25 ROBOTICS

In which agents are endowed with physical effectors with which to do mischief.

25.1 INTRODUCTION

ROBOT
EFFECTOR
SENSOR

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 have a single purpose: to assert physical forces on the environment.[1] Robots are also equipped with **sensors**, which allow them to perceive their environment. Present day robotics employs a diverse set of sensors, including cameras and lasers to measure the environment, and gyroscopes and accelerometers to measure the robot's own motion.

MANIPULATOR

Most of today's robots fall into one of three primary categories. **Manipulators**, or robot arms (Figure 25.1(a)), are physically anchored to their workplace, for example in a factory assembly line or on the International Space Station. Manipulator motion usually involves a chain of controllable joints, enabling such robots to place their effectors in any position within the workplace. Manipulators are by far the most common type of industrial robots, with approximately one million units installed worldwide. Some mobile manipulators are used in hospitals to assist surgeons. Few car manufacturers could survive without robotic manipulators, and some manipulators have even been used to generate original artwork.

MOBILE ROBOT
UGV
PLANETARY ROVER

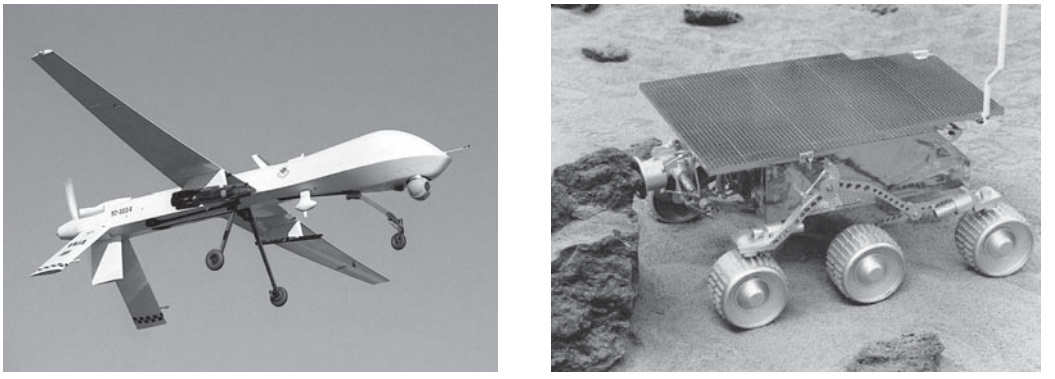
The second category is the **mobile robot**. Mobile robots move about their environment using wheels, legs, or similar mechanisms. They have been put to use delivering food in hospitals, moving containers at loading docks, and similar tasks. **Unmanned ground vehicles**, or UGVs, drive autonomously on streets, highways, and off-road. The **planetary rover** shown in Figure 25.2(b) explored Mars for a period of 3 months in 1997. Subsequent NASA robots include the twin Mars Exploration Rovers (one is depicted on the cover of this book), which landed in 2003 and were still operating six years later. Other types of mobile robots include **unmanned air vehicles** (UAVs), commonly used for surveillance, crop-spraying, and

UAV

[1] In Chapter 2 we talked about **actuators**, not effectors. Here we distinguish the effector (the physical device) from the actuator (the control line that communicates a command to the effector).

(a) (b)

Figure 25.1 (a) An industrial robotic manipulator for stacking bags on a pallet. Image courtesy of Nachi Robotic Systems. (b) Honda's P3 and Asimo humanoid robots.



(a) (b)

Figure 25.2 (a) Predator, an unmanned aerial vehicle (UAV) used by the U.S. Military. Image courtesy of General Atomics Aeronautical Systems. (b) NASA's Sojourner, a mobile robot that explored the surface of Mars in July 1997.

military operations. Figure 25.2(a) shows a UAV commonly used by the U.S. military. **Autonomous underwater vehicles** (AUVs) are used in deep sea exploration. Mobile robots deliver packages in the workplace and vacuum the floors at home.

AUV

The third type of robot combines mobility with manipulation, and is often called a **mobile manipulator**. **Humanoid robots** mimic the human torso. Figure 25.1(b) shows two early humanoid robots, both manufactured by Honda Corp. in Japan. Mobile manipulators

MOBILE MANIPULATOR

HUMANOID ROBOT

can apply their effectors further afield than anchored manipulators can, but their task is made harder because they don't have the rigidity that the anchor provides.

The field of robotics also includes prosthetic devices (artificial limbs, ears, and eyes for humans), intelligent environments (such as an entire house that is equipped with sensors and effectors), and multibody systems, wherein robotic action is achieved through swarms of small cooperating robots.

Real robots must cope with environments that are partially observable, stochastic, dynamic, and continuous. Many robot environments are sequential and multiagent as well. Partial observability and stochasticity are the result of dealing with a large, complex world. Robot cameras cannot see around corners, and motion commands are subject to uncertainty due to gears slipping, friction, etc. Also, the real world stubbornly refuses to operate faster than real time. In a simulated environment, it is possible to use simple algorithms (such as the Q-learning algorithm described in Chapter 21) to learn in a few CPU hours from millions of trials. In a real environment, it might take years to run these trials. Furthermore, real crashes really hurt, unlike simulated ones. Practical robotic systems need to embody prior knowledge about the robot, its physical environment, and the tasks that the robot will perform so that the robot can learn quickly and perform safely.

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

25.2 Robot Hardware

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

25.2.1 Sensors

PASSIVE 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 SENSOR

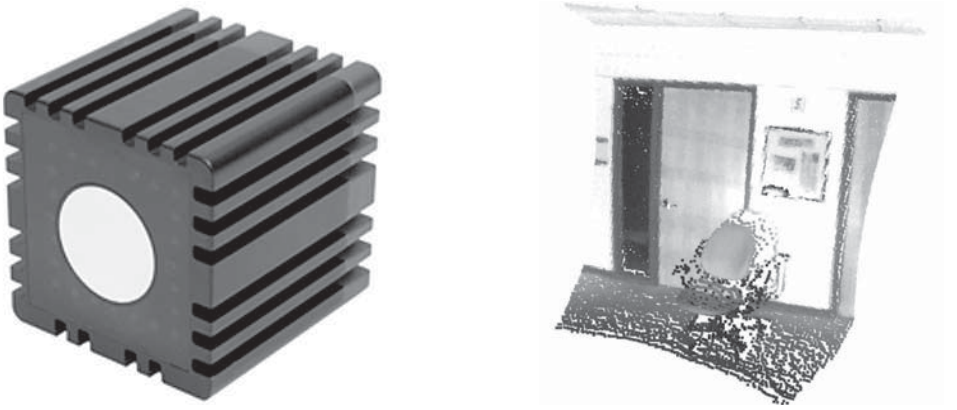
Active sensors, such as sonar, send energy into the environment. They rely on the fact that this energy is reflected back to the sensor. Active sensors tend to provide more information than passive sensors, but at the expense of increased power consumption and with a danger of interference when multiple active sensors are used at the same time. Whether active or passive, sensors can be divided into three types, depending on whether they sense the environment, the robot's location, or the robot's internal configuration.

RANGE FINDER

Range finders are sensors that measure the distance to nearby objects. In the early

SONAR SENSORS

days of robotics, robots were commonly equipped with **sonar sensors**. Sonar sensors emit directional sound waves, which are reflected by objects, with some of the sound making it



(a) (b)

Figure 25.3 (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.

back into 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.

STEREO VISION

Stereo vision (see Section 24.4.2) relies on multiple cameras to image the environment from slightly 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.

Most ground robots are now equipped with 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 25.3(a) shows a **time of flight camera**. This camera acquires range images like the one shown in Figure 25.3(b) at up to 60 frames per second. Other range sensors use laser beams and special 1-pixel cameras that can be directed using complex arrangements of mirrors or rotating elements. These sensors are called **scanning lidars** (short for *light detection and ranging*). Scanning lidars tend to provide longer ranges than time of flight cameras, and tend to perform better in bright daylight.

TIME OF FLIGHT CAMERA

SCANNING LIDARS

Other common range sensors include radar, which is often the sensor of choice for UAVs. Radar sensors can measure distances of multiple kilometers. On the other extreme 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.

TACTILE SENSORS

A second important class of sensors 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 satellites that emit pulsed signals. At present, there are 31 satellites in orbit, transmitting signals on multiple frequencies. GPS receivers can recover the distance to these satellites by analyzing phase shifts. By triangulating signals from multiple satellites, GPS

LOCATION SENSORS

GLOBAL POSITIONING SYSTEM

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. 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.

DIFFERENTIAL GPS

PROPRIOCEPTIVE SENSOR

SHAFT DECODER

ODOMETRY

INERTIAL SENSOR

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 count the revolution of motors in small increments. On robot arms, shaft decoders can provide accurate information over any period of time. On mobile robots, shaft decoders that report wheel revolutions can be used 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 the current for AUVs and the wind for UAVs, increase positional uncertainty. **Inertial sensors**, such as gyroscopes, rely on the resistance of mass to the change of velocity. They can help reduce uncertainty.

FORCE SENSOR

TORQUE 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 shape and location is 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. Good 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.

25.2.2 Effectors

Effectors are the means by which robots move and change the shape of their bodies. To understand the design of effectors, it will help to talk about motion and shape in the abstract, using the concept of a **degree of freedom** (DOF) We count one degree of freedom for each independent direction in which a robot, or one of its effectors, can move. For example, a rigid mobile robot such as an AUV has six degrees of freedom, three for its (x, y, z) location in space and three for its angular orientation, known as *yaw*, *roll*, and *pitch*. These six degrees define the **kinematic state**[2] or **pose** of the robot. The **dynamic state** of a robot includes these six plus an additional six dimensions for the rate of change of each kinematic dimension, that is, their velocities.

DEGREE OF FREEDOM

KINEMATIC STATE

POSE

DYNAMIC STATE

For nonrigid bodies, there are additional degrees of freedom within the robot itself. For example, the elbow of a human arm possesses two degree of freedom. It can flex the upper arm towards or away, and can rotate right or left. The wrist has three degrees of freedom. It can move up and down, side to side, and can also rotate. Robot joints also have one, two, or three degrees of freedom each. Six degrees of freedom are required to place an object, such as a hand, at a particular point in a particular orientation. The arm in Figure 25.4(a)

[2] "Kinematic" is from the Greek word for *motion*, as is "cinema."

REVOLUTE JOINT
PRISMATIC JOINT

has exactly six degrees of freedom, created by five **revolute joints** that generate rotational motion and one **prismatic joint** that generates sliding motion. You can verify that the human arm as a whole has more than six degrees of freedom by a simple experiment: put your hand on the table and notice that you still have the freedom to rotate your elbow without changing the configuration of your hand. Manipulators that have extra degrees of freedom are easier to control than robots with only the minimum number of DOFs. Many industrial manipulators therefore have seven DOFs, not six.

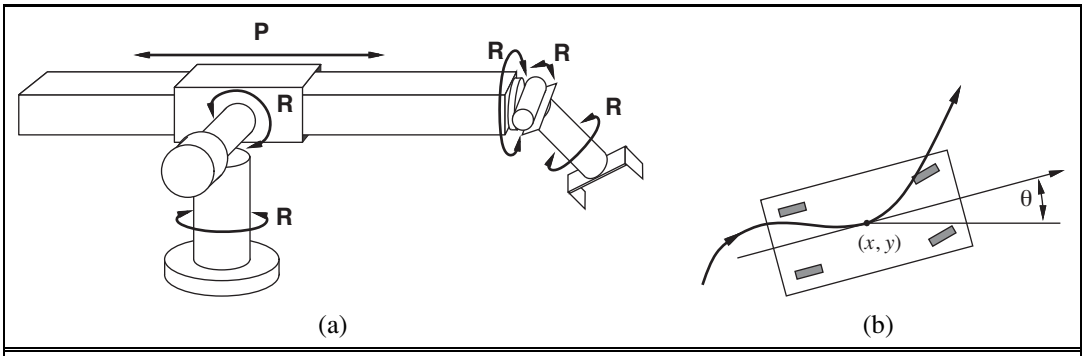


Figure 25.4 (a) The Stanford Manipulator, an early robot arm with five revolute joints (R) and one prismatic joint (P), for a total of six degrees of freedom. (b) Motion of a nonholonomic four-wheeled vehicle with front-wheel steering.

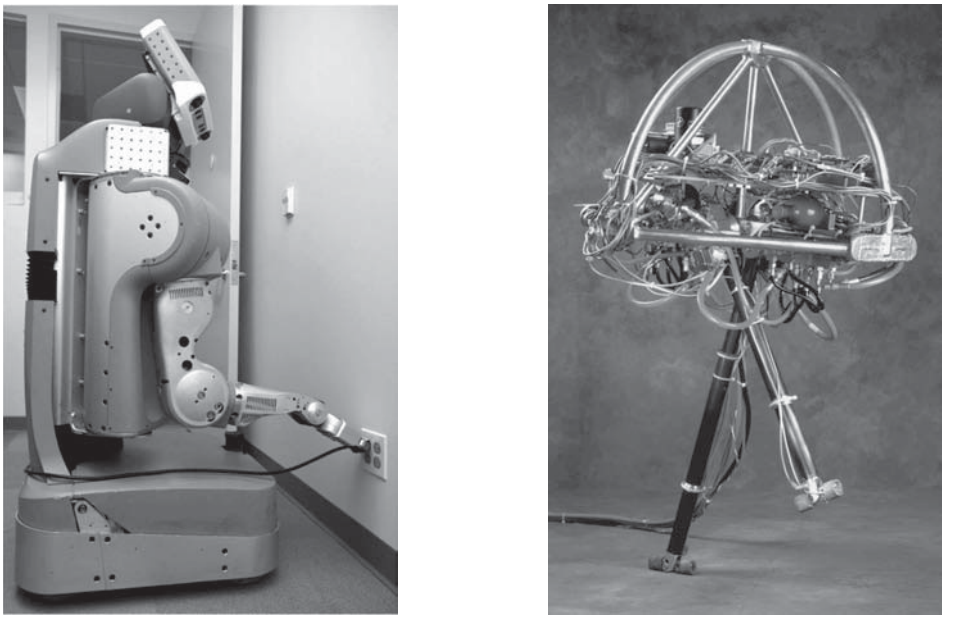
For mobile robots, the DOFs are not necessarily the same as the number of actuated elements. Consider, for example, your average car: it can move forward or backward, and it can turn, giving it two DOFs. In contrast, a car's kinematic configuration is three-dimensional: on an open flat surface, one can easily maneuver a car to any (x, y) point, in any orientation. (See Figure 25.4(b).) Thus, the car has three **effective degrees of freedom** but two **controllable degrees of freedom**. We say a robot is **nonholonomic** if it has more effective DOFs than controllable DOFs and **holonomic** if the two numbers are the same. Holonomic robots are easier to control—it would be much easier to park a car that could move sideways as well as forward and backward—but holonomic robots are also mechanically more complex. Most robot arms are holonomic, and most mobile robots are nonholonomic.

EFFECTIVE DOF
CONTROLLABLE DOF
NONHOLONOMIC

Mobile robots have a range of mechanisms for locomotion, including wheels, tracks, and legs. **Differential drive** robots possess two independently actuated wheels (or tracks), one on each side, as on a military tank. If both wheels move at the same velocity, the robot moves on a straight line. If they move in opposite directions, the robot turns on the spot. An alternative is the **synchro drive**, in which each wheel can move and turn around its own axis. To avoid chaos, the wheels are tightly coordinated. When moving straight, for example, all wheels point in the same direction and move at the same speed. Both differential and synchro drives are nonholonomic. Some more expensive robots use holonomic drives, which have three or more wheels that can be oriented and moved independently.

DIFFERENTIAL DRIVE
SYNCHRO DRIVE

Some mobile robots possess arms. Figure 25.5(a) displays a two-armed robot. This robot's arms use springs to compensate for gravity, and they provide minimal resistance to



(a) (b)

Figure 25.5 (a) Mobile manipulator plugging its charge cable into a wall outlet. Image courtesy of Willow Garage, © 2009. (b) One of Marc Raibert's legged robots in motion.

external forces. Such a design minimizes the physical danger to people who might stumble into such a robot. This is a key consideration in deploying robots in domestic environments.

Legs, unlike wheels, can handle rough terrain. However, legs are notoriously slow on flat surfaces, and they are mechanically difficult to build. Robotics researchers have tried designs ranging from one leg up to dozens of legs. Legged robots have been made to walk, run, and even hop—as we see with the legged robot in Figure 25.5(b). This robot is **dynamically stable**, meaning that it can remain upright while hopping around. A robot that can remain upright without moving its legs is called **statically stable**. A robot is statically stable if its center of gravity is above the polygon spanned by its legs. The quadruped (four-legged) robot shown in Figure 25.6(a) may appear statically stable. However, it walks by lifting multiple legs at the same time, which renders it dynamically stable. The robot can walk on snow and ice, and it will not fall over even if you kick it (as demonstrated in videos available online). Two-legged robots such as those in Figure 25.6(b) are dynamically stable.

DYNAMICALLY STABLE

STATICALLY STABLE

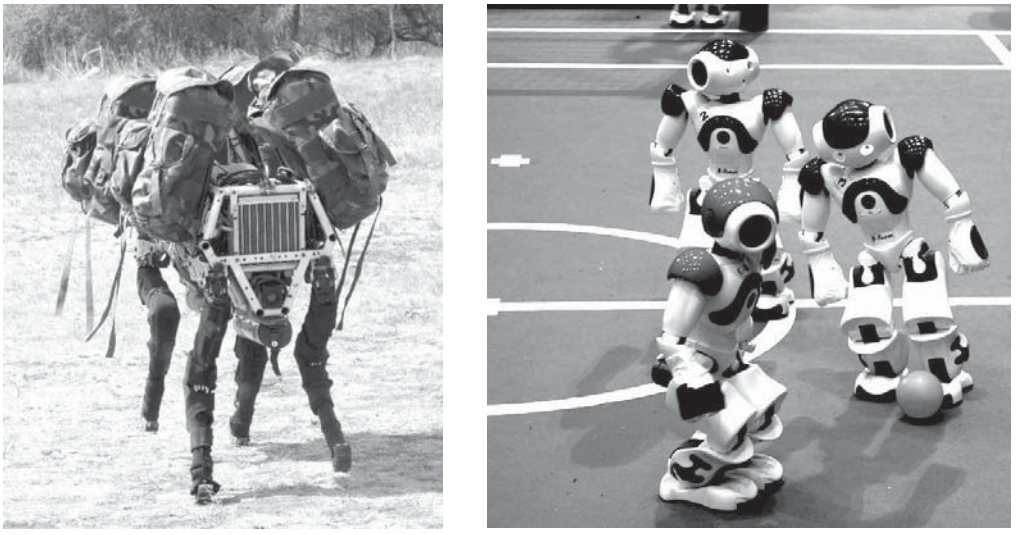
Other methods of movement are possible: air vehicles use propellers or turbines; underwater vehicles use propellers or thrusters, similar to those used on submarines. Robotic blimps rely on thermal effects to keep themselves aloft.

Sensors and effectors alone do not make a robot. A complete robot also needs a source of power to drive its effectors. The **electric motor** is the most popular mechanism for both manipulator actuation and locomotion, but **pneumatic actuation** using compressed gas and **hydraulic actuation** using pressurized fluids also have their application niches.

ELECTRIC MOTOR

PNEUMATIC ACTUATION

HYDRAULIC ACTUATION



(a) (b)

Figure 25.6 (a) Four-legged dynamically-stable robot "Big Dog." Image courtesy Boston Dynamics, © 2009. (b) 2009 RoboCup Standard Platform League competition, showing the winning team, B-Human, from the DFKI center at the University of Bremen. Throughout the match, B-Human outscored their opponents 64:1. Their success was built on probabilistic state estimation using particle filters and Kalman filters; on machine-learning models for gait optimization; and on dynamic kicking moves. Image courtesy DFKI, © 2009.

25.3 Robotic Perception

Perception is the process by which robots map sensor measurements into internal representations of the environment. 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 15.2. As a rule of thumb, good internal representations for robots have three properties: they contain enough information for the robot to make good decisions, they are structured so that they can be updated efficiently, and they are natural in the sense that internal variables correspond to natural state variables in the physical world.

In Chapter 15, 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 15. For robotics problems, we include the robot's own past actions as observed variables in the model. Figure 25.7 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.

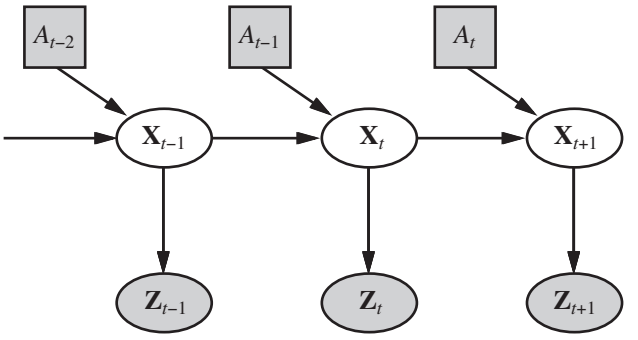


Figure 25.7 Robot perception can be viewed as temporal inference from sequences of actions and measurements, as illustrated by this dynamic Bayes network.

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 15.2, but here there are two differences: we condition explicitly on the actions as well as the observations, and we deal with *continuous* rather than *discrete* variables. Thus, we modify the recursive filtering equation (15.5 on page 572) to use integration rather than summation:

$$\begin{aligned}&\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{z}_{1:t+1}, a_{1:t}) \\ &= \alpha \mathbf{P}(\mathbf{z}_{t+1} \mid \mathbf{X}_{t+1}) \int \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t, a_t)\, P(\mathbf{x}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})\, d\mathbf{x}_t \,. \end{aligned} \tag{25.1}$$

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 be 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 (25.1) 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

25.3.1 Localization and mapping

LOCALIZATION

Localization is the problem of finding out where things are—including the robot itself. Knowledge about where things are is at the core of any successful physical interaction with the environment. For example, robot manipulators must know the location of objects they seek to manipulate; navigating robots must know where they are to find their way around.

To keep things simple, let us consider a mobile robot that moves slowly in a flat 2D world. Let us also assume the robot is given an exact map of the environment. (An example of such a map appears in Figure 25.10.) 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 25.8(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.

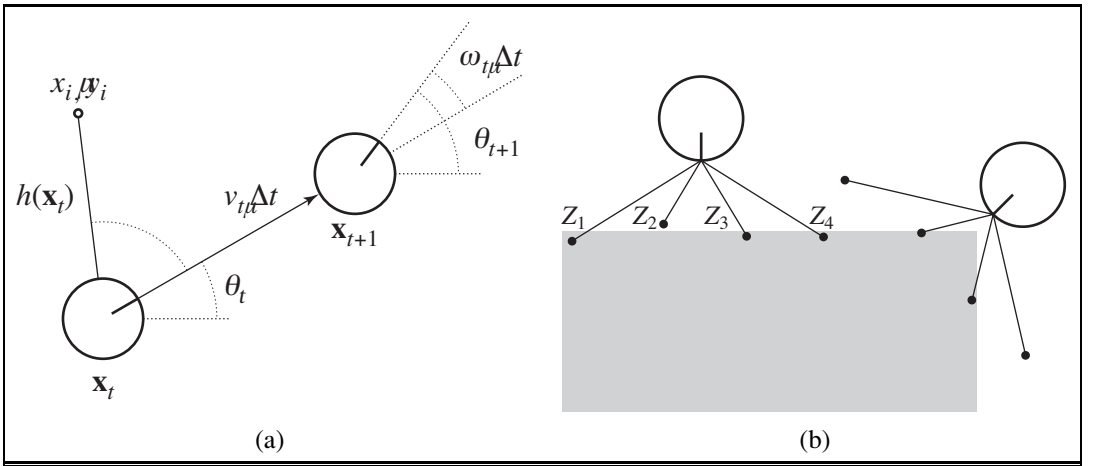


Figure 25.8 (a) A simplified kinematic model of a mobile robot. The robot is shown as a circle with an interior line marking the forward direction. The state $\mathbf{x}_t$ consists of the (x_t, y_t) position (shown implicitly) and the orientation θ_t. The new state $\mathbf{x}_{t+1}$ is obtained by 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.

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

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

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 $\mathbf{\Sigma}_x$. (See Appendix A for a mathematical definition.)

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

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 model. 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, the range and bearing can be calculated by simple geometry. (See Figure 25.8(a).) The exact prediction of the observed range and bearing would be

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, one might assume Gaussian noise with covariance $\boldsymbol{\Sigma}_z$, giving us the sensor model

$$P(\mathbf{z}_t \mid \mathbf{x}_t) = N(\hat{\mathbf{z}}_t, \boldsymbol{\Sigma}_z) \ .$$

A somewhat different sensor model is used for an 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 exact 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 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 25.8(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 25.8(b), the robot faces a featureless wall. On the other hand, if there *are* visible, identifiable landmarks, they may provide instant localization.

Chapter 15 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 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 alfgorithm is an instance of the particle-filtering algorithm of Figure 15.17 (page 598). All we need to do is supply the appropriate motion model and sensor model. Figure 25.9 shows one version using the range-scan model. The operation of the algorithm is illustrated in Figure 25.10 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 $\boldsymbol{\mu}_t$ and its covariance $\boldsymbol{\Sigma}_t$. The main problem with Gaussian beliefs is that they are only closed 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 25.11 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 $\boldsymbol{\mu}_t$, the mean of our state estimate at time t. Such a linearization

LINEARIZATION

```
function MONTE-CARLO-LOCALIZATION(a, z, N, P(X'|X, v, ω), P(z|z*), m) returns
a set of samples for the next time step
  inputs: a, robot velocities v and ω
          z, range scan z_1, ..., z_M
          P(X'|X, v, ω), motion model
          P(z|z*), range sensor noise model
          m, 2D map of the environment
  persistent: S, a vector of samples of size N
  local variables: W, a vector of weights of size N
                   S', a temporary vector of particles of size N
                   W', a vector of weights of size N

  if S is empty then        /* initialization phase */
    for i = 1 to N do
      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], m)
        W'[i] ← W'[i] · P(z_j | z*)
    S ← WEIGHTED-SAMPLE-WITH-REPLACEMENT(N, S', W')
  return S
```

Figure 25.9 A Monte Carlo localization algorithm using a range-scan sensor model with independent noise.

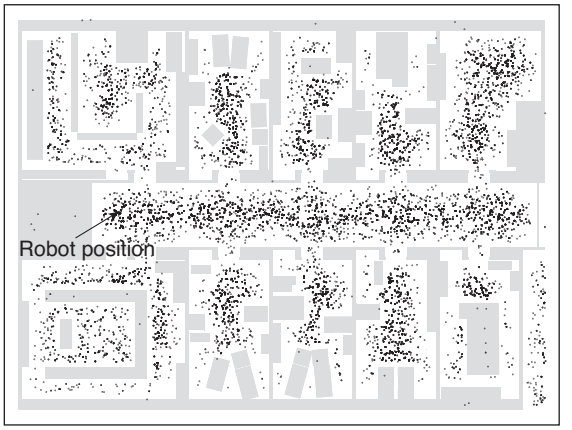
TAYLOR EXPANSION

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 25.12 shows a sequence of estimates of a robot running an extended Kalman filter localization algorithm. 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 25.10(b). The problem of needing to know the identity of landmarks is an instance of the **data association** problem discussed in Figure 15.6.

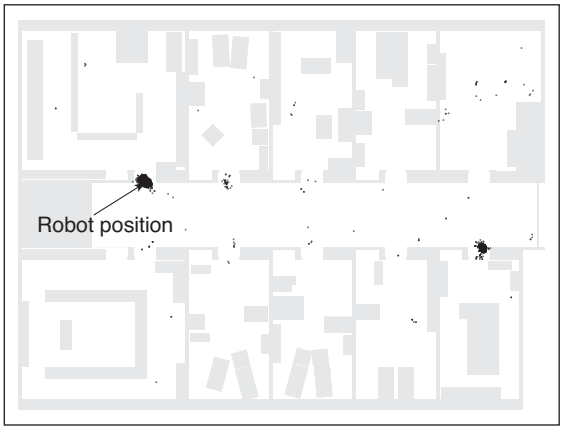
In some situations, no map of the environment is available. Then the robot will have to acquire a map. This is a bit of a chicken-and-egg problem: the navigating robot will have to determine its location relative to a map it doesn't quite know, at the same time building this map while it doesn't quite 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

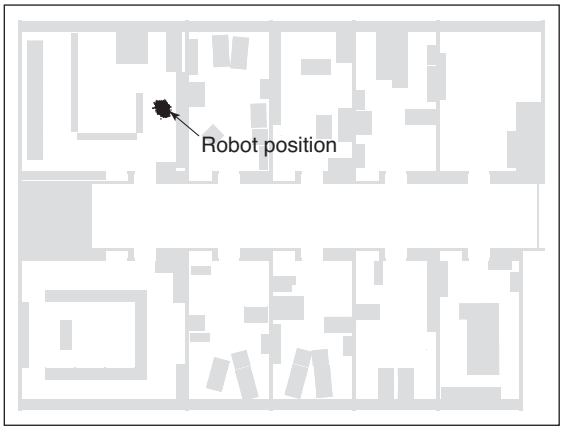
SLAM problems are solved using many different probabilistic techniques, including the extended Kalman filter discussed above. Using the EKF is straightforward: just augment



(a)



(b)



(c)

Figure 25.10 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.

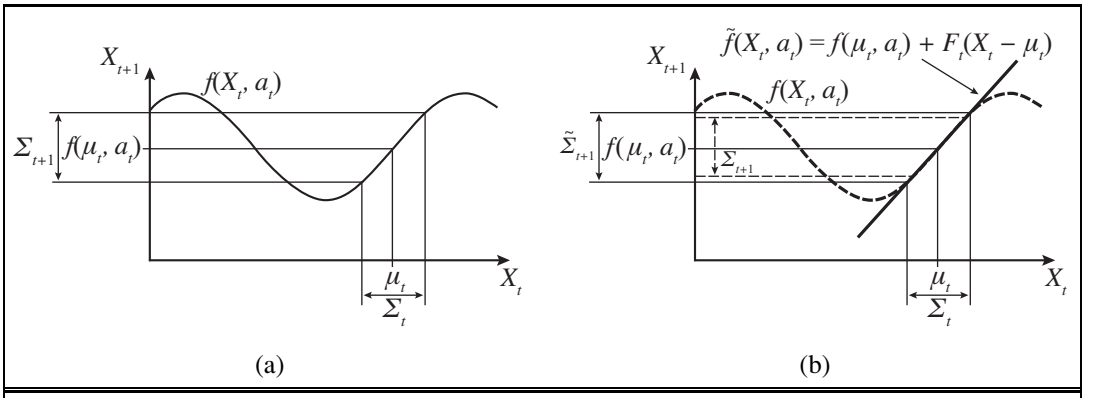


Figure 25.11 One-dimensional illustration of a linearized motion model: (a) The function f, and the projection of a mean $\boldsymbol{\mu}_t$ and a covariance interval (based on $\boldsymbol{\Sigma}_t$) into time $t+1$. (b) The linearized version is the tangent of f at $\boldsymbol{\mu}_t$. The projection of the mean $\boldsymbol{\mu}_t$ is correct. However, the projected covariance $\tilde{\boldsymbol{\Sigma}}_{t+1}$ differs from $\boldsymbol{\Sigma}_{t+1}$.

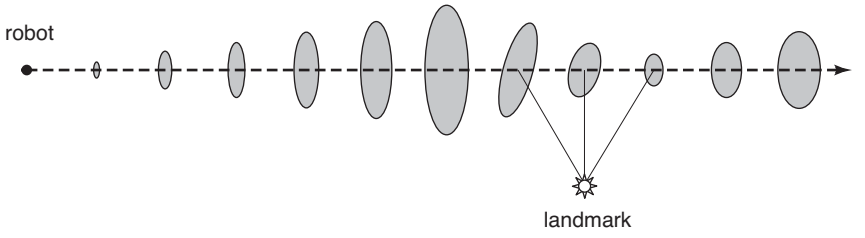


Figure 25.12 Example of localization using the extended Kalman filter. The robot moves on a straight line. As it progresses, its uncertainty increases gradually, as illustrated by the error ellipses. When it observes a landmark with known position, the uncertainty is reduced.

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 14. Expectation–maximization is also used for SLAM.

25.3.2 Other types of perception

Not all of robot perception is about localization or mapping. Robots also perceive the temperature, odors, acoustic signals, 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 25.6.3.

The trend in robotics is clearly towards representations with well-defined semantics.

(a) (b) (c)

Figure 25.13 Sequence of "drivable surface" classifier results using adaptive vision. In (a) only the road is classified as drivable (striped area). The V-shaped dark line shows where the vehicle is heading. In (b) the vehicle is commanded to drive off the road, onto a grassy surface, and the classifier is beginning to classify some of the grass as drivable. In (c) the vehicle has updated its model of drivable surface to correspond to grass as well as road.

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.

25.3.3 Machine learning in robot perception

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 18). 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 representations.

LOW-DIMENSIONAL EMBEDDING

Another machine learning technique enables robots to continuously adapt to broad changes in sensor measurements. Picture yourself walking from a sun-lit space into a dark neon-lit room. Clearly things are darker inside. But the change of light source also affects all the colors: Neon light has a stronger component of green light than sunlight. 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 suddenly their faces 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 25.13, 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 right 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 20 is then trained to recognize the specific color and texture coefficients of the small sample patch. The images in Figure 25.13 are the result of applying this classifier to the full image.

SELF-SUPERVISED LEARNING

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.

25.4 PLANNING TO MOVE

All of a robot's deliberations ultimately come down to deciding how to move effectors. The

POINT-TO-POINT MOTION

point-to-point motion problem is to deliver the robot or its end effector to a designated target

COMPLIANT MOTION

location. A greater challenge is the **compliant motion** problem, in which a robot moves while being in physical contact with an obstacle. An example of compliant motion is a robot manipulator that screws in a light bulb, or a robot that pushes a box across a table top.

We begin by finding a suitable representation in which motion-planning problems can be described and solved. It turns out that the **configuration space**—the space of robot states defined by location, orientation, and joint angles—is a better place to work than the original

PATH PLANNING

3D space. The **path planning** problem is to find a path from one configuration to another in configuration space. We have already encountered various versions of the path-planning problem throughout this book; the complication added by robotics is that path planning involves *continuous* spaces. There are two main approaches: **cell decomposition** and **skeletonization**. Each reduces the continuous path-planning problem to a discrete graph-search problem. In this section, we assume that motion is deterministic and that localization of the robot is exact. Subsequent sections will relax these assumptions.

25.4.1 Configuration space

We will start with a simple representation for a simple robot motion problem. Consider the robot arm shown in Figure 25.14(a). It has two joints that move independently. Moving the joints alters the (x, y) coordinates of the elbow and the gripper. (The arm cannot move in the z direction.) This suggests that the robot's configuration can be described by a four-dimensional coordinate: (x_e, y_e) for the location of the elbow relative to the environment and (x_g, y_g) for the location of the gripper. Clearly, these four coordinates characterize the full

WORKSPACE REPRESENTATION

state of the robot. They constitute what is known as **workspace representation**, since the coordinates of the robot are specified in the same coordinate system as the objects it seeks to manipulate (or to avoid). Workspace representations are well-suited for collision checking, especially if the robot and all objects are represented by simple polygonal models.

The problem with the workspace representation is that not all workspace coordinates are actually attainable, even in the absence of obstacles. This is because of the **linkage con-**

LINKAGE CONSTRAINTS

straints on the space of attainable workspace coordinates. For example, the elbow position (x_e, y_e) and the gripper position (x_g, y_g) are always a fixed distance apart, because they are joined by a rigid forearm. A robot motion planner defined over workspace coordinates faces the challenge of generating paths that adhere to these constraints. This is particularly tricky

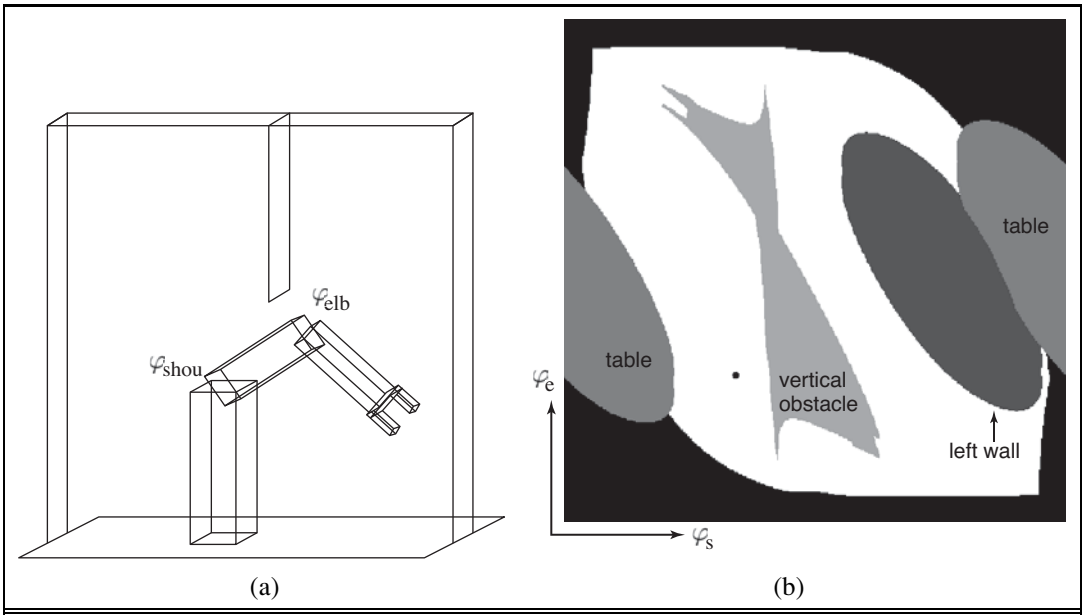


Figure 25.14 (a) Workspace representation of a robot arm with 2 DOFs. The workspace is a box with a flat obstacle hanging from the ceiling. (b) Configuration space of the same robot. Only white regions in the space are configurations that are free of collisions. The dot in this diagram corresponds to the configuration of the robot shown on the left.

CONFIGURATION SPACE

because the state space is continuous and the constraints are nonlinear. It turns out to be easier to plan with a **configuration space** representation. Instead of representing the state of the robot by the Cartesian coordinates of its elements, we represent the state by a configuration of the robot's joints. Our example robot possesses two joints. Hence, we can represent its state with the two angles φ_s and φ_e for the shoulder joint and elbow joint, respectively. In the absence of any obstacles, a robot could freely take on any value in configuration space. In particular, when planning a path one could simply connect the present configuration and the target configuration by a straight line. In following this path, the robot would then move its joints at a constant velocity, until a target location is reached.

Unfortunately, configuration spaces have their own problems. The task of a robot is usually expressed in workspace coordinates, not in configuration space coordinates. This raises the question of how to map between workspace coordinates and configuration space. Transforming configuration space coordinates into workspace coordinates is simple: it involves a series of straightforward coordinate transformations. These transformations are linear for prismatic joints and trigonometric for revolute joints. This chain of coordinate transformation is known as **kinematics**.

KINEMATICS

INVERSE KINEMATICS

The inverse problem of calculating the configuration of a robot whose effector location is specified in workspace coordinates is known as **inverse kinematics**. Calculating the inverse kinematics is hard, especially for robots with many DOFs. In particular, the solution is seldom unique. Figure 25.14(a) shows one of two possible configurations that put the gripper in the same location. (The other configuration would has the elbow below the shoulder.)

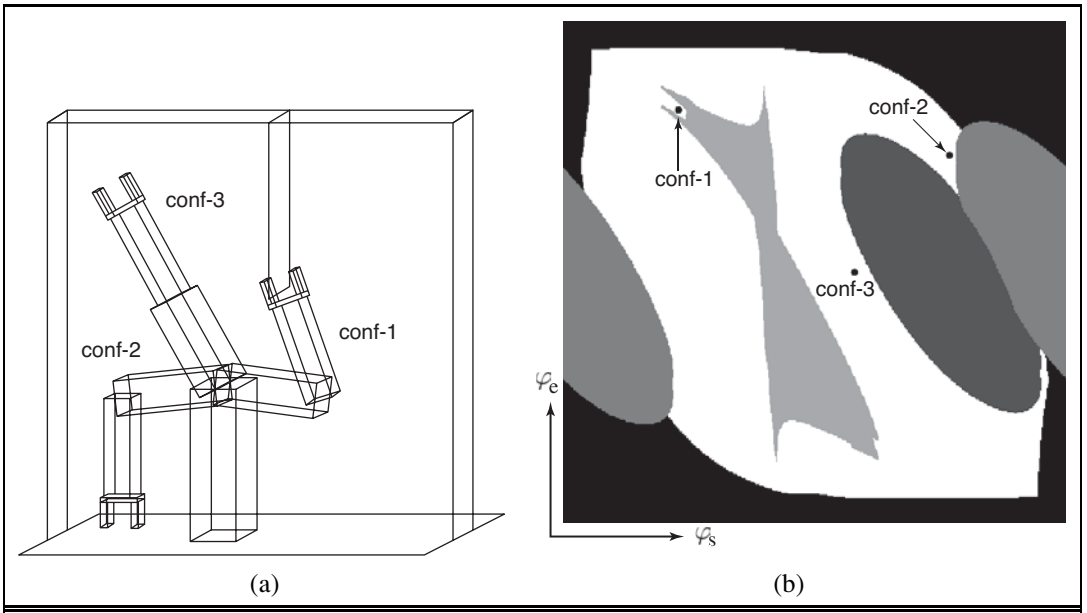


Figure 25.15 Three robot configurations, shown in workspace and configuration space.

In general, this two-link robot arm has between zero and two inverse kinematic solutions for any set of workspace coordinates. Most industrial robots have sufficient degrees of freedom to find infinitely many solutions to motion problems. To see how this is possible, simply imagine that we added a third revolute joint to our example robot, one whose rotational axis is parallel to the ones of the existing joints. In such a case, we can keep the location (but not the orientation!) of the gripper fixed and still freely rotate its internal joints, for most configurations of the robot. With a few more joints (how many?) we can achieve the same effect while keeping the orientation of the gripper constant as well. We have already seen an example of this in the "experiment" of placing your hand on the desk and moving your elbow. The kinematic constraint of your hand position is insufficient to determine the configuration of your elbow. In other words, the inverse kinematics of your shoulder–arm assembly possesses an infinite number of solutions.

The second problem with configuration space representations arises from the obstacles that may exist in the robot's workspace. Our example in Figure 25.14(a) shows several such obstacles, including a free-hanging obstacle that protrudes into the center of the robot's workspace. In workspace, such obstacles take on simple geometric forms—especially in most robotics textbooks, which tend to focus on polygonal obstacles. But how do they look in configuration space?

Figure 25.14(b) shows the configuration space for our example robot, under the specific obstacle configuration shown in Figure 25.14(a). The configuration space can be decomposed into two subspaces: the space of all configurations that a robot may attain, commonly called **free space**, and the space of unattainable configurations, called **occupied space**. The white area in Figure 25.14(b) corresponds to the free space. All other regions correspond to occu-

FREE SPACE

OCCUPIED SPACE

pied space. The different shadings of the occupied space corresponds to the different objects in the robot's workspace; the black 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 is not at all obvious! The dot inside Figure 25.14(b) marks the configuration of the robot, as shown in Figure 25.14(a). Figure 25.15 depicts three additional configurations, both in workspace and in configuration space. In configuration conf-1, the gripper encloses the vertical obstacle.

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.

25.4.2 Cell decomposition methods

CELL DECOMPOSITION

The first approach to path planning uses **cell decomposition**—that is, it decomposes the free space into a finite number of contiguous regions, called cells. These regions have the important property that the path-planning problem within a single region can be solved by simple means (e.g., moving along a straight line). The path-planning problem then becomes a discrete graph-search problem, very much like the search problems introduced in Chapter 3.

The simplest cell decomposition consists of a regularly spaced grid. Figure 25.16(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—i.e., 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 17.4 on page 653.) Figure 25.16(b) shows the corresponding workspace trajectory for the arm. Of course, we can also use the A* algorithm to find a shortest path.

Such a decomposition has the advantage that it is extremely simple to implement, but it also 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!dimensionality@of dimensionality. Second, 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 cross the cell in the desired direction in a straight line. 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

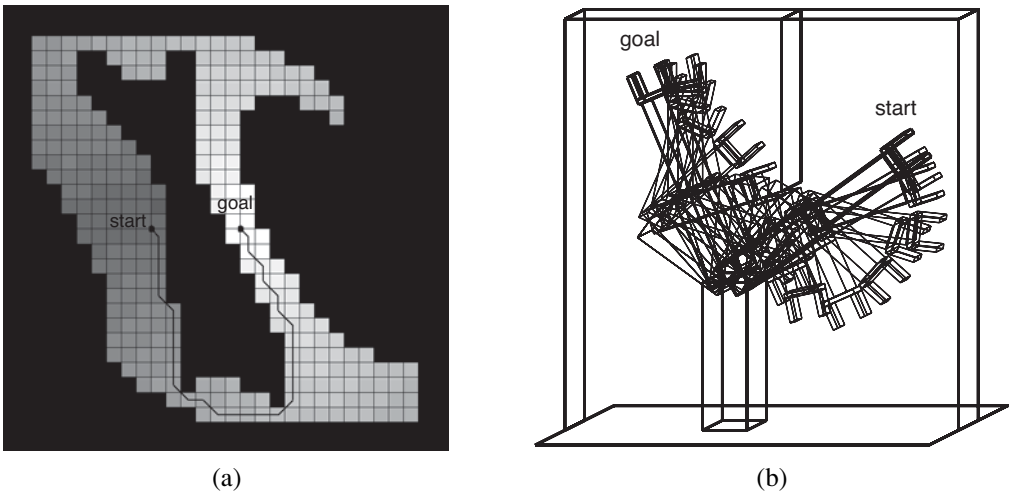


Figure 25.16 (a) Value function and path found for a discrete grid cell approximation of the configuration space. (b) The same path visualized in workspace coordinates. Notice how the robot bends its elbow to avoid a collision with the vertical obstacle.

be the case that the only paths to the goal go through mixed cells—especially if the cell size is comparable to that of the passageways and clearances in the space. And third, any path through a discretized state space will not be smooth. It is generally difficult to guarantee that a smooth solution exists near the discrete path. So a robot may not be able to execute the solution found through this decomposition.

Cell decomposition methods can be improved in a number of ways, to alleviate some of these problems. The first approach allows *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. (Of course, the method only works 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.) This method is complete provided there is a bound on the smallest passageway through which a solution must pass. Although it focuses most of the computational effort on the tricky areas within the configuration space, it still fails to scale well to high-dimensional problems because each recursive splitting of a cell creates 2^d smaller cells. A second way to obtain a complete algorithm is to insist on an **exact cell decomposition** of the free space. This method must allow cells to be irregularly shaped where they meet the boundaries of free space, but the shapes must still be "simple" in the sense that it should be easy to compute a traversal of any free cell. This technique requires some quite advanced geometric ideas, so we shall not pursue it further here.

EXACT CELL DECOMPOSITION

Examining the solution path shown in Figure 25.16(a), we can see an additional difficulty that will have to be resolved. The path contains arbitrarily sharp corners; a robot moving at any finite speed could not execute such a path. This problem is solved by storing certain continuous values for each grid cell. Consider an algorithm which stores, for each grid cell,

the exact, continuous state that was attained with the cell was first expanded 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. In doing so, 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*

25.4.3 Modified cost functions

Notice that in Figure 25.16, the path goes very close to the obstacle. Anyone who has driven a car knows that a parking space with one millimeter of clearance on either side is not really a parking space at all; for the same reason, we would prefer solution paths that are robust with respect to small motion errors.

POTENTIAL FIELD

This problem can be solved by introducing a **potential field**. A potential field is a function defined over state space, whose value grows with the distance to the closest obstacle. Figure 25.17(a) shows such a potential field—the darker a configuration state, the closer it is to an obstacle.

The potential field can be used as an additional cost term in the shortest-path calculation. This induces an interesting tradeoff. On the one hand, the robot seeks to minimize path length to the goal. On the other hand, it tries to stay away from obstacles by virtue of minimizing the potential function. With the appropriate weight balancing the two objectives, a resulting path may look like the one shown in Figure 25.17(b). This figure also displays the value function derived from the combined cost function, again calculated by value iteration. Clearly, the resulting path is longer, but it is also safer.

There exist many other ways to modify the cost function. For example, it may be desirable to *smooth* the control parameters over time. For example, when driving a car, a smooth path is better than a jerky one. In general, such higher-order constraints are not easy to accommodate in the planning process, unless we make the most recent steering command a part of the state. However, it is often easy to smooth the resulting trajectory after planning, using conjugate gradient methods. Such post-planning smoothing is essential in many real-world applications.

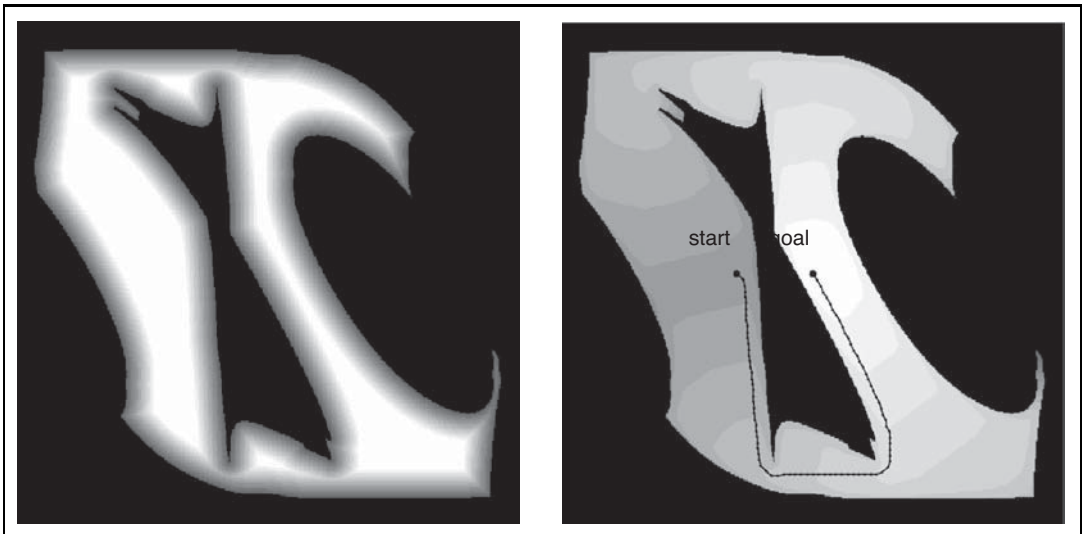
25.4.4 Skeletonization methods

SKELETONIZATION

The second major family of path-planning algorithms is based on the idea of **skeletonization**. These algorithms reduce the robot's free space to a one-dimensional representation, for which the planning problem is easier. This lower-dimensional representation is called a **skeleton** of the configuration space.

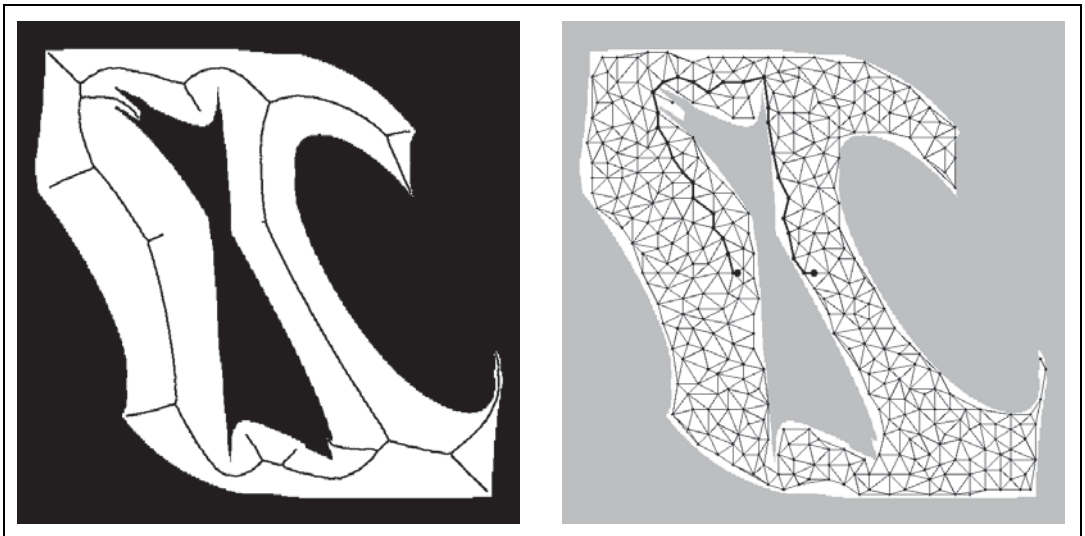
VORONOI GRAPH

Figure 25.18 shows an example skeletonization: it is a **Voronoi graph** of the free space—the set of all points that are equidistant to two or more obstacles. To do path planning with a Voronoi graph, the robot first changes its present configuration to a point on the Voronoi graph. It is easy to show that this can always be achieved by a straight-line motion in configuration space. Second, the robot follows the Voronoi graph until it reaches the point nearest to the target configuration. Finally, the robot leaves the Voronoi graph and moves to the target. Again, this final step involves straight-line motion in configuration space.



(a) (b)

Figure 25.17 (a) A repelling potential field pushes the robot away from obstacles. (b) Path found by simultaneously minimizing path length and the potential.



(a) (b)

Figure 25.18 (a) The Voronoi graph is the set of points equidistant to two or more obstacles in configuration space. (b) A probabilistic roadmap, composed of 400 randomly chosen points in free space.

In this way, the original path-planning problem is reduced to finding a path on the Voronoi graph, which is generally one-dimensional (except in certain nongeneric cases) and has finitely many points where three or more one-dimensional curves intersect. Thus, finding

the shortest path along the Voronoi graph is a discrete graph-search problem of the kind discussed in Chapters 3 and 4. Following the Voronoi graph may not give us the shortest path, but the resulting paths tend to maximize clearance. Disadvantages of Voronoi graph techniques are that they are difficult to apply to higher-dimensional configuration spaces, and that they tend to induce unnecessarily large detours when the configuration space is wide open. Furthermore, computing the Voronoi graph can be difficult, especially in configuration space, where the shapes of obstacles can be complex.

PROBABILISTIC ROADMAP

An alternative to the Voronoi graphs is the **probabilistic roadmap**, a skeletonization approach that offers more possible routes, and thus deals better with wide-open spaces. Figure 25.18(b) shows an example of a probabilistic roadmap. The graph is created by randomly generating a large number of configurations, and discarding those that do not fall into free space. Two nodes are joined by an arc if it is "easy" to reach one node from the other–for example, by a straight line in free space. The result of all this is a randomized graph in the robot's free space. If we add the robot's start and goal configurations to this graph, path planning amounts to a discrete graph search. Theoretically, this approach is incomplete, because a bad choice of random points may leave us without any paths from start to goal. It is possible to bound the probability of failure in terms of the number of points generated and certain geometric properties of the configuration space. It is also possible to direct the generation of sample points towards the areas where a partial search suggests that a good path may be found, working bidirectionally from both the start and the goal positions. With these improvements, probabilistic roadmap planning tends to scale better to high-dimensional configuration spaces than most alternative path-planning techniques.

25.5 Planning Uncertain Movements

None of the robot motion-planning algorithms discussed thus far addresses a key characteristic of robotics problems: *uncertainty*. 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 provide the robot with an exact belief state even if the stochastic nature of the environment is modeled perfectly.

Most of today's robots use deterministic algorithms for decision making, such as the path-planning algorithms of the previous section. To do so, it is common practice to extract the **most likely state** from the probability distribution produced by the state estimation algorithm. The advantage of this approach is purely computational. Planning paths through configuration space is already a challenging problem; it would be worse if we had to work with a full probability distribution over states. Ignoring uncertainty in this way works when the uncertainty is small. In fact, when the environment model changes over time as the result of incorporating sensor measurements, many robots plan paths online during plan execution. This is the **online replanning** technique of Section 11.3.3.

MOST LIKELY STATE

ONLINE REPLANNING

Unfortunately, ignoring the uncertainty does not always work. In some problems the robot's uncertainty is simply too massive: How can we use a deterministic path planner to control a mobile robot that has no clue where it is? In general, if the robot's true state is not the one identified by the maximum likelihood rule, the resulting control will be suboptimal. Depending on the magnitude of the error this can lead to all sorts of unwanted effects, such as collisions with obstacles.

The field of robotics has adopted a range of techniques for accommodating uncertainty. Some are derived from the algorithms given in Chapter 17 for decision making under uncertainty. If the robot faces uncertainty only in its state transition, but its state is fully observable, the problem is best modeled as a Markov decision process (MDP). The solution of an MDP is an optimal **policy**, which tells the robot what to do in every possible state. In this way, it can handle all sorts of motion errors, whereas a single-path solution from a deterministic planner would be much less robust. In robotics, policies are called **navigation functions**. The value function shown in Figure 25.16(a) can be converted into such a navigation function simply by following the gradient.

NAVIGATION FUNCTION

Just as in Chapter 17, partial observability makes the problem much harder. The resulting robot control problem is a partially observable MDP, or POMDP. In such situations, the robot maintains an internal belief state, like the ones discussed in Section 25.3. The solution to a POMDP is a policy defined over the robot's belief state. Put differently, the input to the policy is an entire probability distribution. This enables the robot to base its decision not only on what it knows, but also on what it does not know. For example, if it is uncertain about a critical state variable, it can rationally invoke an **information gathering action**. This is impossible in the MDP framework, since MDPs assume full observability. Unfortunately, techniques that solve POMDPs exactly are inapplicable to robotics—there are no known techniques for high-dimensional continuous spaces. Discretization produces POMDPs that are far too large to handle. One remedy is to make the minimization of uncertainty a control objective. For example, the **coastal navigation** heuristic requires the robot to stay near known landmarks to decrease its uncertainty. Another approach applies variants of the probabilistic roadmap planning method to the belief space representation. Such methods tend to scale better to large discrete POMDPs.

INFORMATION GATHERING ACTION

COASTAL NAVIGATION

25.5.1 Robust methods

ROBUST CONTROL

Uncertainty can also be handled using so-called **robust control** methods (see page 836) rather than probabilistic methods. A robust method is one that assumes a *bounded* amount of uncertainty in each aspect of a problem, but does not assign probabilities to values within the allowed interval. A robust solution is one that works no matter what actual values occur, provided they are within the assumed interval. An extreme form of robust method is the **conformant planning** approach given in Chapter 11—it produces plans that work with no state information at all.

FINE-MOTION PLANNING

Here, we look at a robust method that is used for **fine-motion planning** (or FMP) in robotic assembly tasks. Fine-motion planning involves moving a robot arm in very close proximity to a static environment object. The main difficulty with fine-motion planning is

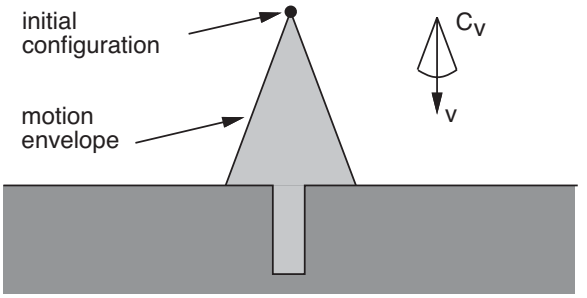


Figure 25.19 A two-dimensional environment, velocity uncertainty cone, and envelope of possible robot motions. The intended velocity is v, but with uncertainty the actual velocity could be anywhere in C_v, resulting in a final configuration somewhere in the motion envelope, which means we wouldn't know if we hit the hole or not.

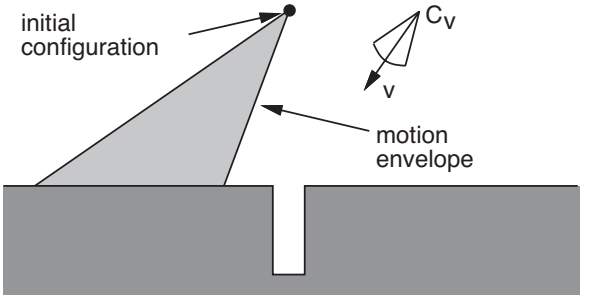


Figure 25.20 The first motion command and the resulting envelope of possible robot motions. No matter what the error, we know the final configuration will be to the left of the hole.

that the required motions and the relevant features of the environment are very small. At such small scales, the robot is unable to measure or control its position accurately and may also be uncertain of the shape of the environment itself; we will assume that these uncertainties are all bounded. The solutions to FMP problems will typically be conditional plans or policies that make use of sensor feedback during execution and are guaranteed to work in all situations consistent with the assumed uncertainty bounds.

GUARDED MOTION

A fine-motion plan consists of a series of **guarded motions**. Each guarded motion consists of (1) a motion command and (2) a termination condition, which is a predicate on the robot's sensor values, and returns true to indicate the end of the guarded move. The motion commands are typically **compliant motions** that allow the effector to slide if the motion command would cause collision with an obstacle. As an example, Figure 25.19 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

COMPLIANT MOTION

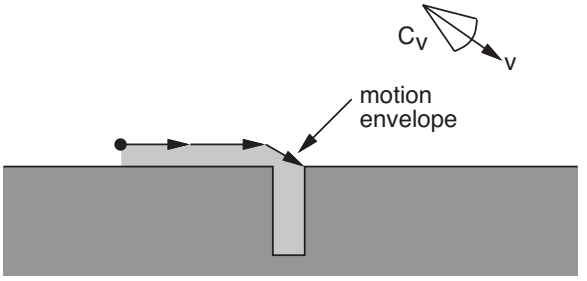


Figure 25.21 The second motion command and the envelope of possible motions. Even with error, we will eventually get into the hole.

if we commanded a velocity 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 25.20 and 25.21. In Figure 25.20, 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 25.21, 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.

As one might imagine, the problem of *constructing* fine-motion plans is not trivial; in fact, it is a good deal harder than planning with exact motions. One can either choose a fixed number of discrete values for each motion or use the environment geometry to choose directions that give qualitatively different behavior. A fine-motion planner takes as input the configuration-space description, the angle of the velocity uncertainty cone, and a specification of what sensing is possible for termination (surface contact in this case). It should produce a multistep conditional plan or policy that is guaranteed to succeed, if such a plan exists.

Our example assumes that the planner has an exact model of the environment, but it is possible to allow for bounded error in this model as follows. If the error can be described in terms of parameters, those parameters can be added as degrees of freedom to the configuration space. In the last example, if the depth and width of the hole were uncertain, we could add them as two degrees of freedom to the configuration space. It is impossible to move the robot in these directions in the configuration space or to sense its position directly. But both those restrictions can be incorporated when describing this problem as an FMP problem by appropriately specifying control and sensor uncertainties. This gives a complex, four-dimensional planning problem, but exactly the same planning techniques can be applied.

Notice that unlike the decision-theoretic methods in Chapter 17, this kind of robust approach results in plans designed for the worst-case outcome, rather than maximizing the expected quality of the plan. Worst-case plans are optimal in the decision-theoretic sense only if failure during execution is much worse than any of the other costs involved in execution.

25.6 MOVING

So far, we have talked about how to *plan* motions, but not about how to *move*. Our plans—particularly those produced by deterministic path planners—assume that the robot can simply follow any path that the algorithm produces. In the real world, of course, this is not the case. Robots have inertia and cannot execute arbitrary paths except at arbitrarily slow speeds. In most cases, the robot gets to exert forces rather than specify positions. This section discusses methods for calculating these forces.

25.6.1 Dynamics and control

Section 25.2 introduced the notion of **dynamic state**, which extends the kinematic state of a robot by its velocity. For example, in addition to the angle of a robot joint, the dynamic state also captures the rate of change of the angle, and possibly even its momentary acceleration. The transition model for a dynamic state representation includes the effect of forces on this rate of change. Such models are typically expressed via **differential equations**, which are equations that relate a quantity (e.g., a kinematic state) to the change of the quantity over time (e.g., velocity). In principle, we could have chosen to plan robot motion using dynamic models, instead of our kinematic models. Such a methodology would lead to superior robot performance, if we could generate the plans. However, the dynamic state has higher dimension than the kinematic space, and the curse of dimensionality would render many motion planning algorithms inapplicable for all but the most simple robots. For this reason, practical robot system often rely on simpler kinematic path planners.

DIFFERENTIAL EQUATION

A common technique to compensate for the limitations of kinematic plans is to use a separate mechanism, a **controller**, for keeping the robot on track. Controllers are techniques for generating robot controls in real time using feedback from the environment, so as to achieve a control objective. If the objective is to keep the robot on a preplanned path, it is often referred to as a **reference controller** and the path is called a **reference path**. Controllers that optimize a global cost function are known as **optimal controllers**. Optimal policies for continuous MDPs are, in effect, optimal controllers.

CONTROLLER

REFERENCE CONTROLLER

REFERENCE PATH

OPTIMAL CONTROLLERS

On the surface, the problem of keeping a robot on a prespecified path appears to be relatively straightforward. In practice, however, even this seemingly simple problem has its pitfalls. Figure 25.22(a) illustrates what can go wrong; it shows the path of a robot that attempts to follow a kinematic path. 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,

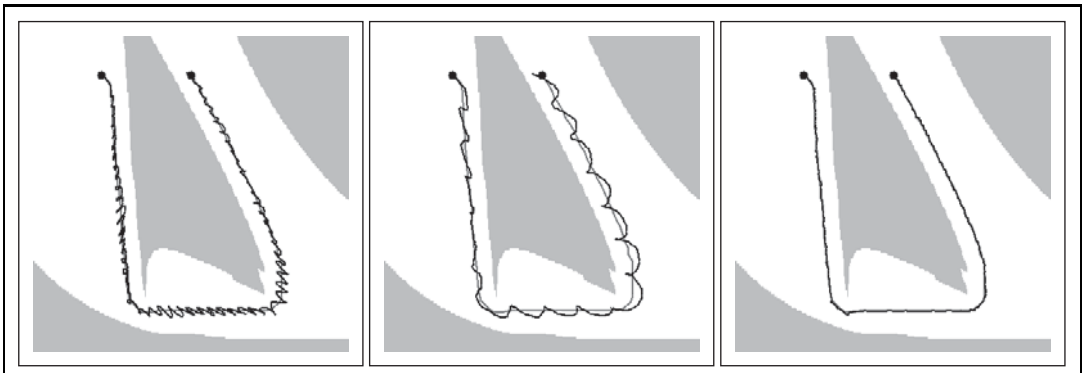


Figure 25.22 Robot arm control using (a) proportional control with gain factor 1.0, (b) proportional control with gain factor 0.1, and (c) PD (proportional derivative) control with gain factors 0.3 for the proportional component and 0.8 for the differential component. In all cases the robot arm tries to follow the path shown in gray.

as Figure 25.22(a) illustrates, our controller causes the robot to vibrate rather violently. The vibration is the result of a natural inertia of the robot arm: once driven back to its reference position the robot then overshoots, which induces a symmetric error with opposite sign. Such overshooting may continue along an entire trajectory, and the resulting robot motion is far from desirable.

Before we can define a better controller, let us formally describe what went wrong. Controllers that provide force in negative proportion to the observed error are known as **P controllers**. The letter 'P' stands for *proportional*, indicating that the actual control is proportional to the error of the robot manipulator. More formally, let $y(t)$ be the reference path, parameterized by time index t. The control a_t generated by a P controller has the form:

P CONTROLLER

$$a_t = K_P(y(t) - x_t) \ .$$

Here x_t is the state of the robot at time t and K_P is a constant known as the **gain parameter** of the controller and its value is called the gain factor); K_p regulates how strongly the controller corrects for deviations between the actual state x_t and the desired one $y(t)$. In our example, $K_P = 1$. At first glance, one might think that choosing a smaller value for K_P would remedy the problem. Unfortunately, this is not the case. Figure 25.22(b) shows a trajectory for $K_P = .1$, still exhibiting oscillatory behavior. Lower values of the gain parameter may simply slow down the oscillation, but do 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.

GAIN PARAMETER

Traditionally, problems of this type fall into the realm of **control theory**, a field of increasing importance to researchers in AI. Decades of research in this field have led to a large number of controllers that are superior to the simple control law given above. In particular, a reference 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

STABLE

STRICTLY STABLE

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:

$$a_t = K_P(y(t) - x_t) + K_D\frac{\partial(y(t) - x_t)}{\partial t} . \tag{25.2}$$

As this equation suggests, PD controllers extend P controllers by a differential component, which adds to the value of a_t a term that is proportional to the first derivative of the error $y(t) - x_t$ 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 $(y(t)-x_t)$ 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 25.22(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. An autonomous car driving on a banked surface, for example, may find itself systematically pulled to one side. Wear and tear in robot arms cause 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:

$$a_t = K_P(y(t) - x_t) + K_I\int(y(t) - x_t)dt + K_D\frac{\partial(y(t) - x_t)}{\partial t} . \tag{25.3}$$

Here K_I is yet another gain parameter. The term $\int(y(t) - x_t)dt$ 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. If, for example, x_t is smaller than $y(t)$ for a long period of time, this integral will grow until the resulting control a_t forces this error to shrink. Integral terms, then, ensure that a controller does not exhibit systematic error, at the expense of increased danger of oscillatory behavior. 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.

PID CONTROLLER

25.6.2 Potential-field control

We introduced potential fields as an additional cost function in robot motion planning, but they can also be used for generating robot motion directly, dispensing with the path planning phase altogether. To achieve this, we have to define an attractive force that pulls the robot towards its goal configuration and a repellent potential field that pushes the robot away from obstacles. Such a potential field is shown in Figure 25.23. Its single global minimum is

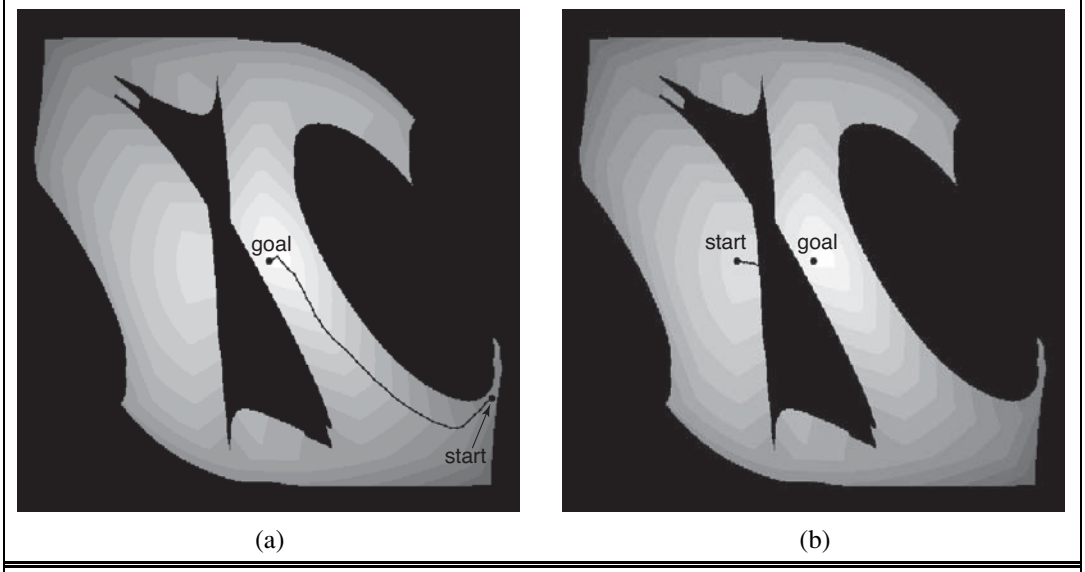
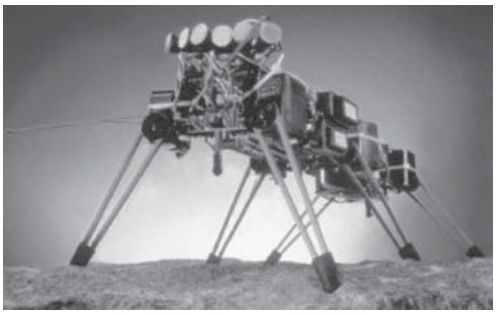
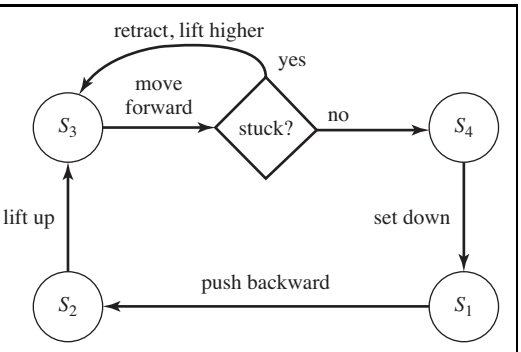


Figure 25.23 Potential field control. The robot ascends a potential field composed of repelling forces asserted from the obstacles and an attracting force that corresponds to the goal configuration. (a) Successful path. (b) Local optimum.

the goal configuration, and the value is the sum of the distance to this goal configuration and the proximity to obstacles. No planning was involved in generating the potential field shown in the figure. Because of this, potential fields are well suited to real-time control. Figure 25.23(a) shows a trajectory of a robot that performs hill climbing in the potential field. In many applications, the potential field can be calculated efficiently for any given configuration. Moreover, optimizing the potential amounts to calculating the gradient of the potential for the present robot configuration. These calculations can be extremely efficient, especially when compared to path-planning algorithms, all of which are exponential in the dimensionality of the configuration space (the DOFs) in the worst case.

The fact that the potential field approach manages to find a path to the goal in such an efficient manner, even over long distances in configuration space, raises the question as to whether there is a need for planning in robotics at all. Are potential field techniques sufficient, or were we just lucky in our example? The answer is that we were indeed lucky. Potential fields have many local minima that can trap the robot. In Figure 25.23(b), the robot approaches the obstacle by simply rotating its shoulder joint, until it gets stuck on the wrong side of the obstacle. The potential field is not rich enough to make the robot bend its elbow so that the arm fits under the obstacle. In other words, potential field control is great for local robot motion but sometimes we still need global planning. Another important drawback with potential fields is that the forces they generate depend only on the obstacle and robot positions, not on the robot's velocity. Thus, potential field control is really a kinematic method and may fail if the robot is moving quickly.





(a) (b)

Figure 25.24 (a) Genghis, a hexapod robot. (b) An augmented finite state machine (AFSM) for the control of a single leg. Notice that this AFSM reacts to sensor feedback: if a leg is stuck during the forward swinging phase, it will be lifted increasingly higher.

25.6.3 Reactive control

So far we have considered control decisions that require some model of the environment for constructing either a reference path or a potential field. There are some difficulties with this approach. First, models that are sufficiently accurate are often difficult to obtain, especially in complex or remote environments, such as the surface of Mars, or for robots that have few sensors. Second, even in cases where we can devise a model with sufficient accuracy, computational difficulties and localization error might render these techniques impractical.

REACTIVE CONTROL

In some cases, a reflex agent architecture using **reactive control** is more appropriate.

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 25.24(a), designed for walking through rough terrain. The robot's sensors are inadequate to obtain models of the terrain for path planning. Moreover, even if we added sufficiently accurate sensors, the twelve degrees of freedom (two for each leg) would render the resulting path planning problem computationally intractable.

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; it did, however, require a reference path generated from a kinematic model.) For the hexapod robot we first

GAIT

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,*

Figure 25.25 Multiple exposures of an RC helicopter executing a flip based on a policy learned with reinforcement learning. Images courtesy of Andrew Ng, Stanford University.

and try again. The resulting controller is shown in Figure 25.24(b) as a 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).

Variants of this simple feedback-driven controller have been found to generate remarkably robust walking patterns, capable of maneuvering the robot over rugged terrain. Clearly, such a controller is model-free, and it does not deliberate or use search for generating controls. Environmental feedback plays a crucial role in the controller's execution. The software alone does not specify what will actually happen when the robot is placed in an environment. Behavior that emerges through the interplay of a (simple) controller and a (complex) environment is often referred to as **emergent behavior**. Strictly speaking, all robots discussed in this chapter exhibit emergent behavior, due to the fact that no model is perfect. Historically, however, the term has been reserved for control techniques that do not utilize explicit environmental models. Emergent behavior is also characteristic of biological organisms.

EMERGENT BEHAVIOR

25.6.4 Reinforcement learning control

One particularly exciting form of control is based on the **policy search** form of reinforcement learning (see Section 21.5). This work has been enormously influential in recent years, at is has solved challenging robotics problems for which previously no solution existed. An example is acrobatic autonomous helicopter flight. Figure 25.25 shows an autonomous flip of a small RC (radio-controlled) helicopter. This maneuver is challenging due to the highly nonlinear nature of the aerodynamics involved. Only the most experienced of human pilots are able to perform it. Yet a policy search method (as described in Chapter 21), using only a few minutes of computation, learned a policy that can safely execute a flip every time.

Policy search needs an accurate model of the domain before it can find a policy. The input to this model is the state of the helicopter at time t, the controls at time t, and the resulting state at time $t + \Delta t$. The state of a helicopter can be described by the 3D coordinates of the vehicle, its yaw, pitch, and roll angles, and the rate of change of these six variables. The controls are the manual controls of of the helicopter: throttle, pitch, elevator, aileron, and rudder. All that remains is the resulting state—how are we going to define a model that accurately says how the helicopter responds to each control? The answer is simple: Let an expert human pilot fly the helicopter, and record the controls that the expert transmits over the radio and the state variables of the helicopter. About four minutes of human-controlled flight suffices to build a predictive model that is sufficiently accurate to simulate the vehicle.

What is remarkable about this example is the ease with which this learning approach solves a challenging robotics problem. This is one of the many successes of machine learning in scientific fields previously dominated by careful mathematical analysis and modeling.

25.7 ROBOTIC SOFTWARE ARCHITECTURES

SOFTWARE ARCHITECTURE

A methodology for structuring algorithms is called a **software architecture**. An architecture includes languages and tools for writing programs, as well as an overall philosophy for how programs can be brought together.

Modern-day software architectures for robotics must decide how to combine reactive control and model-based deliberative planning. In many ways, reactive and deliberate techniques have orthogonal strengths and weaknesses. Reactive control is sensor-driven and appropriate for making low-level decisions in real time. However, it rarely yields a plausible solution at the global level, because global control decisions depend on information that cannot be sensed at the time of decision making. For such problems, deliberate planning is a more appropriate choice.

Consequently, most robot architectures use reactive techniques at the lower levels of control and deliberative techniques at the higher levels. We encountered such a combination in our discussion of PD controllers, where we combined a (reactive) PD controller with a (deliberate) path planner. Architectures that combine reactive and deliberate techniques are called **hybrid architectures**.

HYBRID ARCHITECTURE

25.7.1 Subsumption architecture

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 refereed to as **augmented finite state machines**, or AFSMs, where the augmentation refers to the use of clocks.

AUGMENTED FINITE STATE MACHINE

An example of a simple AFSM is the four-state machine shown in Figure 25.24(b), which generates cyclic leg motion for a hexapod walker. 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 eighteen dimensions (twelve 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 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 towards visible light sources. Second, the lack of deliberation makes it difficult to change the task of the robot. A subsumption-style robot 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 39). Finally, subsumption-style controllers tend to be difficult to understand. In practice, the intricate interplay between dozens of interacting AFSMs (and the environment) is beyond what most human programmers can comprehend. For all these reasons, the subsumption architecture is rarely used in robotics, despite its great historical importance. However, it has had an influence on other architectures, and on individual components of some architectures.

25.7.2 Three-layer architecture

THREE-LAYER ARCHITECTURE

Hybrid architectures combine reaction with deliberation. The most popular hybrid architecture is the **three-layer architecture**, which consists of a reactive layer, an executive layer, and a deliberative layer.

REACTIVE LAYER

The **reactive layer** provides low-level control to the robot. It is characterized by a tight sensor–action loop. Its decision cycle is often on the order of milliseconds.

EXECUTIVE LAYER

The **executive layer** (or sequencing layer) serves as the glue between the reactive layer and the deliberative layer. It accepts directives by the deliberative layer, and sequences them for the reactive layer. For example, the executive layer might handle a set of via-points generated by a deliberative path planner, and make decisions as to which reactive behavior to invoke. Decision cycles at the executive layer are usually in the order of a second. The executive layer is also responsible for integrating sensor information into an internal state representation. For example, it may host the robot's localization and online mapping routines.

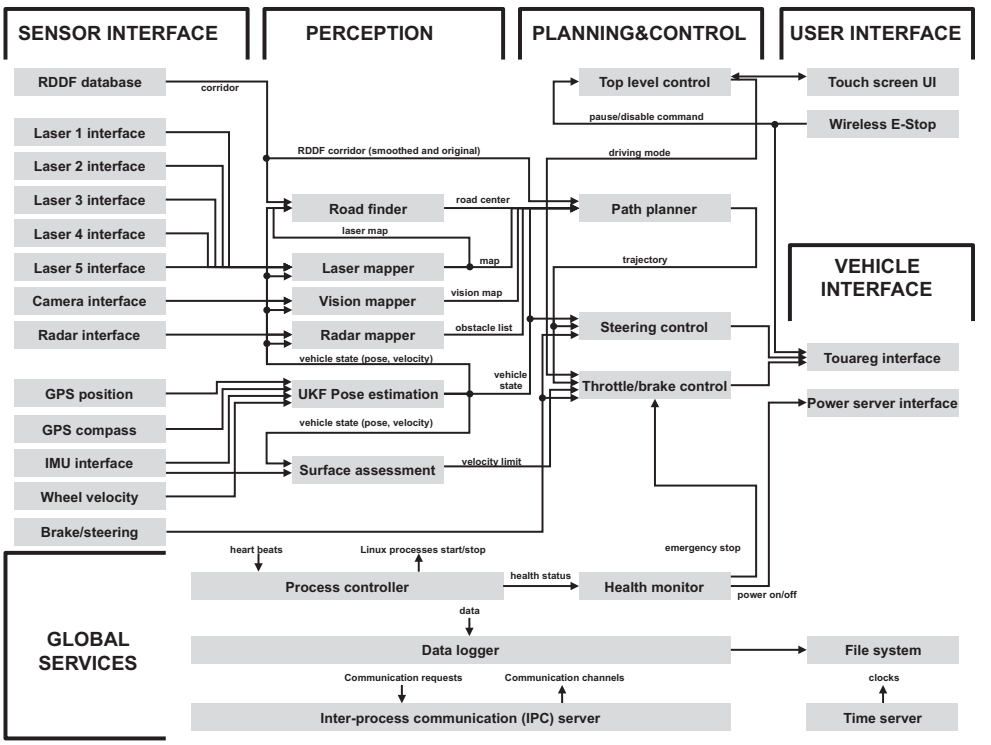


Figure 25.26 Software architecture of a robot car. This software implements a data pipeline, in which all modules process data simultaneously.

DELIBERATIVE LAYER

The **deliberative layer** generates global solutions to complex tasks using planning. Because of the computational complexity involved in generating such solutions, its decision cycle is often in the order of minutes. The deliberative layer (or planning layer) uses models for decision making. Those models might be either learned from data or supplied and may utilize state information gathered at the executive layer.

Variants of the three-layer architecture can be found in most modern-day robot software systems. The decomposition into three layers is not very strict. Some robot software systems possess additional layers, such as user interface layers that control the interaction with people, or a multiagent level for coordinating a robot's actions with that of other robots operating in the same environment.

25.7.3 Pipeline architecture

PIPELINE ARCHITECTURE

Another architecture for robots is known as the **pipeline architecture**. Just like the subsumption architecture, the pipeline architecture executes multiple process in parallel. However, the specific modules in this architecture resemble those in the three-layer architecture.

SENSOR INTERFACE LAYER

PERCEPTION LAYER

Figure 25.26 shows an example pipeline architecture, which is used to control an autonomous car. Data enters this pipeline at the **sensor interface layer**. The **perception layer**

(a) (b)

Figure 25.27 (a) The Helpmate robot transports food and other medical items in dozens of hospitals worldwide. (b) Kiva robots are part of a material-handling system for moving shelves in fulfillment centers. Image courtesy of Kiva Systems.

PLANNING AND CONTROL LAYER

VEHICLE INTERFACE LAYER

then updates the robot's internal models of the environment based on this data. Next, these models are handed to the **planning and control layer**, which adjusts the robot's internal plans turns them into actual controls for the robot. Those are then communicated back to the vehicle through the **vehicle interface layer**.

The key to the pipeline architecture is that this all happens in parallel. While the perception layer processes the most recent sensor data, the control layer bases its choices on slightly older data. In this way, the pipeline architecture is similar to the human brain. We don't switch off our motion controllers when we digest new sensor data. Instead, we perceive, plan, and act all at the same time. Processes in the pipeline architecture run asynchronously, and all computation is data-driven. The resulting system is robust, and it is fast.

The architecture in Figure 25.26 also contains other, cross-cutting modules, responsible for establishing communication between the different elements of the pipeline.

25.8 Application Domains

Here are some of the prime application domains for robotic technology.

Industry and Agriculture. Traditionally, robots have been fielded in areas that require difficult human labor, yet are structured enough to be amenable to robotic automation. The best example is the assembly line, where manipulators routinely perform tasks such as assembly, part placement, material handling, welding, and painting. In many of these tasks, robots have become more cost-effective than human workers. Outdoors, many of the heavy machines that we use to harvest, mine, or excavate earth have been turned into robots. For

(a) (b)

Figure 25.28 (a) Robotic car BOSS, which won the DARPA Urban Challenge. Courtesy of Carnegie Mellon University. (b) Surgical robots in the operating room. Image courtesy of da Vinci Surgical Systems.

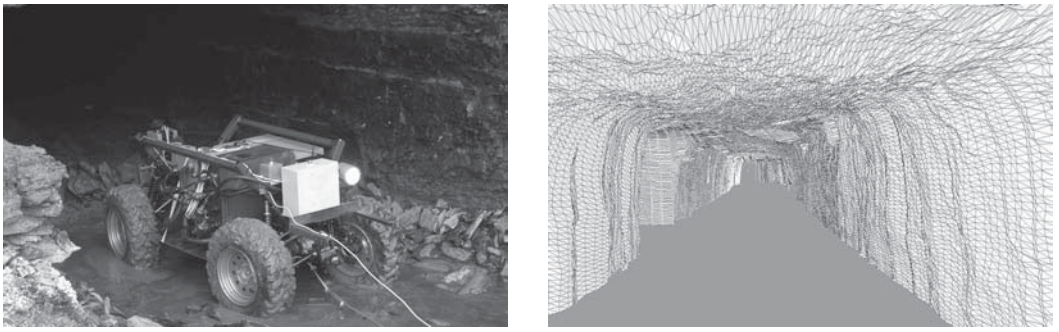
example, a project at Carnegie Mellon University has demonstrated that robots can strip paint off large ships about 50 times faster than people can, and with a much reduced environmental impact. Prototypes of autonomous mining robots have been found to be faster and more precise than people in transporting ore in underground mines. Robots have been used to generate high-precision maps of abandoned mines and sewer systems. While many of these systems are still in their prototype stages, it is only a matter of time until robots will take over much of the semimechanical work that is presently performed by people.

Transportation. Robotic transportation has many facets: from autonomous helicopters that deliver payloads to hard-to-reach locations, to automatic wheelchairs that transport people who are unable to control wheelchairs by themselves, to autonomous straddle carriers that outperform skilled human drivers when transporting containers from ships to trucks on loading docks. A prime example of indoor transportation robots, or gofers, is the Helpmate robot shown in Figure 25.27(a). This robot has been deployed in dozens of hospitals to transport food and other items. In factory settings, autonomous vehicles are now routinely deployed to transport goods in warehouses and between production lines. The Kiva system, shown in Figure 25.27(b), helps workers at fulfillment centers package goods into shipping containers.

Many of these robots require environmental modifications for their operation. The most common modifications are localization aids such as inductive loops in the floor, active beacons, or barcode tags. An open challenge in robotics is the design of robots that can use natural cues, instead of artificial devices, to navigate, particularly in environments such as the deep ocean where GPS is unavailable.

Robotic cars. Most of use cars every day. Many of us make cell phone calls while driving. Some of us even text. The sad result: more than a million people die every year in traffic accidents. Robotic cars like BOSS and STANLEY offer hope: Not only will they make driving much safer, but they will also free us from the need to pay attention to the road during our daily commute.

Progress in robotic cars was stimulated by the DARPA Grand Challenge, a race over 100 miles of unrehearsed desert terrain, which represented a much more challenging task than



(a) (b)

Figure 25.29 (a) A robot mapping an abandoned coal mine. (b) A 3D map of the mine acquired by the robot.

had ever been accomplished before. Stanford's STANLEY vehicle completed the course in less than seven hours in 2005, winning a $2 million prize and a place in the National Museum of American History. Figure 25.28(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.

Health care. Robots are increasingly used to assist surgeons with instrument placement when operating on organs as intricate as brains, eyes, and hearts. Figure 25.28(b) shows such a system. Robots have become indispensable tools in a range of surgical procedures, such as hip replacements, thanks to their high precision. In pilot studies, robotic devices have been found to reduce the danger of lesions when performing colonoscopy. Outside the operating room, researchers have begun to develop robotic aides for elderly and handicapped people, such as intelligent robotic walkers and intelligent toys that provide reminders to take medication and provide comfort. Researchers are also working on robotic devices for rehabilitation that aid people in performing certain exercises.

Hazardous environments. Robots have assisted people in cleaning up nuclear waste, most notably in Chernobyl and Three Mile Island. Robots were present after the collapse of the World Trade Center, where they entered structures deemed too dangerous for human search and rescue crews.

Some countries have used robots to transport ammunition and to defuse bombs—a notoriously dangerous task. A number of research projects are presently developing prototype robots for clearing minefields, on land and at sea. Most existing robots for these tasks are teleoperated—a human operates them by remote control. Providing such robots with autonomy is an important next step.

Exploration. Robots have gone where no one has gone before, including the surface of Mars (see Figure 25.2(b) and the cover). 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 25.29 shows a robot mapping an abandoned coal mine, along with a 3D model of the mine acquired

(a) (b)

Figure 25.30 (a) Roomba, the world's best-selling mobile robot, vacuums floors. Image courtesy of iRobot, © 2009. (b) Robotic hand modeled after human hand. Image courtesy of University of Washington and Carnegie Mellon University.

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. Unmanned air vehicles known as DRONE **drones** are used in military operations. Robots are becoming very effective tools for gathering information in domains that are difficult (or dangerous) for people to access.

Personal Services. Service is an up-and-coming application domain of robotics. Service robots assist individuals in performing daily tasks. Commercially available domestic service robots include autonomous vacuum cleaners, lawn mowers, and golf caddies. The world's most popular mobile robot is a personal service robot: the robotic vacuum cleaner ROOMBA **Roomba**, shown in Figure 25.30(a). More than three million Roombas have been sold. Roomba can navigate autonomously and perform its tasks without human help.

Other service robots operate in public places, such as robotic information kiosks that have been deployed in shopping malls and trade fairs, or in museums as tour guides. Service tasks require human interaction, and the ability to cope robustly with unpredictable and dynamic environments.

Entertainment. Robots have begun to conquer the entertainment and toy industry. ROBOTIC SOCCER In Figure 25.6(b) we see **robotic soccer**, a competitive game very much like human soccer, but played with autonomous mobile robots. Robot soccer provides great opportunities for research in AI, since it raises a range of problems relevant to many other, more serious robot applications. Annual robotic soccer competitions have attracted large numbers of AI researchers and added a lot of excitement to the field of robotics.

Human augmentation. A final application domain of robotic technology is that of human augmentation. Researchers have developed legged walking machines that can carry people around, very much like a wheelchair. Several research efforts presently focus on the development of devices that make it easier for people to walk or move their arms by providing additional forces through extraskeletal attachments. If such devices are attached permanently,

they can be thought of as artificial robotic limbs. Figure 25.30(b) shows a robotic hand that may serve as a prosthetic device in the future.

Robotic teleoperation, or telepresence, is another form of human augmentation. Teleoperation involves carrying out tasks over long distances with the aid of robotic devices. A popular configuration for robotic teleoperation is the master–slave configuration, where a robot manipulator emulates the motion of a remote human operator, measured through a haptic interface. Underwater vehicles are often teleoperated; the vehicles can go to a depth that would be dangerous for humans but can still be guided by the human operator. All these systems augment people's ability to interact with their environments. Some projects go as far as replicating humans, at least at a very superficial level. Humanoid robots are now available commercially through several companies in Japan.

25.9 SUMMARY

Robotics concerns itself with intelligent agents that manipulate the physical world. In this chapter, we have learned the following basics of robot hardware and software.

- Robots are equipped with **sensors** for perceiving their environment and effectors with which they can assert physical forces on their environment. Most robots are either manipulators anchored at fixed locations or mobile robots that can move.
- 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. Common examples of hard perceptual problems include **localization, mapping, and object recognition**.
- **Probabilistic filtering algorithms** such as Kalman filters and particle filters are useful for robot perception. These techniques maintain the belief state, a posterior distribution over state variables.
- The planning of robot motion is usually done in **configuration space**, where each point specifies the location and orientation of the robot and its joint angles.
- Configuration space search algorithms include **cell decomposition** techniques, which decompose the space of all configurations into finitely many cells, and **skeletonization** techniques, which project configuration spaces onto lower-dimensional manifolds. The motion planning problem is then solved using search in these simpler structures.
- A path found by a search algorithm can be executed by using the path as the reference trajectory for a **PID controller**. Controllers are necessary in robotics to accommodate small perturbations; path planning alone is usually insufficient.
- **Potential field** techniques navigate robots by potential functions, defined over the distance to obstacles and the goal location. Potential field techniques may get stuck in local minima, but they can generate motion directly without the need for path planning.
- Sometimes it is easier to specify a robot controller directly, rather than deriving a path from an explicit model of the environment. Such controllers can often be written as simple **finite state machines**.

- There exist different architectures for software design. The **subsumption architecture** enables programmers to compose robot controllers from interconnected finite state machines. **Three-layer architectures** are common frameworks for developing robot software that integrate deliberation, sequencing of subgoals, and control. The related **pipeline architecture** processes data in parallel through a sequence of modules, corresponding to perception, modeling, planning, control, and robot interfaces.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The word **robot** was popularized by Czech playwright Karel Capek in his 1921 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 (Glanc, 1978) it was Capek's brother, Josef, who first combined the Czech words "robota" (obligatory work) and "robotnik" (serf) to yield "robot" in his 1917 short story *Opilec*.

The term *robotics* was first used by Asimov (1950). Robotics (under other names) has a much longer history, however. In ancient Greek mythology, a mechanical man named Talos was supposedly designed and built by Hephaistos, the Greek god of metallurgy. 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 14.

UNIMATE

The first commercial robot was a robot arm called **Unimate**, short for *universal automation*, developed by Joseph Engelberger and George Devol. In 1961, the first Unimate robot was sold to General Motors, where it was used for manufacturing TV picture tubes. 1961 was also the year when Devol obtained the first U.S. patent on a robot. Eleven years later, in 1972, Nissan Corp. was among the first to automate an entire assembly line with robots, developed by Kawasaki with robots supplied by Engelberger and Devol's company Unimation. This development initiated a major revolution 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, short for Programmable Universal Machine for Assembly. The PUMA robot, initially developed for General Motors, was the *de facto* standard for robotic manipulation for the two decades that followed. At present, the number of operating robots is estimated at one million worldwide, more than half of which are installed in Japan.

PUMA

The literature on robotics research can be divided roughly into two parts: mobile robots and stationary manipulators. 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 the early 1960s at Johns Hopkins University, was much more sophisticated; it had 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) describes classic work on autonomous vehicles.

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 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). 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).

OCCUPANCY GRID

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, 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

The study of manipulator robots, originally called **hand–eye machines**, has evolved along quite different lines. 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). After these pioneering efforts, a great deal of work 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.

HAND–EYE MACHINES

PIANO MOVERS

Recursive cell decomposition for configuration space planning was originated by Brooks and Lozano-Perez (1985) and improved significantly by Zhu and Latombe (1991). The ear-

liest 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. 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.* (2004) and LaValle (2006). Kavraki *et al.* (1996) developed probabilistic roadmaps, which are currently one of the most effective methods. 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. Key work applying POMDP methods (Section 17.4) to motion planning under uncertainty in robotics is due to Pineau *et al.* (2003) and Roy *et al.* (2005).

VISIBILITY GRAPH

The control of robots as dynamical systems—whether for manipulation or navigation—has generated a huge literature that is barely touched on by this chapter. Important works 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). 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).

GRASPING

HAPTIC FEEDBACK

Potential-field control, which attempts to solve the motion planning and control problems simultaneously, was introduced into the robotics literature 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). Navigation functions, the robotics version of a control policy for deterministic MDPs, were introduced by Koditschek (1987). Reinforcement learning in robotics took off with the seminal work by Bagnell and Schneider (2001) and Ng *et al.* (2004), who developed the paradigm in the context of autonomous helicopter control.

VECTOR FIELD HISTOGRAMS

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 (1984), whose book, *Vehicles*, 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 a mobile robot capable of retrieving objects that was entirely reactive. Extensions of the behavior-based paradigm to multirobot systems can be found in (Mataric, 1997) and (Parker, 1996). 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.

Research on mobile robotics has been stimulated over the last decade by several important competitions. The earliest competition, 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 more 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 (1997a), aims to "develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer" by 2050. Play occurs in leagues for simulated robots, wheeled robots of different sizes, and humanoid robots. In 2009 teams from 43 countries participated and the event was broadcast to millions of viewers. Visser and Burkhard (2007) track the improvements that have been made in perception, team coordination, and low-level skills over the past decade.

ROBOCUP

DARPA GRAND CHALLENGE

The **DARPA Grand Challenge**, organized by DARPA in 2004 and 2005, required autonomous robots to travel more than 100 miles through unrehearsed desert terrain in less than 10 hours (Buehler *et al.*, 2006). In the original event in 2004, no robot traveled more than 8 miles, leading many to believe the prize would never be claimed. In 2005, Stanford's robot STANLEY won the competition in just under 7 hours of travel (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).

URBAN CHALLENGE

Two early textbooks, by Dudek and Jenkin (2000) and Murphy (2000), cover robotics generally. A more recent overview is due to Bekey (2008). An excellent book on robot manipulation addresses advanced topics such as compliant motion (Mason, 2001). Robot motion planning is covered in Choset *et al.* (2004) and LaValle (2006). Thrun *et al.* (2005) provide an introduction into probabilistic robotics. The premiere conference for robotics is Robotics: Science and Systems Conference, followed by the IEEE International Conference on Robotics and Automation. Leading robotics journals include *IEEE Robotics and Automation*, the *International Journal of Robotics Research*, and *Robotics and Autonomous Systems*.

EXERCISES

25.1 Monte Carlo localization is *biased* for any finite sample size—i.e., the expected value of the location computed by the algorithm differs from the true expected value—because of the way particle filtering works. In this question, you are asked to quantify this bias.

To simplify, consider a world with four possible robot locations: $X = \{x_1, x_2, x_3, x_4\}$. Initially, we draw $N \geq 1$ samples uniformly from among those locations. As usual, it is perfectly acceptable if more than one sample is generated for any of the locations X. Let Z be a Boolean sensor variable characterized by the following conditional probabilities:

$$\begin{aligned} P(z \mid x_1) &= 0.8 & P(\neg z \mid x_1) &= 0.2 \\ P(z \mid x_2) &= 0.4 & P(\neg z \mid x_2) &= 0.6 \\ P(z \mid x_3) &= 0.1 & P(\neg z \mid x_3) &= 0.9 \\ P(z \mid x_4) &= 0.1 & P(\neg z \mid x_4) &= 0.9\,. \end{aligned}$$

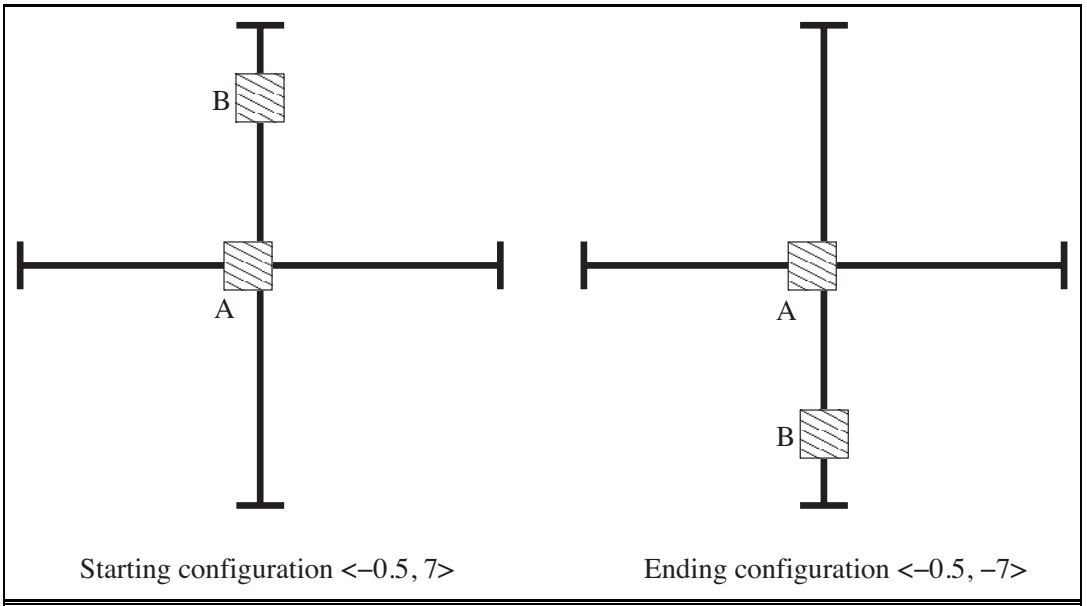


Figure 25.31 A Robot manipulator in two of its possible configurations.

MCL uses these probabilities to generate particle weights, which are subsequently normalized and used in the resampling process. For simplicity, let us assume we generate only one new sample in the resampling process, regardless of N. This sample might correspond to any of the four locations in X. Thus, the sampling process defines a probability distribution over X.

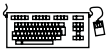
a. What is the resulting probability distribution over X for this new sample? Answer this question separately for $N = 1, \ldots, 10$, and for $N = \infty$.

b. The difference between two probability distributions P and Q can be measured by the KL divergence, which is defined as

$$KL(P, Q) = \sum_i P(x_i) \log \frac{P(x_i)}{Q(x_i)} .$$

What are the KL divergences between the distributions in (a) and the true posterior?

c. What modification of the problem formulation (not the algorithm!) would guarantee that the specific estimator above is unbiased even for finite values of N? Provide at least two such modifications (each of which should be sufficient).



25.2 Implement Monte Carlo localization for a simulated robot with range sensors. A grid map and range data are available from the code repository at `aima.cs.berkeley.edu`. You should demonstrate successful global localization of the robot.

25.3 Consider a robot with two simple manipulators, as shown in figure 25.31. Manipulator A is a square block of side 2 which can slide back and on a rod that runs along the x-axis from x=-10 to x=10. Manipulator B is a square block of side 2 which can slide back and on a rod that runs along the y-axis from y=-10 to y=10. The rods lie outside the plane of

manipulation, so the rods do not interfere with the movement of the blocks. A configuration is then a pair $\langle x, y \rangle$ where x is the x-coordinate of the center of manipulator A and where y is the y-coordinate of the center of manipulator B. Draw the configuration space for this robot, indicating the permitted and excluded zones.

25.4 Suppose that you are working with the robot in Exercise 25.3 and you are given the problem of finding a path from the starting configuration of figure 25.31 to the ending configuration. Consider a potential function

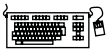
$$D(A, Goal)^2 + D(B, Goal)^2 + \frac{1}{D(A, B)^2}$$

where $D(A, B)$ is the distance between the closest points of A and B.

a. Show that hill climbing in this potential field will get stuck in a local minimum.

b. Describe a potential field where hill climbing will solve this particular problem. You need not work out the exact numerical coefficients needed, just the general form of the solution. (Hint: Add a term that "rewards" the hill climber for moving A out of B's way, even in a case like this where this does not reduce the distance from A to B in the above sense.)

25.5 Consider the robot arm shown in Figure 25.14. Assume that the robot's base element is 60cm long and that its upper arm and forearm are each 40cm long. As argued on page 987, the inverse kinematics of a robot is often not unique. State an explicit closed-form solution of the inverse kinematics for this arm. Under what exact conditions is the solution unique?



25.6 Implement an algorithm for calculating the Voronoi diagram of an arbitrary 2D environment, described by an $n \times n$ Boolean array. Illustrate your algorithm by plotting the Voronoi diagram for 10 interesting maps. What is the complexity of your algorithm?

25.7 This exercise explores the relationship between workspace and configuration space using the examples shown in Figure 25.32.

a. Consider the robot configurations shown in Figure 25.32(a) through (c), ignoring the obstacle shown in each of the diagrams. Draw the corresponding arm configurations in configuration space. (*Hint:* Each arm configuration maps to a single point in configuration space, as illustrated in Figure 25.14(b).)

b. Draw the configuration space for each of the workspace diagrams in Figure 25.32(a)–(c). (*Hint:* The configuration spaces share with the one shown in Figure 25.32(a) the region that corresponds to self-collision, but differences arise from the lack of enclosing obstacles and the different locations of the obstacles in these individual figures.)

c. For each of the black dots in Figure 25.32(e)–(f), draw the corresponding configurations of the robot arm in workspace. Please ignore the shaded regions in this exercise.

d. The configuration spaces shown in Figure 25.32(e)–(f) have all been generated by a single workspace obstacle (dark shading), plus the constraints arising from the self-collision constraint (light shading). Draw, for each diagram, the workspace obstacle that corresponds to the darkly shaded area.

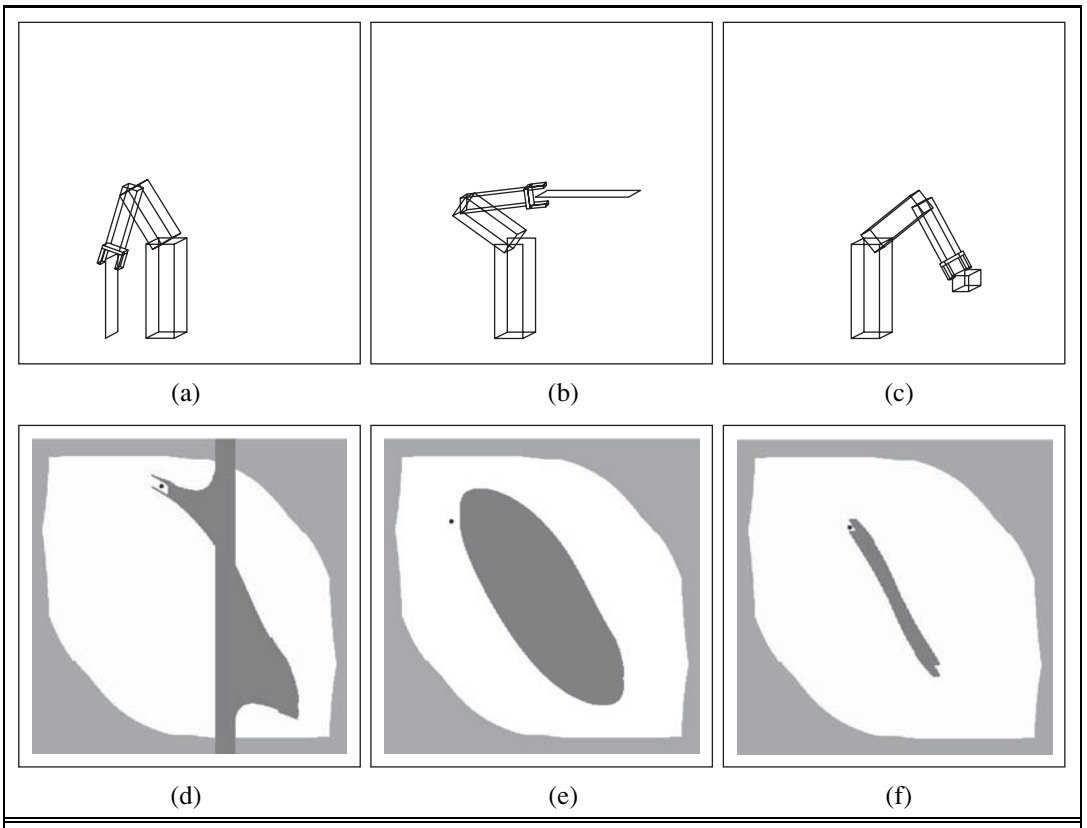


Figure 25.32 Diagrams for Exercise 25.7.

e. Figure 25.32(d) illustrates that a single planar obstacle can decompose the workspace into two disconnected regions. What is the maximum number of disconnected regions that can be created by inserting a planar obstacle into an obstacle-free, connected workspace, for a 2DOF robot? Give an example, and argue why no larger number of disconnected regions can be created. How about a non-planar obstacle?

25.8 Consider a mobile robot moving on a horizontal surface. Suppose that the robot can execute two kinds of motions:

- Rolling forward a specified distance.
- Rotating in place through a specified angle.

The state of such a robot can be characterized in terms of three parameters $\langle x, y, \phi$, the x-coordinate and y-coordinate of the robot (more precisely, of its center of rotation) and the robot's orientation expressed as the angle from the positive x direction. The action "$Roll(D)$" has the effect of changing state $\langle x, y, \phi$ to $\langle x + D\cos(\phi), y + D\sin(\phi), \phi\rangle$, and the action $Rotate(\theta)$ has the effect of changing state $\langle x, y, \phi\rangle$ to $\langle x, y, \phi + \theta\rangle$.

a. Suppose that the robot is initially at $\langle 0, 0, 0\rangle$ and then executes the actions $Rotate(60^\circ)$, $Roll(1)$, $Rotate(25^\circ)$, $Roll(2)$. What is the final state of the robot?

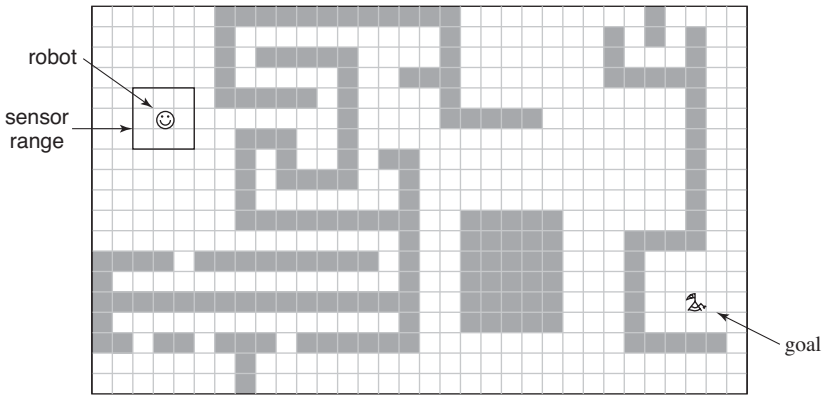


Figure 25.33 Simplified robot in a maze. See Exercise 25.9.

b. Now suppose that the robot has imperfect control of its own rotation, and that, if it attempts to rotate by θ, it may actually rotate by any angle between $\theta - 10°$ and $\theta + 10°$. In that case, if the robot attempts to carry out the sequence of actions in (A), there is a range of possible ending states. What are the minimal and maximal values of the x-coordinate, the y-coordinate and the orientation in the final state?

c. Let us modify the model in (B) to a probabilistic model in which, when the robot attempts to rotate by θ, its actual angle of rotation follows a Gaussian distribution with mean θ and standard deviation $10°$. Suppose that the robot executes the actions $Rotate(90°)$, $Roll(1)$. Give a simple argument that (a) the expected value of the location at the end is not equal to the result of rotating exactly $90°$ and then rolling forward 1 unit, and (b) that the distribution of locations at the end does not follow a Gaussian. (Do not attempt to calculate the true mean or the true distribution.)

The point of this exercise is that rotational uncertainty quickly gives rise to a lot of positional uncertainty and that dealing with rotational uncertainty is painful, whether uncertainty is treated in terms of hard intervals or probabilistically, due to the fact that the relation between orientation and position is both non-linear and non-monotonic.

25.9 Consider the simplified robot shown in Figure 25.33. Suppose the robot's Cartesian coordinates are known at all times, as are those of its goal location. However, the locations of the obstacles are unknown. The robot can sense obstacles in its immediate proximity, as illustrated in this figure. For simplicity, let us assume the robot's motion is noise-free, and the state space is discrete. Figure 25.33 is only one example; in this exercise you are required to address all possible grid worlds with a valid path from the start to the goal location.

a. Design a deliberate controller that guarantees that the robot always reaches its goal location if at all possible. The deliberate controller can memorize measurements in the form of a map that is being acquired as the robot moves. Between individual moves, it may spend arbitrary time deliberating.

b. Now design a *reactive* controller for the same task. This controller may not memorize past sensor measurements. (It may not build a map!) Instead, it has to make all decisions based on the current measurement, which includes knowledge of its own location and that of the goal. The time to make a decision must be independent of the environment size or the number of past time steps. What is the maximum number of steps that it may take for your robot to arrive at the goal?

c. How will your controllers from (a) and (b) perform if any of the following six conditions apply: continuous state space, noise in perception, noise in motion, noise in both perception and motion, unknown location of the goal (the goal can be detected only when within sensor range), or moving obstacles. For each condition and each controller, give an example of a situation where the robot fails (or explain why it cannot fail).

25.10 In Figure 25.24(b) on page 1001, we encountered an augmented finite state machine for the control of a single leg of a hexapod robot. In this exercise, the aim is to design an AFSM that, when combined with six copies of the individual leg controllers, results in efficient, stable locomotion. For this purpose, you have to augment the individual leg controller to pass messages to your new AFSM and to wait until other messages arrive. Argue why your controller is efficient, in that it does not unnecessarily waste energy (e.g., by sliding legs), and in that it propels the robot at reasonably high speeds. Prove that your controller satisfies the dynamic stability condition given on page 977.

25.11 (This exercise was first devised by Michael Genesereth and Nils Nilsson. It works for first graders through graduate students.) Humans are so adept at basic household tasks that they often forget how complex these tasks are. In this exercise you will discover the complexity and recapitulate the last 30 years of developments in robotics. Consider the task of building an arch out of three blocks. Simulate a robot with four humans as follows:

Brain. The Brain direct the hands in the execution of a plan to achieve the goal. The Brain receives input from the Eyes, but *cannot see the scene directly*. The brain is the only one who knows what the goal is.

Eyes. The Eyes report a brief description of the scene to the Brain: "There is a red box standing on top of a green box, which is on its side" Eyes can also answer questions from the Brain such as, "Is there a gap between the Left Hand and the red box?" If you have a video camera, point it at the scene and allow the eyes to look at the viewfinder of the video camera, but not directly at the scene.

Left hand and **right hand.** One person plays each Hand. The two Hands stand next to each other, each wearing an oven mitt on one hand, Hands execute only simple commands from the Brain—for example, "Left Hand, move two inches forward." They cannot execute commands other than motions; for example, they cannot be commanded to "Pick up the box." The Hands must be *blindfolded*. The only sensory capability they have is the ability to tell when their path is blocked by an immovable obstacle such as a table or the other Hand. In such cases, they can beep to inform the Brain of the difficulty.