainty

about the robot's position. In the second image, the first set of measurements arrives and the particles form clusters in the areas of high posterior belief. In the third, enough measurements are available to push all the particles to a single location.

The Kalman filter is the other major way to localize. A Kalman filter represents the posterior $\mathbf{P}(\mathbf{X}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})$ by a Gaussian. The mean of this Gaussian will be denoted μ_t and its covariance Σ_t. The main problem with Gaussian beliefs is that they are closed only under linear motion models f and linear measurement models h. For nonlinear f or h, the result of updating a filter is in general not Gaussian. Thus, localization algorithms using the Kalman filter **linearize** the motion and sensor models. Linearization is a local approximation of a nonlinear function by a linear function. Figure 26.8 illustrates the concept of linearization for a (one-dimensional) robot motion model. On the left, it depicts a nonlinear motion model $f(\mathbf{x}_t, a_t)$ (the control a_t is omitted in this graph since it plays no role in the linearization). On the right, this function is approximated by a linear function $\tilde{f}(\mathbf{x}_t, a_t)$. This linear function is tangent to f at the point μ_t, the mean of our state estimate at time t. Such a linearization is called first degree **Taylor expansion**. A Kalman filter that linearizes f and h via Taylor expansion is called an **extended Kalman filter** (or EKF). Figure 26.9 shows a sequence of estimates of a robot running an extended Kalman filter localization algorithm.

Linearization

Taylor expansion

As the robot moves, the uncertainty in its location estimate increases, as shown by the error ellipses. Its error decreases as it senses the range and bearing to a landmark with known location and increases again as the robot loses sight of the landmark. EKF algorithms work well if landmarks are easily identified. Otherwise, the posterior distribution may be multimodal, as in Figure 26.7(b). The problem of needing to know the identity of landmarks is an instance of the **data association** problem discussed in Figure 18.3.

In some situations, no map of the environment is available. Then the robot will have to acquire a map. This is a bit of a chicken-and-egg problem: the navigating robot will have to determine its location relative to a map it doesn't quite know, at the same time building this map while it doesn't quite know its actual location. This problem is important for many robot applications, and it has been studied extensively under the name **simultaneous localization and mapping**, abbreviated as **SLAM**.

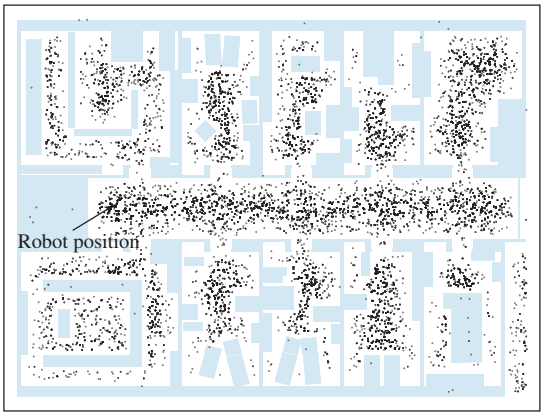
Simultaneous localization and mapping

SLAM problems are solved using many different probabilistic techniques, including the extended Kalman filter discussed above. Using the EKF is straightforward: just augment the state vector to include the locations of the landmarks in the environment. Luckily, the EKF update scales quadratically, so for small maps (e.g., a few hundred landmarks) the computation is quite feasible. Richer maps are often obtained using graph relaxation methods, similar to the Bayesian network inference techniques discussed in Chapter 13. Expectation–maximization is also used for SLAM.

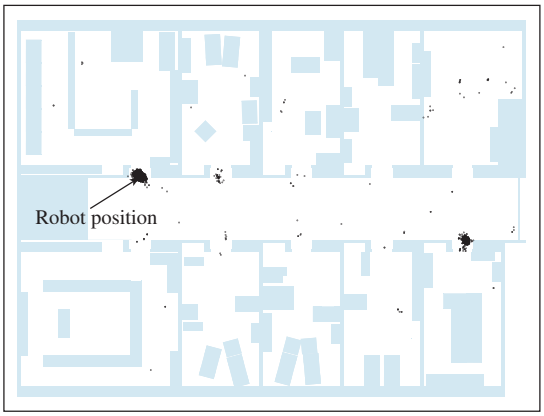
26.4.2 Other types of perception

Not all of robot perception is about localization or mapping. Robots also perceive temperature, odors, sound, and so on. Many of these quantities can be estimated using variants of dynamic Bayes networks. All that is required for such estimators are conditional probability distributions that characterize the evolution of state variables over time, and sensor models that describe the relation of measurements to state variables.

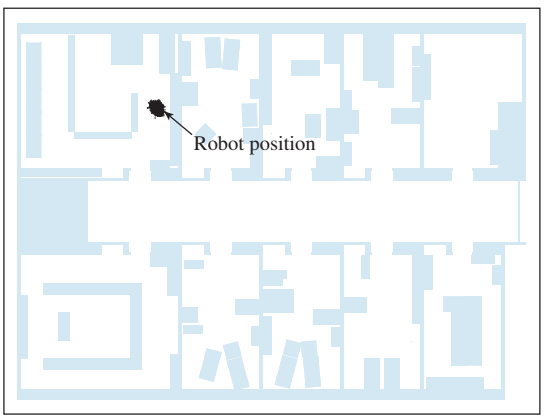
It is also possible to program a robot as a reactive agent, without explicitly reasoning about probability distributions over states. We cover that approach in Section 26.9.1.



(a)



(b)



(c)

Figure 26.7 Monte Carlo localization, a particle filtering algorithm for mobile robot localization. (a) Initial, global uncertainty. (b) Approximately bimodal uncertainty after navigating in the (symmetric) corridor. (c) Unimodal uncertainty after entering a room and finding it to be distinctive.

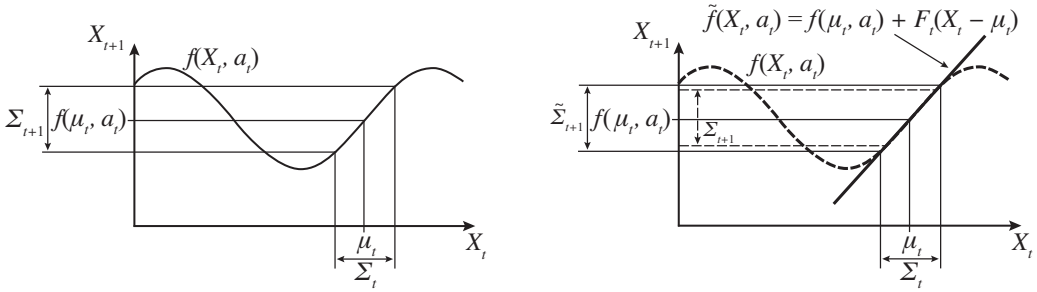


Figure 26.8 One-dimensional illustration of a linearized motion model: (a) The function f, and the projection of a mean μ_t and a covariance interval (based on Σ_t) into time $t+1$. (b) The linearized version is the tangent of f at μ_t. The projection of the mean μ_t is correct. However, the projected covariance $\tilde{\Sigma}_{t+1}$ differs from Σ_{t+1}.

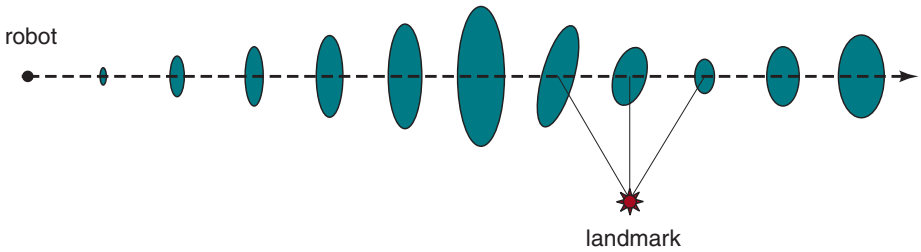


Figure 26.9 Localization using the extended Kalman filter. The robot moves on a straight line. As it progresses, its uncertainty in its location estimate increases, as illustrated by the error ellipses. When it observes a landmark with known position, the uncertainty is reduced.

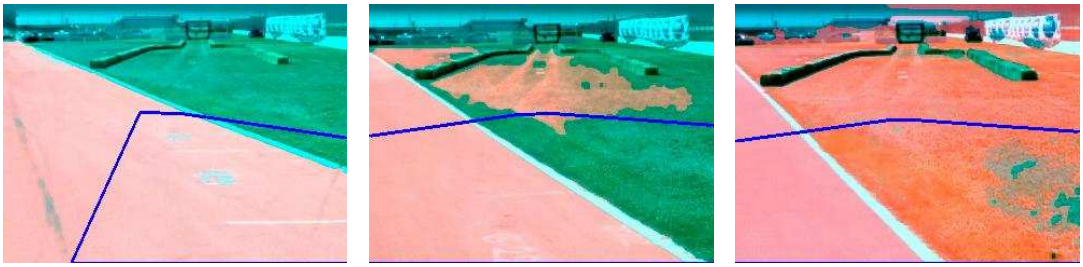
The trend in robotics is clearly towards representations with well-defined semantics. Probabilistic techniques outperform other approaches in many hard perceptual problems such as localization and mapping. However, statistical techniques are sometimes too cumbersome, and simpler solutions may be just as effective in practice. To help decide which approach to take, experience working with real physical robots is your best teacher.

26.4.3 Supervised and unsupervised learning in robot perception

Low-dimensional embedding

Machine learning plays an important role in robot perception. This is particularly the case when the best internal representation is not known. One common approach is to map high-dimensional sensor streams into lower-dimensional spaces using unsupervised machine learning methods (see Chapter 19). Such an approach is called **low-dimensional embedding**. Machine learning makes it possible to learn sensor and motion models from data, while simultaneously discovering a suitable internal representation.

Another machine learning technique enables robots to continuously adapt to big changes in sensor measurements. Picture yourself walking from a sunlit space into a dark room with neon lights. Clearly, things are darker inside. But the change of light source also affects all



(a) (b) (c)

Figure 26.10 Sequence of "drivable surface" classifications using adaptive vision. (a) Only the road is classified as drivable (pink area). The V-shaped blue line shows where the vehicle is heading. (b) The vehicle is commanded to drive off the road, and the classifier is beginning to classify some of the grass as drivable. (c) The vehicle has updated its model of drivable surfaces to correspond to grass as well as road. Courtesy of Sebastian Thrun.

the colors: neon light has a stronger component of green light than sunlight has. Yet somehow we seem not to notice the change. If we walk together with people into a neon-lit room, we don't think that their faces suddenly turned green. Our perception quickly adapts to the new lighting conditions, and our brain ignores the differences.

Adaptive perception techniques enable robots to adjust to such changes. One example is shown in Figure 26.10, taken from the autonomous driving domain. Here an unmanned ground vehicle adapts its classifier of the concept "drivable surface." How does this work? The robot uses a laser to provide classification for a small area immediately in front of the robot. When this area is found to be flat in the laser range scan, it is used as a positive training example for the concept "drivable surface." A mixture-of-Gaussians technique similar to the EM algorithm discussed in Chapter 21 is then trained to recognize the specific color and texture coefficients of the small sample patch. The images in Figure 26.10 are the result of applying this classifier to the full image.

Methods that make robots collect their own training data (with labels!) are called **self-supervised**. In this instance, the robot uses machine learning to leverage a short-range sensor that works well for terrain classification into a sensor that can see much farther. That allows the robot to drive faster, slowing down only when the sensor model says there is a change in the terrain that needs to be examined more carefully by the short-range sensors.

Self-supervised learning

26.5 Planning and Control

The robot's deliberations ultimately come down to deciding how to move, from the abstract task level all the way down to the currents that are sent to its motors. In this section, we simplify by assuming that perception (and, where needed, prediction) are given, so the world is observable. We further assume deterministic transitions (dynamics) of the world.

We start by separating motion from control. We define a **path** as a sequence of points in geometric space that a robot (or a robot part, such as an arm) will follow. This is related to the notion of path in Chapter 3, but here we mean a sequence of points in space rather than a sequence of discrete actions. The task of finding a good path is called **motion planning**.

Path

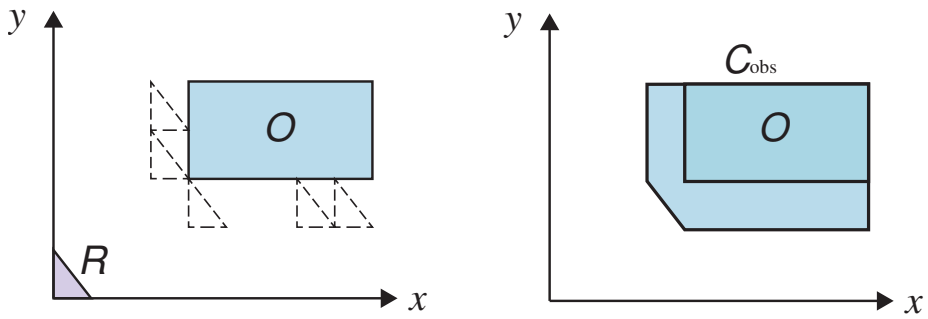


Figure 26.11 A simple triangular robot that can translate, and needs to avoid a rectangular obstacle. On the left is the workspace, on the right is the configuration space.

Trajectory tracking control
Trajectory

Once we have a path, the task of executing a sequence of actions to follow the path is called **trajectory tracking control**. A **trajectory** is a path that has a time associated with each point on the path. A path just says "go from A to B to C, etc." and a trajectory says "start at A, take 1 second to get to B, and another 1.5 seconds to get to C, etc."

26.5.1 Configuration space

Workspace

Imagine a simple robot, $\mathcal{R}$, in the shape of a right triangle as shown by the lavender triangle in the lower left corner of Figure 26.11. The robot needs to plan a path that avoids a rectangular obstacle, $\mathcal{O}$. The physical space that a robot moves about in is called the **workspace**. This particular robot can move in any direction in the $x-y$ plane, but cannot rotate. The figure shows five other possible positions of the robot with dashed outlines; these are each as close to the obstacle as the robot can get.

The body of the robot could be represented as a set of (x,y) points (or (x,y,z) points for a three-dimensional robot), as could the obstacle. With this representation, avoiding the obstacle means that no point on the robot overlaps any point on the obstacle. Motion planning would require calculations on sets of points, which can be complicated and time-consuming.

We can simplify the calculations by using a representation scheme in which all the points that comprise the robot are represented as a single point in an abstract multidimensional space, which we call the **configuration space**, or **C-space**. The idea is that the set of points that comprise the robot can be computed if we know (1) the basic measurements of the robot (for our triangle robot, the length of the three sides will do) and (2) the current **pose** of the robot—its position and orientation.

Configuration space
C-space

For our simple triangular robot, two dimensions suffice for the C-space: if we know the (x,y) coordinates of a specific point on the robot—we'll use the right-angle vertex—then we can calculate where every other point of the triangle is (because we know the size and shape of the triangle and because the triangle cannot rotate). In the lower-left corner of Figure 26.11, the lavender triangle can be represented by the configuration $(0,0)$.

If we change the rules so that the robot can rotate, then we will need three dimensions, (x,y,θ), to be able to calculate where every point is. Here θ is the robot's angle of rotation in the plane. If the robot also had the ability to stretch itself, growing uniformly by a scaling factor s, then the C-space would have four dimensions, (x,y,θ,s).

For now we'll stick with the simple two-dimensional C-space of the non-rotating triangle robot. The next task is to figure out where the points in the obstacle are in C-space. Consider the five dashed-line triangles on the left of Figure 26.11 and notice where the right-angle vertex is on each of these. Then imagine all the ways that the triangle could slide about. Obviously, the right-angle vertex can't go inside the obstacle, and neither can it get any closer than it is on any of the five dashed-line triangles. So you can see that the area where the right-angle vertex can't go—the **C-space obstacle**—is the five-sided polygon on the right of Figure 26.11 labeled $\mathcal{C}_{obs}$

C-space obstacle

In everyday language we speak of there being multiple obstacles for the robot—a table, a chair, some walls. But the math notation is a bit easier if we think of all of these as combining into one "obstacle" that happens to have disconnected components. In general, the C-space obstacle is the set of all points q in $\mathcal{C}$ such that, if the robot were placed in that configuration, its workspace geometry would intersect the workspace obstacle.

Let the obstacles in the workspace be the set of points O, and let the set of all points on the robot in configuration q be $\mathcal{A}(q)$. Then the C-space obstacle is defined as

$$C_{obs} = \{q : q \in C \textbf{ and } \mathcal{A}(q) \cap O \neq \{\}\}$$

and the **free space** is $C_{free} = C - C_{obs}$.

Free space

The C-space becomes more interesting for robots with moving parts. Consider the two-link arm from Figure 26.12(a). It is bolted to a table so the base does not move, but the arm has two joints that move independently—we call these **degrees of freedom (DOF)**. Moving the joints alters the (x, y) coordinates of the elbow, the gripper, and every point on the arm. The arm's configuration space is two-dimensional: $(\theta_{shou}, \theta_{elb})$, where θ_{shou} is the angle of the shoulder joint, and θ_{elb} is the angle of the elbow joint.

Degrees of freedom (DOF)

Knowing the configuration for our two-link arm means we can determine where each point on the arm is through simple trigonometry. In general, the **forward kinematics** mapping is a function

Forward kinematics

$$\phi_b : C \to W$$

that takes in a configuration and outputs the location of a particular point b on the robot when the robot is in that configuration. A particularly useful forward kinematics mapping is that for the robot's end effector, ϕ_{EE}. The set of all points on the robot in a particular configuration q is denoted by $\mathcal{A}(q) \subset W$:

$$\mathcal{A}(q) = \bigcup_b \{\phi_b(q)\}.$$

The inverse problem, of mapping a desired location for a point on the robot to the configuration(s) the robot needs to be in for that to happen, is known as **inverse kinematics**:

Inverse kinematics

$$IK_b : x \in W \mapsto \{q \in C \; s.t. \; \phi_b(q) = x\}.$$

Sometimes the inverse kinematics mapping might take not just a position, but also a desired orientation as input. When we want a manipulator to grasp an object, for instance, we can compute a desired position and orientation for its gripper, and use inverse kinematics to determine a goal configuration for the robot. Then a planner needs to find a way to get the robot from its current configuration to the goal configuration without intersecting obstacles.

Workspace obstacles are often depicted as simple geometric forms—especially in robotics textbooks, which tend to focus on polygonal obstacles. But how do the obstacles look in configuration space?

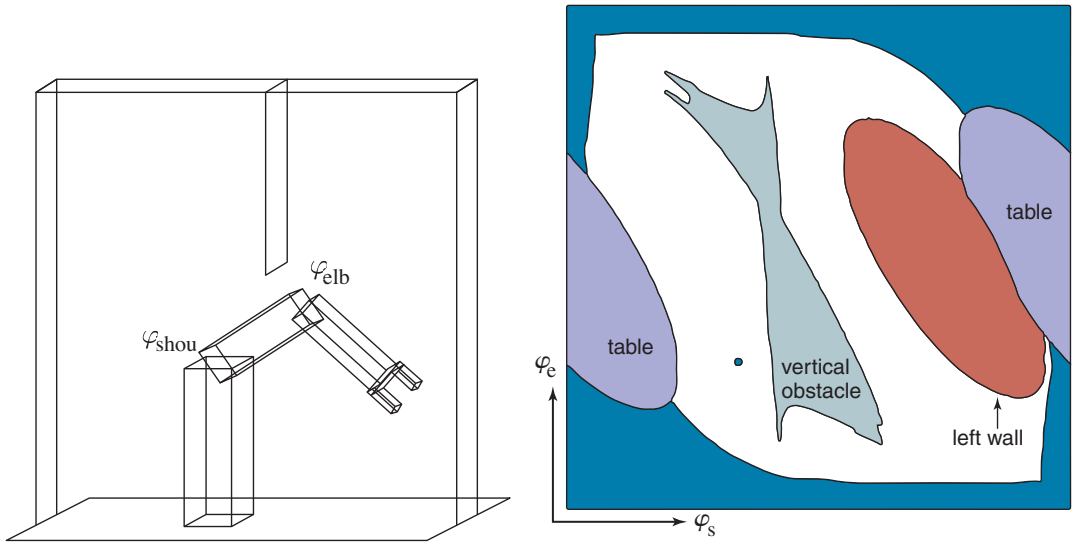


Figure 26.12 (a) Workspace representation of a robot arm with two degrees of freedom. The workspace is a box with a flat obstacle hanging from the ceiling. (b) Configuration space of the same robot. Only white regions in the space are configurations that are free of collisions. The dot in this diagram corresponds to the configuration of the robot shown on the left.

For the two-link arm, simple obstacles in the workspace, like a vertical line, have very complex C-space counterparts, as shown in Figure 26.12(b). The different shadings of the occupied space correspond to the different objects in the robot's workspace: the dark region surrounding the entire free space corresponds to configurations in which the robot collides with itself. It is easy to see that extreme values of the shoulder or elbow angles cause such a violation. The two oval-shaped regions on both sides of the robot correspond to the table on which the robot is mounted. The third oval region corresponds to the left wall.

Finally, the most interesting object in configuration space is the vertical obstacle that hangs from the ceiling and impedes the robot's motions. This object has a funny shape in configuration space: it is highly nonlinear and at places even concave. With a little bit of imagination the reader will recognize the shape of the gripper at the upper left end.

We encourage the reader to pause for a moment and study this diagram. The shape of this obstacle in C-space is not at all obvious! The dot inside Figure 26.12(b) marks the configuration of the robot in Figure 26.12(a). Figure 26.13 depicts three additional configurations, both in workspace and in configuration space. In configuration conf-1, the gripper is grasping the vertical obstacle.

We see that even if the robot's workspace is represented by flat polygons, the shape of the free space can be very complicated. In practice, therefore, one usually *probes* a configuration space instead of constructing it explicitly. A planner may generate a configuration and then test to see if it is in free space by applying the robot kinematics and then checking for collisions in workspace coordinates.

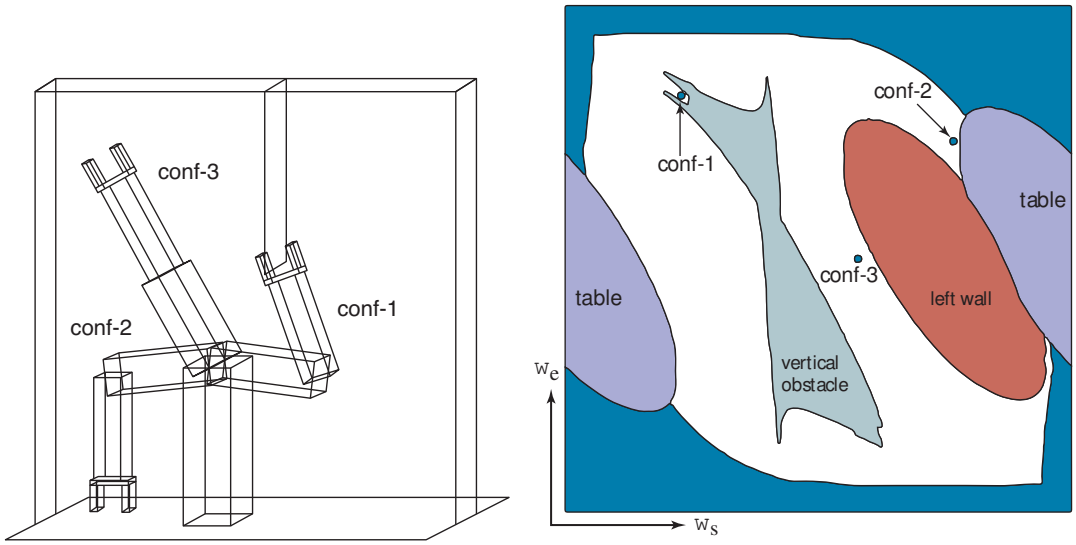


Figure 26.13 Three robot configurations, shown in workspace and configuration space.

26.5.2 Motion planning

Motion planning

The **motion planning** problem is that of finding a plan that takes a robot from one configuration to another without colliding with an obstacle. It is a basic building block for movement and manipulation. In Section 26.5.4 we will discuss how to do this under complicated dynamics, like steering a car that may drift off the path if you take a curve too fast. For now, we will focus on the simple motion planning problem of finding a geometric path that is collision free. Motion planning is a quintessentially continuous-state **search problem**, but it is often possible to discretize the space and apply the search algorithms from Chapter 3.

Piano mover's problem

The motion planning problem is sometimes referred to as the **piano mover's problem**. It gets its name from a mover's struggles with getting a large, irregular-shaped piano from one room to another without hitting anything. We are given:

- a workspace *world* W in either $\mathbb{R}^2$ for the plane or $\mathbb{R}^3$ for three dimensions,
- an *obstacle region* $O \subset W$,
- a robot with a configuration space C and set of points $\mathcal{A}(q)$ for $q \in C$,
- a starting configuration $q_s \in C$, and
- a goal configuration $q_g \in C$.

The obstacle region induces a C-space obstacle C_{obs} and its corresponding free space C_{free} defined as in the previous section. We need to find a continuous **path** through free space. We will use a parameterized curve, $\tau(t)$, to represent the path, where $\tau(0) = q_s$ and $\tau(1) = q_g$ and $\tau(t)$ for every t between 0 and 1 is some point in C_{free}. That is, t parameterizes how far we are along the path, from start to goal. Note that t acts somewhat like time in that as t increases the distance along the path increases, but t is always a point on the interval $[0, 1]$ and is not measured in seconds.

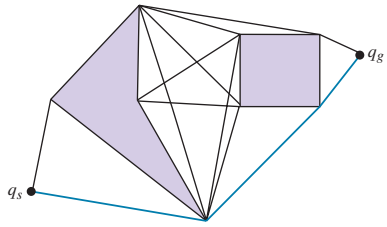


Figure 26.14 A visibility graph. Lines connect every pair of vertices that can "see" each other—lines that don't go through an obstacle. The shortest path must lie upon these lines.

The motion planning problem can be made more complex in various ways: defining the goal as a set of possible configurations rather than a single configuration; defining the goal in the workspace rather than the C-space; defining a cost function (e.g., path length) to be minimized; satisfying constraints (e.g., if the path involves carrying a cup of coffee, making sure that the cup is always oriented upright so the coffee does not spill).

The spaces of motion planning: Let's take a step back and make sure we understand the spaces involved in motion planning. First, there is the workspace or world W. Points in W are points in the everyday three-dimensional world. Next, we have the space of configurations, C. Points q in C are d-dimensional, with d the robot's number of degrees of freedom, and map to sets of points $\mathcal{A}(q)$ in W. Finally, there is the space of paths. The space of paths is a space of functions. Each point in this space maps to an entire curve through C-space. This space is ∞-dimensional! Intuitively, we need d dimensions for each configuration along the path, and there are as many configurations on a path as there are points in the number line interval $[0,1]$. Now let's consider some ways of solving the motion planning problem.

Visibility graphs

Visibility graph

For the simplified case of two-dimensional configuration spaces and polygonal C-space obstacles, **visibility graphs** are a convenient way to solve the motion planning problem with a guaranteed shortest-path solution. Let $V_{obs} \subset C$ be the set of vertices of the polygons making up C_{obs}, and let $V = V_{obs} \cup \{q_s, q_g\}$.

We construct a graph $G = (V, E)$ on the vertex set V with edges $e_{ij} \in E$ connecting a vertex v_i to another vertex v_j if the line connecting the two vertices is collision-free—that is, if $\{\lambda v_i + (1-\lambda)v_j : \lambda \in [0,1]\} \cap C_{obs} = \{\}$. When this happens, we say the two vertices "can see each other," which is where "visibility" graphs got their name.

To solve the motion planning problem, all we need to do is run a discrete graph search (e.g., best-first search) on the graph G with starting state q_s and goal q_g. In Figure 26.14 we see a visibility graph and an optimal three-step solution. An optimal search on visibility graphs will always give us the optimal path (if one exists), or report failure if no path exists.

Voronoi diagrams

Visibility graphs encourage paths that run immediately adjacent to an obstacle—if you had to walk around a table to get to the door, the shortest path would be to stick as close to the table as possible. However, if motion or sensing is nondeterministic, that would put you at risk of bumping into the table. One way to address this is to pretend that the robot's body is a bit

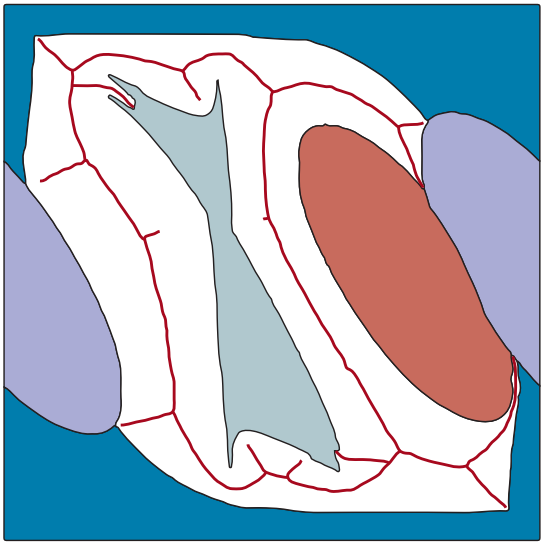


Figure 26.15 A Voronoi diagram showing the set of points (black lines) equidistant to two or more obstacles in configuration space.

larger than it actually is, providing a buffer zone. Another way is to accept that path length is not the only metric we want to optimize. Section 26.8.2 shows how to learn a good metric from human examples of behavior.

A third way is to use a different technique, one that puts paths as far away from obstacles as possible rather than hugging close to them. A **Voronoi diagram** is a representation that allows us to do just that. To get an idea for what a Voronoi diagram does, consider a space where the obstacles are, say, a dozen small points scattered about a plane. Now surround each of the obstacle points with a **region** consisting of all the points in the plane that are closer to that obstacle point than to any other obstacle point. Thus, the regions partition the plane. The Voronoi diagram consists of the set of regions, and the **Voronoi graph** consists of the edges and vertices of the regions.

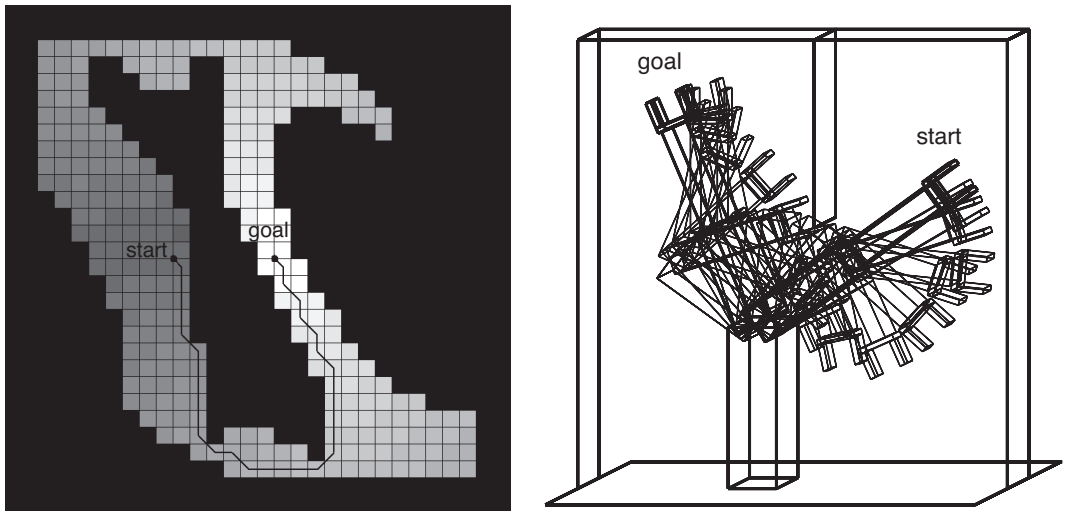
Voronoi diagram

Region

Voronoi graph

When obstacles are areas, not points, everything stays pretty much the same. Each region still contains all the points that are closer to one obstacle than to any other, where distance is measured to the closest point on an obstacle. The boundaries between regions still correspond to points that are equidistant between two obstacles, but now the boundary may be a curve rather than a straight line. Computing these boundaries can be prohibitively expensive in high-dimensional spaces.

To solve the motion planning problem, we connect the start point q_s to the closest point on the Voronoi graph via a straight line, and the same for the goal point q_g. We then use discrete graph search to find the shortest path on the graph. For problems like navigating through corridors indoors, this gives a nice path that goes down the middle of the corridor. However, in outdoor settings it can come up with inefficient paths, for example suggesting an unnecessary 100 meter detour to stick to the middle of a wide-open 200-meter space.



(a) (b)

Figure 26.16 (a) Value function and path found for a discrete grid cell approximation of the configuration space. (b) The same path visualized in workspace coordinates. Notice how the robot bends its elbow to avoid a collision with the vertical obstacle.

Cell decomposition

Cell decomposition

An alternative approach to motion planning is to discretize the C-space. **Cell decomposition** methods decompose the free space into a finite number of contiguous regions, called cells. These cells are designed so that the path-planning problem within a single cell can be solved by simple means (e.g., moving along a straight line). The path-planning problem then becomes a discrete graph search problem (as with visibility graphs and Voronoi graphs) to find a path through a sequence of cells.

The simplest cell decomposition consists of a regularly spaced grid. Figure 26.16(a) shows a square grid decomposition of the space and a solution path that is optimal for this grid size. Grayscale shading indicates the *value* of each free-space grid cell—the cost of the shortest path from that cell to the goal. (These values can be computed by a deterministic form of the VALUE-ITERATION algorithm given in Figure 16.6 on page 563.) Figure 26.16(b) shows the corresponding workspace trajectory for the arm. Of course, we could also use the A* algorithm to find a shortest path.

This grid decomposition has the advantage that it is simple to implement, but it suffers from three limitations. First, it is workable only for low-dimensional configuration spaces, because the number of grid cells increases exponentially with d, the number of dimensions. (Sounds familiar? This is the curse of dimensionality.) Second, paths through discretized state space will not always be smooth. We see in Figure 26.16(a) that the diagonal parts of the path are jagged and hence very difficult for the robot to follow accurately. The robot can attempt to smooth out the solution path, but this is far from straightforward.

Third, there is the problem of what to do with cells that are "mixed"—that is, neither entirely within free space nor entirely within occupied space. A solution path that includes

such a cell may not be a real solution, because there may be no way to safely cross the cell. This would make the path planner *unsound*. On the other hand, if we insist that only completely free cells may be used, the planner will be *incomplete*, because it might be the case that the only paths to the goal go through mixed cells—it might be that a corridor is actually wide enough for the robot to pass, but the corridor is covered only by mixed cells.

The first approach to this problem is *further subdivision* of the mixed cells—perhaps using cells of half the original size. This can be continued recursively until a path is found that lies entirely within free cells. This method works well and is complete if there is a way to decide if a given cell is a mixed cell, which is easy only if the configuration space boundaries have relatively simple mathematical descriptions.

It is important to note that cell decomposition does not necessarily require explicitly representing the obstacle space C_{obs}. We can decide to include a cell or not by using a **collision checker**. This is a crucial notion to motion planning. A collision checker is a function $\gamma(q)$ that maps to 1 if the configuration collides with an obstacle, and 0 otherwise. It is much easier to check whether a specific configuration is in collision than to explicitly construct the entire obstacle space C_{obs}.

Collision checker

Examining the solution path shown in Figure 26.16(a), we can see an additional difficulty that will have to be resolved. The path contains arbitrarily sharp corners, but a physical robot has momentum and cannot change direction instantaneously. This problem can be solved by storing, for each grid cell, the exact continuous state (position and velocity) that was attained when the cell was reached in the search. Assume further that when propagating information to nearby grid cells, we use this continuous state as a basis, and apply the continuous robot motion model for jumping to nearby cells. So we don't make an instantaneous 90° turn; we make a rounded turn governed by the laws of motion. We can now guarantee that the resulting trajectory is smooth and can indeed be executed by the robot. One algorithm that implements this is **hybrid A***.

Hybrid A*

Randomized motion planning

Randomized motion planning does graph search on a *random* decomposition of the configuration space, rather than a regular cell decomposition. The key idea is to sample a random set of points and to create edges between them if there is a very simple way to get from one to the other (e.g., via a straight line) without colliding; then we can search on this graph.

A **probabilistic roadmap (PRM)** algorithm is one way to leverage this idea. We assume access to a collision checker γ (defined on page 953), and to a **simple planner** $B(q_1, q_2)$ that returns a path from q_1 to q_2 (or failure) but does so *quickly*. This simple planner is not going to be complete—it might return failure even if a solution actually exists. Its job is to quickly try to connect q_1 and q_2 and let the main algorithm know if it succeeds. We will use it to define whether an edge exists between two vertices.

Probabilistic roadmap (PRM)
Simple planner

The algorithm starts by sampling M **milestones**—points in C_{free}—in addition to the points q_s and q_g. It uses rejection sampling, where configurations are sampled randomly and collision-checked using γ until a total of M milestones are found. Next, the algorithm uses the simple planner to try to connect pairs of milestones. If the simple planner returns success, then an edge between the pair is added to the graph; otherwise, the graph remains as is. We try to connect each milestone either to its k nearest neighbors (we call this k-PRM), or to all milestones in a sphere of a radius r. Finally, the algorithm searches for a path on this

Milestone

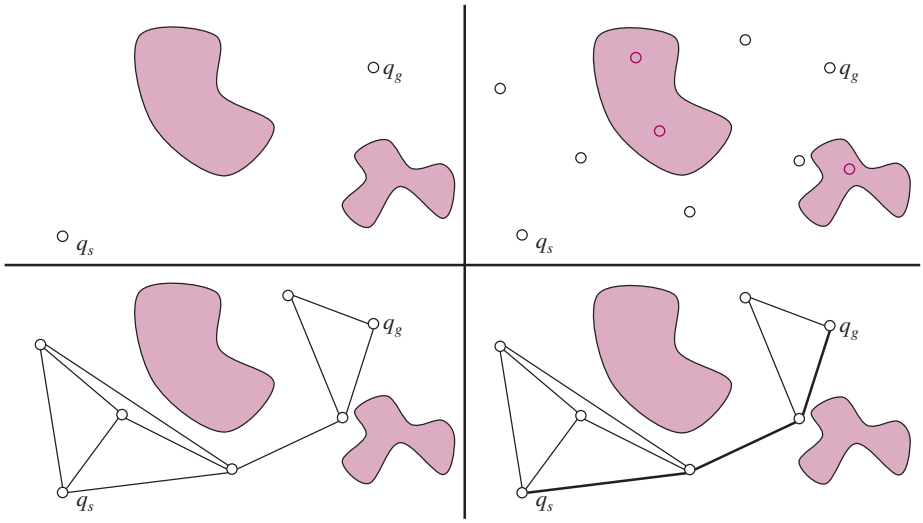


Figure 26.17 The probabilistic roadmap (PRM) algorithm. **Top left**: the start and goal configurations. **Top right**: sample M collision-free milestones (here $M = 5$). **Bottom left**: connect each milestone to its k nearest neighbors (here $k = 3$). **Bottom right**: find the shortest path from the start to the goal on the resulting graph.

graph from q_s to q_g. If no path is found, then M more milestones are sampled, added to the graph, and the process is repeated.

Probabilistically complete

Figure 26.17 shows a roadmap with the path found between two configurations. PRMs are not complete, but they are what is called **probabilistically complete**—they will eventually find a path, if one exists. Intuitively, this is because they keep sampling more milestones. PRMs work well even in high-dimensional configuration spaces.

Multi-query planning

PRMs are also popular for **multi-query planning**, in which we have multiple motion planning problems within the same C-space. Often, once the robot reaches a goal, it is called upon to reach another goal in the same workspace. PRMs are really useful, because the robot can dedicate time up front to constructing a roadmap, and amortize the use of that roadmap over multiple queries.

Rapidly-exploring random trees

Rapidly exploring random trees (RRTs)

An extension of PRMs called **rapidly exploring random trees (RRTs)** is popular for single-query planning. We incrementally build two trees, one with q_s as the root and one with q_g as the root. Random milestones are chosen, and an attempt is made to connect each new milestone to the existing trees. If a milestone connects both trees, that means a solution has been found, as in Figure 26.18. If not, the algorithm finds the closest point in each tree and adds to the tree a new edge that extends from the point by a distance δ towards the milestone. This tends to grow the tree towards previously unexplored sections of the space.

Roboticists love RRTs for their ease of use. However, RRT solutions are typically nonoptimal and lack smoothness. Therefore, RRTs are often followed by a post-processing step. The most common one is "short-cutting," in which we randomly select one of the vertices on the solution path and try to remove it by connecting its neighbors to each other (via the simple

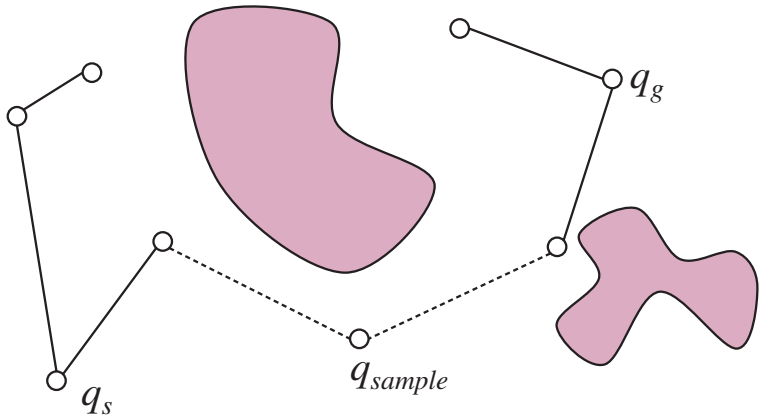


Figure 26.18 The bidirectional RRT algorithm constructs two trees (one from the start, the other from the goal) by incrementally connecting each sample to the closest node in each tree, if the connection is possible. When a sample connects to both trees, that means we have found a solution path.

(a) (b) (c)

Figure 26.19 Snapshots of a trajectory produced by an RRT and post-processed with short-cutting. Courtesy of Anca Dragan.

planner). We do this repeatedly for as many steps as we have compute time for. Even then, the trajectories might look a little unnatural due to the random positions of the milestone that were selected, as shown in Figure 26.19.

RRT* is a modification to RRT that makes the algorithm asymptotically optimal: the solution converges to the optimal solution as more and more milestones are sampled. The key idea is to pick the nearest neighbor based on a notion of cost to come rather than distance from the milestone only, and to rewire the tree, swapping parents of older vertices if it is cheaper to reach them via the new milestone. RRT*

Trajectory optimization for kinematic planning

Randomized sampling algorithms tend to first construct a complex but feasible path and then optimize it. Trajectory optimization does the opposite: it starts with a simple but infeasible path, and then works to push it out of collision. The goal is to find a path that optimizes a cost

function[1] over paths. That is, we want to minimize the cost function $J(\tau)$, where $\tau(0) = q_s$ and $\tau(1) = q_g$.

J is called a **functional** because it is a function over functions. The argument to J is τ, which is itself a function: $\tau(t)$ takes as input a point in the $[0,1]$ interval and maps it to a configuration. A standard cost functional trades off between two important aspects of the robot's motion: collision avoidance and efficiency,

$$J = J_{obs} + \lambda J_{eff}$$

where the efficiency J_{eff} measures the length of the path and may also measure smoothness. A convenient way to define efficiency is with a quadratic: it integrates the squared first derivative of τ (we will see in a bit why this does in fact incentivize short paths):

$$J_{eff} = \int_0^1 \frac{1}{2} \|\dot{\tau}(s)\|^2 ds\,.$$

For the obstacle term, assume we can compute the distance $d(x)$ from any point $x \in W$ to the nearest obstacle edge. This distance is positive outside of obstacles, 0 at the edge, and negative inside. This is called a **signed distance field**. We can now define a cost field in the workspace, call it c, that has high cost inside of obstacles, and a small cost right outside. With this cost, we can make points in the workspace really hate being inside obstacles, and dislike being right next to them (avoiding the visibility graph problem of their always hanging out by the edges of obstacles). Of course, our robot is not a point in the workspace, so we have some more work to do—we need to consider all points b on the robot's body:

Signed distance field

$$J_{obs} = \int_0^1 \int_b c(\underbrace{\phi_b(\tau(s))}_{\in W}) \| \frac{d}{ds} \underbrace{\phi_b(\tau(s))}_{\in W} \| \, db \, ds\,.$$

This is called a **path integral**—it does not just integrate c along the way for each body point, but it multiplies by the derivative to make the cost invariant to *retiming* of the path. Imagine a robot sweeping through the cost field, accumulating cost as is moves. Regardless of how fast or slow the arm moves through the field, it must accumulate the exact same cost.

Path integral

The simplest way to solve the optimization problem above and find a path is *gradient descent*. If you are wondering how to take gradients of functionals with respect to functions, something called the *calculus of variations* is here to help. It is especially easy for functionals of the form

$$J[\tau] = \int_0^1 F(s, \tau(s), \dot{\tau}(s)) ds$$

which are integrals of functions that depend just on the parameter s, the value of the function at s, and the derivative of the function at s. In such a case, the **Euler-Lagrange equation** says that the gradient is

Euler-Lagrange equation

$$\nabla_\tau J(s) = \frac{\partial F}{\partial \tau(s)}(s) - \frac{d}{dt} \frac{\partial F}{\partial \dot{\tau}(s)}(s)\,.$$

If we look closely at J_{eff} and J_{obs}, they both follow this pattern. In particular for J_{eff}, we have $F(s, \tau(s), \dot{\tau}(s)) = \|\dot{\tau}(s)\|^2$. To get a bit more comfortable with this, let's compute the gradient

[1] Roboticists like to minimize a cost function J, whereas in other parts of AI we try to maximize a utility function U or a reward R.

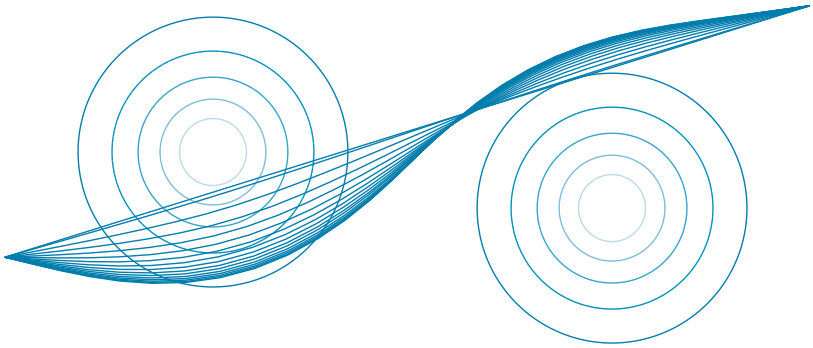


Figure 26.20 Trajectory optimization for motion planning. Two point-obstacles with circular bands of decreasing cost around them. The optimizer starts with the straight line trajectory, and lets the obstacles bend the line away from collisions, finding the minimum path through the cost field.

for J_{eff} only. We see that F does not have a direct dependence on $\tau(s)$, so the first term in the formula is 0. We are left with

$$\nabla_\tau J(s) = 0 - \frac{d}{dt}\dot{\tau}(s)$$

since the partial of F with respect to $\dot{\tau}(s)$ is $\dot{\tau}(s)$.

Notice how we made things easier for ourselves when defining J_{eff}—it's a nice quadratic of the derivative (and we even put a $\frac{1}{2}$ in front so that the 2 nicely cancels out). In practice, you will see this trick happen a lot for optimization—the art is not just in choosing how to optimize the cost function, but also in choosing a cost function that will play nicely with how you will optimize it. Simplifying our gradient, we get

$$\nabla_\tau J(s) = -\ddot{\tau}(s)\,.$$

Now, since J_{eff} is a quadratic, setting this gradient to 0 gives us the solution for τ if we didn't have to deal with obstacles. Integrating once, we get that the first derivative needs to be constant; integrating again we get that $\tau(s) = a \cdot s + b$, with a and b determined by the endpoint constraints for $\tau(0)$ and $\tau(1)$. The optimal path with respect to J_{eff} is thus the straight line from start to goal! It is indeed the most efficient way to go from one to the other if there are no obstacles to worry about.

Of course, the addition of J_{obs} is what makes things difficult—and we will spare you deriving its gradient here. The robot would typically initialize its path to be a straight line, which would plow right through some obstacles. It would then calculate the gradient of the cost about the current path, and the gradient would serve to push the path away from the obstacles (Figure 26.20). Keep in mind that gradient descent will only find a *locally optimal* solution—just like hill climbing. Methods such as simulated annealing (Section 4.1.2) can be used for exploration, to make it more likely that the local optimum is a good one.

26.5.3 Trajectory tracking control

We have covered how to *plan* motions, but not how to actually *move*—to apply current to motors, to produce torque, to move the robot. This is the realm of **control theory**, a field of increasing importance in AI. There are two main questions to deal with: how do we turn

Control theory

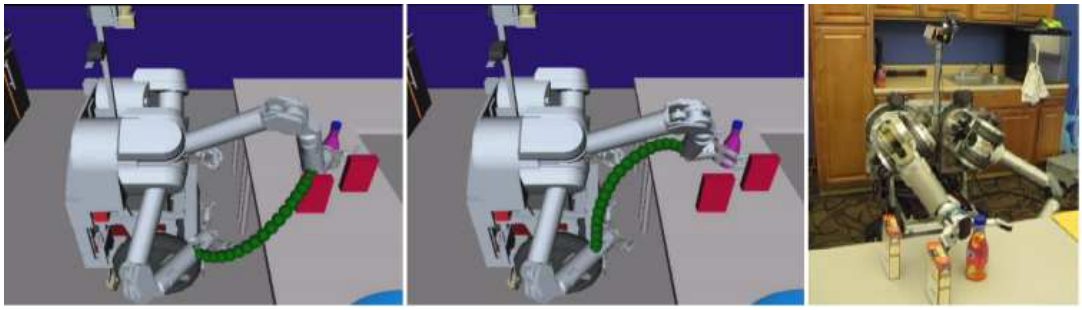


Figure 26.21 The task of reaching to grasp a bottle solved with a trajectory optimizer. Left: the initial trajectory, plotted for the end effector. Middle: the final trajectory after optimization. Right: the goal configuration. Courtesy of Anca Dragan. See Ratliff *et al.* (2009).

a mathematical description of a path into a sequence of actions in the real world (open-loop control), and how do we make sure that we are staying on track (closed-loop control)?

From configurations to torques for open-loop tracking: Our path $\tau(t)$ gives us configurations. The robot starts at rest at $q_s = \tau(0)$. From there the robot's motors will turn currents into torques, leading to motion. But what torques should the robot aim for, such that it ends up at $q_g = \tau(1)$?

Dynamics model

This is where the idea of a **dynamics model** (or transition model) comes in. We can give the robot a function f that computes the effects torques have on the configuration. Remember $F = ma$ from physics? Well, there is something like that for torques too, in the form $u = f^{-1}(q, \dot{q}, \ddot{q})$, with u a torque, $\dot{q}$ a velocity, and $\ddot{q}$ an acceleration.[2] If the robot is at configuration q and velocity $\dot{q}$, and applied torque u, that would lead to acceleration $\ddot{q} = f(q, \dot{q}, u)$.

Dynamic state
Kinematic state

The tuple $(q, \dot{q})$ is a **dynamic state**, because it includes velocity, whereas q is the **kinematic state** and is not sufficient for computing exactly what torque to apply. f is a deterministic dynamics model in the MDP over dynamic states with torques as actions. f^{-1} is the **inverse dynamics**, telling us what torque to apply if we want a particular acceleration, which leads to a change in velocity and thus a change in dynamic state.

Inverse dynamics

Now, naively, we could think of $t \in [0, 1]$ as "time" on a scale from 0 to 1 and select our torque using inverse dynamics:

$$u(t) = f^{-1}(\tau(t), \dot{\tau}(t), \ddot{\tau}(t)) \tag{26.2}$$

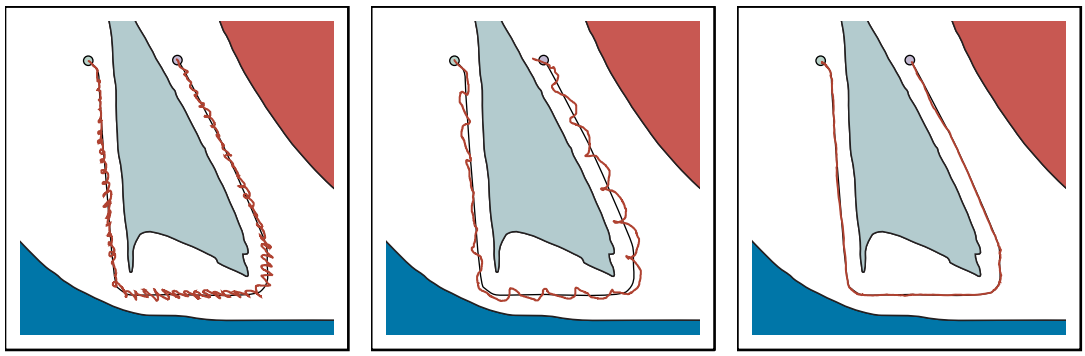
assuming that the robot starts at $(\tau(0), \dot{\tau}(0))$. In reality though, things are not that easy.

The path τ was created as a sequence of points, without taking velocities and accelerations into account. As such, the path may not satisfy $\dot{\tau}(0) = 0$ (the robot starts at 0 velocity), or even that τ is differentiable (let alone twice differentiable). Further, the meaning of the endpoint "1" is unclear: how many seconds does that map to?

Retiming

In practice, before we even think of tracking a reference path, we usually **retime** it, that is, transform it into a trajectory $\xi(t)$ that maps the interval $[0, T]$ for some time duration T into points in the configuration space C. (The symbol ξ is the Greek letter Xi.) Retiming is trickier than you might think, but there are approximate ways to do it, for instance by picking a maximum velocity and acceleration, and using a profile that accelerates to that maximum

[2] We omit the details of f^{-1} here, but they involve mass, inertia, gravity, and Coriolis and centrifugal forces.



(a) (b) (c)

Figure 26.22 Robot arm control using (a) proportional control with gain factor 1.0, (b) proportional control with gain factor 0.1, and (c) PD (proportional derivative) control with gain factors 0.3 for the proportional component and 0.8 for the differential component. In all cases the robot arm tries to follow the smooth line path, but in (a) and (b) deviates substantially from the path.

velocity, stays there as long as it can, and then decelerates back to 0. Assuming we can do this, Equation (26.2) above can be rewritten as

$$u(t) = f^{-1}(\xi(t), \dot{\xi}(t), \ddot{\xi}(t)) . \tag{26.3}$$

Even with the change from τ to ξ, an actual trajectory, the equation of applying torques from above (called a **control law**) has a problem in practice. Thinking back to the reinforcement learning section, you might guess what it is. The equation works great in the situation where f is exact, but pesky reality gets in the way as usual: in real systems, we can't measure masses and inertias exactly, and f might not properly account for physical phenomena like **stiction** in the motors (the friction that tends to prevent stationary surfaces from being set in motion—to make them stick). So, when the robot arm starts applying those torques but f is wrong, the errors accumulate and you deviate further and further from the reference path.

Control law

Stiction

Rather than just letting those errors accumulate, a robot can use a control process that looks at where it thinks it is, compares that to where it wanted to be, and applies a torque to minimize the error.

A controller that provides force in negative proportion to the observed error is known as a proportional controller or **P controller** for short. The equation for the force is:

P controller

$$u(t) = K_P(\xi(t) - q_t)$$

where q_t is the current configuration, and K_P is a constant representing the **gain factor** of the controller. K_P regulates how strongly the controller corrects for deviations between the actual state q_t and the desired state $\xi(t)$.

Gain factor

Figure 26.22(a) illustrates what can go wrong with proportional control. Whenever a deviation occurs—whether due to noise or to constraints on the forces the robot can apply—the robot provides an opposing force whose magnitude is proportional to this deviation. Intuitively, this might appear plausible, since deviations should be compensated by a counterforce to keep the robot on track. However, as Figure 26.22(a) illustrates, a proportional controller can cause the robot to apply too much force, overshooting the desired path and zig-zagging

back and forth. This is the result of the natural inertia of the robot: once driven back to its reference position the robot has a velocity that can't instantaneously be stopped.

In Figure 26.22(a), the parameter $K_P = 1$. At first glance, one might think that choosing a smaller value for K_P would remedy the problem, giving the robot a gentler approach to the desired path. Unfortunately, this is not the case. Figure 26.22(b) shows a trajectory for $K_P = .1$, still exhibiting oscillatory behavior. The lower value of the gain parameter helps, but does not solve the problem. In fact, in the absence of friction, the P controller is essentially a spring law; so it will oscillate indefinitely around a fixed target location.

There are a number of controllers that are superior to the simple proportional control law.

Stable
Strictly stable

A controller is said to be **stable** if small perturbations lead to a bounded error between the robot and the reference signal. It is said to be **strictly stable** if it is able to return to and then stay on its reference path upon such perturbations. Our P controller appears to be stable but not strictly stable, since it fails to stay anywhere near its reference trajectory.

PD controller

The simplest controller that achieves strict stability in our domain is a **PD controller**. The letter 'P' stands again for *proportional*, and 'D' stands for *derivative*. PD controllers are described by the following equation:

$$u(t) = K_P(\xi(t) - q_t) + K_D(\dot{\xi}(t) - \dot{q}_t)\,. \tag{26.4}$$

As this equation suggests, PD controllers extend P controllers by a differential component, which adds to the value of $u(t)$ a term that is proportional to the first derivative of the error $\xi(t) - q_t$ over time. What is the effect of such a term? In general, a derivative term dampens the system that is being controlled. To see this, consider a situation where the error is changing rapidly over time, as is the case for our P controller above. The derivative of this error will then counteract the proportional term, which will reduce the overall response to the perturbation. However, if the same error persists and does not change, the derivative will vanish and the proportional term dominates the choice of control.

Figure 26.22(c) shows the result of applying this PD controller to our robot arm, using as gain parameters $K_P = .3$ and $K_D = .8$. Clearly, the resulting path is much smoother, and does not exhibit any obvious oscillations.

PD controllers do have failure modes, however. In particular, PD controllers may fail to regulate an error down to zero, even in the absence of external perturbations. Often such a situation is the result of a systematic external force that is not part of the model. For example, an autonomous car driving on a banked surface may find itself systematically pulled to one side. Wear and tear in robot arms causes similar systematic errors. In such situations, an over-proportional feedback is required to drive the error closer to zero. The solution to this problem lies in adding a third term to the control law, based on the *integrated* error over time:

$$u(t) = K_P(\xi(t) - q_t) + K_I \int_0^t (\xi(s) - q_s)ds + K_D(\dot{\xi}(t) - \dot{q}_t)\,. \tag{26.5}$$

Here K_I is a third gain parameter. The term $\int_0^t(\xi(s)$ calculates the integral of the error over time. The effect of this term is that long-lasting deviations between the reference signal and the actual state are corrected. Integral terms, then, ensure that a controller does not exhibit systematic long-term error, although they do pose a danger of oscillatory behavior.

PID controller

A controller with all three terms is called a **PID controller** (for proportional integral derivative). PID controllers are widely used in industry, for a variety of control problems. Think of the three terms as follows—proportional: try harder the farther away you are from

the path; derivative: try even harder if the error is increasing; integral: try harder if you haven't made progress for a long time.

A middle ground between open-loop control based on inverse dynamics and closed-loop PID control is called **computed torque control**. We compute the torque our model thinks we will need, but compensate for model inaccuracy with proportional error terms:

Computed torque control

$$u(t) = \underbrace{f^{-1}(\xi(t), \dot{\xi}(t), \ddot{\xi}(t))}_{feedforward} + \underbrace{m(\xi(t))\left(K_P(\xi(t) - q_t) + K_D(\dot{\xi}(t) - \dot{q}_t)\right)}_{feedback}. \quad (26.6)$$

The first term is called the **feedforward component** because it looks forward to where the robot needs to go and computes what torque might be required. The second is the **feedback component** because it feeds the current error in the dynamic state back into the control law. $m(q)$ is the inertia matrix at configuration q—unlike normal PD control, the gains change with the configuration of the system.

Feedforward component

Feedback component

Plans versus policies

Let's take a step back and make sure we understand the analogy between what happened so far in this chapter and what we learned in the search, MDP, and reinforcement learning chapters. With motion in robotics, we are really considering an underlying MDP where the states are dynamic states (configuration and velocity), and the actions are control inputs, usually in the form of torques. If you take another look at our control laws above, they are *policies*, not *plans*—they tell the robot what action to take from *any* state it might reach. However, they are usually far from *optimal* policies. Because the dynamic state is continuous and high dimensional (as is the action space), optimal policies are computationally difficult to extract.

Instead, what we did here is to break up the problem. We come up with a plan first, in a simplified state and action space: we use only the kinematic state, and assume that states are reachable from one another without paying attention to the underlying dynamics. This is motion planning, and it gives us the reference path. If we knew the dynamics perfectly, we could turn this into a plan for the original state and action space with Equation (26.3).

But because our dynamics model is typically erroneous, we turn it instead into a policy that tries to follow the plan—getting back to it when it drifts away. When doing this, we introduce suboptimality in two ways: first by planning without considering dynamics, and second by assuming that if we deviate from the plan, the optimal thing to do is to return to the original plan. In what follows, we describe techniques that compute policies directly over the dynamic state, avoiding the separation altogether.

26.5.4 Optimal control

Rather than using a planner to create a kinematic path, and only worrying about the dynamics of the system after the fact, here we discuss how we might be able to do it all at once. We'll take the trajectory optimization problem for kinematic paths, and turn it into true trajectory optimization with dynamics: we will optimize directly over the actions, taking the dynamics (or transitions) into account.

This brings us much closer to what we've seen in the search and MDP chapters. If we know the system's dynamics, then we can find a sequence of actions to execute, as we did in Chapter 3. If we're not sure, then we might want a policy, as in Chapter 16.

In this section, we are looking more directly at the underlying MDP the robot works in. We're switching from the familiar discrete MDPs to continuous ones. We will denote our dynamic state of the world by x, as is common practice—the equivalent of s in discrete MDPs. Let x_s and x_g be the starting and goal states.

We want to find a sequence of actions that, when executed by the robot, result in state-action pairs with low cumulative cost. The actions are torques which we denote with $u(t)$ for t starting at 0 and ending at T. Formally, we want to find the sequence of torques u that minimize a cumulative cost J:

$$\min_u \int_0^T J(x(t), u(t))dt \tag{26.7}$$

subject to the constraints

$$\forall t,\ \dot{x}(t) = f(x(t), u(t))$$
$$x(0) = x_s,\ x(T) = x_g\,.$$

How is this connected to motion planning and trajectory tracking control? Well, imagine we take the notion of efficiency and clearance away from the obstacles and put it into the cost function J, just as we did before in trajectory optimization over kinematic state. The dynamic state is the configuration and velocity, and torques u change it via the dynamics f from open-loop trajectory tracking. The difference is that now we're thinking about the configurations and the torques at the same time. Sometimes, we might want to treat collision avoidance as a hard constraint as well, something we've also mentioned before when we looked at trajectory optimization for the kinematic state only.

To solve this optimization problem, we can take gradients of J—not with respect to the sequence τ of configurations anymore, but directly with respect to the controls u. It is sometimes helpful to include the state sequence x as a decision variable too, and use the dynamics constraints to ensure that x and u are consistent. There are various trajectory optimization techniques using this approach; two of them go by the names **multiple shooting** and **direct collocation**. None of these techniques will find the global optimal solution, but in practice they can effectively make humanoid robots walk and make autonomous cars drive.

Magic happens when in the problem above, J is quadratic and f is linear in x and u. We want to minimize

$$\min \int_0^\infty x^T Q x + u^T R u dt \qquad \text{subject to} \qquad \forall t,\ \dot{x}(t) = Ax(t) + Bu(t)\,.$$

We can optimize over an infinite horizon rather than a finite one, and we obtain a policy from any state rather than just a sequence of controls. Q and R need to be positive definite matrices for this to work. This gives us the **linear quadratic regulator (LQR)**. With LQR, the optimal value function (called **cost to go**) is quadratic, and the optimal policy is linear. The policy looks like $u = -Kx$, where finding the matrix K requires solving an algebraic **Riccati equation**—no local optimization, no value iteration, no policy iteration are needed!

Linear quadratic regulator (LQR)

Riccati equation

Because of the ease of finding the optimal policy, LQR finds many uses in practice despite the fact that real problems seldom actually have quadratic costs and linear dynamics. A really useful method is called **iterative LQR (ILQR)**, which works by starting with a solution and then iteratively computing a linear approximation of the dynamics and a quadratic approximation of the cost around it, then solving the resulting LQR system to arrive at a new solution. Variants of LQR are also often used for trajectory tracking.

Iterative LQR (ILQR)

26.6 Planning Uncertain Movements

In robotics, uncertainty arises from partial observability of the environment and from the stochastic (or unmodeled) effects of the robot's actions. Errors can also arise from the use of approximation algorithms such as particle filtering, which does not give the robot an exact belief state even if the environment is modeled perfectly.

The majority of today's robots use deterministic algorithms for decision making, such as the path-planning algorithms of the previous section, or the search algorithms that were introduced in Chapter 3. These deterministic algorithms are adapted in two ways: first, they deal with the continuous state space by turning it into a discrete space (for example with visibility graphs or cell decomposition). Second, they deal with uncertainty in the current state by choosing the **most likely state** from the probability distribution produced by the state estimation algorithm. That approach makes the computation faster and makes a better fit for the deterministic search algorithms. In this section we discuss methods for dealing with uncertainty that are analogous to the more complex search algorithms covered in Chapter 4.

Most likely state

First, instead of deterministic plans, uncertainty calls for policies. We already discussed how trajectory tracking control turns a plan into a policy to compensate for errors in dynamics. Sometimes though, if the most likely hypothesis changes enough, tracking the plan designed for a different hypothesis is too suboptimal. This is where **online replanning** comes in: we can recompute a new plan based on the new belief. Many robots today use a technique called **model predictive control (MPC)**, where they plan for a shorter time horizon, but replan at every time step. (MPC is therefore closely related to real-time search and game-playing algorithms.) This effectively results in a policy: at every step, we run a planner and take the first action in the plan; if new information comes along, or we end up not where we expected, that's OK, because we are going to replan anyway and that will tell us what to do next.

Online replanning

Model predictive control (MPC)

Second, uncertainty calls for **information gathering** actions. When we consider only the information we have and make a plan based on it (this is called separating estimation from control), we are effectively solving (approximately) a new MDP at every step, corresponding to our current belief about where we are or how the world works. But in reality, uncertainty is better captured by the POMDP framework: there is something we don't directly observe, be it the robot's location or configuration, the location of objects in the world, or the parameters of the dynamics model itself—for example, where exactly is the center of mass of link two on this arm?

What we lose when we don't solve the POMDP is the ability to reason about *future information* the robot will get: in MDPs we only plan with what we know, not with what we *might* eventually know. Remember the value of information? Well, robots that plan using their current belief as if they will never find out anything more fail to account for the value of information. They will never take actions that seem suboptimal right now according to what they know, but that will actually result in a lot of information and enable the robot to do well.

What does such an action look like for a navigation robot? The robot could get close to a landmark to get a better estimate of where it is, even if that landmark is out of the way according to what it currently knows. This action is optimal only if the robot considers the new observations it will get, as opposed to looking only at the information it already has.

To get around this, robotics techniques sometimes define information gathering actions explicitly—such as moving a hand until it touches a surface (called **guarded movements**)—

Guarded movement

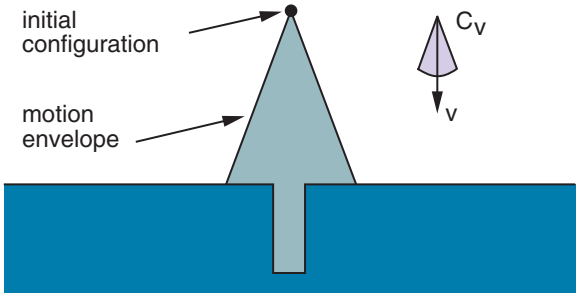


Figure 26.23 A two-dimensional environment, velocity uncertainty cone, and envelope of possible robot motions. The intended velocity is v, but with uncertainty the actual velocity could be anywhere in C_v, resulting in a final configuration somewhere in the motion envelope, which means we wouldn't know if we hit the hole or not.

and make sure the robot does that before coming up with a plan for reaching its actual goal. Each guarded motion consists of (1) a motion command and (2) a termination condition, which is a predicate on the robot's sensor values saying when to stop.

Sometimes, the goal itself could be reached via a sequence of guarded moves guaranteed to succeed regardless of uncertainty. As an example, Figure 26.23 shows a two-dimensional configuration space with a narrow vertical hole. It could be the configuration space for insertion of a rectangular peg into a hole or a car key into the ignition. The motion commands are constant velocities. The termination conditions are contact with a surface. To model uncertainty in control, we assume that instead of moving in the commanded direction, the robot's actual motion lies in the cone C_v about it.

The figure shows what would happen if the robot attempted to move straight down from the initial configuration. Because of the uncertainty in velocity, the robot could move anywhere in the conical envelope, possibly going into the hole, but more likely landing to one side of it. Because the robot would not then know which side of the hole it was on, it would not know which way to move.

A more sensible strategy is shown in Figures 26.24 and 26.25. In Figure 26.24, the robot deliberately moves to one side of the hole. The motion command is shown in the figure, and the termination test is contact with any surface. In Figure 26.25, a motion command is given that causes the robot to slide along the surface and into the hole. Because all possible velocities in the motion envelope are to the right, the robot will slide to the right whenever it is in contact with a horizontal surface.

It will slide down the right-hand vertical edge of the hole when it touches it, because all possible velocities are down relative to a vertical surface. It will keep moving until it reaches the bottom of the hole, because that is its termination condition. In spite of the control uncertainty, all possible trajectories of the robot terminate in contact with the bottom of the hole—that is, unless surface irregularities cause the robot to stick in one place.

Coastal navigation

Other techniques beyond guarded movements change the cost function to incentivize actions we know will lead to information—like the **coastal navigation** heuristic which requires the robot to stay near known landmarks. More generally, techniques can incorporate the expected **information gain** (reduction of entropy of the belief) as a term in the cost function,

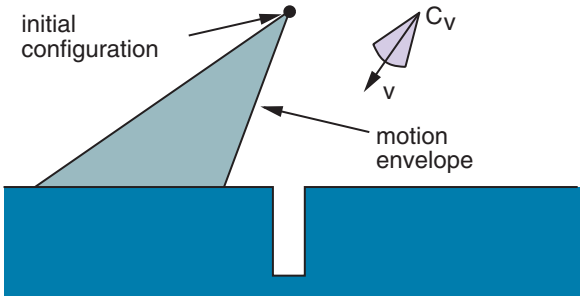


Figure 26.24 The first motion command and the resulting envelope of possible robot motions. No matter what actual motion ensues, we know the final configuration will be to the left of the hole.

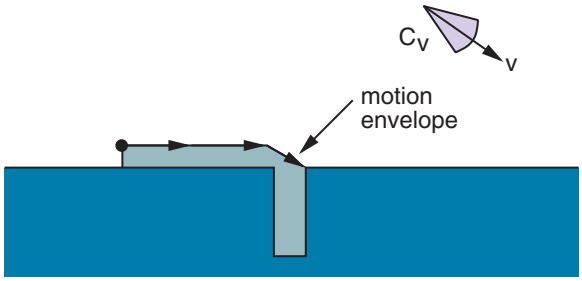


Figure 26.25 The second motion command and the envelope of possible motions. Even with error, we will eventually get into the hole.

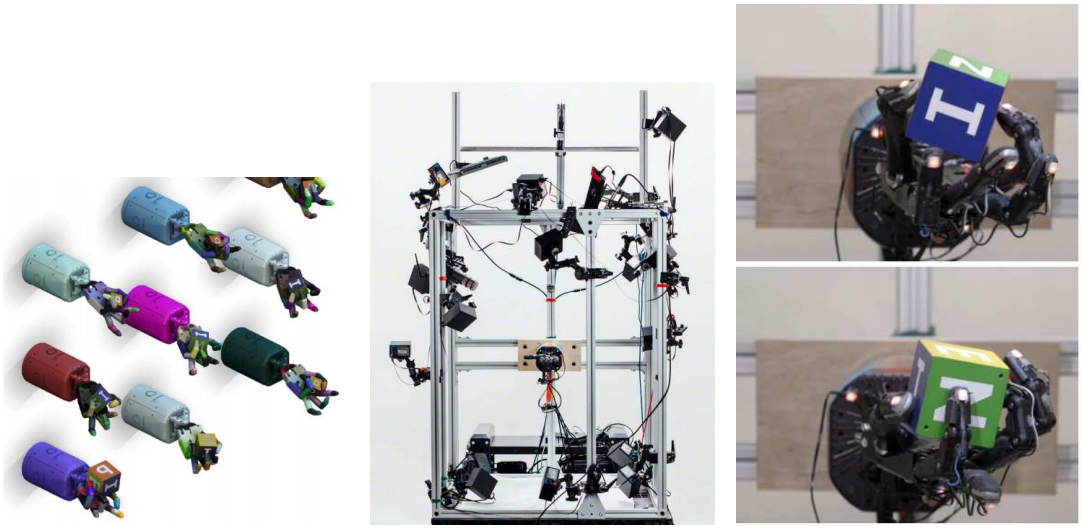
leading to the robot explicitly reasoning about how much information each action might bring when deciding what to do. While more difficult computationally, such approaches have the advantage that the robot invents its own information gathering actions rather than relying on human-provided heuristics and scripted strategies that often lack flexibility.

26.7 Reinforcement Learning in Robotics

Thus far we have considered tasks in which the robot has access to the dynamics model of the world. In many tasks, it is very difficult to write down such a model, which puts us in the domain of reinforcement learning (RL).

One challenge of RL in robotics is the continuous nature of the state and action spaces, which we handle either through discretization, or, more commonly, through function approximation. Policies or value functions are represented as combinations of known useful features, or as deep neural networks. Neural nets can map from raw inputs directly to outputs, and thus largely avoid the need for feature engineering, but they do require more data.

A bigger challenge is that robots operate in the real world. We have seen how reinforcement learning can be used to learn to play chess or Go by playing simulated games. But when a real robot moves in the real world, we have to make sure that its actions are safe (things



(a) (b) (c)

Figure 26.26 Training a robust policy. (a) Multiple simulations are run of a robot hand manipulating objects, with different randomized parameters for physics and lighting. Courtesy of Wojciech Zaremba. (b) The real-world environment, with a single robot hand in the center of a cage, surrounded by cameras and range finders. (c) Simulation and real-world training yields multiple different policies for grasping objects; here a pinch grasp and a quadpod grasp. Courtesy of OpenAI. See Andrychowicz *et al.* (2018a).

break!), and we have to accept that progress will be slower than in a simulation because the world refuses to move faster than one second per second. Much of what is interesting about using reinforcement learning in robotics boils down to how we might reduce the real world sample complexity—the number of interactions with the physical world that the robot needs before it has learned how to do the task.

26.7.1 Exploiting models

A natural way to avoid the need for many real-world samples is to use as much knowledge of the world's dynamics as possible. For instance, we might not know exactly what the coefficient of friction or the mass of an object is, but we might have equations that describe the dynamics as a function of these parameters.

In such a case, **model-based reinforcement learning** (Chapter 23) is appealing, where the robot can alternate between fitting the dynamics parameters and computing a better policy. Even if the equations are incorrect because they fail to model every detail of physics, researchers have experimented with learning an error term, in addition to the parameters, that can compensate for the inaccuracy of the physical model. Or, we can abandon the equations and instead fit locally linear models of the world that each approximate the dynamics in a region of the state space, an approach that has been successful in getting robots to master complex dynamic tasks like juggling.

Sim-to-real

A model of the world can also be useful in reducing the sample complexity of model-free reinforcement learning methods by doing **sim-to-real** transfer: transferring policies that work

in simulation to the real world. The idea is to use the model as a simulator for a policy search (Section 23.5). To learn a policy that transfers well, we can add noise to the model during training, thereby making the policy more robust. Or, we can train policies that will work with a *variety* of models by sampling different parameters in the simulations—sometimes referred to as **domain randomization**. An example is in Figure 26.26, where a dexterous manipulation task is trained in simulation by varying visual attributes, as well as physical attributes like friction or damping.

Domain randomization

Finally, hybrid approaches that borrow ideas from both model-based and model-free algorithms are meant to give us the best of both. The hybrid approach originated with the Dyna architecture, where the idea was to iterate between acting and improving the policy, but the policy improvement would come in two complementary ways: 1) the standard model-free way of using the experience to directly update the policy, and 2) the model-based way of using the experience to fit a model, then plan with it to generate a policy.

More recent techniques have experimented with fitting local models, planning with them to generate actions, and using these actions as supervision to fit a policy, then iterating to get better and better models around the areas that the policy needs. This has been successfully applied in **end-to-end learning**, where the policy takes pixels as input and directly outputs torques as actions—it enabled the first demonstration of deep RL on physical robots.

Models can also be exploited for the purpose of ensuring **safe exploration**. Learning slowly but safely may be better than learning quickly but crashing and burning half way through. So arguably, more important than reducing real-world samples is reducing real-world samples in *dangerous* states—we don't want robots falling off cliffs, and we don't want them breaking our favorite mugs or, even worse, colliding with objects and people. An approximate model, with uncertainty associated to it (for example by considering a range of values for its parameters), can guide exploration and impose constraints on the actions that the robot is allowed to take in order to avoid these dangerous states. This is an active area of research in robotics and control.

26.7.2 Exploiting other information

Models are useful, but there is more we can do to further reduce sample complexity.

When setting up a reinforcement learning problem, we have to select the state and action spaces, the representation of the policy or value function, and the reward function we're using. These decisions have a large impact on how easy or how hard we are making the problem.

One approach is to use higher-level **motion primitives** instead of low-level actions like torque commands. A motion primitive is a parameterized skill that the robot has. For example, a robotic soccer player might have the skill of "pass the ball to the player at (x, y)." All the policy needs to do is to figure out how to combine them and set their parameters, instead of reinventing them. This approach often learns much faster than low-level approaches, but does restrict the space of possible behaviors that the robot can learn.

Motion primitive

Another way to reduce the number of real-world samples required for learning is to reuse information from previous learning episodes on other tasks, rather than starting from scratch. This falls under the umbrella of **metalearning** or **transfer learning**.

Finally, people are a great source of information. In the next section, we talk about how to interact with people, and part of it is how to use their actions to guide the robot's learning.

26.8 Humans and Robots

Thus far, we've focused on a robot planning and learning how to act *in isolation*. This is useful for some robots, like the rovers we send out to explore distant planets on our behalf. But, for the most part, we do not build robots to work in isolation. We build them to help us, and to work in human environments, around and with us.

This raises two complementary challenges. First is optimizing reward when there are people acting in the same environment as the robot. We call this the **coordination problem** (see Section 17.1). When the robot's reward depends on not just its own actions, but also the actions that people take, the robot has to choose its actions in a way that meshes well with theirs. When the human and the robot are on the same team, this turns into **collaboration**.

Second is the challenge of optimizing for what people actually want. If a robot is to help people, its reward function needs to incentivize the actions that people want the robot to execute. Figuring out the right reward function (or policy) for the robot is itself an interaction problem. We will explore these two challenges in turn.

26.8.1 Coordination

Let's assume for now, as we have been, that the robot has access to a clearly defined reward function. But, instead of needing to optimize it in isolation, now the robot needs to optimize it around a human who is also acting. For example, as an autonomous car merges on the highway, it needs to negotiate the maneuver with the human driver coming in the target lane—should it accelerate and merge in front, or slow down and merge behind? Later, as it pulls to a stop sign, preparing to take a right, it has to watch out for the cyclist in the bicycle lane, and for the pedestrian about to step onto the crosswalk.

Or, consider a mobile robot in a hallway. Someone heading straight toward the robot steps slightly to the right, indicating which side of the robot they want to pass on. The robot has to respond, clarifying its intentions.

Humans as approximately rational agents

One way to formulate coordination with a human is to model it as a game between the robot and the human (Section 17.2). With this approach, we explicitly make the assumption that people are agents incentivized by objectives. This does not automatically mean that they are perfectly rational agents (i.e., find optimal solutions in the game), but it does mean that the robot can structure the way it reasons about the human via the notion of possible objectives that the human might have. In this game:

- the state of the environment captures the configurations of both the robot and human agents; call it $x = (x_R, x_H)$;
- each agent can take actions, u_R and u_H respectively;
- each agent has an objective that can be represented as a cost, J_R and J_H: each agent wants to get to its goal safely and efficiently;
- and, as in any game, each objective depends on the state and on the actions of *both* agents: $J_R(x, u_R, u_H)$ and $J_H(x, u_H, u_R)$. Think of the car-pedestrian interaction—the car should stop if the pedestrian crosses, and should go forward if the pedestrian waits.

Three important aspects complicate this game. First is that the human and the robot don't necessarily know each other's objectives. This makes it an **incomplete information game**.

Incomplete information game

Second is that the state and action spaces are *continuous*, as they've been throughout this chapter. We learned in Chapter 6 how to do tree search to tackle discrete games, but how do we tackle continuous spaces?

Third, even though at the high level the game model makes sense—humans do move, and they do have objectives—a human's behavior might not always be well-characterized as a solution to the game. The game comes with a computational challenge not only for the robot, but for us humans too. It requires thinking about what the robot will do in response to what the person does, which depends on what the robot thinks the person will do, and pretty soon we get to "what do you think I think you think I think"— it's turtles all the way down! Humans can't deal with all of that, and exhibit certain suboptimalities. This means that the robot should account for these suboptimalities.

So, then, what is an autonomous car to do when the coordination problem is this hard? We will do something similar to what we've done before in this chapter. For motion planning and control, we took an MDP and broke it up into planning a trajectory and then tracking it with a controller. Here too, we will take the game, and break it up into making predictions about human actions, and deciding what the robot should do given these predictions.

Predicting human action

Predicting human actions is hard because they depend on the robot's actions and vice versa. One trick that robots use is to pretend the person is ignoring the robot. The robot assumes people are noisily optimal with respect to their objective, which is unknown to the robot and is modeled as no longer dependent on the robot's actions: $J_H(x, u_H)$. In particular, the higher the value of an action for the objective (the lower the cost to go), the more likely the human is to take it. The robot can create a model for $P(u_H \mid x, J_H)$, for instance using the softmax function from page 862:

$$P(u_H \mid x, J_H) \propto e^{-Q(x,u_H;J_H)} \tag{26.8}$$

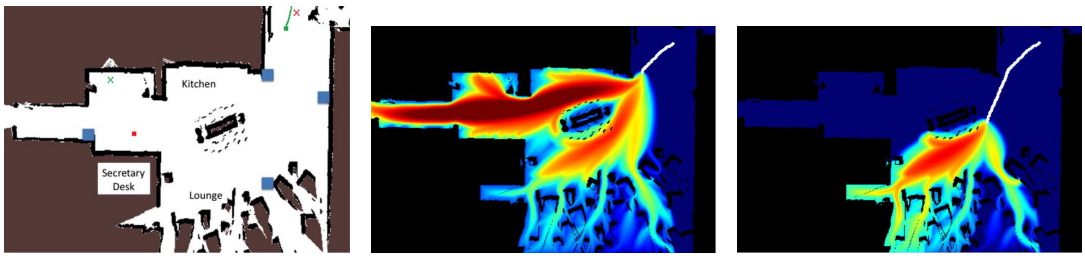
with $Q(x, u_H; J_H)$ the Q-value function corresponding to J_H (the negative sign is there because in robotics we like to minimize cost, not maximize reward). Note that the robot does not assume perfectly optimal actions, nor does it assume that the actions are chosen based on reasoning about the robot at all.

Armed with this model, the robot uses the human's ongoing actions as evidence about J_H. If we have an observation model for how human actions depend on the human's objective, each human action can be incorporated to update the robot's belief over what objective the person has:

$$b'(J_H) \propto b(J_H) P(u_H \mid x, J_H)\,. \tag{26.9}$$

An example is in Figure 26.27: the robot is tracking a human's location and as the human moves, the robot updates its belief over human goals. As the human heads toward the windows, the robot increases the probability that the goal is to look out the window, and decreases the probability that the goal is going to the kitchen, which is in the other direction.

This is how the human's past actions end up informing the robot about what the human will do in the future. Having a belief about the human's goal helps the robot anticipate what next actions the human will take. The heatmap in the figure shows the robot's future predictions: red is most probable; blue least probable.



(a) (b) (c)

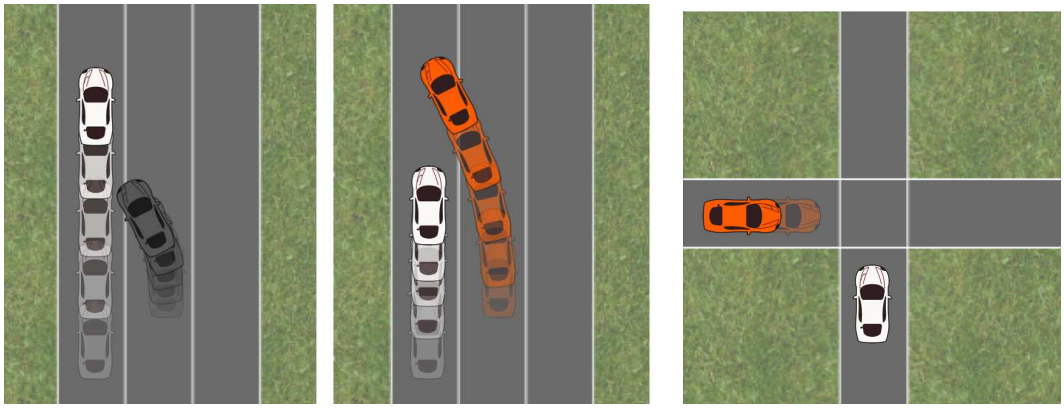
Figure 26.27 Making predictions by assuming that people are noisily rational given their goal: the robot uses the past actions to update a belief over what goal the person is heading to, and then uses the belief to make predictions about future actions. (a) The map of a room. (b) Predictions after seeing a small part of the person's trajectory (white path); (c) Predictions after seeing more human actions: the robot now knows that the person is not heading to the hallway on the left, because the path taken so far would be a poor path if that were the person's goal. Images courtesy of Brian D. Ziebart. See Ziebart *et al.* (2009).

The same can happen in driving. We might not know how much another driver values efficiency, but if we see them accelerate as someone is trying to merge in front of them, we now know a bit more about them. And once we know that, we can better anticipate what they will do in the future—the same driver is likely to come closer behind us, or weave through traffic to get ahead.

Once the robot can make predictions about human future actions, it has reduced its problem to solving an MDP. The human actions complicate the transition function, but as long as the robot can anticipate what action the person will take from any future state, the robot can calculate $P(x' \mid x, u_R)$: it can compute $P(u_H \mid x)$ from $P(u_H \mid x, J_H)$ by marginalizing over J_H, and combine it with $P(x' \mid x, u_R, u_H)$, the transition (dynamics) function for how the world updates based on both the robot's and the human's actions. In Section 26.5 we focused on how to solve this in continuous state and action spaces for deterministic dynamics, and in Section 26.6 we discussed doing it with stochastic dynamics and uncertainty.

Splitting prediction from action makes it easier for the robot to handle interaction, but sacrifices performance much as splitting estimation from motion did, or splitting planning from control.

A robot with this split no longer understands that its actions can influence what people end up doing. In contrast, the robot in Figure 26.27 anticipates where people will go and then optimizes for reaching its own goal and avoiding collisions with them. In Figure 26.28, we have an autonomous car merging on the highway. If it just planned in reaction to other cars, it might have to wait a long time while other cars occupy its target lane. In contrast, a car that reasons about prediction and action jointly knows that different actions it could take will result in different reactions from the human. If it starts to assert itself, the other cars are likely to slow down a bit and make room. Roboticists are working towards coordinated interactions like this so robots can work better with humans.



(a) (b)

Figure 26.28 (a) Left: An autonomous car (middle lane) predicts that the human driver (left lane) wants to keep going forward, and plans a trajectory that slows down and merges behind. Right: The car accounts for the influence its actions can have on human actions, and realizes it can merge in front and rely on the human driver to slow down. (b) That same algorithm produces an unusual strategy at an intersection: the car realizes that it can make it more likely for the person (bottom) to proceed faster through the intersection by starting to inch backwards. Images courtesy of Anca Dragan. See Sadigh *et al.* (2016).

Human predictions about the robot

Incomplete information is often two-sided: the robot does not know the human's objective and the human, in turn, does not know the *robot*'s objective—people need to be making predictions about robots. As robot designers, we are not in charge of how the human makes predictions; we can only control what the robot does. However, the robot can act in a way to make it *easier* for the human to make correct predictions. The robot can assume that the human is using something roughly analogous to Equation (26.8) to estimate the robot's objective J_R, and thus the robot will act so that its true objective can be easily inferred.

A special case of the game is when the human and the robot are on the same team, working toward the same goal or objective: $J_H = J_R$. Imagine getting a personal home robot that is helping you make dinner or clean up—these are examples of **collaboration**.

We can now define a **joint agent** whose actions are tuples of human–robot actions, (u_H, u_R) and who optimizes for $J_H(x, u_H, u_R) = J_R(x, u_R, u_H)$, and we're solving a regular planning problem. We compute the optimal plan or policy for the joint agent, and voila, we now know what the robot and human should do.

Joint agent

This would work really well if people were perfectly optimal. The robot would do its part of the joint plan, the human theirs. Unfortunately, in practice, people don't seem to follow the perfectly laid out joint-agent plan; they have a mind of their own! We've already learned one way to handle this though, back in Section 26.6. We called it **model predictive control (MPC)**: the idea was to come up with a plan, execute the first action, and then replan. That way, the robot always adapts its plan to what the human is actually doing.

Let's work through an example. Suppose you and the robot are in your kitchen, and have decided to make waffles. You are slightly closer to the fridge, so the optimal joint plan would

have you grab the eggs and milk from the fridge, while the robot fetches the flour from the cabinet. The robot knows this because it can measure quite precisely where everyone is. But suppose you start heading for the flour cabinet. You are going against the optimal joint plan. Rather than sticking to it and stubbornly also going for the flour, the MPC robot recalculates the optimal plan, and now that you are close enough to the flour it is best for the robot to grab the waffle iron instead.

If we know that people might deviate from optimality, we can account for it ahead of time. In our example, the robot can try to anticipate that you are going for the flour the moment you take your first step (say, using the prediction technique above). Even if it is still technically optimal for you to turn around and head for the fridge, the robot should not assume that's what is going to happen. Instead, the robot can compute a plan in which you keep doing what you seem to want.

Humans as black box agents

We don't have to treat people as objective-driven, intentional agents to get robots to coordinate with us. An alternative model is that the human is merely some agent whose policy π_H "messes" with the environment dynamics. The robot does not know π_H, but can model the problem as needing to act in an MDP with unknown dynamics. We have seen this before: for general agents in Chapter 23, and for robots in particular in Section 26.7.

The robot can fit a policy model π_H to human data, and use it to compute an optimal policy for itself. Due to scarcity of data, this has been mostly used so far at the task level. For instance, robots have learned through interaction what actions people tend to take (in response to its own actions) for the task of placing and drilling screws in an industrial assembly task.

Then there is also the model-free reinforcement learning alternative: the robot can start with some initial policy or value function, and keep improving it over time via trial and error.

26.8.2 Learning to do what humans want

Another way interaction with humans comes into robotics is in J_R itself—the robot's cost or reward function. The framework of rational agents and the associated algorithms reduce the problem of generating good behavior to specifying a good reward function. But for robots, as for many other AI agents, getting the cost right is still difficult.

Take autonomous cars: we want them to reach the destination, to be safe, to drive comfortably for their passengers, to obey traffic laws, etc. A designer of such a system needs to trade off these different components of the cost function. The designer's task is hard because robots are built to help end users, and not every end user is the same. We all have different preferences for how aggressively we want our car to drive, etc.

Below, we explore two alternatives for trying to get robot behavior to match what we actually want the robot to do. The first is to learn a cost function from human input. The second is to bypass the cost function and imitate human demonstrations of the task.

Preference learning: Learning cost functions

Imagine that an end user is showing a robot how to do a task. For instance, they are driving the car in the way they would like it to be driven by the robot. Can you think of a way for the robot to use these actions—we call them "demonstrations"—to figure out what cost function it should optimize?

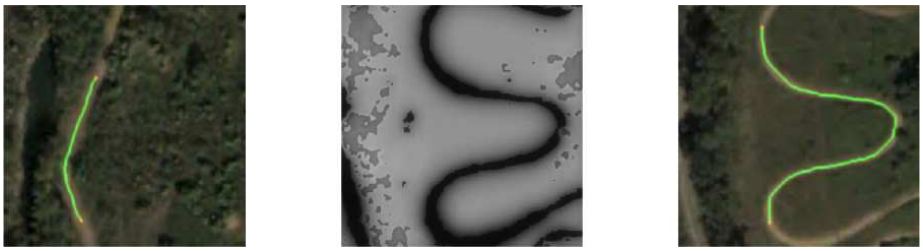


Figure 26.29 Left: A mobile robot is shown a demonstration that stays on the dirt road. Middle: The robot infers the desired cost function, and uses it in a new scene, knowing to put lower cost on the road there. Right: The robot plans a path for the new scene that also stays on the road, reproducing the preferences behind the demonstration. Images courtesy of Nathan Ratliff and James A. Bagnell. See Ratliff *et al.* (2006).

We have actually already seen the answer to this back in Section 26.8.1. There, the setup was a little different: we had another person taking actions in the same space as the robot, and the robot needed to predict what the person would do. But one technique we went over for making these predictions was to assume that people act to noisily optimize some cost function J_H, and we can use their ongoing actions as evidence about what cost function that is. We can do the same here, except not for the purpose of predicting human behavior in the future, but rather acquiring the cost function the robot itself should optimize. If the person drives defensively, the cost function that will explain their actions will put a lot of weight on safety and less so on efficiency. The robot can adopt this cost function as its own and optimize it when driving the car itself.

Roboticists have experimented with different algorithms for making this cost inference computationally tractable. In Figure 26.29, we see an example of teaching a robot to prefer staying on the road to going over the grassy terrain. Traditionally in such methods, the cost function has been represented as a combination of hand-crafted features, but recent work has also studied how to represent it using a deep neural network, without feature engineering.

There are other ways for a person to provide input. A person could use language rather than demonstration to instruct the robot. A person could act as a critic, watching the robot perform a task one way (or two ways) and then saying how well the task was done (or which way was better), or giving advice on how to improve.

Learning policies directly via imitation

An alternative is to bypass cost functions and learn the desired robot *policy* directly. In our car example, the human's demonstrations make for a convenient data set of states labeled by the action the robot should take at each state: $\mathcal{D} = \{(x_i, u_i)\}$. The robot can run supervised learning to fit a policy $\pi : x \mapsto u$, and execute that policy. This is called **imitation learning** or **behavioral cloning**.

Behavioral cloning

A challenge with this approach is in **generalization** to new states. The robot does not know why the actions in its database have been marked as optimal. It has no causal rule; all it can do is run a supervised learning algorithm to try to learn a policy that will generalize to unknown states. However, there is no guarantee that the generalization will be correct.

Generalization

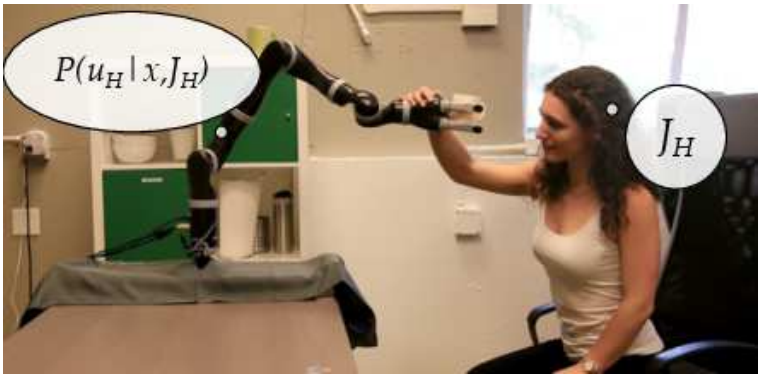


Figure 26.30 A human teacher pushes the robot down to teach it to stay closer to the table. The robot appropriately updates its understanding of the desired cost function and starts optimizing it. Courtesy of Anca Dragan. See Bajcsy *et al.* (2017).

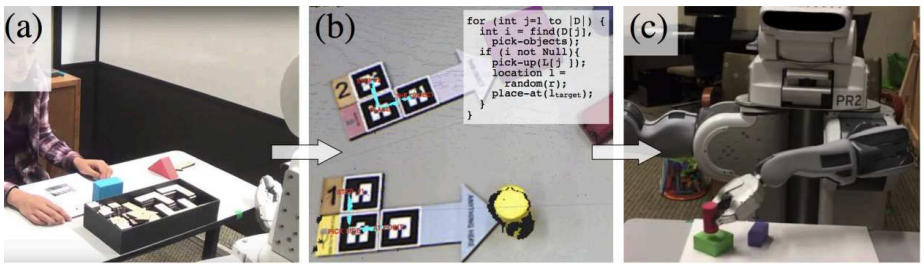


Figure 26.31 A programming interface that involves placing specially designed blocks in the robot's workspace to select objects and specify high-level actions. Images courtesy of Maya Cakmak. See Sefidgar *et al.* (2017).

The ALVINN autonomous car project used this approach, and found that even when starting from a state in $\mathcal{D}$, π will make small errors, which will take the car off the demonstrated trajectory. There, π will make a larger error, which will take the car even further off the desired course.

We can address this at training time if we interleave collecting labels and learning: start with a demonstration, learn a policy, then roll out that policy and ask the human for what action to take at every state along the way, then repeat. The robot then learns how to correct its mistakes as it deviates from the human's desired actions.

Alternatively, we can address it by leveraging reinforcement learning. The robot can fit a dynamics model based on the demonstrations, and then use optimal control (Section 26.5.4) to generate a policy that optimizes for staying close to the demonstration. A version of this has been used to perform very challenging maneuvers at an expert level in a small radio-controlled helicopter (see Figure 23.9(b)).

The DAGGER (Data Aggregation) system starts with a human expert demonstration. From that it learns a policy, π_1 and uses the policy to generate a data set $\mathcal{D}$. Then from $\mathcal{D}$ it generates a new policy π_2 that best imitates the original human data. This repeats, and

on the nth iteration it uses π_n to generate more data, to be added to $\mathcal{D}$, which is then used to create π_{n+1}. In other words, at each iteration the system gathers new data under the current policy and trains the next policy using all the data gathered so far.

Related recent techniques use **adversarial training**: they alternate between training a classifier to distinguish between the robot's learned policy and the human's demonstrations, and training a new robot policy via reinforcement learning to fool the classifier. These advances enable the robot to handle states that are near demonstrations, but generalization to far-off states or to new dynamics is a work in progress.

Teaching interfaces and the correspondence problem. So far, we have imagined the case of an autonomous car or an autonomous helicopter, for which human demonstrations use the same actions that the robot can take itself: accelerating, braking, and steering. But what happens if we do this for tasks like cleaning up the kitchen table? We have two choices here: either the person demonstrates using their own body while the robot watches, or the person physically guides the robot's effectors.

Correspondence problem

The first approach is appealing because it comes naturally to end users. Unfortunately, it suffers from the **correspondence problem**: how to map human actions onto robot actions. People have different kinematics and dynamics than robots. Not only does that make it difficult to *translate* or *retarget* human motion onto robot motion (e.g., retargeting a five-finger human grasp to a two-finger robot grasp), but often the high-level strategy a person might use is not appropriate for the robot.

Kinesthetic teaching

Keyframe

Visual programming

The second approach, where the human teacher moves the robot's effectors into the right positions, is called **kinesthetic teaching**. It is not easy for humans to teach this way, especially to teach robots with multiple joints. The teacher needs to coordinate all the degrees of freedom as it is guiding the arm through the task. Researchers have thus investigated alternatives, like demonstrating **keyframes** as opposed to continuous trajectories, as well as the use of **visual programming** to enable end users to program primitives for a task rather than demonstrate from scratch (Figure 26.31). Sometimes both approaches are combined.

26.9 Alternative Robotic Frameworks

Deliberative

Reactive

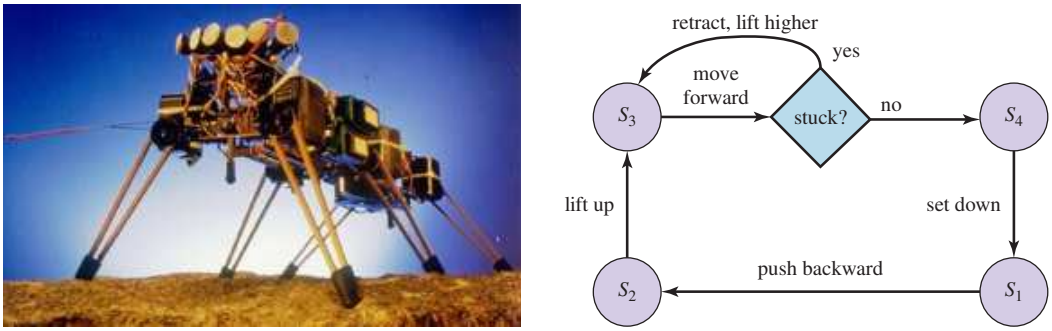
Thus far, we have taken a view of robotics based on the notion of defining or learning a reward function, and having the robot optimize that reward function (be it via planning or learning), sometimes in coordination or collaboration with humans. This is a **deliberative** view of robotics, to be contrasted with a **reactive** view.

26.9.1 Reactive controllers

In some cases, it is easier to set up a good policy for a robot than to model the world and plan. Then, instead of a *rational* agent, we have a *reflex* agent.

For example, picture a legged robot that attempts to lift a leg over an obstacle. We could give this robot a rule that says lift the leg a small height h and move it forward, and if the leg encounters an obstacle, move it back and start again at a higher height. You could say that h is modeling an aspect of the world, but we can also think of h as an auxiliary variable of the robot controller, devoid of direct physical meaning.

One such example is the six-legged (hexapod) robot, shown in Figure 26.32(a), designed for walking through rough terrain. The robot's sensors are inadequate to obtain accurate



(a) (b)

Figure 26.32 (a) Genghis, a hexapod robot. (Image courtesy of Rodney A. Brooks.) (b) An augmented finite state machine (AFSM) that controls one leg. The AFSM reacts to sensor feedback: if a leg is stuck during the forward swinging phase, it will be lifted increasingly higher.

models of the terrain for path planning. Moreover, even if we added high-precision cameras and rangefinders, the 12 degrees of freedom (two for each leg) would render the resulting path planning problem computationally difficult.

It is possible, nonetheless, to specify a controller directly without an explicit environmental model. (We have already seen this with the PD controller, which was able to keep a complex robot arm on target *without* an explicit model of the robot dynamics.)

Gait

For the hexapod robot we first choose a **gait**, or pattern of movement of the limbs. One statically stable gait is to first move the right front, right rear, and left center legs forward (keeping the other three fixed), and then move the other three. This gait works well on flat terrain. On rugged terrain, obstacles may prevent a leg from swinging forward. This problem can be overcome by a remarkably simple control rule: *when a leg's forward motion is blocked, simply retract it, lift it higher, and try again.* The resulting controller is shown in Figure 26.32(b) as a simple finite state machine; it constitutes a reflex agent with state, where the internal state is represented by the index of the current machine state (s_1 through s_4).

26.9.2 Subsumption architectures

Subsumption architecture

The **subsumption architecture** (Brooks, 1986) is a framework for assembling reactive controllers out of finite state machines. Nodes in these machines may contain tests for certain sensor variables, in which case the execution trace of a finite state machine is conditioned on the outcome of such a test. Arcs can be tagged with messages that will be generated when traversing them, and that are sent to the robot's motors or to other finite state machines. Additionally, finite state machines possess internal timers (clocks) that control the time it takes to traverse an arc. The resulting machines are called **augmented finite state machines (AFSMs)**, where the augmentation refers to the use of clocks.

Augmented finite state machine (AFSM)

An example of a simple AFSM is the four-state machine we just talked about, shown in Figure 26.32(b). This AFSM implements a cyclic controller, whose execution mostly does not rely on environmental feedback. The forward swing phase, however, does rely on sensor feedback. If the leg is stuck, meaning that it has failed to execute the forward swing, the

robot retracts the leg, lifts it up a little higher, and attempts to execute the forward swing once again. Thus, the controller is able to *react* to contingencies arising from the interplay of the robot and its environment.

The subsumption architecture offers additional primitives for synchronizing AFSMs, and for combining output values of multiple, possibly conflicting AFSMs. In this way, it enables the programmer to compose increasingly complex controllers in a bottom-up fashion. In our example, we might begin with AFSMs for individual legs, followed by an AFSM for coordinating multiple legs. On top of this, we might implement higher-level behaviors such as collision avoidance, which might involve backing up and turning.

The idea of composing robot controllers from AFSMs is quite intriguing. Imagine how difficult it would be to generate the same behavior with any of the configuration-space path-planning algorithms described in the previous section. First, we would need an accurate model of the terrain. The configuration space of a robot with six legs, each of which is driven by two independent motors, totals 18 dimensions (12 dimensions for the configuration of the legs, and six for the location and orientation of the robot relative to its environment). Even if our computers were fast enough to find paths in such high-dimensional spaces, we would have to worry about nasty effects such as the robot sliding down a slope.

Because of such stochastic effects, a single path through configuration space would almost certainly be too brittle, and even a PID controller might not be able to cope with such contingencies. In other words, generating motion behavior deliberately is simply too complex a problem in some cases for present-day robot motion planning algorithms.

Unfortunately, the subsumption architecture has its own problems. First, the AFSMs are driven by raw sensor input, an arrangement that works if the sensor data is reliable and contains all necessary information for decision making, but fails if sensor data has to be integrated in nontrivial ways over time. Subsumption-style controllers have therefore mostly been applied to simple tasks, such as following a wall or moving toward visible light sources.

Second, the lack of deliberation makes it difficult to change the robot's goals. A robot with a subsumption architecture usually does just one task, and it has no notion of how to modify its controls to accommodate different goals (just like the dung beetle on page 59).

Third, in many real-world problems, the policy we want is often too complex to encode explicitly. Think about the example from Figure 26.28, of an autonomous car needing to negotiate a lane change with a human driver. We might start off with a simple policy that goes into the target lane. But when we test the car, we find out that not every driver in the target lane will slow down to let the car in. We might then add a bit more complexity: make the car nudge towards the target lane, wait for a response form the driver in that lane, and then either proceed or retreat back. But then we test the car, and realize that the nudging needs to happen at a different speed depending on the speed of the vehicle in the target lane, on whether there is another vehicle in front in the target lane, on whether there is a vehicle behind the car in the initial, and so on. The number of conditions that we need to consider to determine the right course of action can be very large, even for such a deceptively simple maneuver. This in turn presents scalability challenges for subsumption-style architectures.

All that said, robotics is a complex problem with many approaches: deliberative, reactive, or a mixture thereof; based on physics, cognitive models, data, or a mixture thereof. The right approach is still a subject for debate, scientific inquiry, and engineering prowess.

(a) (b)

Figure 26.33 (a) A patient with a brain–machine interface controlling a robot arm to grab a drink. Image courtesy of Brown University. (b) Roomba, the robot vacuum cleaner. Photo by HANDOUT/KRT/Newscom.

26.10 Application Domains

Robotic technology is already permeating our world, and has the potential to improve our independence, health, and productivity. Here are some example applications.

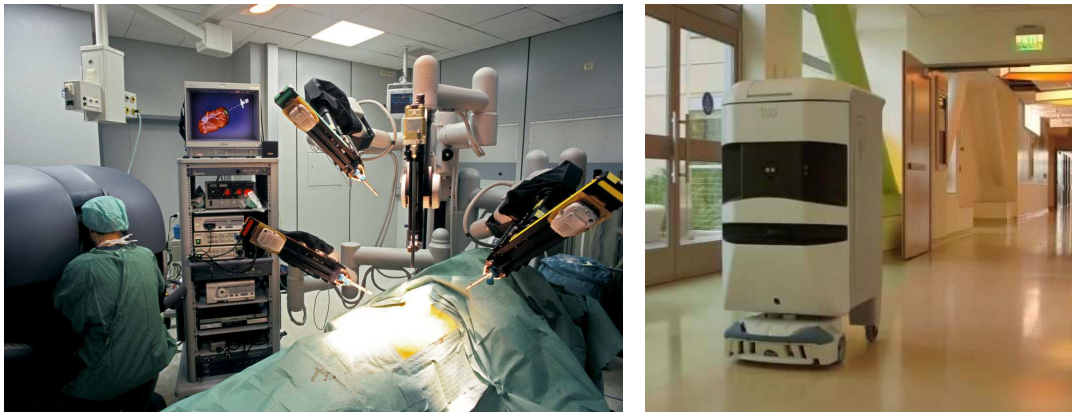
Home care: Robots have started to enter the home to care for older adults and people with motor impairments, assisting them with activities of daily living and enabling them to live more independently. These include wheelchairs and wheelchair-mounted arms like the Kinova arm from Figure 26.1(b). Even though they start off as being operated by a human directly, these robots are gaining more and more autonomy. On the horizon are robots operated by **brain–machine interfaces**, which have been shown to enable people with quadriplegia to use a robot arm to grasp objects and even feed themselves (Figure 26.33(a)). Related to these are prosthetic limbs that intelligently respond to our actions, and exoskeletons that give us superhuman strength or enable people who can't control their muscles from the waist down to walk again.

Personal robots are meant to assist us with daily tasks like cleaning and organizing, freeing up our time. Although manipulation still has a way to go before it can operate seamlessly in messy, unstructured human environments, navigation has made some headway. In particular, many homes already enjoy a mobile robot vacuum cleaner like the one in Figure 26.33(b).

Health care: Robots assist and augment surgeons, enabling more precise, minimally invasive, safer procedures with better patient outcomes. The Da Vinci surgical robot from Figure 26.34(a) is now widely deployed at hospitals in the U.S.

Services: Mobile robots help out in office buildings, hotels, and hospitals. Savioke has put robots in hotels delivering products like towels or toothpaste to your room. The Helpmate and TUG robots carry food and medicine in hospitals (Figure 26.34(b)), while Diligent Robotics' Moxi robot helps out nurses with back-end logistical responsibilities. Co-Bot roams the halls of Carnegie Mellon University, ready to guide you to someone's office. We can also use **telepresence robots** like the Beam to attend meetings and conferences remotely, or check in on our grandparents.

Telepresence robots



(a) (b)

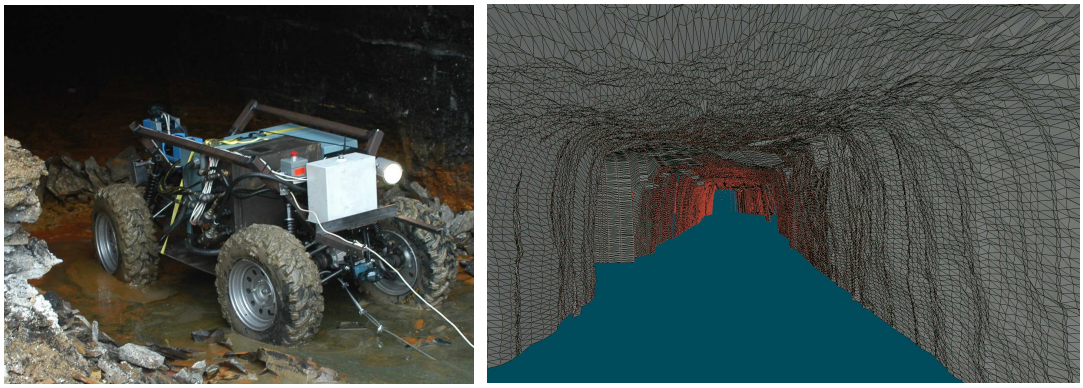
Figure 26.34 (a) Surgical robot in the operating room. Photo by Patrick Landmann/Science Source. (b) Hospital delivery robot. Photo by Wired.

(a) (b)

Figure 26.35 (a) Autonomous car BOSS which won the DARPA Urban Challenge. Photo by Tangi Quemener/AFP/Getty Images/Newscom. Courtesy of Sebastian Thrun. (b) Aerial view showing the perception and predictions of the Waymo autonomous car (white vehicle with green track). Other vehicles (blue boxes) and pedestrians (orange boxes) are shown with anticipated trajectories. Road/sidewalk boundaries are in yellow. Photo courtesy of Waymo.

Autonomous cars: Some of us are occasionally distracted while driving, by cell phone calls, texts, or other distractions. The sad result: more than a million people die every year in traffic accidents. Further, many of us spend a lot of time driving and would like to recapture some of that time. All this has led to a massive ongoing effort to deploy autonomous cars.

Prototypes have existed since the 1980s, but progress was stimulated by the 2005 DARPA Grand Challenge, an autonomous vehicle race over 200 challenging kilometers of unrehearsed desert terrain. Stanford's Stanley vehicle completed the course in less than seven hours, winning a $2 million prize and a place in the National Museum of American History.



(a) (b)

Figure 26.36 (a) A robot mapping an abandoned coal mine. (b) A 3D map of the mine acquired by the robot. Courtesy of Sebastian Thrun.

Figure 26.35(a) depicts BOSS, which in 2007 won the DARPA Urban Challenge, a complicated road race on city streets where robots faced other robots and had to obey traffic rules.

In 2009, Google started an autonomous driving project (featuring many of the researchers who had worked on Stanley and BOSS), which has now spun off as Waymo. In 2018 Waymo started driverless testing (with nobody in the driver seat) in the suburbs of Pheonix, Arizona. In the meantime, other autonomous driving companies and ride-sharing companies are working on developing their own technology, while car manufacturers have been selling cars with more and more assistive intelligence, such as Tesla's **driver assist**, which is meant for highway driving. Other companies are targeting non-highway driving applications including college campuses and retirement communities. Still other companies are focused on non-passenger applications such as trucking, grocery delivery, and valet parking.

Driver assist

Animatronics

Autonomatronics

Entertainment: Disney has been using robots (under the name **animatronics**) in their parks since 1963. Originally, these robots were restricted to hand-designed, open-loop, unvarying motion (and speech), but since 2009 a version called **autonomatronics** can generate autonomous actions. Robots also take the form of intelligent toys for children; for example, Anki's Cozmo plays games with children and may pound the table with frustration when it loses. Finally, quadrotors like Skydio's R1 from Figure 26.2(b) act as personal photographers and videographers, following us around to take action shots as we ski or bike.

Exploration and hazardous environments: Robots have gone where no human has gone before, including the surface of Mars. Robotic arms assist astronauts in deploying and retrieving satellites and in building the International Space Station. Robots also help explore under the sea. They are routinely used to acquire maps of sunken ships. Figure 26.36 shows a robot mapping an abandoned coal mine, along with a 3D model of the mine acquired using range sensors. In 1996, a team of researches released a legged robot into the crater of an active volcano to acquire data for climate research. Robots are becoming very effective tools for gathering information in domains that are difficult (or dangerous) for people to access.

Robots have assisted people in cleaning up nuclear waste, most notably in Three Mile Island, Chernobyl, and Fukushima. Robots were present after the collapse of the World Trade

Center, where they entered structures deemed too dangerous for human search and rescue crews. Here too, these robots are initially deployed via teleoperation, and as technology advances they are becoming more and more autonomous, with a human operator in charge but not having to specify every single command.

Industry: The majority of robots today are deployed in factories, automating tasks that are difficult, dangerous, or dull for humans. (The majority of factory robots are in automobile factories.) Automating these tasks is a positive in terms of efficiently producing what society needs. At the same time, it also means displacing some human workers from their jobs. This has important policy and economics implications—the need for retraining and education, the need for a fair division of resources, etc. These topics are discussed further in Section 28.3.5.

Summary

Robotics is about physically embodied agents, which can change the state of the physical world. In this chapter, we have learned the following:

- The most common types of robots are **manipulators** (robot arms) and **mobile robots**. They have **sensors** for perceiving the world and **actuators** that produce motion, which then affects the world via **effectors**.
- The general robotics problem involves **stochasticity** (which can be handled by MDPs), **partial observability** (which can be handled by POMDPs), and acting with and around **other agents** (which can be handled with game theory). The problem is made even harder by the fact that most robots work in continuous and high-dimensional state and action spaces. They also operate in the real world, which refuses to run faster than real time and in which failures lead to real things being damaged, with no "undo" capability.
- Ideally, the robot would solve the entire problem in one go: observations in the form of raw sensor feeds go in, and actions in the form of torques or currents to the motors come out. In practice though, this is too daunting, and roboticists typically decouple different aspects of the problem and treat them independently.
- We typically separate perception (estimation) from action (motion generation). Perception in robotics involves computer vision to recognize the surroundings through cameras, but also localization and mapping.
- Robotic perception concerns itself with estimating decision-relevant quantities from sensor data. To do so, we need an internal representation and a method for updating this internal representation over time.
- **Probabilistic filtering algorithms** such as particle filters and Kalman filters are useful for robot perception. These techniques maintain the belief state, a posterior distribution over state variables.
- For generating motion, we use **configuration spaces**, where a point specifies everything we need to know to locate every **body point** on the robot. For instance, for a robot arm with two joints, a configuration consists of the two joint angles.
- We typically decouple the motion generation problem into **motion planning**, concerned with producing a plan, and **trajectory tracking control**, concerned with producing a policy for control inputs (actuator commands) that results in executing the plan.

- Motion planning can be solved via graph search using **cell decomposition**; using **randomized motion planning** algorithms, which sample milestones in the continuous configuration space; or using **trajectory optimization**, which can iteratively push a straight-line path out of collision by leveraging a **signed distance field**.
- A path found by a search algorithm can be executed using the path as the reference trajectory for a **PID controller**, which constantly corrects for errors between where the robot is and where it is supposed to be, or via **computed torque control**, which adds a **feedforward** term that makes use of **inverse dynamics** to compute roughly what torque to send to make progress along the trajectory.
- **Optimal control** unites motion planning and trajectory tracking by computing an optimal trajectory directly over control inputs. This is especially easy when we have quadratic costs and linear dynamics, resulting in a linear quadratic regulator (**LQR**). Popular methods make use of this by linearizing the dynamics and computing second-order approximations of the cost (**ILQR**).
- Planning under uncertainty unites perception and action by **online replanning** (such as model predictive control) and **information gathering** actions that aid perception.
- Reinforcement learning is applied in robotics, with techniques striving to reduce the required number of interactions with the real world. Such techniques tend to exploit models, be it estimating models and using them to plan, or training policies that are robust with respect to different possible model parameters.
- Interaction with humans requires the ability to **coordinate** the robot's actions with theirs, which can be formulated as a game. We usually decompose the solution into **prediction**, in which we use the person's ongoing actions to estimate what they will do in the future, and **action**, in which we use the predictions to compute the optimal motion for the robot.
- Helping humans also requires the ability to learn or infer what they want. Robots can approach this by learning the desired cost function they should optimize from human input, such as demonstrations, corrections, or instruction in natural language. Alternatively, robots can imitate human behavior, and use reinforcement learning to help tackle the challenge of generalization to new states.

Bibliographical and Historical Notes

The word **robot** was popularized by Czech playwright Karel Čapek in his 1920 play *R.U.R.* (Rossum's Universal Robots). The robots, which were grown chemically rather than constructed mechanically, end up resenting their masters and decide to take over. It appears that it was Čapek's brother, Josef, who first combined the Czech words "robota" (obligatory work) and "robotnik" (serf) to yield "robot" in his 1917 short story *Opilec* (Glanc, 1978). The term *robotics* was invented for a science fiction story (Asimov, 1950).

The idea of an autonomous machine predates the word "robot" by thousands of years. In 7th century BCE Greek mythology, a robot named Talos was built by Hephaistos, the Greek god of metallurgy, to protect the island of Crete. The legend is that the sorceress Medea defeated Talos by promising him immortality but then draining his life fluid. Thus, this is the

first example of a robot making a mistake in the process of changing its objective function. In 322 BCE, Aristotle anticipated technological unemployment, speculating "If every tool, when ordered, or even of its own accord, could do the work that befits it... then there would be no need either of apprentices for the master workers or of slaves for the lords."

In the 3rd century BCE an actual humanoid robot called the Servant of Philon could pour wine or water into a cup; a series of valves cut off the flow at the right time. Wonderful automata were built in the 18th century—Jacques Vaucanson's mechanical duck from 1738 being one early example—but the complex behaviors they exhibited were entirely fixed in advance. Possibly the earliest example of a programmable robot-like device was the Jacquard loom (1805), described on page 33.

Grey Walter's "turtle," built in 1948, could be considered the first autonomous mobile robot, although its control system was not programmable. The "Hopkins Beast," built in 1960 at Johns Hopkins University, was much more sophisticated; it had sonar and photocell sensors, pattern-recognition hardware, and could recognize the cover plate of a standard AC power outlet. It was capable of searching for outlets, plugging itself in, and then recharging its batteries! Still, the Beast had a limited repertoire of skills.

The first general-purpose mobile robot was "Shakey," developed at what was then the Stanford Research Institute (now SRI) in the late 1960s (Fikes and Nilsson, 1971; Nilsson, 1984). Shakey was the first robot to integrate perception, planning, and execution, and much subsequent research in AI was influenced by this remarkable achievement. Shakey appears on the cover of this book with project leader Charlie Rosen (1917–2002). Other influential projects include the Stanford Cart and the CMU Rover (Moravec, 1983). Cox and Wilfong (1990) describe classic work on autonomous vehicles.

The first commercial robot was an arm called UNIMATE, for *universal automation*, developed by Joseph Engelberger and George Devol in their compnay, Unimation. In 1961, the first UNIMATE robot was sold to General Motors for use in manufacturing TV picture tubes. 1961 was also the year when Devol obtained the first U.S. patent on a robot.

In 1973, Toyota and Nissan started using an updated version of UNIMATE for auto body spot welding. This initiated a major revolution in automobile manufacturing that took place mostly in Japan and the U.S., and that is still ongoing. Unimation followed up in 1978 with the development of the Puma robot (Programmable Universal Machine for Assembly), which was the *de facto* standard for robotic manipulation for the two decades that followed. About 500,000 robots are sold each year, with half of those going to the automotive industry.

In manipulation, the first major effort at creating a hand–eye machine was Heinrich Ernst's MH-1, described in his MIT Ph.D. thesis (Ernst, 1961). The Machine Intelligence project at Edinburgh also demonstrated an impressive early system for vision-based assembly called FREDDY (Michie, 1972).

Research on mobile robotics has been stimulated by several important competitions. AAAI's annual mobile robot competition began in 1992. The first competition winner was CARMEL (Congdon *et al.*, 1992). Progress has been steady and impressive: in recent competitions robots entered the conference complex, found their way to the registration desk, registered for the conference, and even gave a short talk.

The **RoboCup** competition, launched in 1995 by Kitano and colleagues (1997), aims to "develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer" by 2050. Some competitions use wheeled robots, some

humanoid robots, and some software simulations. Stone (2016) describes recent innovations in RoboCup.

The **DARPA Grand Challenge**, organized by DARPA in 2004 and 2005, required autonomous vehicles to travel more than 200 kilometers through the desert in less than ten hours (Buehler *et al.*, 2006). In the original event in 2004, no robot traveled more than eight miles, leading many to believe the prize would never be claimed. In 2005, Stanford's robot Stanley won the competition in just under seven hours (Thrun, 2006). DARPA then organized the **Urban Challenge**, a competition in which robots had to navigate 60 miles in an urban environment with other traffic. Carnegie Mellon University's robot BOSS took first place and claimed the $2 million prize (Urmson and Whittaker, 2008). Early pioneers in the development of robotic cars included Dickmanns and Zapp (1987) and Pomerleau (1993).

The field of robotic mapping has evolved from two distinct origins. The first thread began with work by Smith and Cheeseman (1986), who applied Kalman filters to the simultaneous localization and mapping (SLAM) problem. This algorithm was first implemented by Moutarlier and Chatila (1989) and later extended by Leonard and Durrant-Whyte (1992); see Dissanayake *et al.* (2001) for an overview of early Kalman filter variations. The second thread began with the development of the **occupancy grid** representation for probabilistic mapping, which specifies the probability that each (x, y) location is occupied by an obstacle (Moravec and Elfes, 1985).

Occupancy grid

Kuipers and Levitt (1988) were among the first to propose topological rather than metric mapping, motivated by models of human spatial cognition. A seminal paper by Lu and Milios (1997) recognized the sparseness of the simultaneous localization and mapping problem, which gave rise to the development of nonlinear optimization techniques by Konolige (2004) and Montemerlo and Thrun (2004), as well as hierarchical methods by Bosse *et al.* (2004). Shatkay and Kaelbling (1997) and Thrun *et al.* (1998) introduced the EM algorithm into the field of robotic mapping for data association. An overview of probabilistic mapping methods can be found in (Thrun *et al.*, 2005).

Early mobile robot localization techniques are surveyed by Borenstein *et al.* (1996). Although Kalman filtering was well known as a localization method in control theory for decades, the general probabilistic formulation of the localization problem did not appear in the AI literature until much later, through the work of Tom Dean and colleagues (Dean *et al.*, 1990) and of Simmons and Koenig (1995). The latter work introduced the term **Markov localization**. The first real-world application of this technique was by Burgard *et al.* (1999), through a series of robots that were deployed in museums. Monte Carlo localization based on particle filters was developed by Fox *et al.* (1999) and is now widely used. The **Rao-Blackwellized particle filter** combines particle filtering for robot localization with exact filtering for map building (Murphy and Russell, 2001; Montemerlo *et al.*, 2002).

Markov localization

Rao-Blackwellized particle filter

A great deal of early work on **motion planning** focused on geometric algorithms for deterministic and fully observable motion planning problems. The PSPACE-hardness of robot motion planning was shown in a seminal paper by Reif (1979). The configuration space representation is due to Lozano-Perez (1983). A series of papers by Schwartz and Sharir on what they called **piano movers** problems (Schwartz *et al.*, 1987) was highly influential.

Piano movers

Recursive cell decomposition for configuration space planning was originated in the work of Brooks and Lozano-Perez (1985) and improved significantly by Zhu and Latombe (1991). The earliest skeletonization algorithms were based on Voronoi diagrams (Rowat, 1979) and

visibility graphs (Wesley and Lozano-Perez, 1979). Guibas *et al.* (1992) developed efficient techniques for calculating Voronoi diagrams incrementally, and Choset (1996) generalized Voronoi diagrams to broader motion planning problems. Visibility graph

John Canny (1988) established the first singly exponential algorithm for motion planning. The seminal text by Latombe (1991) covers a variety of approaches to motion planning, as do the texts by Choset *et al.* (2005) and LaValle (2006). Kavraki *et al.* (1996) developed the theory of probabilistic roadmaps. Kuffner and LaValle (2000) developed rapidly exploring random trees (RRTs).

Involving optimization in geometric motion planning began with elastic bands (Quinlan and Khatib, 1993), which refine paths when the configuration-space obstacles change. Ratliff *et al.* (2009) formulated the idea as the solution to an optimal control problem, allowing the initial trajectory to start in collision, and deforming it by mapping workspace obstacle gradients via the Jacobian into the configuration space. Schulman *et al.* (2013) proposed a practical second-order alternative.

The control of robots as dynamical systems—whether for manipulation or navigation—has generated a vast literature. While this chapter explained the basics of **trajectory tracking control** and optimal control, it left out entire subfields, including adaptive control, robust control, and Lyapunov analysis. Rather than assuming everything about the system is known a priori, adaptive control aims to adapt the dynamics parameters and/or the control law online. Robust control, on the other hand, aims to design controllers that perform well in spite of uncertainty and external disturbances.

Lyapunov analysis was originally developed in the 1890s for the stability analysis of general nonlinear systems, but it was not until the early 1930s that control theorists realized its true potential. With the development of optimization methods, Lyapunov analysis was extended to control barrier functions, which lend themselves nicely to modern optimization tools. These methods are widely used in modern robotics for real-time controller design and safety analysis.

Crucial works in robotic control include a trilogy on impedance control by Hogan (1985) and a general study of robot dynamics by Featherstone (1987). Dean and Wellman (1991) were among the first to try to tie together control theory and AI planning systems. Three classic textbooks on the mathematics of robot manipulation are due to Paul (1981), Craig (1989), and Yoshikawa (1990). Control for manipulation is covered by Murray (2017).

The area of **grasping** is also important in robotics—the problem of determining a stable grasp is quite difficult (Mason and Salisbury, 1985). Competent grasping requires touch sensing, or **haptic feedback**, to determine contact forces and detect slip (Fearing and Hollerbach, 1985). Understanding how to grasp the the wide variety of objects in the world is a daunting task. (Bousmalis *et al.*, 2017) describe a system that combines real-world experimentation with simulations guided by sim-to-real transfer to produce robust grasping. Haptic feedback

Potential-field control, which attempts to solve the motion planning and control problems simultaneously, was developed for robotics by Khatib (1986). In mobile robotics, this idea was viewed as a practical solution to the collision avoidance problem, and was later extended into an algorithm called **vector field histograms** by Borenstein (1991). Vector field histogram

ILQR is currently widely used at the intersection of motion planning and control and is due to Li and Todorov (2004). It is a variant of the much older differential dynamic programming technique (Jacobson and Mayne, 1970).

Fine-motion planning with limited sensing was investigated by Lozano-Perez *et al.* (1984) and Canny and Reif (1987). Landmark-based navigation (Lazanas and Latombe, 1992) uses many of the same ideas in the mobile robot arena. Navigation functions, the robotics version of a control policy for deterministic MDPs, were introduced by Koditschek (1987). Key work applying POMDP methods (Section 16.4) to motion planning under uncertainty in robotics is due to Pineau *et al.* (2003) and Roy *et al.* (2005).

Reinforcement learning in robotics took off with the seminal work by Bagnell and Schneider (2001) and Ng *et al.* (2003), who developed the paradigm in the context of autonomous helicopter control. Kober *et al.* (2013) offers an overview of how reinforcement learning changes when applied to the robotics problem. Many of the techniques implemented on physical systems build approximate dynamics models, dating back to locally weighted linear models due to Atkeson *et al.* (1997). But policy gradients played their role as well, enabling (simplified) humanoid robots to walk (Tedrake *et al.*, 2004), or a robot arm to hit a baseball (Peters and Schaal, 2008).

Levine *et al.* (2016) demonstrated the first **deep reinforcement learning** application on a real robot. At the same time, model-free RL in simulation was being extended to continuous domains (Schulman *et al.*, 2015a; Heess *et al.*, 2016; Lillicrap *et al.*, 2015). Other work scaled up physical data collection massively to showcase the learning of grasps and dynamics models (Pinto and Gupta, 2016; Agrawal *et al.*, 2017; Levine *et al.*, 2018). Transfer from simulation to reality or **sim-to-real** (Sadeghi and Levine, 2016; Andrychowicz *et al.*, 2018a), **metalearning** (Finn *et al.*, 2017), and sample-efficient model-free reinforcement learning (Andrychowicz *et al.*, 2018b) are active areas of research.

Early methods for predicting **human actions** made use of filtering approaches (Madhavan and Schlenoff, 2003), but seminal work by Ziebart *et al.* (2009) proposed prediction by modeling people as approximately rational agents. Sadigh *et al.* (2016) captured how these predictions should actually depend on what the robot decides to do, building toward a game-theoretic setting. For collaborative settings, Sisbot *et al.* (2007) pioneered the idea of accounting for what people want in the robot's cost function. Nikolaidis and Shah (2013) decomposed collaboration into learning how the human will act, but also learning how the human wants the robot to act, both achievable from demonstrations. For learning from demonstration see Argall *et al.* (2009). Akgun *et al.* (2012) and Sefidgar *et al.* (2017) studied teaching by end users rather than by experts.

Tellex *et al.* (2011) showed how robots can infer what people want from natural language instructions. Finally, not only do robots need to infer what people want and plan on doing, but people too need to make the same inferences about robots. Dragan *et al.* (2013) incorporated a model of the human's inferences into robot motion planning.

The field of **human–robot interaction** is much broader than what we covered in this chapter, which focused primarily on the planning and learning aspects. Thomaz *et al.* (2016) provides a survey of interaction more broadly from a computational perspective. Ross *et al.* (2011) describe the DAGGER system.

The topic of software architectures for robots engenders much religious debate. The good old-fashioned AI candidate—the three-layer architecture—dates back to the design of Shakey and is reviewed by Gat (1998). The subsumption architecture is due to Brooks (1986), although similar ideas were developed independently by Braitenberg, whose book, *Vehicles* (1984), describes a series of simple robots based on the behavioral approach.

The success of Brooks's six-legged walking robot was followed by many other projects. Connell, in his Ph.D. thesis (1989), developed an entirely reactive mobile robot that was capable of retrieving objects. Extensions of the paradigm to multirobot systems can be found in work by Parker (1996) and Mataric (1997). GRL (Horswill, 2000) and COLBERT (Konolige, 1997) abstract the ideas of concurrent behavior-based robotics into general robot control languages. Arkin (1998) surveys some of the most popular approaches in this field.

Two early textbooks, by Dudek and Jenkin (2000) and by Murphy (2000), cover robotics generally. More recent overviews are due to Bekey (2008) and Lynch and Park (2017). An excellent book on robot manipulation addresses advanced topics such as compliant motion (Mason, 2001). Robot motion planning is covered in Choset *et al.* (2005) and LaValle (2006). Thrun *et al.* (2005) introduces probabilistic robotics. The *Handbook of Robotics* (Siciliano and Khatib, 2016) is a massive, comprehensive overview of all of robotics.

The premiere conference for robotics is Robotics: Science and Systems Conference, followed by the IEEE International Conference on Robotics and Automation. Human–Robot Interaction is the premiere venue for interaction. Leading robotics journals include *IEEE Robotics and Automation*, the *International Journal of Robotics Research*, and *Robotics and Autonomous Systems*.