Joseph Lorenzetti, Marco Pavone

Principles of Robot Autonomy

OCTOBER 3, 2023

Forward

This collection of notes is meant to provide a fundamental understanding of the theoretical and algorithmic aspects associated with robotic autonomy[1]. In particular, these notes cover topics spanning the three main pillars of autonomy: motion planning and control, perception, and decision-making, and also include some information on useful software tools for robot programming, such as the Robot Operating System (ROS). By avoiding extremely in-depth discussions on specific algorithms or techniques, these notes focus on providing a high-level understanding of the full "autonomy stack" and are a good starting point for any engineer or researcher interested in robotics. Some other great references that cover a wide range of robotics topics include:

[1] The field of robotic autonomy is vast and diverse, encompassing theory and algorithms from many fields of science, technology, and engineering. These notes cannot cover all material and therefore focuses on the most foundational and widely used techniques.

R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, 2011

S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005

While these notes are meant to be as self-contained as is practical, prior knowledge of several topics is generally assumed. Specifically, familiarity with the basics of calculus, differential equations, linear algebra, probability and statistics, and programming is helpful.

Acknowledgments

These notes accompany (and are based largely on the content of) the courses *AA274A: Principles of Robot Autonomy I* and *AA274B: Principles of Robot Autonomy II*[2] at Stanford University. We would therefore like to acknowledge the students who have taken the course and provided useful feedback since its initial offering in 2017. Special acknowledgements are also reserved for the course assistants:
AA274A, Winter 2017: Andrew Bylard, Benoit Landry, Ed Schmerling,
AA274A, Winter 2018: Tommy Hu, Benoit Landry, Karen Leung, Ed Schmerling,
AA274A, Winter 2019: Christopher Covert, Amine Elhafsi, Karen Leung, Apoorva Sharma,
AA274A, Autumn 2019: Andrew Bylard, Boris Ivanovic, Jenna Lee, Toki Migimatsu, Apoorva Sharma,
AA274A, Autumn 2020: Somrita Banerjee, Abhishek Cauligi, Boris Ivanovic, Mengxi Li, Joseph Lorenzetti,
AA274B, Winter 2020: Ashar Alam, Erdem Bıyık, Jenna Lee, Toki Migimatsu,
AA274B, Winter 2021: Erdem Bıyık, Abhishek Cauligi,
who were instrumental in developing and refining the course material. In large part, additional material for the course such as homework and lectures are also publicly available[3].

[2] Co-taught with Professors Jeannette Bogh and Dorsa Sadigh.

[3] `https://github.com/PrinciplesofRobotAutonomy/CourseMaterials`

Contents

PART III ROBOT LOCALIZATION

Part I

Robot Motion Planning & Control

1
Mobile Robot Kinematics

Mobile Robot Kinematics

Motion planning and control are fundamental components of robotic autonomy[1]. For example, in order for an autonomous car to move from place to place it must plan a trajectory and determine what control inputs, such as throttle and steering, will enable it to follow the trajectory. Robotic grasping and object manipulation[2] tasks provide another classic example, where motion plans that specify how to grasp an object are executed by controlling actuators. Both of these components, motion planning and control, require an understanding of the physical behavior of the robot in order to develop reasonable and actionable plans and controls. In the context of motion planning and control, a robot's physical behavior is characterized by its *dynamics* and *kinematics*.

[1] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, 2011

[2] D. Prattichizzo and J. C. Trinkle. "Grasping". In: *Springer Handbook of Robotics*. Springer, 2016, pp. 955–988

Definition 1.0.1 (Dynamics)**.** *A robot's dynamics describe the relationship between forces acting on the robot and changes to the robot's physical state.*

In other words, dynamics can be thought of as the result of Newton's Second Law ($F = ma$) in the context of a particular robot. For example, the dynamics of an autonomous car are characterized by the relationship between its acceleration and external forces such as tire friction, gravity, and aerodynamics.

Definition 1.0.2 (Kinematics)**.** *A robot's kinematics describe additional restrictions (constraints) on the robot's motion that are not induced by forces.*

A trivial example of a kinematic constraint is that the rate of change of the robot's position must equal its velocity. More generally, a robot's kinematics describe limitations on its motion that are a function of the robot's physical state or geometry. For example, a robotic arm with multiple joints is kinematically constrained by the rigid connections at each joint which only allow rotation about a single axis, and static friction kinematically restricts a robot's wheels from moving in the direction parallel to the rotation axis.

[3] One simple heuristic for determining how a particular constraint/relationship should be classified is to remember that dynamics are affected by changing the robot's mass, while kinematics are *not*.

We can see from these examples that a robot's dynamics and kinematics describe limitations on its motion in different ways[3], and are defined by the robot's design, geometry, mass, and other physical characteristics. Therefore,

we must first study and understand a particular robot's dynamics and kinematics before we can begin to design motion planning and control algorithms. Specifically, we should try to answer the following questions:

1. What kinematic constraints are imposed by the robot's construction and design, and what constraints are imposed by its interactions with the environment?
2. What internal and external forces can act on the robot to make it move? Which laws of physics are relevant for describing how these forces can cause motion and deformation of the robot's physical structure?
3. Are all of the robot's dynamics and kinematics important or significant? Are they all relevant to the motion planning and control task we are trying to accomplish?
4. Which of the dynamics and kinematics relationships can be simplified, approximated, or ignored to make the motion planning and control task easier?

From the relevant dynamics and kinematics relationships we can produce a mathematical *model* of the physical behavior of the robot.

Definition 1.0.3 (Kinematic (Dynamic) Model)**.** *A kinematic (dynamic) model is a mathematical representation*[4] *of the kinematic (dynamic) relationships governing a particular robot.*

[4] Typically in the form of a set of ordinary differential equations.

If we determine that it is acceptable to make simplifications to the model, we might say our model is *low-fidelity*, and alternatively if we make very few approximations we might say our model is *high-fidelity*. In practice, the fidelity of the model is generally defined through an analysis of the trade-offs between complexity and accuracy.

For an autonomous car, an extremely high-fidelity model could include engine combustion dynamics, suspension dynamics, tire deformation dynamics, aerodynamics, and more. Incorporating high-fidelity tire dynamics models may be critical for accurately capturing the phenomenon of drifting, which is important for motion planning and control in autonomous racing applications. However, in an autonomous grocery delivery service application we might find it beneficial to simplify the model by replacing the tire *dynamics* with a simple *kinematic* constraint that the tires cannot move laterally[5], which could simplify the algorithms needed for planning and control.

[5] This constraint is typically referred to as a *no side-slip* constraint, and is very common in basic vehicle models.

In fact, in the context of motion planning and control for robotics, models built entirely from kinematics can be very useful (and much simpler). For this reason this chapter specifically focuses on robot kinematics, and in particular:

1. How to express the configuration of a robot in terms of *general coordinates*
2. How to mathematically express kinematic constraints in terms of general coordinates

3. How to identify different types of kinematic constraints, namely *holonomic* and *nonholonomic* constraints
4. Examples of kinematic models, specifically for wheeled robots

1.1 Generalized Coordinates

A robot's physical state (also commonly referred to as its "configuration") can usually be represented (i.e. quantified) in different ways. The particular choice of representation defines a finite set of numbers known as *generalized coordinates*.

Definition 1.1.1 (Generalized Coordinates)**.** *Generalized coordinates refer to a set of coordinates that can completely specify the unique position of your robot.*

For example, the wheel rolling on a plane in Figure 1.1 can be represented by three parameters, $x, y,$ and θ, where (x, y) indicates the position at which the wheel touches the ground, and θ indicates the direction the wheel is traveling in the general frame. This set of parameters (x, y, θ) that define the wheel's configuration are generalized coordinates for this system. Note that in practice people often use "configuration" and "generalized coordinates" interchangeably, even though the specific choice of generalized coordinates are not necessarily the only possible representation of the robot's configuration.

The generalized coordinates are mathematically expressed by the vector $\boldsymbol{\xi} \in \mathbb{R}^n$, where n is the number of generalized coordinates used to describe the robot's configuration. A robot's motion through time (i.e. its trajectory) is then expressed by the function

$$\boldsymbol{\xi}(t) : \mathbb{R} \to \mathbb{R}^n,$$

where t denotes time. In the case of the wheel in Figure 1.1 the generalized coordinate vector would be $\boldsymbol{\xi} = \begin{bmatrix} x & y & \theta \end{bmatrix}^\top$.

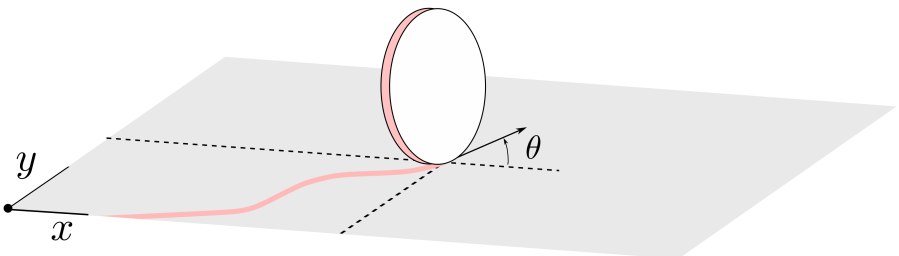


Figure 1.1: Generalized coordinates for a wheel rolling without slipping on a plane.

1.2 Kinematic Constraints

Once a set of generalized coordinates $\boldsymbol{\xi}$ has been identified, they can be used to mathematically define kinematic constraints that define a robot's motion. A more formal definition of general kinematic constraints is first presented:

Definition 1.2.1 (Kinematic Constraints)**.** *Let the generalized coordinates of a robot be denoted as* $\boldsymbol{\xi} = [\xi_1, \ldots, \xi_n]^\top$*. Constraints that depend on these generalized coordinates and their velocities are called kinematic constraints and are expressed as*

$$a_i(\boldsymbol{\xi}, \dot{\boldsymbol{\xi}}) = 0, \qquad i = 1, \ldots, k < n \tag{1.1}$$

where $\dot{\boldsymbol{\xi}} = \frac{d\boldsymbol{\xi}}{dt}$ *are the velocities.*

Kinematic constraints in robotics applications are often linear with respect to the generalized velocities. Constraints of this kind are referred to as *Pfaffian constraints* and are expressed as

$$a_i^\top(\boldsymbol{\xi})\dot{\boldsymbol{\xi}} = 0, \qquad i = 1, \ldots, k < n \tag{1.2}$$

where $a_i(\boldsymbol{\xi}) \in \mathbb{R}^n$. For notational simplicity these constraints can be compactly expressed in matrix form as

$$A^\top(\boldsymbol{\xi})\dot{\boldsymbol{\xi}} = \mathbf{0}, \tag{1.3}$$

where $A(\boldsymbol{\xi}) \in \mathbb{R}^{n \times k}$.

Example 1.2.1 (Pendulum)**.** Figure 1.2 shows a simple pendulum that is assumed to rotate about a fixed pivot point. Let the position of the mass be given by the Cartesian coordinates (x, y), which can be used as the generalized coordinates for this system (i.e. $\boldsymbol{\xi} = [x, y]^\top$). Since the rod connecting the pivot point to the mass is assumed to be rigid this implies a kinematic constraint. Assuming the pivot point is at the origin $(0, 0)$ this constraint can be expressed as

$$h_1(\boldsymbol{\xi}) = x^2 + y^2 - L^2 = 0, \tag{1.4}$$

where L is the length of the rod. Note that while this does not appear to be a Pfaffian constraint, it can be equivalently expressed as one. In particular, consider the derivative of the expression with respect to time, which yields the Pfaffian constraint

$$\frac{dh_1(\boldsymbol{\xi})}{dt} = \frac{dh_1(\boldsymbol{\xi})}{d\boldsymbol{\xi}}\dot{\boldsymbol{\xi}} = 2x\dot{x} + 2y\dot{y} = 0, \tag{1.5}$$

where we make the identification

$$\frac{dh_1(\boldsymbol{\xi})}{d\boldsymbol{\xi}} = \begin{bmatrix} 2x & 2y \end{bmatrix} := a_1^\top(\boldsymbol{\xi}).$$

The satisfaction of (1.5) implies that (1.4) holds for all time as long as the pendulum starts in a state $\boldsymbol{\xi}(0)$ satisfying $h_1(\boldsymbol{\xi}(0)) = 0$.

In this particular case a more natural choice of coordinates would simply be $\boldsymbol{\xi} = [\theta]$, which also fully specifies the system's configuration and eliminates the need to enforce additional kinematic constraints. In fact, it can be noted that since $x = L\sin\theta$ and $y = -L\cos\theta$ that the above kinematic constraint is trivially satisfied for all θ.

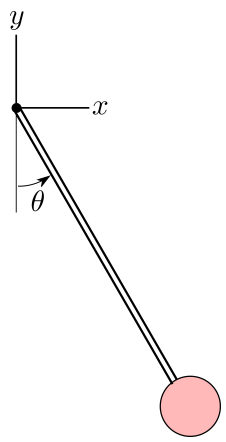


Figure 1.2: Generalized coordinates for a simple pendulum.

Example 1.2.2 (No-Slip Wheel)**.** Consider again the wheel illustrated in Figure 1.1 with generalized coordinates $\boldsymbol{\xi} = [x, y, \theta]^\top$, and assume that there is a no-slip condition between the wheel and the plane it rolls on. This no-slip condition means that the velocity component of the wheel in the lateral direction is always zero. Since the heading of the wheel is given by the unit vector $e_v = [\cos\theta, \sin\theta]^\top$, the lateral direction can be described by the perpendicular unit vector $e_{v,\perp} = [\sin\theta, -\cos\theta]^\top$.

Since the velocity vector is $v = [\dot{x}, \dot{y}]^\top$, the no-slip kinematic constraint can be expressed by the inner product $v \cdot e_{v,\perp} = 0$, which is equivalently expressed as

$$a_1(\boldsymbol{\xi}, \dot{\boldsymbol{\xi}}) = \dot{x}\sin\theta - \dot{y}\cos\theta = 0. \tag{1.6}$$

Note that this constraint is linear in the generalized velocities $(\dot{x}, \dot{y})$ and therefore is a Pfaffian constraint.

1.3 Holonomic and Nonholonomic Constraints

It is useful to further classify different types of kinematic constraints based on how they restrict the motion of the system. In particular, the most common classifications for kinematic constraints are *holonomic* or *nonholonomic*.

1.3.1 Holonomic Constraints

Holonomic constraints are kinematic constraints that can be expressed as a function of *only* the generalized coordinates (without dependence on generalized velocities). In robotics applications, holonomic constraints generally arise due to mechanical interconnections, such as rigid links and joints of a robotic arm.

Definition 1.3.1 (Holonomic Constraints)**.** *Constraints that can be expressed in the form*

$$h_i(\boldsymbol{\xi}) = 0, \quad i = 1, \ldots, k < n \tag{1.7}$$

are called holonomic.

Additionally, a *holonomic system* is a system that is only subject to holonomic constraints. Note that these constraints can *always* be equivalently expressed as Pfaffian constraints of the form (1.2) by differentiating the expression

$$\frac{dh_i(\boldsymbol{\xi})}{dt} = \frac{dh_i(\boldsymbol{\xi})}{d\boldsymbol{\xi}}\dot{\boldsymbol{\xi}} = a_i^\top(\boldsymbol{\xi})\dot{\boldsymbol{\xi}} = 0. \quad i = 1, \ldots, k < n \tag{1.8}$$

However, it is important to note that not all Pfaffian constraints are holonomic. A Pfaffian constraint is only holonomic if it is *integrable* to the form (1.7).

Holonomic constraints are a unique subclass of kinematic constraints that *restrict the accessible configurations of the system*. In fact, the space of accessible configurations for a system with n generalized coordinates under k holonomic constraints will have dimension $n - k$.

Examples: Consider again the pendulum from Example 1.2.1, where the kinematic constraint (1.4) can be expressed as $h_i(\boldsymbol{\xi}) = 0$ (equivalently where the Pfaffian constraint (1.5) is integrable into the form $h_i(\boldsymbol{\xi}) = 0$). This constraint restricts the pendulum mass to lie on a circle of radius L, which is a one dimensional subset ($n - k = 2 - 1 = 1$).

Alternatively, consider the wheel from Example 1.2.2, where the kinematic constraint (1.6) *cannot* be integrated to yield a constraint of the form $h_i(\boldsymbol{\xi}) = 0$. In contrast to the pendulum, this system has no restriction on what configuration it can be in as it can potentially move to any point (x, y).

1.3.2 Nonholonomic Constraints

While holonomic constraints are kinematic constraints which restrict the accessible configurations of the system, not all kinematic constraints are holonomic. In particular, it is possible to have kinematic constraints that *do not* restrict accessible configurations, but rather restrict the motion *between* configurations. These constraints are referred to as *nonholonomic* constraints.

Definition 1.3.2 (Nonholonomic Constraints)**.** *Constraints that can be described in Pfaffian form, but cannot be integrated to $h_i(\boldsymbol{\xi}) = 0$ form are called nonholonomic.*

Additionally, a *nonholonomic system* is a system that is subject to at least one nonholonomic constraint. The restriction of instantaneous motion that is induced by a nonholonomic constraint can be interpreted by considering the Pfaffian form $a_i(\boldsymbol{\xi})^\top \dot{\boldsymbol{\xi}} = 0$. It is clear that for any coordinate $\boldsymbol{\xi}$, this constraint limits the motion ($\dot{\boldsymbol{\xi}}$) to lie in the null space of $a_i(\boldsymbol{\xi})^\top$.

Examples: Consider again the wheel example from Example 1.2.2 which has a nonholonomic constraint

$$a_i(\boldsymbol{\xi})^\top \dot{\boldsymbol{\xi}} = \begin{bmatrix} \sin\theta & -\cos\theta & 0 \end{bmatrix} \dot{\boldsymbol{\xi}} = 0.$$

The null space of $a_i(\boldsymbol{\xi})^\top$ in this case is spanned by the vectors $[\cos\theta,\ \sin\theta,\ 0]$ and $[0,\ 0,\ 1]$ which suggests that any potential motion must be made up of a linear combination of these vectors. Intuitively this would be expected because $[\cos\theta,\ \sin\theta,\ 0]$ is the unit vector in the direction of rolling, and $[0,\ 0,\ 1]$ would correspond to the wheel spinning but not rolling.

1.4 Kinematic Models

Once an appropriate set of generalized coordinates $\boldsymbol{\xi} \in \mathbb{R}^n$ and all relevant kinematic constraints have been identified for a particular robot the next step is to develop a kinematic model. In particular these kinematic models will consist of a set of differential equations of the form $\dot{\boldsymbol{\xi}}(t) = G(\boldsymbol{\xi}(t))\boldsymbol{u}(t)$, where $\boldsymbol{u}(t)$ is referred to as a system input or *control*. Given a particular input $\boldsymbol{u}(t)$ and an initial condition $\boldsymbol{\xi}(0)$ this model will define a trajectory of the system.

Definition 1.4.1 (Kinematic Model)**.** *Given a generalized coordinate vector $\boldsymbol{\xi} \in \mathbb{R}^n$ and k Pfaffian kinematic constraints $A^\top(\boldsymbol{\xi})\dot{\boldsymbol{\xi}} = \mathbf{0}$, a kinematic model can be defined as $\dot{\boldsymbol{\xi}} = G(\boldsymbol{\xi})u$ where the column space of $G(\boldsymbol{\xi}) \in \mathbb{R}^{n\times n-k}$ spans the null space of $A^\top(\boldsymbol{\xi})$. Additionally, for any input u the solutions to the kinematic model are guaranteed to satisfy the Pfaffian constraints.*

Consider k Pfaffian constraints written in matrix form as $A^\top(\boldsymbol{\xi})\dot{\boldsymbol{\xi}} = \mathbf{0}$ (which can be a combination of holonomic and nonholonomic constraints). As was noted earlier these constraints imply that a generalized velocity $\dot{\boldsymbol{\xi}}$ is only admissible at a configuration $\boldsymbol{\xi}$ if it lies in the $n-k$ dimensional null space of the matrix $A^\top(\boldsymbol{\xi})$. A new matrix, $G(\boldsymbol{\xi}) \in \mathbb{R}^{n\times n-k}$ can therefore be defined such that the columns of $G(\boldsymbol{\xi})$ span the null space of $A^\top(\boldsymbol{\xi})$. In other words, for each column g_i of G it holds that $A^\top(\boldsymbol{\xi})g_i = 0$. To ensure that the generalized velocity $\dot{\boldsymbol{\xi}}$ satisfies the kinematics constraints it is therefore sufficient to require that $\dot{\boldsymbol{\xi}} = G(\boldsymbol{\xi})u$ where $\boldsymbol{u} \in \mathbb{R}^{n-k}$ can be *any* vector. To explicitly show why this is true, consider any vector u and write $\dot{\boldsymbol{\xi}} = G(\boldsymbol{\xi})\boldsymbol{u} = \sum_{i=1}^{n-k} g_i(\boldsymbol{\xi})u_i$. When this is substituted into the Pfaffian constraints the expression becomes

$$\begin{aligned} A^\top(\boldsymbol{\xi})\dot{\boldsymbol{\xi}} &= A^\top(\boldsymbol{\xi})\Big(\sum_{i=1}^{n-k} g_i(\boldsymbol{\xi})u_i\Big), \\ &= \sum_{i=1}^{n-k} A^\top(\boldsymbol{\xi})g_i(\boldsymbol{\xi})u_i, \\ &= 0, \end{aligned}$$

which shows that the kinematic constraints are satisfied.

Examples: Consider again the wheel example from Example 1.2.2 which has a single nonholonomic constraint

$$a_i(\boldsymbol{\xi})^\top\dot{\boldsymbol{\xi}} = \begin{bmatrix}\sin\theta & -\cos\theta & 0\end{bmatrix}\dot{\boldsymbol{\xi}} = 0,$$

where $\boldsymbol{\xi} = [x,\ y,\ \theta]^\top$. The null space of $a_i(\boldsymbol{\xi})^\top$ in this case is spanned by the vectors $[\cos\theta,\ \sin\theta,\ 0]$ and $[0,\ 0,\ 1]$ and therefore the kinematic model is given by

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}\cos\theta & 0\\ \sin\theta & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}u_1\\ u_2\end{bmatrix}. \tag{1.9}$$

Note that in many cases the control inputs u_1 and u_2 also have an intuitive physical meaning. In this problem u_1 is the speed at which the wheel is moving, and u_2 is the angular rate at which it rotates.

1.5 Kinematic Models of Wheeled Robots

Robots come in all shapes, sizes, and configurations and with varying forms of mobility. However, wheeled robots are perhaps the most widely used because

of their high mobility and simple design. For this reason several standard kinematic models for different wheeled robot configurations will now be given.

1.5.1 *Unicycle Model*

The unicycle model of a robot is the simplest kinematic model, and assumes that the robot can be approximated by a single wheel. In this case the kinematic constraints are exactly the same as the wheel rolling on a plane discussed previously in Example 1.2.2. A simplified diagram showing the generalized coordinates of this model is given in Figure 1.3, and the kinematic model is the same as (1.9):

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}, \tag{1.10}$$

where v is the forward speed of the unicycle and ω is the rate of rotation.

The advantage of the unicycle model lies in its simplicity and its ability to capture one of the most fundamental behaviors of wheeled robots. Such a model might be suited for higher level motion planning tasks, such as planning geometric paths to get a robot from point A to point B. Often times such a model might be complemented with models of higher fidelity (e.g. dynamics models) for performing lower level tasks such as control or for refining motion plans created by the unicycle model.

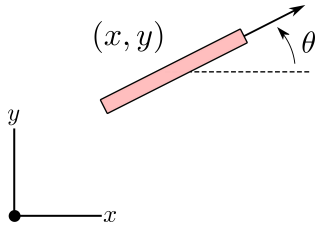


Figure 1.3: Generalized coordinates for a unicycle.

1.5.2 *Differential Drive Model*

The differential drive model is a slight variation on the unicycle model (see Section 1.5.1) that does not lump all of the wheels together. Instead, this model assumes two wheels are fixed on a rear shared axle, with a passive wheel that induces no kinematic constraints in the front. As shown in Figure 1.4 this model has the same generalized coordinates as the unicycle model ($\xi = [x, y, \theta]^\top$) but also includes some geometry of the robot by assuming the width of the rear axle is denoted by L.

Same as for the unicycle model, this model assumes the wheels roll without slipping. The derivation of the kinematic constraints is therefore similar to Example 1.2.2. In particular, the heading of each wheel is always given by $e_v = [\cos\theta, \sin\theta]^\top$, the lateral direction is given by $e_{v,\perp} = [\sin\theta, -\cos\theta]^\top$, and thus the two no-slip kinematic constraints can be expressed as

$$\dot{p}_l \cdot e_{v,\perp} = 0, \quad \dot{p}_r \cdot e_{v,\perp} = 0,$$

where $\dot{p}_l$ and $\dot{p}_r$ are the left and right wheel velocity vectors, respectively. The next step is to determine how to express $\dot{p}_l$ and $\dot{p}_r$ as functions of the generalized coordinates and generalized velocities. From the geometry of the robot it can be seen that

$$p_l = [x - \frac{L}{2}\sin\theta, y + \frac{L}{2}\cos\theta], \quad p_r = [x + \frac{L}{2}\sin\theta, y - \frac{L}{2}\cos\theta],$$

where p_l and p_r are the positions of the left and right wheels. By taking the derivative with respect to time the velocities are given by

$$\dot{p}_l = [\dot{x} - \dot{\theta}\frac{L}{2}\cos\theta,\ \dot{y} - \dot{\theta}\frac{L}{2}\sin\theta], \quad \dot{p}_r = [\dot{x} + \dot{\theta}\frac{L}{2}\cos\theta,\ \dot{y} + \dot{\theta}\frac{L}{2}\sin\theta].$$

It turns out that after some algebraic manipulation the no-slip kinematic constraints simply become:

$$\dot{p}_l \cdot e_{v,\perp} = \dot{p}_r \cdot e_{v,\perp} = \dot{x}\sin\theta - \dot{y}\cos\theta = 0,$$

which means having the no-slip kinematic constraint on both wheels is actually redundant! This also makes intuitive sense because the wheels are rigidly connected together, so if one wheel cannot move laterally then the other must not be able to. Additionally, it is noted that this nonholonomic constraint is identical to the one for the unicycle and so the kinematic model is also identical to (1.10). However the difference is that the *control inputs* can now be expressed in a more realistic form with respect to the actual geometry of the robot.

In particular, instead of the inputs being the forward speed v and body rotation rate ω as in (1.10), the inputs will be chosen to be the left and right wheel rotation rates, ω_l and ω_r. A relationship between these sets of inputs can be derived by exploiting the geometry of the robot and the no-slip wheel assumption. In particular, since the position $p = [x,\ y]$ can be written as $p = \frac{1}{2}(p_l + p_r)$ the velocity vector $\dot{p} = \frac{1}{2}(\dot{p}_l + \dot{p}_r)$. From the no-slip wheel assumption the speed can be expressed as $v = e_v \cdot \dot{p}$, which can be simplified to

$$\begin{aligned} v &= e_v \cdot \dot{p}, \\ &= e_v \cdot \frac{1}{2}(\dot{p}_l + \dot{p}_r), \\ &= \frac{1}{2}(e_v \cdot \dot{p}_l + e_v \cdot \dot{p}_r), \\ &= \frac{1}{2}(v_l + v_r), \\ &= \frac{r}{2}(\omega_l + \omega_r), \end{aligned}$$

where r is the radius of the wheel and v_l and v_r are the speeds of the left and right wheels.

Additionally, the no-slip condition on each individual wheel can be expressed as $v_l = e_v \cdot \dot{p}_l$ and $v_r = e_v \cdot \dot{p}_r$ which can be expanded to

$$\begin{aligned} \dot{x}\cos\theta + \dot{y}\sin\theta - \dot{\theta}\frac{L}{2} &= v_l, \\ \dot{x}\cos\theta + \dot{y}\sin\theta + \dot{\theta}\frac{L}{2} &= v_r. \end{aligned}$$

Noting that $\dot{x}\cos\theta + \dot{y}\sin\theta = v$ these expressions can be written as $\frac{L}{2}\dot{\theta} = v_r - v$ and $\frac{L}{2}\dot{\theta} = v - v_l$. Finally, combining these expressions yields

$$\begin{aligned} L\dot{\theta} &= v_r - v_l, \\ &= r(\omega_r - \omega_l). \end{aligned}$$

In summary, a one-to-one mapping between the inputs is given by

$$v = \frac{r}{2}(\omega_l + \omega_r), \quad \omega = \frac{r}{L}(\omega_r - \omega_l).$$

Finally, the differential drive kinematic model is given by

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \frac{r}{2}\cos\theta & \frac{r}{2}\cos\theta \\ \frac{r}{2}\sin\theta & \frac{r}{2}\sin\theta \\ \frac{r}{L} & -\frac{r}{L} \end{bmatrix} \begin{bmatrix} \omega_r \\ \omega_l \end{bmatrix}. \tag{1.11}$$

Overall, the complexity of this model over the unicycle model has not increased. However, by leveraging the geometry of the robot the inputs to this model may be more intuitive for motion planning and control tasks since the actual control mechanism is generally a motor attached to the wheel axles.

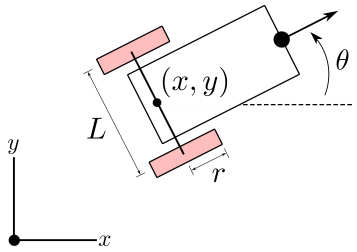


Figure 1.4: Generalized coordinates for a differential drive robot.

1.6 Dynamic Models

As was discussed in the introduction, mobile robot kinematic models are useful for describing fundamental physical behavior in a simple way, but they do not *completely* capture all real world influences on the robot's motion. The unicycle and differential drive models are examples of kinematic models that are approximations of the true system behavior. In particular they both make the no-slip wheel assumption, which directly lead to the kinematic constraints. Additionally, the choice of the inputs for the kinematic models ignores other important dynamics of the robot. In the unicycle model it is assumed the velocity is the input, but in practice directly commanding a desired velocity is not always straightforward since the amount of force required to change velocities varies with the mass of the robot ($F = ma$). In the differential drive model the inputs are the rotational rates of the wheels, but again in practice the amount of torque output required by the motor to change the rotation rate can vary depending on the robot's mass as well as other motor dynamics.

One common extension to kinematic models to incorporate *dynamics* is to simply add integrators to replace the input variables. The most common example of this is to replace a velocity input v with an acceleration input a and add the integrator $\dot{v} = a$. The force that generates the acceleration can then be considered as the input by using the dynamics equation $\dot{v} = \frac{1}{m}F$ where m is the mass of the robot. Similarly, a rotation rate input ω could be replaced by a rotational acceleration input. For example, the unicycle model (1.10) could be extended with integrator states to be

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{v} \\ \dot{\theta} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ a \\ \omega \\ \alpha \end{bmatrix}. \tag{1.12}$$

where a is linear acceleration in the forward direction and α is the angular acceleration (which of course could also be written with forces and torques as inputs).

In summary, factors to consider when deciding whether a certain kinematic model is sufficient. or if additional kinematics/dynamics are needed. include the robot's configuration/geometry and the task at hand (e.g. planning, control, etc.).

2
Open-Loop Motion Planning & Control

The previous chapter on motion planning and control introduced techniques for developing mathematical models to describe robot motion by analyzing its kinematics and dynamics. These models are typically expressed in the form of differential equations that are functions of a set of generalized coordinates/velocities and inputs to the system. The next step is to discover how these models can be leveraged for robot motion planning and control. In particular this chapter and the next will focus on robot control, where the goal is to determine what inputs to apply to the system to achieve desirable behavior. To address the robot control problem a *control law* must be developed, which is a set of rules or a mathematical function that determines what inputs should be applied to the system at any given time.

The ecosystem of techniques for robot control is vast, and control laws can generally be categorized in several ways. One of the most fundamental classifications for a control law is if it is *open-loop* or *closed-loop*. Open-loop control laws do not rely on observations to influence the choice of control input, while closed-loop control laws do. As a practical example, suppose you are standing in a room and wanted to walk to the other side and sit in a chair. For open-loop control you might look at where the chair is relative to your current position, think about how to walk there, and then *with your eyes closed* walk to the chair and sit. Alternatively, for closed-loop control you might keep *your eyes open* the whole time.

In practice, open-loop control laws suffer from robustness issues since they do not make corrections based on real-time observations. However, open-loop control is still an extremely important topic within the context of robotics. In particular, suppose you are interested not just in getting your robot from one point to another, but doing so in the *best* or *optimal* way. This problem, known as *trajectory optimization* or *optimal control*[1], can be solved to obtain an optimal trajectory for the robot along with the corresponding sequence of control inputs. In theory, applying this optimal control sequence as an open-loop control law would then make the robot follow the optimal trajectory.

[1] The terms trajectory optimization and optimal control will often be used interchangeably.

This chapter will discuss several common techniques related to optimal control and trajectory optimization, including a brief review on dynamic/kinematic

models, the formulation of the optimal control problem, approaches for solving optimal control problems, and some other topics useful in the context of robotics. The next chapter will then focus on the development of closed-loop control laws, including approaches that leverage the open-loop optimal control techniques discussed here.

Open-Loop Motion Planning & Control

This chapter and the next will focus on two of the most fundamental classifications for a control law, namely whether it is *open-loop* or *closed-loop*. In particular, this chapter will focus on open-loop control laws that arise from the study of optimal control and trajectory optimization problems[2,3]. In general, open-loop control laws depend only on time and initial condition of the system.

[2] D. E. Kirk. *Optimal Control Theory: An Introduction*. Dover Publications, 2004

[3] R. M. Murray. *Optimization-Based Control*. California Institute of Technology, 2009

Definition 2.0.1 (Open-loop control)**.** *If the control law is determined as a function of time for a specified initial state value, i.e.,*

$$\boldsymbol{u}(t) = f(\boldsymbol{x}(t_0), t), \tag{2.1}$$

then it is said to be in open-loop form.

2.1 *Kinematic and Dynamic Models*

Chapter 1 discussed techniques for deriving kinematic and dynamic models of a robot in the form of ordinary differential equations (ODE). Such models are extremely useful in the context of robot motion planning and control, and are essential in the context of optimal control. For the remainder of this chapter it will be assumed that such a model has already been identified and is expressed in the form

$$\dot{\boldsymbol{x}}(t) = a(\boldsymbol{x}(t), \boldsymbol{u}(t), t), \tag{2.2}$$

where $\boldsymbol{x} \in \mathbb{R}^n$ may be comprised of generalized coordinates ξ and velocities $\dot{\xi}$ and will be referred to as the robot's *state*, $\boldsymbol{u} \in \mathbb{R}^m$ is the control input, and the function $a : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^n$ defines the model. While the set of ODEs (2.2) may have been derived by considering kinematics, dynamics, or a combination of the two, this model will be generally referred to as the robot's *dynamics* model.

For clarity, note that (2.2) is a compact expression written in vector form for the system of n first-order differential equations

$$\begin{aligned}
\dot{x}_1(t) &= a_1(x_1(t), x_2(t), \ldots, x_n(t), u_1(t), u_2(t), \ldots, u_m(t), t) \\
\dot{x}_2(t) &= a_2(x_1(t), x_2(t), \ldots, x_n(t), u_1(t), u_2(t), \ldots, u_m(t), t) \\
&\vdots \\
\dot{x}_n(t) &= a_n(x_1(t), x_2(t), \ldots, x_n(t), u_1(t), u_2(t), \ldots, u_m(t), t),
\end{aligned}$$

where x_i is the i-th component of the vector $\boldsymbol{x}$ and u_j is the j-th component of the vector $\boldsymbol{u}$.

Solutions to the set of differential equations (2.2) are trajectories of the system. Given an initial condition $\boldsymbol{x}(t_0)$ and a control function $\boldsymbol{u}(t)$ defined for $t \geq t_0$, any technique for solving ODEs can be applied to compute the state trajectory $\boldsymbol{x}(t)$ for $t > t_0$. Common numerical integration approaches for solving the ODE system include the Runge-Kutta schemes, of which the most common are the forward or backward Euler schemes. The forward Euler scheme approximates $\dot{\boldsymbol{x}}(t) \approx \frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_i}{\Delta t}$ with $\Delta t = t_{i+1} - t_i$ and evaluates a at time t_i. This leads to the recursive update

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + a(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i)\Delta t, \quad i = 0, 1, \ldots \tag{2.3}$$

where $\boldsymbol{u}_i = \boldsymbol{u}(t_i)$ and $\boldsymbol{x}_i = \boldsymbol{x}(t_i)$.

2.2 *Optimal Control Problem*

Perhaps the most common open-loop control laws used for motion planning and control in robotics are synthesized by formulating and solving optimal control problems. These problems are designed to answer the question: from the current state of the robot, $\boldsymbol{x}(t_0)$, what future control inputs $\boldsymbol{u}(t)$ would make the robot follow an optimal future trajectory? In general, generating optimal open-loop control laws require three major components:

1. A model (2.2) that describes the robot's motion as a function of the input, developed by analyzing the robot's kinematics/dynamics.
2. A metric that defines the quality of a particular trajectory, known as a *cost function* or a *reward function*[4].
3. An algorithm for searching the space of possible control inputs to find one that corresponds to an optimal trajectory[5].

[4] The term *cost* is more commonly used in optimal control literature, while *reward* is used in the reinforcement learning literature.

[5] For example, convex optimization solvers

2.2.1 *Problem Formulation*

In this chapter the performance metric that defines the quality of a particular trajectory will be referred to as the *cost function*. The standard form for defining the cost function in optimal control problems is

$$J(\boldsymbol{x}(t), \boldsymbol{u}(t), t) = h(\boldsymbol{x}(t_f), t_f) + \int_{t_0}^{t_f} g(\boldsymbol{x}(t), \boldsymbol{u}(t), t)dt. \tag{2.4}$$

where $h(\boldsymbol{x}(t_f), t_f)$ is referred to as a *terminal cost* and where the integral can be viewed as a sum of *stage costs* induced along the path from times t_0 to t_f. In robotics, the function J might quantify objectives such as "get from point A to point B as quickly as possible" or "get from point A to point B while using as little effort as possible".

Constraints can also be considered in the optimal control problem. In the field of robotics it is common to consider constraints on the state and control that are expressed compactly as

$$\boldsymbol{x}(t) \in \mathcal{X}, \quad \boldsymbol{u}(t) \in \mathcal{U}, \tag{2.5}$$

where $\mathcal{X}$ is the set of all *admissible* states and $\mathcal{U}$ is the set of all *admissible* control inputs. A common way to define the sets $\mathcal{X}$ and $\mathcal{U}$ is by a set of inequalities on x and u, respectively. For example, let's assume the first element of $\boldsymbol{x}$ is constrained by $x_1 \geq 0$, then $\mathcal{X} = \{x \mid x_1 \geq 0\}$ such that any vector $\boldsymbol{x}$ with $x_1 \geq 0$ belongs to the set $\mathcal{X}$ (and is therefore *admissible*).

Constraints are commonly used in the context of robotics to account for actuator limits (e.g. how fast the wheels can turn, how much torque a motor can produce), or constraints on the trajectory itself (e.g. avoid collisions with surrounding objects).

The optimal control problem is then expressed as an optimization problem over the state trajectory $\boldsymbol{x}(t)$ and control inputs $\boldsymbol{u}(t)$ with the goal of minimizing the cost function (2.4) while also satisfying the constraints (2.5).

Definition 2.2.1 (Optimal Control Problem)**.** *An optimal control problem seeks an admissible control $\boldsymbol{u}(t)$ which causes the system (2.2) to follow an admissible trajectory $\boldsymbol{x}(t)$ that minimizes a performance metric $J(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$. This problem can be expressed as an optimization problem:*

$$\begin{aligned} \underset{\boldsymbol{u},\boldsymbol{x}}{\text{minimize}} \; & h(\boldsymbol{x}(t_f), t_f) + \int_{t_0}^{t_f} g(\boldsymbol{x}(t), \boldsymbol{u}(t), t)dt, \\ \text{s.t.} \; & \dot{\boldsymbol{x}}(t) = a(\boldsymbol{x}(t), \boldsymbol{u}(t), t), \\ & \boldsymbol{x}(t) \in \mathcal{X}, \quad \boldsymbol{u}(t) \in \mathcal{U}, \\ & \boldsymbol{x}(t_0) = \boldsymbol{x}_0, \end{aligned} \tag{2.6}$$

where t_0 is the initial time, t_f is either a fixed final time or an optimization variable, and x_0 is a known initial condition.

The solution to the optimal control problem (2.6) is an admissible and optimal trajectory defined over the interval $t \in [t_0, t_f]$, and is denoted by $\boldsymbol{u}^*(t)$ and $\boldsymbol{x}^*(t)$.

2.2.2 *Solving the Optimal Control Problem*

Once the optimal control problem (2.6) has been formulated, the next step is to find a solution. However, this can be challenging since (2.6) is an infinite-dimensional optimization problem (because the optimization is over an infinite-dimensional function and not a finite set of parameters). Unless an analytical solution to the problem can be found, this problem must be transformed into a finite dimensional problem so that it can be solved numerically on a computer. In general, algorithms for numerically solving optimal control problems can be classified as either *direct* or *indirect* methods.

Direct Methods: Direct methods follow a "first discretize, then optimize" approach. In the first step the problem (2.6) is converted into a finite-dimensional

problem by discretizing the functions $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$. For example this might be accomplished by defining the new optimization variables to be $\boldsymbol{x}(t_i)$ and $\boldsymbol{u}(t_i)$ for a finite number of time points t_i. This finite-dimensional optimization problem is generally referred to as a *nonlinear program* (NLP), which can be solved with existing numerical algorithms[6].

[6] Several solvers for solving general NLPs include IPOPT and SNOPT, and software packages for solving optimal control problems using the direct method include DIDO, PROPT, and GPOPS.

Indirect Methods: Indirect methods follow a "first optimize, then discretize" approach. These methods first derive the necessary conditions of optimality, which are expressed as a two-point boundary value problem. This two-point boundary value problem is essentially a set of ODEs with boundary conditions at two points[7] that must be numerically solved.

[7] This is in contrast to initial value problems, which have a single boundary condition and can easily be numerical integrated to find a solution.

Indirect methods are less commonly used in robotics because the derivation of the necessary conditions of optimality must be done on a case by case basis, and can become quite challenging. They become particularly difficult to use when constraints are imposed in the problem. In contrast, direct methods offer much more flexibility and have been quite successful in practice.

2.3 *Differential Flatness*

Solving optimal control problems to compute optimal trajectories and optimal control inputs for a system can sometimes be computationally challenging. In fact, sometimes it is more desirable to have a computationally efficient way of generating "good" trajectories, rather than a challenging way of generating "optimal" ones.

For a special class of models, which are referred to as *differentially flat*, computing "good" trajectories without having to formulate optimal control problems is quite easy. There are several models that are common in robotics that are differentially flat, including a simple car model and quadrotor models.

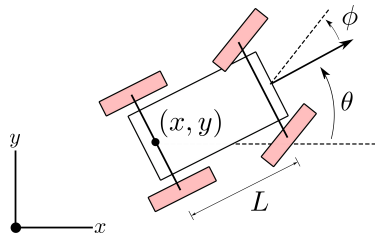


Figure 2.1: Simple model for an automobile. The state consists of the (x, y) position of the center of the rear axle and the heading angle θ. The control inputs are the steering angle ϕ and the forward velocity.

Example 2.3.1 (Simple Car Model)**.** Consider the car model corresponding to Figure 2.1:

$$\begin{aligned} \dot{x} &= v\cos\theta, \\ \dot{y} &= v\sin\theta, \\ \dot{\theta} &= \frac{v}{L}\tan\phi, \end{aligned} \tag{2.7}$$

where (x, y) is the position and θ is the orientation of the vehicle, v is the speed, ϕ is the steering angle, and L is the length of the wheelbase. The state $\boldsymbol{x}$ is therefore defined as $\boldsymbol{x} = [x,\, y,\, \theta]^\top$ and the control is defined as $\boldsymbol{u} = [v,\, \phi]^\top$.

Suppose the motion planning task is to find a control sequence $\boldsymbol{u}(t)$ that will take the car from an initial state $\boldsymbol{x}_0$ to a final desired state $\boldsymbol{x}_f$. One option would be to formulate an optimal control problem with constraints $\boldsymbol{x}(t_0) = x_0$ and $\boldsymbol{x}(t_f) = \boldsymbol{x}_f$. However, it turns out that for this model there is a simpler approach. In fact, for this model it is sufficient to specify a differentiable trajectory

for $x(t)$ and $y(t)$, and the remaining state variables and control inputs can be *analytically* determined!

To see why this is, consider a differentiable trajectory for $x(t)$ and $y(t)$ with derivatives $\dot{x}(t)$ and $\dot{y}(t)$. From the dynamics model (2.7) it can be seen that the first two equations can be leveraged to compute $\theta(t)$:

$$\theta = \tan^{-1}(\dot{y}/\dot{x}).$$

Furthermore, once $\theta(t)$ has been computed the speed is defined:

$$v = \dot{x}/\cos\theta, \quad \text{or} \quad v = \dot{y}/\sin\theta.$$

Finally, given $\theta(t)$ and $v(t)$ it is possible to directly solve for the steering angle:

$$\phi = \tan^{-1}(\frac{L\dot{\theta}}{v}).$$

This property, that from the specification of a few variables and their derivatives the remaining state and control values are defined, is known as *differential flatness*.

Definition 2.3.1 (Differential Flatness)**.** *A non-linear system*

$$\dot{\boldsymbol{x}}(t) = a(\boldsymbol{x}(t), \boldsymbol{u}(t)), \tag{2.8}$$

is differentially flat with flat output $\boldsymbol{z}$ if there exists a function α such that

$$\boldsymbol{z} = \alpha(\boldsymbol{x}, \boldsymbol{u}, \dot{\boldsymbol{u}}, \ldots, \boldsymbol{u}^{(p)}), \tag{2.9}$$

and such that the solutions to the system $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ can be written as functions of the flat output $\boldsymbol{z}$ and a finite number of its derivatives:

$$\begin{aligned} \boldsymbol{x} &= \beta(\boldsymbol{z}, \dot{\boldsymbol{z}}, \ldots, \boldsymbol{z}^{(q)}) \\ \boldsymbol{u} &= \gamma(\boldsymbol{z}, \dot{\boldsymbol{z}}, \ldots, \boldsymbol{z}^{(q)}). \end{aligned} \tag{2.10}$$

For a differentially flat system, all of the feasible trajectories for the system can be written as functions of a flat output $\boldsymbol{z}(t)$ and its time derivatives. Additionally, note that the number of flat outputs is always equal to the number of system inputs. In the context of motion planning and control this is extremely useful for trajectory design because the flat outputs can be specified and then *directly mapped* to the corresponding control inputs.

2.3.1 *Trajectory Design for Differentially Flat Systems*

As previously mentioned, trajectory design for differentially flat systems only requires specification of the trajectories of the flat outputs, which greatly simplifies motion planning and control.

Consider a nonlinear system model of the form (2.8) that is differentially flat with flat output $\boldsymbol{z}$ where the objective is to design a trajectory from $\boldsymbol{x}_0$ to $\boldsymbol{x}_f$ over

a horizon of T seconds. First, find the boundary conditions for the flat output $\boldsymbol{z}(0)$ and $\boldsymbol{z}(T)$ that satisfy the boundary conditions on $\boldsymbol{x}$ by noting that

$$\begin{aligned} \boldsymbol{x}_0 &= \beta(\boldsymbol{z}(0), \dot{\boldsymbol{z}}(0), \ldots, \boldsymbol{z}^{(q)}(0)), \\ \boldsymbol{x}_f &= \beta(\boldsymbol{z}(T), \dot{\boldsymbol{z}}(T), \ldots, \boldsymbol{z}^{(q)}(T)). \end{aligned} \tag{2.11}$$

Second, compute *any* smooth trajectory for the flat outputs $\boldsymbol{z}(t)$ that satisfy these boundary conditions. Third, use (2.10) to map the flat output trajectory $\boldsymbol{z}(t)$ to the state and control trajectories $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$.

Since the flat outputs can be specified as any smooth trajectory, a common choice is to parameterize them using N smooth basis functions:

$$z_j(t) = \sum_{i=1}^{N} \alpha_i^{[j]} \psi_i(t), \tag{2.12}$$

where z_j is the j-th element of $\boldsymbol{z}$, $\alpha_i^{[j]} \in \mathbb{R}$ are variables that parameterize the trajectory and $\psi_i(t)$ are the smooth basis functions. One potential choice is to use polynomial basis functions $\psi_1(t) = 1$, $\psi_2(t) = t$, $\psi_3(t) = t^2$, and so on. Another advantage of choosing this parameterization of $z_j(t)$ is that it is linear in the variables $\alpha_i^{[j]}$. This makes it easy to map specifications on $\boldsymbol{z}$ into values for α_i that define the trajectory. Consider differentiating (2.12) q times:

$$\begin{aligned} \dot{z}_j(t) &= \sum_{i=1}^{N} \alpha_i^{[j]} \dot{\psi}_i(t), \\ &\vdots \\ z_j^{(q)}(t) &= \sum_{i=1}^{N} \alpha_i^{[j]} \psi_i^{(q)}(t). \end{aligned} \tag{2.13}$$

Now, from the initial and final conditions $z_j(0)$, $\dot{z}_j(0)$, $\ldots, z_j^{(q)}(0)$ and $z_j(T)$, $\dot{z}_j(T)$, $\ldots, z_j^{(q)}(T)$ the coefficients $\alpha_i^{[j]}$ can be computed by solving the following linear system (assuming the matrix is full rank):

$$\begin{bmatrix} \psi_1(0) & \psi_2(0) & \ldots & \psi_N(0) \\ \dot{\psi}_1(0) & \dot{\psi}_2(0) & \ldots & \dot{\psi}_N(0) \\ \vdots & \vdots & & \vdots \\ \psi_1^{(q)}(0) & \psi_2^{(q)}(0) & \ldots & \psi_N^{(q)}(0) \\ \psi_1(T) & \psi_2(T) & \ldots & \psi_N(T) \\ \dot{\psi}_1(T) & \dot{\psi}_2(T) & \ldots & \dot{\psi}_N(T) \\ \vdots & \vdots & & \vdots \\ \psi_1^{(q)}(T) & \psi_2^{(q)}(T) & \ldots & \psi_N^{(q)}(T) \end{bmatrix} \begin{bmatrix} \alpha_1^{[j]} \\ \alpha_2^{[j]} \\ \vdots \\ \alpha_N^{[j]} \end{bmatrix} = \begin{bmatrix} z_j(0) \\ \dot{z}_j(0) \\ \vdots \\ z_j^{(q)}(0) \\ z_j(T) \\ \dot{z}_j(T) \\ \vdots \\ z_j^{(q)}(T) \end{bmatrix}. \tag{2.14}$$

Once the values for $\alpha_i^{[j]}$ are known, the entire trajectory $z_j(t)$ is therefore known!

Note that this approach is not strictly limited to specifying the initial and final conditions. It is also possible to specify other constraints on z_j and its

derivatives as long as they are *equality* constraints. This is accomplished by simply adding equations corresponding to the desired constraints to the linear system of equations (2.14). However, if too many constraints are added the linear system (2.14) may not have a solution (i.e. the system is over-determined). Assuming the constraints are not conflicting, this problem can typically be fixed by adding additional basis functions.

To summarize, for differentially flat nonlinear systems, the motion planning and control problem can be greatly simplified by planning in the flat output space. This is possible because of nonlinear functions that allow the flat output trajectory to be directly mapped to state and control trajectories that satisfy the system dynamics.

2.3.2 *Constraints and Time Scaling*

As previously shown, some constraints (e.g. boundary conditions) can be imposed on the trajectory by converting them into conditions on z and its derivatives, and then solving the linear system of equations (2.14). However, applying *bound* constraints can be slightly more challenging since they are expressed as inequality constraints rather than equality constraints. Nonetheless, bound constraints are common in robotics and therefore it is important to be able to consider them in the trajectory generation process. For example, the simple car robot from Example 2.3.1 could have an upper bound on its speed:

$$|v(t)| \leq v_{\max}.$$

One technique for handling these types of constraints is to use *time scaling*. The general approach to satisfy bound constraints by time scaling is:

1. Specify boundary conditions and solve the linear system of equations (2.14) to get a candidate trajectory $\boldsymbol{x}(t)$ with control inputs $\boldsymbol{u}(t)$.
2. If the candidate trajectory violates any bound constraints, generate a new trajectory by keeping the same geometric *path* but decreasing the rate at which it moves along the path.

2.3.3 *Geometric Path*

A geometric path is a sequence of states for the robot that is not associated with time. Given a candidate trajectory $\boldsymbol{x}(t)$, the geometric path can be defined by alternatively expressing the trajectory as $\boldsymbol{x}(t) = \boldsymbol{x}(s(t))$ where s is a new "path" parameter and $s(t)$ is defined with $s(0) = s_0$, $s(T) = s_f$, and $\dot{s}(t) > 0$. A common choice for the path parameter s is the arc length along the path. The geometric trajectory is then written as just $\boldsymbol{x}(s)$, such that the state is now a function of the position along the path and not time. Note that $\boldsymbol{x}(t) : [0, T] \to \mathbb{R}^n$ and $\boldsymbol{x}(s) : [s_0, s_f] \to \mathbb{R}^n$ are actually two different functions. In particular, the function $\boldsymbol{x}(t)$ can be derived from $\boldsymbol{x}(s)$ by the definition of the function $s(t) : [0, T] \to [s_0, s_f]$ and the composition $\boldsymbol{x}(s(t))$.

2.3.4 Time Scaling

For some systems, once the geometric path $\boldsymbol{x}(s)$ has been extracted from the candidate trajectory $\boldsymbol{x}(t)$, it is possible to arbitrarily redefine new trajectories with different time scales by simply redefining $s(t)$. In other words parts of the original candidate trajectory can be sped up or slowed down as desired.

To motivate why time scaling is important we can consider a simplified problem that does not involve a dynamics model. In particular, consider a scalar variable $x \in \mathbb{R}$ and a desired geometric path that connects x_0 and x_f that is parameterized as $x(s) = x_0 + s(x_f - x_0)$ for $s \in [0, 1]$ (note that $x(0) = x_0$ and $x(1) = x_f$). By choosing how s varies in time (i.e. the function $s(t)$) this geometric path can be transformed into many different *trajectories*, $x(t)$. As a simple choice, the function $s(t)$ can be parameterized as the cubic polynomial:

$$s(t) = \frac{3}{T^2}t^2 - \frac{2}{T^3}t^3.$$

This specific choice ensures that $s(0) = 0$, $s(T) = 1$, and $\dot{s}(0) = \dot{s}(T) = 0$ such that the trajectory will be defined over the time interval $t \in [0, T]$. Substituting this function into $x(s)$ then yields an expression for the trajectory $x(t)$:

$$x(t) = x_0 + \big(\frac{3}{T^2}t^2 - \frac{2}{T^3}t^3\big)(x_f - x_0).$$

One easy way to scale the trajectory in this case is to simply change T, with larger values of T meaning that it will take longer for x to traverse the geometric path from x_0 to x_f. In fact, the maximum velocity can also be computed as:

$$\dot{x}_{\max} = \frac{3}{2T}(x_f - x_0).$$

Therefore, not only does rescaling the trajectory by changing T make the path traversal time change, but it can also be used to decrease quantities such as the maximum velocity!

Time Scaling with Differential Models: Some additional considerations need to be made when time-scaling trajectories that must also satisfy differential models. First, note that the time derivative of the state can be rewritten by using the chain rule:

$$\dot{\boldsymbol{x}}(t) = \frac{d\boldsymbol{x}(t)}{dt} = \frac{d\boldsymbol{x}(s)}{ds}\frac{ds(t)}{dt}.$$

Now consider a candidate trajectory $\boldsymbol{x}(t)$ and an associated geometric path $\boldsymbol{x}(s)$ for some $s(t)$ that is defined over the interval $t \in [0, T]$ with $s(0) = s_0$ and $s(T) = s_f$. Since $\boldsymbol{x}(t)$ is a trajectory of the dynamics (2.8), the geometric path $\boldsymbol{x}(s)$ and time scaling law $s(t)$ satisfy

$$\frac{d\boldsymbol{x}(s)}{ds}\frac{ds(t)}{dt} = a(\boldsymbol{x}(s), \boldsymbol{u}(s)), \tag{2.15}$$

for every point $s \in [s_0, s_f]$.

To design a new time scaling law $\tilde{s}(t)$ over some potentially new time interval $t \in [0, \tilde{T}]$ where $\tilde{s}(0) = s_0$ and $\tilde{s}(\tilde{T}) = s_f$, it is important to note that the dynamics equations must still be satisfied[8]. In other words, for every $\tilde{s} \in [s_0, s_f]$:

$$\frac{d\boldsymbol{x}(\tilde{s})}{d\tilde{s}}\dot{\tilde{s}} = a(\boldsymbol{x}(\tilde{s}), \tilde{\boldsymbol{u}}(\tilde{s})). \tag{2.16}$$

[8] The geometric path is still defined on the interval $[s_0, s_f]$ so this interval must remain the same for any new time scaling law, but the time interval can change.

Since the geometric path is fixed, the terms $\frac{d\boldsymbol{x}(\tilde{s})}{d\tilde{s}}$ and $\boldsymbol{x}(\tilde{s})$ are fixed. Thus a new time scaling law $\tilde{s}(t)$ is only admissible if a new control $\tilde{\boldsymbol{u}}(\tilde{s})$ can also be found that guarantees that (2.16) holds. Luckily, for some specific systems this is easy with the appropriate choice of path parameter s.

Example 2.3.2 (Time Scaling for Simple Car Model)**.** Consider again the simple car model (2.7) from Example 2.3.1. Suppose a candidate trajectory $\boldsymbol{x}_c(t)$ with control $\boldsymbol{u}_c(t)$ has been defined by leveraging the differential flatness of the model (i.e. setting up and solving (2.14) and then mapping the flat outputs $\boldsymbol{z}_c(t)$ into the state and control). For this system a good choice for the path parameter is the arc-length, such that

$$s(t) = \int_0^\top v(t')dt', \quad \dot{s}(t) = v(t).$$

With this choice of path parameter the geometric path function $\boldsymbol{x}_c(s)$, $s_0 = 0$, and $s_f = L_{\text{path}}$ are all fixed (where L_{path} is the total length of the path). Rewriting the dynamics (2.16) based on the simple car model:

$$\begin{aligned}
\frac{dx_c(\tilde{s})}{d\tilde{s}}\dot{\tilde{s}} &= v(\tilde{s})\cos\theta_c(\tilde{s}),\\
\frac{dy_c(\tilde{s})}{d\tilde{s}}\dot{\tilde{s}} &= v(\tilde{s})\sin\theta_c(\tilde{s}),\\
\frac{d\theta_c(\tilde{s})}{d\tilde{s}}\dot{\tilde{s}} &= \frac{v(\tilde{s})}{L}\tan\phi(\tilde{s}).
\end{aligned}$$

Any choice of the time scaling function $\tilde{s}(t)$ must be able to satisfy these equations, and note that the trivial choice of $\tilde{s}(t) = s(t)$ will automatically satisfy these equations with the candidate control inputs $\boldsymbol{u}_c(t)$.

Since the choice of the path parameter yields $\dot{\tilde{s}} = v(\tilde{s})$, these equations can be further simplified:

$$\begin{aligned}
\frac{dx_c(\tilde{s})}{d\tilde{s}} &= \cos\theta_c(\tilde{s}),\\
\frac{dy_c(\tilde{s})}{d\tilde{s}} &= \sin\theta_c(\tilde{s}),\\
\frac{d\theta_c(\tilde{s})}{d\tilde{s}} &= \frac{1}{L}\tan\phi(\tilde{s}).
\end{aligned}$$

The first two equations are guaranteed to be satisfied for all $\tilde{s} \in [s_0, s_f]$ because the original candidate trajectory satisfies the dynamics. Additionally, the third equation is guaranteed to be satisfied by choosing $\phi(\tilde{s}) = \phi_c(\tilde{s})$ (i.e. using the same steering input as with the candidate trajectory).

This is interesting because it means that the equations are all satisfied *independently* of the choice of $\dot{\tilde{s}}$. Therefore, since $\dot{\tilde{s}} = v(\tilde{s})$ this means that the speed input can be chosen arbitrarily while maintaining the same geometric path! This is extremely useful because it means that bound constraints on the speed $|v(t)| \leq v_{\max}$ can be easily enforced.

Time Scaling with Kinematic Models: Time-scaling trajectories is much more straightforward when kinematic models are used. Consider the case where the model of the system is derived from k Pfaffian constraints $\boldsymbol{A}^\top(\boldsymbol{x})\dot{\boldsymbol{x}} = 0$. In this case the kinematic model can be written in the form:

$$\dot{\boldsymbol{x}} = G(\boldsymbol{x})\boldsymbol{u}, \tag{2.17}$$

where the columns of the matrix $G(\boldsymbol{x})$ span the null space of the matrix $A^\top(\boldsymbol{x})$. Now again consider a path parameter s that is used to reparameterize trajectories $\boldsymbol{x}(t)$ as $\boldsymbol{x}(s(t))$, and satisfies $s(0) = s_0$, $s(T) = s_f$, and $\dot{s}(t) > 0$[9]. Rewriting the time derivative of the state using the chain rule yields:

$$\frac{d\boldsymbol{x}(s)}{ds}\dot{s} = G(\boldsymbol{x})\boldsymbol{u}(t). \tag{2.18}$$

[9] The condition $\dot{s}(t) > 0$ is critical to ensure that the function $s(t)$ is invertible. In other words, to guarantee that there is a one-to-one mapping between t and s.

By making a substitution that $\boldsymbol{u}(t) = \boldsymbol{u}_g(s)\dot{s}$ the dynamics can be further written as:

$$\frac{d\boldsymbol{x}(s)}{ds} = G(\boldsymbol{x})\boldsymbol{u}_g(s). \tag{2.19}$$

The terms $\boldsymbol{u}_g(s)$ are referred to as *geometric controls*, since they are defined only with respect to the path parameter s. Critically, (2.19) says that once the geometric controls $\boldsymbol{u}_g(s)$ are defined, the entire geometric path $\boldsymbol{x}(s)$ is also defined! The choice of the timing law $s(t)$ can then be chosen in any manner and it will not change the geometric path, but will change the time trajectory $\boldsymbol{x}(t)$. In particular, once the geometric control $\boldsymbol{u}_g(s)$ and timing law are chosen, the actual controls are computed simply by the previous relationship $\boldsymbol{u}(t) = \boldsymbol{u}_g(s)\dot{s}$.

Based on this analysis, the procedure for *rescaling* a trajectory of a kinematic model can be made more concrete. First, consider a given trajectory $\boldsymbol{x}(t)$ with control $\boldsymbol{u}(t)$ defined over $t \in [0, T]$ that satisfies the kinematic model (2.17). For simplicity, consider the path parameter s to be arc-length of the trajectory such that $s(0) = 0$ and $s(T) = L_{\text{path}}$. The following steps can then be used to define a new control input $\tilde{\boldsymbol{u}}(t)$ that will make the kinematic model follow the same geometric path but with a different time scale:

1. Determine $s(t)$ based on the original trajectory $\boldsymbol{x}(t)$. In other words, figure out how far along the trajectory the system is at each time t. Then reparameterize the control $\boldsymbol{u}(t)$ as a function of s, $\boldsymbol{u}(s(t))$.
2. Compute the geometric controls $\boldsymbol{u}_g(s) = \boldsymbol{u}(s(t))/\dot{s}(t)$ for each point $s \in [s_0, s_f]$.
3. Define a new timing law $\tilde{s}(t)$ that satisfies $\tilde{s}(0) = 0$ and $\tilde{s}(\tilde{T}) = L_{\text{path}}$ with $\dot{\tilde{s}} > 0$ over the interval $[0, \tilde{T}]$.

4. Compute the new control $\tilde{\boldsymbol{u}}(t) = \boldsymbol{u}_g(\tilde{s}(t))\dot{\tilde{s}}(t)$ for all $t \in [0, \tilde{T}]$.

Example 2.3.3 (Time Scaling for Unicycle Model)**.** Consider the kinematic unicycle model:

$$\begin{aligned} \dot{x} &= v\cos\theta, \\ \dot{y} &= v\sin\theta, \\ \dot{\theta} &= \omega, \end{aligned} \tag{2.20}$$

where (x, y) is the position and θ is the orientation, v is the speed, and ω is the rotation rate. The state $\boldsymbol{x}$ is defined as $\boldsymbol{x} = [x,\ y,\ \theta]^\top$ and the control is defined as $\boldsymbol{u} = [v,\ \omega]^\top$.

To time-scale trajectories of this system, consider the use of arc-length as path parameter:

$$s(t) = \int_0^t v(\tau)d\tau, \quad \dot{s}(t) = v(t),$$

such that for a trajectory defined on the interval $t \in [0, T]$ with total length L_{path}, the path parameter is defined with $s(0) = 0$ and $s(T) = L_{\text{path}}$. With this choice, the geometric controls are given by:

$$\begin{aligned} v_g(s) &= \frac{v(s)}{\dot{s}(t)} = 1, \\ \omega_g(s) &= \frac{\omega(s)}{\dot{s}(t)} = \frac{\omega(s)}{v(s)}, \end{aligned}$$

where $v(s(t))$ has been substituted in for $\dot{s}(t)$. Therefore if a new timing law $\tilde{s}(t)$ is introduced this will automatically define a new velocity $\tilde{v}(\tilde{s})$ at each point $\tilde{s}$, which can then be used to solve for the new $\tilde{\omega}$ inputs by:

$$\tilde{\omega}(\tilde{s}) = \omega_g(\tilde{s})\dot{\tilde{s}}(t) = \frac{\omega(\tilde{s})}{v(\tilde{s})}\tilde{v}(\tilde{s}).$$

Alternatively, since it is easier to work with the velocity directly rather than $\tilde{s}(t)$, in this case it is possible to just specify $\tilde{v}(\tilde{s})$ for all $\tilde{s} \in [0, L_{\text{path}}]$ and then to compute $\tilde{\omega}(\tilde{s}) = \frac{\omega(\tilde{s})}{v(\tilde{s})}\tilde{v}(\tilde{s})$. Then, to determine the new controls as functions of time rather than $\tilde{s}$, it can be noted that

$$\tau(s) = \int_0^s \frac{ds'}{\tilde{v}(s')},$$

defines a function $\tau(s)$ that maps each point $s \in [0, L_{\text{path}}]$ to a new time.

2.4 Exercises

2.4.1 Trajectory Generation via Differential Flatness

Complete *Problem 1: Trajectory Generation via Differential Flatness* located in the online repository:

`https://github.com/PrinciplesofRobotAutonomy/AA274A_HW1`,

where you will use an extended unicycle model to practice generating dynamically feasible trajectories by levering the system's differential flatness property. You will also have the chance to use time scaling techniques to design trajectories that satisfy control constraints.

3
Closed-Loop Motion Planning & Control

The previous chapter introduced the concepts of *open-loop* and *closed-loop* control laws, and then dove into techniques for designing open-loop control laws for robots based on optimal control and differential flatness. These techniques are useful for determining control inputs that accomplish different objectives, such as "move from point A to point B in a minimal amount of time while satisfying some constraints". Additionally, computing open-loop control laws is often computationally less challenging that computing closed-loop control laws. However in practice open-loop control is not very robust since observations are not leveraged to update the control input. One solution to this robustness problem is to convert the open-loop control law into a closed-loop control law, typically referred to as *trajectory tracking controllers*. Another solution is to not use any open-loop techniques but rather to directly synthesize a closed-loop control law, for example by performing a *Lyapunov stability analysis*. This chapter will introduce techniques for synthesizing closed-loop controllers in both of these ways.

Closed-loop Motion Planning & Control

Recall from the previous chapter that open-loop control laws are defined as a function of time for a given initial condition. In contrast, closed-loop control laws are a function of the *current* state, and therefore are reactive.

Definition 3.0.1 (Closed-loop Control)**.** *If the control law is a function of the state and time, i.e.,*

$$\boldsymbol{u}(t) = \pi(\boldsymbol{x}(t), t) \tag{3.1}$$

then the control is said to be in closed-loop form.

Closed-loop controllers (also sometimes referred to as *feedback controllers* or *policies*), are much more robust than open-loop controllers. For example, suppose a controller needs to be designed to make a wheeled robot move from point to point. If the model used for open-loop controller design wasn't perfect, if the initial state was not perfectly known, or if external disturbances affected

the system (e.g. wheel slipping), then the robot would not exactly reach its desired destination. Alternatively, a closed-loop control law can continuously correct for these errors since it is always taking in new information.

3.1 *Trajectory Tracking*

One common approach to closed-loop control is to simply extend the open-loop control techniques from the previous chapter to include a feedback component. Such an approach consists of two steps:

1. Use open-loop control techniques to design a desired trajectory $\boldsymbol{x}_d(t)$ and corresponding control $\boldsymbol{u}_d(t)$.

2. Design a closed-loop control law that is designed to make sure the system stays close to the desired trajectory.

These controllers are referred to as *trajectory tracking* controllers, and their control law is defined as

$$\boldsymbol{u}(t) = \boldsymbol{u}_d(t) + \pi(\boldsymbol{x}(t) - \boldsymbol{x}_d(t), t). \tag{3.2}$$

This type of control law is also said to be a "feedforward plus feedback" controller. This is because the term $\boldsymbol{u}_d(t)$ is an open-loop "feedforward" term that attempts to generally make the system follow the desired trajectory, while the term $\pi(\boldsymbol{x}(t) - \boldsymbol{x}_d(t), t)$ is a "feedback" term that attempts to correct for any errors.

The previous chapter discussed techniques for solving open-loop control problems to define the desired trajectory, and additionally there are several approaches for designing the feedback component $\pi(\boldsymbol{x}(t) - \boldsymbol{x}_d(t), t)$:

- *Geometric* approaches generally leverage some sort of insight about the system and are therefore hard to discuss in general settings. They are also typically difficult to derive theoretical guarantees for.

- *Linearization* based approaches typically linearize nonlinear dynamics models about points along the desired trajectory. These linearized models are then used to design linear controllers (e.g. linear quadratic regulators). For some nonlinear systems, instead of linearizing about specific points it possible to *feedback linearize* the system. This essentially means that the non-linearities can be exactly "canceled" out such that the system can be considered linear. Linear control theory can then be applied to design a feedback control scheme.

- *Non-linear control* techniques also exist which do not rely on linearization. These approaches are also heavily system dependent, but one common tool for non-linear control is based on *Lyapunov* theory.

- *Optimization-based* feedback control laws can also be designed. These approaches often leverage optimal control theory, some of which was presented in the previous chapter. One common optimization-based approach for closed-loop control is known as *model predictive control* (MPC).

3.1.1 Trajectory Tracking for Differentially Flat Systems

For differentially flat systems linearization based approaches to designing trajectory tracking controllers are particularly useful[1]. In fact, every flat system can be linearized via dynamic feedback and a coordinate change to yield a dynamical system of the form

[1] J. Levine. *Analysis and Control of Nonlinear Systems: A Flatness-based Approach*. Springer, 2009

$$\boldsymbol{z}^{(q+1)} = \boldsymbol{w}, \tag{3.3}$$

where $\boldsymbol{z}^{(q+1)}$ is the $q+1$-th order derivative of the flat outputs z and q is the degree of the flat output space (i.e. the highest order of derivatives of the flat output that are needed to describe system dynamics), and $\boldsymbol{w}$ is a modified "virtual" input term.

The set of ODEs (3.3) are *linear*, which means that techniques from linear control theory can be applied to design a control law for $\boldsymbol{w}$. In particular, for trajectory tracking problems suppose a reference flat output trajectory $\boldsymbol{z}_d(t)$ has been defined which corresponds to the virtual input $\boldsymbol{w}_d(t)$. Let the error between the actual flat output and desired flat output be defined as $\boldsymbol{e}(t) = \boldsymbol{z}(t) - \boldsymbol{z}_d(t)$ and consider a closed-loop control law of the form

$$w_i(t) = w_{i,d}(t) - \sum_{j=0}^{q} k_{i,j} e_i^{(j)}(t), \tag{3.4}$$

where $(\cdot)_i$ denotes the i-th component of the vector, $e^{(j)} = \boldsymbol{z}^{(j)} - \boldsymbol{z}_d^{(j)}$ is the j-th order derivative of the error, and $k_{i,j}$ are called controller *gains*. The application of this control law to the system (3.3) will result in *closed-loop dynamics* of the form

$$\boldsymbol{z}^{(q+1)} = \boldsymbol{w}_d - \sum_{j=0}^{q} K_j \boldsymbol{e}^{(j)},$$

where K_j is a diagonal matrix with i-th diagonal element $k_{i,j}$. Since $\boldsymbol{z}_d^{(q+1)} = \boldsymbol{w}_d(t)$ this can be simplified to give the closed-loop error dynamics:

$$\boldsymbol{e}^{(q+1)} + \sum_{j=0}^{q} K_j \boldsymbol{e}^{(j)} = 0. \tag{3.5}$$

This set of linear ODEs describes the dynamics of the error, and many classical techniques from linear control theory can be used to choose the gains $k_{i,j}$ that will guarantee this system is *stable*. Having stable error dynamics means that the error will decay to zero, which in this case means the system will track the desired trajectory.

Example 3.1.1 (Extended Unicycle Trajectory Tracking)**.** Consider the dynamically extended unicycle model

$$\begin{aligned}\dot{x}(t) &= v\cos(\theta(t)),\\ \dot{y}(t) &= v\sin(\theta(t)),\\ \dot{v}(t) &= a(t),\\ \dot{\theta}(t) &= \omega(t),\end{aligned} \tag{3.6}$$

where the two inputs are the acceleration $a(t)$ and the rotation rate $\omega(t)$. This system is differentially flat with flat outputs x and y and order $q = 1$. It can therefore be expressed as:

$$\ddot{\boldsymbol{z}} = \begin{bmatrix}\ddot{x}\\ \ddot{y}\end{bmatrix} = \underbrace{\begin{bmatrix}\cos(\theta) & -v\sin(\theta)\\ \sin(\theta) & v\cos(\theta)\end{bmatrix}}_{:=J}\begin{bmatrix}a\\ \omega\end{bmatrix} := \begin{bmatrix}w_1\\ w_2\end{bmatrix},$$

and a trajectory tracking controller can be defined as

$$\begin{aligned}w_1 &= \ddot{x}_d - k_{px}(x - x_d) - k_{dx}(\dot{x} - \dot{x}_d),\\ w_2 &= \ddot{y}_d - k_{py}(y - y_d) - k_{dy}(\dot{y} - \dot{y}_d),\end{aligned}$$

where $(\cdot)_d$ represents a term associated with the desired trajectory. The control inputs $a(t)$ and $\omega(t)$ can then be computed by solving the linear system

$$J\begin{bmatrix}a\\ \omega\end{bmatrix} = \begin{bmatrix}w_1\\ w_2\end{bmatrix},$$

assuming that J is full rank.

3.2 *Closed-loop Control*

Trajectory tracking is just one example of closed-loop control, which assumes the existence of a desired trajectory for which to track. As previously discussed, one way of computing the desired trajectory is by solving an open-loop optimal control problem. However, in the context of optimal control, modifying an open-loop optimal control with feedback is not always the most desirable option. Instead, it may be preferred to just directly solve a closed-loop optimal control problem to obtain an optimal policy $\boldsymbol{u}^* = \pi(\boldsymbol{x}(t), t)$. Techniques for solving closed-loop optimal control problems typically are based on either the Hamilton-Jacobi-Bellman equation or dynamic programming.

Another common closed-loop control problem is to drive to or stabilize the system about a particular state (often called *regulation*). For systems with linear dynamics models the most controller for regulation problems is called the *linear quadratic regulator*. However, for nonlinear systems, stabilizing closed-loop controllers are commonly designed through *Lyapunov analysis*[2].

[2] J.-J. E. Slotine and W. Li. *Applied Nonlinear Control*. Pearson, 1991

3.2.1 Lyapunov-based Control

A Lyapunov stability analysis is a common tool for analyzing the stability of nonlinear systems. This analysis is based on the definition of a Lyapunov function, which can be thought of as a measure of the "energy" of the system. Similar to mechanical systems, if the energy does not increase in time then the system is considered stable[3].

[3] Note there are more technical definitions of stability, but for simplicity these will not be discussed here

The most challenging part of a Lyapunov stability analysis is finding a suitable Lyapunov function, and for many complex systems this may be extremely difficult. However, one of the advantages of the method is that it provides nice theoretical guarantees regarding the stability of the system, and is applicable to any system of interest.

Example 3.2.1 (Pose Stabilization)**.** [4] Consider a robot that is modeled by the unicycle robot model (differential drive robot model) represented graphically in Figure 3.1

[4] M. Aicardi et al. "Closed loop steering of unicycle like vehicles via Lyapunov techniques". In: *IEEE Robotics & Automation Magazine* 2.1 (1995), pp. 27–35

$$\begin{aligned} \dot{x}(t) &= v(t)\cos\theta(t), \\ \dot{y}(t) &= v(t)\sin\theta(t), \\ \dot{\theta}(t) &= \omega(t), \end{aligned} \tag{3.7}$$

where the control inputs are the robot speed v and the rotational rate ω. The objective is to design a closed-loop controller that will drive the robot the origin (i.e. $x = 0$, $y = 0$, $\theta = 0$).

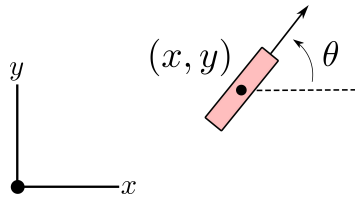


Figure 3.1: Pose stabilization of a unicycle robot in Cartesian coordinates.

To make the controller design easier the dynamics will be alternatively expressed in polar coordinates. This can be accomplished by defining

$$\begin{aligned} \rho &= \sqrt{x^2 + y^2}, \\ \alpha &= \text{atan2}(y, x) - \theta + \pi, \\ \delta &= \alpha + \theta, \end{aligned} \tag{3.8}$$

where ρ is the Euclidean distance to the origin, α is the heading angle with respect to the line from the robot to the origin, and δ is the angle between the x-axis and the line from the robot to the origin. These coordinates are graphically shown in Figure 3.2.

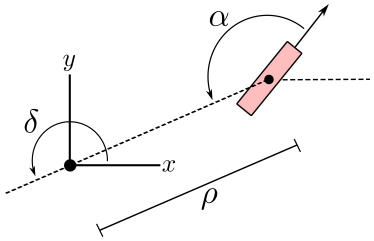


Figure 3.2: Pose stabilization of a unicycle robot using polar coordinates.

Using the newly defined polar coordinates, the dynamics equations (3.7) can be equivalently expressed as

$$\begin{aligned} \dot{\rho}(t) &= -v(t)\cos\alpha(t), \\ \dot{\alpha}(t) &= \frac{v(t)\sin\alpha(t)}{\rho(t)} - \omega(t), \\ \dot{\delta}(t) &= \frac{v(t)\sin\alpha(t)}{\rho(t)}. \end{aligned} \tag{3.9}$$

By expressing the dynamics in polar form, a Lyapunov stability analysis can now be easily performed. Consider the following *candidate* Lyapunov function:

$$V(\rho, \alpha, \theta) = \frac{1}{2}\rho^2 + \frac{1}{2}(\alpha^2 + k_3\delta^2), \tag{3.10}$$

and consider the following closed-loop control law:

$$\begin{aligned} v &= k_1 \rho \cos \alpha, \\ \omega &= k_2 \alpha + k_1 \frac{\sin \alpha \cos \alpha}{\alpha} (\alpha + k_3 \delta), \end{aligned} \tag{3.11}$$

where $k_1, k_2, k_3 > 0$.

The candidate Lyapunov function is quadratic and therefore is positive everywhere, $V \geq 0$, and is equal to zero only at the origin with $\rho = 0$, $\alpha = 0$, $\delta = 0$. Therefore, if it is possible to show that along *all* closed-loop system trajectories the Lyapunov function is *decreasing* ($\dot{V} < 0$), then it can be guaranteed that the system will converge to the origin! To show that the Lyapunov function decreases along trajectories of the system, begin by taking the derivative of V:

$$\dot{V} = \rho \dot{\rho} + \alpha \dot{\alpha} + k_3 \delta \dot{\delta}.$$

This quantity can now be shown to decrease along *all* closed-loop trajectories by substituting in the dynamics equations (3.8) with the closed-loop control law as defined by (3.11):

$$\begin{aligned} \dot{V} &= \rho \dot{\rho} + \alpha \dot{\alpha} + k_3 \delta \dot{\delta}, \\ &= -\rho v \cos \alpha + \alpha \big(\frac{v \sin \alpha}{\rho} - \omega \big) + \frac{k_3 \delta v \sin \alpha}{\rho}, \\ &= -k_1 \rho^2 \cos^2 \alpha - k_2 \alpha^2, \end{aligned}$$

where in the last line the control laws were substituted in for v and ω and algebraically simplified. Note that since k_1 and k_2 have been chosen to be strictly positive, this function must be strictly negative for all values of ρ and α! Therefore this Lyapunov stability analysis has theoretically proven that the system under the closed-loop control law (3.11) will converge to the origin.

3.3 Exercises

Both exercises for this chapter can be found in the online repository:

`https://github.com/PrinciplesofRobotAutonomy/AA274A_HW1`.

3.3.1 Pose Stabilization

Complete *Problem 2: Pose Stabilization*, where you will implement the Lyapunov-based pose controller for the unicycle robot described in Example 3.2.1.

3.3.2 Trajectory Tracking

Complete *Problem 3: Trajectory Tracking*, where you will implement the differential flatness-based trajectory tracking controller for the extended unicycle robot described in Example 3.1.1.

4
Optimal Control and Trajectory Optimization

Previously, the idea of using *optimal control*[1] techniques (also referred to as *trajectory optimization*) for robot motion planning and control was presented. In this chapter, the optimal control problem is revisited in more detail, including a brief discussion on the use of both *indirect* and *direct* methods.

[1] D. E. Kirk. *Optimal Control Theory: An Introduction*. Dover Publications, 2004

Optimal Control and Trajectory Optimization

Consider an optimal control problem (OCP) formulated as the following optimization problem:

$$\begin{aligned} \underset{\boldsymbol{u},\boldsymbol{x}}{\text{minimize}}\ & h(\boldsymbol{x}(t_f), t_f) + \int_{t_0}^{t_f} g(\boldsymbol{x}(t), \boldsymbol{u}(t), t)dt, \\ \text{s.t. } & \dot{\boldsymbol{x}}(t) = a(\boldsymbol{x}(t), \boldsymbol{u}(t), t), \\ & \boldsymbol{x}(t_0) = \boldsymbol{x}_0, \end{aligned} \tag{4.1}$$

where $\boldsymbol{x} \in \mathbb{R}^n$ is the robot state, $\boldsymbol{u} \in \mathbb{R}^m$ is the control input, x_0 is a known robot initial condition, $a(\boldsymbol{x}, \boldsymbol{u}, t)$ is a function describing the robot's dynamics, and the functions $h(\boldsymbol{x}(t_f), t_f)$ and $g(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$ define the cost function[2]. The goal is to solve the optimal control problem (4.1) in order to define an *optimal* open-loop control law of the form

$$\boldsymbol{u}^*(t) = f(\boldsymbol{x}(t_0), t).$$

[2] State constraints $\boldsymbol{x}(t) \in \mathcal{X}$ and control constraints $\boldsymbol{u}(t) \in \mathcal{U}$ are also often included in practice, but for simplicity are not included here.

Unfortunately, this optimization problem is particularly challenging to solve since it is *infinite-dimensional*[3]. Methods for solving (4.1) can be categorized as either *indirect* or *direct*. Both types of methods (almost always) require some form of discretization, such that the problem can be solved numerically. However, the way in which the problem is discretized is what makes each method unique.

[3] It is referred to as infinite-dimensional because it is an optimization over functions and not just a finite set of parameters.

1. *Indirect methods* follow a "first optimize, then discretize" approach. These methods first derive conditions for optimality of the original infinite-dimensional problem. A solution is then recovered by discretizing the optimality conditions.

2. *Direct methods* follow a "first discretize, then optimize" approach. These methods first discretize the original problem into a finite-dimensional problem (called a *nonlinear program*), which is then solved numerically to recover an optimal solution.

4.1 Indirect Methods

As previously mentioned, indirect methods solve the optimal control problem (4.1) by deriving *necessary optimality conditions* (NOC). A numerical procedure is then used to find solutions that satisfy these conditions of optimality, thereby "indirectly" solving the original OCP. As a brief example, for unconstrained finite-dimensional optimization problems the classic first-order necessary optimality condition[4] is that the gradient of the function must be zero (e.g. minimize $f(x) = x^2$ with $x \in \mathbb{R}$ has NOC $\frac{df}{dx} = 0$).

[4] It is important to note that these conditions are called necessary because they are "necessary", but they may not be "sufficient". In other words there may exist solutions that satisfy the NOCs but do not solve the original problem.

4.1.1 Constrained Finite-Dimensional Optimization

Before discussing techniques to derive necessary optimality conditions for the infinite-dimensional OCP (4.1), it is useful to briefly examine analogous conditions in finite-dimensional optimization[5]. Consider the equality-constrained finite-dimensional optimization problem:

[5] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge University Press, 2004

$$\begin{aligned} &\underset{\boldsymbol{x}}{\text{minimize}}\ f(\boldsymbol{x}), \\ &\quad \text{s.t. } h_i(\boldsymbol{x}) = 0, \quad i = 1, \dots, m \end{aligned} \tag{4.2}$$

with variable $\boldsymbol{x} \in \mathbb{R}^n$.
Necessary optimality conditions for (4.2) are derived by first forming a function called the Lagrangian $L(\boldsymbol{x}, \boldsymbol{\lambda})$, which augments the objective function with a weighted sum of the constraint functions:

$$L(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \sum_{i=1}^{m} \lambda_i h_i(\boldsymbol{x}), \tag{4.3}$$

where $\boldsymbol{\lambda} \in \mathbb{R}^m$ is a vector of *Lagrange multipliers*. The NOCs are then given as:

$$\begin{aligned} \nabla_{\boldsymbol{x}} L(\boldsymbol{x}^*, \boldsymbol{\lambda}^*) = 0, \\ \nabla_{\boldsymbol{\lambda}} L(\boldsymbol{x}^*, \boldsymbol{\lambda}^*) = 0, \end{aligned} \tag{4.4}$$

which are the gradients of the Lagrangian with respect to the variables $\boldsymbol{x}$ and the multipliers $\boldsymbol{\lambda}$. Note that the NOCs (4.4) are a set of $n + m$ *algebraic* equations with $n + m$ unknowns. In contrast, it will be seen next that the NOCs for infinite-dimensional problems are not algebraic, but rather differential.

4.1.2 Necessary Optimality Conditions

Analogously to the Lagrangian (4.3) in finite-dimensional optimization, the first step to defining the NOCs for the infinite-dimensional OCP (4.1) is to define a

function called the *Hamiltonian*:

$$H(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}(t), t) := g(\boldsymbol{x}(t), \boldsymbol{u}(t), t) + \boldsymbol{p}^\top(t) a(\boldsymbol{x}(t), \boldsymbol{u}(t), t), \tag{4.5}$$

where $\boldsymbol{p}(t) \in \mathbb{R}^n$ is a multiplier referred to as a *costate*. The NOCs are then given by a set of differential and algebraic equations:

$$\begin{aligned} \dot{\boldsymbol{x}}^*(t) &= \frac{\partial H}{\partial \boldsymbol{p}}(\boldsymbol{x}^*(t), \boldsymbol{u}^*(t), \boldsymbol{p}^*(t), t), \\ \dot{\boldsymbol{p}}^*(t) &= -\frac{\partial H}{\partial \boldsymbol{x}}(\boldsymbol{x}^*(t), \boldsymbol{u}^*(t), \boldsymbol{p}^*(t), t), \\ 0 &= \frac{\partial H}{\partial \boldsymbol{u}}(\boldsymbol{x}^*(t), \boldsymbol{u}^*(t), \boldsymbol{p}^*(t), t), \end{aligned} \tag{4.6}$$

which must be satisfied for all $t \in [t_0, t_f]$. These NOCs consist of $2n$ first order differential equations and m algebraic equations. Identifying unique solutions to the $2n$ differential equations requires $2n$ boundary conditions (actually $2n+1$ if the final time t_f is not fixed). The initial condition $\boldsymbol{x}^*(t_0) = \boldsymbol{x}_0$ specifies n of these conditions, and the remaining conditions are given by

$$\begin{aligned} &\Big(\frac{\partial h}{\partial \boldsymbol{x}}(\boldsymbol{x}^*(t_f), t_f) - \boldsymbol{p}^*(t_f))\Big)^\top \delta \boldsymbol{x}_f \\ &+ \big(H(\boldsymbol{x}^*(t_f), \boldsymbol{u}^*(t_f), \boldsymbol{p}^*(t_f), t_f) + \frac{\partial h}{\partial t}(\boldsymbol{x}^*(t_f), t_f)\big)\delta t_f = 0, \end{aligned} \tag{4.7}$$

where $\delta \boldsymbol{x}_f$ and δt_f are referred to as *variations*. If either the final time or final state is fixed in the optimal control problem the corresponding variation is forced to be zero, which changes the boundary conditions (4.7). The resulting boundary conditions for the four possible scenarios are now summarized:

Fixed Final Time and Fixed Final State: If both t_f and $\boldsymbol{x}(t_f)$ are fixed, both variations δt_f and $\delta \boldsymbol{x}_f$ are set to zero. In this case the boundary conditions (4.7) are trivially satisfied, and the remaining boundary conditions on the NOCs (4.6) are given by:

$$\begin{aligned} \boldsymbol{x}^*(t_0) &= \boldsymbol{x}_0, \\ \boldsymbol{x}^*(t_f) &= \boldsymbol{x}_f. \end{aligned}$$

Fixed Final Time and Free Final State: If only t_f is fixed, then only the variation $\delta t_f = 0$. In this case the conditions (4.7) simplify and the boundary conditions for the NOCs (4.6) are given by:

$$\begin{aligned} \boldsymbol{x}^*(t_0) &= \boldsymbol{x}_0, \\ \frac{\partial h}{\partial \boldsymbol{x}}(\boldsymbol{x}^*(t_f), t_f) - \boldsymbol{p}^*(t_f) &= 0. \end{aligned}$$

Free Final Time and Fixed Final State: If only $\boldsymbol{x}_f$ is fixed, then only the variation $\delta \boldsymbol{x}_f = 0$. In this case the conditions (4.7) simplify and the boundary conditions

for the NOCs (4.6) are given by:

$$\begin{aligned}
&\boldsymbol{x}^*(t_0) = \boldsymbol{x}_0, \\
&\boldsymbol{x}^*(t_f) = \boldsymbol{x}_f, \\
&H(\boldsymbol{x}^*(t_f), \boldsymbol{u}^*(t_f), \boldsymbol{p}^*(t_f), t_f) + \frac{\partial h}{\partial t}(\boldsymbol{x}^*(t_f), t_f) = 0.
\end{aligned}$$

Note that in this case since the final time is free an additional boundary condition is added, so there are now $2n + 1$ total conditions.

Free Final Time and Free Final State: If neither t_f or $\boldsymbol{x}(t_f)$ is fixed, then the boundary conditions for the NOCs (4.6) are given by:

$$\begin{aligned}
&\boldsymbol{x}^*(t_0) = \boldsymbol{x}_0, \\
&\frac{\partial h}{\partial \boldsymbol{x}}(\boldsymbol{x}^*(t_f), t_f) - \boldsymbol{p}^*(t_f) = 0, \\
&H(\boldsymbol{x}^*(t_f), \boldsymbol{u}^*(t_f), \boldsymbol{p}^*(t_f), t_f) + \frac{\partial h}{\partial t}(\boldsymbol{x}^*(t_f), t_f) = 0.
\end{aligned}$$

Again, since the final time is free an additional boundary condition is added such that there are $2n + 1$ total. Note that last two conditions are both extracted from (4.7) because the variations $\delta \boldsymbol{x}_f$ and δt_f are independent.

4.1.3 Two-Point Boundary Value Problems

Finding solutions that satisfy the necessary optimality conditions (4.6) for the optimal control problem is challenging. In particular, any solution must satisfy a set of $2n$ differential equations with boundary conditions specified at both t_0 and t_f. The problem of finding solutions to differential equations with boundary conditions specified at two points is called a *two-point boundary value problem*. Luckily, numerical procedures have been developed for solving these types of problems. For example the `scikits.bvp_solver` package in Python or the function `bvp4c` in Matlab implement schemes for solving these problems.

Most solvers for two-point boundary value problems typically assume the NOCs (4.6) and their boundary conditions are expressed in the standard form:

$$\dot{\boldsymbol{z}} = g(\boldsymbol{z}, t), \quad l(\boldsymbol{z}(t_0), \boldsymbol{z}(t_f)) = 0. \tag{4.8}$$

However, some types of problems may not directly fit into this standard form. For such instances, it is sometimes possible to convert a non-standard form problem into the standard form (4.8) [6].

In optimal control settings one common case where the two-point boundary value problem cannot directly be expressed in standard form is free final time problems, where t_f needs to be determined but does not have any associated dynamics. A useful trick in this case is to define a new variable $\tau = \frac{t}{t_f} \in [0, 1]$ to replace the time variable t (since before t_f wasn't known but now $\tau_f = 1$ is known). With this new variable the following changes can be made:

[6] U. Ascher and R. D. Russell. "Reformulation of boundary value problems into "standard" form". In: *SIAM Review* 23.2 (1981), pp. 238–254

1. Replace all derivatives with respect to t with derivatives with respect to τ, using $\frac{d(\cdot)}{d\tau} = t_f \frac{d(\cdot)}{dt}$ (chain rule).

2. Introduce a "dummy" state r that corresponds to t_f with dynamics $\dot{r} = 0$.

3. Replace t_f with r in all NOCs and in all boundary conditions.

The "dummy" state r can then be included in the vector $\boldsymbol{z}$ and the NOCs expressed in the standard form (4.8). In summary, this approach can be thought of as "tricking" the standard-form solver to think that the final time is 1 and that t_f is actually a state with dynamics (although the dynamics are $\dot{t}_f = 0$).

Example 4.1.1 (Free Final Time OCP)**.** Consider a double integrator system

$$\ddot{x} = u,$$

where $x \in \mathbb{R}$ is the state and $u \in \mathbb{R}$ is the control input where the control task is to find a trajectory that minimizes the cost function

$$J = \frac{1}{2}\alpha t_f^2 + \int_0^{t_f} \frac{1}{2}\beta u^2(t)\mathrm{d}t,$$

and satisfies the boundary conditions

$$x(0) = 10, \quad \dot{x}(0) = 0, \quad x(t_f) = 0, \quad \dot{x}(t_f) = 0.$$

This problem is a free final time problem with a fixed final state, and the cost is formulated to find a trajectory that minimizes a combination of the time to reach the final state and the amount of control effort required to get there. A trade-off between minimizing final time and minimizing control effort is made by adjusting the weighting parameters[7] α and β. From the cost function it is apparent that:

[7] What does intuition suggest the optimal behavior would be for $\alpha = 0$ or for $\beta = 0$?

$$h(\boldsymbol{x}(t_f), t_f) = \frac{1}{2}\alpha t_f^2, \quad g(\boldsymbol{x}(t), \boldsymbol{u}(t), t) = \frac{1}{2}\beta u^2(t),$$

and the dynamics equation can be equivalently expressed as a first-order system of ODEs by setting $x_1 = x$ and $x_2 = \dot{x}$:

$$\dot{x}_1 = x_2,$$
$$\dot{x}_2 = u,$$

such that $\boldsymbol{x} = [x_1,\, x_2]^\top$ and the boundary conditions become:

$$x_1(0) = 10, \quad x_2(0) = 0, \quad x_1(t_f) = 0, \quad x_2(t_f) = 0.$$

Now that the problem has been introduced, the first step is to derive the Hamiltonian:

$$H = \frac{1}{2}\beta u^2 + p_1 x_2 + p_2 u,$$

where p_1 and p_2 are the costates. Next, the NOCs (4.6) can be derived by taking the partial derivatives of H with respect to p, x, and u:

$$\begin{aligned}
\dot{x}_1^* &= x_2^*, \\
\dot{x}_2^* &= u^*, \\
\dot{p}_1^* &= 0, \\
\dot{p}_2^* &= -p_1^*, \\
0 &= \beta u^* + p_2^*.
\end{aligned}$$

The next step is then to determine appropriate boundary conditions for the NOCs. As mentioned before, this problem is a free final time and fixed final state problem. Therefore the boundary conditions are given by

$$\begin{aligned}
&x_1^*(0) = 10, \\
&x_2^*(0) = 0, \\
&x_1^*(t_f) = 0, \\
&x_2^*(t_f) = 0, \\
&\frac{1}{2}\beta u^*(t_f)^2 + p_1^*(t_f)x_2^*(t_f) + p_2^*(t_f)u^*(t_f) + \alpha t_f = 0.
\end{aligned}$$

Now, from the last NOC it can be seen that the optimal control u^* can be solved for in terms of the costate p_2^*:

$$u^* = -\frac{1}{\beta}p_2^*.$$

This expression can then be substituted into the second NOC and into the boundary conditions. At this point the resulting two-point boundary value problem can be expressed in the standard form (4.8) (by using the free final time trick previously discussed), and solved numerically. However, it also turns out that this problem is simple enough to solve analytically as well.

Analytical Solution: Integrating the differential equations for the costates p_1 and p_2 gives:

$$\begin{aligned}
p_1^* &= C_1, \\
p_2^* &= -C_1 t + C_2,
\end{aligned}$$

where C_1 and C_2 are constants. Therefore, the optimal control u^* can be expressed as $u^* = \frac{C_1}{\beta}t - \frac{C_2}{\beta}$ and the states x_1 and x_2 can be integrated to yield:

$$\begin{aligned}
x_2^* &= \frac{C_1}{2\beta}t^2 - \frac{C_2}{\beta}t + C_3, \\
x_1^* &= \frac{C_1}{6\beta}t^3 - \frac{C_2}{2\beta}t^2 + C_3 t + C_4,
\end{aligned}$$

where C_3 and C_4 are additional constants. There are now five unknown quantities, C_1, C_2, C_3, C_4, and t_f, which can be determined by leveraging the five

boundary conditions. In particular from the condition $x_1^*(0) = 10$ and $x_2^*(0) = 0$ it is easy to see that $C_3 = 0$ and $C_4 = 10$. The remaining boundary conditions can then be used to analytically solve for the remaining constants, and in particular:

$$t_f = (1800\frac{\beta}{\alpha})^{1/5}.$$

For a couple of interesting insights, it can be noted that as $\beta \to 0$ the cost function penalizes the final time more, and from the expression for t_f we can see that $t_f \to 0$. Additionally, as $\alpha \to 0$ the cost function penalizes control inputs, and correspondingly it can be seen in the expression for t_f that $t_f \to \infty$. Further, note that the optimal control takes the form

$$u^*(t) = \frac{C_1}{\beta}t - \frac{C_2}{\beta}.$$

Thus the control input is linear in time and its magnitude is inversely proportional to β.

4.2 *Direct Methods*

Unlike indirect methods, direct methods do not require a derivation of the necessary optimality conditions. Instead these methods directly discretize the original optimal control problem (4.1) to turn it into a finite-dimensional constrained optimization problem called a *nonlinear programming problem*.

While several approaches for discretizing the OCP exist, one simple approach is to just use a forward Euler time discretization. Recall that the forward Euler time discretization method (the simplest of the Runge-Kutta methods) can be used to numerically solve differential equations. In particular, with the choice of a time step h_i the differential equations $\dot{\boldsymbol{x}} = a(\boldsymbol{x}, \boldsymbol{u}, t)$ are discretized as:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + h_i a(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i), \tag{4.9}$$

where $\boldsymbol{x}_i = \boldsymbol{x}(t_i)$, $\boldsymbol{u}_i = \boldsymbol{u}(t_i)$, and $t_{i+1} - t_i = h_i$. With this recursive expression (4.9), an initial condition $\boldsymbol{x}(t_0)$, and a sequence of inputs $\boldsymbol{u}(t_i)$ for $i \geq 0$, the states $\boldsymbol{x}(t_i)$ can be computed easily. Suppose the optimal control problem (4.1) was defined over the time interval $[t_0, t_f]$. Applying a forward Euler time discretization essentially partitions this interval into a finite set of N times $\{t_0, t_1, \ldots, t_N\}$ where $t_N = t_f$ and the time step between each is $h_i = t_{i+1} - t_i$. Then the parameters of the optimization problem will simply become the state and controls at these times, $\boldsymbol{x}_i = \boldsymbol{x}(t_i)$ and $\boldsymbol{u}_i = \boldsymbol{u}(t_i)$ for $i = 0, \ldots, N$.

Rewriting the original OCP (4.1) as a function of the discrete set of parameters t_i, $\boldsymbol{x}_i$, and $\boldsymbol{u}_i$ will require modifications to both the constraints and to the cost function. First, the recursive formula (4.9) is used to replace the dynamics constraint $\dot{\boldsymbol{x}} = a(\boldsymbol{x}, \boldsymbol{u}, t)$ in the OCP[8]. Updating the cost function is going to require a numerical approximation of the integral, such as by using one of the

[8] The original dynamics model $\dot{\boldsymbol{x}} = a(\boldsymbol{x}, \boldsymbol{u}, t)$ is sometimes called the *continuous time* model and the recursive formula $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + ha(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i)$ is called the *discrete time* model.

Newton-Cotes formulas. The simplest of which would yield the approximation:

$$\int_{t_0}^{t_f} g(\boldsymbol{x}(t), \boldsymbol{u}(t), t)dt \approx \sum_{i=0}^{N-1} h_i g(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i).$$

The OCP (4.1) can now be expressed completely as the finite-dimensional nonlinear program (NLP):

$$\begin{aligned} \underset{\boldsymbol{u}_i, \boldsymbol{x}_i}{\text{minimize}} \; & h(\boldsymbol{x}_N, t_N) + \sum_{i=0}^{N-1} h_i g(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i), \\ \text{s.t. } & \boldsymbol{x}_{i+1} = \boldsymbol{x}_i + h_i a(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i), \quad i = 0, \ldots, N-1, \\ & \boldsymbol{x}_0 = \boldsymbol{x}(t_0). \end{aligned} \tag{4.10}$$

4.3 *Consistency of Time Discretization*

The finite-dimensional problem (4.10) is only an *approximation* of the original problem (4.1), so it is important to justify that this approximation method is *consistent* with the original problem. This is accomplished by taking a look at the necessary optimality conditions for the NLP (4.10) and comparing them to the necessary optimality conditions for the original OCP (4.1).

Recall that the necessary conditions of optimality for equality-constrained finite-dimensional optimization problems have previously been discussed in Section 4.1.1. In particular, the Lagrangian is first formulated, which for (4.10) takes the form:

$$L = h(\boldsymbol{x}_N, t_N) + \sum_{i=0}^{N-1} h_i g(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i) + \sum_{i=0}^{N-1} \boldsymbol{\lambda}_i^\top (\boldsymbol{x}_i + h_i a(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i) - \boldsymbol{x}_{i+1}).$$

Note that even though the initial condition constraint is included in (4.10) it can be ignored in the Lagrangian by simply assuming $\boldsymbol{x}_0$ is not actually a decision variable in the optimization problem (since it is fixed). The NOCs are then given by:

$$\begin{aligned} \nabla_{\boldsymbol{x}_i} L = h_i \frac{\partial g}{\partial \boldsymbol{x}}(\boldsymbol{x}_i, \boldsymbol{u}_i) + h_i \left(\frac{\partial a}{\partial \boldsymbol{x}}(\boldsymbol{x}_i, \boldsymbol{u}_i)\right)^\top \boldsymbol{\lambda}_i + (\boldsymbol{\lambda}_i - \boldsymbol{\lambda}_{i-1}) = 0, \quad & i = 1, \ldots, N-1 \\ \nabla_{\boldsymbol{x}_N} L = \frac{\partial h}{\partial \boldsymbol{x}}(\boldsymbol{x}_N) - \boldsymbol{\lambda}_{N-1} = 0, & \\ \nabla_{\boldsymbol{u}_i} L = h_i \frac{\partial g}{\partial \boldsymbol{u}}(\boldsymbol{x}_i, \boldsymbol{u}_i) + h_i \left(\frac{\partial a}{\partial \boldsymbol{u}}(\boldsymbol{x}_i, \boldsymbol{u}_i)\right)^\top \boldsymbol{\lambda}_i = 0, \quad & i = 0, \ldots, N-1 \\ \boldsymbol{x}_i + h_i a(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i) - \boldsymbol{x}_{i+1} = 0, \quad & i = 0, \ldots, N-1 \end{aligned} \tag{4.11}$$

Now, from the indirect method with equations (4.5), (4.6), and boundary conditions (4.7) with fixed final time and free final state, the NOCs for the infinite-

dimensional OCP can be written as:

$$\begin{aligned}
\frac{\partial g}{\partial \boldsymbol{x}}(\boldsymbol{x}(t),\boldsymbol{u}(t)) + \left(\frac{\partial a}{\partial \boldsymbol{x}}(\boldsymbol{x}(t),\boldsymbol{u}(t))\right)^\top \boldsymbol{p}(t) + \dot{\boldsymbol{p}}(t) &= 0, \quad t \in [t_0, t_f] \\
\frac{\partial h}{\partial \boldsymbol{x}}(\boldsymbol{x}(t_f)) - \boldsymbol{p}(t_f) &= 0, \\
\frac{\partial g}{\partial \boldsymbol{u}}(\boldsymbol{x}(t),\boldsymbol{u}(t)) + \left(\frac{\partial a}{\partial \boldsymbol{u}}(\boldsymbol{x}(t),\boldsymbol{u}(t))\right)^\top \boldsymbol{p}(t) &= 0, \quad t \in [t_0, t_f] \\
\dot{\boldsymbol{x}}(t) - a(\boldsymbol{x}(t),\boldsymbol{u}(t),t) &= 0, \quad t \in [t_0, t_f] \\
\boldsymbol{x}_0 - \boldsymbol{x}(t_0) &= 0.
\end{aligned} \tag{4.12}$$

The NOCs (4.11) for the discretized problem and the NOCs for the original OCP (4.12) are remarkably similar. In fact, the NOCs (4.11) can be seen as themselves simply the discretized versions of (4.12). To see this, simply perform a forward Euler discretization of the equations in (4.12) with:

$$\begin{aligned}
\dot{\boldsymbol{p}}(t) = \frac{\boldsymbol{\lambda}_i - \boldsymbol{\lambda}_{i-1}}{h_i}, \quad \boldsymbol{p}(t_i) = \boldsymbol{\lambda}_i, \quad i = 0, \dots, N-1, \\
\dot{\boldsymbol{x}}(t) = \frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_i}{h_i}, \quad \boldsymbol{x}(t_i) = \boldsymbol{x}_i, \quad \boldsymbol{u}(t_i) = \boldsymbol{u}_i, \quad i = 0, \dots, N-1.
\end{aligned}$$

Therefore, as the time step $h_i \to 0$ the NOCs for the discretized (direct method) problem converge to the NOCs derived directly for the original infinite-dimensional OCP (indirect method)!

4.4 Exercises

4.4.1 Optimal Control and Trajectory Optimization

Complete *Extra Problem: Optimal Control and Trajectory Optimization* located in the online repository:

`https://github.com/PrinciplesofRobotAutonomy/AA274A_HW1`,

where you will compute a dynamically feasible and *optimal* trajectory for a unicycle robot by using an indirect method to set up the necessary optimality conditions and solve them using a two-point boundary value solver.

5
Search-Based Motion Planning

Previous chapters addressed the problem of robotic motion planning and control by leveraging techniques from control theory and optimal control. In particular, these techniques were used to generate open and closed-loop control laws to accomplish specific tasks such as trajectory generation, trajectory tracking, and stabilization or regulation about a particular robot state. One common component among all of these algorithms was the use of a model of the robot's kinematics or dynamics, which mathematically defines how the robot transitions from state to state based on control inputs.

In this chapter yet another set of algorithms for motion planning/trajectory generation is discussed[1]. These algorithms are particularly well suited for higher-level motion planning tasks, such as motion planning in environments with obstacles. This is accomplished by focusing on formulating the motion planning problem for a robot with respect to the robot's *configuration space* rather than the *state space* that was used in previous chapters. While the robot's configuration is derivable from its state (and still characterizes all of the robot's degrees of freedom), the definition of the configuration space can be useful because it can be tailored to collision avoidance tasks[2]. Historically speaking, these approaches were developed alongside many of the techniques from previous chapters, and are still being researched today.

[1] S. M. LaValle. *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006

[2] In some cases the choice of configuration and state may end up being the same.

Search-Based Motion Planning

Recall the general definition of the motion planning problem:

Definition 5.0.1 (Motion planning problem)**.** *Compute a sequence of actions to go from an initial condition to a terminal condition while respecting constraints and possibly optimizing a cost function.*

Previous chapters approached this problem by formulating mathematical optimization problems that minimized a cost function subject to constraints on the motion (i.e. from dynamics/kinematics, control limits, or conditions on the robot's state), or leveraged differential flatness properties of the model. In these approaches, the robot's trajectory was parameterized by its state $\boldsymbol{x}$ and the

corresponding control inputs $\boldsymbol{u}$ which satisfied a set of differential equations

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

In this chapter, the motion planning problem will instead be addressed with respect to a *configuration space* (C-space). The configuration $\boldsymbol{q}$ of a robot is derivable from the full dynamics state $\boldsymbol{x}$ and captures all of the degrees of freedom of the robot (i.e. all rigid body transformations). In some cases the state and configuration of the robot may be the same, but in other cases the definition of the configuration can be tailored to simplify the motion planning problem. One important example of this is for *geometric path planning*, where paths in the configuration space can be planned without considering the robot kinematic/-dynamics model.

Example 5.0.1 (Motivating Example)**.** Consider the L-shaped robot from Figure 5.1 that lives in a 2D world with obstacles, and is trying to get from one point to another. Additionally, suppose this robot has a state $\boldsymbol{x} = [x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}]^\top$, and consider a configuration space defined by $\boldsymbol{q} = [x, y, \theta]^\top$ which fully captures the robot's degrees of freedom. Since the motion planning problem in this case involves obstacle avoidance, it might be easier to just plan a sequence of configurations $\boldsymbol{q}$ that are collision free (as is shown in the right-side graphic of Figure 5.1).

In this case, the use of the configuration space has simplified the motion planning problem by abstracting away the consideration of the robot's dynamics. Once the geometric path has been defined in configuration space, other techniques (such as those discussed in previous chapters) could be used to perform lower-level control functions for path tracking.

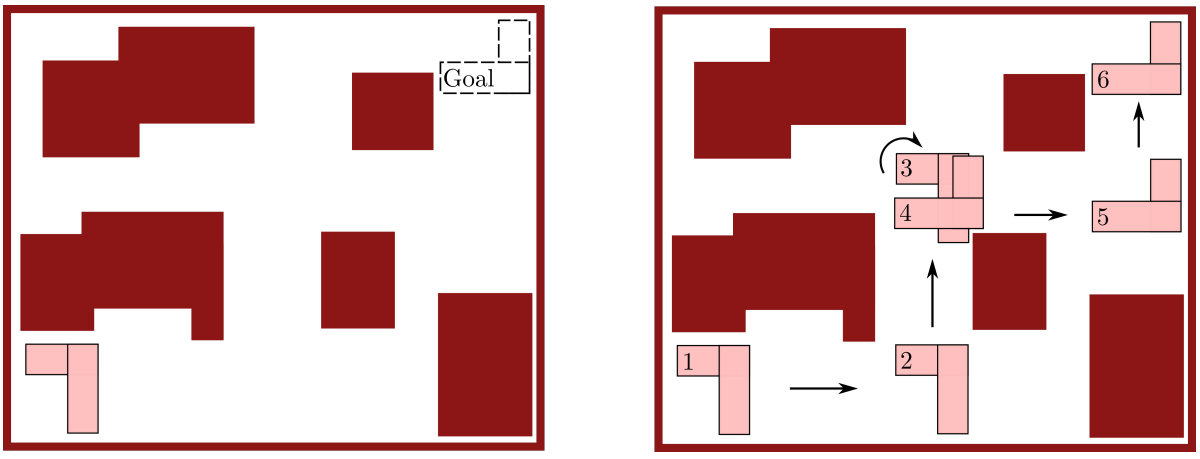


Figure 5.1: Motivating example: motion planning in a 2D workspace with obstacles.

Additionally, it is important to note that the C-space is a subset of $\mathbb{R}^3$, and in particular the C-space is $\mathbb{R}^2 \times \mathcal{S}^1$. This subspace is special because it includes the *manifold* $\mathcal{S}^1$, which characterizes the fact that the rotational degree of freedom θ satisfies $\theta = \theta \pm 2\pi k$ for all $k = 1, 2, \ldots$. This distinction is important

to make because it endows the planner with the ability to move from one angle to another in two different ways (i.e. the robot can turn left or turn right). For example, suppose the robot in Figure 5.1 has a current heading of θ_0 and wants move to have a heading θ_g subject to the constraint of avoiding a $\mathcal{C}$-space obstacle (see Figure 5.2). If the equivalence between the angles 0 and 2π is not established in the definition of the configuration space, the robot would not be able to traverse a collision-free path to the desired heading in the configuration space (see red trajectory). Instead, since the configuration space is defined with respect to $\mathcal{S}^1$, the robot is able to achieve the desired heading (see green trajectory).

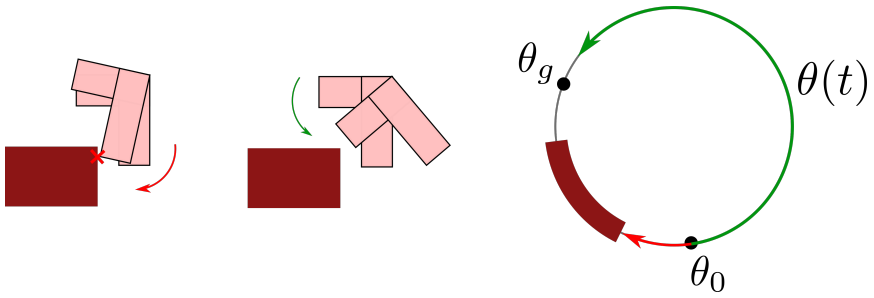


Figure 5.2: Example trajectory planning where the description of the configuration space using the manifold $\mathcal{S}^1$ is crucial to path planning. In particular, rotating clockwise leads to collision but rotating counter-clockwise is a feasible path.

In this chapter two types of motion planning algorithms that plan in the configuration space will be discussed. The first class consists of *grid-based methods*, and the second class consists of methods referred to as *combinatorial planners*.

5.1 Grid-based Motion Planners

Suppose the robot's configuration $\boldsymbol{q}$ is a d dimensional vector, then the C-space is a subset of $\mathbb{R}^d$. Critically, this is a continuous space and therefore there are an infinite number of potential configurations the robot could be in. To simplify this problem, grid-based motion planners use a grid to discretize the C-space into a finite number of allowable configurations. For example, in a simple C-space in two dimensions the grid might look like that shown in Figure 5.3. In grid-based planners, undesirable configurations are simply represented by identifying some cells of the grid to be forbidden (e.g. for obstacle avoidance). The dynamics/kinematics of the robot are also abstracted away and it is assumed that the robot has the ability to move freely between adjacent cells (configurations). After this discretization, the resulting motion planning problem is sometimes referred to as a *discrete planning* problem because only a finite number of options are available at each step, and only a finite number of configurations are possible. The planning problem then reduces to finding a way to traverse through the cells from the initial configuration to a desired final configuration.

Mathematically, problems of this type are commonly expressed using discrete *graphs*. A graph $G = (V, E)$, is simply defined by a set of vertices V and a set

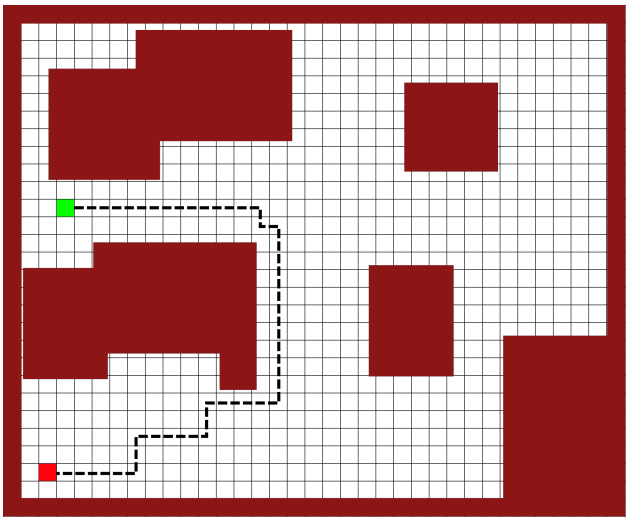


Figure 5.3: Discretizing the configuration space using a grid.

of edges E. In the context of grid-based motion planners, each vertex $v \in V$ represents a *free* cell of the grid, and each edge $(v, u) \in E$ corresponds to a connection between adjacent cells. With the graph representation, the planning problem is to find a way to traverse through the graph to reach the desired vertex. Algorithms for solving such problems are referred to as *graph search methods*.

The advantages of such approaches are that they are simple and easy to use, and for some problems can be very fast. The disadvantages are primarily the result of the discretization procedure. In some cases, if the resolution of the grid is not fine enough the search algorithm may not always be able to find a solution. Additionally, for a fixed resolution the size of the graph grows exponentially with respect to the dimension of the configuration space. Therefore this approach is generally limited to simple robots with a low-dimensional configuration space.

5.1.1 *Label Correcting Algorithms*

Since the graph is defined by a *finite* number of vertices (also referred to as *nodes*) and edges, it should be theoretically possible to solve a graph search problem in finite time. However in order to achieve this in practice, several simple "accounting" tricks need to be used to keep track of how the search has progressed and to avoid redundant exploration. Additionally, it is desirable to find a "best" path, and so a mechanism for keeping track of the current best path is required during the search.

A general set of algorithms known as *label correcting algorithms* employ such accounting techniques to guarantee good performance. In these algorithms, the notion of a "best" path is logged in terms of a cost-of-arrival.

Definition 5.1.1 (Cost-of-Arrival)**.** *The cost-of-arrival associated with a vertex q with*

respect to a starting vertex q_I is the cost associated with taking the best known route from q_I to q along edges of the graph, and is denoted $C(q)$.

Additionally, in a slight abuse of notation the cost from traversing an edge from vertex q to vertex q' is denoted as $C(q, q')$. To keep track of what nodes have already been visited and which still need further exploration, label correcting algorithms define a set of *frontier vertices* (sometimes also referred to as *alive*). This allows guarantees to be made that the search algorithm will avoid redundant exploration, and will terminate in finite time. It also guarantees that if a path from the initial vertex q_I to the goal vertex q_G exists, that it will be found.

In general, label correcting algorithms take the following steps to find the best path from an initial vertex q_I to a desired vertex q_G[3]:

1. Initialize the set of frontier vertices as $Q = \{q_I\}$ and set $C(q_I) = 0$. Initialize the cost-of-arrival of all other vertices q' as $C(q') = \infty$.
2. Remove a vertex from Q and explore each of its connected vertices q'. For each q', determine the candidate cost-of-arrival $\tilde{C}(q')$ associated with moving from q to q' as $\tilde{C}(q') = C(q) + C(q, q')$. If the candidate cost-of-arrival $\tilde{C}(q')$ is lower than the current cost-of-arrival $C(q')$ AND is lower than the current cost-of-arrival $C(q_G)$, then set $C(q') = \tilde{C}(q')$, define q as the parent of q', and add q' to the set Q if q' is not q_G.
3. Repeat step 2 until the set of frontier vertices Q is empty.

[3] In terms of robot motion planning this would be a search over paths through the discretized configuration space. Therefore the vertices of the graph are referred to as q to better connect this abstraction with their physical interpretation being a particular robot configuration $\boldsymbol{q}$.

The bulk of the work is done is Step 2. In particular, for the selected q from Q, these algorithms search its connected neighbors q' to see if moving from q to q' will lead to a lower overall cost than previously found paths to q'. This is why the algorithms are called "label correcting", since they "correct" the cost-of-arrival as better paths are found throughout the search process. Eventually, once the best path from q_I to q is found, q will never again be added to the set Q and therefore the algorithm is guaranteed to eventually terminate.

Theorem 5.1.2 (Label Correcting Algorithms)**.** *If a feasible path exists from q_I to q_G, then the label correcting algorithm will terminate in finite time with $C(q_G)$ equal to the optimal cost of traversal, $C^*(q_G)$.*

The primary way in which label correcting algorithms differ from each other is in how they select the next vertex q from the set of frontier nodes Q. In fact, the set Q is often referred to as a *priority queue* since the algorithm might assign priority values to the order in which vertices are selected. Different approaches for prioritizing include *depth-first search*, *breadth-first search*, and *best-first search*.

Depth-First Search Depth-first search in a directed graph expands each node up to the deepest level of the graph, until a chosen node has no more successors. Another way to think about this in terms of the set Q is "last in/first out", where whenever a new vertex q is selected from Q it chooses those vertices that were most recently added.

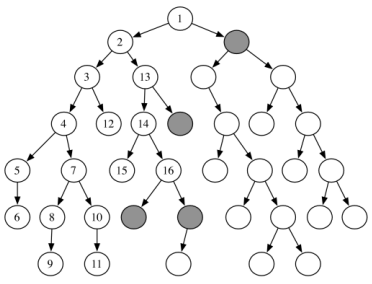


Figure 5.4: Depth-First Search

Breadth-First Search Breadth-first search begins with the start node and explores all of its neighboring nodes. Then for each of these nodes, it explores all their unexplored neighbors and so on. In terms of Q, this is like storing the frontier nodes as a queue with the first node added is the first node selected.

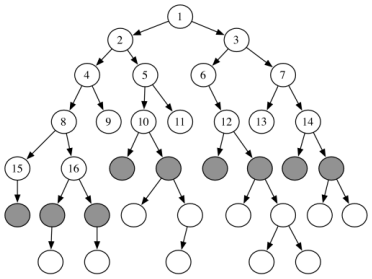


Figure 5.5: Breadth First Search

Best-First Search Also commonly known as *Dijkstra's algorithm*, this approach greedily selects vertices q from Q by looking at the current best cost-of-arrivals. Mathematically,

$$q = \arg\min_{q \in Q} C(q).$$

This approach is sometimes considered an "optimistic" approach since it is essentially making the assumption that the best current action will always correspond to the best overall plan. In practice this approach typically provides a more efficient search procedure relative to depth-first or breadth-first approaches because it can account for the cost of the path, however additional improvements can be made.

5.1.2 A* Algorithm

A* is a label correcting algorithm that is a modified version of Dijkstra's algorithm. In Dijkstra's algorithm the goal vertex q_G is not taken into account, potentially leading to wasted effort in cases where the greedy choice makes no progress towards the goal. This is quantified by a quantity called the *cost-to-go*.

Definition 5.1.3 (Cost-to-Go)**.** *The cost-to-go associated with a vertex q with respect to a goal vertex q_G is the cost associated with taking the best known route from q to q_G along edges of the graph.*

In practice, the cost-to-go is not usually known, and therefore *heuristics* are used to provide approximate cost-to-go values $h(q)$. In order for the heuristic to be useful, it must be a positive *underestimate* of the true cost-to-go. An example of a heuristic h is to simply use distance to the goal.

While Djikstra's algorithm only prioritizes a vertex q based on its cost-of-arrival $C(q)$, A* prioritizes based on cost-of-arrival $C(q)$ plus an approximate cost-to-go $h(q)$. This provides a better estimate of the total quality of a path than just using the cost-of-arrival alone. The A* algorithm is defined in Algorithm 1. Note that in the case that the heuristic is chosen to be $h(q) = 0$ for all q then A* is the same as Djikstra's algorithm.

5.2 Combinatorial Motion Planning

Combinatorial approaches to motion planning find paths through the continuous configuration space without resorting to discretizations like in grid-based planners. Recall that in grid-based planners, cells in the discretized configuration space that were undesirable were blocked out and simply not considered in the resulting path search. However, in the case of combinatorial planners

Algorithm 1: A* Algorithm

Data: q_I, q_G, G
Result: path
$C(q) = \infty,\ f(q) = \infty,\ \forall q$
$C(q_I) = 0, f(q_I) = h(q_I)$
$Q = \{q_I\}$
while Q *is not empty* **do**
  $q = \arg\min_{q' \in Q} f(q')$
  if $q = q_G$ **then**
    return *path*
  Q.remove(q)
  for $q' \in \{q' \mid (q, q') \in E\}$ **do**
    $\tilde{C}(q') = C(q) + C(q, q')$
    if $\tilde{C}(q') < C(q')$ **then**
      q'.parent = q
      $C(q') = \tilde{C}(q')$
      $f(q') = C(q') + h(q')$
      if $q' \notin Q$ **then**
        Q.add(q')
return *failure*

the structure of the free portion of the configuration space is considered in a different way.

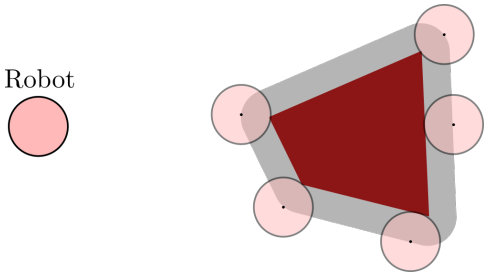


Figure 5.6: Free (white) and forbidden spaces (grey and red) of the configuration space for a simple circular robot in a 2D world. Note that the forbidden space accounts for the physical dimensions of the robot.

First, the subset of the configuration space C that is free (i.e. results in no collisions) is denoted as C_{free} and is called the *free space* (see Figures 5.6 and 5.7). Combinatorial motion planning approaches operate by computing *roadmaps* through the free space C_{free}. A roadmap is a graph G where each vertex represents a point in C_{free} and each edge represents a path through C_{free} that connects a pair of vertices. The set S is then defined for a particular roadmap graph G as the set of all points in C_{free} that are either vertices of G or lie on any edge of G. This graph structure is similar to that used in grid-based planners, with the important distinction that the vertices can potentially be *any* configuration $q \in C_{free}$ while in grid-based planners the vertices are defined ahead of time by

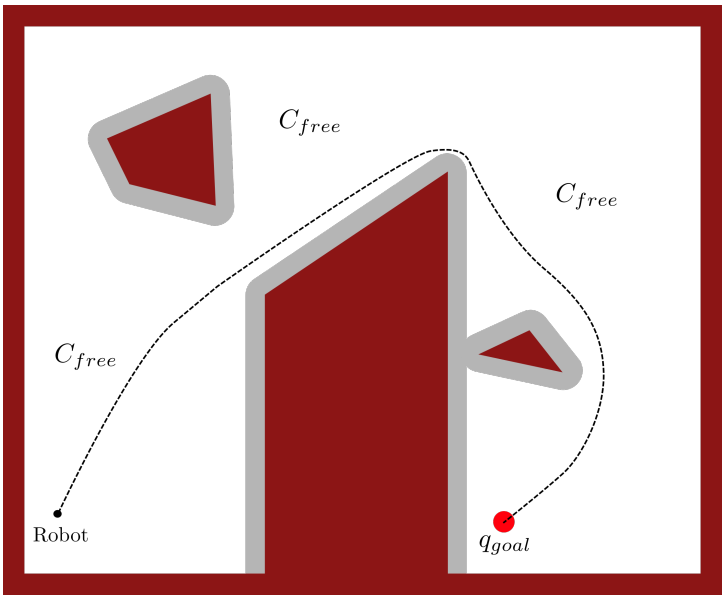


Figure 5.7: Once the free (white) and forbidden (grey and red) configurations have been identified, the physical dimensions of the robot can be ignored. This figure shows an example of a path planning problem in C-space with obstacles.

discretization. This distinction is very important because the flexibility of choosing the vertices does not result in any loss of information! Once the roadmap has been defined, a path can be defined by first connecting the initial configuration q_I and goal configuration q_G to the roadmap and then solving a discrete graph search over the roadmap graph G.

In general combinatorial planners are *complete* (i.e. the algorithm will either find a solution or will correctly report that no solution exists), and can even be optimal in some cases. However, often times in practice they are not computationally feasible to implement except in problems with low-dimensional configuration spaces and/or simple geometric representations of the environment. Additionally, it requires that the free space be completely defined in advance, which is not necessarily a realistic requirement.

5.2.1 *Cell Decomposition*

One common approach for deriving the roadmap is to use *cell decomposition* to decompose C_{free}. Cell decomposition refers to the process of partitioning C_{free} into a finite set of regions called cells, which should generally satisfy:

- Each cell should be easy to traverse and ideally convex.
- Decomposition should be easy to compute.
- Adjacencies between cells should be straightforward to determine, in order to build the roadmap.

Example 5.2.1 (2D Cell Decomposition)**.** Consider a two-dimensional configuration space as shown in Figure 5.8. This space is decomposed into cells that are

either lines or trapezoids by a process called vertical cell decomposition. Once the cells have been defined, the roadmap is generated by placing a vertex in each cell (e.g. at the centroid) as well as a vertex on each shared edge between cells.

If the forbidden space is polygonal, cell decomposition methods work pretty well and each cell can be made to be convex. In general, there exist several approaches for performing cell decomposition. However, cell decomposition in higher dimensions becomes increasingly challenging.

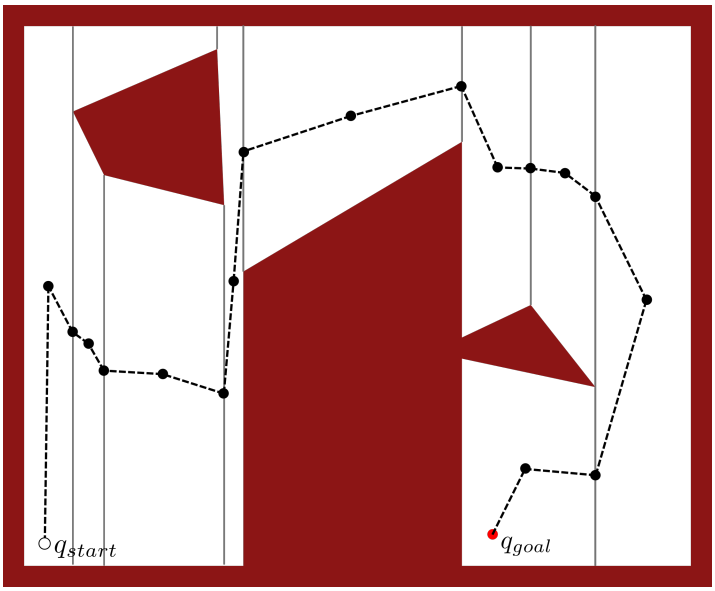


Figure 5.8: Example of 2D Cell Decomposition with C_{free} colored white. A roadmap is defined as the graph G with vertices shown as black dots and edges connecting them. To solve a planning problem with q_{start} and q_{goal} these points are first connected to the roadmap, and then the path is easily defined.

5.2.2 *Other Roadmaps*

Other ways to define roadmaps besides using cell decomposition exist. Two possible examples include a maximum clearance or minimum distance approach. Maximum clearance roadmaps simply try to always keep as far from obstacles as possible, for example by following the centerline of corridors. These roadmaps are also sometimes referred to as "generalized Voronoi diagrams". Minimum distance roadmaps are generally the exact opposite of maximum clearance roadmaps in that they tend to graze the corners of the forbidden space. In practice this is likely not desirable and therefore these approaches are less commonly used (without modification).

5.3 *Exercises*

5.3.1 *A* Motion Planning*

Complete *Problem 1: A* Motion Planning* located in the online repository:

`https://github.com/PrinciplesofRobotAutonomy/AA274A_HW2,`

where you will implement the A* grid-based motion planning algorithm for some simple 2D environments.

6
Sampling-Based Motion Planning

The previous chapter introduced motion planning problems that are formulated with respect to the robot's configuration space (C-space). In particular, two specific approaches for motion planning in C-space were discussed: grid-based methods and combinatorial planning methods. Grid-based methods discretize the continuous C-space into a grid, and then use graph search methods such as A* to compute paths through the grid. Combinatorial planners compute a *roadmap* that consists of a finite set of points in the C-space, but avoids the use of a rigid grid structure. Planning with the roadmap then consists of connecting the initial configuration and desired configuration to the roadmap, and then performing a graph search to find a path along the roadmap.

Generally speaking, grid-based methods suffer from the rigidity of the discretization that is performed. In contrast, combinatorial planners have much more flexibility because *any* configuration q can be a part of the roadmap. However, both types of planners require a complete characterization of the free configuration space (e.g. points in the configuration space that don't result in a collision with obstacles) in advance. In this chapter, a class of motion planning algorithms is presented which builds a roadmap that is similar to combinatorial planners, but without requiring a full characterization of the free configuration space. Instead, these algorithms build roadmaps one point at a time by sampling a point in the configuration space, and then querying an independent module to determine if the sample is admissible. This class of planners are referred to as *sampling-based methods*[1].

[1] S. M. LaValle. *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006

Sampling-Based Motion Planning

In contrast to the search-based motion planners discussed in the last chapter, sampling-based methods leverage an independent module that can be queried to determine if a configuration is admissible. In the context of robotics, an admissible configuration in motion planning problems is often one that is collision-free and therefore this module is often referred to as a collision detection module (or simply a *collision checker*). The collision detection module is used to probe and incrementally build a roadmap in the configuration space,

rather than attempting to completely characterize the free space in advance (as is done in combinatorial planners).

Sampling-based algorithms are a common choice for practical applications as they are conceptually simple, flexible, relatively easy to implement, and can be extended beyond the geometric case (i.e. they can consider differential constraints). The disadvantages of the approach are typically with respect to theoretical guarantees, for example these approaches cannot certify that a solution doesn't exist. In this chapter the focus will be on two popular sampling-based methods: probabilistic roadmaps (PRM) and the rapidly-exploring random trees (RRT) algorithm. Additional techniques such as the fast-marching tree algorithm (FMT*), kinodynamic planning, and deterministic sampling-based methods will also be briefly mentioned.

6.1 Probabilistic Roadmap (PRM)

It is easiest to start with the probabilistic roadmap algorithm because it is conceptually quite similar to combinatorial planners from the previous section. In particular, the PRM algorithm also generates a topological graph G called a *roadmap* where the vertices are configurations q in the free part of the configuration space C_{free}, and edges connect the vertices (and must also entirely lie in C_{free}). Once the roadmap is generated, a motion plan can be found for a given initial configuration q_I and goal configuration q_G by first connecting them to the roadmap, and then using a graph-search algorithm (e.g. A*) to find a path along the roadmap graph G. The difference between PRM and combinatorial planners lies in the method in which the roadmap is generated.

The key insight of the PRM algorithm is that a complete characterization of the free configuration space (which is computationally expensive) can be avoided by sampling configurations q at random and then using a collision checker to validate if $q \in C_{free}$. The general outline of the algorithm follows:

1. Randomly sample n configurations q_i from the configuration space.
2. Query a collision checker for each q_i to determine if $q_i \in C_{free}$, if $q_i \notin C_{free}$ then it is removed from the sample set.
3. Create a graph $G = (V, E)$ with vertices from the sampled configurations $q_i \in C_{free}$. Define a radius r and create edges for every pair of vertices q and q' where: (i) $\|q - q'\| \leq r$ and (ii) the straight line path between q and q' is also collision free.

An example of the roadmap resulting from applying this algorithm is shown in Figure 6.1. Note that using the *connectivity radius* r is a simple and efficient way of connecting the sampled vertices without having a burdensome number of edges. This is desirable because having too many edges is unnecessary, will make the graph-search more challenging, and will require more collision checks to be made[2]. On the flip side, making the radius r too small could mean not

[2] Edge validation is usually performed by densely sampling the edge and checking for collisions at each.

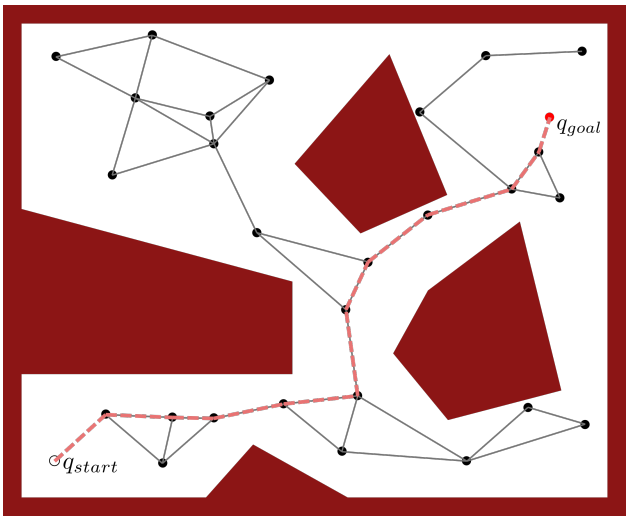


Figure 6.1: Example solution found via the PRM algorithm. The black dots represent the randomly sampled vertices of the graph, and the grey lines represent the edges created between vertices within a predefined radius r of each other. The initial configuration q_{start} and goal configuration q_{goal}, are connected through this roadmap along the pink line, which is found by a graph-search algorithm.

enough connections are made.

The downside of PRM is that finding good solutions may require a large number of samples n to sufficiently cover the configuration space. Similar to why having too many edges is not good, having too many samples will require a lot of queries of the collision checker, which may be costly. However, there are some scenarios where building a roadmap that completely covers the space C_{free} is beneficial, namely in *multi-query* planning problems. In multi-query problems, it is assumed that the motion planner will be called many times for different initial q_I and goal q_G configurations. In this case the PRM graph can be built once to cover C_{free}, and then it can be reused as many times as needed. In other words, the costly sampling and collision checking only needs to be done once at the start, so it may be worth the "investment". Note however that this only works if the environment stays the same in between each query of the motion planner. If the environment changes, the entire PRM roadmap would have to be rebuilt from scratch!

6.2 *Rapidly-exploring Random Trees (RRT)*

In multi-query problems where the environment does not change in between each query, the probabilistic roadmap (PRM) algorithm offers the advantage of front-loading some work to provide efficient queries later. However, many problems in robotics are alternatively classified as *single-query* problems, where it is assumed that only a single query will be made to the motion planner. A common single-query planning scenario arises from changing environments, such as if there is a moving obstacle. In this case building up a roadmap over the entire free configuration space C_{free} may result in wasted effort. The RRT algorithm solves this problem by incrementally sampling and building the graph, starting at the initial configuration q_I, until the goal configuration q_G is reached.

Additionally, the graph is built as a *tree*, which is a special type of graph that has only one path between any two vertices in the graph.

In general, the RRT algorithm begins by initializing a tree[3] $T = (V, E)$ with a vertex at the initial configuration (i.e. $V = \{q_I\}$). At each iteration the RRT algorithm then performs the following steps:

1. Randomly sample a configuration $q \in C$.
2. Find the vertex $q_{near} \in V$ that is closest to the sampled configuration q.
3. Compute a new configuration q_{new} that lies on the line connecting q_{near} and q such that the entire line from q_{near} to q_{new} is contained in the free configuration space C_{free}.
4. Add a vertex q_{new} and an edge (q_{near}, q_{new}) to the tree T.

Thus after each iteration only a single point is sampled and potentially added to the tree. Additionally, every so often the sampled point q can be set to be the goal configuration q_G. Then, if the nearest point q_{near} can be connected to q_G by a collision-free line the search can be terminated. Intuitively, this approach works because of a phenomenon referred to as the *Voronoi bias*, which essentially describes the fact that there is more "empty space" near the nodes on the frontier of the tree. Therefore, a randomly sampled point is more likely to be drawn in this "empty space", causing the frontier to be extended (and therefore driving exploration).

Note that variations on this standard algorithm exist, in particular there exist different ways of connecting a sampled point to the existing tree. One popular variant that modifies the way a sampled point is connected to the tree is known as RRT* (pronounced RRT star). This modified RRT algorithm introduces a notion of optimality into the planner and will in fact return an optimal solution as the number of samples approaches infinity. Another variant of RRT is called RRT-Connect, which simultaneously builds a tree from both the initial configuration q_I and the goal configuration q_G and tries to connect them.

6.3 Theoretical Results for PRM and RRT

One of the main challenges of sampling-based motion planning is that it is unclear how many samples are needed to find a solution. However, some theoretical guarantees can be provided regarding their asymptotic behavior (i.e. behavior as number of samples $n \to \infty$). In particular, both PRM[4] and RRT are guaranteed[5] to eventually find a solution if it exists [6,7]. Regarding solution *quality*, it has been shown that PRM (with the appropriate choice of the radius r) can find optimal paths as the number of samples $n \to \infty$. However, RRT can be arbitrarily bad with non-negligible probability [8].

[3] The tree is a graph, however since it has special structure it is denoted as T rather than G.

[4] With a constant connectivity radius r.

[5] These guarantees also require an assumption that the configuration space is bounded, for example if C is the d-dimensional unit hypercube with $2 \leq d \leq \infty$.

[6] S. M. LaValle. *Rapidly-Exploring Random Trees: A New Tool for Path Planning*. 1998

[7] L. E. Kavraki et al. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580

[8] S. Karaman and E. Frazzoli. "Sampling-based Algorithms for Optimal Motion Planning". In: *Int. Journal of Robotics Research* 30.7 (2011), pp. 846–894

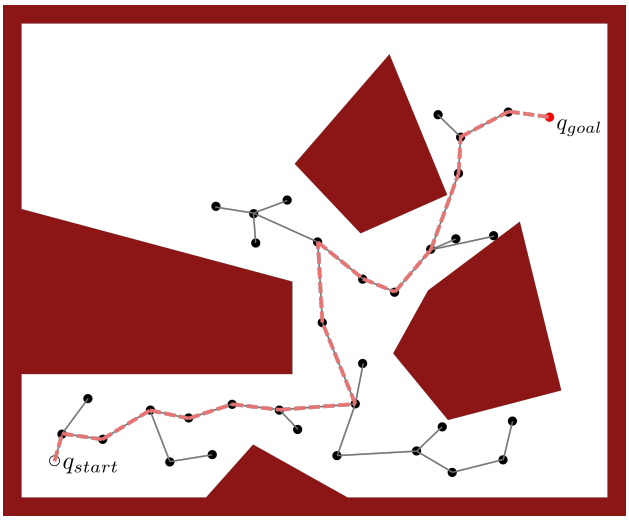


Figure 6.2: Example exploration tree by the RRT algorithm. The black dots represent points sampled at each iteration of the algorithm, which are connected to the nearest vertex that is currently part of the tree.

6.4 *Fast Marching Tree Algorithm (FMT*)*

As previously mentioned, PRM is an asymptotically optimal algorithm which means that with enough samples it will find good paths. However, in practice PRM with a large number of samples also requires a lot of collision checks and is therefore costly. On the other hand, RRT is fast but in general will not find good paths. FMT* is a an advanced sampling-based motion planning algorithm that maintains the advantages of both of these algorithms (i.e. fast and asymptotically optimal) [9].

FMT* builds a tree structured graph in the same way RRT does (which maintains the efficiency of RRT), but makes connections in a way that allows for asymptotic optimality. In particular, the technique used for making new connections is referred to as *dynamic programming*. Dynamic programming can be used to find the best paths with respect to a *cost-of-arrival*, denoted $c(q)$, which represents the cost to move from the initial configuration q_I to the configuration q. An example of a common metric is simply the Euclidean distance, which would result in a "shortest" path. In the context of motion planning, dynamic programming leverages Bellman's *principle of optimality*, which states that the optimal paths satisfy:

$$c(v) = \min_{u:\|u-v\|<r_n} \text{Cost}(u,v) + c(u), \tag{6.1}$$

where u are nodes within radius r_n of node v, $\text{Cost}(u,v)$ is the cost of an edge between u and v, and $c(u)$ is the cost-to-arrive at u. In words, this relationship says that the cost-of-arrival at any configuration v on the optimal path is defined by searching over all local neighboring configurations to find which would result in the best path. FMT* uses this principle repeatedly every time it needs to connect a new sample to the tree. However, in practice using the condition (6.1) is complicated by the fact that the resulting edge may result in a collision.

[9] L. Janson et al. "Fast Marching Tree: A Fast Marching Sampling-Based Method for Optimal Motion Planning in Many Dimensions". In: *Int. Journal of Robotics Research* 34.7 (2015), pp. 883–921

FMT* handles this by ignoring obstacles when using the condition (6.1) to connect a new sample to the tree, and then if a collision occurs from the resulting connection it is simply skipped and the algorithm moves on to a new sample. Therefore this application of dynamic programming is referred to as *lazy* because it only checks for collisions after the fact. It turns out that this substantially reduces the total amount of collision checks required, and only leads to sub-optimality in rare cases.

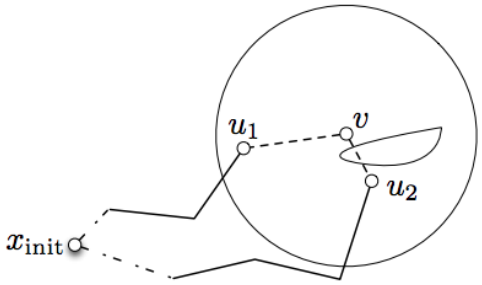


Figure 6.3: Example of a step in FMT*. Suppose the sample v has been selected to be the next point to be added to the tree. The candidate costs $\text{Cost}(u_1, v) + c(u_1)$ and $\text{Cost}(u_2, v) + c(u_2)$ are evaluated to see which connection would minimize c_v. Suppose u_2 was selected by this criteria (i.e. u_2 satisfies (6.1)), then the collision checker would see that the edge (u_2, v) results in a collision and the sample v would be skipped (but could be added later).

6.5 *Kinodynamic Planning*

The geometric motion planning algorithms previously considered assume that the robot does not have any constraints on its motion and only a collision-free solution is required. This makes the planning task easier because two configurations q and q' can be simply connected by the planner with a *straight line*. However, robots do typically have kinematic/dynamical constraints on their motion, and for some motion planning problems it is desirable or even necessary to take those constraints into account. The problem of planning a path through the free configuration space C_{free} that satisfies a given set of differential constraints is referred to as *kinodynamic motion planning*[10].

Similar to previous chapter, it is assumed that the robot operates in a state space $X \subseteq R^n$ and can apply controls $\boldsymbol{u} \in U \subseteq \mathbb{R}^m$, and that the motion constraints are given by the differential model (i.e. from kinematic or dynamics constraints):

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u}), \tag{6.2}$$

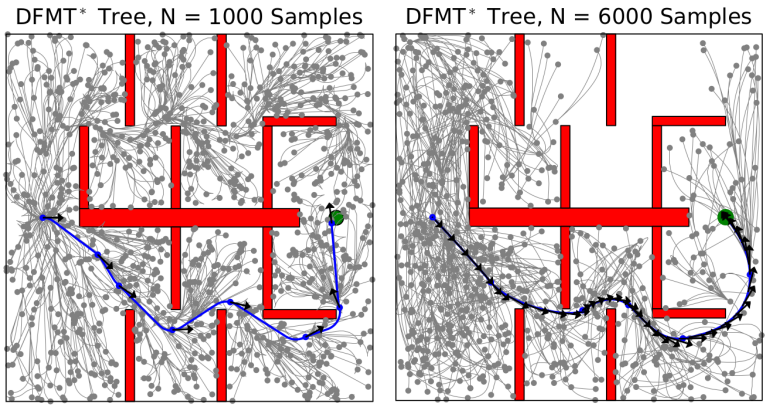
where $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{u} \in \mathbb{R}^m$. Note that the state space X is not necessarily the same as the configuration space C, but the configuration q is derivable from the state x. As was previously mentioned, the configuration space is something that can be chosen to capture the information that is necessary for obstacle avoidance. However to include dynamics constraints it is required that the motion planning now be done in the state space X.

The RRT algorithm can be extended to the kinodynamic case with relative simplicity. In particular, a random state x is sampled from the state space X and its nearest neighbor x_{near} on the current tree T is found. Instead of connecting x and x_{near} with a straight line (which is likely not dynamically feasible), a

[10] E. Schmerling, L. Janson, and M. Pavone. "Optimal sampling-based motion planning under differential constraints: the driftless case". In: *IEEE International Conference on Robotics and Automation*. 2015, pp. 2368–2375

random control $u \in U$ and random time t are sampled. Then, the state is propagated forward by integrating the differential equations (6.2) with the chosen u for a time t and initial condition x_{near}. The resulting state x_{new} is then added to the tree if the path from x_{near} to x_{new} is collision free. This is referred to as a *forward-propagation-based* approach.

Another approach to kinodynamic planning leverage *steering-based* algorithms. In these approaches, the planner selects two points in the state space x and x' and then uses a steering subroutine to find a feasible trajectory to connect these points. Crucially, these approaches only work well if the steering subroutine is *efficient*. This approach is be particularly well suited for differential flat systems.



(a) Reeds-Shepp car (b) Double integrator system

Figure 6.4: Results from a kinodynamic planner called Differential FMT* (DFMT*) (Schmerling et al.). The figure on the left shows the results for a Reeds-Shepp car model, and on the right is a double integrator model.

6.6 *Deterministic Sampling-Based Motion Planning*

Probabilistic sampling-based algorithms, such as the probabilistic roadmap (PRM) and the rapidly exploring random tree (RRT) algorithms, have been quite successful in practice for robotic motion planning and often have nice theoretical properties (e.g. in terms of probabilistic completeness or even asymptotic optimality). Such algorithms are *probabilistic* because they compute a path by connecting independently and identically distributed (i.i.d.) random points in the configuration space. However, this randomization introduces several challenges for practical use, including certification for safety-critical applications and the ability to use offline computation to improve real-time execution. Hence, it is important to ask whether similar (or better) theoretical guarantees and practical performance could be obtained by considering *deterministic* approaches.

An important metric for answering this question is referred to as the l_2-dispersion.

Definition 6.6.1 (l_2-dispersion)**.** *For a finite set S of points contained in $X \subset R^d$, its*

l_2-dispersion $D(S)$ is defined as:

$$D(S) := \sup_{x \in X} \min_{s \in S} \|s - x\|_2. \tag{6.3}$$

Intuitively, the l_2-dispersion of S quantifies how well a space is covered by the set of points in S in terms of the largest Euclidean ball that touches and contains none of the points. For a fixed number of samples, a small l_2-dispersion (only a small radius ball can be fit among the points of S without touching or containing any) means that the points are more uniformly distributed.

To create a deterministic sampling based motion planning algorithm, it is desirable to generate a set of samples S with low-dispersion. In fact, low-dispersion sampling sequences exist that give sets S with l_2-dispersion $D(S)$ on the order of $O(n^{-1/d})$ where d is the dimension of the space. Additionally, for $d = 2$ it is possible to create sequences of points S that *minimize* the l_2-dispersion. Then, if the set S of n samples has l_2-dispersion that satisfies

$$D(S) \leq \gamma n^{-1/d},$$

for some $\gamma > 0$, and if $\lim_{n \to \infty} n^{1/d} r_n = \infty$, then the arc length of the path c_n returned will converge to the optimal path c^* as $n \to \infty$.

In summary, deterministic sampling can be used to generate motion planning algorithms. These deterministic algorithms still maintain the asymptotic optimality guarantees that probabilistic planners do, and can even use a smaller connection radius r_n.

6.7 Exercises

All exercises for this chapter can be found in the online repository:

`https://github.com/PrinciplesofRobotAutonomy/AA274A_HW2`.

6.7.1 Rapidly-Exploring Random Trees

Complete *Problem 2: Rapidly-Exploring Random Trees (RRT)*, where you will implement the RRT sample-based motion planning algorithm to plan paths in simple 2D environments. Additionally, in this problem you will start with a simple geometric planner that does not consider robot dynamics, but will then extend the RRT algorithm to consider a wheeled robot modeled with Dubins car dynamics.

6.7.2 Motion Planning & Control

Complete *Problem 3: Motion Planning & Control*, where you will combine an A* planner with a differential flatness-based tracking controller and a pose stabilization controller to enable a unicycle robot to autonomously move through a 2D environment. Note that this problem requires exercises from previous chapters to be completed first.

6.7.3 Bi-Directional Sampling-based Motion Planning

Complete *Extra Problem: Bi-Directional Sampling-based Motion Planning*, where you will implement a variation of the RRT algorithm known as RRT-Connect, which uses a bi-direction approach to building the RRT tree. This algorithm will be implemented for both a simple geometric path planner as well as for a Dubins car robot.

Part II

Robot Perception

7
Introduction to Robot Sensors

The three main pillars of robotic autonomy can broadly be characterized as perception, planning, and control (i.e. the "see, think, act" cycle). Perception categorizes those challenges associated with a robot sensing and understanding its environment, which are addressed by using various sensors and then extracting meaningful information from their measurements. The next few chapters will focus on the perception/sensing problem in robotics, and in particular will introduce common sensors utilized in robotics, their key performance characteristics, as well as strategies for extracting useful information from the sensor outputs.

Introduction to Robot Sensors

Robots operate in diverse environments and often require diverse sets of sensors to appropriately characterize them. For example, a self-driving car may utilize cameras, stereo cameras, lidar, and radar. Additionally, sensors are also required for characterizing the physical state of the vehicle itself, for example wheel encoders, heading sensors, GNSS positioning sensors[1], and more[2].

[1] Global Navigation Satellite System

[2] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, 2011

7.1 Sensor Classifications

To distinguish between sensors that measure the environment and sensors that measure quantities related the robot itself, sensors are categorized as either *proprioceptive* or *exteroceptive*.

Definition 7.1.1 (Proprioceptive)**.** *Proprioceptive sensors measure values internal to the robot, for example motor speed, wheel load, robot arm joint angles, and battery voltage.*

Definition 7.1.2 (Exteroceptive)**.** *Exteroceptive sensors acquire information from the robot's environment, for example distance measurements, light intensity, and sound amplitude.*

Generally speaking, exteroceptive sensor measurements are often more likely to require interpretation by the robot in order to extract meaningful environ-

mental features. In addition to characterizing *what* the sensor measures, it is also useful to characterize sensors based on *how* they operate. In particular it is common to characterize a sensor as either *passive* or *active*.

Definition 7.1.3 (Passive Sensor)**.** *Passive sensors measure ambient environmental energy entering the sensor, for example thermometers and cameras.*

Definition 7.1.4 (Active Sensor)**.** *Active sensors emit energy into the environment and measure the reaction, for example ultrasonic sensors and laser rangefinders.*

Classifying a sensor as active or passive is important because this property introduces unique challenges. For example the performance of passive sensors depend heavily on the environment, such as a camera being dependent on the ambient lighting to get a good image.

7.2 Sensor Performance

Different types sensors also have different types of performance characteristics. Some sensors provide extreme accuracy in well-controlled laboratory settings but are overcome with error when subjected to real-world environmental variations. Other sensors provide narrow, high-precision data in a wide variety of settings. In order to quantify and compare such performance characteristics it is necessary to define relevant metrics. These metrics are generally either related to *design specifications* or *in situ* performance (i.e. how well a sensor performs in the real environment).

7.2.1 Design Specification Metrics

A number of performance characteristics are specifically considered when designing the sensor, and are also used to quantify its overall nominal performance capabilities.

1. *Dynamic range* quantifies the ratio between the lower and upper limits of the sensor inputs under normal operation. This metric is usually expressed in decibels (dB), which is computed as

$$\mathrm{DR} = 10 \log_{10}(r) \ \ [\mathrm{dB}],$$

   where r is the ratio between the upper and lower limits. In addition to dynamic range (ratio), the actual range is also an important sensor metric. For example, an optical rangefinder will have a minimum operating range and can thus provide spurious data when measurements are taken with the object closer than that minimum.

2. *Resolution* is the minimum difference between two values that can be detected by a sensor. Usually, the lower limit of the dynamic range of a sensor is equal to its resolution. However, this is not necessarily the case for digital sensors.

3. *Linearity* characterizes whether or not the sensor's output depends linearly on the input.

4. *Bandwidth* or *frequency* is used to measure the speed with which a sensor can provide a stream of readings. This metric is usually expressed in units of Hertz (Hz), which is measurements per second. High bandwidth sensors are usually desired so that information can be updated at a higher rate. For example, mobile robots may have a limit on their maximum speed based on the bandwidth of their obstacle detection sensors.

7.2.2 *In Situ Performance Metrics*

Metrics related to the design specifications can be reasonably quantified in a laboratory environment and then extrapolated to predict performance during real-world deployment. However, a number of important sensor metrics cannot be adequately characterized in laboratories settings since they are influenced by complex interactions between the environment.

1. *Sensitivity* defines the ratio of change in the output from the sensor to a change in the input. High sensitivity is often undesirable because any noise to the input can be amplified, but low sensitivity might degrade the ability to extract useful information from the sensor's measurements. *Cross-sensitivity* defines the sensitivity to environmental parameters that are unrelated to the sensor's target quantity. For example, a flux-gate compass can demonstrate high sensitivity to magnetic north and is therefore useful for mobile robot navigation. However, the compass also has high sensitivity to ferrous building materials, so much so that its cross-sensitivity often makes the sensor useless in some indoor environments. High cross-sensitivity of a sensor is generally undesirable, especially when it cannot be modeled.

2. *Error* of a sensor is defined as the difference between the sensor's output measurements and the true values being measured, within some specific operating context. Given a true value v and a measured value m, the error is defined as $e := m - v$.

3. *Accuracy* is defined as the degree of conformity between the sensor's measurement and the true value, and is often expressed as a proportion of the true value (e.g., 97.5% accuracy). Thus small error corresponds to high accuracy and vice versa. For a measurement m and true value v, the accuracy is defined as $a := 1 - |m - v|/v$. Since obtaining the true value v can be difficult or impossible, characterizing sensor accuracy can be challenging.

4. *Precision* defines the reproducibility of the sensor results. For example, a sensor has high precision if multiple measurements of the same environmental quantity are similar. It is important to note that precision is not the same as accuracy, a highly precise sensor can still be highly inaccurate.

7.2.3 Sensor Errors

When discussing in situ performance metrics such as accuracy and precision, it is often important to also be able to reason about the sources of sensor errors. In particular it is important to distinguish between two main types of error, *systematic errors* and *random errors*.

1. *Systematic errors* are caused by factors or processes that can in theory be modeled (i.e. they are deterministic and therefore reproducible and predictable). Calibration error is a classic source of systematic error in sensors.

2. *Random errors* cannot be predicted using a sophisticated model (i.e. they are stochastic and unpredictable). Hue instability in a color camera, spurious rangefinding errors, and black level noise in a camera are all examples of random errors.

In order to reliably use a sensor in practice it is useful to have a characterization of the systematic and random errors, which could allow for corrections to make the sensor more accurate and provide information about its precision. Quantifying the sensor error and identifying sources of error is referred to as *error analysis*. Error analysis for a typical sensor might involve identifying all of the sources of systematic errors, modeling random errors (e.g. by Gaussian distributions), and then propagating the errors from each identified source to determine the overall impact on the sensor output.

Unfortunately, it is typically challenging to perform a complete error analysis in practice for several reasons. One of the main reasons is due to a *blurring* between systematic and random errors that is the result of changes to the operating environment. For example, exteroceptive sensors on a mobile robot will have constantly changing measurement sources as the robot moves through the environment, and could even be influenced by the motion of the robot itself. Therefore, an exteroceptive sensor's error profile may be heavily dependent on the particular environment and even the particular state of the robot! As a more concrete example, active ranging sensors tend to have failure modes that are triggered largely by specific relative positions of the sensor and environment targets. For example, when oriented at specific angles to a smooth sheetrock wall a sonar sensor will produce specular reflections that result in highly inaccurate range measurements. During the motion of a robot, these particular relative angles would likely occur at stochastic intervals and therefore this error source might be considered *random*. Yet, if the robot were to stop at the specific angle for inducing specular reflections, the error would be persistent and could be modeled as a systematic error. In summary, while systematic and random sensor errors might be well defined in controlled settings, in practical settings characterizing error becomes a lot more challenging due to the complexity and quantity of potential error sources.

7.2.4 Modeling Uncertainty

If all sensor measurement errors were systematic and could be modeled then theoretically they could be corrected for. However in practice this is not the case and therefore some alternative representation of the sensor error is needed. In particular, characterizing uncertainty due to random errors is typically accomplished by using *probability distributions*.

Since it is effectively impossible to know all of the sources of random error for a sensor it is common to make assumptions about what the distribution of the sensor error looks like. For example, it is commonly assumed that random errors are zero-mean and symmetric, or to go slightly further that they are Gaussian. More broadly, it is commonly assumed that the distribution is *unimodal*. These assumptions are usually made because they simplify the mathematical tools used for performing theoretical analyses.

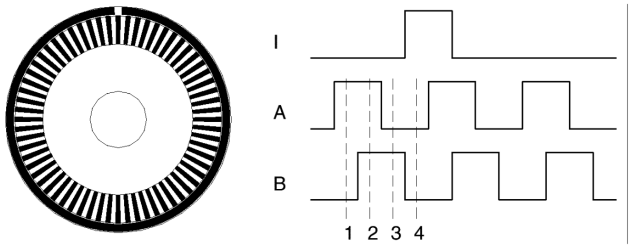
However, it is also crucial to understand the limitations of these assumptions. In fact, in many cases even the most broad assumptions (e.g. that the distribution is unimodal) can be quite wrong in practice. As an example consider the sonar sensor once again. When ranging an object that reflects the sound signal well, the sonar will exhibit high accuracy and the random errors will generally be based on noise (e.g. from the timing circuitry). In this operating instance it might be a perfectly fine assumption that the noise distribution is unimodal and perhaps even Gaussian. However, if the sonar sensor is moving through an environment and is faced with materials that cause coherent reflection (rather than directly returning the sound signal to the sonar sensor) then overestimates of the distance to the object are likely. In this case, the error will be biased toward positive measurement error and will be far from the correct value. Therefore it can be seen that modeling the sonar sensor uncertainty over *all* operating regimes of the robot would at least require a bimodal distribution in this case. Additionally, since overestimation is more common than underestimation, the distribution should also be asymmetric. As a second example, consider ranging via stereo vision. Once again, at least two modes of operation can be identified. If the stereo vision system correctly correlates two images, then the resulting random error will be caused by camera noise and will limit the measurement accuracy. But the stereo vision system can also correlate two images incorrectly. In such a case stereo vision will exhibit gross measurement error, and one can easily imagine such behavior violating both the unimodal and the symmetric assumptions.

7.3 Common Sensors on Mobile Robots

7.3.1 Encoders

Encoders are electro-mechanical devices that convert motion into a sequence of digital pulses, which can then be converted to relative or absolute position measurements. These sensors are commonly used for wheel/motor sensing

to determine rotation angle and rotation rate. Since these sensors have vast applications outside of mobile robotics there has been substantial development in low-cost encoders that offer excellent resolution. In mobile robotics, encoders are one of the most popular means to control the position or speed of wheels and other motor-driven joints. These sensors are proprioceptive and therefore their estimates are expressed in the reference frame of the robot.



State	Ch A	Ch B
s_1	high	low
s_2	high	high
s_3	low	high
s_4	low	low

Figure 7.1: Quadrature optical wheel encoder. (Figure from Siegwart et al.)

Optical encoders shine light onto a photodiode through slits in a metal or glass disc, and measure the sine or square wave pulses that result from disk rotation (see Figure 7.1). After some signal processing it is possible to integrate the number of wave peaks to determine how much the disk has rotated. Encoder resolution is measured in cycles per revolution (CPR) and the minimum angular resolution can be readily computed from an encoder's CPR rating. A typical encoder in mobile robotics may have 2000 CPR, while the optical encoder industry can readily manufacture encoders with 10,000 CPR. In terms of bandwidth, it is of course critical that the encoder is sufficiently fast to handle the expected shaft rotation rates. Luckily, industrial optical encoders present no bandwidth limitation to mobile robot applications. Usually in mobile robotics the quadrature encoder is used. In this case, a second illumination and detector pair is placed 90 degrees shifted with respect to the original in terms of the rotor disc. The resulting twin square waves, shown in Figure 7.1, provide significantly more information. The ordering of which square wave produces a rising edge first identifies the direction of rotation. Furthermore, the resolution is improved by a factor of four with no change to the rotor disc. Thus, a 2000 CPR encoder in quadrature yields 8000 counts.

As with most proprioceptive sensors, encoders typically operate in a very predictable and controlled environment. Therefore systematic errors and cross-sensitivities can be accounted for. In practice, the accuracy of optical encoders is often assumed to be 100% since any encoder errors are dwarfed by errors in downstream components.

7.3.2 Heading Sensors

Heading sensors can be proprioceptive (e.g. gyroscopes, inclinometers) or exteroceptive (e.g. compasses). They are used to determine the robot's orientation in space. Additionally, they can also be used to obtain position estimates by fusing

the orientation and velocity information and integrating, a process known as *dead reckoning*.

Compasses: Compasses are exteroceptive sensors that measure the earth's magnetic field to provide a rough estimate of direction. In mobile robotics, digital compasses using the Hall effect are popular and they are inexpensive but often suffer from poor resolution and accuracy. Flux gate compasses have improved resolution and accuracy, but are more expensive and physically larger. Both compass types are vulnerable to vibrations and disturbances in the magnetic field, and are therefore less well suited for indoor applications.

Gyroscopes: Gyroscopes are heading sensors that preserve their orientation with respect to a fixed *inertial* reference frame. Gyroscopes can be classified in two categories: *mechanical gyroscopes* and *optical gyroscopes*. Mechanical gyroscopes rely on the angular momentum of a fast-spinning rotor to keep the axis of rotation inertially stable. Generally the inertial stability increases with the spinning speed ω, the precession speed Ω, and the wheel's inertia I since the reactive torque τ can be expressed as:

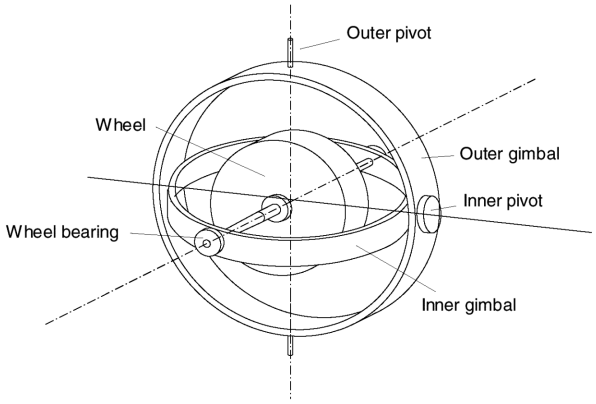


Figure 7.2: Two-axis mechanical gyroscope. (Figure from Siegwart et al.)

$$\tau = I\omega\Omega.$$

Mechanical gyroscopes are configured with an inner and outer gimbal as seen in Figure 7.2 such that no torque can be transmitted from the outer pivot to the wheel axis. This means that the spinning axis will therefore be space-stable (i.e. fixed in an inertial reference frame). Nevertheless, friction in the bearings of the gimbals may introduce small torques, which over time introduces small errors. A high quality mechanical gyroscope can cost up to $100,000 and has an angular drift of about 0.1 degrees in 6 hours.

Optical gyroscopes are a relatively new invention. They use angular speed sensors with two monochromatic light beams, or lasers, emitted from the same source. Two beams are sent, one clockwise and the other counterclockwise,

through an optical fiber. Since the laser traveling in the direction of rotation has a slightly shorter path, it will have a higher frequency. This frequency difference δf is proportional to the angular velocity, which can therefore be estimated. In modern optical gyroscopes, bandwidth can easily exceed 100 kHz, while resolution can be smaller than 0.0001 degrees/hr.

7.3.3 *Accelerometer*

An accelerometer is a device used to measure net accelerations (i.e. the net external forces acting on the sensor, including gravity). Mechanical accelerometers are essentially spring-mass-damper systems that can be represented by the second order differential equation[3]:

$$F_{\text{applied}} = m\ddot{x} + c\dot{x} + kx$$

where m is the proof mass, c is the damping coefficient, k is the spring constant, and x is the relative position to a reference equilibrium. When a static force is applied, the system will oscillate until it reaches a steady state where the steady state acceleration would be given as:

$$a_{applied} = \frac{kx}{m}.$$

The design of the sensor chooses m, c, and k such that system can stabilize quickly and then the applied acceleration can be calculated from steady state. Modern accelerometers, such as the ones in mobile phones, are usually very small and use Micro Electro-Mechanical Systems (MEMS), which consist of a cantilevered beam and a proof mass. The deflection of the proof mass from its neutral position is measured using capacitive or piezoelectric effects.

7.3.4 *Inertial Measurement Unit (IMU)*

Inertial measurement units (IMU) are devices that use gyroscopes and accelerometers to estimate their relative position, orientation, velocity, and acceleration with respect to an inertial reference frame. Their general working principle is shown in Figure 7.3.

The gyroscope data is integrated to estimate the vehicle orientation while the three accelerometers are used to estimate the instantaneous acceleration of the vehicle. The acceleration is then transformed to the local navigation frame by means of the current estimate of the vehicle orientation relative to gravity. At this point the gravity vector can be subtracted from the measurement. The resulting acceleration is then integrated to obtain the velocity and then integrated again to obtain the position, provided that both the initial velocity and position are a priori known. To overcome the need of knowing of the initial velocity, the integration is typically started at rest when the velocity is zero.

One of the fundamental issues with IMUs is the phenomenon called *drift*, which describes the slow accumulation of errors over time. Drift in any one

[3] G. Dudek and M. Jenkin. "Inertial Sensors, GPS, and Odometry". In: *Springer Handbook of Robotics*. Springer, 2008, pp. 477–490

component will also effect the downstream components as well. For example, drift in the gyroscope unavoidably undermines the estimation of the vehicle orientation relative to gravity, which results in incorrect cancellation of the gravity vector. Additionally, errors in acceleration measurements will cause the integrated velocity to drift in time (which will in turn also cause position estimate drift). To account for drift periodic references to some external measurement is required. In many robot applications, such an external reference may come from GNSS position measurements, cameras, or other sensors.

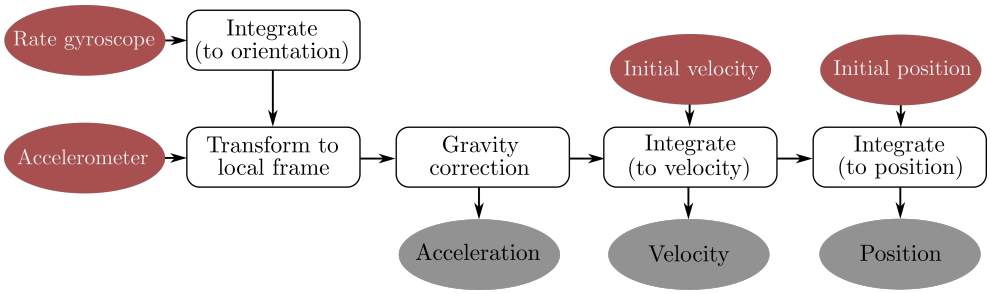


Figure 7.3: Inertial measurement unit (IMU) block diagram.

7.3.5 *Beacons*

Beacons are signaling devices with precisely known positions (e.g. stars and lighthouses are classic examples). Position of a mobile robot can be determined by knowing the position of the beacon and by having access to relative position measurements. The GNSS positioning system and camera-based motion capture system for indoor use are more advanced examples. GNSS based positioning is extremely popular in robotics, and works by processing synchronized signals from at least four satellites. Signals from four satellites are needed (at a minimum) to enable the estimation of four unknown quantities (the three position coordinates plus a clock correction). Modified GNSS-based methods, such as differential GPS, can be used to increase positioning accuracy.

7.3.6 *Active Ranging*

Active ranging sensors provide direct measurements of distance to objects in the vicinity of the sensor. These sensors are important in robotics for both localization and environment reconstruction. There are two main types of active ranging sensors: time-of-flight active ranging sensors (e.g. ultrasonic, laser rangefinder, and time-of-flight cameras) and geometric active ranging sensors (e.g. based on optical triangulation and structured light).

Time-of-flight Active Ranging: Time-of-flight active ranging sensors make use of the propagation speed of sounds or electromagnetic waves. In particular, the travel distance is given by

$$d = ct,$$

Figure 7.4: The Velodyne HDL-64E High Definition Real-Time 3D Lidar sensor, a time-of-flight active ranging sensor. (Image retrieved from velodynelidar.com)

where d is the distance traveled, c is the speed of wave propagation, and t is the time of flight. The propagation speed c of sound is approximately $0.3m/ms$ whereas the speed of electromagnetic signals is $0.3m/ns$, which is 1 million times faster! The time of flight for a distance of 3 meters is 10 milliseconds for an ultrasonic system, but only 10 nanoseconds for a laser rangefinder, which makes measuring the time of flight t for electromagnetic signals more technologically challenging. This explains why laser range sensors have only recently become affordable and robust for use on mobile robots. The quality of different time-of-flight range sensors may depend on:

1. uncertainties in determining the exact time of arrival of the reflected signal,
2. inaccuracies in the time-of-flight measurement (particularly with laser range sensors),
3. the dispersal cone of the transmitted beam (mainly with ultrasonic range sensors),
4. interaction with the target (e.g. surface absorption, specular reflections),
5. variation of propagation speed,
6. the speed of the mobile robot and target (in the case of a dynamic target).

Geometric Active Ranging: Geometric active ranging sensors use geometric properties in the measurements to establish distance readings. Generally, these sensors project a known pattern of light and then geometric properties can be used to analyze the reflection and estimate range via triangulation. Optical triangulation sensors (1D) transmit a collimated (parallel rays of light) beam toward the target and use a lens to collect reflected light and project it onto a position-sensitive device or linear camera. Structured light sensors (2D or 3D) project a known light pattern (e.g. point, line, or texture) onto the environment. The reflection is captured by a receiver and then, together with known geometric values, range is estimated via triangulation.

7.3.7 Other Sensors

Some classical examples of other sensors include radar, tactile sensors, and vision based sensors (e.g. cameras). Radar sensors leverage the Doppler effect to produce velocity relative velocity measurements. Tactile sensors are particularly useful for robots that interact physically with their environment.

7.4 Computer Vision

Vision sensors have become crucial sensors for perception in the context of robotics. This is generally due to the fact that vision provides an enormous

amount of information about the environment and enables rich, intelligent interaction in dynamic environments[4]. The main challenges associated with vision-based sensing are related to processing digital images to extract salient information like object depth, motion and object detection, color tracking, feature detection, scene recognition, and more. The analysis and processing of images are generally referred to as *computer vision* and *image processing*. Tremendous advances and new theoretical findings in these fields over the last several decades have led to sophisticated computer vision and image processing techniques to be utilized in industrial and consumer applications such as photography, defect inspection, monitoring and surveillance, video games, movies, and more. This section introduces some fundamental concepts related to these fields, and in particular will focus on cameras and camera models.

[4] In fact, the human eye provides millions of bits of information per second.

7.4.1 Digital Cameras

While the basic idea of a camera has existed for thousands of years, the first clear description of one was given by Leonardo Da Vinci in 1502 and the oldest known published drawing of a *camera obscura* (a dark room with a pinhole to image a scene) was shown by Gemma Frisius in 1544. By 1685, Johann Zahn had designed the first portable camera, and in 1822 Joseph Nicephore Niepce took the first physical photograph.

Modern cameras consist of a sensor that captures light and converts the resulting signal into a digital image. Light falling on an imaging sensor is usually picked up by an active sensing area, integrated for the duration of the exposure (usually expressed as the shutter speed, e.g. $1/125, 1/60, 1/30$ of a second), and then passed to a set of sense amplifiers. The two main kinds of sensors used in digital cameras today are charge coupled devices (CCD) and complementary metal oxide on silicon (CMOS) sensors. A CCD chip is an array of light-sensitive picture elements (pixels), and can contain between 20,000 and several million pixels total. Each pixel can be thought of as a light-sensitive discharging capacitor that is 5 to 25μm in size. While complementary metal oxide semiconductor (CMOS) chips also consist of an array of pixels, they are quite different from CCD chips. In particular, along the side of each pixel are several transistors specific to that pixel. CCD sensors have typically outperformed CMOS for quality sensitive applications such as digital single-lens-reflex cameras, while CMOS sensors are better for low-power applications. However, today CMOS sensors are standard in most digital cameras.

7.4.2 Image Formation

Before reaching the camera's sensor, light rays first originate from a light source. In general the rays of light reflected by an object tend to be scattered in many directions and may consist of different wavelengths. Averaged over time, the emitted wavelengths and directions for a specific object can be precisely described using object-specific probability distribution functions. In particular,

the light reflection properties of a given object are the result of how light is reflected, scattered, or absorbed based on the object's surface properties and the wavelength of the light. For example, an object might look blue because blue wavelengths of light are primarily scattered off the surface while other wavelengths are absorbed. Similarly, a black object looks black because it absorbs most wavelengths of light, and a perfect mirror reflects all visible wavelengths.

Cameras capture images by sensing these light rays on a photoreceptive surface (e.g. a CCD or a CMOS sensor). However, since light reflecting off an object is generally scattered in many directions, simply exposing a planar photoreceptive surface to these reflected rays would result in many rays being captured at each pixel. This would lead to blurry images! A solution to this issue is to add a barrier in front of the photoreceptive surface that blocks most of these rays, and only lets some of them pass through an aperture (see Figure 7.5). The earliest approach to filtering light rays in this way was to have a small hole in the barrier surface. Cameras with this type of filter were referred to as *pinhole* cameras.

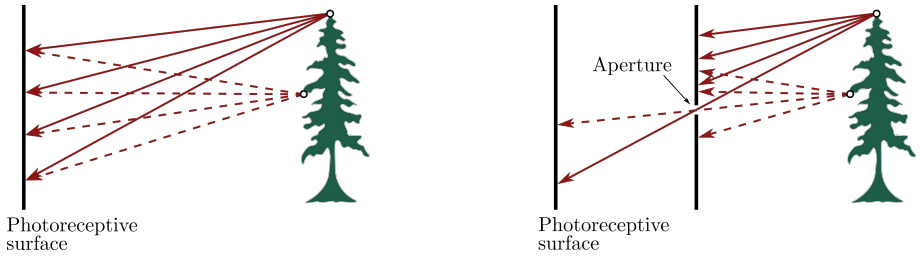


Figure 7.5: Light rays on a photoreceptive surface referred to as the image plane. On the left, numerous rays being reflected and scattered by the object leads to blurry images whereas (on the right), a barrier has been added so that the scattered light rays can be distinguished.

7.4.3 *Pinhole Camera Model*

A pinhole camera has no lens but rather a single very small aperture. Light from the scene passes through this pinhole aperture and projects an inverted image onto the image plane (see Figure 7.6). While modern cameras do not operate in this way, the principles of the pinhole camera can be used to derive useful mathematical models.

To develop the mathematical pinhole camera model, several useful reference frames are defined. First, the *camera reference frame* is centered at a point O (see Figure 7.6) that is at a focal length f in front of the image plane. This reference frame with directions (i, j, k) is defined with the k axis coincident with the *optical axis* that points toward the image plane. The coordinates of a point in the camera frame are denoted by uppercase $P = (X, Y, Z)$. When a ray of light is emitted from a point P and passes through the pinhole at point O, it gets captured on the image plane at a point p. Since these points are all collinear it is possible to deduce the following relationships between the coordinates $P = (X, Y, Z)$ and $p = (x, y, z)$:

$$x = \lambda X, \quad y = \lambda Y, \quad z = \lambda Z,$$

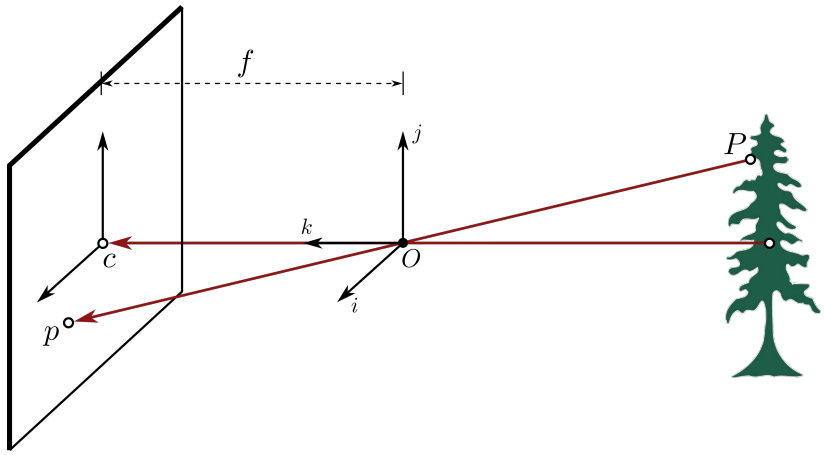


Figure 7.6: Pinhole camera model. Due to the geometry of the pinhole camera system, the object's image is inverted on the image plane. In this figure, O is the camera center, c is the image center, and p the principal point.

for some $\lambda \in \mathbb{R}$. This leads to the relationship:

$$\lambda = \frac{x}{X} = \frac{y}{Y} = \frac{z}{Z}.$$

Further, from the geometry of the camera it can be seen that $z = f$ where f is the focal length, such that these expressions can be rewritten as:

$$x = f\frac{X}{Z}, \quad y = f\frac{Y}{Z}. \tag{7.1}$$

Therefore the position of the pixel on the image plane that captures a ray of light from the point P can be computed.

7.4.4 *Thin Lens Model*

One of the main issues with having a fixed pinhole aperture is that there is a trade-off associated with the aperture's size. A large aperture allows a greater number of light rays to pass through, which leads to blurring of the image. However, a small aperture lets through fewer light rays and the resulting image is darker. As a solution, lenses can focus light via refraction and can be used to replace the aperture, therefore avoiding the need for these trade-offs.

A similar mathematical model to the pinhole model can be introduced for lenses by using properties from Snell's law. Figure 7.7 shows a diagram of the most basic lens model, which is the *thin lens model* (which assumes no optical distortion due to the curvature of the lens). Snell's law states that rays passing through the center of the lens are not refracted, and those that are parallel to the optical axis are focused on the focal point labeled F'. In addition, all rays passing through P are focused by the thin lens on the point p. From the geometry of similar triangles, a mathematical model similar to (7.1) is developed:

$$\frac{y}{Y} = \frac{z}{Z}, \quad \frac{y}{Y} = \frac{z-f}{f} = \frac{z}{f} - 1, \tag{7.2}$$

where again the point P has coordinates (X, Y, Z), its corresponding point p on the image plane has coordinates (x, y, z), and f is the focal length. Combining

these two equations yields the *thin lens equation*:

$$\frac{1}{z} + \frac{1}{Z} = \frac{1}{f}. \tag{7.3}$$

Note that in this model for a particular focal length f, a point P is only in sharp focus if the image plane is located a distance z from the lens. However, in practice an acceptable focus is possible withing some range of distances (called depth of field or depth of focus). Additionally, if Z approaches infinity light would focus a distance of f away from the lens. Therefore, this model is essentially the same as a pinhole model if the lens is focused at a distance of infinity. As can be seen, this formula can also be used to estimate the distance to an object by knowing the focal length f and the current distance of the image plane to the lens z. This technique is called *depth from focus*.

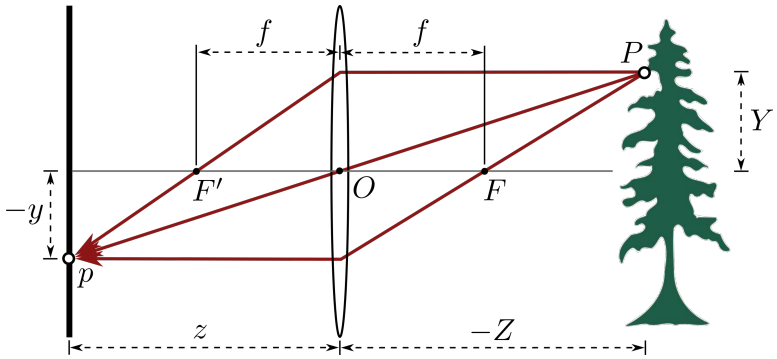


Figure 7.7: The thin lens model.

8
Camera Models and Calibration

The previous chapter began an introduction to the problem of robotic perception, which consists of tasks related to sensing and understanding the robot's own movements as well as the environment in which it operates[1]. This chapter continues that discussion by diving more deeply into one of the most powerful and challenging tools in robotic perception: computer vision. In particular, this chapter will focus on some of the fundamental mathematical tools for calibrating cameras and processing their images to extract some useful information about the scene[2,3].

[1] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, 2011

[2] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2011

[3] R. Hartley and A. Zisserman. "Camera Models". In: *Multiple View Geometry in Computer Vision*. Academic Press, 2002

Camera Models and Calibration

As was discussed in the previous chapter, cameras provide a crucial sensing modality in the context of robotics. This is generally due to the fact that images inherently contain an enormous amount of information about the environment. However, while images do contain a lot of information, extracting the information that is relevant to the robot is quite challenging. One of the most basic tasks related to image processing is determining how a particular point in the scene maps to a point in the camera image, which is sometimes referred to as *perspective projection*. Last chapter, the *pinhole camera model* and the *thin lens model* were presented, and in this chapter the pinhole camera model is leveraged to further explore perspective projection[4].

[4] All results also hold under the thin lens model, assuming the camera is focused at ∞.

8.1 Perspective Projection

The pinhole camera model, shown graphically in Figure 8.1, can be used to mathematically define relationships between points P in the scene and points p on the image plane. Notice that any point P in the scene can represented in two ways: in camera frame coordinates (denoted as P_C) or in world frame coordinates (denoted as P_W). The overall objective of this section is to find derive a mathematical model that can be used to map a point P_W expressed in world frame coordinates to a point p on the image plane. To accomplish this two transformations are combined together, namely a transformation of P from

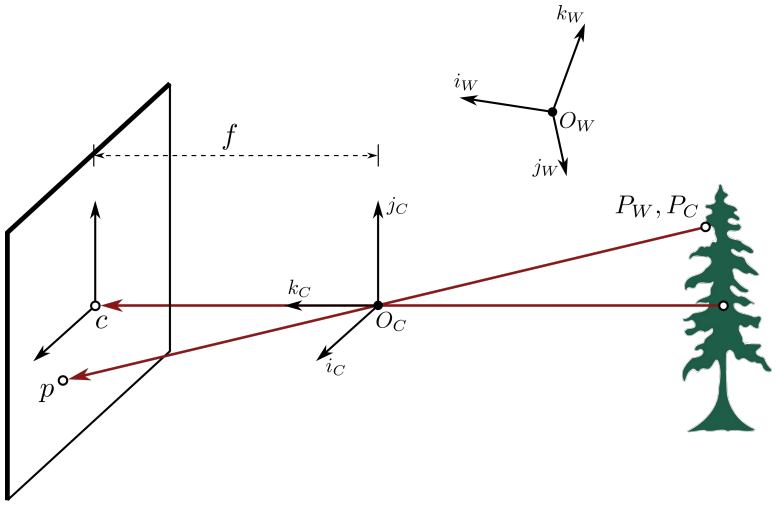


Figure 8.1: Graphical representation of the pinhole camera model. In this model the point O_C is the camera center, c is the image center, and f is the focal length of the camera. It is assumed that all light rays from point P in the scene pass through point O_C and are captured on the image plane at point p.

world frame coordinates to camera frame coordinates (P_W to P_C) and a transformation from camera coordinates to image coordinates (P_C to p).

8.1.1 *Mapping Camera Frame Coordinates to Image Coordinates ($P_C \rightarrow p$)*

The first step considered is the mapping from a point in the scene expressed in camera frame coordinates, P_C, to the corresponding point on the image plane, p, using the pinhole camera model. Recall from the previous chapter the pinhole camera equations:

$$x = f\frac{X_C}{Z_C}, \quad y = f\frac{Y_C}{Z_C}, \tag{8.1}$$

where $P_C = (X_C, Y_C, Z_C)$, $p = (x, y)$, and f is the focal length of the pinhole camera[5].

Note that the quantities x and y are coordinates in the *camera frame*, but it is often desirable to express the point p in terms of *pixel coordinates*. However, pixel coordinates are generally defined with respect to a reference frame in the lower corner of the image plane (to avoid negative coordinates). This new reference frame is shown in Figure 8.2, where the image center c is defined in this new reference frame with coordinates $(\tilde{x}_0, \tilde{y}_0)$, where $\tilde{(\cdot)}$ is the notation used to denote a coordinate with respect to this new reference frame. In this new reference frame, the point P_C gets mapped to the coordinates $(\tilde{x}, \tilde{y})$ by:

$$\tilde{x} = f\frac{X_C}{Z_C} + \tilde{x}_0, \quad \tilde{y} = f\frac{Y_C}{Z_C} + \tilde{y}_0. \tag{8.2}$$

Finally, these new coordinates can be mapped to pixel coordinates if the number of pixels per unit distance are known. In particular, the point P_C is mapped to pixel coordinates (u, v) by:

$$u = \alpha\frac{X_C}{Z_C} + u_0, \quad v = \beta\frac{Y_C}{Z_C} + v_0, \tag{8.3}$$

[5] The z term of p is generally not included simply because $z = f$ is a fixed value.

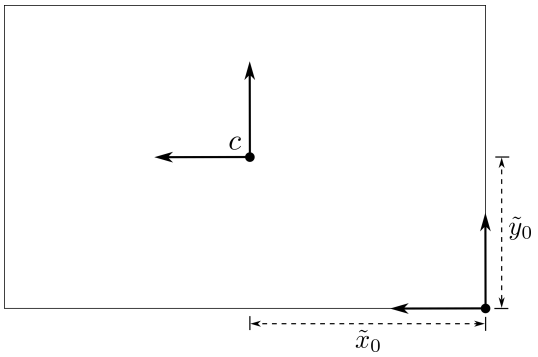


Figure 8.2: A new reference frame with coordinates denoted by $(\tilde{\cdot})$ is defined with its origin in the lower corner of the image plane. The image center coordinates in this new frame are denoted $(\tilde{x}_0, \tilde{y}_0)$.

where $\alpha = k_x f$, $u_0 = k_x \tilde{x}_0$, $\beta = k_y f$, $v_0 = k_y \tilde{y}_0$, and k_x and k_y are the number of pixels per unit distance in image coordinates.

Homogeneous Coordinates: Note that the transformation from the point P_C in camera frame coordinates to p in pixel coordinates given by (8.3) is not linear. However, this transformation can be represented as a linear mapping[6] through an additional change of coordinates. In particular, the points P_C and p will be expressed in *homogeneous coordinates*.

[6] Expressing the perspective projection as a linear map will simplify the mathematics later on.

For a 2D point (x_1, x_2) or a 3D point (x_1, x_2, x_3) in Euclidean space, the point can be represented in homogeneous coordinates by the transformation:

$$(x_1,\, x_2) \implies (\alpha x_1,\, \alpha x_2,\, \alpha), \quad \text{and} \quad (x_1,\, x_2,\, x_3) \implies (\alpha x_1,\, \alpha x_2,\, \alpha x_3,\, \alpha), \quad (8.4)$$

for any $\alpha \neq 0$. These new coordinates are called homogeneous coordinates because the scaling factor α can be chosen arbitrarily as long as $\alpha \neq 0$. A set of homogeneous coordinates can then be transformed back by:

$$(y_1,\, y_2,\, y_3) \implies \left(\frac{y_1}{y_3},\, \frac{y_2}{y_3}\right), \quad \text{and} \quad (y_1,\, y_2,\, y_3,\, y_4) \implies \left(\frac{y_1}{y_4},\, \frac{y_2}{y_4},\, \frac{y_3}{y_4}\right). \quad (8.5)$$

To denote when a point is described in homogeneous coordinates the superscript h will be used. For example, the point $P_C = (X_C, Y_C, Z_C)$ in camera frame coordinates can be expressed by:

$$P_C^h = (X_C, Y_C, Z_C, 1),$$

by choosing $\alpha = 1$, and the point $p = (u, v)$ in pixel coordinates can be expressed in homogeneous coordinates by:

$$p^h = (Z_C u,\, Z_C v,\, Z_C) = (\alpha X_C + u_0 Z_C,\, \beta Y_C + v_0 Z_C),$$

by choosing $\alpha = Z_C$ and substituting the expressions (8.3). With the expression of these points in homogeneous coordinates it can be seen that their relationship is transformed from the nonlinear relationship (8.3) to the *linear* relationship:

$$\begin{bmatrix} \alpha & 0 & u_0 & 0 \\ 0 & \beta & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha X_c + u_0 Z_c \\ \beta Y_c + v_0 Z_c \\ Z_c \end{pmatrix}. \quad (8.6)$$

Often in practice a skewness parameter γ is also added (which generally ends up being close to 0), and this relationship can be written in the more compact form:

$$\begin{bmatrix} K & 0_{3\times 1} \end{bmatrix} P_C^h = p^h, \quad K = \begin{bmatrix} \alpha & \gamma & u_0 \\ 0 & \beta & v_0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{8.7}$$

The matrix K defined in (8.7) is sometimes referred to as the *camera matrix* or *matrix of intrinsic parameters*. It is referred to in this way because it contains the five parameters that define the fundamental characteristics of the camera (from the perspective of the pinhole camera model). While these parameters may be specified by the camera manufacturer, they are often extracted by performing a camera calibration.

8.1.2 *Mapping World Coordinates to Camera Coordinates ($P_W \rightarrow P_C$)*

Recall from Figure 8.1 that a point P in the scene can either be expressed in terms of camera frame coordinates P_C or world frame coordinates P_W. While the previous section discussed the use of the pinhole model to map P_C coordinates to pixel coordinates p, this section will discuss the mapping between the camera and world frame coordinates of the point P (see Figure 8.3).

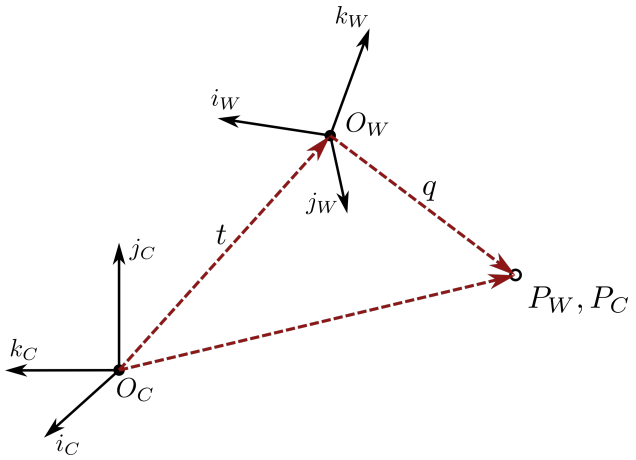


Figure 8.3: A depiction of the point P expressed either in camera coordinates, P_C, or in world frame coordinates, P_W. The world frame origin is denoted by O_W and the camera frame origin is denoted by O_C.

From Figure 8.3 it can be seen that P_C can be written as:

$$P_C = t + q, \tag{8.8}$$

where t is the vector from O_C to O_W expressed in camera frame coordinates and q is the vector from O_W to P expressed in camera frame coordinates. However, the vector q is in fact the same vector as P_W, just expressed in different coordinates (i.e. with respect to a different frame). The coordinates can be related by a rotation:

$$q = RP_W, \tag{8.9}$$

where R is the rotation matrix relating the camera frame to world frame and is defined as:

$$R = \begin{bmatrix} i_w \cdot i & j_w \cdot i & k_w \cdot i \\ i_w \cdot j & j_w \cdot j & k_w \cdot j \\ i_w \cdot k & j_w \cdot k & k_w \cdot k \end{bmatrix}, \tag{8.10}$$

where i, j, and k are the unit vectors that define the camera frame and i_w, j_w, and k_w are the unit vectors that define the world frame. To summarize, the point P_W can be mapped to camera frame coordinates P_C as:

$$P_C = t + RP_W, \tag{8.11}$$

where t is the vector in camera frame coordinates from O_C to O_W and R is the rotation matrix defined in (8.10). Similar to the previous section, these expressions can also be equivalently expressed for the case where the points P_W and P_C are expressed in homogeneous coordinates:

$$\begin{pmatrix} P_C \\ 1 \end{pmatrix} = \begin{bmatrix} R & t \\ 0_{1\times 3} & 1 \end{bmatrix} \begin{pmatrix} P_W \\ 1 \end{pmatrix}. \tag{8.12}$$

8.1.3 Mapping World Frame Coordinates to Image Coordinates ($P_W \rightarrow p$)

The original objective of perspective projection was to find a way to mathematically relate the position of a point P in world frame coordinates (denoted P_W) to the corresponding pixel coordinates p on the image plane. With the relationship (8.12) developed for mapping P_W to the camera frame coordinates P_C, and the relationship (8.7) for mapping P_C to pixel coordinates p, the direct mapping from P_W to p can now be defined. In particular, simply combining the two transformation together yields:

$$p^h = \begin{bmatrix} K & 0_{3\times 1} \end{bmatrix} \begin{bmatrix} R & t \\ 0_{1\times 3} & 1 \end{bmatrix} P_W^h,$$

which can then be simplified to:

$$p^h = K \begin{bmatrix} R & t \end{bmatrix} P_W^h. \tag{8.13}$$

In (8.13), P_W^h is the homogeneous coordinate representation of P_W and p^h is the homogeneous coordinate representation of p. Additionally, recall that the matrix $K \in \mathbb{R}^{3\times 3}$ is the matrix of intrinsic camera parameters, and the matrix $[R \quad t] \in \mathbb{R}^{3\times 4}$ contains *extrinsic* parameters (i.e. that describe the camera's position and orientation relative the points in the scene). Note that the total number of degrees of freedom is 11, where 5 are from the intrinsic parameters that define K, 3 are from the rotation matrix R, and 3 are from the position vector t.

8.2 Camera Calibration: Direct Linear Method

Before the expression (8.13) can be used in practice, the camera's intrinsic and extrinsic parameters need to be determined (i.e. K, R, and t). One approach is to use the direct linear transformation method for camera calibration, which requires a set of known correspondences $p_i \leftrightarrow P_{W,i}$ for $i = 1, \ldots, n$.

8.2.1 Direct Linear Calibration: Step 1

First, each corresponding pair of points $p_i = (u_i, v_i)$ and $P_{W,i} = (X_{W,i}, Y_{W,i}, Z_{W,i})$ is written in homogeneous coordinates and the expression (8.13) is used to write:

$$p_i^h = MP_{W,i}^h, \quad i = 1, ...n \tag{8.14}$$

where $M = K[R \;\; t]$ is referred to as the *homography*. The first step of the camera calibration process is to use the n correspondences to compute the homography M, and then later the intrinsic and extrinsic parameters can be extracted from M. To determine M, a useful first step is to rewrite M in terms of its rows:

$$M = \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix}, \tag{8.15}$$

where $m_i \in \mathbb{R}^{1\times 4}$ is the i-th row of M. By considering the rows of M individually, the relationship (8.14) can be written as:

$$\begin{bmatrix} \alpha u_i \\ \alpha v_i \\ \alpha \end{bmatrix} = \begin{bmatrix} m_1 \cdot P_{W,i}^h \\ m_2 \cdot P_{W,i}^h \\ m_3 \cdot P_{W,i}^h \end{bmatrix}, \quad i = 1, ...n$$

which by mapping the homogeneous coordinates p_i^h back to the original coordinates p_i yields the $2n$ expressions:

$$u_i = \frac{m_1 \cdot P_{W,i}^h}{m_3 \cdot P_{W,i}^h}, \quad i = 1, \ldots, n$$
$$v_i = \frac{m_2 \cdot P_{W,i}^h}{m_3 \cdot P_{W,i}^h}, \quad i = 1, \ldots, n,$$

or equivalently (via algebraic manipulation) the expressions:

$$\begin{aligned} u_i(m_3 \cdot P_{W,i}^h) - (m_1 \cdot P_{W,i}^h) = 0, \quad i = 1, \ldots, n \\ v_i(m_3 \cdot P_{W,i}^h) - (m_2 \cdot P_{W,i}^h) = 0, \quad i = 1, \ldots, n. \end{aligned} \tag{8.16}$$

Now, these $2n$ equations can be combined together in one large matrix equation:

$$\tilde{P}m = 0, \quad m = \begin{bmatrix} m_1^\top \\ m_2^\top \\ m_3^\top \end{bmatrix}, \tag{8.17}$$

where $m \in \mathbb{R}^{12\times 1}$ is a vector consisting of the stacked rows of M and $\tilde{P} \in \mathbb{R}^{2n\times 12}$ is a matrix of *known* coefficients determined by the quantities u_i, v_i, and $P^h_{W,i}$. For a more concrete representation of how $\tilde{P}$ is defined, the first couple rows are given by:

$$\tilde{P} = \begin{bmatrix} -(P^h_{W,1})^\top & 0_{1\times 4} & u_1(P^h_{W,1})^\top \\ 0_{1\times 4} & -(P^h_{W,1})^\top & v_1(P^h_{W,1})^\top \\ -(P^h_{W,2})^\top & 0_{1\times 4} & u_2(P^h_{W,2})^\top \\ \vdots & \vdots & \vdots \end{bmatrix}. \tag{8.18}$$

Note that $n \geq 6$ (i.e. at least 6 correspondences have been made) is a requirement to ensure that m can be uniquely defined. Ideally, with this sufficient number of correspondences the equation (8.18) could be directly solved. However, in practice a more robust procedure is to build $\tilde{P}$ with more than 6 points, which would lead to an overdetermined set of equations that may not have a solution[7]! Therefore, the determination of m is accomplished by formulation the optimization problem:

$$\begin{aligned} \min_m \ & \|\tilde{P}m\|^2, \\ \text{s.t. } & \|m\|^2 = 1, \end{aligned} \tag{8.19}$$

where the constraint $\|m\|^2 = 1$ is required to ensure that the optimization problem does not simply choose $m_i = 0$ for each $i = 1, \ldots, 12$. This optimization problem is called a *constrained least-squares* problem.

[7] This is particularly true in real-world applications where noise corrupts the data.

Example 8.2.1 (Constrained Least-Squares)**.** The constrained least squares problem

$$\begin{aligned} \min_x \ & \|Ax\|^2, \\ \text{s.t. } & \|x\|^2 = 1, \end{aligned}$$

with $x \in \mathbb{R}^n$ and $A \in \mathbb{R}^{m\times n}$ and $m > n$ is a finite-dimensional optimization problem. Consider the corresponding Lagrangian:

$$L = x^\top A^\top Ax + \lambda(1 - x^\top x),$$

and the necessary optimality conditions:

$$\begin{aligned} \nabla_x L &= 2(A^\top A - \lambda I)x = 0, \\ \nabla_\lambda L &= 1 - x^\top x = 0. \end{aligned}$$

The first NOC can be rewritten as $A^\top Ax = \lambda x$, and therefore any x that satisfies this condition must be an eigenvector of the matrix $A^\top A$. Additionally, while all the eigenvectors satisfy this condition the minimizer is the eigenvector associated with the smallest eigenvalue. This eigenvector can efficiently be computed by a singular value decomposition of $A = U\Sigma V^\top$ and then choosing m to be the column of V associated with the smallest singular value (since $A^\top A = V\Sigma^2 V^\top$).

8.2.2 Direct Linear Calibration: Step 2

Once the optimization problem (8.19) has been solved for m the homography M is completely defined. The next step in the camera calibration process is to extract the intrinsic and extrinsic camera parameters from the matrix M. For this section the matrix M is expressed in terms of its columns:

$$M = \begin{bmatrix} c_1 & c_2 & c_3 & c_4 \end{bmatrix},$$

where c_i is the i-th column of M. It is now possible to factorize M as:

$$M = K \begin{bmatrix} R & t \end{bmatrix}, \tag{8.20}$$

by taking the first three columns of M and performing a *RQ factorization*:

$$\begin{bmatrix} c_1 & c_2 & c_3 \end{bmatrix} = KR, \tag{8.21}$$

where R is an orthogonal matrix and K is an upper triangular matrix. Once K is known the vector t can be computed by $t = K^{-1}c_4$.

8.2.3 A Flexible Camera Calibration Method (Zhang, 2000):

The homography M is defined for a *specific* set of extrinsic parameters R and t. In practice it might be desirable to estimate the camera's intrinsic parameters from N *different* images from different perspectives (and therefore with N different homographies). In this case the procedure described in [8] can be used to extract the intrinsic parameters K.

[8] Z. Zhang. "A Flexible New Technique for Camera Calibration". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22 (2000)

This approach begins by assuming that the known points P_W for each individual image lie on a plane. For example the calibration "scene" might consist of a pattern (e.g. a checkerboard pattern) on a planar surface. In this case, it can simply be assumed that the world frame origin also lies on this plane such that $Z_W = 0$ for all points on the plane. Since $Z_W = 0$ the relationship between p^h and P_W^h given by (8.13) can be simplified to:

$$p^h = \tilde{M}\tilde{P}_W^h, \tag{8.22}$$

with

$$\tilde{M} = K \begin{bmatrix} r_1 & r_2 & t \end{bmatrix}, \quad \tilde{P}_W^h = \begin{bmatrix} X_W & Y_W & 1 \end{bmatrix}^\top, \tag{8.23}$$

where $\tilde{M}$ is the simplified homography matrix, $\tilde{P}_W^h$ is the simplified position of the point P in world frame written in homogeneous coordinates, and r_i is the i-th column of the rotation matrix R. Note that the homography matrix $\tilde{M}$ can still be estimated using the same procedure discussed before.

A set of constraints on the intrinsic parameter matrix K are next identified by writing the homography $\tilde{M}$ as:

$$\begin{bmatrix} \tilde{c}_1 & \tilde{c}_2 & \tilde{c}_3 \end{bmatrix} = \begin{bmatrix} Kr_1 & Kr_2 & Kt \end{bmatrix}.$$

This relationship, and the knowledge that r_1 and r_2 are orthonormal, leads to the following constraints:

$$\tilde{c}_1^\top B \tilde{c}_2 = 0, \quad \tilde{c}_1^\top B \tilde{c}_1 = \tilde{c}_2^\top B \tilde{c}_2, \tag{8.24}$$

where $B = K^{-\top} K^{-1} \in \mathbb{R}^{3\times 3}$ is a *symmetric* matrix. Solving for the intrinsic camera parameters K can therefore be accomplished by using the constraints (8.24) to solve for the symmetric matrix B, and then to use the definition of B to back out the parameters that define K.

Several useful tricks can be employed to compute the matrix B from the constraints (8.24). The main trick is to notice that even though B consists of nine parameters, since it is symmetric only six parameters are required to fully specify it. Therefore $B \in \mathbb{R}^{3\times 3}$ is reparameterized as a vector $b \in \mathbb{R}^6$ as:

$$b = \begin{bmatrix} B_{11} & B_{12} & B_{22} & B_{13} & B_{23} & B_{33} \end{bmatrix}^\top. \tag{8.25}$$

This reparameterization is useful because it allows us to rewrite the expression $\tilde{c}_i^\top B \tilde{c}_j$ as:

$$\tilde{c}_i^\top B \tilde{c}_j = v_{ij}^\top b, \tag{8.26}$$

where:

$$v_{ij} = \begin{bmatrix} \tilde{c}_{i1}\tilde{c}_{j1}, & \tilde{c}_{i1}\tilde{c}_{j2} + \tilde{c}_{i2}\tilde{c}_{j1}, & \tilde{c}_{i2}\tilde{c}_{j2}, & \tilde{c}_{i3}\tilde{c}_{j1} + \tilde{c}_{i1}\tilde{c}_{j3}, & \tilde{c}_{i3}\tilde{c}_{j2} + \tilde{c}_{i2}\tilde{c}_{j3}, & \tilde{c}_{i3}\tilde{c}_{j3} \end{bmatrix}^\top,$$

where $\tilde{c}_{ik}$ is the k-th element of the column vector $\tilde{c}_i$ and $\tilde{c}_{jk}$ is the k-th element of the column vector $\tilde{c}_j$. With this reparameterization, the constraints (8.24) can be rewritten as:

$$\begin{aligned} \tilde{c}_1^\top B \tilde{c}_2 = 0 &\implies v_{12}^\top b = 0 \\ \tilde{c}_1^\top B \tilde{c}_1 = \tilde{c}_2^\top B \tilde{c}_2 &\implies (v_{11} - v_{22})^\top b = 0, \end{aligned}$$

or by combining them:

$$\begin{bmatrix} v_{12}^\top \\ (v_{11} - v_{22})^\top \end{bmatrix} b = 0, \tag{8.27}$$

which is linear in the unknowns b. Importantly, while the homographies M are different for each image, the intrinsic camera parameters (i.e. the vector b) are the same! Therefore for N images from the same camera (but with potentially different perspectives) these constraints (8.27) can be stacked to give:

$$Vb = 0, \tag{8.28}$$

where $V \in \mathbb{R}^{2N\times 6}$. In the case where the skewness parameter γ is included in K there must be $N \geq 3$ images in order to specify B uniquely. Similar to how the homography for an image M was computed in the previous section, the vector b will be specified by the solution to the constrained least squares problem:

$$\begin{aligned} &\min_b \ \|Vb\|^2, \\ &\text{s.t. } \|b\|^2 = 1. \end{aligned} \tag{8.29}$$

Once b has been determined, the intrinsic camera parameters K can be solved for recalling the definition of $B = K^{-T}K^{-1}$. In particular, the intrinsic parameters are given by:

$$\begin{aligned} v_0 &= \frac{B_{12}B_{13} - B_{11}B_{23}}{B_{11}B_{22} - B_{12}^2}, \\ \lambda &= B_{33} - \frac{B_{13}^2 + v_0(B_{12}B_{13} - B_{11}B_{23})}{B_{11}}, \\ \alpha &= \sqrt{\frac{\lambda}{B_{11}}}, \\ \beta &= \sqrt{\frac{\lambda B_{11}}{B_{11}B_{22} - B_{12}^2}}, \\ \gamma &= \frac{-B_{12}\alpha^2\beta}{\lambda}, \\ u_0 &= \frac{\gamma v_0}{\beta} - \frac{B_{13}\alpha^2}{\lambda}, \end{aligned} \tag{8.30}$$

where λ can be though of as a scaling parameter that accounts for the fact that there are five unknown camera intrinsic parameters but six degrees of freedom in B.

Once the camera intrinsic parameters K have been extracted from this procedure, given any new homography $\tilde{M}$ the extrinsic parameters can be computed by:

$$\begin{aligned} r_1 &= \frac{K^{-1}\tilde{c}_1}{\|K^{-1}\tilde{c}_1\|}, \\ r_2 &= \frac{K^{-1}\tilde{c}_2}{\|K^{-1}\tilde{c}_2\|}, \\ r_3 &= r_1 \times r_2, \\ t &= \frac{K^{-1}\tilde{c}_3}{\|K^{-1}\tilde{c}_1\|}. \end{aligned} \tag{8.31}$$

As one final step, it is noted that the matrix R defined with columns r_1, r_2, and r_3 will not in generally satisfy the properties of a rotation matrix (i.e. orthonormality). One final step to this overall procedure is to correct this issue by finding the rotation matrix that best corresponds to these column vectors. This is accomplished again by optimization, and in particular by formulating the problem:

$$\begin{aligned} &\min_R \ \|R - Q\|^2, \\ &\text{s.t. } R^\top R = I, \end{aligned} \tag{8.32}$$

where

$$Q = \begin{bmatrix} r_1 & r_2 & r_3 \end{bmatrix}.$$

This problem is solved by choosing $R = UV^\top$ where U and V are defined by the singular value decomposition of $Q = U\Sigma V^\top$.

8.3 Limitations

8.3.1 Radial Distortion

The pinhole camera model provides a nominal camera model for which it is relatively straightforward to develop a mathematical model of the perspective projection. However, in practice this model is not a perfect representation of the imaging process. One such effect that is not captured by the pinhole model is *radial distortion*, which is an effect seen in real lenses where either barrel distortion or pincushion distortion will affect the real pixel coordinates. Images showing both barrel and pincushion distortion are provided in Figure 8.4.

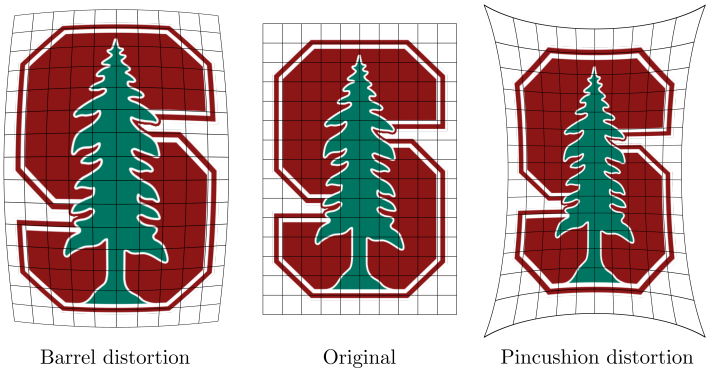


Figure 8.4: Different kinds of radial distortions that are seen in real lenses, which may affect the accuracy of the pinhole camera model.

There are methods that can be used to correct for image distortion. A simple and efficient way is to model the relationship between the ideal pixel coordinates (u, v) and the distorted pixel coordinates (u_d, v_d) as:

$$\begin{bmatrix} u_d \\ v_d \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix} (1 + kr^2) \begin{bmatrix} u - u_{cd} \\ v - v_{cd} \end{bmatrix} + \begin{bmatrix} u_{cd} \\ v_{cd} \end{bmatrix} \tag{8.33}$$

where $k \in \mathbb{R}$ is the radial distortion factor, (u_{cd}, v_{cd}) are the pixel coordinates of the image center, and $r^2 = (u - u_{cd})^2 + (v - v_{cd})^2$ is the square of the distance between the ideal pixel location and the center of distortion.. Note that k differs in different cameras and needs to be pre-determined.

8.3.2 Measuring Depth

Once the camera intrinsic and extrinsic parameters K, R, and t are known it is still not possible to map pixel coordinates to the corresponding point in space. Mathematically this is a result of the matrix M in (8.14) not being invertible, but intuitively this is because the distance along the line of sight from p to P in Figure 8.1 cannot be determined!

However, there are some techniques that can enable depth estimates to be made with a single camera. One approach is known as *depth from focus*, where several images are taken until the projection of point P is in focus. Based on the

thin lens model, when this occurs:

$$\frac{1}{z} + \frac{1}{Z} = \frac{1}{f},$$

where f is the focal length, Z is the depth of the point P in camera frame, and z is the depth of the image plane in the camera frame when the projection of point P is in focus. Since f and z are known, the depth Z can therefore be computed. If two cameras are used, depth estimation is possible via *binocular reconstruction* or *stereo vision*. This approach requires known corresponding pixel coordinates p and p' of each camera, and then uses *triangulation* to determine the 3D position of the source point P in the scene.

8.4 Exercises

8.4.1 Camera Calibration

Complete *Problem 1: Camera Calibration* located in the online repository:

```
https://github.com/PrinciplesofRobotAutonomy/AA274A_HW3,
```

where you will estimate the intrinsic parameters of a camera using the method described in Section 8.2.3.

9
Stereo Vision and Structure From Motion

The previous chapter developed a mathematical relationship between the position of a point P in a scene (expressed in world frame coordinates P_W), and the corresponding point p in pixel coordinates that gets projected onto the image plane of the camera. This relationship was derived based on the pinhole camera model, and required knowledge about the camera's intrinsic and extrinsic parameters. Nonetheless, even in the case where all of these camera parameters are known it is still impossible to reconstruct the depth of P with a single image (without additional information). However, in the context of robotics, recovering 3D information about the structure of the robot's environment through computer vision is often a very important task (e.g. for obstacle avoidance). Two approaches for using cameras to gather 3D information are therefore presented in this chapter, namely *stereo vision* and *structure from motion*[1,2].

[1] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, 2011

[2] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2011

Stereo Vision and Structure From Motion

Recovering scene structure from images is extremely important for mobile robots to safely operate in their environment and successfully perform tasks. While a number of other sensors can also be used to recover 3D scene information, such as ultrasonic sensors or laser rangefinders, cameras capture a broad range of information that goes beyond depth sensing. Additionally, cameras are a well developed technology and can be an attractive option for robotics based on cost or size.

Unfortunately, unlike sensors that are specifically designed to measure depth like laser rangefinders, the camera's projection of 3D data onto a 2D image makes it impossible to gather some information from a single image[3]. Techniques for extracting 3D scene information from 2D images have therefore been developed that leverage *multiple* images of a scene. Examples of such techniques include *depth-from-focus* (uses images with different focuses), *stereo vision* (uses images from different viewpoints), or *structure from motion* (uses images captured by a moving camera).

[3] Unless you are willing to make some strong assumptions, for example that you know the physical dimensions of the objects in the environment.

9.1 Stereo Vision

Stereopsis (from *stereo* meaning solidity, and *opsis* meaning vision or sight) is the process in visual perception leading to the sensation of depth from two slightly different projections of the world onto the retinas of the two eyes. The difference in the two retinal images is called horizontal *disparity*, retinal disparity, or binocular disparity, and arise from the eyes' different positions in the head. It is the disparity that makes our brain fuse (perceive as a single image) the two retinal images, making us perceive the object as one solid object. For example, if you hold your finger vertically in front of you and alternate closing each eye you will see that the finger jumps from left to right. The distance between the left and right appearance of the finger is the disparity.

Computational stereopsis, or *stereo vision*, is the process of obtaining depth information of a 3D scene via images from two cameras which look at the same scene from different perspectives. This process consists of two major steps: fusion and reconstruction. Fusion is a problem of correspondence, in other words how do you correlate each point in the 3D environment to their corresponding pixels in *each* camera. Reconstruction is then a problem of *triangulation*, which uses the pixel correspondences to determine the full position of the source point in the scene (including depth).

9.1.1 Epipolar Constraints

As previously mentioned, the first step in the stereo vision process is to fuse the two (or more) images and generate point correspondences[4]. This task can be quite challenging, and erroneously matching features can lead to large errors in the reconstruction step. Therefore, several techniques are leveraged to make this task simpler. The most important simplifying technique is to impose an *epipolar constraint*.

[4] This generally assumes that the perspective of each image is only a slight variation from the other, such that the features appear similarly in each.

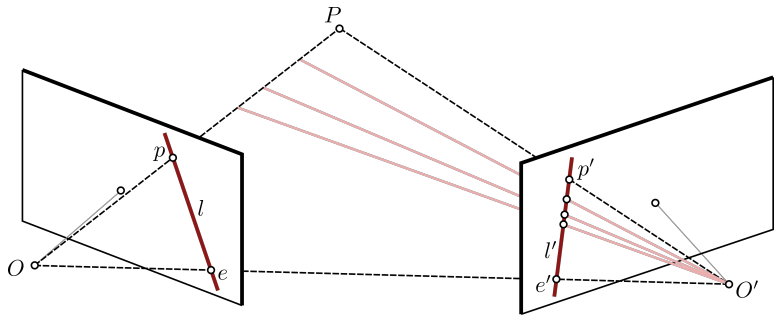


Figure 9.1: The point P in the scene, the optical centers O and O' of the two cameras, and the two images p and p' of P all lie in the same plane, referred to as the epipolar plane. The lines l and l' are the epipolar lines of the points p and p', respectively. Note that if the point p is observed in one image, the corresponding point in the second image must lie on the epipolar line l'!

Consider the images p and p' of a point P observed by two cameras with optical centers O and O' (see Figure 9.1). These five points all belong to the *epipolar plane* defined by the two intersecting rays OP and $O'P$. In particular, the point p lies on the line l where the epipolar plane and the image plane intersect. The line l is referred to as the *epipolar line* associated with the point p, and it

passes through the point e (referred to as the *epipole*). Based on this geometry, if p and p' are images of the same point P, then p must lie on the epipolar line l and p' must lie on the epipolar line l'.

Therefore, when searching for correspondences between p and p' for a particular point P in the scene it makes sense to restrict the search to the corresponding epipolar line. This is referred to as an *epipolar constraint*, and greatly simplifies the correspondence problem by restricting the possible candidate points to a line rather than the entire image (i.e. a one dimensional search rather than a two dimensional search). Mathematically, the epipolar constraints can be written as:

$$\overline{Op} \cdot [\overline{OO'} \times \overline{O'p'}] = 0, \tag{9.1}$$

since $\overline{Op}$, $\overline{O'p'}$, and $\overline{OO'}$ are coplanar. Assuming the world reference frame is co-located with camera 1 (with an origin at point O) this constraint can be written as:

$$p^\top F p' = 0, \tag{9.2}$$

where F, referred to as the *fundamental matrix*, has seven degrees of freedom and is singular. For a derivation of the epipolar constraint see Section 7.1 from Forsyth et al.[5]. Additionally, the matrix F is only dependent on the intrinsic camera parameters for each camera and the geometry that defines their relative positioning, and can be assumed to be constant. The expression for the fundamental matrix in terms of the camera intrinsic parameters is:

$$F = K^{-\top} E K'^{-1}, \quad E = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix} R, \tag{9.3}$$

where K and K' are the intrinsic parameter matrices for cameras 1 and 2 respectively, and R and $t = [t_1, t_2, t_3]^\top$ are the rotation matrix and translation vector that map camera 2 frame coordinates into camera 1 frame coordinates. Note that with the epipolar constraint defined by the fundamental matrix (9.2), the epipolar lines l and l' can be expressed by $l = Fp'$ and $l' = F^\top p$. Additionally, it can be shown that $F^\top e = Fe' = 0$ where e and e' are the epipoles in the image frames of cameras 1 and 2, since by definition the translation vector t is parallel to the coordinate vectors of the epipoles in the camera frames. This in turn guarantees that the fundamental matrix F is singular.

If the parameters K, K', R, and t are not already known, the fundamental matrix F can be determined in a manner similar to the intrinsic parameter matrix K in the previous chapter. Suppose a number of corresponding points $p^h = [u, v, 1]^\top$ and $(p^h)' = [u', v', 1]^\top$ are known and are expressed as homogeneous coordinates. Each pair of points has to satisfy the epipolar constraint (9.2), which can be written as:

$$\begin{bmatrix} u & v & 1 \end{bmatrix} \begin{bmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{bmatrix} \begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = 0$$

[5] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2011

This expression can then be equivalently expressed by reparameterizing the matrix F in vector form f as:

$$\begin{bmatrix} uu' & uv' & u & vu' & vv' & v & u' & v' & 1 \end{bmatrix} f = 0 \tag{9.4}$$

where $f = [F_{11}, F_{12}, F_{13}, F_{21}, F_{22}, F_{23}, F_{31}, F_{32}, F_{33}]^\top$. For n known correspondences (p, p') these constraints can be stacked to give:

$$Wf = 0, \tag{9.5}$$

where $W \in \mathbb{R}^{n \times 9}$. Given $n \geq 8$ correspondences, an estimate $\tilde{F}$ of the fundamental matrix estimate is given by:

$$\begin{aligned} &\min_f \ \|Wf\|^2, \\ &\text{s.t. } \|f\|^2 = 1. \end{aligned} \tag{9.6}$$

Note that the estimate $\tilde{F}$ computed by (9.6) is not guaranteed to be singular. A second step is therefore taken to enforce this additional condition. In particular it is desirable to find the matrix F that is closest to the estimate $\tilde{F}$ that has a rank of two:

$$\begin{aligned} &\min_F \ \|F - \tilde{F}\|^2, \\ &\text{s.t. } \det(F) = 0, \end{aligned} \tag{9.7}$$

which can be accomplished by computing a singular value decomposition of the matrix $\tilde{F}$.

9.1.2 *Image Rectification*

Given a pair of stereo images, epipolar rectification is a transformation of each image plane such that all corresponding epipolar lines become colinear and parallel to one of the image axes, for convenience usually the horizontal axis. The resulting rectified images can be thought of as acquired by a new stereo camera obtained by rotating the original cameras about their optical centers. The great advantage of the epipolar rectification is the correspondence search becomes simpler and computationally less expensive because the search is done along the horizontal lines of the rectified images. The steps of the epipolar rectification algorithm are illustrated in Figure 9.2. Observe that after the rectification, all the epipolar lines in the left and right image are colinear and horizontal. For an in-depth discussion on algorithms for image rectification see [6,7].

[6] A. Fusiello, E. Trucco, and A. Verri. "A compact algorithm for rectification of stereo pairs". In: *Machine Vision and Applications* 12.1 (2000), pp. 16–22

[7] C. Loop and Z. Zhang. "Computing rectifying homographies for stereo vision". In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Vol. 1. 1999, pp. 125–131

9.1.3 *Correspondence Problem*

Epipolar constraints and image rectification are commonly used in stereo vision to address the problem of correspondence, which is the problem of determining the pixels p and p' from two different cameras with different perspectives

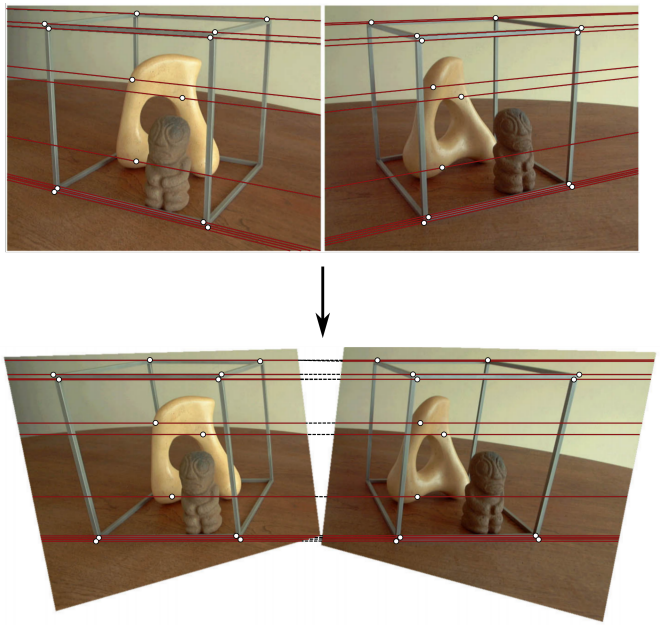


Figure 9.2: Epipolar rectification example from Loop et al. (1999).

that correspond to the same scene feature P. While these concepts make finding correspondences easier, there are still several challenges that must be overcome. These include challenges related to feature occlusions, repetitive patterns, distortions, and others.

9.1.4 *Reconstruction Problem*

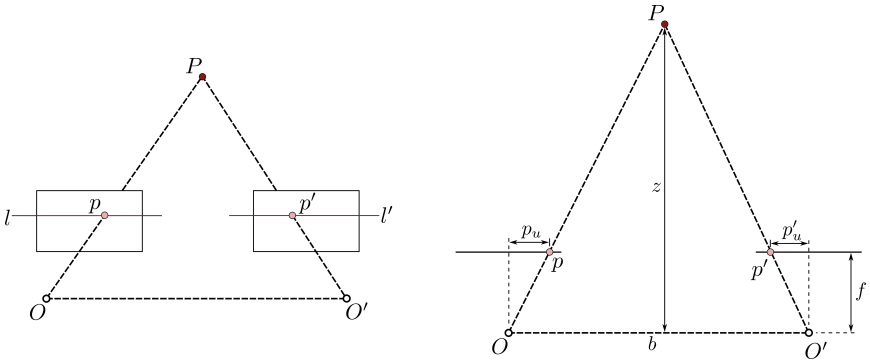


Figure 9.3: Triangulation with rectified images (horizontal view on the left, top-down view on the right).

In a stereo vision setup, once a correspondence between the two images is identified it is possible to reconstruct the 3D scene point based on *triangulation*. This process of triangulation has already been covered by the discussion on the epipolar geometry. However if the images have also be rectified such that the epipolar lines become parallel to the horizontal image axis the triangulation problem becomes simpler. This occurs, for example, when the two cameras have the same orientation, are placed with their optical axes parallel, and are

separated by some distance b called the *baseline* (see Figure 9.3).

In Figure 9.3, a point P on the object is described as being at coordinate (x, y, z) with respect to the origin located in the left camera at point O. The horizontal pixel coordinate in the left and right image are denoted by p_u and p'_u respectively. Based on the geometry the depth of the point P can be computed from the properties of similar triangles:

$$\frac{z}{b} = \frac{z - f}{b - p_u + p'_u}, \tag{9.8}$$

which can be algebraically simplified to:

$$z = \frac{bf}{p_u - p'_u}, \tag{9.9}$$

where f is the focal length. Generally a small baseline b will lead to larger depth errors, but a large baseline b may cause features to be visible from one camera but not the other. The difference in the image coordinates, $p_u - p'_u$, is referred to as *disparity*. This is an important term in stereo vision, because it is only by measuring disparity that depth information can be recovered. The disparity can also be visually represented in a *disparity map* (for example see Figure 9.4), which is simply a map of the disparity values for each pixel in an image. The largest disparities occur from nearby objects (i.e. since disparity is inversely proportional to z).

Figure 9.4: Disparity map from a pair of stereo images. Notice that the lighter values of the disparity map represent larger disparity, and correspond to the point in the scene that are closer to the cameras. The black points represent points that were occluded from one of the images and therefore no correspondence could be made. Images from Scharstein et al. (2003) .

9.2 *Structure From Motion (SFM)*

The structure from motion (SFM) method uses a similar principle as stereo vision, but uses *one* camera to capture multiple images from different perspectives while moving within the scene. In this case, the intrinsic camera parameter matrix K will be constant, but the extrinsic parameters (i.e. the rotation matrix R and relative position vector t) will be different for each image. Consider a case where m images of n fixed 3D points are taken from different perspectives. This would involve m homography matrices M_k and n 3D points P_j that would need to be determined by leveraging the relationships:

$$p^h_{j,k} = M_k P^h_j, \quad j = 1, \ldots, n, \quad k = 1, \ldots, m.$$

However, SFM also has some unique disadvantages, such as an ambiguity in the absolute scale of the scene that cannot be determined. For example a bigger

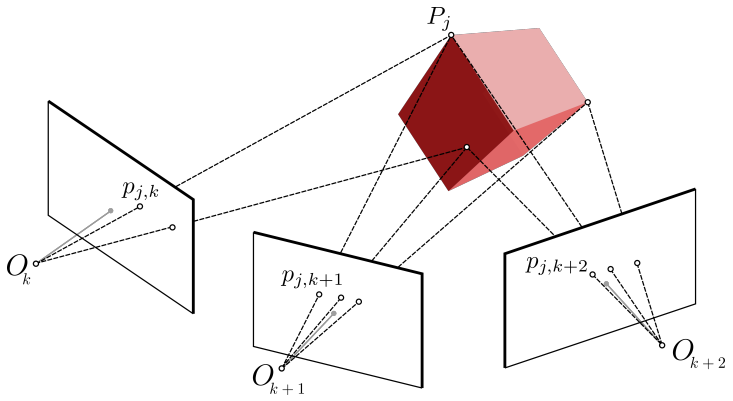


Figure 9.5: A depiction of the structure from motion (SFM) method. A single camera is used to take multiple images from different perspectives, which provides enough information to reconstruct the 3D scene.

object at a longer distance and a smaller object at a closer distance may yield the same projections.

One application of the SFM concept is known as *visual odometry*. Visual odometry estimates the motion of a robot by using visual inputs (and possible additional information). This approach is commonly used in practice, for example by rovers on Mars, and is useful because it not only allows for 3D scene reconstruction but also to recover the motion of the camera.

10
Image Processing

The previous chapters focused on using camera models to identify the relationship between points in a 3D scene and their projections onto the camera image, as well as how to leverage those models to reconstruct 3D scene structure from 2D images. Alternatively, this chapter begins to look at methods for extracting other types of information through *image processing*, for example to answer the question "what object am I seeing?" rather than "how far away is this object?". Extracting this type of visual content from raw images is important for mobile robots to be able to intelligently interpret their surroundings. In fact, it can have a major impact on the ability of the robot to perform several tasks including localization and mapping or decision making. This chapter focuses on some of the more commonly used tools in image processing including image filtering, feature detection, and feature description[1,2].

[1] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, 2011

[2] H. P. Moravec. "Towards automatic visual obstacle avoidance". In: *5th International Joint Conference on Artificial Intelligence*. 1977

Image Processing

Image processing is a form of signal processing where the input signal is an image (such as a photo or a video) and the output is either an image or a set of parameters associated with the image. While a large number of image processing techniques exist, this chapter focuses on some of the more fundamental methods that are relevant for robotics. In particular, these methods will be related to image filtering, feature detection, and feature description[3].

[3] The software library OpenCV implements a number of useful image filtering algorithms: `https://docs.opencv.org`.

In the following methods, grayscale images are treated as functions $I\colon [a,b] \times [c,d] \to [0,L]$, where $I(x,y)$ represents the grayscale pixel intensity at (x,y). For a color image, I is a vector valued function with three components, one each for the red, green, and blue color channels of the image.

10.1 Image Filtering

Image filtering is one of the principal tasks in image processing. The terminology "filter" comes from frequency domain signal processing and refers to the process of accepting or rejecting certain frequency components of a signal (e.g. eliminating high-frequency noise).

Perhaps the most common type of image filtering is *spatial filtering*. The basic principle of spatial filtering is that a particular pixel is modified in the filtered image based only on the pixels in the immediate spatial neighborhood (see Figure 10.1). To be more specific, a spatial filter for an image $I(x, y)$ consists of:

1. A neighborhood S_{xy} of pixels around a particular point (x, y) under examination, typically rectangular.
2. A predefined operation F that is performed on the image pixels encompassed by the neighborhood S_{xy}.

Once the operation F has been applied to all pixels (x, y) in the image I a new image $I'(x, y)$ is defined.

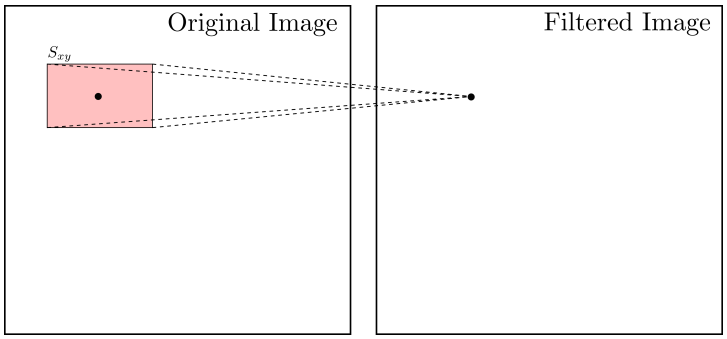


Figure 10.1: Illustration of the concept of spatial filtering. The spatial filter operates on a neighborhood S_{xy} of each point in the original image to produce a new pixel in the filtered image.

In general filters can be linear or nonlinear, but many of the most fundamental filters are linear and can be expressed mathematically as:

$$I'(x, y) = F \circ I = \sum_{i=-N}^{N} \sum_{j=-M}^{M} F(i, j) I(x + i, y + j), \tag{10.1}$$

where N and M are integers that define the width and height of a rectangular neighborhood S_{xy}. Based on the size of this neighborhood, it is said that this filter is of size $(2N + 1) \times (2M + 1)$. Additionally, the filter operation F is usually called a *mask* or *kernel*. Broadly speaking, filters expressed by (10.1) are referred to as *correlation filters*.

Another type of linear filters that are commonly used are referred to as *convolution* filters. Convolution filters are similar to correlation filters but use reverse image indices (in fact correlation and convolution filters are identical when the filter mask is symmetric in both the horizontal and vertical directions). In particular, these filters are expressed mathematically as:

$$I'(x, y) = F * I = \sum_{i=-N}^{N} \sum_{j=-M}^{M} F(i, j) I(x - i, y - j). \tag{10.2}$$

Convolution filters are associative, meaning that for two different filter masks F and G it is true that $F * (G * I) = (F * G) * I$. One example of how the associative property is useful is for smoothing an image *before* applying a differentiation

filter. Suppose the mask F implemented a derivative filter and G implemented a smoothing filter, then sequentially applying these filters would result in $F * (G * I)$. However, because of the associative property the masks can be convolved together *first* such that only one filter needs to be applied to the image (i.e. $(F * G) * I$).

Note that in both the correlation and convolution filters the boundaries of the image need some special care because of the width and height of the mask. For example, Figure 10.2 shows how the filtered image is smaller than the original due to the width and height of the mask. Some possible options to handle this include padding the image, cropping it, extending it, or wrapping it. However, as images are generally quite large the exact approach likely won't vary the final result significantly.

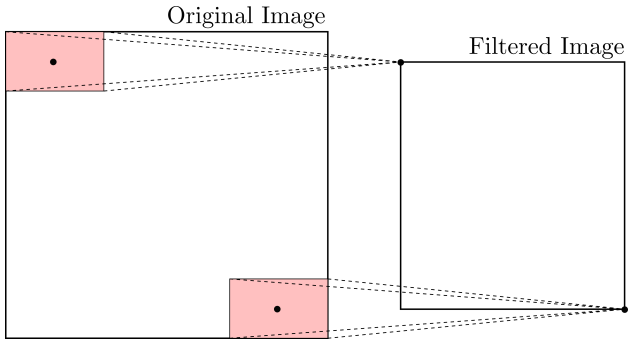


Figure 10.2: Due to the width and height of the mask, the filtered image may be smaller than the original. However this can be fixed with several techniques, such as padding.

Example 10.1.1 (Practical Considerations for Image Filtering)**.** Implementation of correlation and convolution filters typically leverages some additional "tricks" to make things easier to implement. In this example two such tricks will be introduced: zero-padding and a change in indexing.

First, to more simply accommodate varying sizes of filters (including even and odd sized filters) the indexing is often changed such that the coordinate of interest is associated with the top-left element in the window rather than the center. In particular, for a correlation filter this would correspond to:

$$I'(x,y) = F \circ I = \sum_{i=1}^{K} \sum_{j=1}^{L} F(x,y) I(x+i-1, y+j-1), \tag{10.3}$$

where K and L are integers that define the width and height of the filter and the pixel (x,y) is at row x and column y. However, note that with this formulation the output image I' will be shifted up and to the left. To see this consider the pixel at $x = 1$ and $y = 1$ in the new image I', which would correspond to the top-left pixel I'. This new pixel value is generated by applying the filter F over the pixels in the original image I at rows $\{1, \ldots, K\}$ and columns $\{1, \ldots, L\}$ (which is not centered at $(1,1)$ in the original image I). Therefore it will appear as if the image has been shifted! But in practice this isn't an issue as long as you always index with respect to the top-left corner. An example of top-left indexing is shown in Figure 10.3

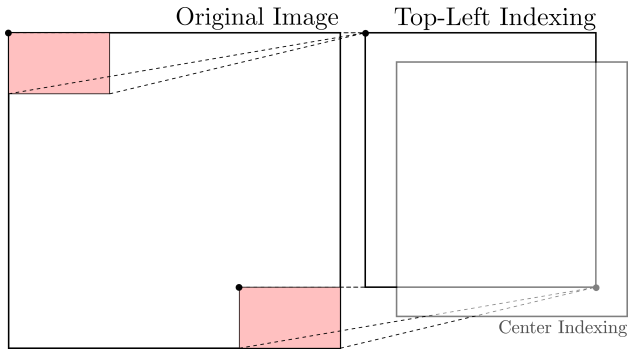


Figure 10.3: Top-left indexing is typically easier to implement than center indexing. Notice that when top-left indexing, it appears as if the filtered image has shifted with respect to when center indexing is used.

Zero-padding (also commonly referred to as *same padding*) is another simple trick that can be used to ensure that the output filtered image I' has the same dimension as the input image I. In this approach the left and right boundaries of the image are *each* padded by $\lfloor K/2 \rfloor$ columns of zeros, and the top and bottom boundaries are padded by $\lfloor L/2 \rfloor$ rows of zeros ($\lfloor \cdot \rfloor$ denotes the "floor" operation). For example the image:

$$I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix},$$

would become

$$I_{\text{padded}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ 0 & 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

for filters $F \in \mathbb{R}^{3\times3}$, $F \in \mathbb{R}^{2\times2}$, $F \in \mathbb{R}^{2\times3}$ and $F \in \mathbb{R}^{3\times2}$. When using this padding rule with the correlation filter (10.3) and a filter F with $K = 2,3$ and $L = 2,3$, the new image I' can be defined for values $x \in \{1,2,3\}$ and $y \in \{1,2,3\}$, resulting in I' being the same dimension as the original image I. The use of padding (along with top-left indexing) is also shown graphically in Figure 10.4

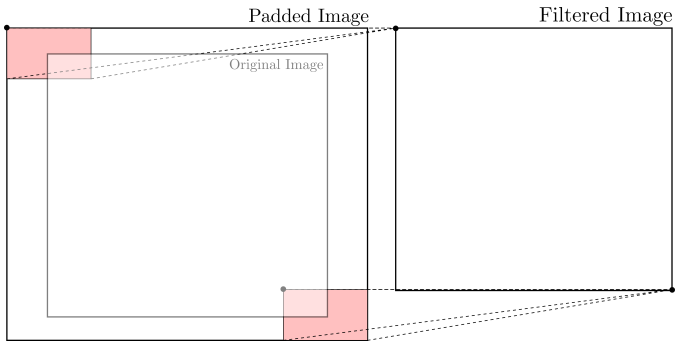


Figure 10.4: Image padding is a commonly used technique to ensure that the size of the filtered image is the same size as the original.

10.1.1 Moving Average Filter

The moving average filter returns the average of pixels in the mask, which achieves a smoothing effect (i.e. removes sharp features in the image). For example, a moving average filter with a normalized 3×3 mask is defined with the operation F in (10.1) chosen as:

$$F = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Note that due to symmetry of the mask, the correlation (10.1) and convolution (10.2) filters will be identical. Additionally, the normalization is used to maintain the overall brightness of the image.

10.1.2 Gaussian Smoothing Filter

Gaussian smoothing filters are similar to the moving average filer, but instead of weighting all of the pixels evenly they are weighted by the Gaussian function:

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(- \frac{x^2 + y^2}{2\sigma^2} \right).$$

This function is used to obtain the mask operation F by sampling the function about the center pixel (i.e. for the center pixel with $i = j = 0$ in (10.1), sample $G_\sigma(0, 0)$). For example, for a normalized 3×3 mask with $\sigma = 0.85$ this filter is approximately defined by:

$$F = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

Like the moving average filter, this filter mask is symmetric and therefore yields identical results with respect to the correlation (10.1) or convolution (10.2) filters. The advantage of the Gaussian filter is that it provides more weight to the neighboring pixels that are closer. An example of this filter is shown in Figure 10.5.

10.1.3 Separable Masks

A mask F is called *separable* if it can be broken down into the convolution of two kernels $F = F_1 * F_2$. If a mask is separable into "smaller" masks, then it is often cheaper to apply F_1 followed by F_2, rather than by F directly. One special case of this is when the mask can be represented as an outer product of two vectors (meaning it is equivalent to the 2D convolution of those two vectors). If the mask is of shape $M \times M$, and the input image has size $w \times h$, then the computational complexity of directly performing the convolution is $O(M^2wh)$. However, by separating the masks the computational cost is $O(2Mwh)$, which is

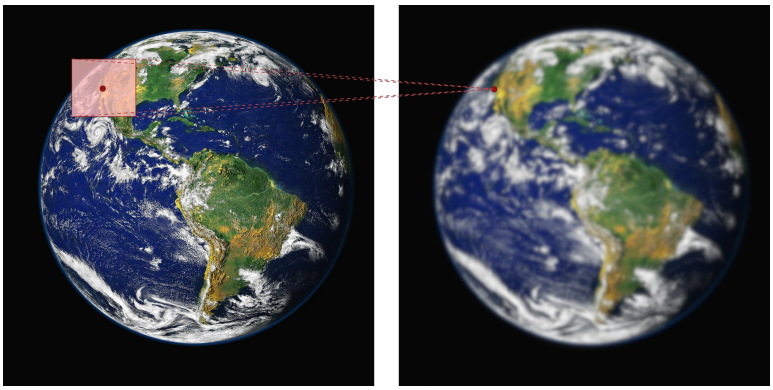


Figure 10.5: Example of a Gaussian smoothing filter, which produces a smoothing (blurring) effect on the filtered image.

linear in M rather than quadratic. As an example, consider the moving average filter mask from before:

$$F = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \frac{1}{9}\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}.$$

As another example, note that the Gaussian smoothing filter mask is also separable. To see why this is, note that the Gaussian weighting function can be decomposed as:

$$\begin{aligned} G_\sigma(x,y) &= \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right), \\ &= \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{x^2}{2\sigma^2}\right)\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{y^2}{2\sigma^2}\right), \\ &= g_\sigma(x)\cdot g_\sigma(y). \end{aligned}$$

10.1.4 Image Differentiation Filters

Taking the derivative of an image can be used to identify certain features, such as edges. On a basic level, the derivative of an image quantifies changes in pixel intensity in both the vertical and horizontal direction. However, since images are represented as functions defined over a discrete domain the traditional method for differentiating continuous functions cannot be used. Instead it is more common to just compute differences between pixels, such as using a central difference method:

$$\begin{aligned} \frac{\partial I}{\partial x} &= \frac{I(x+1,y)-I(x-1,y)}{2}, \\ \frac{\partial I}{\partial y} &= \frac{I(x,y+1)-I(x,y-1)}{2}. \end{aligned} \tag{10.4}$$

where $\partial I/\partial x$ is the derivative in the horizontal direction and $\partial I/\partial y$ is the derivative in the vertical direction. It is of course also possible to define the derivatives using just one side, for example $\frac{\partial I}{\partial x} = I(x+1,y) - I(x,y)$.

It is also possible to differentiate an image using convolution filters. In particular, one common approach is to use a convolution filter (10.2) defined with a mask F called a *Sobel mask* (also referred to as simply a Sobel operator). For the x direction this mask is denoted as S_x and for the y direction as S_y:

$$S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \quad S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{10.5}$$

Sobel masks are similar to the central difference method but use more neighboring pixels when calculating the derivative (i.e. they also consider the rows above and below to compute the difference). Note that Sobel masks are also separable.

10.1.5 Similarity Measures

Filtering can also be used to find similar features in different images, which can be useful for solving the correspondence problem in stereo vision or structure-from-motion techniques. In particular, the similarity between the pixel (x, y) in image I_1 and pixel (x', y') in image I_2 can be computed by:

$$\begin{aligned} SAD &= \sum_{i=-N}^{N} \sum_{j=-M}^{M} |I_1(x+i, y+j) - I_2(x'+i, y'+j)|, \\ SSD &= \sum_{i=-N}^{N} \sum_{j=-M}^{M} [I_1(x+i, y+j) - I_2(x'+i, y'+j)]^2, \end{aligned} \tag{10.6}$$

where SAD is an acronym for "sum of absolute differences", SSD is an acronym for "sum of squared differences", and N and M define the size of the window around the pixels that is considered.

10.2 Image Feature Detection

A local feature (also sometimes referred to as interest points, interest regions, or keypoints) in an image is a pattern that differs from its immediate neighborhood in terms of intensity, color, or texture. Local features can generally be categorized in several ways, for example whether they provide semantic content or not. For example, features that may provide semantic content include edges or other geometric shapes (e.g. lanes of a road or blobs corresponding to blood cells in medical images). These types of features were some of the first for which feature detectors were proposed in the image processing literature. Features that do not provide semantic content may also be useful, for example in feature tracking, camera calibration, 3D reconstruction, image mosaicing, and panorama stitching. In these cases it may be more important that the feature be able to be located accurately and robustly over time. A third category of features are those that may not have semantic interpretations individually, but may have meaning as a collection. For instance, a scene could be recognized

by counting the number of feature matches between the observed scene and a query image. In this case only the number of matches is relevant and not the location or type of feature. Applications where these types of features are important include texture analysis, scene classification, video mining, and image retrieval.

In this section several feature detection strategies will be discussed. While many strategies exist for different types of features, the focus here will be on two common features that are often useful in robotics: edges and corners.

10.2.1 Edge Detection

An *edge* in an image is a region where there is a significant change in intensity values along one direction, and negligible change along the orthogonal direction. In one dimension an edge corresponds to a point where there is a sharp change in intensity, which mathematically can be thought of as a point of a function having a large first derivative and a small second derivative. Many edge detectors rely on this concept by differentiating images and looking for spikes in the derivative. An edge detector can be evaluated based on several criteria for robustness and performance, including accuracy, localization, and single response. Good accuracy implies few false positives or negatives (missed edges), good localization implies that the detected edge should be exactly where the true edge is in the image, and a single response implies *only* one edge is detected for each real edge. In practice, noise and discretization can make edge detection challenging.

Most edge detection methods rely on two key steps: smoothing and differentiation. Differentiation is performed in both the vertical and horizontal directions to find locations in the image with high intensity gradients. However, differentiation alone is vulnerable to false positives due to image noise, which is why many algorithms will first smooth the image.

Edge Detection in 1D: An example of how noise can corrupt image differentiation is given in Figure 10.6. Notice that in this case it is impossible to identify the jump in the signal due to the noise levels. Smoothing filters, such as the Gaussian smoothing filter discussed earlier, can help remedy this problem. In particular, suppose the original signal in Figure 10.6 is defined by $I(x)$. Then a smoothed version can be defined by applying a smoothing convolution filter:

$$s(x) = g_\sigma(x) * I(x),$$

where $g_\sigma(x)$ represents a Gaussian smoothing filter, and then by applying the differentiation filter:

$$s'(x) = \frac{d}{dx} * s(x).$$

This process is shown in Figure 10.7. Note however that since these filters are convolutions, the associativity property can be leveraged to actually combine

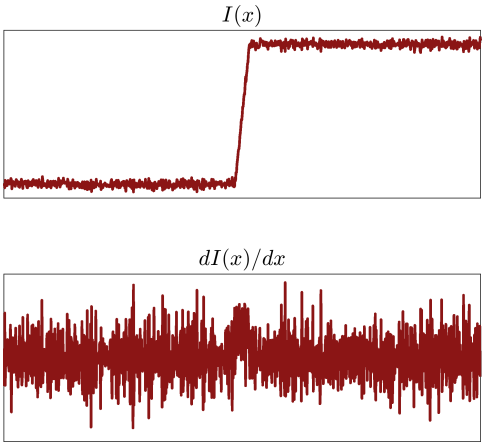


Figure 10.6: Differentiation of signal (e.g. for edge detection) with noise can be particularly challenging, which can be addressed by first smoothing the signal.

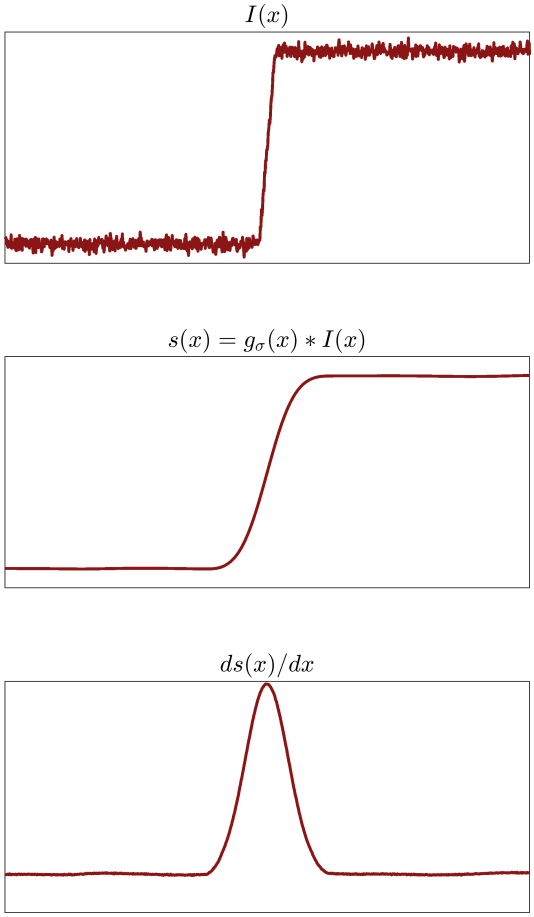


Figure 10.7: Edge detection through convolution with a Gaussian smoothing filter, followed by a differentiation filter.

them into a single filter:

$$s' = (\frac{d}{dx} * g_\sigma) * I.$$

Edge Detection in 2D: Edge detection in a two-dimensional image is quite similar to the example previously discussed in 1D. Let the smoothing filter be the Gaussian smoothing filter from before, and a differentiation filter such as the Sobel filter. The gradient of the smoothed image in both the x and y directions can be written as:

$$\nabla S = \begin{bmatrix} \frac{\partial}{\partial x} * G_\sigma * I \\ \frac{\partial}{\partial y} * G_\sigma * I \end{bmatrix} = \begin{bmatrix} G_{\sigma,x} * I \\ G_{\sigma,y} * I \end{bmatrix} = \begin{bmatrix} S_x \\ S_y \end{bmatrix},$$

where I is the original image and the associativity properties of the smoothing and differentiation convolution filters is used to define the combined filters $G_{\sigma,x}$ and $G_{\sigma,y}$. The magnitude of the gradient can then be computed by:

$$|\nabla S| = \sqrt{S_x^2 + S_y^2},$$

which can be used to check against a predefined threshold value for edge detection. To guarantee thin edges it is also possible to filter out points whose gradient magnitude are above the threshold but are not local maxima. Examples of this process are shown in Figures 10.8 and 10.9.

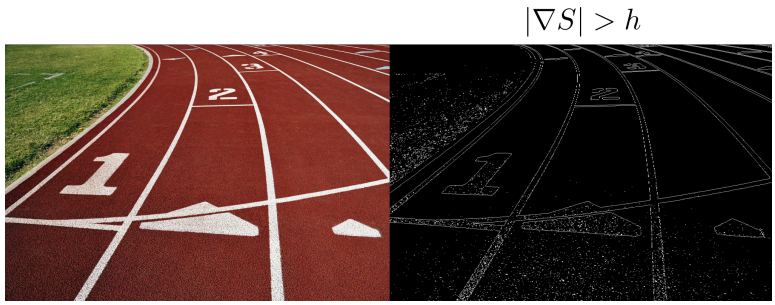


Figure 10.8: Edge detection using the "Sobel" edge detector.

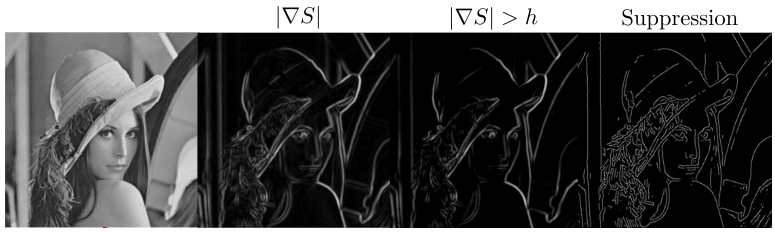


Figure 10.9: Edge detection using the "Canny" edge detector.

10.2.2 *Corner Detection*

A *corner* in an image is defined as an intersection of two or more edges, and also sometimes as a point where there is a large intensity variation in every direction. Important properties of corner detectors include repeatability and distinctiveness. Repeatability quantifies how well the same features can be found in multiple images even under geometric and photometric transformations. Distinctiveness refers to whether the information carried by the patch surrounding

the feature is distinctive, which can be used to reliably produce correspondences. Both of these properties are particularly important in applications such as panorama stitching and 3D reconstruction.

Generally corner detection can be thought of in a similar way to edge detection, except that instead of looking for change along one direction there should be changes in all directions. One well known corner detector is known as the Harris detector [4], which has the useful property that the detection is invariant to rotations and linear intensity changes (i.e. geometric and photometric invariance). However the Harris detector is not invariant to scale changes or geometric affine changes, which has led to the development of scale-invariant detectors such as the Harris-Laplacian detector or the scale-invariant feature transform (SIFT) detector.

[4] C. Harris and M. Stephens. "A combined corner and edge detector". In: *4th Alvey Vision Conference*. 1988

10.3 Image Descriptors

Image *descriptors* describe features so that they can be compared across images, or used for object detection and matching. Similar to image detectors, it is also desirable for image descriptors to be repeatable (i.e. invariant with respect to pose, scale, illumination, etc.) and distinct. Perhaps the simplest example of a descriptor is an $n \times m$ window of pixel intensities centered at the feature, which can be normalized to be illumination invariant. However, such a descriptor is not invariant to pose or scale and is not distinctive, and therefore is generally not useful in practice. Alternative detectors/descriptors that have become popular include SIFT, SURF, FAST, BRIEF, ORB, and BRISK.

10.4 Exercises

10.4.1 Linear Filtering

Complete *Problem 3: Linear Filtering* located in the online repository:

`https://github.com/PrinciplesofRobotAutonomy/AA274A_HW3`,

where you will explore the use of linear filters for image processing.

11
Information Extraction

The last chapter introduced some fundamental topics related to image processing, namely filtering, feature detection, and feature description. While these techniques are quite useful for a large number of computer vision applications, they may not be sufficient to extract higher-level information from images. For example, the features that were discussed are *local features* that describe important keypoints of the image, but these may be too localized to discuss higher-level features or semantic content. In some cases it may be possible to correlate local features to extract higher-level information (e.g. image matching), but in other cases higher-level algorithms may be useful (e.g. identifying a particular object in a scene, such as a person). In particular, object recognition is a very important task in robotics and therefore some common methods for object recognition will be discussed in this chapter.

This chapter will additionally focus on geometric feature extraction[1], which is used to extract structure from data in the form of geometric primitives (e.g. lines, circles, planes). This is very useful in robotics for localization and mapping, and these algorithms can generally be applied to different types of data, such as data extracted from images or even data collected via laser rangefinders or radar.

[1] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, 2011

Information Extraction

This chapter will focus on common methods for extracting higher-level environmental information from sensor data that is useful for robotics. In particular, common algorithms for *geometric feature extraction* will be presented, as well as methods for *object recognition* in images. Such information is crucial for robots operating in real environments to enable intelligent decision making and task planning, as well as to execute plans safely in unknown environments with obstacles.

11.1 Geometric Feature Extraction

It is very common in robotic localization and mapping to represent the environment using simple geometric primitives (e.g. lines, circles, corners, planes) that can be efficiently extracted from sensor data. In particular, in this section techniques for line extraction from range data[2] will be presented. Lines in particular are one of the most fundamental geometric primitives to be extracted, and generally the techniques for extracting other primitives are conceptually similar.

[2] Range data can generally come from a variety of sources, including laser rangefinders, radar, or even computer vision.

There are two main challenges with extracting lines from range data. The first is called *segmentation*, which is the task of identifying which data points belong to which line (and inherently also identifying how many lines there are). The second is *fitting*, which is the task of estimating the parameters that define a line given a set of points. For simplicity this chapter will consider line extraction problems based on two-dimensional range data.

11.1.1 Line Segmentation

The line segmentation problem is to determine how many lines exist in a given set of data and also which data points correspond to each line. Three popular algorithms for line segmentation will be discussed, the *split-and-merge* algorithm, the *random sample consensus (RANSAC)* algorithm, and the *Hough-transform* algorithm.

Split-and-Merge: The split-and-merge algorithm is perhaps the most popular line extraction algorithm and is arguably the fastest (but not as robust to outliers). The concept of this algorithm is quite simple: repeatedly fit lines to sets of points and then split the set of points into two sets if any point lies more than distance d from the line. By repeating this process until no more splits occur, it is guaranteed that all points will lie less than distance d to a line. After this "split" process is completed, a second step merges any of the newly formed lines that are colinear. This algorithm is presented in more detail in Algorithm 2. A popular variant of the split-and-merge algorithm is known as the iterative-end-point-fit algorithm. This algorithm is simply the split-and-merge algorithm given in Algorithm 2 where the line is simply constructed by connecting the first and the last points of the set. This approach is shown graphically in Figure 11.1.

Random Sample Consensus (RANSAC): Random Sample Consensus (RANSAC) is an algorithm to estimate the parameters of a model from a set of data that may contain outliers (i.e. *robust* model parameter estimation). Outliers are data points that do not fit the model and may be the result of high noise in the data, incorrect measurements, or simply points which come from objects that are unrelated to the current model. For example, a typical laser scan of an indoor en-

Algorithm 2: Split-and-Merge

Data: Set S of N points, distance threshold $d > 0$
Result: A list L of sets of points each resembling a line
$L \leftarrow [S]$
$i \leftarrow 1$
while $i \leq len(L)$ **do**
  fit a line (α, r) to the set $L[i]$
  detect the point $P \in L[i]$ with maximum distance D to the line (α, r)
  if $D < d$ **then**
    $i \leftarrow i + 1$
  else
    split $L[i]$ at P into new sets S_1 and S_2
    $L[i] \leftarrow S_1$
    $L[i+1] \leftarrow S_2$
Merge colinear sets in L

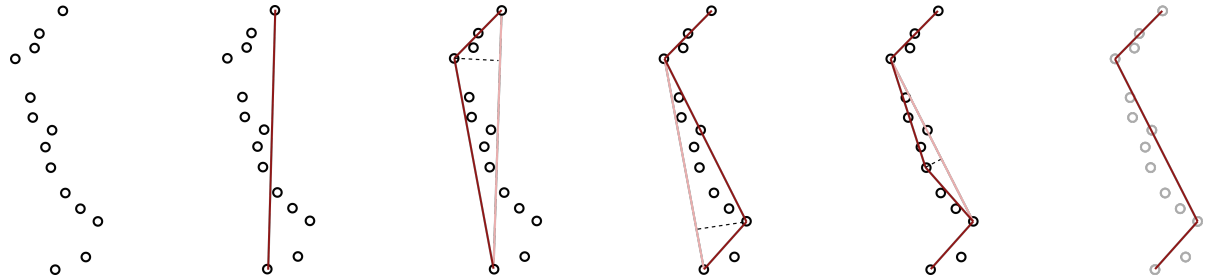


Figure 11.1: Iterative-end-point-fit variation of the split-and-merge algorithm for extracting lines from data.

vironment may contain distinct lines from the surrounding walls but also points from other static and dynamic objects (e.g. chairs or humans). In this case, if the goal was to extract lines to represent the walls then any data point corresponding to other objects would be an outlier. In general, RANSAC can be applied to many parameter estimation problems, and typical applications in robotics include line extraction from 2D range data, plane extraction from 3D point clouds, and structure-from-motion (where the goal is to identify image correspondences which satisfy a rigid body transformation). However for simplicity this section focuses on using RANSAC for line extraction from 2D data.

RANSAC is an iterative method and is non-deterministic (i.e. stochastic or random). Given a dataset S of N points, the algorithm starts by randomly selecting a sample of two points from S. Then a line is constructed from these two points and the distance of all other points to this line is computed. A set of *inliers* comprised of all the points whose distance to the line is within a predefined threshold d is then defined. By repeating this process k times, k inlier sets (and their associated lines) are generated and the inlier set with the most points is returned. This procedure is detailed in Algorithm 3 and is also illustrated in Figure 11.2.

Algorithm 3: Random Sample Consensus (RANSAC) for Line Extraction

Data: Set S of N points, distance threshold d
Result: Set with maximum number of inliers and corresponding line
while $i \leq k$ **do**
  randomly select 2 points from S
  fit line l_i through the 2 points
  compute distance of all other points to l_i
  construct set of points $\tilde{S}_i$ with distance less than d to l_i
  store line l_i and set of points $\tilde{S}_i$ $i \leftarrow i + 1$
Choose set $\tilde{S}_i$ with maximum number of points

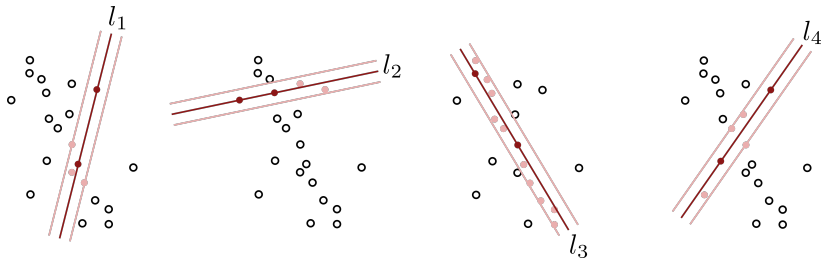


Figure 11.2: Example of the RANSAC algorithm, showing four iterations of the algorithm. If the algorithm was terminated after these four iterations, line l_3 would be returned since it contains the maximum number of points.

Due to the probabilistic nature of the algorithm, as the number of iterations k increases the probability of finding a good solution increases. This approach is used over a brute force search of all possible combinations of two points since the total number of combinations is $N(N-1)/2$, which can be extremely large. In fact, a simple statistical analysis of RANSAC can be performed.

Let p be the *desired* probability of finding a set of points free of outliers and let w be the probability of selecting an inlier from the dataset S of N points, which can be expressed as:

$$w = \frac{\text{\# inliers}}{N}.$$

Assuming point samples are drawn independently from S, the probability of drawing two inliers is w^2 (and $1 - w^2$ is the probability that at least one is an outlier). Therefore, with k iterations the probability that RANSAC *never* selects two points that are both inliers is $(1 - w^2)^k$. Therefore the minimum number of iterations $\bar{k}$ needed to find an outlier-free set with probability p can be found by solving:

$$1 - p = (1 - w^2)^k,$$

for k. In other words, $\bar{k}$ can be computed as:

$$\bar{k} = \frac{\log(1-p)}{\log(1-w^2)}.$$

While the value of w may not be known exactly[3], this expression can still be used to get a good estimate of the number of iterations k that are needed for good results. It is important to note that this probabilistic approach often leads

[3] There also exist advanced versions of RANSAC that can estimate w in an adaptive online fashion.

to a much smaller number of iterations than for brute force searching through all combinations. This can be attributed to the fact that $\bar{k}$ is only a function of w and not the total number of samples N in the dataset.

Overall, the main advantage of RANSAC is that it is a generic extraction method and can be used with many types of features given a feature *model*. It is also simple to implement and is robust with respect to outliers in the data. The main disadvantages are that the algorithm needs to be run multiple times if multiple features are to be extracted, and there are no guarantees that the solutions will be optimal.

Hough Transform: In the Hough transform algorithm, each point (x_i, y_i) of the set S "votes" for a *set* of possible line parameters (m, b) (i.e. slope and intercept). For any given point (x_i, y_i) the candidate set of line parameters (m, b) that could pass through this point must satisfy $y_i = mx_i + b$, which can also be written as:

$$b = -mx_i + y_i.$$

Therefore it can be noted that each point in the original space space (x, y) maps to a *line* in the Hough space (m, b) (see Figure 11.3). The Hough transform algorithm exploits this fact by noting that two points on the same line in the original space will yield two *intersecting* lines in Hough space. In particular, the point where they intersect in the Hough space corresponds to the parameters m^* and b^* that defines the line passing between the points in the original space (see Figure 11.4).

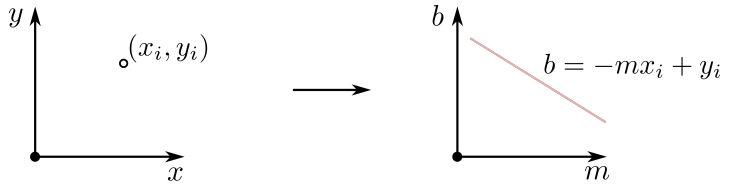


Figure 11.3: Each point (x_i, y_i) in the original space maps to a *line* in the Hough space which describes all possible parameters m and b that would generate a line passing through the point (x_i, y_i).

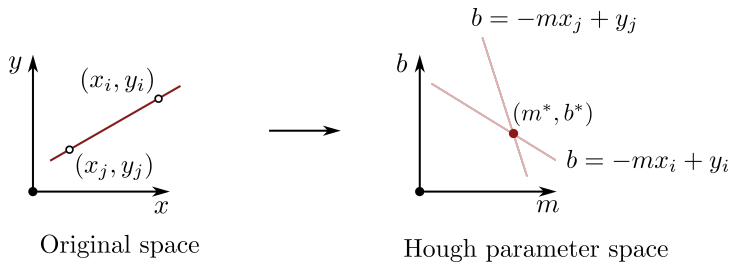


Figure 11.4: All points on a line in the original space yield lines in the Hough space that intersect at a common point.

This concept can be applied to line segmentation by searching in the Hough space for intersections among the lines that correspond to each point (x, y) in the set S. In practice, this can be done by discretizing the Hough space with a grid and simply counting for each grid cell the number of lines corresponding

to (x_i, y_i) points from S that pass through it. Local maxima among the cells then can be chosen as lines that "fit" the data set S.

However, performing a discretization of the Hough space requires a trade-off between range and resolution (in particular because m can range from $-\infty$ to ∞. Alternatively, it is possible to use a polar coordinate representation of the Hough space which defines a line as:

$$x \cos \alpha + y \sin \alpha = r,$$

where (α, r) are the new line parameters. With this representation, a point (x_i, y_i) from the original space gets mapped to the polar Hough space (α, r) as a sinusoidal curve (see Figure 11.5). An example of the Hough transform using the polar representation is given in Figure 11.6.

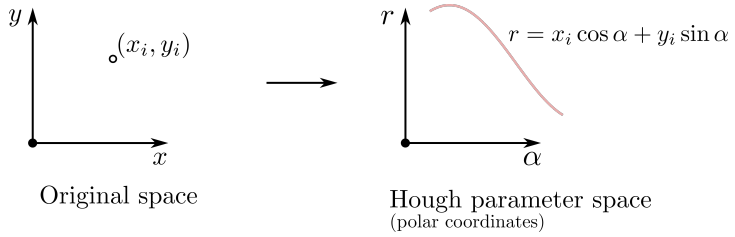


Figure 11.5: Representation of a point (x_i, y_i) in the Hough space when using a polar coordinate representation of a line with parameters α and r.

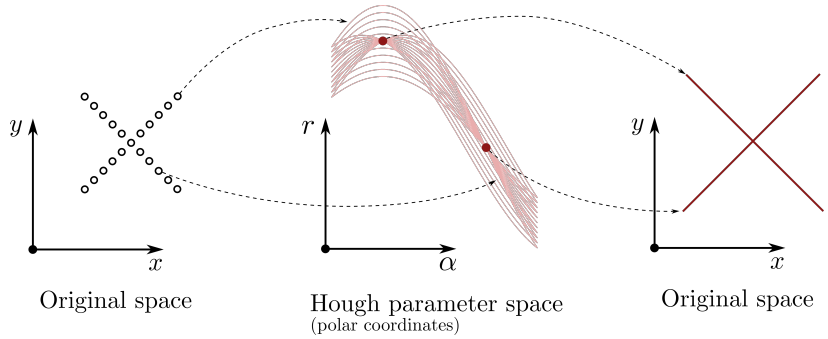


Figure 11.6: Example of the Hough transformation using a polar coordinate representation of lines.

11.1.2 *Line Fitting*

Line segmentation is the process of identifying which data points belong to a line, and line fitting is the process of estimating parameters of a line for those corresponding data points. For the line segmentation algorithms previously discussed (i.e. split-and-merge, RANSAC, and Hough-transform), a line associated with the segmented data points was also implicitly defined. However, the lines implicitly defined from the segmentation algorithms may not always be ideal and so other techniques have been developed to specifically address the line fitting task.

Line fitting algorithms search for lines that best fit a set of data points. In almost all cases the problem is over-determined (i.e. there are more data points

than parameters to choose) and noise in the data means that there is not perfect solution. Therefore one of the most common approaches to line fitting is based on *least-squares estimation*, which tries to find a line that minimizes the overall error in the fit. For this approach it is useful to work in polar coordinates defined by:

$$x = \rho \cos\theta, \quad y = \rho \sin\theta,$$

where (x, y) is the 2D Cartesian coordinate of a data point and (ρ, θ) is the 2D polar coordinate. In polar coordinates the equation of a line is given by

$$\rho \cos(\theta - \alpha) = r, \quad \text{or} \quad x \cos\alpha + y \sin\alpha = r, \tag{11.1}$$

where α and r are the parameters that define the line. For a visual representation of these definitions see Figure 11.7.

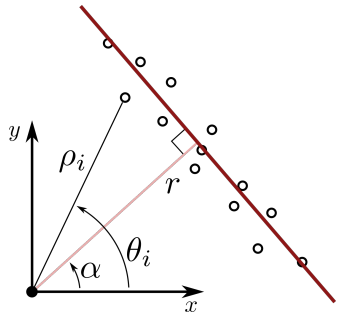


Figure 11.7: Representation of a line in polar coordinates, defined by the parameters r and α which are the distance and angle to the closest point on the line to the origin.

For a collection S of N points (ρ_i, θ_i), the error d_i corresponding to the perpendicular distance from a point to a line defined by parameters α and r can be computed by:

$$d_i = \rho_i \cos(\theta_i - \alpha) - r. \tag{11.2}$$

The line fitting task can then be formulated as an optimization problem over the parameters α and r to minimize the combined errors d_i for $i = 1, \ldots, N$. In particular, the combined errors are aggregated using a sum of the squared errors:

$$S(r, \alpha) = \sum_{i=1}^{N} d_i^2 = \sum_{i=1}^{N} (\rho_i \cos(\theta_i - \alpha) - r)^2. \tag{11.3}$$

This is a classic least squares optimization problem that can be efficiently solved. However, this cost function generally assumes that each of the data points is equally affected by noise (i.e. the uncertainty of each measurement is the same). In some cases it might be beneficial to account for differences in data quality for each point i, which could give preference to well known points.

Accounting for unique uncertainties in each data point leads to a *weighted least squares estimation* problem. In particular, it is assumed that the variance of each range measurement ρ_i is given by σ_i. The cost function (11.3) is then modified to be:

$$S_w(r, \alpha) = \sum_{i=1}^{N} w_i d_i^2 = \sum_{i=1}^{N} w_i (\rho_i \cos(\theta_i - \alpha) - r)^2, \tag{11.4}$$

where the weights w_i are given by:

$$w_i = \frac{1}{\sigma_i^2}.$$

It can be shown that the solution to the optimization problem defined by the

weighted cost function (11.4) is given by:

$$r = \frac{\sum_{i=1}^{N} w_i \rho_i \cos(\theta_i - \alpha)}{\sum_{i=1}^{N} w_i},$$
$$\alpha = \frac{1}{2}\text{atan2}\left(\frac{\sum_{i=1}^{N} w_i \rho_i^2 \sin(2\theta_i) - \frac{2}{\sum_{i=1}^{N} w_i}\sum_{i=1}^{N}\sum_{j=1}^{N} w_i w_j \rho_i \rho_j \cos\theta_i \sin\theta_j}{\sum_{i=1}^{N} w_i \rho_i^2 \cos(2\theta_i) - \frac{1}{\sum_{i=1}^{N} w_i}\sum_{i=1}^{N}\sum_{j=1}^{N} w_i w_j \rho_i \rho_j \cos(\theta_i + \theta_j)}\right) + \frac{\pi}{2}. \tag{11.5}$$

11.2 Object Recognition

Another high-level information extraction task that is common in robotics is *object recognition*. Object recognition is the task of classifying or naming discrete objects in the world (usually based on images or video). This can be a particularly challenging task because real world scenes are commonly made up of many varying types of objects which can appear at different poses and can occlude each other. Additionally, objects within a specific class can have a large amount of variability (e.g. breeds of dogs or car models). In this section three common methods for object recognition will be introduced, namely template matching, bag of visual words, and neural network methods.

11.2.1 Template Matching

Template matching[4] is a machine vision technique for identifying parts of an image that match a given image pattern[5]. This approach has seen success in a variety of applications, including manufacturing quality control, mobile robotics, and more. The two primary components needed for template matching are the source image I and a template image T

Given a source and template image, one approach to template matching is to leverage the linear spatial correlation filters discussed in the previous chapter. In particular, a naive approach would be to use the normalized template image as a filter mask in a correlation filter. By applying this filter mask to every pixel in the source image the resulting output would quantify the similarity of that region of the source image to the template. This type of approach is sometimes referred to as a *cross-correlation*. Another approach based on linear spatial filters from the previous chapter would be to leverage the similarity filters that compute the sum of absolute differences (SAD) metric for each pixel in the source image. Regions of the source image similar to the template would correspond to low SAD scores. The disadvantages of these approaches is that do not handle rotations or scale changes, which are quite common in real world applications.

One solution to the scaling issue in correlation filter based template matching is to simply re-scale the source image multiple times and perform template matching on each. This concept, referred to as using *image pyramids*[6], can also be used to accelerate object search by using a coarser resolution image first to localize the object and then using finer resolution images for actual detection.

[4] N. Perveen, D. Kumar, and I. Bhardwaj. "An overview on template matching methodologies and its applications". In: *International Journal of Research in Computer and Communication Technology* 2.10 (2013), pp. 988–995

[5] Advanced template matching algorithms allow finding pattern occurrences regardless of their orientation and local brightness.

[6] R. Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010

Building image pyramids can be accomplished in several ways. One naive approach is to simply eliminate some rows and columns of the image. Another approach is to first use a Gaussian smoothing filter to remove high frequency content form the image and *then* subsample the image. The sequence of images resulting from this approach is referred to as a *Gaussian pyramid*.

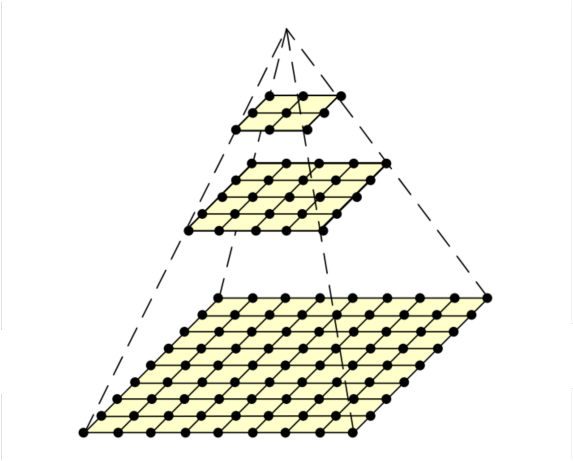


Figure 11.8: A traditional image pyramid: each level has half the resolution (width and height), and hence a quarter of the pixels, of its parent level. Figure from Szeliski (2010) .

11.2.2 Bag of Visual Words

The key idea behind the bag of visual words[7] approach is that object representations can be simplified by considering them as a collection of their subparts (e.g. a bike is an object with wheels, a frame, and handlebars), and the subparts are referred to as *visual words*. In this approach a source image is searched for *visual words*, and a distribution of visual words that are found in the image is created (in the form of a histogram). Object detection can then be performed by comparing this distribution to a set of training images. For example, suppose the source image contains a human face and the recognized features included eyes and a nose. Then by comparing the distribution to training images, it would likely be determined that training images that also have eyes and a nose are also images of faces.

[7] The model originated in natural language processing, where we consider texts such as documents, paragraphs, and sentences as collections of words - effectively "bags" of words.

11.2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) represent a relatively new and very powerful paradigm in object recognition. These approaches were first introduced in the field of computer vision for image recognition in 1989, and since then have significantly boosted performance in image recognition and classification tasks. Research in this field is also still very active.

11.3 Exercises

All exercises for this chapter can be found in the online repository:

`https://github.com/PrinciplesofRobotAutonomy/AA274A_HW3`.

11.3.1 Line Extraction

Complete *Problem 2: Line Extraction*, where you will implement a line extraction algorithm (Split-and-Merge) to fit lines to simulated Lidar range data.

11.3.2 Template Matching

Complete *Problem 4: Template Matching*, where you will explore the use of the classic template matching algorithm, implemented in the open-source OpenCV library.

11.3.3 Image Pyramids

Complete *Extra Problem: Image Pyramids*, where you will learn about how template matching algorithms can be enhanced through the use of image pyramids (and image filtering).

12
Modern Computer Vision Techniques

Machine learning has become an extremely powerful tool in the realm of computer vision, particularly through the use of convolutional neural networks (CNNs) and the increased availability of data and computational resources. This chapter introduces the fundamentals of CNNs, as well as their application to computer vision and robotics.

Modern Computer Vision

Modern computer vision techniques[1] rely heavily on deep learning and convolutional neural network architectures[2]. A convolutional neural network is a type of neural network with additional structure that is beneficial for image processing tasks. In fact, CNNs can be said to be "regularized" neural networks since the additional structure reduces the ability of the network to overfit to data. This chapter will introduce each component[3] in the architecture of a CNN, and then discuss how CNNs can be applied to problems in robotics.

[1] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2011

[2] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016

[3] Convolutional layers, nonlinear activations, pooling layers, and fully-connected layers.

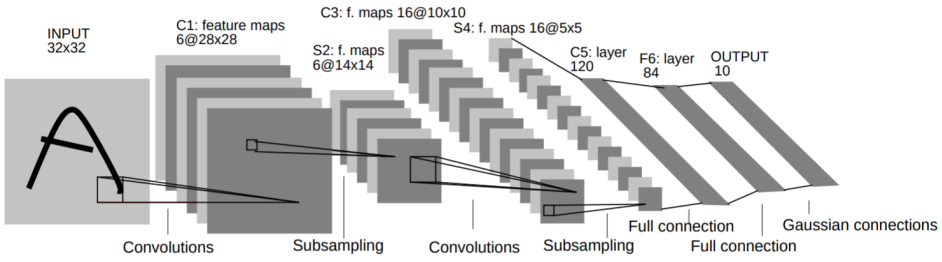


Figure 12.1: Example convolutional neural network architecture from LeCun et al. (1998).

12.1 Convolutional Neural Networks

12.1.1 Convolution Layers

One of the main structural concepts that is unique to the architecture of a CNN is the use of convolution layers. These layers exploit the underlying *spatial locality* structure in images by using sliding "learned" filters, which are often much smaller than the image itself. Mathematically these filters perform operations in a similar way as other linear filters that have been used in image processing, such as Gaussian smoothing filters, and can be expressed as affine functions:

$$f(x) = w^\top x + b,$$

where w is a vectorized representation of the weight parameters that define the filter, x is a vectorized version of the image pixels covered by the filter, and b is a scalar bias term. For example in Figure 12.2 a filter is applied over an image with three color channels (red, green, blue). In this case the filter may have dimension $m \times n \times 3$, which could be vectorized to a weight vector w with $3mn$ elements. Additionally, the *stride* of the filter describes how many positions it shifts by when sliding over the input. The output of the filter is also passed through a nonlinear activation, typically a ReLU function.

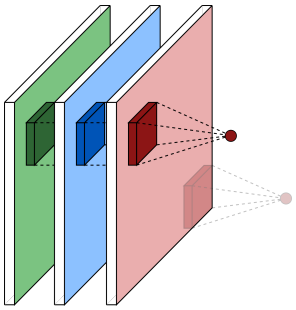


Figure 12.2: A convolution filter being applied to a 3-channel RGB image.

Once the filter has been applied to the entire image, the collection of outputs from the activation function will create a new "filtered image" typically referred to as an *activation map.* In practice a number of different filters are usually learned in each convolution layer, which would simply produce a corresponding number of activation maps as the output[4]. This is crucial such that each filter can focus on learning one specific relevant feature. Examples of different filters that might be learned in different convolution layers of a CNN are shown in Figure 12.4[5]. Notice that the low-level features which are learned in earlier convolution layers look a lot like edge detectors (i.e. are more basic/fundamental) while later convolution layers have filters that look more like actual objects.

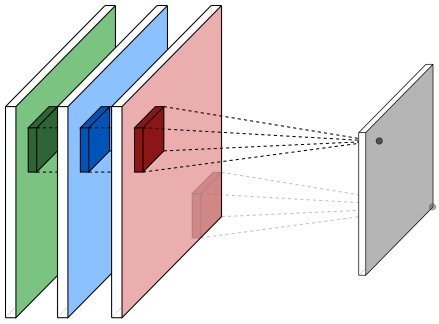


Figure 12.3: The outputs of a convolution filter and activation function applied across an image make up a new image, called an *activation* map.

In general, the use of convolution layers to exploit the spatial locality of images provides several benefits including:

1. *Parameter sharing*: the (small) filter's parameters are applied at all points on

[4] Besides the number of filters applied to the input, the width and height of the filter, the amount of padding on the input, and the stride of the filter are other hyperparameters.

[5] M. D. Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks". In: *European Conference on Computer Vision (ECCV)*. Springer, 2014, pp. 818–833

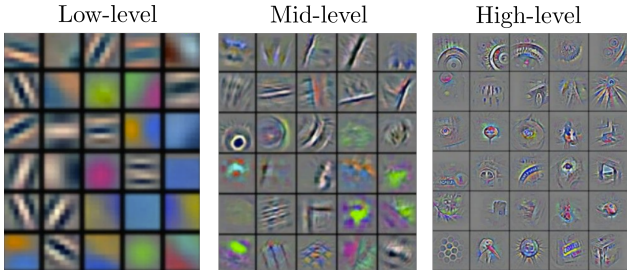


Figure 12.4: Low-level, mid-level, and high-level feature visualizations in a convolutional neural network from Zeiler and Fergus (2014).

the image. Therefore the total number of learned parameters in the model is much smaller than if a fully-connected layer was used.

2. *Sparse interactions*: having the filter be smaller than the image allows for better detection of smaller, more meaningful features and improves computation time by requiring fewer mathematical operations to evaluate.

3. *Equivariant representation:* the convolution layer is equivariant to translation, meaning that the convolution of a shifted image is equivalent to the shifted convolution of the original image[6].

4. The ability to work with images of *varying size* if needed.

[6] However, convolution is not equivariant to changes in scale or rotation.

12.1.2 Pooling Layers

Pooling is the second major structural component in CNNs. Pooling layers typically come after convolution layers (and their nonlinear activation functions). Their primary function is to replace the output of the convolution layer's activation map at particular locations with a "summary statistic" from other spatially local outputs. This helps make the network more robust against small translations in the input, helps improve computational efficiency by reducing the size of the input (i.e. it lowers the resolution), and is useful in enabling the input images to vary in size[7]. The most common type of pooling is *max* pooling, but other types also exist (such as *mean* pooling).

Computationally, both max and mean pooling layers operate with the same filtering idea as in the convolution layers. Specifically, a filter of width m and height n slides around the layer's input with a particular stride. The difference between the two comes from the mathematical operation performed by the filter, which as their names suggest are either a maximum element or the mean over the filter. If the output of the convolution layer has N activation maps, the output of the pooling layer will also have N "images", since the pooling filter is only applied across the spatial dimensions.

[7] The size of the pooling can be modified to keep the size of the pooling layer output constant.

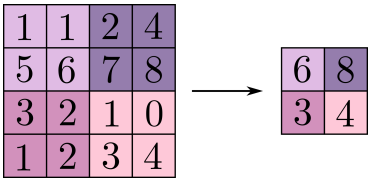


Figure 12.5: Max pooling example with 2 $\times$ 2 filter and stride of 2.

12.1.3 Fully Connected Layers

Downstream of the convolution and pooling layers are fully connected layers. These layers make up what is essentially just a standard neural network, which

is appended to the end of the network. The function of these layers is to take the output of the convolution and pooling layers (which can be thought of as a highly condensed representation of the image) and actually perform a classification or regression. Generally the total number of fully connected layers at the end of the CNN will only make up a fraction of the total number of layers.

12.1.4 CNN Performance

A CNN can be said to learn how to process images *end-to-end* because it essentially learns how to perform two steps simultaneously: feature extraction and classification or regression (i.e. it learns the entire process from image input to the desired output). In contrast, classical approaches to image processing use hand-engineered feature extractors. Since 2012, the performance of end-to-end learning approaches to image processing have dominated and continue to improve[8]. This continuous improvement has generally been realized with the use of deeper networks.

[8] Of course in some specific applications hand-engineered features may still be better! For example if engineering insight can identify a structure to the problem that a CNN could not.

12.2 CNNs for Object Detection and Localization

Modern computer vision techniques such as convolutional neural networks have a large variety of applications in robotic perception, including object localization and detection.

In object localization problems, the goal is to identify the position of an object in the image. This is usually accomplished by specifying four numbers that define a *bounding box* for the object in the image[9] (see Figure 12.6). To solve object localization problems with a CNN, the standard approach is to have the output of the network be both the bounding box coordinates and an object class. This can be accomplished by reusing the convolution and pooling layers of the CNN but then have two separate branches of the fully connected layers: one trained for classification and the other for localization[10]. To train a network to simultaneously perform classification and localization, a multi-task loss function can be defined by adding together the loss functions of each branch.

[9] Box coordinates are usually the (x, y) position of the top-left corner and the width w and height h of the box.

[10] Since the output of the localization branch is four real numbers (x, y, w, h), this would be considered a regression problem and it is common to use and l_2 loss function.

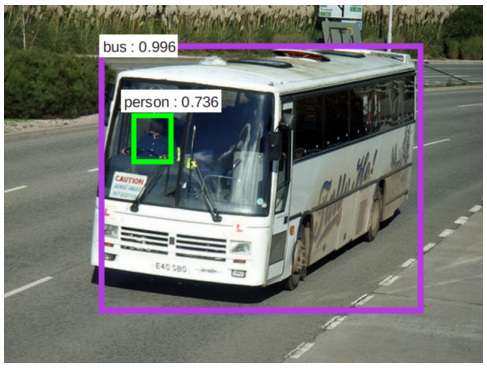


Figure 12.6: Bounding box prediction for several objects in an image from Ren, He, et al. (2017)

If multiple objects exist within a single image the object localization and classification problem becomes more difficult. First, the number of outputs of the network may change! For example, outputting a bounding box for n objects would require $4n$ outputs. A practical solution to handling the problem of varying outputs is to simply apply the CNN to a series of cropped images produced from the original image, where the network can also classify the image as "background" (i.e. not an object)[11]. However, a naive implementation of this idea would likely result in an excessive number of CNN evaluations. Instead, different approaches have been developed for making this idea efficient by reducing the number of areas in the image that need to be evaluated. For example this has been accomplished by identifying "regions of interest" in the image through some other approach, or even partitioning the image into a grid.

[11] This could be thought of as applying the entire CNN as a filter that slides across the image.

Part III

Robot Localization

13
Introduction to Localization and Filtering

Previous chapters introduced some of the fundamental concepts related to robotic perception, and specifically techniques for sensing the environment and extracting useful semantic information. While these techniques provide *local* information that is crucial for robots to navigate autonomously, additional *global* information is often required. For example, distance measurements from a laser rangefinder might be useful for detecting objects in an environment, but they only provide information *relative* to the robot's current position. Alternatively, object detection via computer vision only provides information about what is in the robot's *current* view. Robotic autonomy, in particular autonomous decision making and planning, generally requires more than just local information to answer questions such as "have I seen this object before?" and "have I been here before?". These new challenges, associated with building a global understanding of the environment from local measurements, are often referred to as *localization and mapping*[1].

[1] Localization and mapping is the component of the "think" part of the "see, think, act" cycle that connects with robotic perception.

Introduction to Localization and Filtering

The problem of *localization* is to endow the robot with the ability to understand its current position with respect to its environment in a global sense[2,3]. One of the main classes of techniques for robot localization are *map-based*, where the robot explicitly localizes its position with respect to a *map* of the environment. For example, consider the floor plan (the environment map) in Figure 13.1: before a robot can navigate to a particular room it must know where in the building it is currently located.[4]

[2] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005

[3] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, 2011

[4] Other approaches to navigate in environments include *behavioral* approaches, which rely on a specified set of behaviors that will result in a desired global behavior without the need for explicit mapping or localization. An example of this approach would be to have a left-wall following behavior for movement about a building.

There are two primary components to map-based localization: map representation and belief representation. This chapter focuses on belief representation, which addresses the problem of how to best represent the robot's belief of its current position with respect to the map. One simple approach would be to simply store a best guess of the robot's position (in some map-based coordinate system). However in practice localization information is often *uncertain*, and representing the belief by only a best guess does not capture this important fact. Therefore one common approach is to use a *probabilistic* representation of the

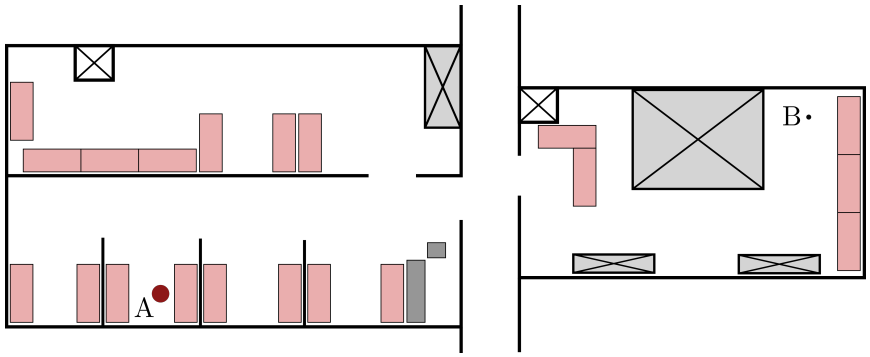


Figure 13.1: An example environment where localization is crucial for robotic autonomy. For a robot to move from location A to location B it must first understand which room it is in, and that the only path to B is through the hallway. Extracting such global information about the environment from local measurements (e.g. from a range sensor) requires specialized algorithms.

robot's belief since probability distributions can be used to model uncertainty (and extract best guesses, for example by finding the mean of a unimodal distribution). A variety of probabilistic representations can be used, for example singe-hypothesis or multiple-hypothesis representations as well as continuous or discrete representations. A few examples showing the differences between these types of representations are given in Figure 13.2. Some representations are more expressive than others, but there is usually a trade-off with computational complexity of the resulting algorithms that support the representation. Algorithms based on these different probabilistic representations will be presented in this chapter and in subsequent chapters.

13.1 Basic Concepts in Probability

Before discussing different types of robot localization algorithms it is useful to provide a review of some of the fundamental concepts from probability.

13.1.1 Random Variables

Uncertain quantities such as sensor measurements, robot state, and environment variables can be modeled as discrete or continuous *random variables*.

Definition 13.1.1 (Discrete Random Variable)**.** *A discrete random variable X is a random variable that can only take on values from a countable set. Discrete random variables are characterized by a probability mass function $p(x)$ (which can be read as $p(X = x)$, "the probability that X takes on value x") that satisfies:*

$$\sum_x p(x) = 1,$$

where the summation is over all possible values of X.

Definition 13.1.2 (Continuous Random Variable)**.** *A continuous random variable X is a random variable that can take on values from a continuous range. Continuous random variables are characterized by a probability density function $p(x)$ that satisfies:*

$$\int_{-\infty}^{\infty} p(x)dx = 1.$$

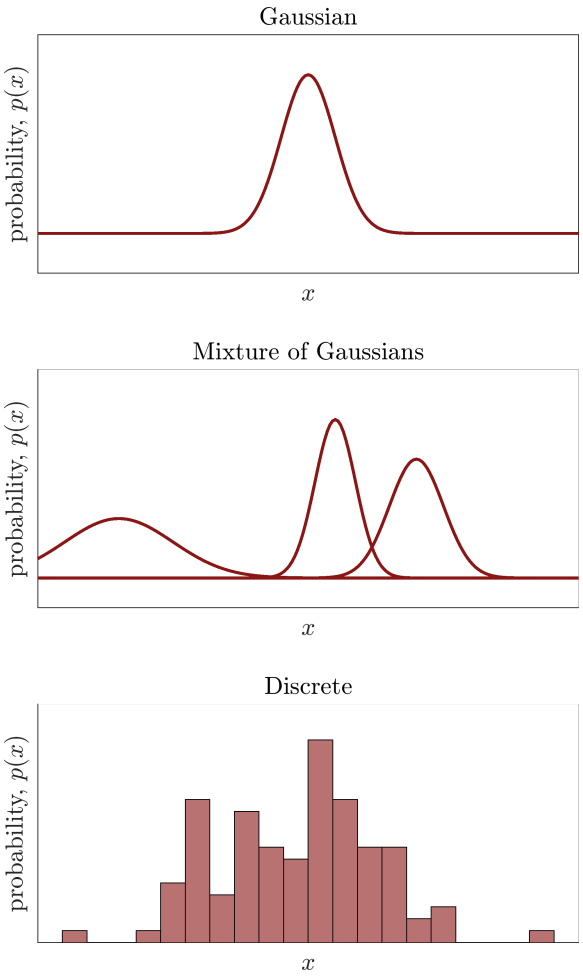


Figure 13.2: A graphical representation of different types of probabilistic representations: (a) a continuous single-hypothesis belief (e.g. from a single Gaussian distribution), (b) a continuous multiple-hypothesis belief (e.g. a mixture of Gaussians), (c) discrete representation with a finite number of possible values.

The probability of the random variable taking on a value in the interval $[a, b]$ *is similarly defined as:*

$$P(a \leq X \leq b) = \int_a^b p(x)dx = 1.$$

A common example of a discrete random variable is the result of a coin flip, which can only take on two values: heads or tails. In robotics, a common example of a continuous random variable may be the position of the robot, which could take on an infinite number of values.

13.1.2 *Joint Distributions, Independence, and Conditioning*

Many applications of probability theory rely on more than one random variable. In these instances it is useful to be able to quantify probabilities associated with multiple random variables at the same time. One of the most fundamental tools when dealing with multiple variables is the *joint distribution*.

Definition 13.1.3 (Joint Distribution)**.** *The joint distribution of two random variables*

X and Y defines the probability associated with both taking on specific values at the same time. This is denoted mathematically as $p(x, y)$, which can be read as $p(X = x \text{ and } Y = y)$.

It is also useful to determine whether different random variables have any relationship to each other. In particular, two random variables that do not have any influence on each other in a probabilistic sense are considered to be probabilistically independent.

Definition 13.1.4 (Independence)**.** *Two random variables X and Y are independent if and only if:*

$$p(x, y) = p(x)p(y). \tag{13.1}$$

Independence holds when the occurrence of one value of a random variable does not affect the probability of another random variable taking on a specific value.

Another useful tool that relates two random variables is the conditional probability, which defines the probability of a random variable when the value of a second random variable is *known* or *fixed*.

Definition 13.1.5 (Conditional Probability)**.** *The conditional probability of a random variable X taking on a value given that a second random variable Y has a specific value is defined as:*

$$p(x \mid y) \coloneqq \frac{p(x, y)}{p(y)}. \tag{13.2}$$

This can be read as "the probability of X taking on value x conditioned on the fact that Y has taken on value y".

Notice that if the random variables X and Y are independent, then the conditional probability definition simplifies to $p(x \mid y) = p(x)$, which suggests that knowing that Y has taken on value y has provided no new information about the random variable X (which of course is in line with the definition of independence). Additionally, another notion of independence can be defined based on whether or not two random variables are independent when *conditioned* a third random variable.

Definition 13.1.6 (Conditional Independence)**.** *Two random variables X and Y are conditionally independent given the value of a third random variable Z if and only if:*

$$p(x, y \mid z) = p(x \mid z)p(y \mid z). \tag{13.3}$$

It is important to note however that conditional independence does not imply independence, and vice versa.

13.1.3 Law of Total Probability

The law of total probability defines a relationship between probabilities, joint probabilities, and conditional probabilities.

Definition 13.1.7 (Law of Total Probability)**.** *For discrete random variables X and Y the law of total probability states that:*

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

Similarly, for continuous random variables this law is given by:

$$p(x) = \int p(x, y)dy = \int p(x \mid y)p(y)dy.$$

In words, this law says that the probability of a random variable X taking on a value x can be found by looking at the joint probabilities between X and Y and accounting for *all* possible values of Y. The second part of the law is a direct result of applying the definition of conditional probabilities.

13.1.4 Bayes' Rule

The joint probability $p(x, y)$ between two random variables X and Y can be related to the conditional probabilities $p(x \mid y)$ and $p(y \mid x)$ via the definition of conditional probabilities (13.2). In particular, since the joint probability can be equivalently expressed in two ways it can be seen that:

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

This relationship is commonly referred to as Bayes' rule:

Definition 13.1.8 (Bayes' Rule)**.** *For discrete random variables X and Y, Bayes' rule states that:*

$$p(x \mid y) = \frac{p(y \mid x)p(x)}{p(y)}. \tag{13.4}$$

Bayes' rule is useful as it provides a relationship between the "inverse" conditional probabilities $p(x \mid y)$ and $p(y \mid x)$. This is particularly important for *probabilistic inference*, which is the problem of inferring the value of a random variable from another.

For example, suppose you had a good initial guess of the probability distribution $p(x)$ for a random variable X (the distribution $p(x)$ in this case is often called the *prior*, because it is the guess that comes before any new information is taken into account). Then, suppose some new information regarding the value of a random variable Y is obtained. Using Bayes' rule it is possible to update your belief about the probability distribution of X based on this new information. In particular, the new belief is the conditional probability $p(x \mid y)$ (which is often called the *posterior* because it comes after new information is introduced). These two distributions are related by Bayes' rule!

Bayes' rule can also be extended to cases with additional random variables. For example with three random variables X, Y, and Z, Bayes' rule is:

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

13.1.5 Expectation and Covariance

Probability distributions define in a very precise way the probability associated with any particular value of a random variable. However, sometimes it is useful to aggregate this information into more practically useful metrics. Two of the most commonly used metrics are the *expected value* and the *covariance*.

Definition 13.1.9 (Expected Value)**.** *The expected value for a random variable X is denoted as $E[X]$. For discrete random variables the expected value can be computed by:*

$$E[X] = \sum_x xp(x).$$

Similarly, the expected value for a continuous random variable can be computed by:

$$E[X] = \int xp(x)dx.$$

The expected value can be thought of as the average result of an experiment over an infinite number of trials, and is also sometimes referred to as the *first moment* of the distribution. Additionally, expectation is a *linear operator*, such that:

$$E[aX + b] = aE[X] + b,$$

for any values a and b. In the case that the random variable X is a vector-valued random variable, the expectation of the random vector is simply the vector of expectations of each element.

Definition 13.1.10 (Covariance)**.** *The covariance between two random variables X and Y is denoted $cov(x, y)$ and is computed by:*

$$cov(x, y) = E[(X - E[X])(Y - E[Y])^\top] = E[XY^\top] - E[X]E[Y]^\top$$

Covariance is a metric used to describe the relationship between random variables and is positive if greater values of one variable generally corresponds to greater values of the other (and same for lesser values). Similarly, it is negative if the variables tend to show opposite behavior of each other. If there is no general relationship between the two then their covariance is zero (e.g. independent random variables have zero covariance).

13.2 Markov Models

Recall from previous chapters the kinematic and dynamic models that were developed to describe the physical behavior of a robot. These models consisted of a robot state $\boldsymbol{x}$, and a set of equations that described how $\boldsymbol{x}$ varied in time given some control inputs $\boldsymbol{u}$. In this section another type of model will be developed that is based on these same core ideas. These new models, referred to as *Markov models*, are commonly used in robotics for localization tasks as well as higher level planning tasks.

13.2.1 States, Measurements, and Controls

Similar to previous chapters, the state $\boldsymbol{x}$ is a collection of variables that contains information required to define the physical state of the robot. However unlike previous chapters, the state might also include information about the environment (this state has a higher-level perspective). In the context of robotics, the state may include the robot pose (i.e. location and orientation information), velocity, as well as locations and features of surrounding objects in the environment. Note that in general the state discussed in this section might be different from the state defined for robot kinematics and dynamics (even if the robot is the same). This is because the choice of model is usually specific to the task at hand, and while the kinematic and dynamic models are useful for control, they may not strictly be necessary (or sufficient) for use in localization and planning tasks.

A discrete time formulation is also used in this context, where the state is specified for discrete time instances and denoted by $\boldsymbol{x}_t$ (rather than $\boldsymbol{x}(t)$, as was done in previous chapters). The models developed in this section then describe the changes in the state between time steps, for example between $\boldsymbol{x}_t$ and $\boldsymbol{x}_{t+1}$. It is also useful to define the notation $\boldsymbol{x}_{t_1:t_n} := \boldsymbol{x}_{t_1}, \boldsymbol{x}_{t_2}, ..., \boldsymbol{x}_{t_n}$ for describing a sequence of states between times t_1 and t_n.

The robot interacts with the environment through control actions and by gathering information through measurements[5]. In this context, the measurement data collected at a time t will be denoted as $\boldsymbol{z}_t$, and the control data is denoted as $\boldsymbol{u}_t$. Similar to the state, a useful notation for representing a sequence of measurements or controls is given by $\boldsymbol{z}_{t_1:t_n} := \boldsymbol{z}_{t_1}, \boldsymbol{z}_{t_2}, ..., \boldsymbol{z}_{t_n}$ and $\boldsymbol{u}_{t_1:t_n} := \boldsymbol{u}_{t_1}, \boldsymbol{u}_{t_2}, ..., \boldsymbol{u}_{t_n}$. In general, the measurements can come from any number of the sensors discussed in previous sections on robotic perception, including cameras and laser rangefinders.

[5] In the context of robot localization, measurements increase the robot's knowledge and control actions tend to result in a loss of knowledge.

13.2.2 Model

The kinematic and dynamic models from previous chapters (expressed as a set of ordinary differential equations) were deterministic models. However, to leverage a probabilistic framework for robot localization it is typically required that the model also be probabilistic. In the most general sense a probabilistic model can be defined by:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{0:t-1}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}), \tag{13.5}$$

which defines a probability distribution over the possible current state $\boldsymbol{x}_t$ given the state, measurement, and control histories. Note that here the convention that will be used is that the robot executes control $\boldsymbol{u}_t$ first, and then the measurement $\boldsymbol{z}_t$ can be made based on the resulting state $\boldsymbol{x}_t$. A general probabilistic measurement model can also be defined as:

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_{0:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}). \tag{13.6}$$

In many cases however, the state is defined such that it is *complete*. A state $\boldsymbol{x}_t$ is complete if no variables prior to $\boldsymbol{x}_t$ can influence the future states. In other words, $\boldsymbol{x}$ contains a sufficient amount of information that the history is not important. This is also known as the *Markov property*. If the Markov property holds, the probabilistic model (13.5) can be simplified to:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t), \tag{13.7}$$

and the measurement model (13.6) can be simplified to:

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_t). \tag{13.8}$$

The resulting overall model with the Markov property, consisting of the state transition probability (13.7) and the measurement model (13.7) is referred to as a Bayes network model or a hidden Markov model. Graphically this model can be represented as shown in Figure 13.3, where the sequencing of the control and measurements are more clearly shown (first control, then measurement).

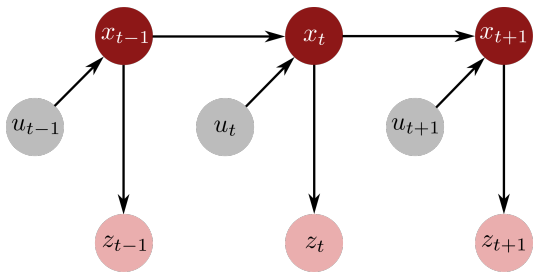


Figure 13.3: Graphical representation of the Bayes network model (hidden Markov model). Note that the sequencing assumes that the control is applied, and then a measurement is taken.

13.3 Bayes Filter

Given a Bayes network model defined by a state transition model (13.7) and a measurement model (13.8), the next task is to determine a way to use this information for robot localization. In particular, the desired task is to estimate the current robot state $\boldsymbol{x}_t$ given the measurement and control information that is available. In the probabilistic framework this estimate is referred to as a *belief distribution*, which is a probability distribution over $\boldsymbol{x}$. This distribution assigns a probability to each hypothesis with respect to the true state. Mathematically the belief distribution is denoted as $bel(\boldsymbol{x}_t)$ and is defined as:

$$bel(\boldsymbol{x}_t) := p(\boldsymbol{x}_t \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}). \tag{13.9}$$

In other words, the belief $bel(\boldsymbol{x}_t)$ is a posterior probability distribution over the state variables conditioned on the available data. A similar distribution, known as the *prediction* distribution, can also be defined as:

$$\overline{bel}(\boldsymbol{x}_t) := p(\boldsymbol{x}_t \mid \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}), \tag{13.10}$$

which is does not include the most recent measurement z_t. The process of computing a belief from a predicted belief (i.e. the process of accounting for the new measurement z_t) is called a *correction* or *measurement update*.

The most general algorithm for computing beliefs $bel(x_t)$ (which leverages Bayes network models that satisfy the Markov property) is known as the *Bayes filter*. This filter is a recursive algorithm that consists of a prediction step for computing $\overline{bel}(x_t)$ and a correction step for computing $bel(x_t)$ given a new measurement z_t.

13.3.1 Algorithm

The Bayes filter algorithm is given in Algorithm 4. In this algorithm, the probability associated with each potential state x_t is updated via a prediction and a correction. The term η in the correction step is simply a normalization constant that ensures the resulting posterior $bel(x_t)$ satisfies the requirements of a probability density function[6]. This algorithm is typically initialized with a prior distribution $bel(x_0)$ that may come from a best guess or simply a uniform distribution. Note that the prediction step is essentially just using the state transition

[6] In fact this normalization constant comes from the denominator in Bayes' rule.

Algorithm 4: Bayes Filter Algorithm

Data: $bel(x_{t-1}), u_t, z_t$
Result: $bel(x_t)$
foreach x_t **do**
  $\overline{bel}(x_t) = \int p(x_t \mid u_t, x_{t-1}) bel(x_{t-1}) dx_{t-1}$
  $bel(x_t) = \eta p(z_t \mid x_t) \overline{bel}(x_t)$
return $bel(x_t)$

model (13.7) to guess what might happen to each state for the given control u_t. The correction step is then modifying the prediction to actually account for what was observed in the real world.

13.3.2 Derivation

Recall that the belief distribution is defined as (13.9), which can be expanded using Bayes' rule to yield:

$$\begin{aligned} bel(x_t) &:= p(x_t \mid z_{1:t}, u_{1:t}), \\ &= \eta p(z_t \mid x_t, z_{1:t-1}, u_{1:t}) p(x_t \mid z_{1:t-1}, u_{1:t}), \end{aligned}$$

where

$$\eta = \frac{1}{p(z_t \mid z_{1:t-1}, u_{1:t})}.$$

The Markov property can then be leveraged to simplify $p(z_t \mid x_t, z_{1:t-1}, u_{1:t}) = p(z_t \mid x_t)$ and the definition of the prediction belief can be used to give:

$$bel(x_t) = \eta p(z_t \mid x_t) \overline{bel}(x_t),$$

which is precisely the second step of the Bayes filter algorithm. Now the derivation of the prediction can be given by again starting from its definition and leveraging the law of total probability:

$$\begin{aligned}\overline{bel}(\boldsymbol{x}_t) &= p(\boldsymbol{x}_t \mid \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}), \\ &= \int p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) p(\boldsymbol{x}_{t-1} \mid \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) d\boldsymbol{x}_{t-1}.\end{aligned}$$

Again the Markov property can now be used to simplify $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) = p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)$, and additionally the structure of the model makes it possible to remove the $\boldsymbol{u}_t$ term from the prior distribution $p(\boldsymbol{x}_{t-1} \mid \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t})$ since the control $\boldsymbol{u}_t$ has no impact on the state $\boldsymbol{x}_{t-1}$ (see Figure 13.3). Therefore the expression above can be simplified to:

$$\overline{bel}(\boldsymbol{x}_t) = \int p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) bel(\boldsymbol{x}_{t-1}) d\boldsymbol{x}_{t-1},$$

since by definition $bel(\boldsymbol{x}_{t-1}) = p(\boldsymbol{x}_{t-1} \mid \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t-1})$. This result is precisely the prediction step from the Bayes filter algorithm.

13.3.3 *Practical Considerations*

The Bayes filter is a great starting point to derive many useful algorithms, but is itself often not practical to implement. In particular it is generally not reasonable to assume that the integrals in Algorithm 4 can be computed, and if they could be approximated via a numerical scheme this may computationally still be challenging.

13.4 *Discrete Bayes Filter*

The discrete Bayes filter is a discrete version of the Bayes filter previously introduced. This filter can be applied to problems where the state space is finite (i.e. only a finite number of values of $\boldsymbol{x}$ are possible). This makes the Bayes filter approach more tractable because the integrals do not need to be computed over an infinite set.

In the discrete Bayes filter the belief $bel(\boldsymbol{x}_t)$ is represented using a probability mass function rather than a probability density function (as is the case with the *continuous* Bayes filter). In particular, this probability mass function is simply a finite collection of probabilities $\{p_{k,t}\}$ where $p_{k,t}$ is the probability associated with state k at timestep t. The algorithm generally follows the exact procedure as the Bayes filter in Algorithm 4, but with summations replacing the integrals. In particular, the discrete Bayes filter algorithm is provided in Algorithm 5.

Algorithm 5: Discrete Bayes Filter Algorithm

Data: $\{p_{k,t-1}\}, \boldsymbol{u}_t, \boldsymbol{z}_t$
Result: $\{p_{k,t}\}$
foreach k **do**
  $\overline{p}_{k,t} = \sum_i p(\boldsymbol{x}_t \mid \boldsymbol{u}_t, \boldsymbol{x}_i) p_{i,t-1}$
  $p_{k,t} = \eta p(\boldsymbol{z}_t \mid \boldsymbol{x}_k) \overline{p}_{k,t}$
return $p_{k,t}$

14
Parametric Filters

The previous chapter introduced a probabilistic framework that can be used to design algorithms for robot localization and state estimation. The chapter concluded with the introduction of the Bayes filter, which is a fundamental algorithm for maintaining and updating a belief distribution (a probability distribution over possible states). While the Bayes filter is generally intractable to implement in practice, it lays a mathematical foundation for the development of algorithms that can exploit structure or other approximations to generate tractable approaches. One such example is the discrete Bayes filter, which assumes that the number of possible states is finite such that the belief distribution can be represented by simply storing the probability of each state individually. This type of distribution is referred to as *non-parametric* since the belief distribution is not required to have a particular structure.

Alternatively, it is possible to develop tractable algorithms for probabilistic localization and state estimation by leveraging *parametric* belief distributions. Parametric distributions are distributions that are fully specified by a fixed number of parameters, for example Gaussian distributions are defined by the mean and covariance parameters. These *parametric filters* can generally be viewed as practical implementations of Bayes filter that exploit structure for efficiency, and include the Kalman filter family of algorithms.

Parametric Filters

Parametric filters[1] are a family of algorithms for robot localization and state estimation that model the robot's belief with parametric distributions. Therefore, as the robot's state evolves and new measurement information arrives, updating the belief distribution is accomplished by simply updating the parameters that define the distribution. This can lead to practically implementable algorithms since the number of parameters is generally not too large. For example, a Gaussian distribution in one dimension only requires the specification of two parameters: the mean and standard deviation. Yet with these two parameters a probability distribution is defined over an infinite number of values! This is an example of how parametric filters exploit structure for efficiency.

[1] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005

14.1 Gaussian Distribution

The Gaussian distribution (also referred to as a normal distribution) is likely the most commonly used parametric distribution in many disciplines, including robotics. The probability density function for a one-dimensional (univariate) Gaussian distribution is given by:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}}, \tag{14.1}$$

where the parameters are the mean μ and standard deviation σ (the quantity σ^2 is referred to as the variance). A shorthand notation for saying that a random variable X is distributed according to a Gaussian (normal) distribution is $X \sim \mathcal{N}(\mu, \sigma^2)$. For the multi-dimensional case, the multivariate Gaussian distribution is defined by the probability density function:

$$p(\boldsymbol{x}) = \frac{1}{\sqrt{\det(2\pi\boldsymbol{\Sigma})}} \exp\big(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\big), \tag{14.2}$$

where $\boldsymbol{x} \in \mathbb{R}^n$ and the parameters are the mean $\boldsymbol{\mu} \in \mathbb{R}^n$ and the covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{n\times n}$. Again, a shorthand to say a random variable X is distributed according to the multivariate Gaussian distribution is $X \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. These distributions are represented graphically for the univariate and bivariate case in Figure 14.1. These distributions exhibit several important properties which

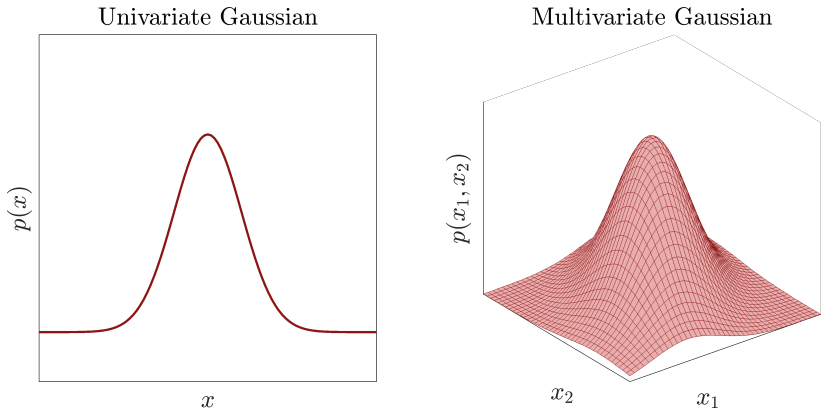


Figure 14.1: Univariate and multivariate Gaussian distributions.

make them particularly attractive for algorithm development:

1. The affine transformation of a Gaussian random variable is also a Gaussian random variable. In particular, suppose the random variable X has a multivariate Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$. Then the random variable Y resulting from an affine transformation:

   $$Y = AX + b,$$

   also has a multivariate Gaussian distribution with expected value $A\boldsymbol{\mu} + b$ and covariance $A\boldsymbol{\Sigma}A^\top$. In other words, if $X \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ then $Y \sim \mathcal{N}(A\boldsymbol{\mu} + b, A\boldsymbol{\Sigma}A^\top)$.

2. The sum of two independent Gaussian random variables is also a Gaussian random variable. In particular, suppose X_1 and X_2 have multivariate Gaussian distributions with means $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ and covariances $\boldsymbol{\Sigma}_1$ and $\boldsymbol{\Sigma}_2$. Then the random variable Y given by the sum:

$$Y = X_1 + X_2,$$

   also has a multivariate Gaussian distribution with expected value $\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2$ and covariance $\boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2$. In other words, if $X_1 \sim \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)$ and $X_2 \sim \mathcal{N}(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)$ then $Y \sim \mathcal{N}(\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2)$.

3. The product of two Gaussian random variables is also a Gaussian random variable.

14.2 Kalman Filter

The Kalman filter is an extremely well known algorithm for state estimation that leverages the Gaussian distribution for efficiency. Unlike the discrete Bayes filter from the previous chapter, this filter can be applied to problems with *continuous* states. In particular, a multivariate Gaussian distribution is used to parameterize the belief distribution over possible states, in other words the state $\boldsymbol{x}_t \sim \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$, and in long form this can be expressed as:

$$bel(\boldsymbol{x}_t) = p(\boldsymbol{x}_t) = \frac{1}{\sqrt{\det(2\pi\boldsymbol{\Sigma}_t)}} \exp\big(-\frac{1}{2}(\boldsymbol{x}_t - \boldsymbol{\mu}_t)^\top \boldsymbol{\Sigma}_t^{-1}(\boldsymbol{x}_t - \boldsymbol{\mu}_t)\big).$$

14.2.1 Assumptions

To ensure that the belief *remains* Gaussian after the prediction and measurement update steps of the filtering algorithm, several additional assumptions are required. First, it is assumed that the initial belief $bel(\boldsymbol{x}_0)$ is Gaussian with $\boldsymbol{x}_0 \sim \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$ and that the state transition model is linear and is given by:

$$\boldsymbol{x}_t = A_t \boldsymbol{x}_{t-1} + B_t \boldsymbol{u}_t + \boldsymbol{\epsilon}_t, \tag{14.3}$$

where $\boldsymbol{x}_{t-1}$ is the previous state, $\boldsymbol{u}_t$ is the most recent control input, and $\boldsymbol{\epsilon}_t$ is an independent *process noise* that is normally distributed with $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{R}_t)$. Because of the properties of the Gaussian distribution presented earlier, this affine model preserves the Gaussian structure. In particular, the state transition model can be explicitly written as:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) = \frac{1}{\sqrt{\det(2\pi\boldsymbol{R}_t)}} \exp\big(-\frac{1}{2}(\boldsymbol{x}_t - A_t\boldsymbol{x}_{t-1} - B_t\boldsymbol{u}_t)^\top \boldsymbol{R}_t^{-1}(\boldsymbol{x}_t - A_t\boldsymbol{x}_{t-1} - B_t\boldsymbol{u}_t)\big).$$

In other words, this can be expressed in shorthand as $\boldsymbol{x}_t \sim \mathcal{N}(A_t\boldsymbol{x}_{t-1} + B_t\boldsymbol{u}_t, \boldsymbol{R}_t)$.

Second, the measurement model is also assumed to be linear, which again preserves the Gaussian structure:

$$\boldsymbol{z}_t = C_t \boldsymbol{x}_t + \boldsymbol{\delta}_t, \tag{14.4}$$

where $\boldsymbol{\delta}_t$ is an independent measurement noise that is normally distributed with $\mathcal{N}(\mathbf{0}, \boldsymbol{Q}_t)$. Again, this implies the measurement model can be expressed probabilistically as:

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_t) = \frac{1}{\sqrt{\det(2\pi \boldsymbol{Q}_t)}} \exp\Big(-\frac{1}{2}(\boldsymbol{z}_t - C_t \boldsymbol{x}_t)^\top \boldsymbol{Q}_t^{-1} (\boldsymbol{z}_t - C_t \boldsymbol{x}_t)\Big),$$

and in shorthand as $\boldsymbol{z}_t \sim \mathcal{N}(C_t \boldsymbol{x}_t, \boldsymbol{Q}_t)$.

To summarize, if the belief is modeled as a Gaussian distribution and the state transition and measurement models are both linear with Gaussian noise, then the Bayes filter updates can be applied and the belief will always remain Gaussian (i.e. the prediction and measurement correction steps will not warp or alter the *structure* of the belief distribution)! This results in a very practically efficient algorithm since now only the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ need to be updated by the algorithm.

14.2.2 *Algorithm*

The Kalman filter algorithm is a recursive Bayes filter whose prediction and measurement correction steps take on a special form due to the structure of the Gaussian belief distributions and the assumptions listed above. In particular, the Kalman filter algorithm is given in Algorithm 6.

Algorithm 6: Kalman Filter Algorithm

Data: $\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t$
Result: $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$
$\bar{\boldsymbol{\mu}}_t = A_t \boldsymbol{\mu}_{t-1} + B_t \boldsymbol{u}_t$
$\bar{\boldsymbol{\Sigma}}_t = A_t \boldsymbol{\Sigma}_{t-1} A_t^T + \boldsymbol{R}_t$
$K_t = \bar{\boldsymbol{\Sigma}}_t C_t^T (C_t \bar{\boldsymbol{\Sigma}}_t C_t^T + \boldsymbol{Q}_t)^{-1}$
$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t + K_t(\boldsymbol{z}_t - C_t \bar{\boldsymbol{\mu}}_t)$
$\boldsymbol{\Sigma}_t = (I - K_t C_t)\bar{\boldsymbol{\Sigma}}_t$
return $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$

In this algorithm, the first two steps define the predicted mean $\bar{\boldsymbol{\mu}}_t$ and covariance $\bar{\boldsymbol{\Sigma}}_t$, and the remaining steps perform the measurement correction. The matrix K_t that is computed for the measurement correction is typically referred to as the *Kalman gain*.

14.2.3 *Practical Considerations*

Due to the exploitation of the Gaussian distribution, the Kalman filter is a computationally efficient algorithm that can handle continuous state values. However, the consideration of Gaussian beliefs also restricts the flexibility of the probabilistic model. In particular, the belief is forced to be unimodal which may limit performance. Additionally, the assumption about the linearity of the state transition and measurement models may not be very accurate for some robots

and certain sensors, which can make the Kalman filter not perform well for some robotics applications.

14.2.4 Derivation

While it is possible to derive the Kalman filter algorithm by evaluating the Bayes filter updates from the previous chapter (i.e. computing the integral of $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)p(\boldsymbol{x}_{t-1}$, etc.), it is more intuitive to directly leverage the properties of Gaussians presented in the preceding section. First, from the prior belief distribution $bel(\boldsymbol{x}_{t-1}) \sim \mathcal{N}(\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1})$ the predicted belief $\overline{bel}(\boldsymbol{x}_{t-1})$ can be computed by using the affine transformation property of Gaussian random variables and the sum of two independent Gaussian random variables property. Specifically, these properties can be applied to the assumed linear state transition model (14.3) to give:

$$\bar{\boldsymbol{\mu}}_t = A_t \boldsymbol{\mu}_{t-1} + B_t \boldsymbol{u}_t + \mathbf{0},$$

where the $\mathbf{0}$ is the mean of the independent noise $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{R}_t)$. The covariance properties can similarly be used to give:

$$\bar{\boldsymbol{\Sigma}}_t = A_t \boldsymbol{\Sigma}_{t-1} A_t^\top + \boldsymbol{R}_t.$$

For the measurement update it is possible to again use the properties of Gaussians to simplify the derivation of the Kalman filter measurement correction step. In particular, that the product of two Gaussians is also Gaussian. In fact, the product of the two Gaussians:

$$bel(x_t) = p(\boldsymbol{z}_t \mid \boldsymbol{x}_t)\overline{bel}(\boldsymbol{x}_t) = \mathcal{N}(C_t \boldsymbol{x}_t, \boldsymbol{Q}_t)\mathcal{N}(\bar{\boldsymbol{\mu}}_t, \bar{\boldsymbol{\Sigma}}_t),$$

can be expressed as:

$$bel(x_t) = \eta \exp\big(-\frac{1}{2} J_t\big),$$

where η is a normalization constant and:

$$J_t = (\boldsymbol{z}_t - C_t \boldsymbol{x}_t)^\top \boldsymbol{Q}_t^{-1} (\boldsymbol{z}_t - C_t \boldsymbol{x}_t) + (\boldsymbol{x}_t - \bar{\boldsymbol{\mu}}_t)^\top \bar{\boldsymbol{\Sigma}}_t^{-1} (\boldsymbol{x}_t - \bar{\boldsymbol{\mu}}_t).$$

To determine the mean $\boldsymbol{\mu}_t$ and covariance $\boldsymbol{\Sigma}_t$ of this new Gaussian, one simple approach is just take the first and second derivative of J_t with respect to $\boldsymbol{x}_t$. The mean is found where the first derivative is zero, and the covariance is the (inverse) of the constant second derivative:

$$\begin{aligned} 0 &= -C_t^\top \boldsymbol{Q}_t^{-1} (\boldsymbol{z}_t - C_t \boldsymbol{\mu}_t) + \bar{\boldsymbol{\Sigma}}_t^{-1} (\boldsymbol{\mu}_t - \bar{\boldsymbol{\mu}}_t), \\ \boldsymbol{\Sigma}_t^{-1} &= C_t^\top \boldsymbol{Q}_t^{-1} C_t + \bar{\boldsymbol{\Sigma}}_t^{-1}. \end{aligned}$$

Through some algebraic manipulation the mean can be written in terms of the covariance $\boldsymbol{\Sigma}_t$:

$$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t + \boldsymbol{\Sigma}_t C_t^\top \boldsymbol{Q}_t^{-1} (\boldsymbol{z}_t - C_t \bar{\boldsymbol{\mu}}_t),$$

and of course the covariance can be written as:

$$\boldsymbol{\Sigma}_t = (C_t^\top \boldsymbol{Q}_t^{-1} C_t + \bar{\boldsymbol{\Sigma}}_t^{-1})^{-1}.$$

While it is technically possible to stop here, this is not quite the form of the Kalman filter equations. In particular a few more algebraic steps are needed, based on the matrix inversion lemma result:

$$(C_t^\top \boldsymbol{Q}_t^{-1} C_t + \bar{\boldsymbol{\Sigma}}_t^{-1})^{-1} = \bar{\boldsymbol{\Sigma}}_t - \bar{\boldsymbol{\Sigma}}_t C_t^\top (C_t \bar{\boldsymbol{\Sigma}}_t C_t^\top + \boldsymbol{Q}_t)^{-1} C_t \bar{\boldsymbol{\Sigma}}_t.$$

By choosing to define the Kalman gain as $K_t = \bar{\boldsymbol{\Sigma}}_t C_t^T (C_t \bar{\boldsymbol{\Sigma}}_t C_t^T + \boldsymbol{Q}_t)^{-1}$ it can be seen that the covariance can be expressed as:

$$\boldsymbol{\Sigma}_t = \bar{\boldsymbol{\Sigma}}_t - K_t C_t \bar{\boldsymbol{\Sigma}}_t,$$

Through some additional algebra, the expression for the mean can also be expressed in terms of the Kalman gain and simplified to its final form:

$$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t + K_t(z_t - C_t \bar{\boldsymbol{\mu}}_t).$$

For more details on the algebraic steps see Thrun et al.[2].

[2] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005

14.3 Extended Kalman Filter (EKF)

The extended Kalman filter (EKF) is a modified version of the Kalman filter that revisits the assumption of linearity for the state transition and measurement models. This extension still exploits the Gaussian distribution to represent the belief in a computationally efficient parametric way, but by generalizing to more complex models the EKF can be applied to a wider variety of robotics state estimation and localization problems. In particular, the EKF considers general nonlinear state transition and measurement models defined as:

$$\begin{aligned} \boldsymbol{x}_t &= g(\boldsymbol{u}_t, \boldsymbol{x}_{t-1}) + \boldsymbol{\epsilon}_t, \\ \boldsymbol{z}_t &= h(\boldsymbol{x}_t) + \boldsymbol{\delta}_t, \end{aligned} \tag{14.5}$$

where again $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{R}_t)$ and $\boldsymbol{\delta}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}_t)$ are normally distributed noise terms.

The EKF handles these nonlinear functions by taking advantage of their first order Taylor series expansions (which are linear functions, similar to those used in the Kalman filter). In particular the Taylor series expansion of the state transition model g is performed about the *most likely state* from the current belief distribution, which is the expected value $\boldsymbol{\mu}_{t-1}$:

$$g(\boldsymbol{u}_t, \boldsymbol{x}_{t-1}) \approx g(\boldsymbol{u}_t, \boldsymbol{\mu}_{t-1}) + G_t(\boldsymbol{x}_{t-1} - \boldsymbol{\mu}_{t-1}),$$

where $G_t = \nabla_{\boldsymbol{x}} g(\boldsymbol{u}_t, \boldsymbol{\mu}_{t-1})$ is the Jacobian of g evaluated at $\boldsymbol{\mu}_{t-1}$. From this linear approximation the state transition model can be expressed as:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) = \frac{1}{\sqrt{\det(2\pi \boldsymbol{R}_t)}} \exp\Big(-\frac{1}{2} \Delta \boldsymbol{x}_t^\top \boldsymbol{R}_t^{-1} \Delta \boldsymbol{x}_t\Big),$$

where

$$\Delta \boldsymbol{x}_t = \boldsymbol{x}_t - g(\boldsymbol{u}_t, \boldsymbol{\mu}_{t-1}) - G_t(\boldsymbol{x}_{t-1} - \boldsymbol{\mu}_{t-1}).$$

From this result, the linear predictions that are used in the Kalman filter algorithm can be replaced by the nonlinear generalizations:

$$\begin{aligned}\bar{\boldsymbol{\mu}}_t &= g(\boldsymbol{u}_t, \boldsymbol{\mu}_{t-1}),\\ \bar{\boldsymbol{\Sigma}}_t &= G_t \boldsymbol{\Sigma}_{t-1} G_t^T + \boldsymbol{R}_t.\end{aligned}$$

As can be seen the prediction of the new mean is simply an evaluation of the nonlinear function g, and the updated covariance is very similar to the Kalman filter but leverages the Jacobian G_t.

A very similar procedure is used for the measurement corrections. The measurement model is also Taylor series expanded to yield (this time about the *predicted* point $\bar{\boldsymbol{\mu}}_t$):

$$h(\boldsymbol{x}_t) \approx h(\bar{\boldsymbol{\mu}}_t) + H_t(\boldsymbol{x}_t - \bar{\boldsymbol{\mu}}_t),$$

where $H_t = \nabla_{\boldsymbol{x}} h(\bar{\boldsymbol{\mu}}_t)$ is the Jacobian of h evaluated at $\bar{\boldsymbol{\mu}}_t$. The measurement model can then be expressed using this approximation as:

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_t) = \frac{1}{\sqrt{\det(2\pi \boldsymbol{Q}_t)}} \exp\Big(-\frac{1}{2} \Delta \boldsymbol{z}_t^\top \boldsymbol{Q}_t^{-1} \Delta \boldsymbol{z}_t\Big),$$

where $\Delta \boldsymbol{z}_t = \boldsymbol{z}_t - h(\bar{\boldsymbol{\mu}}_t) - H_t(\boldsymbol{x}_t - \bar{\boldsymbol{\mu}}_t)$. From this result the measurement correction in the EKF can be shown to be similar to the Kalman filter, but where the Jacobians H_t are used:

$$\begin{aligned}\boldsymbol{\mu}_t &= \bar{\boldsymbol{\mu}}_t + K_t(\boldsymbol{z}_t - h(\bar{\boldsymbol{\mu}}_t)),\\ \boldsymbol{\Sigma}_t &= (I - K_t H_t)\bar{\boldsymbol{\Sigma}}_t,\end{aligned}$$

where the Kalman gain is $K_t = \bar{\boldsymbol{\Sigma}}_t H_t^T (H_t \bar{\boldsymbol{\Sigma}}_t H_t^T + \boldsymbol{Q}_t)^{-1}$.

14.3.1 Algorithm

The extended Kalman filter algorithm is quite similar to the Kalman filter algorithm outlined in Algorithm 6. In particular the main differences are that the updates use the nonlinear functions and their Jacobians rather than assuming strictly linear models. The EKF algorithm is outlined in Algorithm 7.

14.3.2 Practical Considerations

The extended Kalman filter can provide more accurate results than the Kalman filter in many applications due to its ability to consider more general nonlinear models. However, the approximation of the nonlinear models by a Taylor series expansion can lead to the filter to diverge if the approximation is not accurate enough. Additionally, the EKF still suffers from the same unimodal modeling assumption as the Kalman filter since the beliefs are still represented by a single Gaussian distribution.

Algorithm 7: Extended Kalman Filter Algorithm

Data: $\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t$
Result: $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$
$\bar{\boldsymbol{\mu}}_t = g(\boldsymbol{u}_t, \boldsymbol{\mu}_{t-1})$
$\bar{\boldsymbol{\Sigma}}_t = G_t \boldsymbol{\Sigma}_{t-1} G_t^T + \boldsymbol{R}_t$
$K_t = \bar{\boldsymbol{\Sigma}}_t H_t^T (H_t \bar{\boldsymbol{\Sigma}}_t H_t^T + \boldsymbol{Q}_t)^{-1}$
$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t + K_t(\boldsymbol{z}_t - h(\bar{\boldsymbol{\mu}}_t))$
$\boldsymbol{\Sigma}_t = (I - K_t H_t)\bar{\boldsymbol{\Sigma}}_t$
return $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$

14.4 Unscented Kalman Filter

The unscented Kalman filter (UKF) is another variation of the Kalman filter (still uses Gaussian distribution to parameterize the belief). This filter is also similar to the EKF in that it can handle nonlinear state transition and measurement models. However, this filter improves upon the EKF by not relying on Taylor series expansions, which can cause filter divergence due to approximation errors. This is accomplished by representing the Gaussian through a set of *sigma-points* that are transformed through the nonlinear functions. Once each sigma-point has been updated, a new Gaussian distribution is computed to represent the updated belief.

14.5 Exercises

14.5.1 EKF Localization

Complete *Problem 1: EKF Localization* located in the online repository:

```
https://github.com/PrinciplesofRobotAutonomy/AA274A_HW4,
```

where you will implement an EKF for robot localization based on linear feature extraction and a map of known features.

15
Nonparametric Filters

Previous chapters introduced several algorithms for robot localization and state estimation that are based on a probabilistic framework. In particular, the Bayes filter was first introduced as a fundamental approach to the problem, which uses a probabilistic state transition model and a measurement model to recursively update a belief distribution over possible states. A set of tractable implementations of the Bayes filter that model the belief distribution in a *parametric* way, for example using Gaussian distributions, was then presented (in particular the Kalman and extended Kalman filters). These filters leverage the structure of the parametric belief distribution to provide a computationally efficient approach to dealing with continuous state spaces (which have an infinite number of states). For example the Gaussian distribution represents a continuous distribution through a *finite* set of parameters: the mean and covariance. However there are also other implementations of Bayes filter that can be efficiently used in continuous state spaces that are *non-parametric*.

Nonparametric Filters

[1] In contrast to parametric filters, *non-parametric* filters do not make assumptions on the structure of the belief distribution. This can be a desirable property for applications in robotics where rigid structures in the belief distribution may result in poor performance. A classic example is that the Gaussian distributions used in the Kalman filter and EKF are unimodal, which cannot express the possibility that two distinct "high probability" states might exist at the same time. Non-parametric filters on the other hand generally represent the belief distribution in an unstructured way, for example through a finite number of samples drawn from the distribution, which allows for more expressive distributions. This chapter introduces two main approaches for non-parametric filtering: the *histogram filter* and the *particle filter*.

[1] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005

15.1 Histogram Filter

The histogram filter is essentially a modification of the discrete Bayes filter presented earlier to work in continuous state spaces. In particular, the continuous state space is decomposed into a finite number of regions and the belief is represented over the discretized space by collecting the finite number of probabilities of the state being in each discretized region.

In particular for the random state variable X_t, the continuous state space $\text{dom}(X_t)$ is decomposed into a finite set of regions (often called *bins* in the context of histogram filters):

$$\text{dom}(X_t) = \boldsymbol{x}_{1,t} \cup \boldsymbol{x}_{2,t} \cup ... \cup \boldsymbol{x}_{K,t}, \tag{15.1}$$

where $\boldsymbol{x}_{k,t}$ is the k-th "bin". For example, if the one-dimensional random variable X could take on values in the interval $[a, b]$ then one possible decomposition would be to split the interval into a finite number of sub-intervals with equal width. The belief distribution is then defined in non-parametric way by simply specifying a probability $p_{k,t}$ to each bin $\boldsymbol{x}_{k,t}$. A probability density function can then be defined in a piecewise manner:

$$p(\boldsymbol{x}_t) = \frac{p_{k,t}}{|\boldsymbol{x}_{k,t}|}, \quad \boldsymbol{x}_t \in \boldsymbol{x}_{k,t}, \tag{15.2}$$

where $|\boldsymbol{x}_{k,t}|$ denotes the "area" or "volume" of the bin. This definition implies that the probability that the random variable X_t takes on *any* value in the bin $\boldsymbol{x}_{k,t}$ is equal to $p_{k,t}$.

The prediction and measurement update steps of the Bayes filter are then accomplished by also discretizing the state transition and measurement models by computing a representative "mean" state for each bin:

$$\hat{\boldsymbol{x}}_{k,t} = |\boldsymbol{x}_{k,t}|^{-1} \int_{\boldsymbol{x}_{k,t}} x_t d\boldsymbol{x}_t. \tag{15.3}$$

The state transition model $p(\boldsymbol{x}_{k,t} \mid \boldsymbol{u}_t, \boldsymbol{x}_{i,t-1})$ that defines the probability of transitioning from one bin to another is then approximated in terms of the mean bin states by:

$$p(\boldsymbol{x}_{k,t} \mid \boldsymbol{u}_t, \boldsymbol{x}_{i,t-1}) \approx \eta |\boldsymbol{x}_{k,t}| p(\hat{\boldsymbol{x}}_{k,t} \mid \boldsymbol{u}_t, \hat{\boldsymbol{x}}_{i,t-1}), \tag{15.4}$$

where $p(\hat{\boldsymbol{x}}_{k,t} \mid u_t, \hat{\boldsymbol{x}}_{i,t-1})$ is the original (non-discretized) state transition model evaluated at the mean bin states $\hat{\boldsymbol{x}}$ and η is a normalization constant[2].

[2] In the case that the bin areas $|\boldsymbol{x}_{k,t}|$ are equal, these terms can be absorbed into the normalization constant.

The discretization of the measurement model is accomplished in a similar manner, with the discretized model given by:

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_{k,t}) \approx p(\boldsymbol{z}_t \mid \hat{\boldsymbol{x}}_{k,t}). \tag{15.5}$$

In other words, the measurement probability associated with a bin is approximated by the measurement probability associated with the mean bin state $\hat{\boldsymbol{x}}_{k,t}$.

After the discretization has been performed, the discrete Bayes filter algorithm from before can be directly applied by iterating over each bin and updating the probability $p_{k,t}$.

15.2 Particle Filter

The particle filter is another non-parametric filter that provides a computationally tractable implementation of the Bayes filter for continuous state spaces. This filter represents the belief distribution by a finite set of random samples called particles, which are denoted by:

$$\mathcal{X}_t \coloneqq \{\boldsymbol{x}_t^{[1]}, \boldsymbol{x}_t^{[2]}, ..., \boldsymbol{x}_t^{[M]}\}. \tag{15.6}$$

Each particle $\boldsymbol{x}_t^{[m]}$ represents a hypothesis about the true state $\boldsymbol{x}_t$, and therefore regions of the state space with more particles correspond to regions of high probability. Ideally, the particles are distributed according to the current belief:

$$\boldsymbol{x}_t^{[m]} \sim p(\boldsymbol{x}_t \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}) = bel(\boldsymbol{x}_t), \tag{15.7}$$

but theoretically this only occurs as $M \to \infty$. Instead the set of particles approximately represents the belief distribution, and in practice around $M \approx 1000$ samples tends to be sufficient (but of course this depends on the application).

The particle filter updates the belief (via a prediction and measurement correction step) by manipulating the prior set of particles $\mathcal{X}_{t-1}$ to yield a new set of particles $\mathcal{X}_t$. The prediction step is implemented by considering each particle $\boldsymbol{x}_{t-1}^{[m]}$ in the prior set $\mathcal{X}_{t-1}$ and sampling from the state transition model a new "predicted" sample $\bar{\boldsymbol{x}}_t^{[m]} \sim p(\boldsymbol{x}_t \mid \boldsymbol{u}_t, \boldsymbol{x}_{t-1}^{[m]})$. An importance factor $w_t^{[m]}$ is then defined for the predicted sample $\bar{\boldsymbol{x}}_t^{[m]}$ based on how well the observed measurement matches the prediction. Specifically, the importance factor is computed as $w_t^{[m]} = p(\boldsymbol{z}_t \mid \bar{\boldsymbol{x}}_t^{[m]})$. The predicted particles $\bar{\boldsymbol{x}}_t^{[m]}$ and their associated weights $w_t^{[m]}$ can then be collected in a new particle set $\bar{\mathcal{X}}_t$, which represents the predicted belief distribution $\overline{bel}(\boldsymbol{x}_t)$. The correction step is then accomplished by simply resampling (with replacement) a new set of M particles from the predicted set $\bar{\mathcal{X}}_t$ with a probability proportional to the weights $w_t^{[m]}$. This procedure performs the measurement correction by giving preference in the new sample set to those predicted particles that showed higher correlation to the measurement $\boldsymbol{z}_t$. The resampled points are then collected in a new set $\mathcal{X}_t$ that defines the posterior belief distribution. This algorithm is also outlined in Algorithm 8 and a few iterations of the algorithm for a simple robot localization problem are shown in Figure 15.1.

Note that the concept of resampling in the correction step can be quite important for reasons beyond just updating the belief for the measurement correction. In particular, without the resampling step over time some of the particles would drift to regions of low probability and there would be fewer particles to represent the regions of high probability. The resampling step can therefore be viewed as a probabilistic implementation of the Darwinian idea of survival of the fittest: it refocuses the particle set to regions in state space with high posterior probability. This helps from a computational efficiency standpoint because it reduces the number of particles that are needed by focusing them on the regions of the state space that matter (i.e. regions of high probability).

Algorithm 8: Particle Filter Algorithm

Data: $\mathcal{X}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t$
Result: $\mathcal{X}_t$
$\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
for $m = 1$ **to** M **do**
  Sample $\bar{\boldsymbol{x}}_t^{[m]} \sim p(\boldsymbol{x}_t \mid \boldsymbol{u}_t, \boldsymbol{x}_{t-1}^{[m]})$
  $w_t^{[m]} = p(\boldsymbol{z}_t \mid \bar{\boldsymbol{x}}_t^{[m]})$
  $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t \cup (\bar{\boldsymbol{x}}_t^{[m]}, w_t^{[m]})$
for $m = 1$ **to** M **do**
  Draw i with probability $\propto w_t^{[i]}$
  Add $\bar{\boldsymbol{x}}_t^{[i]}$ to $\mathcal{X}_t$
return $\mathcal{X}_t$

15.3 Exercises

15.3.1 Monte Carlo Localization

Complete *Extra Credit: Monte Carlo Localization* located in the online repository:

```
https://github.com/PrinciplesofRobotAutonomy/AA274A_HW4,
```

where you will implement a particle filter for localizing a robot with line feature extraction, similar to the exercise on EKF localization from the previous chapter.

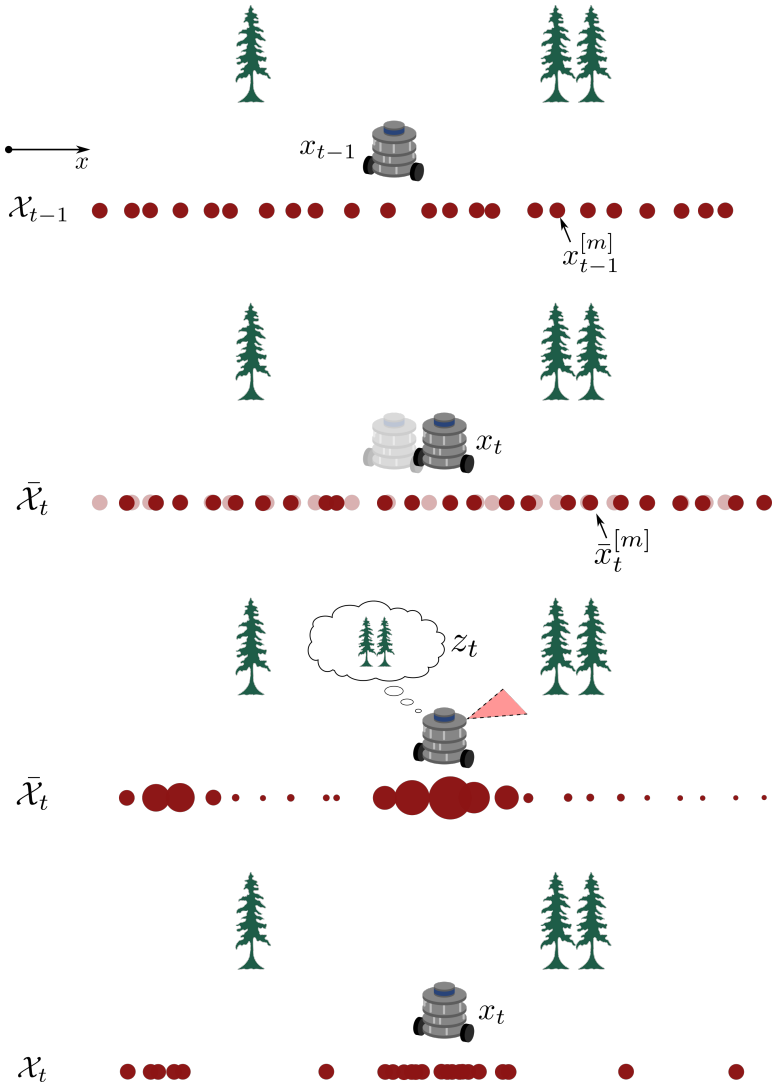


Figure 15.1: Particle filter used for robot localization. The initial set of particles are first updated according to the transition model, and then weighted according to the observation. Finally, a new set of particles is generated through weighted resampling.

16
Robot Localization

The last few chapters introduced some of the most widely used algorithms based on Bayes' filter for probabilistic robot localization and state estimation. However these fundamental algorithms still need further enhancements before application to many robot localization tasks, since in their standard form they don't incorporate a notion of a local *map*. For example, a particle filter could be applied in its original form to a problem of global localization based on Global Navigation Satellite System (GNSS) measurements, but localizing based on range measurements requires knowledge about *what* object is being ranged, and *where* that object is with respect to the local environment (i.e. the map). In this chapter a more specific definition of mobile robot localization is considered[1], namely the problem of determining the pose of a robot relative to a *given map* of the environment.

[1] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005

Robot Localization

Localization with respect to a map can be interpreted as a problem of coordinate transformation. Maps are described in a global coordinate system, which is independent of a robot's pose. Localization can then be viewed as the process of establishing a correspondence between the map coordinate system and the robot's local coordinate system. Knowing this coordinate transformation then enables the robot to express the location of objects of interest within its own coordinate frame (a necessary prerequisite for robot autonomy).

In 2D problems, knowing the pose $\boldsymbol{x}_t = [x, y, \theta]^\top$ of a robot is sufficient to establish this correspondence, and an ideal sensor would directly be able to measure this pose. However in practice no such sensor exists, and therefore *indirect* (often noisy) measurements $\boldsymbol{z}_t$ of the pose are used. Since it is almost impossible to be able to reliably estimate $\boldsymbol{x}_t$ from a single measurement $\boldsymbol{z}_t$, localization algorithms typically integrate additional data *over time* to build reliable localization estimates. For example, consider a robot located inside a building where many corridors look alike. In this case a single sensor measurement (e.g. a range scan) is usually insufficient to disambiguate the identity of the corridor from the others.

In this chapter it will be seen how this map-based localization problem can be cast in the Bayesian filtering framework, such that the algorithms from previous chapters can be leveraged.

16.1 A Taxonomy of Localization Problems

To understand the broad scope of challenges related to robot localization, it is useful to develop a brief taxonomy of localization problems. This categorization will divide localization problems along a number of important dimensions pertaining to the nature of the environment (e.g. static versus dynamic), the initial knowledge that a robot may possess, and how information about the environment is gathered (e.g. passive or active, with one robot or collaboratively with several robots).

16.1.1 Local vs. Global

Localization problems can be characterized by the type of knowledge that is available initially, which has a significant impact on what type of localization algorithm is most appropriate for the problem.

- *Position tracking* problems assume that the initial pose of the robot is known. In these types of problems only incremental updates are required (i.e. the localization error is generally always small), and therefore unimodal Gaussian filters (e.g. Kalman filters) can be efficiently applied.
- *Global localization* problems assume that the initial pose of the robot is unknown. In these scenarios the use of a unimodal parametric belief distribution cannot adequately capture the global uncertainty. Therefore it is more appropriate to use non-parametric, multi-hypothesis filters, such as the particle filter.
- The *kidnapped robot problem* is a variant of the global localization problem (i.e. unknown initial pose) where the robot can get "kidnapped" and "teleported" to some other location. This problem is more difficult than the global localization problem since the localization algorithm needs to have an awareness that sudden drastic to the robot's pose are possible. While robots are typically not "kidnapped" in practice, the consideration of this type of problem is useful for ensuring the localization algorithm is *robust*, since the ability to recover from failure is essential for truly autonomous robots. Similar to the global localization problem, these problems are often best addressed using non-parametric, multi-hypothesis filters.

16.1.2 Static vs. Dynamic

Environmental changes are another important consideration in mobile robot localization, specifically whether they are static or dynamic.

- In *static* environments the robot is the only object that moves. Static environments are generally much easier to perform localization in.
- *Dynamic* environments possess objects other than the robot whose locations or configurations change over time. This problem is usually addressed by augmenting the state vector to include the movement of dynamic entities, or by filtering the sensor data to remove the effects of environment dynamics.

16.1.3 Passive vs. Active

Information collected via measurements is crucial for robot localization. Therefore it is reasonable to consider localization problems where the robot can *explicitly* choose its actions to gather more (or more specific) information from the environment.

- *Passive localization* problems assume that the robot's motion is unrelated to its localization process.
- *Active localization* problems consider the ability of the robot to choose its actions (at least partially) to improve its understanding of the environment. For example, a robot in the corner of a room might choose to reorient itself to face the rest of the room, so it can collect environmental information as it moves along the wall. Hybrid approaches are also possible, since it may be inefficient to use active localization all of the time.

16.1.4 Single Robot vs. Multi-Robot

It is of course also possible to consider problems where several robots all gather independent information and then share that information with each other.

- *Single-robot localization* problems are the most commonly studied and utilized approach, and are often simpler because all data is collected on a single platform.
- *Multi-robot localization* problems consider teams of robots that share information in such a way that one robot's belief can be used to influence another robot's belief if the relative location between robots is known.

16.2 Robot Localization via Bayesian Filtering

The parametric (e.g. EKF) and non-parametric (e.g. particle) filters from the previous chapters are all variations of the Bayes filter. In particular they rely on a Markov process assumption and the identification of probabilistic measurement models. In this section it is shown how map-based robot localization can be cast into this framework, such that the previously discussed algorithms can be applied.

Similar to the general filtering context from the previous chapters, at time t the state is denoted by $\boldsymbol{x}_t$, the control input is denoted by $\boldsymbol{u}_t$, and the measurements are denoted by $\boldsymbol{z}_t$. For example, a differential drive robot equipped with a laser range-finder (returning a set of range measurements r_i and bearings ϕ_i), the state, control, and measurements would be:

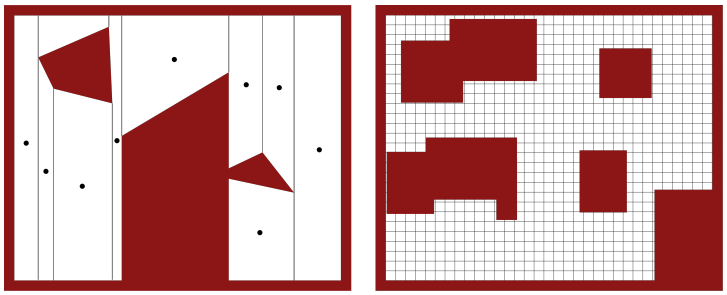
$$\boldsymbol{x}_t = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}, \quad \boldsymbol{u}_t = \begin{bmatrix} v \\ \omega \end{bmatrix}, \quad \boldsymbol{z}_t = \begin{bmatrix} r_1 \\ \phi_1 \\ \vdots \end{bmatrix}. \tag{16.1}$$

However, the critical new component is the concept of a *map* (denoted as $\boldsymbol{m}$), which is a list of objects in the environment along with their properties:

$$\boldsymbol{m} = \{m_1, m_2, \ldots, m_N\}, \tag{16.2}$$

where m_i represents the properties of a specific object. Generally there are two types of maps that will be considered, location-based maps and feature-based maps, which typically have differences in both computational efficiency and expressiveness.

For location-based maps, the index i associated with object m_i corresponds to a specific *location* (i.e. m_i are volumetric objects). For example, objects m_i in a location-based map might represent cells in a cell decomposition or grid representation of a map (see Figure 16.1). One potential disadvantage of the



Cell Decomposition Map Grid-based Map

Figure 16.1: Two examples of location-based maps, both represent the map as a set of volumetric objects (i.e. cells in these cases).

cell-based maps is that their resolution is dependent on the size of the cells, but their advantage is that they can explicitly encode information about presence (or absence) of objects in specific locations.

For feature-based maps, an index i is a feature index, and m_i contains information about the properties of that feature, including its Cartesian location. These types of maps can typically be thought of as a collection of landmarks. Figure 16.2 gives two examples of feature-based maps, one which is represented by a set of lines, and another which is represented by nodes and edges like a graph (i.e. a topological map). Feature-based maps can be more finely tuned to specific environments, for example the line-based map might make sense to use in highly structured environments such as buildings. While feature-based maps can be computationally efficient, their main disadvantage is that they typically do not capture spatial information about all potential obstacles.

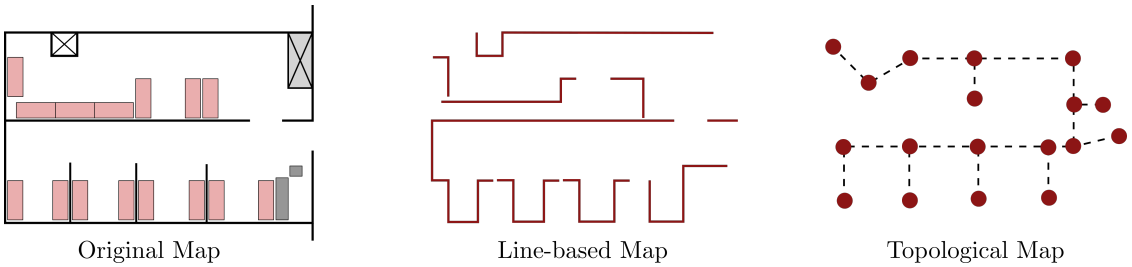


Figure 16.2: Two examples of feature-based maps.

16.2.1 State Transition Model

In the previous chapters on Bayesian filtering the probabilistic state transition model $p(\boldsymbol{x}_t \mid \boldsymbol{u}_t, \boldsymbol{x}_{t-1})$ describes the posterior distribution over the states that the robot could transition to when executing control $\boldsymbol{u}_t$ from $\boldsymbol{x}_{t-1}$. However in robot localization problems it might be important to take into account how the map $\boldsymbol{m}$ could affect the state transition since in general:

$$p(\boldsymbol{x}_t \mid \boldsymbol{u}_t, \boldsymbol{x}_{t-1}) \neq p(\boldsymbol{x}_t \mid \boldsymbol{u}_t, \boldsymbol{x}_{t-1}, \boldsymbol{m}).$$

For example, $p(\boldsymbol{x}_t \mid \boldsymbol{u}_t, \boldsymbol{x}_{t-1})$ cannot account for the fact that a robot cannot move through walls since it doesn't know that walls exist!

However, a common approximation is to make the assumption that:

$$p(\boldsymbol{x}_t \mid \boldsymbol{u}_t, \boldsymbol{x}_{t-1}, \boldsymbol{m}) \approx \eta \frac{p(\boldsymbol{x}_t \mid \boldsymbol{u}_t, \boldsymbol{x}_{t-1}) p(\boldsymbol{x}_t \mid \boldsymbol{m})}{p(\boldsymbol{x}_t)}, \tag{16.3}$$

where η is a normalization constant. This approximation can be derived from Bayes' rule by assuming that $p(\boldsymbol{m} \mid \boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) \approx p(\boldsymbol{m} \mid \boldsymbol{x}_t)$ (which is a tight approximation under high update rates). More specifically:

$$\begin{aligned}
p(\boldsymbol{x}_t \mid \boldsymbol{u}_t, \boldsymbol{x}_{t-1}, \boldsymbol{m}) &= \frac{p(\boldsymbol{m} | \boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)}{p(\boldsymbol{m} \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)}, \\
&= \eta' p(\boldsymbol{m} | \boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t), \\
&\approx \eta' p(\boldsymbol{m} | \boldsymbol{x}_t) p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t), \\
&= \eta \frac{p(\boldsymbol{x}_t \mid \boldsymbol{u}_t, \boldsymbol{x}_{t-1}) p(\boldsymbol{x}_t \mid \boldsymbol{m})}{p(\boldsymbol{x}_t)},
\end{aligned}$$

where η' and η are normalization constants (such that the total probability density integrates to one).

In this approximation the term $p(\boldsymbol{x}_t \mid \boldsymbol{m})$ is the state probability conditioned on the map which can be thought of as describing the "consistency" of state with respect to the map. The approximation (16.3) can therefore be viewed as making a probabilistic guess using the original state transition model (without map knowledge), and then using the consistency term $p(\boldsymbol{x}_t \mid \boldsymbol{m})$ to check the plausibility of the new state $\boldsymbol{x}_t$ given the map.

16.2.2 Measurement Model

The probabilistic measurement model $p(z_t \mid x_t)$ from previous chapters also needs to be modified to take map information into account. This new measurement model can simply be expressed as $p(z_t \mid x_t, m)$ (i.e. measurement is also conditioned on the map). This is obviously important because the local measurements can have significant influence from the environment. For example a range measurement is dependent on what object is currently in the line of sight.

Additionally, since the suite of sensors on a robot may generate more than one measurement when queried, it is also common to make another measurement model assumption for simplicity. Suppose K measurements are taken at time t, such that:

$$z_t = \begin{bmatrix} z_t^1 \\ \vdots \\ z_t^K \end{bmatrix}.$$

Then it can often be assumed that each of the K measurements are conditionally independent from each other (i.e. when conditioned on x_t and m the probability of measuring z_t^k is independent from the other measurements). With this assumption the probabilistic measurement model can be expressed as:

$$p(z_t \mid x_t, m) = \prod_{k=1}^{K} p(z_t^k \mid x_t, m). \tag{16.4}$$

16.3 Markov Localization

With the probabilistic state transition and measurement models that include the map, the Bayes' filter can be directly modified as shown in Algorithm 9. As can

Algorithm 9: Markov Localization Algorithm

Data: $bel(x_{t-1}), u_t, z_t, m$
Result: $bel(x_t)$
foreach x_t **do**
  $\overline{bel}(x_t) = \int p(x_t \mid u_t, x_{t-1}, m) bel(x_{t-1}) dx_{t-1}$
  $bel(x_t) = \eta p(z_t \mid x_t, m) \overline{bel}(x_t)$
return $bel(x_t)$

be seen, this algorithm is conceptually identical to the Bayes' filter except for the inclusion of the model m. This algorithm is referred to as the *Markov localization* algorithm, and the localization problem it is trying to solve is generally referred to as simply *Markov localization*[2].

The Markov localization algorithm can be used to address global localization, position tracking, and kidnapped robot problems, but generally some implementation details might be different. The choice for the initial (prior) belief

[2] Recall the use of the Markov property assumption in the derivation of the Bayes' filter.

distribution $bel(\boldsymbol{x}_0)$ is one such parameter that may be different depending on the type of localization problem.

Specifically, since the initial belief encodes any prior knowledge about the robot pose, the best choice of distribution depends on what (if any) knowledge is available. For example, in the position tracking problem it is assumed that an initial pose of the robot is known. Therefore choosing a (unimodal) Gaussian distribution $bel(\boldsymbol{x}_0) \sim \mathcal{N}(\bar{\boldsymbol{x}}_0, \boldsymbol{\Sigma}_0)$ with a small covariance might be a good choice. Alternatively, for a global localization problem the initial pose is not known. In this case an appropriate choice for the initial belief would be a uniform distribution $bel(\boldsymbol{x}_0) = 1/|X|$ over all possible states $\boldsymbol{x}$.

Similarly to the original Bayes' filter from previous chapters, the Markov localization algorithm 9 is generally not possible to implement in a computationally tractable way. However, practical implementations can still be developed by again leveraging some sort of structure to the belief distribution $bel(\boldsymbol{x}_t)$ (e.g. through Gaussian or particle representations). Two commonly used implementations based on specific structured beliefs will now be discussed: extended Kalman filter localization and Monte Carlo localization.

16.4 Extended Kalman Filter (EKF) Localization

The extended Kalman filter (EKF) localization algorithm is essentially equivalent to the EKF algorithm presented in previous chapters, except that it also takes the map $\boldsymbol{m}$ into account. In particular, it still makes a Gaussian belief assumption, $bel(\boldsymbol{x}_t) \sim \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$, to add structure to the filtering problem. As a brief review, the assumed state transition model is given by:

$$\boldsymbol{x}_t = g(\boldsymbol{u}_t, \boldsymbol{x}_{t-1}) + \boldsymbol{\epsilon}_t,$$

where $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{R}_t)$ is Gaussian zero-mean noise. The Jacobian G_t is again defined by $G_t = \nabla_{\boldsymbol{x}} g(\boldsymbol{u}_t, \boldsymbol{\mu}_{t-1})$, where $\boldsymbol{\mu}_{t-1}$ is the expected value of the previous belief distribution $bel(\boldsymbol{x}_{t-1})$.

The main difference in EKF localization is the assumption that a feature-based map is available, consisting of point landmarks given by:

$$\boldsymbol{m} = \{m_1, m_2, \ldots, m_N\}, \quad m_j = (m_{j,x}, m_{j,y}),$$

where N is the total number of landmarks, and each landmark m_j encapsulates the location $(m_{j,x}, m_{j,y})$ of the landmark in the global coordinate frame. Measurements $\boldsymbol{z}_t$ associated with these point landmarks at a time t are denoted by:

$$\boldsymbol{z}_t = \{\boldsymbol{z}_t^1, \boldsymbol{z}_t^2, \ldots\},$$

where $\boldsymbol{z}_t^i$ is associated with a particular landmark and is assumed to be generated by the measurement model:

$$\boldsymbol{z}_t^i = h(\boldsymbol{x}_t, j, \boldsymbol{m}) + \boldsymbol{\delta}_t,$$

where $\boldsymbol{\delta}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{Q}_t)$ is Gaussian zero-mean noise and j is the index of the map feature $m_j \in \boldsymbol{m}$ that measurement i is associated with.

One fundamental problem that now needs to be addressed is the *data association problem*, which arises due to uncertainty in which measurements are associated with which landmark. To begin addressing this problem, the correspondences are modeled through a variable $c_t^i \in \{1, \ldots, N+1\}$, which take on the values $c_t^i = j$ if measurement i corresponds to landmark j, and $c_t^i = N+1$ if measurement i has no corresponding landmark. Then, given a correspondence c_t^i of measurement i (associated with a specific landmark), the Jacobian H_t^i used in the EKF measurement correction step can be determined. Specifically, for the i-th measurement the Jacobian of the new measurement model can be computed by $H_t^{c_t^i} = \nabla_{\boldsymbol{x}} h(\bar{\boldsymbol{\mu}}_t, c_t^i, \boldsymbol{m})$, where $\bar{\boldsymbol{\mu}}_t$ is the predicted mean (that results from the EKF prediction step).

16.4.1 EKF Localization with Known Correspondences

In practice the correspondences between measurements $\boldsymbol{z}_t^i$ and landmarks m_j are generally unknown. However, it is useful from a pedagogical standpoint to first consider the case where these correspondences $\boldsymbol{c}_t = [c_t^1, \ldots]^\top$ are assumed to be *known*.

In the EKF localization algorithm given in Algorithm 10, the main difference from the original EKF filter algorithm is that multiple measurements are processed at the same time. Crucially, this is accomplished in a computationally efficient way by exploiting the conditional independence assumption (16.4) for the measurements. In fact, by exploiting this assumption and some special properties of Gaussians, the multi-measurement update can be implemented by just looping over each measurement individually and applying the standard EKF correction.

Algorithm 10: Extended Kalman Filter Localization Algorithm

Data: $\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t, \boldsymbol{c}_t, \boldsymbol{m}$
Result: $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$
$\bar{\boldsymbol{\mu}}_t = g(\boldsymbol{u}_t, \boldsymbol{\mu}_{t-1})$
$\bar{\boldsymbol{\Sigma}}_t = G_t \boldsymbol{\Sigma}_{t-1} G_t^T + \boldsymbol{R}_t$
foreach $\boldsymbol{z}_t^i$ **do**
  $j = c_t^i$
  $S_t^i = H_t^j \bar{\boldsymbol{\Sigma}}_t [H_t^j]^T + \boldsymbol{Q}_t$
  $K_t^i = \bar{\boldsymbol{\Sigma}}_t [H_t^j]^T [S_t^i]^{-1}$
  $\bar{\boldsymbol{\mu}}_t = \bar{\boldsymbol{\mu}}_t + K_t^i (\boldsymbol{z}_t^i - h(\bar{\boldsymbol{\mu}}_t, j, \boldsymbol{m}))$
  $\bar{\boldsymbol{\Sigma}}_t = (I - K_t^i H_t^j) \bar{\boldsymbol{\Sigma}}_t$
$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t$
$\boldsymbol{\Sigma}_t = \bar{\boldsymbol{\Sigma}}_t$
return $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$

16.4.2 EKF Localization with Unknown Correspondences

For EKF localization with *unknown* correspondences, the correspondence variables must also be estimated! The simplest way to determine the correspondences online is to use maximum likelihood estimation, in which the most likely value of the correspondences $\boldsymbol{c}_t$ is determined by maximizing the data likelihood:

$$\hat{\boldsymbol{c}}_t = \arg\max_{\boldsymbol{c}_t} p(\boldsymbol{z}_t \mid \boldsymbol{c}_{1:t}, \boldsymbol{m}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t})$$

In other words, the set of correspondence variables is chosen to maximize the probability of getting the current measurement given the history of correspondence variables, the map, the history of measurements, and the history of controls. By marginalizing over the current pose $\boldsymbol{x}_t$ this distribution can be written as:

$$\begin{aligned} p(\boldsymbol{z}_t \mid \boldsymbol{c}_{1:t}, \boldsymbol{m}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) &= \int p(\boldsymbol{z}_t \mid \boldsymbol{c}_{1:t}, \boldsymbol{x}_t, \boldsymbol{m}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) p(\boldsymbol{x}_t \mid \boldsymbol{c}_{1:t}, \boldsymbol{m}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) d\boldsymbol{x}_t, \\ &= \int p(\boldsymbol{z}_t \mid \boldsymbol{c}_t, \boldsymbol{x}_t, \boldsymbol{m}) \overline{bel}(\boldsymbol{x}_t) d\boldsymbol{x}_t. \end{aligned}$$

Note that the term $p(\boldsymbol{z}_t \mid \boldsymbol{c}_{1:t}, \boldsymbol{x}_t, \boldsymbol{m})$ is essentially the assumed measurement model given *known* correspondences. Then, by again leveraging the conditional independence assumption for the measurements $\boldsymbol{z}_t^i$ from (16.4), this can be written as:

$$p(\boldsymbol{z}_t \mid \boldsymbol{c}_{1:t}, \boldsymbol{m}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) = \int \prod_i p(\boldsymbol{z}_t^i \mid c_t^i, \boldsymbol{x}_t, \boldsymbol{m}) \overline{bel}(\boldsymbol{x}_t) d\boldsymbol{x}_t.$$

Importantly, each decision variable c_t^i in the maximization of this quantity shows up in separate terms of the product! Therefore it is possible to maximize each parameter independently by solving the optimization problems:

$$\hat{c}_t^i = \arg\max_{c_t^i} \int p(\boldsymbol{z}_t^i \mid c_t^i, \boldsymbol{x}_t, \boldsymbol{m}) \overline{bel}(\boldsymbol{x}_t) d\boldsymbol{x}_t.$$

This problem can be solved quite efficiently since it is assumed that the measurement models and belief distributions are Gaussian[3]. In particular, the probability distribution resulting from the integral is a Gaussian with mean and covariance:

$$\int p(\boldsymbol{z}_t^i \mid c_t^i, \boldsymbol{x}_t, \boldsymbol{m}) \overline{bel}(\boldsymbol{x}_t) d\boldsymbol{x}_t \sim \mathcal{N}(h(\bar{\boldsymbol{\mu}}_t, c_t^i, \boldsymbol{m}), H_t^{c_t^i} \bar{\boldsymbol{\Sigma}}_t [H_t^{c_t^i}]^\top + \boldsymbol{Q}_t).$$

The maximum likelihood optimization problem can therefore be expressed as:

$$\hat{c}_t^i = \arg\max_{c_t^i} \; \mathcal{N}(\boldsymbol{z}_t^i \mid \hat{\boldsymbol{z}}_t^{c_t^i}, S_t^{c_t^i}),$$

where $\hat{\boldsymbol{z}}_t^j = h(\bar{\boldsymbol{\mu}}_t, j, \boldsymbol{m})$ and $S_t^j = H_t^j \bar{\boldsymbol{\Sigma}}_t [H_t^j]^\top + \boldsymbol{Q}_t$. To solve this maximization problem, recall the definition of the Gaussian distribution:

$$\mathcal{N}(\boldsymbol{z}_t^i \mid \hat{\boldsymbol{z}}_t^j, S_t^j) = \eta \exp\Big(-\frac{1}{2} (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^j)^\top [S_t^j]^{-1} (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^j) \Big),$$

[3] Similar to the previous chapters, in this case the product of terms inside the integral will be Gaussian since both terms are Gaussian.

where η is a normalization constant. Since the exponential function is monotonically increasing and since η is a positive constant, the maximum likelihood estimation problem can be equivalently expressed as:

$$\hat{c}_t^i = \arg\min_{c_t^i} \; d_t^{i,c_t^i}, \tag{16.5}$$

where

$$d_t^{ij} = (z_t^i - \hat{z}_t^j)^\top [S_t^j]^{-1} (z_t^i - \hat{z}_t^j), \tag{16.6}$$

is referred to as the *Mahalanobis distance*.

The EKF localization algorithm with unknown correspondences is very similar to Algorithm 10, except with the addition of this maximum likelihood estimation step. This new algorithm is given in Algorithm 11.

Algorithm 11: EKF Localization Algorithm, Unknown Correspondences

Data: $\mu_{t-1}, \Sigma_{t-1}, u_t, z_t, m$
Result: μ_t, Σ_t
$\bar{\mu}_t = g(u_t, \mu_{t-1})$
$\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$
foreach z_t^i **do**
  foreach *landmark k in the map* **do**
    $\hat{z}_t^k = h(\bar{\mu}_t, k, m)$
    $S_t^k = H_t^k \bar{\Sigma}_t [H_t^k]^T + Q_t$
  $j = \arg\min_k \; (z_t^i - \hat{z}_t^k)^\top [S_t^k]^{-1} (z_t^i - \hat{z}_t^k)$
  $K_t^i = \bar{\Sigma}_t [H_t^j]^T [S_t^j]^{-1}$
  $\bar{\mu}_t = \bar{\mu}_t + K_t^i (z_t^i - \hat{z}_t^j)$
  $\bar{\Sigma}_t = (I - K_t^i H_t^j) \bar{\Sigma}_t$
$\mu_t = \bar{\mu}_t$
$\Sigma_t = \bar{\Sigma}_t$
return μ_t, Σ_t

One of the disadvantages of using the maximum likelihood estimation is that it can be brittle with respect to outliers and in cases where there are equally likely hypothesis for the correspondence. An alternative approach to estimating correspondences that is more robust to outliers is to use a *validation gate*. In this approach the Mahalanobis smallest distance d_t^{ij} must also pass a *thresholding* test:

$$(z_t^i - \hat{z}_t^j)^\top [S_t^j]^{-1} (z_t^i - \hat{z}_t^j) \leq \gamma,$$

in order for a correspondence to be created.

Example 16.4.1 (Differential Drive Robot with Range and Bearing Measurements)**.** Consider a differential drive robot with state $x = [x, y, \theta]^\top$, and suppose a sensor is available on the robot which measures the range r and bearing ϕ of landmarks $m_j \in m$ relative to the robot's local coordinate frame. Additionally,

multiple measurements corresponding to different features can be collected at each time step:

$$\boldsymbol{z}_t = \{[r_t^1, \phi_t^1]^\top, [r_t^2, \phi_t^2]^\top, \dots\},$$

where each measurement $\boldsymbol{z}_t^i$ contains the range r_t^i and bearing ϕ_t^i.

Assuming the correspondences are known, the measurement model for the range and bearing is:

$$h(\boldsymbol{x}_t, j, \boldsymbol{m}) = \begin{bmatrix} \sqrt{(m_{j,x} - x)^2 + (m_{j,y} - y)^2} \\ \text{atan2}(m_{j,y} - y, m_{j,x} - x) - \theta \end{bmatrix}. \tag{16.7}$$

The measurement Jacobian H_t^j corresponding to a measurement from landmark j is then given by:

$$H_t^j = \begin{bmatrix} -\frac{m_{j,x} - \bar{\mu}_{t,x}}{\sqrt{(m_{j,x} - \bar{\mu}_{t,x})^2 + (m_{j,y} - \bar{\mu}_{t,y})^2}} & -\frac{m_{j,y} - \bar{\mu}_{t,y}}{\sqrt{(m_{j,x} - \bar{\mu}_{t,x})^2 + (m_{j,y} - \bar{\mu}_{t,y})^2}} & 0 \\ \frac{m_{j,y} - \bar{\mu}_{t,y}}{(m_{j,x} - \bar{\mu}_{t,x})^2 + (m_{j,y} - \bar{\mu}_{t,y})^2} & -\frac{m_{j,x} - \bar{\mu}_{t,x}}{(m_{j,x} - \bar{\mu}_{t,x})^2 + (m_{j,y} - \bar{\mu}_{t,y})^2} & -1 \end{bmatrix}. \tag{16.8}$$

It is also common to assume that the covariance of the measurement noise is given by:

$$\boldsymbol{Q}_t = \begin{bmatrix} \sigma_r^2 & 0 \\ 0 & \sigma_\phi^2 \end{bmatrix},$$

where σ_r is the standard deviation of the range measurement noise and σ_ϕ is the standard deviation of the bearing measurement noise. This diagonal covariance matrix is typically used since these two measurements can be assumed to be uncorrelated.

16.5 Monte Carlo Localization (MCL)

Another approach to Markov localization is the Monte Carlo localization (MCL) algorithm. This algorithm leverages the non-parametric particle filter algorithm from the previous chapter, and is therefore much better suited to solving *global* localization problems (unlike EKF localization which only solves position tracking problems). MCL can also be used to solve the kidnapped robot problem through some small modifications, such as injecting new random particles at each step to ensure that a "particle collapse" problem does not occur.

As a brief review, the particle filter represents the belief $bel(\boldsymbol{x}_t)$ by a set of M particles:

$$\mathcal{X}_t := \{\boldsymbol{x}_t^{[1]}, \boldsymbol{x}_t^{[2]}, ..., \boldsymbol{x}_t^{[M]}\},$$

where each particle $\boldsymbol{x}_t^{[m]}$ represents a hypothesis about the true state $\boldsymbol{x}_t$. At each step of the algorithm the state transition model is used to propagate forward the particles, and then the measurement model is used to resample particles based on the measurement likelihood. This algorithm is shown in Algorithm 12, and is nearly identical to the particle filter algorithm except that the map $\boldsymbol{m}$ is used in the probabilistic state transition and measurement models.

Algorithm 12: Monte Carlo Localization Algorithm

Data: $\mathcal{X}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t, \boldsymbol{m}$
Result: $\mathcal{X}_t$
$\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
for $m = 1$ **to** M **do**
  Sample $\bar{\boldsymbol{x}}_t^{[m]} \sim p(\boldsymbol{x}_t \mid \boldsymbol{u}_t, \boldsymbol{x}_{t-1}^{[m]}, \boldsymbol{m})$
  $w_t^{[m]} = p(\boldsymbol{z}_t \mid \bar{\boldsymbol{x}}_t^{[m]}, \boldsymbol{m})$
  $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t \cup (\bar{\boldsymbol{x}}_t^{[m]}, w_t^{[m]})$
for $m = 1$ **to** M **do**
  Draw i with probability $\propto w_t^{[i]}$
  Add $\bar{\boldsymbol{x}}_t^{[i]}$ to $\mathcal{X}_t$
return $\mathcal{X}_t$

17
Simultaneous Localization and Mapping (SLAM)

The previous chapter introduced the robot localization problem, but assumed that the map $\boldsymbol{m}$ was *given*. However, in many real-world robotics applications a map might not be known ahead of time, and therefore it would need to be built on-the-fly. This problem, which involves using information about measurements $\boldsymbol{z}$ and controls $\boldsymbol{u}$ to simultaneously localize the robot in the world and build a map, is known as *simultaneous localization and mapping* (SLAM)[1].

[1] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005

Simultaneous Localization and Mapping (SLAM)

Many real-world settings are challenging for robotic autonomy because both the map and the relative pose of the robot are unknown. For example, such a situation would occur in autonomous search-and-rescue operations where a robot needs to explore an unknown environment. The SLAM problem addresses this challenge by estimating the robot pose and constructing a map of the environment at the same time, based only on measurement $\boldsymbol{z}_{1:t}$ and control $\boldsymbol{u}_{1:t}$ data.

Generally speaking there are two types of SLAM problems that can be considered. The *online* SLAM problem aims to estimate the posterior $p(\boldsymbol{x}_t, \boldsymbol{m} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t})$ over the robot's current pose $\boldsymbol{x}_t$ and the map $\boldsymbol{m}$. Alternatively, the *full* SLAM problem estimates the entire path of the robot instead of just the current position, namely $p(\boldsymbol{x}_{1:t}, \boldsymbol{m} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t})$. The difference between these two SLAM problems is demonstrated graphically in Figure 17.1. Both SLAM problems experience the same challenge: error in the pose causes error in map estimation and error in map estimation causes error in the pose estimate. In this chapter, algorithms for both the online and full SLAM problems are studied.

17.1 EKF SLAM Algorithm

One of the earliest approaches to the online SLAM problem leverages the extended Kalman filter, and is essentially an extension of the EKF localization algorithm discussed in the previous chapter. Again, the key aspects to the approach are the exploitation of Gaussian distributions to model the robot's belief

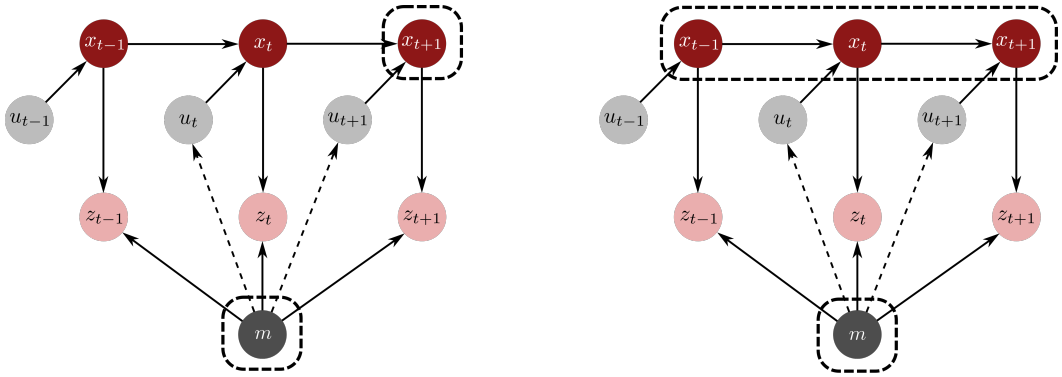


Figure 17.1: Difference between online and full SLAM, where online SLAM only estimates the current robot pose while full SLAM also estimates the robot's history.

distribution $bel(\boldsymbol{x}_t)$, and state transition and measurement models. It will also be assumed that the map is feature-based:

$$\boldsymbol{m} = \{m_1, m_2, \ldots, m_N\},$$

where m_i is the i-th landmark with coordinates $(m_{i,x}, m_{i,y})$. As in the EKF localization problem, the measurement correspondences can either be assumed to be known or unknown (more common in practice).

The main idea behind EKF SLAM is that the coordinates $(m_{i,x}, m_{i,y})$ of each landmark m_i are added, along with the robot pose $\boldsymbol{x}_t$, to an augmented state vector:

$$\boldsymbol{y}_t = \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{m}_1 \\ \vdots \\ \boldsymbol{m}_N \end{bmatrix}, \tag{17.1}$$

where $\boldsymbol{m}_i = [m_{i,x}, m_{i,y}]^\top$. With the new state vector $\boldsymbol{y}$ the online SLAM problem is to compute the posterior:

$$bel(\boldsymbol{y}_t) = p(\boldsymbol{y}_t \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}).$$

EKF SLAM approaches have the advantage of being computationally efficient such that they can be run online, and are also well understood from a theoretical perspective. They can also provide good performance when the uncertainty is low. However, their main disadvantages are that they are restricted by the Gaussian assumption to unimodal estimates, and that performance can degrade in settings with high uncertainty or when the states are not well approximated by normal distributions.

17.1.1 *State Transition and Measurement Models*

Assuming that the landmarks $m_i \in \boldsymbol{m}$ are static, the state transition model for the augmented state vector $\boldsymbol{y}$ is assumed to be given by:

$$\boldsymbol{y}_t = g(\boldsymbol{u}_t, \boldsymbol{y}_{t-1}) + \boldsymbol{\epsilon}_t, \quad \boldsymbol{\epsilon}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{R}_t),$$

where the nonlinear vector function g is defined by:

$$g(\boldsymbol{u}_t, \boldsymbol{y}_{t-1}) = \begin{bmatrix} \tilde{g}(\boldsymbol{u}_t, \boldsymbol{x}_{t-1}) \\ \boldsymbol{m}_{1,t-1} \\ \vdots \\ \boldsymbol{m}_{N,t-1} \end{bmatrix},$$

and $\tilde{g}$ is the original robot motion model (e.g. differential drive robot model). The noise covariance is also defined as:

$$\boldsymbol{R}_t = \begin{bmatrix} \tilde{\boldsymbol{R}}_t & 0 \\ 0 & 0 \end{bmatrix},$$

where $\tilde{\boldsymbol{R}}_t$ is the noise covariance associated with the original robot motion model and the rest of the matrix are zeros. The Jacobian of the augmented motion model is defined as $G_t := \nabla_y g(\boldsymbol{u}_t, \boldsymbol{\mu}_{t-1})$ where $\boldsymbol{\mu}_{t-1}$ is the expected value of the belief distribution $bel(\boldsymbol{y}_{t-1})$ at the previous time.

The measurement model is defined in the same way as the previous chapter:

$$\boldsymbol{z}_t^i = h(\boldsymbol{y}_t, j) + \boldsymbol{\delta}_t,$$

where $\boldsymbol{\delta}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}_t)$ is Gaussian zero-mean noise and j is the index of the map feature $m_j \in \boldsymbol{m}$ that measurement i is associated with. The Jacobian is also defined in the same way with $H_t^j = \nabla_y h(\bar{\boldsymbol{\mu}}_t, j)$, where $\bar{\boldsymbol{\mu}}_t$ is the predicted mean (that results from the EKF prediction step) of the distribution $\overline{bel}(\boldsymbol{y}_t)$.

17.1.2 EKF SLAM with Known Correspondences

As was the case in EKF localization, it is important to specify whether the correspondences c_t^i between the i-th measurement $\boldsymbol{z}_t^i$ and the associated landmark in the map is known. In this section an EKF SLAM algorithm will be developed which assumes the correspondences $\boldsymbol{c}_t = [c_t^1, \dots]^\top$ are *known*.

Algorithm 13 presents the EKF SLAM algorithm with known correspondences. It is almost identical to the EKF localization algorithm from last chapter, except that the state vector is augmented with the landmark positions and the positions of these landmarks are initialized when they are first seen. For this algorithm a general initialization of the belief distribution $bel(\boldsymbol{y}_0)$ is with:

$$\boldsymbol{\mu}_0 = \begin{bmatrix} \boldsymbol{x}_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \boldsymbol{\Sigma}_0 = \begin{bmatrix} \tilde{\boldsymbol{\Sigma}}_0 & 0 & \cdots & 0 \\ 0 & \infty & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \infty \end{bmatrix},$$

where:

$$\boldsymbol{x}_0 = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \tilde{\boldsymbol{\Sigma}}_0 = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix},$$

Algorithm 13: Extended Kalman Filter Online SLAM Algorithm

Data: $\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t, \boldsymbol{c}_t$
Result: $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$
$\bar{\boldsymbol{\mu}}_t = g(\boldsymbol{u}_t, \boldsymbol{\mu}_{t-1})$
$\bar{\boldsymbol{\Sigma}}_t = G_t \boldsymbol{\Sigma}_{t-1} G_t^T + \boldsymbol{R}_t$
foreach $\boldsymbol{z}_t^i$ **do**
  $j = c_t^i$
  if *landmark j never seen before* **then**
    Initialize $\begin{bmatrix} \bar{\mu}_{j,x} \\ \bar{\mu}_{j,y} \end{bmatrix}$ as expected position based on $\boldsymbol{z}_t^i$
  $S_t^i = H_t^j \bar{\boldsymbol{\Sigma}}_t [H_t^j]^T + \boldsymbol{Q}_t$
  $K_t^i = \bar{\boldsymbol{\Sigma}}_t [H_t^j]^T [S_t^i]^{-1}$
  $\bar{\boldsymbol{\mu}}_t = \bar{\boldsymbol{\mu}}_t + K_t^i (\boldsymbol{z}_t^i - h(\bar{\boldsymbol{\mu}}_t, j))$
  $\bar{\boldsymbol{\Sigma}}_t = (I - K_t^i H_t^j) \bar{\boldsymbol{\Sigma}}_t$
$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t$
$\boldsymbol{\Sigma}_t = \bar{\boldsymbol{\Sigma}}_t$
return $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$

and $\boldsymbol{x}_0$ and $\bar{\boldsymbol{\Sigma}}$ are the initial robot state and associated covariance (which is set to zero). Since the reference frame for the map can be defined arbitrarily, this initialization is used to say that the initial robot pose is known to be at the origin with certainty (and the map is built with respect to that origin). The covariance of the map positions is set to infinity to reflect that there is initially no knowledge of their position.

17.2 EKF SLAM with Unknown Correspondences

Performing EKF SLAM when the correspondences between measurements and landmarks are *unknown* poses a more challenging problem. In the EKF localization case (when the map was known), a maximum likelihood method was used to determine correspondence. A similar approach is taken for EKF SLAM, which uses a maximum likelihood approach based on the *estimated* landmark positions. The main difference is that now a mechanism for hypothesizing that a new landmark has been found is also required. The EKF SLAM with unknown correspondences algorithm is given in Algorithm 14.

As can be seen there are a couple differences between Algorithm 13 and Algorithm 14. First, the measurements $\boldsymbol{z}_k^i$ are used to *hypothesize* the position of a new landmark. The Mahalanobis distance d_t^{ik} is then computed for all currently tracked landmarks, and the hypothesized landmark is added if the distance exceeds a threshold α (i.e. $d_t^{ik} > \alpha$ for all $k = 1, \ldots, N_t$).

While this EKF-based algorithm can be used to solve the online SLAM problem without correspondences, it is not necessarily the most robust approach.

Algorithm 14: EKF Online SLAM Algorithm, Unknown Correspondences

Data: $\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t, N_{t-1}$
Result: $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$
$N_t = N_{t-1}$
$\bar{\boldsymbol{\mu}}_t = g(\boldsymbol{u}_t, \boldsymbol{\mu}_{t-1})$
$\bar{\boldsymbol{\Sigma}}_t = G_t \boldsymbol{\Sigma}_{t-1} G_t^T + \boldsymbol{R}_t$
foreach $\boldsymbol{z}_t^i$ **do**
  Hypothesize position $\begin{bmatrix} \bar{\mu}_{N_t+1,x} \\ \bar{\mu}_{N_t+1,y} \end{bmatrix}$ from $\boldsymbol{z}_t^i$
  foreach $k = 1$ **to** $N_t + 1$ **do**
    $\hat{\boldsymbol{z}}_t^k = h(\bar{\boldsymbol{\mu}}_t, k)$
    $S_t^k = H_t^k \bar{\boldsymbol{\Sigma}}_t [H_t^k]^T + \boldsymbol{Q}_t$
    $d_t^{ik} = (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^k)^\top [S_t^k]^{-1} (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^k)$
  $d_t^{i(N_t+1)} = \alpha$
  $j = \arg\min_k \; d_t^{ik}$
  $N_t = \max\{N_t, j\}$
  $K_t^i = \bar{\boldsymbol{\Sigma}}_t [H_t^j]^T [S_t^j]^{-1}$
  $\bar{\boldsymbol{\mu}}_t = \bar{\boldsymbol{\mu}}_t + K_t^i (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^j)$
  $\bar{\boldsymbol{\Sigma}}_t = (I - K_t^i H_t^j) \bar{\boldsymbol{\Sigma}}_t$
$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t$
$\boldsymbol{\Sigma}_t = \bar{\boldsymbol{\Sigma}}_t$
return $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$

In particular, extraneous measurements can result in the creation of fake landmarks, which will then propagate forward to future steps and cannot be corrected! There are several techniques to mitigate these issues, such as using outlier rejection schemes or strategies to enhance the distinctiveness of landmarks (which may require prior knowledge or assumptions). Another important disadvantage of EKF SLAM is that its computational complexity is quadratic with the number of landmarks N, but generally a large number of landmarks is required for good localization accuracy!

Example 17.2.1 (Differential Drive Robot with Range and Bearing Measurements)**.** Consider a differential drive robot with state $\boldsymbol{x} = [x, y, \theta]^\top$, and suppose a sensor is available on the robot which measures the range r and bearing ϕ of landmarks $m_j \in \boldsymbol{m}$ relative to the robot's local coordinate frame. Additionally, multiple measurements corresponding to different features can be collected at each time step:

$$\boldsymbol{z}_t = \{[r_t^1, \phi_t^1]^\top, [r_t^2, \phi_t^2]^\top, \dots\},$$

where each measurement $\boldsymbol{z}_t^i$ contains the range r_t^i and bearing ϕ_t^i.

For the SLAM problem, the augmented state $\boldsymbol{y}_t$ is defined as:

$$\boldsymbol{y}_t = \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{m}_1 \\ \vdots \\ \boldsymbol{m}_N \end{bmatrix} = \begin{bmatrix} x & y & \theta & m_{1,x} & m_{1,y} & \dots & m_{N,x} & m_{N,y} \end{bmatrix}^\top.$$

Assuming the correspondences are known, the measurement model for the range and bearing is:

$$h(\boldsymbol{y}_t, j) = \begin{bmatrix} \sqrt{(m_{j,x} - x)^2 + (m_{j,y} - y)^2} \\ \text{atan2}(m_{j,y} - y, m_{j,x} - x) - \theta \end{bmatrix}. \tag{17.2}$$

The measurement Jacobian H_t^j corresponding to a measurement from landmark j is then given by:

$$H_t^j = \begin{bmatrix} -\frac{\bar{\mu}_{j,x} - \bar{\mu}_{t,x}}{\sqrt{q_{t,j}}} & -\frac{\bar{\mu}_{j,y} - \bar{\mu}_{t,y}}{\sqrt{q_{t,j}}} & 0 & 0 & \dots & 0 & \frac{\bar{\mu}_{j,x} - \bar{\mu}_{t,x}}{\sqrt{q_{t,j}}} & \frac{\bar{\mu}_{j,y} - \bar{\mu}_{t,y}}{\sqrt{q_{t,j}}} & 0 & \dots \\ \frac{\bar{\mu}_{j,y} - \bar{\mu}_{t,y}}{q_{t,j}} & -\frac{\bar{\mu}_{j,x} - \bar{\mu}_{t,x}}{q_{t,j}} & -1 & 0 & \dots & 0 & -\frac{\bar{\mu}_{j,y} - \bar{\mu}_{t,y}}{q_{t,j}} & \frac{\bar{\mu}_{j,x} - \bar{\mu}_{t,x}}{q_{t,j}} & 0 & \dots \end{bmatrix}, \tag{17.3}$$

where:

$$q_{t,j} := (\bar{\mu}_{j,x} - \bar{\mu}_{t,x})^2 + (\bar{\mu}_{j,y} - \bar{\mu}_{t,y})^2,$$

and $\bar{\mu}_{j,x}$ and $\bar{\mu}_{j,y}$ are the estimate of the x and y coordinates of landmark m_j from $\bar{\boldsymbol{\mu}}_t$.

With both a range and bearing measurement, the *expected* position of landmark m_j is given by:

$$\begin{bmatrix} \bar{\mu}_{j,x} \\ \bar{\mu}_{j,y} \end{bmatrix} = \begin{bmatrix} \bar{\mu}_{t,x} \\ \bar{\mu}_{j,y} \end{bmatrix} + \begin{bmatrix} r_t^i \cos(\phi_t^i + \bar{\mu}_{t,\theta}) \\ r_t^i \sin(\phi_t^i + \bar{\mu}_{t,\theta}) \end{bmatrix}.$$

This can be used in the known-correspondence EKF SLAM algorithm (Algorithm 13) to initialize the landmark position and can be used in the unknown-correspondence case (Algorithm 14) to hypothesize the position of new landmarks.

17.3 Particle SLAM Algorithm

Another approach to the robot SLAM problem is to leverage the non-parametric particle filter. In fact, particle SLAM can be used to solve the *full* SLAM problem, unlike EKF SLAM which only solves the online SLAM problem. Specifically, the full SLAM problem is to estimate the posterior distribution $p(\boldsymbol{x}_{1:t}, \boldsymbol{m} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t})$, which includes the full robot path $\boldsymbol{x}_{1:t}$ up to time t and the map $\boldsymbol{m}$. Similar to the EKF SLAM case, the robot state $\boldsymbol{x}_{1:t}$ and map feature positions $\boldsymbol{m}$ are combined into an augmented state vector $\boldsymbol{y}_{1:t}$ as in (17.1).

A naïve implementation of the particle filter in the context of full SLAM would be computationally intractable, since the number of particles required

to belief distribution would be extremely large. However, the key insight that makes this approach tractable is that the posterior over the map elements is *conditionally independent* given the true path of the robot. Therefore the mapping component to the problem can be split up into separate problems, corresponding to each feature in the map! Splitting the problem in this way makes the overall problem much easier to solve.

Overall, particle filter SLAM approaches can be used with *any* noise distribution and can express multimodal beliefs since they are non-parametric. Additionally, in practice they can be relatively easy to implement and can also be more robust to data association errors. Their main disadvantages are that they typically do not scale well to large scale problems (too many particles are needed), and that without enough particles convergence may not occur.

17.3.1 Factoring the Posterior

The key insight of particle SLAM that makes it a computationally tractable algorithm is that the posterior $p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t})$ can be factored as:

$$p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) = p(\boldsymbol{x}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) \prod_{n=1}^{N} p(\boldsymbol{m}_n \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}), \tag{17.4}$$

where $\boldsymbol{m}_n$ is the n-th feature in the map $\boldsymbol{m}$, the term $p(\boldsymbol{x}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t})$ is referred to as the *path posterior*, and the terms $p(\boldsymbol{m}_n \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t})$ are referred to as the *feature posteriors*.

This factorization can be derived by first using Bayes' rule

$$p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) = p(\boldsymbol{x}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}),$$

and then noting that since the feature posterior is conditioned on $\boldsymbol{x}_{1:t}$, the dependence on $\boldsymbol{u}_{1:t}$ is redundant:

$$p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) = p(\boldsymbol{x}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}).$$

Now the feature posterior $p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t})$ can be explored in more detail. In particular two cases can be considered for each landmark $\boldsymbol{m}_n$: the case when the measurement at time t is not associated with n and the case when it is:

$$p(\boldsymbol{m}_n \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}) = \begin{cases} p(\boldsymbol{m}_n \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}), & c_t \neq n, \\ \frac{p(\boldsymbol{z}_t \mid \boldsymbol{m}_n, \boldsymbol{x}_t, c_t) p(\boldsymbol{m}_n \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1})}{p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t-1}, c_{1:t})}, & c_t = n, \end{cases}$$

where in the second case Bayes' rule was applied. It is now possible to show the result (17.4) by induction. First, suppose that:

$$p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}) = \prod_{n=1}^{N} p(\boldsymbol{m}_n \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}).$$

Then, using Bayes' rule at time t:

$$p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}) = \frac{p(\boldsymbol{z}_t \mid \boldsymbol{m}, \boldsymbol{x}_t, c_t) p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1})}{p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t-1}, c_{1:t})},$$
$$= \frac{p(\boldsymbol{z}_t \mid \boldsymbol{m}, \boldsymbol{x}_t, c_t)}{p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t-1}, c_{1:t})} \prod_{n=1}^{N} p(\boldsymbol{m}_n \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}).$$

Next, applying the analysis above for the cases where $c_t \neq n$ and $c_t = n$:

$$p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}) = p(\boldsymbol{m}_{c_t} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}) \prod_{n \neq c_t} p(\boldsymbol{m}_n \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}),$$
$$= \prod_{n=1}^{N} p(\boldsymbol{m}_n \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}).$$

17.3.2 Fast SLAM with Known Correspondences

The particle SLAM algorithm referred to as Fast SLAM uses the factorization of the posterior $p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t})$ in (17.4) to decompose the full SLAM problem into more manageable sub-problems. Specifically, the path posterior $p(\boldsymbol{x}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t})$ is estimated using a particle filter and the feature posteriors $p(\boldsymbol{m}_n \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t})$ are estimated by EKFs conditioned on the robot path $\boldsymbol{x}_{1:t}$ (i.e. there is a separate EKF for each feature $\boldsymbol{m}_n$).

Accordingly, the set of particles is given as:

$$\mathcal{Y}_t \coloneqq \{Y_t^{[1]}, Y_t^{[2]}, ..., Y_t^{[M]}\},$$

where the k-th particle is defined by:

$$Y_t^{[k]} = \{\boldsymbol{x}_t^{[k]}, \boldsymbol{\mu}_{1,t}^{[k]}, \boldsymbol{\Sigma}_{1,t}^{[k]}, \ldots, \boldsymbol{\mu}_{N,t}^{[k]}, \boldsymbol{\Sigma}_{N,t}^{[k]}\},$$

where $\boldsymbol{x}_t^{[k]}$ is a hypothesis of the robot state at time t, $(\boldsymbol{\mu}_{n,t}^{[k]}, \boldsymbol{\Sigma}_{n,t}^{[k]})$ are the mean and covariance of the EKF associated with landmark $\boldsymbol{m}_n$, and where it is assumed that there are N total landmarks in the map $\boldsymbol{m}$. As can be seen, with a total of M particles there are a total of NM EKFs! To summarize, the Fast SLAM algorithm is a particle based algorithm where each particle keeps track of a hypothesis of the robot state as well as the location (and uncertainty) of each landmark in the map! The algorithm is defined in Algorithm 15.

Note the blending of the classical particle filter algorithm with the EKF localization algorithm. In particular, the particle filter steps can be seen with the sampling of the new pose $\boldsymbol{x}_t$ from the state transition model and the use of the weights w for resampling a new set of particles (i.e. the measurement correction step). The EKF portions of the algorithm correspond to how the features are tracked, and in particular how the mean and covariance of the Gaussian corresponding to each landmark are updated based on new measurements.

Algorithm 15: Fast SLAM Algorithm

Data: $\mathcal{Y}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t, c_t$
Result: $\mathcal{Y}_t$
for $k = 1$ **to** M **do**
  Sample $\boldsymbol{x}_t^{[k]} \sim p(\boldsymbol{x}_t \mid \boldsymbol{u}_t, \boldsymbol{x}_{t-1}^{[k]})$
  $j = c_t$
  if *landmark j never seen before* **then**
    Initialize feature: $(\boldsymbol{\mu}_{j,t-1}^{[k]}, \boldsymbol{\Sigma}_{j,t-1}^{[k]})$
  else
    $\hat{\boldsymbol{z}}^{[k]} = h(\boldsymbol{\mu}_{j,t-1}^{[k]}, \boldsymbol{x}_t^{[k]})$
    $S = H^j \boldsymbol{\Sigma}_{j,t-1}^{[k]} [H^j]^T + \boldsymbol{Q}_t$
    $K = \boldsymbol{\Sigma}_{j,t-1}^{[k]} [H^j]^T [S]^{-1}$
    $\boldsymbol{\mu}_{j,t}^{[k]} = \boldsymbol{\mu}_{j,t-1}^{[k]} + K(\boldsymbol{z}_t - \hat{\boldsymbol{z}}^{[k]})$
    $\boldsymbol{\Sigma}_{j,t}^{[k]} = (I - KH^j)\boldsymbol{\Sigma}_{j,t-1}^{[k]}$
    $w^{[k]} = \det(2\pi S)^{-1/2} \exp\big(-\frac{1}{2}(\boldsymbol{z}_t - \hat{\boldsymbol{z}}^{[k]}) \boldsymbol{Q}^{-1} (\boldsymbol{z}_t - \hat{\boldsymbol{z}}^{[k]}) \big)$
  for $n \in \{1, \ldots, N\}$, $n \neq c_t$ **do**
    $\boldsymbol{\mu}_{n,t}^{[k]} = \boldsymbol{\mu}_{n,t-1}^{[k]}$
    $\boldsymbol{\Sigma}_{n,t}^{[k]} = \boldsymbol{\Sigma}_{n,t-1}^{[k]}$
$\mathcal{Y}_t = \emptyset$
for $m = 1$ **to** M **do**
  Draw k with probability $\propto w_t^{[k]}$
  $\mathcal{Y}_t = \mathcal{Y}_t \cup (\bar{\boldsymbol{x}}_t^{[k]}, \boldsymbol{\mu}_{1,t}^{[k]}, \boldsymbol{\Sigma}_{1,t}^{[k]}, \ldots, \boldsymbol{\mu}_{N,t}^{[k]}, \boldsymbol{\Sigma}_{N,t}^{[k]})$
return $\mathcal{Y}_t$

17.4 Exercises

17.4.1 EKF SLAM

Complete *Problem 2: EKF SLAM* located in the online repository:

```
https://github.com/PrinciplesofRobotAutonomy/AA274A_HW4,
```

where you will implement an EKF SLAM algorithm. Note that the EKF localization exercise from the chapter on parametric filters should be completed first.

18
Sensor Fusion

Almost every robot will rely on multiple sensors (including multiple *types* of sensors) for perception and localization tasks. This allows the robot to take advantage of the different strengths of each sensor for a more well-rounded sensing capability. For example a self-driving car may use both laser rangefinders and radar for measuring distances, since in some cases one sensor may work better than the other. As another example, a wheeled robot may use GNSS sensors as well as wheel encoders to estimate position. However, while each sensor may provide data toward a similar goal (e.g. estimating position or orientation) their sensing modalities may be drastically different. This chapter covers the topic of *sensor fusion*[1,2], and provides a discussion on algorithms for effectively leveraging multiple sensing modalities toward a common objective.

[1] F. Gustafsson. *Statistical Sensor Fusion*. Studentlitteratur, 2013, p. 554

[2] D. Simon. *Optimal State Estimation: Kalman, H_∞, and Nonlinear Approaches*. John Wiley & Sons, 2006

Sensor Fusion

Using measurements from multiple sensors (potentially different types of sensors) is an effective technique for reducing the uncertainty in downstream perception and estimation tasks (see Figure 18.1). This is generally the case because

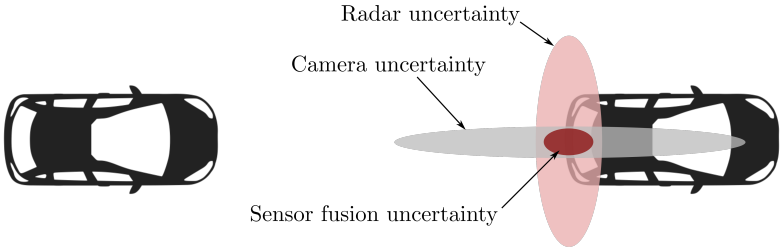


Figure 18.1: Sensor fusion can reduce uncertainty by providing more well-rounded data. For example in this scenario the radar sensor may have good accuracy longitudinally but less accuracy laterally. Contrarily, a camera may provide poor range estimation but good lateral position estimation. By fusing these two sensor measurements the resulting estimate can be accurate both longitudinally and laterally.

individual sensors typically suffer from limited range, limited field of view, or performance degradation under certain environmental conditions. Additionally, in single-sensor systems measurement accuracy degradation and sensor failure can be catastrophic. Alternatively, multi-sensor systems can address these challenges through redundancy of individual sensors (e.g. to provide full field of view measurements or multiple measurements of the same quantity) or through

sensor diversity (e.g. using sensors with different characteristics to offset limitations of others).

18.1 A Taxonomy of Sensor Fusion

To put the sensor fusion problem into a broader perspective, a taxonomy of sensor fusion related challenges will now be presented. This includes challenges associated with both fusion algorithms as well as the measurement data.

18.1.1 Data-related Taxonomy

One of the primary challenges with data fusion is the inherent imperfection in the measurement data, including uncertainty (i.e. resulting from sensor noise), imprecision (i.e. resulting from sensor bias), and granularity (i.e. resulting from sensor resolution). Other important data-related aspects to sensor fusion include data correlation, disparity, and inconsistency (e.g. data conflicts, outliers, disorder). Broadly speaking sensor data can experience multiple types of imperfection at the same time, and so data fusion algorithms should be developed with robustness in mind.

18.1.2 Fusion-related Taxonomy

At the data-fusion level, it is useful to classify the problem based on the type of data that is being fused. Low-level fusion problems typically fuse low-level signal data (i.e. time-series data), intermediate-level problems fuse features and characteristics, and high-level fusion problems consider decisions. Fusion problems can also be categorized based on the relationship among different sensors used in the fusion process. Competitive fusion problems consider redundant sensors that directly measure the same quantity. Complementary fusion is used when different sensors provide complementary information about the environment (e.g. lidar for short distance ranging and radar for long distance ranging). Finally, cooperative fusion considers problems where the required information cannot be inferred from a single sensor (e.g. GNSS localization and stereo vision can be cooperatively used because they measure fundamentally different environmental quantities). Generally speaking competitive fusion increases reliability and accuracy of fused information, complementary fusion increases the completeness of information, and cooperative fusion broadens the types of information that can be gathered.

18.1.3 Architectural Taxonomy

Fusion algorithms can also be classified based on their type of architecture, namely whether they are centralized, decentralized, or distributed. Centralized architectures collect *all* sensor data first, and then perform computations on the entire set of data. This approach is theoretically optimal since all information is

gathered and operated on at once, but the need for high levels of communication and processing can be challenging in practice. Decentralized architectures are essentially collections of centralized systems, and generally still suffer from the same high demands for communication and processing. On the other hand, distributed architectures do not collect all sensor information ahead of time but rather perform computations on local sensor data first, before potentially passing information on for further fusion tasks. These architectures scale better, but can lead to suboptimal performance because each sensor is performing local processing (i.e. without having all information).

18.2 Bayesian Approach to Sensor Fusion

Previous chapters presented several algorithms for robot state estimation and localization based on Bayes' filter. In fact, these algorithms can be viewed as approaches to solve the sensor fusion problem. This section explores the Bayesian approach to sensor fusion in more detail to show exactly how these approaches can blend measurement data to reduce uncertainty.

Recall that the Bayesian approach is a probabilistic approach that models unknowns as random variables and quantifies knowledge in the form of probability distributions over the unknowns. This principled approach is useful for sensors fusion for several reasons. First, it provides a unified framework for representing knowledge that is compatible with any quantity and type of sensors and is interpretable. Second, probability distributions implicitly provide information about uncertainty (e.g. the variance of a Gaussian). Third, Bayes' rule provides a principled approach for updating distributions. Finally, they can be used to deal with missing information and classification of new observations.

Example 18.2.1 (Competitive Fusion Example)**.** As an example to show how a probabilistic approach can be used to reduce uncertainty through sensor fusion, consider a case where two sensors are fused to estimate a single quantity $x \in \mathbb{R}$. Specifically, suppose the two measurements y_1 and y_2 are normally distributed random variables:

$$p(y_1 \mid x) = \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{1}{2}\frac{(x-y_1)^2}{\sigma_1^2}},$$
$$p(y_2 \mid x) = \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{1}{2}\frac{(x-y_2)^2}{\sigma_2^2}},$$

where the first sensor has a higher precision than the second sensor such that $\sigma_1^2 < \sigma_2^2$. Then the *combined* measurement probability is given by:

$$p(y_1, y_2 \mid x) = p(y_1 \mid x)p(y_2 \mid x),$$

by assuming conditional independence. By exploiting the product of two Gaus-

sian property this joint probability distribution is:

$$p(y_1, y_2 \mid x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}},$$

where:

$$\mu = \frac{y_1\sigma_2^2 + y_2\sigma_1^2}{\sigma_1^2 + \sigma_2^2}, \quad \sigma = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2}.$$

Therefore, given two measurements y_1 and y_2 the best estimate of the quantity x is given by μ, which is a *weighted* average of the two measurements. In particular, more weight is given to the measurement with higher precision (i.e. higher variance σ_i^2) and the overall uncertainty will decrease!

18.2.1 Kalman Filter Sensor Fusion

The Kalman filter from the previous chapter on parametric state estimation techniques is a common tool for sensor fusion problems. Recall that the Kalman filter assumes a linear state transition (dynamics) model:

$$\boldsymbol{x}_t = A_t\boldsymbol{x}_{t-1} + B_t\boldsymbol{u}_t + \boldsymbol{\epsilon}_t, \tag{18.1}$$

and a linear measurement model:

$$\boldsymbol{z}_t = C_t\boldsymbol{x}_t + \boldsymbol{\delta}_t, \tag{18.2}$$

where $\boldsymbol{x}$ is the state of the system and $\boldsymbol{z}$ are the measurements. Additionally, the Kalman filter assumes the belief distribution of $\boldsymbol{x}$ and the noise terms $\boldsymbol{\epsilon}$, $\boldsymbol{\delta}$ are all Gaussian:

$$bel(\boldsymbol{x}_t) \sim \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t), \quad \boldsymbol{\epsilon}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{R}_t), \quad \boldsymbol{\delta}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}_t),$$

where $\boldsymbol{R}_t$ and $\boldsymbol{Q}_t$ are the covariances of the state transition and measurement noise models, respectively. With these assumptions the Kalman filter algorithm uses a recursive "predict then correct" approach and the belief will always remain normally distributed.

This algorithm can be used for sensor fusion since the measurement vector $\boldsymbol{z}$ can include measurements from any type of sensor, as long as a linear relationship exists between the measurement and the underlying state $\boldsymbol{x}$ that is to be estimated. At each step of the Kalman filter algorithm, *every* measurement at time t is simultaneously used to update or "correct" the state predicted from the state transition model. Additionally, the Kalman filter takes into account the covariance $\boldsymbol{R}_t$, which includes the covariance of each individual sensor. In fact, the Kalman filter will implicitly favor measurements with lower covariance when performing the correction step[3].

[3] Specifically, this occurs during the computation of the Kalman gain.

A useful trick for applying the Kalman filter to sensor fusion problems is to also note that the state $\boldsymbol{x}$ can contain any type of information, it is not strictly limited to the state usually associated with the robot's dynamics or kinematics. For example, the state could be augmented with auxiliary states such as sensor bias or offsets, or variables to define sensor and actuator health.

Example 18.2.2 (Kalman Filter Multi-Sensor Fusion Example)**.** Consider a self-driving car that has an inertial measurement unit (IMU), a GNSS receiver, and a Lidar unit and where the goal is to leverage all of these sensors to estimate the position, velocity, and acceleration of the vehicle. This suite of sensors can provide noisy position estimates (Lidar and GNSS) as well as noisy acceleration measurements (IMU). For this application, sensor fusion can be accomplished through a Kalman filter.

First, consider a very simple kinematics model that only models longitudinal motion:

$$\dot{p} = v, \quad \dot{v} = a,$$

where p is the longitudinal position, v is the longitudinal velocity, and a is the longitudinal acceleration. This model is then discretized in time by choosing a sampling time T_s, yielding the linear difference equation:

$$\begin{bmatrix} p_{t+1} \\ v_{t+1} \\ a_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & T_s & \frac{T_s^2}{2} \\ 0 & 1 & T_s \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_t \\ v_t \\ a_t \end{bmatrix} + \boldsymbol{\epsilon}_t,$$

where the state is defined as $\boldsymbol{x} = [p, v, a]^\top$, and $\boldsymbol{\epsilon}$ is Gaussian process noise.

It is assumed that the lidar and GNSS sensors directly measure the position p, and that the IMU directly measures the acceleration a, such that the measurement model is:

$$\begin{bmatrix} z_{\text{lidar},t} \\ z_{\text{gnss},t} \\ z_{\text{imu},t} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_t \\ v_t \\ a_t \end{bmatrix} + \boldsymbol{\delta}_t,$$

where $\boldsymbol{\delta}$ is Gaussian measurement noise with zero mean and covariance:

$$\boldsymbol{Q}_t = \begin{bmatrix} \sigma_{\text{lidar}}^2 & 0 & 0 \\ 0 & \sigma_{\text{gnss}}^2 & 0 \\ 0 & 0 & \sigma_{\text{imu}}^2 \end{bmatrix},$$

with $\sigma_{\text{lidar}} = 0.5$, $\sigma_{\text{gnss}} = 0.1$, and $\sigma_{\text{imu}} = 0.2$

Figure 18.2 shows results of the application of the Kalman filter algorithm for fusing these sensor measurements into position estimates. The top plot presents a case where the GNSS sensor is not used, and as can be seen the noisy high-variance lidar measurements result in a noisy estimate of the ground truth position of the car. However, with the addition of the lower-variance GNSS sensor in the bottom figure, the estimate of the position is much more accurate. Generally speaking the estimate would also be more accurate even with the addition of a sensor that was even more noisy than the lidar, but the impact would not be as significant.

18.3 Challenges in Sensor Fusion

Sensor fusion problems can generally be quite challenging, and can vary significantly from application to application. Some of the more common problems in

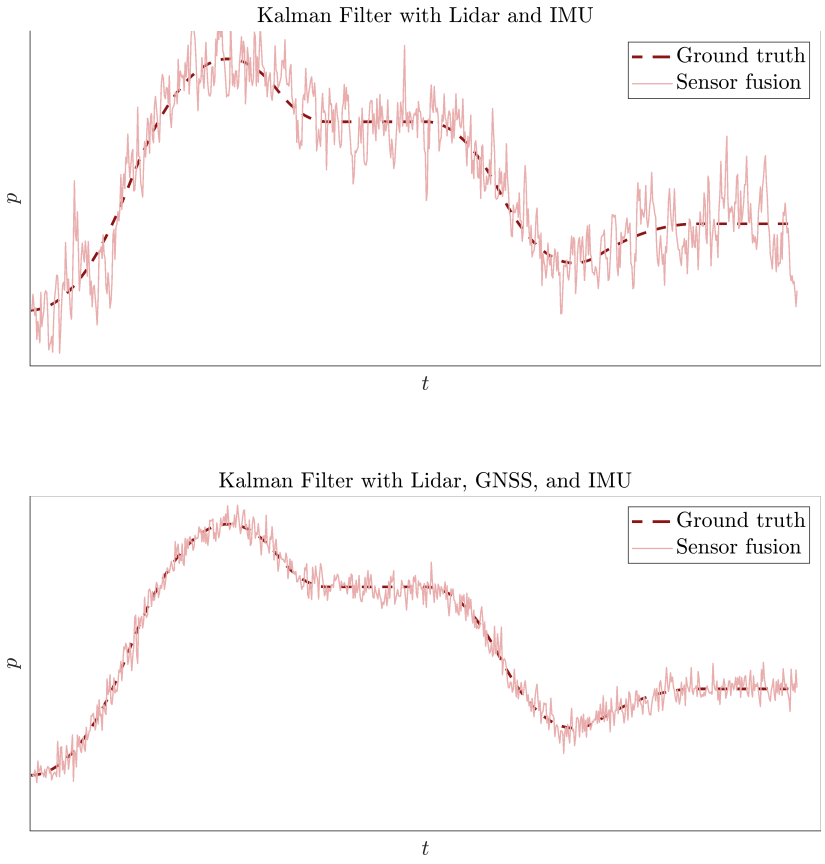


Figure 18.2: Kalman filter sensor fusion for Example 18.2.2. The position of a vehicle is estimated using noisy lidar, GNSS, and IMU data, and the resulting estimate tracks the ground truth. As can be seen, the addition of the lower-variance GNSS results in a better estimate through sensor fusion.

sensor fusion include registration, bias, correlation, data association, and out-of-sequence measurements. The registration problem is that coordinates (both time and space) of different sensors may not always be aligned, which is necessary to ensure they can be appropriately combined. Biases can also arise due to transformations of the data into the unified set of coordinates. Correlation between sensors can also occur, even if they are independently collecting data, and the knowledge of correlation between sensors can have an impact on the best way to fuse the information. In some robotics applications, data association can also be a challenge. One simple example is in multi-target tracking problems, which is similar to the correspondence problem in SLAM problems. Finally, out-of-sequence measurements also pose a logistical challenge in practical sensor fusion applications. These issues often arise due to communication limitations among agents in multi-agent settings. Out-of-sequence measurements might lead to an incorrect temporal order, which in turn causes a negative time measurement update during data fusion fusion. As a consequence, robot localization is biased or a wrong representation of the environment is created. There are a couple of methods to avoid that including external triggering, centralized (time-stamping of data at arrival), or distributed (time-stamping at the time of data acquisition) approaches.

Object Tracking

Autonomous systems, such as self-driving cars or robots, rely critically on an accurate perception of their dynamic environment. Consequently, tracking of other objects, i.e. predicting the state of remote objects given measurement uncertainties, ambiguous measurements, occluded objects, or sensor false alarms, is of great importance for autonomous systems. While single-object tracking (single hypothesis tracking) is well-understood and typically easy to implement (one moving target is tracked by one EKF), the tracking of multiple targets is much more challenging.

18.4 Multi-Object Tracking

Multi-object tracking (MOT) runs a set of estimation filters, one filter for each object to be tracked. While single-model Kalman filters are predominantly in use, a bunch of different approaches exists in the literature such as the interacting multiple model (IMM) filter. Challenges of MOT to be faced are

- inherent uncertainties in prediction (state propagation)
- the data association problem (association of observations and targets)
- track maintenance (creating and deleting tracks)
- multiple reflections from a single object

18.5 Gating

In general, we need to look at every single observation and consider how likely it is to be assigned to a track. To keep the computational efforts low, a screening algorithm (gating) is applied. During this step, observations outside of a specific region for each track are ignored, i.e. the data for assignment is significantly reduced. A rectangular gate is the simplest approach while an ellipsoidal gate is much more intuitive (normal distribution).

18.6 Data Association

Data association or data assignment is the process of linking an observation to a tracked object, i.e. to a track. This can be particularly difficult if we have a large number of targets, many detections, or conflicting hypotheses. Depending on the dimension, we distinguish between 2-D assignment problems (assigning n targets to m observations) and S-D assignment problems (assigning n targets to a set of observations). Within this class, we discuss two typically applied 2-D techniques:

- Global Nearest Neighbor (GNN): is a single hypothesis approach that assigns the global nearest observations to existing tracks and creates new track hypotheses for unassigned observations.
- Joint Probabilistic Data Association (JPDA): is a Bayes-based technique that fuses measurements weighted by the probability of the observation-to-track association. Clustering is usually applied if too many hypotheses are present.

18.7 Track Maintenance

Consists of two steps: deleting a track and creating a track. If a track has not been assigned to a detection at least M times during the last N updates, where N and M are tuning parameters, the track will be deleted. If there is a single unassigned observation, a new tentative track is created. This track is confirmed when detected M times over the last N updates and rejected otherwise.

18.8 Extended Object Tracking

If one moving target generated multiple reflections leading to multiple detections, standard MOT algorithms might fail. This might occur if emerging high-resolution radar sensors are used. These extended objects present new challenges to conventional trackers since those assume a single detection per object per sensor. Extended object tracking (EOT) algorithms are able to deal with this situation. EOT estimate position and velocity, but also the dimensions and the orientation of the moving object. Prominent algorithms are, among others, the Gamma-Gaussian inverse Wishart probability hypothesis density (PHD) tracker and the Gaussian-mixture PHD tracker. Basically, there exist two different but intuitive approaches in MOT:

- Estimating where each individual target is. Each target gets an identity label and targets are tracked while trying to maintain the identities. In situations of targets being closely spaced, that may not be solvable. GNN, JPDA, or multiple hypotheses tracking (MHT) are typically applied along with state estimators.
- Estimating where there are targets. In this case, target identities are not relevant. Typically, a random finite set (RFS) description of the targets is used.

Part IV

Robot Decision Making

19
Finite State Machines

So far a number of algorithms for control, trajectory optimization, motion planning, perception, and localization/state estimation have been presented. Almost all of these instances share a common characteristic: they involve manipulation or observation of *continuous* variables. For example, motion planning and control algorithms manipulate the robot's physical state (i.e. position, velocity, orientation, configuration) which can take on a continuous range of values, and perception and localization tasks try to take (continuously valued) information from the environment and try to estimate the robot's physical state.

However, for higher-level tasks it is often useful to represent the state of the robot or environment in terms of a discrete set of variables. For example, consider a robot whose task is to go from point A to point B, pick up a package, and then deliver it to point C. While the robot's physical (continuous) state is crucial for tasks such as controlling the robot to drive from A to B, it is also important to keep track of what portion of the overall plan that the robot is currently performing (is the robot currently traversing to B or C, has the package been successfully picked up, etc.). Additionally, it might be useful to keep track of other discrete valued states of the robot, such as if a sensor is functioning or not, or whether or not the robot is in the presence of a human (i.e. for safety). Similar to dynamics/kinematics models for the robot's (continuous) physical state, *finite state machines*[1] are a useful framework for modeling discrete higher-level states of the robot and its environment.

[1] L. Kaelbling et al. *6.01SC: Introduction to Electrical Engineering and Computer Science I*. MIT OpenCourseWare. 2011

Finite State Machines

Finite state machines (FSMs) define a computational modeling framework for systems whose output depends on the entire history of their inputs, and where the number of possible states of the system is *finite*. This framework has been used in a wide variety of disciplines, including electrical engineering, linguistics, computer science, philosophy, biology, and more. FSMs can also be used in several different ways, including:

1. *to specify a desired program or behavior*, such as how a vending machine or ATM should function,
2. *to model behavior*, for example to analyze the behavior of a control system interacting with the environment,
3. *or for predicting behavior*, for example to predict what will happen in the future given some set of inputs to the system.

Generally speaking, designing finite state machines for practical robotic systems can be extremely time consuming and challenging. In particular, choosing the appropriate set of states for a particular problem is required to ensure that the model is not overly complex, but the interactions and transitions between states can also be very hard to specify and can still lead to complex models. For example, consider the graphical representation of an example FSM for the popular open source flight software PX4 in Figure 19.1. Specifying the full behavior of the system can lead to a complex FSM, even if there are not very many states. In fact, this FSM is still under continuous development to improve the overall system behavior!

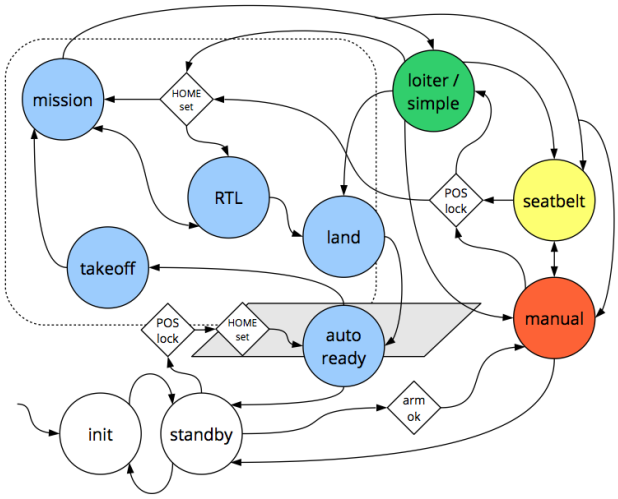


Figure 19.1: A graphical representation of a finite state machine example for the open source flight software PX4, `https://px4.io/`. As can be seen, even for a relatively small number of states the FSM can become quite complex in order to model the full behavior of the system. Image retrieved from diydrones.com.

Mathematically, a finite state machine consists of:

1. a *finite* set of states S,
2. a set of inputs I,
3. a set of outputs O,
4. a next-state function $n(i_t, s_t) \rightarrow s_{t+1}$ that maps the input i_t at time t and current state s_t to the next state s_{t+1},
5. an output function $o(i_t, s_t) \rightarrow o_t$,
6. and an initial state s_0.

While FSMs can be defined through the mathematical notation above, it is often also useful to represent them graphically to get a more intuitive understanding of how the system will behave. In particular, the graph representation is defined with nodes of the graph representing each state in the set S. Each (directed) edge of the graph corresponds to a possible transition between states that is defined by a particular input. In other words, each directed edge is associated with a particular pair (s, i). The outputs for a particular pair (s, i) are also typically included along each directed edge. This is shown in more detail in Figure 19.2.

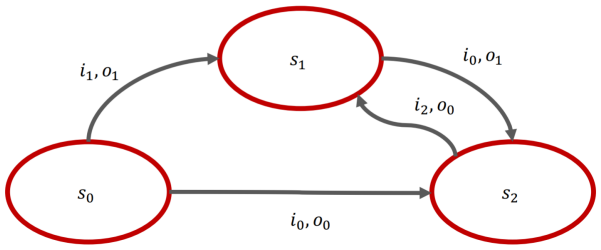


Figure 19.2: A graphical representation of a finite state machine with states $S = \{s_0, s_1, s_2\}$, inputs $I = \{i_0, i_1, i_2\}$ and outputs $O = \{o_0, o_1\}$. The directed edges correspond to the next-state functions and the output associated with each edge is defined by the output function. For example, in this FSM it can be seen that $n(i_1, s_0) \rightarrow s_1$ and $o(i_1, s_0) \rightarrow o_1$.

Example 19.0.1 (Parking Gate Control)**.** Consider a parking gate control finite state machine where the goal is to raise the gate when a car arrives and then lower the gate when the car has passed. Assume sensors are available to tell if a car is at the gate and when the car has passed through the gate, and also the position of the gate. The control actions the gate can take are simply raising, lowering, or holding the gate position fixed. Technically, the position of the gate can vary continuously between the "down" and "up" positions, and the velocity can also vary continuously. However, in designing a finite state machine to define the overall logic/behavior for the parking gate, a higher-level abstraction of the set of gate states can be chosen as:

$$S = \{\text{down, raising, up, lowering}\}.$$

The set of inputs to the finite state machine come from the sensors, and can be chosen as:

$$I = \{\text{car waiting, no car waiting, car passed, car not passed,}$$
$$\text{gate up, gate not up, gate down, gate not down}\}.$$

Finally, the output of the finite state machine (defining the actions for the gate) are simply:

$$O = \{\text{lower, raise, hold}\}.$$

The next-state function then defines the desired behavior for the parking gate. For example, suppose the current state $s_t = \text{down}$ and the sensor measures that a car is waiting ($i_t = \text{car waiting}$). Then, the desired behavior is to output the command $o_t = \text{raise}$, and the next-state function would be:

$$n(\text{car waiting, down}) \rightarrow \text{raising}.$$

Similarly, suppose the gate was just raised for the car to pass such that $s_t = \text{up}$, but that the sensor is giving input $i_t = \text{car not passed}$. In this case the output would be $o_t = \text{hold}$, and the next-state function would be:

$$n(\text{up}, \text{car not passed}) \rightarrow \text{up}.$$

A graphical representation of the full car parking gate FSM is given in Figure 19.3.

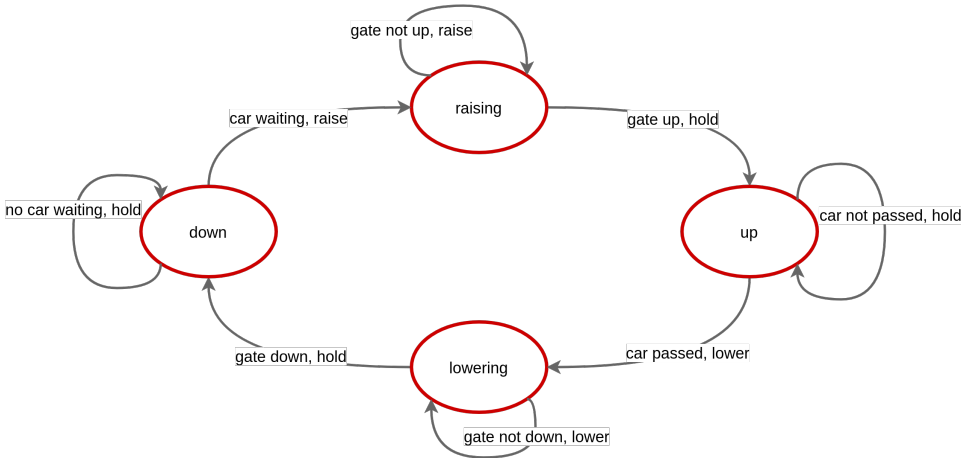


Figure 19.3: A graphical representation of the finite state machine for the parking gate controller discussed in Example 19.0.1.

19.1 FSM Architectures

Finite state machines can become quite complex since for every new state added it is possible to define an exponentially increasing number of new transitions. Strategies for keeping the complexity of FSMs in check include analyzing for (and removing) redundant states, using hierarchical FSMs, and using compositions based on common patterns.

19.1.1 Reducing Number of States

There exist algorithms that can be used to identify and combine states in FSMs that would yield the same overall behavior. In particular, two states are equivalent if they have the same output and for all input combinations transition to the same or equivalent states.

One possible algorithm for reducing states in an FSM is as follows:

1. Place all states into one set.
2. Create a single partition based on the output behavior.
3. Repeatedly partition further based on next state transitions until no further partitions is possible.

To see this procedure in action, consider the following example:

Example 19.1.1 (FSM State Reduction)**.** Consider a finite state machine that is used to detect the sequences 010 or 110. The FSM is shown in Table 19.1, where it can be seen that the states are the partial sequences $S = \{0, 1, 00, 01, 10, 11\}$ and a reset state, the inputs are $I = \{0, 1\}$, and the outputs are booleans $O = \{\text{True}, \text{False}\}$ for whether the sequence 010 or 110 has been created. For example, it can be seen that if the current partial sequence is 01 (s_4) and a 0 is input, the next state will be the reset state and the output will be True.

State, s	$n(0, s)$	$n(1, s)$	$o(0, s)$	$o(1, s)$
Reset	0	1	False	False
0	00	01	False	False
1	10	11	False	False
00	Reset	Reset	False	False
01	Reset	Reset	True	False
10	Reset	Reset	False	False
11	Reset	Reset	True	False

Table 19.1: Finite state machine for a sequence detector that accepts digits 0 and 1 and outputs True if the sequences 010 or 110 is generated.

Now, the FSM in Table 19.1 can be simplified by removing redundant states! This is accomplished by first placing all of the states into a single set $\{\text{Reset}, 0, 1, 00, 01, 10, 11\}$ and creating a partition based on the output behavior. In particular this will generate two sets:

$$\{\text{Reset}, 0, 1, 00, 10\} : \text{always leads to False output},$$
$$\{01, 11\} : \text{does not always lead to False output}.$$

These sets are then further partitioned based on the next-state function until no further partitions can be made. In the first step the set $\{\text{Reset}, 0, 1, 00, 10\}$ is partitioned into:

$$\{\text{Reset}, 00, 10\} : \text{cannot transition to } \{01,11\},$$
$$\{0, 1\} : \text{can transition to } \{01,11\}.$$

and then $\{\text{Reset}, 00, 10\}$ is partitioned as:

$$\{\text{Reset}\} : \text{can transition to } \{0, 1\},$$
$$\{00, 10\} : \text{cannot transition to } \{0, 1\}.$$

Therefore, instead of the original seven states (Reset, 0, 1, 00, 01, 10, 11) there are now only four ($\{01, 11\}$, $\{0, 1\}$, Reset, $\{00, 10\}$). An equivalent (same input/output behavior) but reduced finite state machine can now be defined, and is shown in Table 19.2.

19.1.2 Hierarchical FSMs

In some cases there might be states that are not truly equivalent, but that might still be beneficial to group closely together. With this idea, the concepts of *super-*

State, s	$n(0,s)$	$n(1,s)$	$o(0,s)$	$o(1,s)$
Reset	{0,1}	{0,1}	False	False
{0,1}	{00,10}	{01,11}	False	False
{00,10}	Reset	Reset	False	False
{01,11}	Reset	Reset	True	False

Table 19.2: Reduced finite state machine for a sequence detector that accepts digits 0 and 1 and outputs True if the sequences 010 or 110 is generated.

states (i.e. groups of closely related states) and *generalized transitions* (i.e. transitions between super-states) can be useful. This idea of creating super-states is analogous to graph clustering.

19.1.3 Compositions

Individual state machines can also be composed in a variety of ways depending on their input/output behavior, including *cascade* compositions, *parallel* compositions, and *feedback* compositions. Cascade compositions combine two FSMs in sequence where the output vocabulary of one matches the input vocabulary of the other. The new state of the combined machine is the concatenation of the states of the individual FSMs (see Figure 19.4). Parallel compositions run two FSMs side by side, using the same input. Both the state and output is then the concatenation of the two individual FSMs' state and output. Finally, feedback compositions use only a single FSM but only require a partial input and also reuse the output as input (requires the input and output vocabularies to be the same).

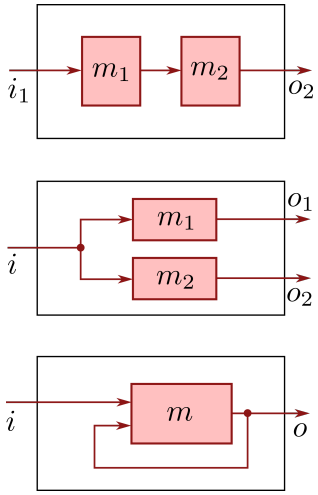


Figure 19.4: Cascade, parallel, and feedback compositions of finite state machines.

19.2 Implementation Details

There are *numerous* ways that finite state machines could be implemented in practice. However, one common approach is to exploit Object Oriented Programming (OOP) by building the finite state machine as a class. In particular, the class would keep track of the state of the FSM in a class variable. The state update process could then occur through the use of if/else statements in an update class method, as well as the definition of the FSM output. An example implementation in Python of the parking gate controller FSM from Example 19.0.1 is given below:

```
import rospy as rp
from std_msgs.msg import String

class ParkingGateFSM():
    """Simple FSM for parking gate control"""
    def __init__(self):
        rp.init_node('parking_gate', anonymous=True)
        self.state = 'down'
        self.cmd = rp.Publisher('/gate_cmd', String)
        rp.Subscriber('/car_sensor', String, self.car_clbk)
```

```
        rp.Subscriber('/gate_sensor', String, self.gate_clbk)

    def car_clbk(self, data):
        self.car_input = data

    def gate_clbk(self, data):
        self.gate_input = data

    def run(self):
        rate = rp.Rate(10) # 10 Hz
        while not rp.is_shutdown():
            if self.state == 'down':
                if self.car_input == 'no_car_waiting':
                    output = 'hold'
                elif self.car_input == 'car_waiting':
                    self.state = 'raising'
                    output = 'raise'
            elif self.state == 'raising':
                if self.gate_input == 'gate_not_up':
                    output = 'raise'
                elif self.gate_input == 'gate_up':
                    self.state = 'up'
                    output = 'hold'
            elif self.state == 'up':
                if self.car_input == 'car_not_passed':
                    output = 'hold'
                elif self.car_input == 'car_passed':
                    self.state = 'lowering'
                    output = 'lower'
            elif self.state == 'lowering':
                if self.gate_input == 'gate_not_down':
                    output = 'lower'
                elif self.gate_input == 'gate_down':
                    self.state = 'down'
                    output = 'hold'
            self.cmd.publish(output)
            rate.sleep()
```

19.3 Other Useful Tools

A useful tool for visualizing finite state machines in ROS is SMACH, which can be though of as an analogue to RViz. More information about SMACH and how it is used can be found on the ROS Wiki[2].

[2] http://wiki.ros.org/smach

20
Sequential Decision Making

This chapter provides an introduction to fundamental topics in decision making, including for problems where there is some uncertainty (e.g. uncertainty about the robot's state or about the environment).

Sequential Decision Making

Two of the fundamental challenges associated with robotic decision making are that *sequences* of decisions must be made (which requires reasoning about future actions and observations) and that uncertainty may exist in the operating environment. This chapter presents a modeling framework for addressing decision making problems and will also introduce *dynamic programming*, a fundamental approach for solving these problems.

20.1 Deterministic Decision Making Problem

The standard mathematical formulation for decision making problems includes several components: a model of the robot's behavior, a set of admissible controls, and a cost function. This set of components is quite similar to the components used in trajectory optimization problems, however decision making problems are generally represented in *discrete-time* rather than in *continuous-time*[1].

[1] There is a continuous-time formulation, known as the Hamilton–Jacobi–Bellman formulation.

In the deterministic decision making problem, the model of the robot is expressed in *discrete-time* as:

$$\boldsymbol{x}_{k+1} = f_k(\boldsymbol{x}_k, \boldsymbol{u}_k), \quad k = 0, \ldots, N-1, \tag{20.1}$$

where $\boldsymbol{x}$ is the robot's state, $\boldsymbol{u}$ is the control, f_k defines how the robot's state changes at time step k, and N is an integer that defines a finite planning horizon for the decision making problem. There are generally no restrictions on how the functions f_k are defined, they could come from a physics-based dynamics/kinematics model or even a higher-level state transition model.

It is also generally assumed that only some control actions are admissible at a given state, which denoted by the set $\mathcal{U}(\boldsymbol{x}_k)$. For example a car may only have

an option to turn left or right when it is at an intersection. Therefore the control constraints for the robot at time step k are given by:

$$\boldsymbol{u}_k \in \mathcal{U}(\boldsymbol{x}_k). \tag{20.2}$$

Again, there are generally no restrictions on how the set of admissible control is defined. For example $\mathcal{U}(\boldsymbol{x}_k)$ could be a finite set of actions, it could be a convex region of allowable inputs, etc.

The cost function is assumed to be *additive*, and is defined as:

$$J(\boldsymbol{x}_0, \boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}) = g_N(\boldsymbol{x}_N) + \sum_{k=0}^{N-1} g_k(\boldsymbol{x}_k, \boldsymbol{u}_k), \tag{20.3}$$

where g_N is a terminal state cost function and g_k for $k = 0, \ldots, N-1$ are stage cost functions. These individual cost functions are also not restricted to a particular form (e.g. convex, differentiable, etc.).

Definition 20.1.1 (Deterministic Decision Making Problem)**.** *The deterministic decision making problem can be expressed for the system model* (20.1)*, control constraints* (20.2)*, and cost function* (20.3) *as:*

$$J^*(\boldsymbol{x}_0) = \min_{\boldsymbol{u}_k \in \mathcal{U}(\boldsymbol{x}_k),\ k=0,\ldots,N-1} J(\boldsymbol{x}_0, \boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}). \tag{20.4}$$

Notice that this problem is used to compute an *open-loop* control sequence $\{\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}\}$ given an initial condition $\boldsymbol{x}_0$. However, this problem is generally quite hard to solve since there is no guarantee that the model (20.1) and cost function (20.3) have any particular structure that can be leveraged to make the optimization problem amenable to numerical optimization algorithms. While it is theoretically possible to solve the problem through a brute force search over all possible combinations of sequences $\{\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}\}$, this leads to a combinatorial explosion of options and is therefore not possible in practical settings (except of course for very small problems).

20.1.1 Principle of Optimality (Deterministic)

Fortunately, there is in fact an underlying structure to the deterministic decision making problem that can be leveraged to make the problem easier to solve. This structure is commonly referred to as the *principle of optimality*.

The principle of optimality for deterministic systems is that for a sequence of optimal decisions, the *tail* of the optimal sequence is also optimal for a *tail subproblem*. For a concrete example see Figure 20.1. This can greatly simplify the overall problem, since you can "reuse" optimal paths for different scenarios. More formally, the principle of optimality is given by the following theorem:

Theorem 20.1.2 (Principle of Optimality (Deterministic))**.** *Let* $\{\boldsymbol{u}_0^*, \boldsymbol{u}_1^* \ldots, \boldsymbol{u}_{N-1}^*\}$ *be an optimal control sequence to the deterministic decision making problem* (20.4) *with a given initial condition* $\boldsymbol{x}_0^*$*, such that the resulting optimal state sequence is*

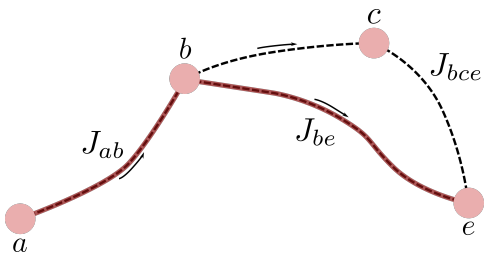


Figure 20.1: Starting from point a, let the red path $a - b - e$ be the optimal path from a to e, with a total cost of $J^*_{ae} = J_{ab} + J_{be}$. The principle of optimality in this case says that the path $b - e$ must therefore be the optimal path when starting from point b. This can be proven by contradiction, since if the path $b - c - e$ had a lower cost than path $b - e$ (i.e. $J_{bce} < J_{be}$), then the original path $a - b - e$ cannot be optimal!

$\{\boldsymbol{x}^*_0, \boldsymbol{x}^*_1 \ldots, \boldsymbol{x}^*_N\}$. Then, the tail sequence $\{\boldsymbol{u}^*_k, \ldots, \boldsymbol{u}^*_{N-1}\}$ is an optimal control sequence when starting from $\boldsymbol{x}^*_k$ and minimizing the cost from time k to time N

$$J_{tail}(\boldsymbol{x}_k, \boldsymbol{u}_k, \ldots, \boldsymbol{u}_{N-1}) = g_N(\boldsymbol{x}_N) + \sum_{m=k}^{N-1} g_m(\boldsymbol{x}_m, \boldsymbol{u}_m).$$

To see how the principle of optimality can be applied to simplify the decision making problem, consider the scenario in Figure 20.2. In this case it is desired to find an optimal path from point b to point f, and it is assumed that optimal paths from c, d, and e to f are already known. A brute force search over all possible paths in this problem would require nine paths to be evaluated:

$$\{b-c-f, \quad b-c-d-f, \quad b-c-d-e-f, \quad b-d-c-f, \quad b-d-f,$$
$$b-d-e-f, \quad b-e-d-c-f, \quad b-e-d-f, \quad b-e-f\}.$$

However, by leveraging the principle of optimality the number of candidate paths is reduced to three:

$$b-c-f, \quad b-d-f, \quad b-e-f.$$

In other words, the principle of optimality allows the search to be performed over *immediate* decisions by also concatenating the optimal tail decisions! This procedure is generally implemented backward in time, for example in Figure 20.2 the point f (the goal) is first evaluated, then the points c, d, and e, and then finally the point b.

20.1.2 Dynamic Programming (Deterministic)

The dynamic programming (DP) algorithm *globally* solves the deterministic decision making problem (20.4) by leveraging the principle of optimality[2]. The dynamic programming algorithm is given in Algorithm 16, where it can be seen that a backward-in-time recursion is used and at each step a *local* optimization is performed (this local optimization is referred to as the *Bellman* equation), leveraging the optimal *tail* costs from the previous iteration.

The output of the dynamic programming algorithm is a set of costs $J^*_k(\boldsymbol{x}_k)$ for each time step $k = 0, \ldots, N$ and states $\boldsymbol{x}_k$, which provide the optimal *tail* cost for the *tail* subproblem.

[2] Note that the principle of optimality is a fundamental property that is actually utilized in almost all decision making algorithms, including reinforcement learning.

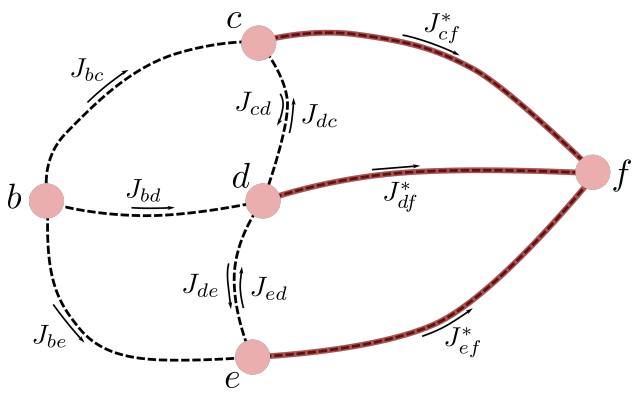


Figure 20.2: Suppose the optimal paths from points c, d and e to f are known (shown in red). By using the principle of optimality, an optimal path from point b to f can be found by *only* searching over paths from b to c, d, and e, and determining the lowest cost from the candidates $\{J_{bc} + J^*_{cf}, J_{bd} + J^*_{df}, J_{be} + J^*_{ef}\}$. In other words, the *optimal tails* can be leveraged to reduce the total number of paths that need to be considered when finding an optimal path from b to f!

Algorithm 16: Dynamic Programming (Deterministic)

$J^*_N(\boldsymbol{x}_N) = g_n(\boldsymbol{x}_N)$, for all $\boldsymbol{x}_N$
for $k = N-1$ **to** 0 **do**
  $J^*_k(\boldsymbol{x}_k) = \min_{\boldsymbol{u}_k \in \mathcal{U}(\boldsymbol{x}_k)} g_k(\boldsymbol{x}_k, \boldsymbol{u}_k) + J^*_{k+1}(f_k(\boldsymbol{x}_k, \boldsymbol{u}_k))$, for all $\boldsymbol{x}_k$
return $J^*_0(\cdot), \ldots, J^*_N(\cdot)$

Given an initial condition $\boldsymbol{x}_0$, the optimal control sequence $\{\boldsymbol{u}^*_0, \ldots, \boldsymbol{u}^*_{N-1}\}$ that solves the deterministic decision making problem can be computed with a "forward pass", where:

$$\boldsymbol{u}^*_0 = \arg\min_{\boldsymbol{u}_0 \in \mathcal{U}(\boldsymbol{x}_0)} g_0(\boldsymbol{x}_0, \boldsymbol{u}_0) + J^*_1(f_0(\boldsymbol{x}_0, \boldsymbol{u}_0)).$$

The next state is then computed as $\boldsymbol{x}^*_1 = f_0(\boldsymbol{x}_0, \boldsymbol{u}^*_0)$, and the process is repeated:

$$\boldsymbol{u}^*_1 = \arg\min_{\boldsymbol{u}_1 \in \mathcal{U}(\boldsymbol{x}^*_1)} g_1(\boldsymbol{x}^*_1, \boldsymbol{u}_1) + J^*_2(f_1(\boldsymbol{x}^*_1, \boldsymbol{u}_1)),$$

until the full trajectory and optimal control is specified.

Note that in practice the DP algorithm is not practical for continuously values states $\boldsymbol{x}$, since an infinite number of states would have to be iterated over. Therefore one possible modification to handle continuously valued states is to *quantize* the state space into a finite set of states (other approaches, such as interpolation, are also possible). Also, it is interesting to note that the addition of control constraints can actually simplify the procedure, since it restricts the number of possible options that need to be considered!

Example 20.1.1 (Deterministic Dynamic Programming)**.** Consider the environment shown in Figure 20.3, where the goal is to start at point a and reach point h while incurring the smallest cost. In this problem the state is represented as the current location (i.e. a, b, etc.), and the control constraints are encoded by the arrows indicating possible directions of travel (e.g. at point c it is possible to either go right or up, but not down or left). The cost of traversing between two points is also denoted in Figure 20.3.

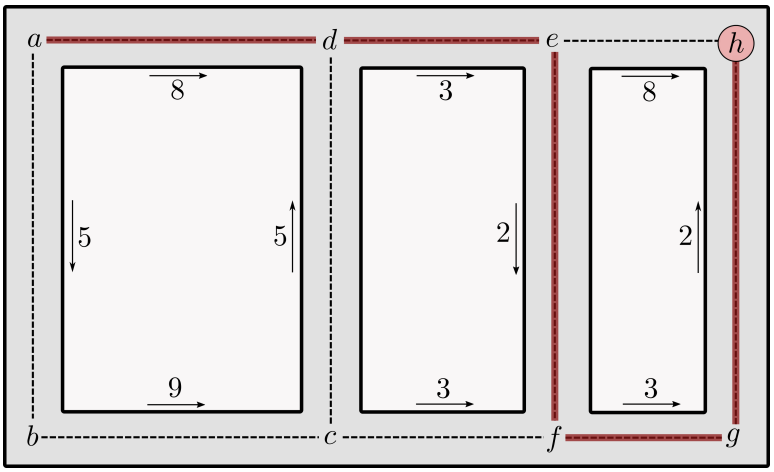


Figure 20.3: A deterministic decision making problem where the goal is to move from point a to point h while incurring the minimal amount of cost. The red path indicates the optimal path. This problem is solved by dynamic programming in Example 20.1.1.

To implement the DP algorithm, the final point h is chosen as $\boldsymbol{x}_N$, and the DP recursion begins with:

$$J_N^*(h) = 0,$$

since there is no cost to stay at point h. Moving backward in time, it can be seen that the possible states $\boldsymbol{x}_{N-1}$ that can transition to $\boldsymbol{x}_N = h$ are the points h, e, and g (assuming it is possible to stay at h with no cost). Therefore in the first step of the DP recursion:

$$\begin{aligned}
J_{N-1}^*(h) &= 0 + J_N^*(h) = 0, \quad u_{N-1}^*(h) = \text{stay}. \\
J_{N-1}^*(e) &= 8 + J_N^*(h) = 8, \quad u_{N-1}^*(e) = \text{right}, \\
J_{N-1}^*(g) &= 2 + J_N^*(h) = 2, \quad u_{N-1}^*(g) = \text{up},
\end{aligned}$$

Note that $J_k^*(h) = 0$ for all $k \leq N$, and therefore it will not be explicitly included in the following steps. In the next step:

$$\begin{aligned}
J_{N-2}^*(e) &= 8 + J_{N-1}^*(h) = 8, \quad u_{N-2}^*(e) = \text{right}, \\
J_{N-2}^*(g) &= 2, \quad u_{N-2}^*(g) = \text{up}, \\
J_{N-2}^*(d) &= 3 + J_{N-1}^*(e) = 11, \quad u_{N-2}^*(d) = \text{right}, \\
J_{N-2}^*(f) &= 3 + J_{N-1}^*(g) = 5, \quad u_{N-2}^*(f) = \text{right},
\end{aligned}$$

At this point, these optimal tail costs can be considered to be the optimal costs associated with control actions that lead from e, g, d, or f to the end point h in *two* steps! Continuing on:

$$\begin{aligned}
J_{N-3}^*(e) &= \min\{8 + J_{N-2}^*(h),\, 2 + J_{N-2}^*(f)\} = 7, \quad u_{N-3}^*(e) = \text{down}, \\
J_{N-3}^*(g) &= 2, \quad u_{N-3}^*(g) = \text{up}, \\
J_{N-3}^*(d) &= 3 + J_{N-2}^*(e) = 11, \quad u_{N-3}^*(d) = \text{right}, \\
J_{N-3}^*(f) &= 5, \quad u_{N-3}^*(f) = \text{right}, \\
J_{N-3}^*(a) &= 8 + J_{N-2}^*(d) = 19, \quad u_{N-3}^*(a) = \text{right}, \\
J_{N-3}^*(c) &= \min\{5 + J_{N-2}^*(d),\, 3 + J_{N-2}^*(f)\} = 8, \quad u_{N-3}^*(c) = \text{right}.
\end{aligned}$$

Interestingly, it can be seen that it is now possible to accomplish the objective (i.e. go from point a to h) in 3 time steps (i.e. on path $a-d-e-h$) and incur an optimal cost of 19. However it turns out that an even lower cost is achievable if the number of time steps is increased further! Continuing the DP recursion:

$$
\begin{aligned}
&J^*_{N-4}(e) = 7, \quad u^*_{N-4}(e) = \text{down},\\
&J^*_{N-4}(g) = 2, \quad u^*_{N-4}(g) = \text{up},\\
&J^*_{N-4}(d) = 3 + J^*_{N-3}(e) = 10, \quad u^*_{N-4}(d) = \text{right},\\
&J^*_{N-4}(f) = 5, \quad u^*_{N-4}(f) = \text{right},\\
&J^*_{N-4}(a) = 8 + J^*_{N-3}(d) = 19, \quad u^*_{N-4}(a) = \text{right}\\
&J^*_{N-4}(c) = \min\{5 + J^*_{N-3}(d),\, 3 + J^*_{N-3}(f)\} = 8, \quad u^*_{N-4}(c) = \text{right},\\
&J^*_{N-4}(b) = 9 + J^*_{N-3}(c) = 17, \quad u^*_{N-4}(b) = \text{right},
\end{aligned}
$$

and finally with one more iteration:

$$
\begin{aligned}
&J^*_{N-5}(e) = 7, \quad u^*_{N-5}(e) = \text{down},\\
&J^*_{N-5}(g) = 2, \quad u^*_{N-5}(g) = \text{up},\\
&J^*_{N-5}(d) = 10, \quad u^*_{N-5}(d) = \text{right},\\
&J^*_{N-5}(f) = 5, \quad u^*_{N-5}(f) = \text{right},\\
&J^*_{N-5}(a) = \min\{8 + J^*_{N-4}(d),\, 5 + J^*_{N-4}(b)\} = 18, \quad u^*_{N-5}(a) = \text{right}\\
&J^*_{N-5}(c) = \min\{5 + J^*_{N-4}(d),\, 3 + J^*_{N-4}(f)\} = 8, \quad u^*_{N-5}(c) = \text{right},\\
&J^*_{N-5}(b) = 9 + J^*_{N-4}(c) = 17, \quad u^*_{N-5}(b) = \text{right}.
\end{aligned}
$$

Additional iterations are not included in this example because the costs and optimal decisions will no longer change with longer horizons (see for yourself!). Therefore it can be seen that with a sufficiently long horizon ($N \geq 5$), the optimal path from a to h is $a-d-e-f-g-h$ and incurs a cost of 18. Note that this process has actually given a lot more information than what was originally asked for. In particular, given *any* starting point and *any* horizon it is straightforward to generate an optimal control sequence! For example, if you wanted to start at point c and get to h in $N = 3$ steps you could immediately see that the optimal path is $c-f-g-h$ and the optimal cost is 8.

20.2 *Stochastic Decision Making Problem*

In the stochastic decision making problem it is assumed that there is some *uncertainty* in the robot's behavior or in the environment. This uncertainty is captured in the stochastic discrete-time robot model:

$$\boldsymbol{x}_{k+1} = f_k(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w}_k), \quad k = 0, \ldots, N-1, \tag{20.5}$$

where $\boldsymbol{w}_k$ represents a stochastic disturbance term. Additionally, it is assumed that this disturbance has a known conditional probability distribution $P_k(\boldsymbol{w}_k \mid$

$\boldsymbol{x}_k, \boldsymbol{u}_k)$. Note that it is assumed that the disturbance is only dependent on the current state $\boldsymbol{x}_k$ and control $\boldsymbol{u}_k$, and not states from earlier in the robot's history.

Another main difference between the stochastic decision making problem and the deterministic problem is that a control *policy* is computed in the stochastic case. A control policy, usually denoted $\boldsymbol{u} = \pi(\boldsymbol{x})$, is a function that maps the state $\boldsymbol{x}$ to a control $\boldsymbol{u}$, and therefore defines a closed-loop controller (whereas in the deterministic setting an open-loop sequence was computed). Generally speaking, the search for control *policies* makes the problem more difficult to solve, but is typically required in stochastic settings because uncertainty would lead to undesirable behavior under open-loop control plans. Specifically, in the stochastic decision making problem the policies $\pi = \{\pi_0, \ldots, \pi_{N-1}\}$ are computed, which define the controls by $\boldsymbol{u}_k = \pi_k(\boldsymbol{x}_k)$.

Of course the cost function is also modified to handle the uncertainty. In particular, a *risk-neutral* formulation is used (i.e. minimize the cost *on average*), where the cost is defined by the *expected* value:

$$J_\pi(\boldsymbol{x}_0) = E_{\boldsymbol{w}}\big[g_N(\boldsymbol{x}_N) + \sum_{k=0}^{N-1} g_k(\boldsymbol{x}_k, \pi_k(\boldsymbol{x}_k), \boldsymbol{w}_k)\big], \tag{20.6}$$

where the expectation is over the stochastic variables $\boldsymbol{w}$. The stochastic decision making problem can now be stated as:

Definition 20.2.1 (Stochastic Decision Making Problem)**.** *The stochastic decision making problem can be expressed for the system model* (20.5)*, control constraints* (20.2)*, and cost function* (20.6) *as:*

$$J^*(\boldsymbol{x}_0) = \min_\pi \; J_\pi(\boldsymbol{x}_0). \tag{20.7}$$

20.2.1 Principle of Optimality (Stochastic)

The principle of optimality can again be applied in the stochastic setting, and the intuition is identical to the deterministic case (however the proof is slightly different because the reasoning is in terms of probability distributions). The principle of optimality in the stochastic setting is stated formally as:

Theorem 20.2.2 (Principle of Optimality (Stochastic))**.** *Let* $\pi^* = \{\pi_0^*, \pi_1^* \ldots, \pi_{N-1}^*\}$ *be an optimal policy for the stochastic decision making problem* (20.7)*, and assume the state* $\boldsymbol{x}_k$ *is reachable. Then, the tail policy sequence* $\{\pi_k^*, \ldots, \pi_{N-1}^*\}$ *is an optimal policy sequence when starting from* $\boldsymbol{x}_k$ *to minimize the cost from time k to time N.*

Again, by leveraging the principle of optimality the decision making problem can be simplified to making immediate decisions by concatenating optimal tail policies.

20.2.2 Dynamic Programming (Stochastic)

The dynamic programming algorithm for the stochastic setting is also quite similar to DP for deterministic problems, and is given in Algorithm 17. Once

Algorithm 17: Dynamic Programming (Stochastic)

$J_N(\boldsymbol{x}_N) = g_n(\boldsymbol{x}_N)$, for all $\boldsymbol{x}_N$
for $k = N-1$ **to** 0 **do**
  $J_k(\boldsymbol{x}_k) = \min_{\boldsymbol{u}_k \in \mathcal{U}(\boldsymbol{x}_k)} E_{\boldsymbol{w}_k}\left[g_k(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w}_k) + J_{k+1}(f_k(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w}_k))\right]$, for all $\boldsymbol{x}_k$
return $J_0(\cdot), \ldots, J_N(\cdot)$

Algorithm 17 is run, the optimal policy is defined by:

$$\pi_k^*(\boldsymbol{x}_k) = \arg\min_{\boldsymbol{u}_k \in \mathcal{U}(\boldsymbol{x}_k)} E_{\boldsymbol{w}_k}\left[g_k(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w}_k) + J_{k+1}(f_k(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w}_k))\right].$$

Example 20.2.1 (Stochastic Dynamic Programming)**.** Consider an inventory control problem, where the available stock of a particular item is the state $x_k \in \mathbb{N}$, the ability to add to the inventory is the control $u_k \in \mathbb{N}$, and the demand for the item is a stochastic variable $w_k \in \mathbb{N}$. The dynamics of the available stock is modeled as:

$$x_{k+1} = \max\{0, x_k + u_k - w_k\},$$

which models the fact that demand reduces available stock but can also never be negative. Additionally, consider the control constraints:

$$x_k + u_k \leq 2,$$

which limits the amount of additional inventory that can be added based on the current available stock to ensure that $x_k \leq 2$. The demand w_k is assumed to be modeled probabilistically with a distribution:

$$p(w_k = 0) = 0.1, \quad p(w_k = 1) = 0.7, \quad p(w_k = 2) = 0.2.$$

Finally, the cost is given for a horizon of $N = 3$ as:

$$E\Big[\sum_{k=0}^{2} u_k + (x_k + u_k - w_k)^2\Big],$$

which penalizes ordering new stock at each time step and also having available stock at the next time step (i.e. having to store stock).

The dynamic programming algorithm can then be applied, starting with the end costs:

$$J_3(x_3) = 0,$$

and then recursively computing:

$$J_2(0) = \min_{u_2 \in \{0,1,2\}} E\left[u_2 + (u_2 - w_2)^2\right] = \min_{u_2 \in \{0,1,2\}} u_2 + 0.1u_2^2 + 0.7(u_2 - 1)^2 + 0.2(u_2 - 2)^2 = 1.3,$$
$$J_2(1) = \min_{u_2 \in \{0,1\}} E\left[u_2 + (1 + u_2 - w_2)^2\right] = 0.3,$$
$$J_2(2) = E\left[(2 - w_2)^2\right] = 1.1,$$

where the last cost is easily evaluated since the constraint makes $u_2 = 0$ the only feasible choice. The optimal stage policies associated with this step are:

$$\pi_2^*(0) = 1,$$
$$\pi_2^*(1) = 0,$$
$$\pi_2^*(2) = 0.$$

In the next step:

$$J_1(0) = \min_{u_1 \in \{0,1,2\}} E\big[u_1 + (u_1 - w_1)^2 + J_2(\max\{0, u_1 - w_1\})\big] = 2.5,$$
$$J_1(1) = \min_{u_1 \in \{0,1,\}} E\big[u_1 + (1 + u_1 - w_1)^2 + J_2(\max\{0, 1 + u_1 - w_1\})\big] = 1.5,$$
$$J_1(2) = E\big[(2 - w_1)^2 + J_2(\max\{0, 2 - w_1\})\big] = 1.68,$$

with optimal stage policies:

$$\pi_1^*(0) = 1,$$
$$\pi_1^*(1) = 0,$$
$$\pi_1^*(2) = 0.$$

Finally, in the last step:

$$J_0(0) = \min_{u_0 \in \{0,1,2\}} E\big[u_0 + (u_0 - w_0)^2 + J_1(\max\{0, u_0 - w_0\})\big] = 3.7,$$
$$J_0(1) = \min_{u_0 \in \{0,1,\}} E\big[u_0 + (1 + u_0 - w_0)^2 + J_1(\max\{0, 1 + u_0 - w_0\})\big] = 2.7,$$
$$J_0(2) = E\big[(2 - w_0)^2 + J_1(\max\{0, 2 - w_0\})\big] = 2.818,$$

with optimal stage policies:

$$\pi_0^*(0) = 1,$$
$$\pi_0^*(1) = 0,$$
$$\pi_0^*(2) = 0.$$

Interestingly, the best scenario occurs with an initial stock of one, rather than have no stock or too much stock. Also, the policy ends up being the same at all time steps: if you have no stock you add one item, otherwise you do nothing.

20.3 Challenges and Extensions of Dynamic Programming

Dynamic programming is a powerful algorithm, but suffers from several practical considerations: the "curse of dimensionality", the "curse of modeling", and the "curse of time". The curse of dimensionality arises because of an exponential growth of the computational and storage requirements based on the dimension of the state. For example if the state has dimension one (i.e. $\boldsymbol{x} \in \mathbb{R}$) and can take on 100 different values, then at each step of the algorithm the Bellman equation must be solved 100 times. While this may be possible from a practical

perspective, if $\boldsymbol{x} \in \mathbb{R}^3$ this would lead to 100^3 solves of the Bellman equation! Additionally, extensions to the problems presented in this chapter where the full state is not *known* (e.g., because you can only measure some parts of the state), the problem also become intractable. The curse of modeling results from the complexity of modeling stochastic systems. In particular, it can be very hard to obtain expressions for transition probabilities for real world systems! Lastly, the curse of time is that the data of the problem may not be known ahead of time (such that the DP algorithm can be run offline). Therefore it may be required to solve the DP algorithm online when the data becomes available, or when the data changes and the problem needs to be resolved.

20.3.1 Reinforcement Learning

The practical challenges related to dynamic programming motivated the development of *suboptimal* dynamic programming approaches, which more commonly are referred to as *reinforcement learning* approaches. The goal of these approaches is to make *approximations* to the original problem that make it more practical for specific settings, such as with high-dimensional states, when the model is not known, and more. Broadly speaking, there are two main categories of approximations. The first category includes approximations in the value space (i.e. where the optimal cost function is approximated). The second category includes approximations in the policy space (i.e. where the policy is approximated by a neural network whose weights are optimized over).

21
Reinforcement Learning

The previous chapter introduced the deterministic and stochastic sequential decision making problems, and demonstrated how these problems can be solved by dynamic programming. While dynamic programming is a powerful algorithm, it also suffers from several practical challenges. This chapter briefly introduces some of the key ideas in *reinforcement learning*[1,2], a set of ideas which aims to solve a more general problem of behaving in an optimal way within a given *unknown* environment. That is, the reinforcement learning setting assumes only the ability to (1) interact with an unknown environment and (2) receive a reward signal from it. How the actions affect the future state evolution or the future reward is not known *a priori*. Reinforcement learning includes a class of approximation algorithms which can be much more practical than dynamic programming in real world applications.

[1] D. Bertsekas. *Reinforcement learning and optimal control*. Athena Scientific, 2019

[2] R. Sutton and A. Barto. *Reinforcement learning: An introduction*. MIT Press, 2018

Reinforcement Learning

Reinforcement learning (RL) is a broad field that studies autonomous sequential decision making, but extends to more general and challenging problems than have been considered in previous chapters. The standard RL problem is to determine closed-loop control policies that drive an agent to maximize an accumulated reward[3]. However, in the general case it is not required that a *system model* be known! This paradigm can be represented by Figure 21.1, where it can be seen that given a control input the environment specifies the next state and reward, and the environment can be considered to be a black box (it is not necessarily known how the state is generated, nor the reward computed).

[3] Note that the maximization of "reward" in the context of reinforcement learning is essentially equivalent to minimization of "cost" in optimal control formulations.

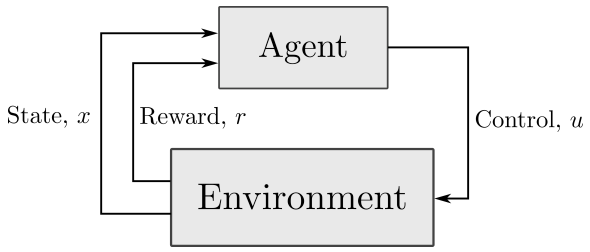


Figure 21.1: In reinforcement learning problems, the robot (agent) learns how to make decisions by interacting with the environment.

To account for this model uncertainty (which is notably distinct from the state transition uncertainty inherent in a stochastic but *known* system model, as considered in the previous chapter), an agent must instead learn from its experience to produce good policies. Concisely, RL deals with the problem of how to learn to act optimally in the long term, from interactions with an unknown environment which provides only a momentary reward signal.

RL is a highly active field of research, and has seen successes in several applications including acrobatic control of helicopters, games, finance, and robotics. In this chapter the fundamentals of reinforcement learning are introduced, including the formulation of the RL problem, RL algorithms that leverage system models ("model-based" methods; value iteration/dynamic programming and policy iteration), and a few RL algorithms that do not require system models ("model-free" methods; Q-learning, policy gradient, actor-critic).

21.1 Problem Formulation

The problem setting of reinforcement learning is similar to that of stochastic sequential decision making from the previous chapter, but here we will adopt slightly different notation more consistent with how Markov Decision Processes (MDPs) are typically framed in this community.[4] The state and control input for the system is denoted as $\boldsymbol{x}$ and $\boldsymbol{u}$, and the set of admissible states and controls are denoted as $\mathcal{X}$ and $\mathcal{U}$. However, the stochastic state transition model will now be written explicitly as a probability distribution (where before this was implicit in the influence of the stochastic variables $\boldsymbol{w}$ on the system dynamics f):

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}), \tag{21.1}$$

which is the conditional probability distribution over $\boldsymbol{x}_t$, given the previous state and control. The environment also has a reward function which defines the reward associated with every state and control

$$r_t = R(\boldsymbol{x}_t, \boldsymbol{u}_t). \tag{21.2}$$

The goal of the RL problem is to interact with the environment over a (possibly infinite) time horizon and *accumulate the highest possible reward in expectation*. To accommodate infinite horizon problems and to account for the fact that an agent is typically more confident about the ramifications of its actions in the short term than the long term, the accumulated reward is typically defined as the *discounted total expected reward* over time

$$E\left[\sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \boldsymbol{u}_t)\right], \tag{21.3}$$

where $\gamma \in (0, 1]$ is referred to as a *discount factor*. The tuple

$$\mathcal{M} = \left(\mathcal{X}, \mathcal{U}, p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}), R(\boldsymbol{x}_t, \boldsymbol{u}_t), \gamma\right)$$

[4] The fields of optimal control and reinforcement learning have significant overlap, but each community has developed its own standard notation. Most often, the state in the optimal control community is represented by $\boldsymbol{x}$ and in the RL community as $\boldsymbol{s}$. Similarly, in control theory the control input is $\boldsymbol{u}$ while in the RL community it is referred to as an action $\boldsymbol{a}$.

defines the Markov Decision Process (MDP), the environment in which the reinforcement learning problem is set.

In this chapter we will consider infinite horizon MDPs for which the notion of a stationary policy π applied at all time steps, i.e.,

$$\boldsymbol{u}_t = \pi(\boldsymbol{x}_t), \tag{21.4}$$

is appropriate. The goal of the RL problem is to choose a policy that maximizes the cumulative discounted reward

$$\pi^* = \arg\max_{\pi} \; E\left[\sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t)))\right] \tag{21.5}$$

where the expectation is notionally computed with respect to the stochastic dynamics p, but in practice is estimated empirically by drawing samples from the environment encoding $\mathcal{M}$ (i.e., in constructing π we may not assume exact knowledge of $\mathcal{M}$).

21.1.1 *Value function*

A policy π defines a value function which corresponds to the expected reward accumulated starting from a state $\boldsymbol{x}$

$$V^{\pi}(\boldsymbol{x}) = E\big[\sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \pi_t(\boldsymbol{x}_t)) \mid \boldsymbol{x}_0 = \boldsymbol{x}\big], \tag{21.6}$$

which can also be expressed in the *tail* formulation

$$V^{\pi}(\boldsymbol{x}) = R(\boldsymbol{x}, \pi(\boldsymbol{x})) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \pi(\boldsymbol{x})) V^{\pi}(\boldsymbol{x}'). \tag{21.7}$$

The optimal policy π^* satisfies *Bellman's equation*

$$\begin{aligned} V^{\pi^*}(\boldsymbol{x}) = V^*(\boldsymbol{x}) &= \max_{\boldsymbol{u} \in \mathcal{U}} \left(R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^*(\boldsymbol{x}') \right) \\ \pi^*(\boldsymbol{x}) &= \arg\max_{\boldsymbol{u} \in \mathcal{U}} \left(R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^*(\boldsymbol{x}') \right) \end{aligned} \tag{21.8}$$

and also satisfies $V^*(\boldsymbol{x}) = V^{\pi^*}(\boldsymbol{x}) \geq V^{\pi}(\boldsymbol{x})$ for all $\boldsymbol{x} \in \mathcal{X}$ for any alternative policy π. That is, the optimal policy induces the maximum value function and solves the RL problem of maximizing the accumulated discounted reward.

21.1.2 *Q-function*

Motivated by Bellman's equation above, in addition to the (state) value function $V^{\pi}(\boldsymbol{x})$ it makes sense to define the state-action value function $Q^{\pi}(\boldsymbol{x}, \boldsymbol{u})$ which corresponds to the expected reward accumulated starting from a state $\boldsymbol{x}$ and taking a first action $\boldsymbol{u}$ before following the policy π for all subsequent time steps. That is,

$$Q^{\pi}(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^{\pi}(\boldsymbol{x}'). \tag{21.9}$$

Similarly, the *optimal* Q-function is:

$$Q^*(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^*(\boldsymbol{x}'),$$

where the shorthand notation $Q^*(\boldsymbol{x}, \boldsymbol{u}) = Q^{\pi^*}(\boldsymbol{x}, \boldsymbol{u})$ is used. Note that from the Bellman equation (21.8) the optimal value function can be written as an optimization over the optimal Q-function:

$$V^*(\boldsymbol{x}) = \max_{\boldsymbol{u} \in \mathcal{U}} Q^*(\boldsymbol{x}, \boldsymbol{u}),$$

so,

$$Q^*(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) \max_{\boldsymbol{u}' \in \mathcal{U}} \Big(Q^*(\boldsymbol{x}', \boldsymbol{u}') \Big).$$

Therefore, instead of computing the optimal value function using value iteration it is possible to deal directly with the Q-function!

21.2 *Model-based Reinforcement Learning*

Model-based reinforcement learning methods rely on the use of an explicit parameterization of the transition model (21.1), which is either fit to observed transition data (i.e., learned) or, in special cases, known *a priori*. For example, for discrete state/control spaces it is possible to empirically approximate the transition probabilities $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1})$ for every pair $(\boldsymbol{x}_t, \boldsymbol{u}_t)$ by counting the number of times each transition occurs in the dataset! More sophisticated models include linear models generated through least squares, or Gaussian process or neural network models trained through an appropriate loss function. Given a learned model, the problem of optimal policy synthesis reduces to the sequential decision making problem of the previous chapter.

21.2.1 *Value Iteration (Dynamic Programming)*

While the dynamic programming algorithm was covered in the previous chapter, it will also be included here in the context of the RL problem formulation. In this case, the "principle of optimality" again says that the optimal *tail* policy is optimal for *tail* subproblems, which leads to the recursion:

$$V^*_{k+1}(\boldsymbol{x}) = \max_{\boldsymbol{u} \in \mathcal{U}} \Big(R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^*_k(\boldsymbol{x}') \Big), \tag{21.10}$$

which is commonly referred to as the *Bellman recursion*. In words, the optimal reward associated with starting at the state $\boldsymbol{x}$ and having $k+1$ steps to go can be found as an optimization over the immediate control by accounting for the (expected) optimal tail rewards. The full dynamic programming algorithm for solving the RL problem (21.5) is given in Algorithm 18.

In the context of RL, this procedure is commonly referred to as *value iteration* and in many cases it is assumed that the horizon N is infinite. For infinite-horizon problems the "value iteration" in Algorithm 18 is performed either over

Algorithm 18: Dynamic Programming/Value Iteration (RL)

$V_0^*(\boldsymbol{x}) = 0$, for all $\boldsymbol{x} \in \mathcal{X}$
for $k = 0$ **to** $N-1$ **do**
  $V_{k+1}^*(\boldsymbol{x}) = \max_{\boldsymbol{u}\in\mathcal{U}} R(\boldsymbol{x},\boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x},\boldsymbol{u}) V_k^*(\boldsymbol{x}')$, for all $\boldsymbol{x} \in \mathcal{X}$
  $\pi_{N-1-k}^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}\in\mathcal{U}} R(\boldsymbol{x},\boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x},\boldsymbol{u}) V_k^*(\boldsymbol{x}')$, for all $\boldsymbol{x} \in \mathcal{X}$
return $V_0^*(\cdot), \ldots, V_N^*(\cdot), \pi_0^*(\cdot), \ldots, \pi_{N-1}^*(\cdot)$

a finite-horizon (which yields an approximate solution), or until convergence to a stationary (i.e. time-invariant) optimal value function/policy[5].

[5] In the infinite horizon case, the optimal value function is unique and the optimal policy is stationary and deterministic, but not necessarily unique.

To solidify the relationship between value iteration in the context of RL and dynamic programming in the context of stochastic decision making from the previous chapter, the inventory control example from the previous chapter is revisited:

Example 21.2.1 (Inventory Control)**.** Consider again the inventory control problem from the previous chapter, where the available stock of a particular item is the state $x_t \in \mathbb{N}$, the control $u_t \in \mathbb{N}$ adds items to the inventory, the demand w_t is uncertain, and the dynamics and constraints are:

$$x_t = \max\{0, x_{t-1} + u_{t-1} - w_{t-1}\},$$
$$p(w=0) = 0.1, \quad p(w=1) = 0.7, \quad p(w=2) = 0.2.$$

and

$$x_t + u_t \leq 2.$$

Based on the dynamics, the probabilistic model (21.1) is given by:

$$p(x_t = \{0,1,2\} \mid x_{t-1} = 0, u_{t-1} = 0) = \{1,0,0\},$$
$$p(x_t = \{0,1,2\} \mid x_{t-1} = 0, u_{t-1} = 1) = \{0.9,0.1,0\},$$
$$p(x_t = \{0,1,2\} \mid x_{t-1} = 0, u_{t-1} = 2) = \{0.2,0.7,0.1\},$$
$$p(x_t = \{0,1,2\} \mid x_{t-1} = 1, u_{t-1} = 0) = \{0.9,0.1,0\},$$
$$p(x_t = \{0,1,2\} \mid x_{t-1} = 1, u_{t-1} = 1) = \{0.2,0.7,0.1\},$$
$$p(x_t = \{0,1,2\} \mid x_{t-1} = 2, u_{t-1} = 0) = \{0.2,0.7,0.1\},$$

where some transition values are not explicitly written due to the control constraints. Next, the reward function is defined as:

$$\begin{aligned} R(x_t, u_t) &= -E\left[u_t + (x_t + u_t - w_t)^2\right], \\ &= -\left(u_t + (x_t + u_t - E[w_t])^2 + Var(w_t)\right), \end{aligned}$$

and a discount factor of $\gamma = 1$ is used. As in the previous chapter, this reward penalizes (a negative reward is a penalty) ordering new stock and having available stock at the next time step (i.e. having to store stock).

Algorithm 18 can now be applied, starting with the value function with no steps to go:

$$V_0^*(x) = 0,$$

and then recursively computing:

$$V_1^*(0) = \max_{u \in \{0,1,2\}} -\left(u + (u - 1.1)^2 + 0.29\right) = -1.3,$$
$$V_1^*(1) = \max_{u_2 \in \{0,1\}} -\left(u + (1 + u - 1.1)^2 + 0.29\right) = -0.3,$$
$$V_1^*(2) = -\left((2 - 1.1)^2 + 0.29\right) = -1.1,$$

where $E[w] = 1.1$ and $Var(w) = 0.29$. The optimal stage policies associated with this step are:

$$\pi_{N-1}^*(0) = 1, \quad \pi_{N-1}^*(1) = 0, \quad \pi_{N-1}^*(2) = 0.$$

In the next step:

$$V_2^*(0) = \max_{u \in \{0,1,2\}} -\left(u + (u - 1.1)^2 + 0.29\right) + \sum_{x'} p(x' \mid x = 0, u) V_1^*(x') = -2.5,$$
$$V_2^*(1) = \max_{u \in \{0,1,\}} -\left(u + (1 + u - 1.1)^2 + 0.29\right) + \sum_{x'} p(x' \mid x = 1, u) V_1^*(x') = -1.5,$$
$$V_2^*(2) = -\left((2 - 1.1)^2 + 0.29\right) + \sum_{x'} p(x' \mid x = 2, u = 0) V_1^*(x') = -1.68,$$

with optimal stage policies:

$$\pi_{N-2}^*(0) = 1, \quad \pi_{N-2}^*(1) = 0, \quad \pi_{N-2}^*(2) = 0.$$

Finally, in the last step:

$$V_3^*(0) = \max_{u \in \{0,1,2\}} -\left(u + (u - 1.1)^2 + 0.29\right) + \sum_{x'} p(x' \mid x = 0, u) V_2^*(x') = -3.7,$$
$$V_3^*(1) = \max_{u \in \{0,1,\}} -\left(u + (1 + u - 1.1)^2 + 0.29\right) + \sum_{x'} p(x' \mid x = 1, u) V_2^*(x') = -2.7,$$
$$V_3^*(2) = -\left((2 - 1.1)^2 + 0.29\right) + \sum_{x'} p(x' \mid x = 2, u = 0) V_2^*(x') = -2.818,$$

with optimal stage policies:

$$\pi_{N-3}^*(0) = 1, \quad \pi_{N-3}^*(1) = 0, \quad \pi_{N-3}^*(2) = 0.$$

These results are, in fact, identical to the results from the example in the previous chapter! The only difference is the formulation of the problem in the RL framework instead of the stochastic decision making problem framework.

21.2.2 *Policy Iteration*

Another common algorithm that can be used to solve the reinforcement learning problem (21.5) is *policy iteration*. The main idea of policy iteration is that if the value function can be computed for any arbitrary finite horizon policy $\pi = \{\pi_0, \pi_1, \ldots, \pi_{N-1}\}$, then the policy can be incrementally improved to yield a better policy $\pi' = \{\pi'_0, \pi'_1, \ldots, \pi'_{N-1}\}$.

Policy Evaluation: The first key element of the policy iteration algorithm is *policy evaluation*, which is used to compute the value function $V_k^\pi(\boldsymbol{x})$ for a given policy π. Policy evaluation is based on the recursion:

$$V_{k+1}^\pi(\boldsymbol{x}) = R(\boldsymbol{x}, \pi(\boldsymbol{x})) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \pi(\boldsymbol{x})) V_k^\pi(\boldsymbol{x}'), \tag{21.11}$$

which is very similar to the Bellman equation (21.8) except that there is no optimization over the control (since it is fixed). The policy evaluation algorithm is given in Algorithm 19.

Algorithm 19: Policy Evaluation

Data: π
Result: $V_0^\pi(\cdot), \ldots, V_N^\pi(\cdot)$
$V_0^\pi(\boldsymbol{x}) = 0$, for all $\boldsymbol{x} \in \mathcal{X}$
for $k = 0$ **to** $N-1$ **do**
  $V_{k+1}^\pi(\boldsymbol{x}) = R(\boldsymbol{x}, \pi_{N-1-k}(\boldsymbol{x})) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \pi_{N-1-k}(\boldsymbol{x})) V_k^\pi(\boldsymbol{x}')$, for all $\boldsymbol{x} \in \mathcal{X}$
return $V_0^\pi(\cdot), \ldots, V_N^\pi(\cdot)$

In the infinite-horizon case where a stationary policy is used, the iteration in Algorithm 19 stops when the value function has converged to its stationary value. Indeed, since the infinite horizon value function is the stationary point of this recursion, it is possible to directly solve for it by setting both $V_{k+1}^\pi = V_k^\pi = V_\infty^\pi$ in (21.11). In the case of a discrete state space with N possible states, this creates a linear system of N equations which can be used to solve for V_∞^π directly.

Policy Iteration Algorithm: The policy iteration algorithm incrementally updates the policy by performing local optimizations of the Q-function. In particular, a single iteration of the policy update is shown in Algorithm 20. It can be proven

Algorithm 20: Policy Iteration Step

Data: π
Result: π'
$V_0^\pi(\cdot), \ldots, V_N^\pi(\cdot) \leftarrow$ PolicyEvaluation(π)
for $k = 0$ **to** $N-1$ **do**
  $Q_{k+1}^\pi(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_k^\pi(\boldsymbol{x}')$ for all $\boldsymbol{x} \in \mathcal{X}$
  $\pi'_{N-1-k}(\boldsymbol{x}) = \arg\max_{\boldsymbol{u} \in \mathcal{U}} Q_{k+1}^\pi(\boldsymbol{x}, \boldsymbol{u})$, for all $\boldsymbol{x} \in \mathcal{X}$
return $\pi' = \{\pi'_0, \ldots, \pi'_{N-1}\}$

theoretically that under the policy iteration algorithm the value function is monotonically increasing with each new policy, and the procedure is run until convergence. While policy iteration and value iteration are quite similar, policy iteration can end up converging faster in some cases.

21.3 Model-free Reinforcement Learning

The value and policy iteration algorithms are applicable only to problems where the model $\mathcal{M}$ is *known*, i.e., they rely on direct access to the probabilistic system dynamics $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1})$ and reward function $R(\boldsymbol{x}_t, \boldsymbol{u}_t)$, or at least learned approximations of these functions fit to observed data. Model-free RL algorithms, on the other hand, sidestep the explicit consideration of p and R entirely.

21.3.1 Q-Learning

The canonical model-free reinforcement learning algorithm is *Q-learning*. The core idea behind Q-learning is that it is possible to collect data samples $(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1})$ from interaction with the environment, and over time learn the long-term value of taking certain actions in certain states, i.e., directly learning the optimal Q-function $Q^*(\boldsymbol{x}, \boldsymbol{u})$. For simplicity an infinite-horizon ($N = \infty$) problem will be considered, such that the optimal value and Q-functions will be stationary, and in particular the optimal Q-function will satisfy:

$$Q^*(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) \max_{\boldsymbol{u}' \in \mathcal{U}} Q^*(\boldsymbol{x}', \boldsymbol{u}').$$

In a model-free context, the dynamics p above are notional (i.e., the problem is described by some MDP $\mathcal{M}$, we just don't know exactly what it is).[6] We may instead rewrite the above equation in terms of an expectation over trajectory samples drawn from p (i.e., drawn from the environment as a "black box") while implementing the policy $\boldsymbol{u}_t = \pi^*(\boldsymbol{x}_t)$:

[6] Or even if we have a environment simulator, in which case it could be argued that the dynamics are exactly described by the simulation code, the dynamics are too complex/opaque to be considered in this form.

$$Q^*(\boldsymbol{x}_t, \boldsymbol{u}_t) = E\big[r_t + \gamma \max_{\boldsymbol{u}' \in \mathcal{U}} Q^*(\boldsymbol{x}_{t+1}, \boldsymbol{u}')\big],$$

or equivalently,

$$E\left[\left(r_t + \gamma \max_{\boldsymbol{u}' \in \mathcal{U}} Q^*(\boldsymbol{x}_{t+1}, \boldsymbol{u}')\right) - Q^*(\boldsymbol{x}, \boldsymbol{u})\right] = 0,$$

where $(r_t + \gamma \max_{\boldsymbol{u}' \in \mathcal{U}} Q^*(\boldsymbol{x}_{t+1}, \boldsymbol{u}')) - Q^*(\boldsymbol{x}, \boldsymbol{u})$ is known as the *temporal difference error*. The idea of Q-learning is that an approximation of the optimal Q-function can be improved over time by collecting data and trying to ensure that the above conditions holds. This leads to the Q-learning algorithm described in Algorithm 21. The iterations of Q-learning, each a local deterministic correction to the Q-function, in aggregate aim to ensure that the expected temporal difference error is 0.

Q-learning is referred to as a *model-free* method because it forgoes explicitly estimating the true (unknown) system dynamics, and directly estimates the Q-function. It is also called a *value-based* model-free method since it does not directly build the policy, but rather estimates the optimal Q-function to implicitly define the policy. Q-learning is also called an *off-policy* algorithm because the Q-function can be learned from stored experiences and does not require interacting with the environment directly.

Algorithm 21: Q-learning

Data: Set $\mathcal{S}$ of trajectory samples $\{\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1}\}$, learning rate α
Result: $Q(\boldsymbol{x}, \boldsymbol{u})$
Initialize $Q(\boldsymbol{x}, \boldsymbol{u})$ for all $\boldsymbol{x} \in \mathcal{X}$ and $\boldsymbol{u} \in \mathcal{U}$
for $\{\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1}\} \in \mathcal{S}$ **do**
  $Q(\boldsymbol{x}_t, \boldsymbol{u}_t) \leftarrow Q(\boldsymbol{x}_t, \boldsymbol{u}_t) + \alpha\Big(r_t + \gamma \max_{\boldsymbol{u} \in \mathcal{U}} Q(\boldsymbol{x}_{t+1}, \boldsymbol{u}) - Q(\boldsymbol{x}_t, \boldsymbol{u}_t)\Big)$
return $Q(\boldsymbol{x}, \boldsymbol{u})$

Q-learning can be guaranteed to converge to the optimal Q-function under certain conditions, but has some practical disadvantages. In particular, unless the number of possible states and controls are finite and relatively small, it can be intractable to store the Q-value associated with each state-control pair. Another disadvantage of Q-learning is that sometimes the Q-function can be complex and therefore potentially hard to learn.

Fitted Q-learning: One variation of the Q-learning algorithm to handle large or continuous state and control spaces is to parameterize the Q-function as $Q_\theta(\boldsymbol{x}, \boldsymbol{u})$ and to simply update the parameters $\boldsymbol{\theta}$. This approach is also known as *fitted Q-learning*. While this method often works well in practice, convergence is not guaranteed.

A principled way of performing fitted Q-learning involves minimizing the expected squared temporal difference error for the Q-function

$$E\left[\left(\left(r_t + \gamma \max_{u' \in \mathcal{U}} Q_\theta(\boldsymbol{x}_{t+1}, \boldsymbol{u}')\right) - Q_\theta(\boldsymbol{x}_t, \boldsymbol{u}_t)\right)^2\right].$$

For a given parameterization θ fitted Q-learning minimizes the total temporal difference error over all collected transition samples

$$\theta^* = \arg\min_\theta \frac{1}{|S_{\text{exp}}|} \sum_{(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{x}_{t+1}, r_t) \in S_{\text{exp}}} \left(r_t + \gamma \max_{u' \in \mathcal{U}} Q_\theta(\boldsymbol{x}_{t+1}, \boldsymbol{u}') - Q_\theta(\boldsymbol{x}_t, \boldsymbol{u}_t)\right)^2$$

where S_{exp} denotes the experience set of all transition tuples with a reward signal. This minimization is typically performed using stochastic gradient descent, yielding the parameter update

$$\theta \leftarrow \theta + \alpha \left(r_t + \gamma \max_{u' \in \mathcal{U}} Q_\theta(\boldsymbol{x}_{t+1}, \boldsymbol{u}') - Q_\theta(\boldsymbol{x}_t, \boldsymbol{u}_t)\right) \nabla_\theta Q_\theta(\boldsymbol{x}_t, \boldsymbol{u}_t)$$

applied iteratively for each $(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{x}_{t+1}, r_t) \in S_{\text{exp}}$.

21.3.2 Policy Gradient

The *policy gradient* method is another algorithm for model-free reinforcement learning. This approach, which directly optimizes the policy, can be particu-

larly useful for scenarios where the optimal policy may be relatively simple compared to the Q-function, in which case Q-learning may be challenging.

In the policy gradient approach, a class of *stochastic*[7] candidate policies $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t)$ is defined based on a set of parameters $\boldsymbol{\theta}$, and the goal is to *directly* modify the parameters $\boldsymbol{\theta}$ to improve performance. This is accomplished by using trajectory data to estimate a gradient of the performance with respect to the policy parameters $\boldsymbol{\theta}$, and then using the gradient to update $\boldsymbol{\theta}$. Because this method works directly on a policy (and does not learn a model or value function), it is referred to as a *model-free policy-based* approach.

[7] A stochastic policy defines a distribution over actions at a given state, is useful for exploration, and sometimes is even required for optimality.

Considering the original problem (21.5), the objective function can be written as:

$$J(\boldsymbol{\theta}) = E\big[\sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t))\big],$$

where the $J(\boldsymbol{\theta})$ notation is used to explicitly show the dependence on the parameters. Implementing a policy gradient approach therefore requires the computation of $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. One of the most common approaches is to *estimate* this quantity using data, using what is known as a *likelihood ratio method*.

Let τ represent a *trajectory* of the system (consisting of sequential states and actions) under the current policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t)$. As a shorthand notation, consider the total discounted reward over a trajectory τ to be defined written as:

$$r(\tau) = \sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t), \tag{21.12}$$

such that $J(\boldsymbol{\theta})$ can be expressed equivalently as $J(\boldsymbol{\theta}) = E\big[r(\tau)\big]$. Additionally, let the probability that the trajectory τ occurs be expressed by the distribution $p_{\boldsymbol{\theta}}(\tau)$. Then the expectation from the objective function can be expanded as:

$$J(\boldsymbol{\theta}) = \int_{\tau} r(\tau) p_{\boldsymbol{\theta}}(\tau) d\tau,$$

and its gradient given by:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \int_{\tau} r(\tau) p_{\boldsymbol{\theta}}(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) d\tau.$$

From standard calculus $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) = \frac{1}{p_{\boldsymbol{\theta}}(\tau)} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau)$, which replaces the use of the gradient $\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau)$ with $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)$. This is a very useful "trick" when it comes to approximately computing the integral, as will be seen shortly.

Rather than explicitly computing this integral it is much easier to approximate using sampled data (i.e. sampled trajectories). This is possible since the integral can be written as the expectation $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = E\big[r(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)\big]$, which can be estimated using a Monte Carlo method. While in general a number of sampled trajectories could be used to estimate the gradient, for data efficiency it is also possible to just use a single sampled trajectory τ and approximate:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx r(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau). \tag{21.13}$$

In particular the sampled quantities $r(\tau)$ can be directly computed from (21.12), and it turns out that the term $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)$ can be evaluated quite easily as[8]:

$$\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) = \sum_{t=0}^{N-1} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t). \tag{21.14}$$

[8] Using Bayes' rule: $p_{\boldsymbol{\theta}}(\tau) = p(\boldsymbol{x}_0) \Pi_{t=1}^{N-1} p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}) \pi_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{u}_{t-1})$. Then the log converts the product into a sum.

Importantly, notice that only the gradient of the policy is needed, and knowledge of the transition model $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1})$ is not! This occurs because only the policy is dependent on the parameters $\boldsymbol{\theta}$.

In summary, the gradient of $J(\theta)$ can be approximated given a trajectories τ under the current policy $\pi_{\boldsymbol{\theta}}$ by:

1. Compute $r(\tau)$ for the sampled trajectory using (21.12).

2. Compute $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)$ for the sampled trajectory using (21.14), which only requires computing gradients related to the current policy $\pi_{\boldsymbol{\theta}}$.

3. Approximate $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ using (21.13).

The process of sampling trajectories from the current policy, approximating the gradient, and performing a gradient descent step on the parameters $\boldsymbol{\theta}$ is referred to as the *REINFORCE* algorithm[9].

In general, policy-based RL methods such as policy gradient can converge more easily than value-based methods, can be effective in high-dimensional or continuous action spaces, and can learn stochastic policies. However, one challenge with directly learning policies is that they can get trapped in undesirable local optima. Policy gradient methods can also be data inefficient since they require data from the *current* policy for each gradient step and cannot easily reuse old data. This is in contrast to Q-learning, which is agnostic to the policy used and therefore doesn't waste data collected from past interactions.

[9] There are some other modified versions of this algorithm, for example some contain a baseline term $b(\boldsymbol{x}_0)$ in the gradient by replacing $r(\tau)$ with $r(\tau) - b(\boldsymbol{x}_0)$ to reduce the variance of the Monte Carlo estimate.

21.3.3 Actor-Critic

Another popular reinforcement learning algorithm is the *actor-critic* algorithm, which blends the concepts of value-based and policy-based model-free RL. In particular, a parameterized policy $\pi_{\boldsymbol{\theta}}$ (actor) is learned through a policy gradient method along side an estimated value function for the policy (critic). The addition of the critic helps to reduce the variance in the gradient estimates for the actor policy, which makes the overall learning process more data-efficient[10].

In particular, the policy $\pi_{\boldsymbol{\theta}}$ is again learned through policy gradient like in the REINFORCE algorithm, but with the addition of a learned approximation of the value function $V_{\boldsymbol{\phi}}(\boldsymbol{x})$ as a baseline:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \sum_{t=0}^{N-1} \big(r(\tau) - V_{\boldsymbol{\phi}}(\boldsymbol{x}_0)\big) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t).$$

[10] This is a similar variance reduction approach to adding a baseline $b(\boldsymbol{x}_\tau)$ to the REINFORCE. In fact the baseline is chosen as the value function!

Recall that the value function $V(\boldsymbol{x})$ quantifies the expected total return starting from state $\boldsymbol{x}$ (i.e. the average performance). Therefore the quantity $r(\tau) - V_{\boldsymbol{\phi}}(\boldsymbol{x}_0)$ now represents a performance increase over average. Of course in this method the learned value function approximation $V_{\boldsymbol{\phi}}(\boldsymbol{x})$ is also updated along with the policy by performing a similar gradient descent on the parameters $\boldsymbol{\phi}$.

21.4 Deep Reinforcement Learning

Neural networks are a powerful function approximator that can be utilized in reinforcement learning algorithms.

Q-learning: In Q-learning the Q-function can be approximated by a neural network to extend the approach to nonlinear, continuous state space domains.

Policy Gradient: In policy gradient methods, the policy π_θ can be parameterized as a neural network, enabling the policy to operate on high-dimensional states including images (i.e. visual feedback)!

Actor-Critic: In actor-critic methods, both the policy π_θ and the value function V_ϕ can be parameterized as a neural network which often leads to a space efficient nonlinear representations of the policy and the value function.

21.5 Exploration vs Exploitation

When learning from experience (e.g. using Q-learning, policy gradient, actor-critic, deep RL, etc.) it is important to ensure that the experienced trajectories (i.e. the collected data points) are meaningful! For example, an abundance of data related to a particular set of actions/states will not necessarily be sufficient to learn good policies for *all* possible situations. Therefore an important part of reinforcement learning is *exploring* different combinations of states and actions. One simple approach to exploration is referred to as ϵ-greedy exploration, where a random control is applied instead of the current (best) policy with probability ϵ.

However, exploration can lead to suboptimal performance since any knowledge accumulated about the optimal policy is ignored[11]. This leads to the *exploration vs exploitation* trade-off: a fundamental challenge in reinforcement learning.

[11] In other words, actions with known rewards may be foregone in the hope that exploring leads to an even better reward.

22
Imitation Learning

As discussed in the previous chapter, the goal of reinforcement learning is to determine closed-loop control policies that result in the maximization of an accumulated reward, and RL algorithms are generally classified as either model-based or model-free. In both cases it is generally assumed that the reward function is known, and both typically rely on collecting system data to either update a learned model (model-based), or directly update a learned value function or policy (model-free).

While successful in many settings, these approaches to RL also suffer from several drawbacks. First, determining an appropriate reward function that can accurately represent the true performance objectives can be challenging[1]. Second, rewards may be *sparse*, which makes the learning process expensive in terms of both the required amount of data and in the number of failures that may be experienced when exploring with a suboptimal policy[2]. This chapter introduces the *imitation learning* approach to RL, where a reward function is not assumed to be known *a priori* but rather it is assumed the reward function is described implicitly through expert demonstrations.

[1] RL agents can sometimes learn how to exploit a reward function without actually producing the desired behavior. This is commonly referred to as *reward hacking*. Consider training an RL agent with a reward for each piece of trash collected. Rather than searching the area to find more trash (the desired behavior), the agent may decide to throw the trash back onto the ground and pick it up again!

[2] This issue of sparse rewards is less relevant if data is cheap, for example when training in simulation.

Imitation Learning

The formulation of the imitation learning problem is quite similar to the RL problem formulation from the previous chapter. The main difference is that instead of leveraging an explicit reward function $r_t = R(\boldsymbol{x}_t, \boldsymbol{u}_t)$ it will be assumed that a set of demonstrations from an expert are provided.

22.1 *Problem Formulation*

It will be assumed that the system is a Markov Decision Process (MDP) with a state $\boldsymbol{x}$ and control input $\boldsymbol{u}$, and the set of admissible states and controls are denoted as $\mathcal{X}$ and $\mathcal{U}$. The system dynamics are expressed by the probabilistic transition model:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}), \tag{22.1}$$

The field of RL often uses $\boldsymbol{s}$ to express the state and $\boldsymbol{a}$ to represent an action, but $\boldsymbol{x}$ and $\boldsymbol{u}$ will be used here for consistency with previous chapters.

which is the conditional probability distribution over $\boldsymbol{x}_t$, given the previous state and control. As in the previous chapter, the goal is to define a *policy* π that defines the closed-loop control law[3]:

[3] This chapter will consider a stationary policy for simplicity.

$$\boldsymbol{u}_t = \pi(\boldsymbol{x}_t). \tag{22.2}$$

The primary difference in formulation from the previous RL problem is that we do not have access to the reward function, and instead we have access to a set of expert demonstrations where each demonstration ξ consists of a sequence of state-control pairs:

$$\xi = \{(\boldsymbol{x}_0, \boldsymbol{u}_0), (\boldsymbol{x}_1, \boldsymbol{u}_1), \dots\}, \tag{22.3}$$

which are drawn from the expert policy π^*. The imitation learning problem is therefore to determine a policy π that imitates the expert policy π^*:

Definition 22.1.1 (Imitation Learning Problem)**.** *For a system with transition model* (22.1) *with states* $\boldsymbol{x} \in \mathcal{X}$ *and controls* $\boldsymbol{u} \in \mathcal{U}$*, the imitation learning problem is to leverage a set of demonstrations* $\Xi = \{\xi_1, \dots, \xi_D\}$ *from an expert policy* π^* *to find a policy* $\hat{\pi}^*$ *that imitates the expert policy.*

There are generally two approaches to imitation learning: the first is to directly learn how to imitate the expert's policy and the second is to indirectly imitate the policy by instead learning the expert's reward function. This chapter will first introduce two classical approaches to imitation learning (*behavior cloning* and the *DAgger* algorithm) that focus on directly imitating the policy. Then a set of approaches for learning the expert's reward function will be discussed, which is commonly referred to as *inverse reinforcement learning*. The chapter will then conclude with a couple of short discussions into related topics on learning from experts (e.g. through comparisons or physical feedback) as well as on interaction-aware control.

22.2 *Behavior Cloning*

Behavior cloning approaches use a set of expert demonstrations $\xi \in \Xi$ to determine a policy π that imitates the expert. This can be accomplished through supervised learning techniques, where the difference between the learned policy and expert demonstrations are minimized with respect to some metric. Concretely, the goal is to solve the optimization problem:

$$\hat{\pi}^* = \arg\min_{\pi} \sum_{\xi \in \Xi} \sum_{\boldsymbol{x} \in \xi} L(\pi(\boldsymbol{x}), \pi^*(\boldsymbol{x})),$$

where L is the cost function[4], $\pi^*(\boldsymbol{x})$ is the expert's action for at the state $\boldsymbol{x}$, and $\hat{\pi}^*$ is the approximated policy.

[4] Different loss functions could include p-norms (e.g. Euclidean norm) or f-divergences (e.g. KL divergence) depending on the form of the policy.

However this approach may not yield very good performance since the learning process is only based on a set of samples provided by the expert. In many cases these expert demonstrations will not be uniformly sampled across the

entire state space and therefore it is likely that the learned policy will perform poorly when not close to states found in ξ. This is particularly true when the expert demonstrations come from a *trajectory* of sequential states and actions, such that the *distribution* of the sampled states $\boldsymbol{x}$ in the dataset is defined by the expert policy. Then, when an estimated policy $\hat{\pi}^*$ is used in practice it produces its own distribution of states that will be visited, which will likely not be the same as in the expert demonstrations! This distributional mismatch leads to compounding errors, which is a major challenge in imitation learning.

22.3 DAgger: Dataset Aggregation

One straightforward idea for addressing the issue of distributional mismatch in states seen under the expert policy and the learned policy is to simply collect new expert data as needed[5]. In other words, when the learned policy $\hat{\pi}^*$ leads to states that aren't in the expert dataset just query the expert for more information! The behavioral cloning algorithm that leverages this idea is known as DAgger[6] (Dataset Aggregation).

[5] Assuming the expert can be queried on demand.

[6] S. Ross, G. Gordon, and D. Bagnell. "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011, pp. 627–635

Algorithm 22: DAgger: Dataset Aggregation

Data: π^*
Result: $\hat{\pi}^*$
$\mathcal{D} \leftarrow 0$
Initialize $\hat{\pi}$
for $i = 1$ **to** N **do**
  $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}$
  Rollout policy π_i to sample trajectory $\tau = \{\boldsymbol{x}_0, \boldsymbol{x}_1, \dots\}$
  Query expert to generate dataset $\mathcal{D}_i = \{(\boldsymbol{x}_0, \pi^*(\boldsymbol{x}_0)), (\boldsymbol{x}_1, \pi^*(\boldsymbol{x}_1)), \dots\}$
  Aggregate datsets, $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$
  Retrain policy $\hat{\pi}$ using aggregated dataset $\mathcal{D}$
return $\hat{\pi}$

As can be seen in Algorithm 22, this approach iteratively improves the learned policy by collecting additional data from the expert. This is accomplished by rolling out the current learned policy for some number of time steps and then asking the expert what actions they would have taken at each step along that trajectory. Over time this process drives the learned policy to better approximate the true policy and reduce the incidence of distributional mismatch. One disadvantage to the approach is that at each step the policy needs to be retrained, which may be computationally inefficient.

22.4 Inverse Reinforcement Learning

Approaches that learn policies to imitate expert actions can be limited by several factors:

1. Behavior cloning provides no way to understand the underlying reasons for the expert behavior (no reasoning about outcomes or intentions).
2. The "expert" may actually be suboptimal[7].
3. A policy that is optimal for the expert may not be optimal for the agent if they have different dynamics, morphologies, or capabilities.

[7] Although the discussion of inverse RL in this section will also assume the expert is optimal, there exist approaches to remove this assumption.

An alternative approach to behavioral cloning is to reason about and try to learn a representation of the underlying reward function R that the expert was using to generate its actions. By learning the expert's intent, the agent can potentially outperform the expert or adjust for differences in capabilities[8]. This approach (learning reward functions) is known as *inverse reinforcement learning*.

[8] Learned reward representations can potentially generalize across different robot platforms that tackle similar problems!

Inverse RL approaches assume a specific parameterization of the reward function, and in this section the fundamental concepts will be presented by parameterizing the reward as a linear combination of (nonlinear) features:

$$R(\boldsymbol{x}, \boldsymbol{u}) = \boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}, \boldsymbol{u}),$$

where $\boldsymbol{w} \in \mathbb{R}^n$ is a weight vector and $\boldsymbol{\phi}(\boldsymbol{x}, \boldsymbol{u}) : \mathcal{X} \times \mathcal{U} \rightarrow \mathbb{R}^n$ is a feature map. For a given feature map $\boldsymbol{\phi}$, the goal of inverse RL can be simplified to determining the weights $\boldsymbol{w}$. Recall from the previous chapter on RL that the total (discounted) reward under a policy π is defined for a time horizon T as:

$$V_T^\pi(\boldsymbol{x}) = E\big[\sum_{t=0}^{T-1} \gamma^t R(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t)) \mid \boldsymbol{x}_0 = \boldsymbol{x}\big].$$

Using the reward function $R(\boldsymbol{x}, \boldsymbol{u}) = \boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}, \boldsymbol{u})$ this value function can be expressed as:

$$V_T^\pi(\boldsymbol{x}) = \boldsymbol{w}^\top \mu(\pi, \boldsymbol{x}), \quad \mu(\pi, \boldsymbol{x}) = E_\pi\big[\sum_{t=0}^{T-1} \gamma^t \boldsymbol{\phi}(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t)) \mid \boldsymbol{x}_0 = \boldsymbol{x}\big],$$

where $\mu(\pi, \boldsymbol{x})$ is defined by an expectation over the trajectories of the system under policy π (starting from state $\boldsymbol{x}$) and is referred to as the *feature expectation*[9]. One insight that can now be leveraged is that by definition the optimal expert policy π^* will always produce a greater value function:

$$V_T^{\pi^*}(\boldsymbol{x}) \geq V_T^\pi(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \mathcal{X}, \quad \forall \pi,$$

[9] Feature expectations are often computed using a Monte Carlo technique (e.g. using the set of demonstrations for the expert policy).

which can be expressed in terms of the feature expectation as:

$$\boldsymbol{w}^{*\top} \mu(\pi^*, \boldsymbol{x}) \geq \boldsymbol{w}^{*\top} \mu(\pi, \boldsymbol{x}), \quad \forall \boldsymbol{x} \in \mathcal{X}, \quad \forall \pi. \tag{22.4}$$

Theoretically, identifying the vector $\boldsymbol{w}^*$ associated with the expert policy can be accomplished by finding a vector $\boldsymbol{w}$ that satisfies this condition. However this can potentially lead to ambiguities! For example, the choice $\boldsymbol{w} = 0$ satisfies this condition trivially! In fact, reward ambiguity is one of the main challenges associated with inverse reinforcement learning[10]. The algorithms discussed in the following chapters will propose techniques for alleviating this issue.

[10] A. Ng and S. Russell. "Algorithms for Inverse Reinforcement Learning". In: *Proceedings of the Seventeenth International Conference on Machine Learning*. 2000, pp. 663–670

22.4.1 *Apprenticeship Learning*

The apprenticeship learning[11] algorithm attempts to avoid some of the problems with reward ambiguity by leveraging an additional insight from condition (22.4). Specifically, the insight is that it doesn't matter how well $\boldsymbol{w}^*$ is estimated as long as a policy π can be found that *matches the feature expectations*. Mathematically, this conclusion is derived by noting that:

$$\|\mu(\pi, \boldsymbol{x}) - \mu(\pi^*, \boldsymbol{x})\|_2 \leq \epsilon \implies |\boldsymbol{w}^\top \mu(\pi, \boldsymbol{x}) - \boldsymbol{w}^\top \mu(\pi^*, \boldsymbol{x})| \leq \epsilon$$

for any $\boldsymbol{w}$ as long as $\|\boldsymbol{w}\|_2 \leq 1$. In other words, as long as the feature expectations can be matched then the performance will be as good as the expert *even if the vector $\boldsymbol{w}$ does not match $\boldsymbol{w}^*$*. Another practical aspect to the approach is that it will be assumed that the initial state $\boldsymbol{x}_0$ is drawn from a distribution D such that the value function is also considered in expectation as:

$$E_{\boldsymbol{x}_0 \sim D}\big[V_T^\pi(\boldsymbol{x}_0)\big] = \boldsymbol{w}^\top \mu(\pi), \quad \mu(\pi) = E_\pi\big[\sum_{t=0}^{T-1} \gamma^t \boldsymbol{\phi}(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t))\big].$$

This is useful to avoid having to consider all $\boldsymbol{x} \in \mathcal{X}$ when matching features[12].

To summarize, the goal of the apprenticeship learning approach is to find a policy π that matches the feature expectations with respect to the expert policy (i.e. makes $\mu(\pi)$ as similar as possible to $\mu(\pi^*)$)[13]. This is accomplished through Algorithm 23, which uses an iterative approach to finding better policies.

Algorithm 23: Apprenticeship Learning

Data: $\mu(\pi^*)$, ϵ
Result: $\hat{\pi}^*$
Initialize policy π_0
for $i = 1$ **to** $\ldots$ **do**
  Compute $\mu(\pi_{i-1})$ (or approximate via Monte Carlo)
  Solve problem (22.5) with policies $\{\pi_0, \ldots, \pi_{i-1}\}$ to compute $\boldsymbol{w}_i$ and t_i

$$(\boldsymbol{w}_i, t_i) = \arg\max_{\boldsymbol{w}, t} \; t, \quad \text{s.t. } \boldsymbol{w}^\top \mu(\pi^*) \geq \boldsymbol{w}^\top \mu(\pi) + t, \quad \forall \pi \in \{\pi_0, \ldots, \pi_{i-1}\}, \quad \|\boldsymbol{w}\|_2 \leq 1. \tag{22.5}$$

  if $t_i \leq \epsilon$ **then**
    $\hat{\pi}^* \leftarrow$ best feature matching policy from $\{\pi_0, \ldots, \pi_{i-1}\}$
    return $\hat{\pi}^*$
  Use RL to find an optimal policy π_i for reward function defined by $\boldsymbol{w}_i$

To better understand this algorithm it is useful to further examine the optimization problem (22.5)[14]. Suppose that instead of making $\boldsymbol{w}$ a decision variable

[11] P. Abbeel and A. Ng. "Apprenticeship Learning via Inverse Reinforcement Learning". In: *Proceedings of the Twenty-First International Conference on Machine Learning*. 2004

[12] Trying to find a policy that matches features for every possible starting state $\boldsymbol{x}$ is likely intractable or even infeasible.

[13] See Example 22.4.1 for an example of why matching features is intuitively useful.

[14] This problem can be thought of as an inverse RL problem that is seeking to find the reward function vector $\boldsymbol{w}$ such that the expert *maximally outperforms* the other policies.

it was actually fixed, then the resulting optimization would be:

$$t^*(\boldsymbol{w}) = \max_t \; t,$$
$$\text{s.t. } \boldsymbol{w}^\top \mu(\pi^*) \geq \boldsymbol{w}^\top \mu(\pi) + t, \quad \forall \pi \in \{\pi_0, \pi_1, \dots\},$$

which is essentially computing the smallest performance loss among the candidate policies $\{\pi_0, \pi_1, \dots\}$ with respect to the expert policy, *assuming the reward function weights are* $\boldsymbol{w}$. If $\boldsymbol{w}$ was known, then if $t^*(\boldsymbol{w}) \leq \epsilon$ it would guarantee that one of the candidate policies would effectively perform as well as the expert.

Since $\boldsymbol{w}$ is not known, the actual optimization problem (22.5) maximizes the smallest performance loss across *all vectors* $\boldsymbol{w}$ *with* $\|\boldsymbol{w}\|_2 \leq 1$. Therefore, if $t_i \leq \epsilon$ (i.e. the termination condition in Algorithm 23), then there must be a candidate policy whose performance loss is small *for all possible choices of* $\boldsymbol{w}$! In other words, there is a candidate policy that matches feature expectations well enough that good performance can be guaranteed without assuming the reward function is known, and without attempting to estimate the reward accurately.

Example 22.4.1 (Apprenticeship Learning vs. Behavioral Cloning)**.** Consider a problem where the goal is to drive a car across a city in as short of time as possible. In the imitation learning formulation it is assumed that the reward function is not known, but that there is an expert who shows how to drive across the city (i.e. what routes to take). A behavioral cloning approach would simply try to mimic the actions taken by the expert, such as memorizing that whenever the agent is at a particular intersection it should turn right. Of course this approach is not robust when at intersections that the expert never visited!

The apprenticeship learning approach tries to avoid the inefficiency of behavioral cloning by instead identifying features of the expert's trajectories that are more generalizable, and developing a policy that experiences the same feature expectations as the expert. For example it could be more efficient to notice that the expert takes routes without stop signs, or routes with higher speed limits, and then try to find policies that also seek out those features!

22.4.2 *Maximum Margin Planning*

The maximum margin planning approach[15] uses an optimization-based approach to computing the reward function weights $\boldsymbol{w}$ that is very similar to (22.5) but with some additional flexibility. In its most standard form the MMP optimization is:

[15] N. Ratliff, J. A. Bagnell, and M. Zinkevich. "Maximum Margin Planning". In: *Proceedings of the 23rd International Conference on Machine Learning*. 2006, pp. 729–736

$$\hat{w}^* = \arg\min_{\boldsymbol{w}} \; \|\boldsymbol{w}\|_2^2,$$
$$\text{s.t. } w^\top \mu(\pi^*) \geq w^\top \mu(\pi) + 1, \quad \forall \pi \in \{\pi_0, \pi_1, \dots\}.$$

Again this problem computes the reward function vector $\boldsymbol{w}$ such that the expert policy *maximally outperforms* the policies in the set $\{\pi_0, \pi_1, \dots\}$.

However the formulation is also improved in two ways: it adds a slack term to account for potential expert suboptimality and it adds a similarity function that gives more "margin" to policies that are dissimilar to the expert policy. This new formulation is:

$$\begin{aligned} \hat{w}^* = \arg\min_{\boldsymbol{w},v} \ & \|\boldsymbol{w}\|_2^2 + Cv, \\ \text{s.t. } & w^\top \mu(\pi^*) \geq w^\top \mu(\pi) + m(\pi^*, \pi) - v, \quad \forall \pi \in \{\pi_0, \pi_1, \dots\}, \end{aligned} \tag{22.6}$$

where v is a slack variable that can account for expert suboptimality, $C > 0$ is a hyperparameter that is used to penalize the amount of assumed suboptimality, and $m(\pi^*, \pi)$ is a function that quantifies how dissimilar two policies are.

One example of where this formulation is advantageous over the apprenticeship learning formulation (22.5) is when the expert is suboptimal. In this case it is possible that there is no $\boldsymbol{w}$ that makes the expert policy outperform all other policies, such that the optimization (22.5) returns $\boldsymbol{w}_i = 0$ and $t_i = 0$ (which is obviously not the appropriate solution). Alternatively the slack variables in the MMP formulation allow for a reasonable $\boldsymbol{w}$ to be computed.

22.4.3 *Maximum Entropy Inverse Reinforcement Learning*

While the apprenticeship learning approach shows that matching feature counts is a necessary and sufficient condition to ensure a policy performs as well as an expert, it also has some ambiguity (similar to the reward weight ambiguity problem discussed before). This ambiguity is associated with the fact that there could be different policies that lead to the same feature expectations!

This issue can also be thought of in a slightly more intuitive way in terms of distributions over trajectories. Specifically, a policy π induces a distribution over trajectories[16] $\tau = \{(\boldsymbol{x}_0, \pi(\boldsymbol{x}_0)), (\boldsymbol{x}_1, \pi(\boldsymbol{x}_1)), \dots\}$ that is denoted as $p_\pi(\tau)$. The feature expectations can be rewritten in terms of this distribution as:

$$\mu(\pi) = E_\pi\big[f(\tau)\big] = \int p_\pi(\tau) f(\tau) d\tau,$$

where $f(\tau) = \sum_{t=0}^{T-1} \gamma^t \phi(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t))$. Now suppose a policy π was found that matched feature expectations[17] with an expert policy π^* such that:

$$\int p_\pi(\tau) f(\tau) d\tau = \int p_{\pi^*}(\tau) f(\tau) d\tau.$$

Crucially this condition is not sufficient to guarantee that $p_\pi(\tau) = p_{\pi^*}(\tau)$ (which would be ideal). In fact, the distribution $p_\pi(\tau)$ could also have an *arbitrary preference* for some paths that is *unrelated* to the feature matching objective.

The main idea in the maximum entropy inverse RL approach[18] is to not only match the feature expectations, but also remove ambiguity in the path distribution $p_\pi(\tau)$ by trying to make $p_\pi(\tau)$ *as broadly uncommitted as possible*. In other words, find a policy that matches feature expectations but otherwise has no additional path preferences. This concept is known as the maximum entropy principle[19].

[16] This distribution can be visualized as a set of paths generated by simulating the system many times with policy π (i.e. using a Monte Carlo method).

[17] For example by using apprenticeship learning.

[18] B. D. Ziebart et al. "Maximum Entropy Inverse Reinforcement Learning". In: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*. 2008, pp. 1433–1438

[19] A maximum entropy distribution can be thought of as the *least informative distribution* of a class of distribution. This is useful in situations where it is undesirable to encode unintended prior information.

The maximum entropy IRL approach finds a minimally preferential, feature expectation matching distribution by solving the optimization problem:

$$\begin{aligned} p^*(\tau) = \arg\max_p \; & \int -p(\tau)\log p(\tau)d\tau, \\ \text{s.t.} \; & \int p(\tau)f(\tau)d\tau = \int p_{\pi^*}(\tau)f(\tau)d\tau, \\ & \int p(\tau)d\tau = 1, \\ & p(\tau) \geq 0, \quad \forall \tau, \end{aligned} \tag{22.7}$$

where the objective is the mathematical definition of a distribution's entropy, the first constraint requires feature expectation matching, and the remaining constraints ensure that $p(\tau)$ is a valid probability distribution. It turns out that the solution to this problem has the exponential form:

$$p^*(\tau, \boldsymbol{\lambda}) = \frac{1}{Z(\boldsymbol{\lambda})} e^{\boldsymbol{\lambda}^\top f(\tau)}, \quad Z(\boldsymbol{\lambda}) = \int e^{\boldsymbol{\lambda}^\top f(\tau)} d\tau,$$

where $Z(\boldsymbol{\lambda})$ normalizes the distribution, and where $\boldsymbol{\lambda}$ must be chosen such that the feature expectations match:

$$\int p^*(\tau, \boldsymbol{\lambda}) f(\tau) = \int p_{\pi^*}(\tau) f(\tau) d\tau.$$

In other words the maximum entropy IRL approach tries to find a distribution parameterized by $\boldsymbol{\lambda}$ that match features, but also requires that the distribution $p^*(\tau, \boldsymbol{\lambda})$ belong to the exponential family.

To determine the value of $\boldsymbol{\lambda}$ that matches features, it is assumed that the expert also selects trajectories with high reward with exponentially higher probability:

$$p_{\pi^*}(\tau) \propto e^{\boldsymbol{w}^{*\top} f(\tau)},$$

and therefore ideally $\boldsymbol{\lambda} = \boldsymbol{w}^*$. Of course $\boldsymbol{w}^*$ (and more generally $p_{\pi^*}(\tau)$) are not known, and therefore a maximum likelihood estimation approach is used to compute $\boldsymbol{\lambda}$ to best approximate $\boldsymbol{w}^*$ based on the sampled expert demonstrations[20].

In particular, an estimate $\hat{\boldsymbol{w}}^*$ of the reward weights is computed from the expert demonstrations $\Xi = \{\xi_0, \xi_1, \dots\}$ (which each demonstration ξ_i is a trajectory) by solving the maximum likelihood problem:

$$\begin{aligned} \hat{\boldsymbol{w}}^* &= \arg\max_{\boldsymbol{\lambda}} \prod_{\xi_i \in \Xi} p^*(\xi_i, \boldsymbol{\lambda}), \\ &= \arg\max_{\boldsymbol{\lambda}} \sum_{\xi_i \in \Xi} \boldsymbol{\lambda}^\top f(\xi_i) - \log Z(\boldsymbol{\lambda}), \end{aligned}$$

which can be solved using a gradient descent algorithm where the gradient is computed by:

$$\nabla_{\boldsymbol{\lambda}} J(\boldsymbol{\lambda}) = \sum_{\xi_i \in \Xi} f(\xi_i) - E_{\tau \sim p^*(\tau, \boldsymbol{\lambda})}[f(\tau)].$$

[20] By assuming the expert policy is also exponential, the maximum likelihood estimate is theoretically *consistent* (i.e. $\boldsymbol{\lambda} \to \boldsymbol{w}^*$ as the number of demonstrations approaches infinity).

The first term of this gradient is easily computable since the expert demonstrations are known, and the second term can be approximated through Monte Carlo sampling. However, this Monte Carlo sampling estimate is based on sampling trajectories from the distribution $p^*(\tau, \boldsymbol{\lambda})$. This leads to a iterative algorithm:

1. Initialize $\boldsymbol{\lambda}$ and collect the set of expert demonstrations $\Xi = \{\xi_0, \xi_1, \dots\}$.
2. Compute the optimal policy[21] $\pi_{\boldsymbol{\lambda}}$ with respect to the reward function with $\boldsymbol{w} = \boldsymbol{\lambda}$.
3. Using the policy $\pi_{\boldsymbol{\lambda}}$, sample trajectories of the system and compute an approximation of $E_{\tau \sim p^*(\tau, \boldsymbol{\lambda})}[f(\tau)]$.
4. Perform a gradient step on $\boldsymbol{\lambda}$ to improve the maximum likelihood cost.
5. Repeat until convergence.

[21] For example through traditional RL methods.

To summarize, the maximum entropy inverse reinforcement learning approach identifies a distribution over trajectories that matches feature expectations with the expert, but by restricting the distribution to belong to the exponential family ensures that spurious preferences (path preferences not motivated by feature matching) are not introduced. Additionally, this distribution over trajectories is parameterized by a value that is an estimate of the reward function weights.

22.5 Learning From Comparisons and Physical Feedback

Both behavioral cloning and inverse reinforcement learning approaches rely on expert demonstrations of behavior. However in some practical scenarios it may actually be difficult for the expert to provide complete/quality demonstrations. For example it has been shown[22] that when humans are asked to demonstrate good driving behavior in simulation they retroactively think their behavior was too aggressive! As another example, if a robot has a high-dimensional control or state space it could be difficult for the expert to specify the full high-dimensional behavior. Therefore another interesting question in imitation learning is to find a way to learn from alternative data sources besides complete demonstrations.

[22] C. Basu et al. "Do You Want Your Autonomous Car to Drive Like You?" In: *12th ACM/IEEE International Conference on Human-Robot Interaction*. 2017, pp. 417–425

22.5.1 Learning from Comparisons

One alternative approach is to use *pairwise comparisons*[23], where an expert is shown two different behaviors and then asked to rank which behavior is better. Through repeated queries it is possible to converge to an understanding of the underlying reward function. For example, suppose two trajectories τ_A and τ_B are shown to an expert and that trajectory τ_A is preferred. Then assuming that the reward function is:

$$R(\tau) = \boldsymbol{w}^\top f(\tau),$$

[23] D. Sadigh et al. "Active Preference-Based Learning of Reward Functions". In: *Robotics: Science and System*. 2017

where $f(\tau)$ are the collective feature counts (same as in Section 22.4), this comparison can be used to conclude that:

$$\boldsymbol{w}^\top f(\tau_A) > \boldsymbol{w}^\top f(\tau_B).$$

In other words, this comparison has split the space of possible reward weights $\boldsymbol{w}$ in half through the hyperplane:

$$(f(\tau_A) - f(\tau_B))^\top \boldsymbol{w} = 0.$$

By continuously querying the expert with new comparisons[24], the space of possible reward weights $\boldsymbol{w}$ will continue to shrink until a good estimate of $\boldsymbol{w}^*$ can be made. In practice the expert decision may be a little noisy and therefore the hyperplanes don't define hard cutoffs, but rather can be used to "weight" the possible reward vectors $\boldsymbol{w}$.

[24] The types of comparisons shown can be selectively chosen to maximally split the remaining space of potential $\boldsymbol{w}$ in order to minimize the total number of expert queries that are required.

22.5.2 *Learning from Physical Feedback*

Another alternative to learning from complete expert demonstrations is to simply allow the expert to physically interact with the robot to correct for undesirable behavior[25]. In this approach, a physical interaction (i.e. a correction) is assumed to occur when the robot takes actions that result in a lower reward than the expert's action.

[25] A. Bajcsy et al. "Learning Robot Objectives from Physical Human Interaction". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 217–226

For a reward function of the form $R(\boldsymbol{x}, \boldsymbol{u}) = \boldsymbol{w}^\top \phi(\boldsymbol{x}, \boldsymbol{u})$ the robot maintains an estimate of the reward weights $\hat{\boldsymbol{w}}^*$ and the expert is assumed to have act according to a true set of optimal weights $\boldsymbol{w}^*$. Suppose the robot's policy, which is based on the estimated reward function with weights $\hat{\boldsymbol{w}}^*$, yields a trajectory τ_R. Then, if the expert physically interacts with the robot to make a correction the resulting actual trajectory τ_H is assumed to satisfy:

$$\boldsymbol{w}^{*\top} f(\tau_H) \geq \boldsymbol{w}^{*\top} f(\tau_R),$$

which simply states that the reward of the new trajectory is higher. This insight is then leveraged in a maximum a posteriori approach for updating the estimate $\hat{\boldsymbol{w}}^*$ after each interaction. Specifically, this update takes the form:

$$\hat{\boldsymbol{w}}^* \leftarrow \hat{\boldsymbol{w}}^* + \beta(\boldsymbol{f}(\tau_H) - \boldsymbol{f}(\tau_R)),$$

where $\beta > 0$ is a scalar step size. The robot then uses the new estimate to change its policy, and the process iterates. Note that this idea yields an approach that is similar to the concept of matching feature expectations from inverse reinforcement learning, except that the approach is iterative rather than requiring a batch of complete expert demonstrations.

22.6 *Interaction-aware Control and Intent Inference*

Yet another interesting problem in robot autonomy arises when robots and humans are interacting to accomplish shared or individual goals. Many classical

examples of this problem arise in autonomous driving settings, when human-driven vehicles interact with autonomous vehicles in settings such as highway merging or at intersections. While the imitation learning problems from the previous sections are focused on understanding the expert's behavior for the purpose of *imitating* the behavior, in this setting the human's behavior needs to be understood in order to ensure safe interactions. However there is an additional component to understanding interactions: *the robot's behavior can influence the human's behavior*[26].

[26] It is particularly important in interaction-aware robot control to understand the effects of the robot's actions on the human's behavior. Otherwise the human's could simply be modeled as dynamic obstacles!

22.6.1 Interaction-aware Control with Known Human Model

One common approach is to model the interaction between humans and robots as a dynamical system that has a combined state $\boldsymbol{x}$, where the robot controls are denoted $\boldsymbol{u}_R$ and the human decisions or inputs are denoted as $\boldsymbol{u}_H$. The transition model is therefore defined as:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{R,t-1}, \boldsymbol{u}_{H,t-1}).$$

In other words the interaction dynamics evolve according to the actions taken by both the robot and the human. In this interaction the robot's reward function is denoted as $R_R(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H)$ and the human's reward function is denoted as $R_H(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H)$, which are both functions of the combined state and both agent's actions[27].

[27] While R_R and R_H do not have to be the same, choosing $R_R = R_H$ may be desirable for the robot to achieve human-like behavior.

Under the assumption that both the robot and the human act optimally[28] with respect to their cost functions:

[28] While not necessarily true, this assumption is important to make the resulting problem formulation tractable to solve in practice.

$$\boldsymbol{u}_R^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}_R} \; R_R(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H^*(\boldsymbol{x})),$$
$$\boldsymbol{u}_H^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}_H} \; R_H(\boldsymbol{x}, \boldsymbol{u}_R^*(\boldsymbol{x}), \boldsymbol{u}_H).$$

Additionally, assuming both reward functions R_R and R_H are known[29], computing $\boldsymbol{u}_R^*$ is still extremely challenging due to the two-player game dynamics of the decision making process. However this problem can be made more tractable by modeling it as a *Stackelberg game*, which restricts the two-player game dynamics to a leader-follower structure. Under this assumption it is assumed that the robot is the "leader" and that as the follower the human acts according to:

[29] The reward function R_H could be approximated using inverse reinforcement learning techniques.

$$\boldsymbol{u}_H^*(\boldsymbol{x}, \boldsymbol{u}_R) = \arg\max_{\boldsymbol{u}_H} \; R_H(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H). \tag{22.8}$$

In other words the human is assumed to see the action taken by the robot *before* deciding on their own action. The robot policy can therefore be computed by solving:

$$\boldsymbol{u}_R^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}_R} \; R_R(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H^*(\boldsymbol{x}, \boldsymbol{u}_R)), \tag{22.9}$$

which can be solved using a gradient descent approach. For the gradient descent approach the gradient of:

$$J(\boldsymbol{x}, \boldsymbol{u}_R) = R_R(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H^*(\boldsymbol{x}, \boldsymbol{u}_R)),$$

can be computed using the chain rule as:

$$\frac{\partial J}{\partial \boldsymbol{u}_R} = \frac{\partial R_R}{\partial \boldsymbol{u}_R} + \frac{\partial R_R}{\partial \boldsymbol{u}_H^*} \frac{\partial \boldsymbol{u}_H^*}{\partial \boldsymbol{u}_R}.$$

Since the reward function R_R is known the terms $\partial R_R / \partial \boldsymbol{u}_R$ and $\partial R_R / \partial \boldsymbol{u}_H^*$ can be easily determined. In order to compute the term $\partial \boldsymbol{u}_H^* / \partial \boldsymbol{u}_R$, which represents how much the robot's actions impact the human's actions, an additional step is required. First, assuming the human acts optimally according to (22.8) the necessary optimality condition is:

$$g(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H^*) = 0, \quad g = \frac{\partial R_H}{\partial \boldsymbol{u}_H},$$

which for the fixed values of $\boldsymbol{x}$ and $\boldsymbol{u}_R$ specifies $\boldsymbol{u}_H^*$. Then, by implicitly differentiating this condition with respect to the robot action $\boldsymbol{u}_R$:

$$\frac{\partial g}{\partial \boldsymbol{u}_R} + \frac{\partial g}{\partial \boldsymbol{u}_H^*} \frac{\partial \boldsymbol{u}_H^*}{\partial \boldsymbol{u}_R} = 0,$$

which can be used to solve for:

$$\frac{\partial \boldsymbol{u}_H^*}{\partial \boldsymbol{u}_R}(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H^*) = -\left(\frac{\partial g}{\partial \boldsymbol{u}_H^*}\right)^{-1} \frac{\partial g}{\partial \boldsymbol{u}_R}.$$

Notice that every term in this expression can be computed[30] and therefore it can be substituted into the gradient calculation:

$$\frac{\partial J}{\partial \boldsymbol{u}_R} = \frac{\partial R_R}{\partial \boldsymbol{u}_R} - \frac{\partial R_R}{\partial \boldsymbol{u}_H^*} \left(\frac{\partial g}{\partial \boldsymbol{u}_H^*}\right)^{-1} \frac{\partial g}{\partial \boldsymbol{u}_R},$$

which can then be computed as long as it is possible to compute $\boldsymbol{u}_H^*(\boldsymbol{x}, \boldsymbol{u}_R)$.

To summarize, one approach to interaction-aware control is to model the interaction as a Stackelberg game, where it is assumed that both the human and the robot act optimally with respect to some reward functions. This formulation of the problem enables the robot to choose actions based on an implicit understanding of how the human will react.

[30] Assuming the human's reward function is known.

22.6.2 *Intent Inference*

One disadvantage to the approach for interaction-aware control from the previous section is that it assumes the human acts optimally with respect to a *known* reward function. While a reward function could be learned through inverse reinforcement learning, this is not practical for real-world settings where different humans behave differently. Returning to the example of interaction between human drivers and autonomous vehicles, the human could exhibit drastically different behavior depending on whether they have an aggressive or passive driving style. In these settings the problem of *intent inference* focuses on identifying underlying behavioral characteristics that can lead to more accurate behavioral models[31].

[31] This problem can be formulated as a partially observable Markov decision process (POMDP) since the underlying behavioral characteristic is not directly observable, yet influences the system's behavior.

One approach to intent inference[32] is to model the underlying behavioral differences through a set of unknown parameters $\boldsymbol{\theta}$ which need to be inferred by observing the human's behavior. Mathematically this is expressed by defining the human's reward function $R_H(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H, \boldsymbol{\theta})$ to be a function of $\boldsymbol{\theta}$, and assuming the human chooses actions according to:

$$p(\boldsymbol{u}_H \mid \boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{\theta}) \propto e^{R_H(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H, \boldsymbol{\theta})}.$$

In other words this model assumes the human is exponentially more likely to pick optimal actions[33], but that they may pick suboptimal actions as well.

The objective of intent inference is therefore to estimate the parameters $\boldsymbol{\theta}$, which can be accomplished through Bayesian inference methods. In the Bayesian approach a *probability distribution* over parameters $\boldsymbol{\theta}$ is updated based on observations. Specifically the belief distribution is denoted as $b(\boldsymbol{\theta})$, and given an observation of the human's actions $\boldsymbol{u}_H$ the belief distribution is updated as:

$$b_{t+1}(\boldsymbol{\theta}) = \frac{1}{\eta} p(\boldsymbol{u}_{H,t} \mid \boldsymbol{x}_t, \boldsymbol{u}_{R,t}, \boldsymbol{\theta}) b_t(\boldsymbol{\theta}),$$

where η is a normalizing constant. This Bayesian update is simply taking the prior belief over $\boldsymbol{\theta}$ and updating the distribution based on the likelihood of observing human action $\boldsymbol{u}_H$ under that prior. Note that this concept is quite similar to the concepts of inverse reinforcement learning: a set of parameters that describe the human's (experts) behavior are continually updated when new observations of their actions are gathered.

While the robot could sit around and *passively* observe the human act to collect samples for the Bayesian updates, it is often more efficient for the robot to *probe* the human to take interesting actions that are more useful for revealing the intent parameters $\boldsymbol{\theta}$. This can be accomplished by choosing the robot's reward function to be:

$$R_R(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H, \boldsymbol{\theta}) = I(b(\boldsymbol{\theta}), \boldsymbol{u}_R) + \lambda R_{\text{goal}}(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H, \boldsymbol{\theta})$$

where $\lambda > 0$ is a tuning parameter and $I(b(\boldsymbol{\theta}), \boldsymbol{u}_R)$ denotes a function that quantifies the amount of information gained with respect to the belief distribution from taking action $\boldsymbol{u}_R$. In other words the robot's reward is a tradeoff between exploiting the current knowledge of $\boldsymbol{\theta}$ to accomplish the objective and taking exploratory actions to improve the intent inference. With this robot reward function the robot's actions are chosen to maximize the expected reward:

$$\boldsymbol{u}_R^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}_R} \; E_{\boldsymbol{\theta}}[R_R(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H, \boldsymbol{\theta})].$$

To summarize, this robot policy will try to simultaneously accomplish the robot's objective and gather more information to improve the inference of the human's intent (modeled through the parameters $\boldsymbol{\theta}$). In a highway lane changing scenario this type of policy might lead the robot to nudge into the other lane to see if the other car will slow down (passive driving behavior) or try to

[32] D. Sadigh et al. "Planning for cars that coordinate with people: leveraging effects on human actions for planning and active information gathering over human internal state". In: *Autonomous Robots* 42.7 (2018), pp. 1405–1426

[33] This assumption was also used in the Maximum Entropy IRL approach.

block the lane change (aggressive driving behavior). Once the robot has a strong enough belief about the human's behavior it may choose to either complete the lane change or slow down to merge behind the human driver.

Part V

Robot Software

23
Robot System Architectures

A robotic system is fundamentally just a collection of sensors and actuators that can interact with the environment to accomplish a set of tasks. While this definition may seem simple, the systems required to implement this definition tend to be extremely complex due to the infinite variability and uncertainty of real-world environments and the diversity among sensors and actuators. Therefore, careful and practical design of robotic systems is crucial for managing complexity, and as a byproduct enabling robust and successful robotic operations. This chapter will introduce some of the fundamental concepts, paradigms, and tools in the design of robot system architectures to enable full robot autonomy while also managing system complexity[1].

[1] D. Kortenkamp, R. Simmons, and D. Brugali. "Robotic Systems Architectures and Programming". In: *Springer Handbook of Robotics*. Springer, 2008, pp. 283–302

Robot System Architectures

The primary objective of a robotic system is to accomplish a specific set of tasks, but there are often many peripheral tasks that must also be handled to ensure the robot operates in a safe and robust way. For example a robot's goal may be to pick up objects and place them in certain locations, but in order to accomplish this task the robot should also be aware of obstacles (static or dynamic) in its environment, should be robust to sensor failures or sensor noise, and more.

Definition 23.0.1 (Robot Goal)**.** *Complete desired tasks while monitoring and reacting to unexpected situations. Handle inputs and outputs (control/perception) from actuators and sensors in real-time*[2] *and under uncertainty.*

[2] Real-time requirements are crucial, some situations require near instantaneous reactions (e.g. less than 1 ms reaction time).

The design of the robot's system architecture is important for enabling the robot to achieve its goal without requiring extremely complex software systems for implementation. In general, the *system architecture* is defined by two major parts: the *structure* and the *style*. The structure defines the way in which the system is broken down into components, as well as how the components interact with each other[3]. Alternatively the style of the architecture refers to the computational concepts that define the implementation of the design.

[3] The structure could be represented visually as a diagram of boxes (components) that are connected by arrows (interactions).

Generally speaking there is no specific architecture that is optimal for every robotic system, but there are some paradigms that have been proven to be use-

ful, which will be introduced in more detail in the following sections. In fact, any given system architecture may consist of multiple types of structures or styles! For a given robot, the specific choice of architecture should aim to reduce complexity[4] while not being overly restrictive and thus limit performance.

[4] For example subsystem segmentation can be useful for reusability as well as validation and unit-testing.

23.1 Architecture Structures

The architecture's *structure* defines how the system is subdivided into subsystems and how the subsystems interact. Some form of hierarchical structure is commonly used for this decomposition, which reduces complexity through abstraction (e.g. tasks at one level are of the hierarchy are composed of a group of tasks from lower-levels of the hierarchy).

23.1.1 Sense-Plan-Act Architecture

This architecture is one of the first developed, and consists of three main subsystems: sensing, planning, and execution. These components were organized in a sequential fashion, with sensor data being passed to the planner, who then passes information to the controller, who sends actuator commands. However this approach has significant drawbacks. First, the planning component was a computational bottleneck that held up the controller subsystem. Second, since the controller did not have direct access to sensor data the overall system was not very *reactive*.

23.1.2 Subsumption Architecture

An alternative to the sense-plan-act architecture that emerged not long after is the *subsumption architecture*[5]. This architecture decomposes the overall desired robot behavior into sub-behaviors in a bottom-up fashion. In this hierarchical structure the higher-level behaviors *subsume* the lower-level behaviors. In other words, the high-level behaviors can outsource smaller scale tasks to be handled by the low-level behaviors. From an implementation standpoint this architecture can be thought of as layers of finite state machines[6] that all connect sensors to actuators, and where multiple behaviors are evaluated in parallel. An arbitration mechanism is also included to choose which of the behaviors is currently activated. For example an explore behavior may sit on top of (subsume) a collision avoidance behavior, and the arbitration mechanism would decide when the exploration behavior should be overridden by the collision avoidance behavior.

[5] R. Brooks. "A robust layered control system for a mobile robot". In: *IEEE Journal on Robotics and Automation* 2.1 (1986), pp. 14–23

[6] Each finite state machine was often referred to as a *behavior*.

While this architecture is much more reactive than the sense-plan-act architecture, there are also disadvantages. The primary disadvantage of this approach is that there is no good way to do long-term planning or behavior optimization. This can make it challenging to design the system to accomplish long-term objectives.

23.1.3 Three-tiered Architecture

The *three-tiered architecture* is one of the most commonly used architectural designs. This architecture contains a planning, an executive, and a behavioral control level that are hierarchically linked.

1. *Planning*: this layer is at the highest-level, and focuses on task-planning for long-term goals.

2. *Executive*: the executive layer is the middle layer connecting the planner and the behavioral control layers. The executive specifies priorities for the behavioral layer to accomplish a specific task. While the task may come directly from the planning layer, the executive can also split higher-level tasks into sub-tasks.

3. *Behavioral control*: at the lowest-level. the behavioral control layer handles the implementation of low-level behaviors and is the interface to the robot's actuators and sensors.

The primary advantage of this architecture is that it combines benefits of the behavioral-based subsumption architecture (i.e. reactive planning) with better long-term planning capabilities (i.e. resulting from the planning level). Each of these levels will now be discussed in slightly further detail, however in practice the division among these levels is often quite blurred!

Behavioral Control Level: The components at the behavioral control level typically focus on small, localized behaviors or skills and directly interface with the robot's sensors and actuators[7]. These behaviors are typically *situated*, meaning that they only make sense with respect to a specific situation that the robot may be in. Importantly, the behavioral control components should have an awareness of the current situation (i.e. they should be able to identify if the current situation is appropriate for a specific behavior), but they are not responsible for knowing how to *change* the situation (this is left to the executive level).

[7] This layer includes algorithms from classical control theory: PID control, Kalman filtering, etc.

The tight interaction between the sensors and actuators in the behavioral control level enables a high level of reactivity in this architecture. However, high reactivity also requires that the behavioral control level not incorporate algorithms with high computational complexity. In general, the algorithms at this level should be able to operate *at least* several times per second.

Executive Level: The components of the executive level are responsible for translating high-level plans into low-level behaviors, orchestrating when low-level behaviors are executed, as well as monitoring for and handling exceptions. This component is typically implemented as a hierarchical finite state machine, but might also incorporate motion planning and decision making algorithms to break a high-level task into a sequence of smaller tasks. To orchestrate the sequence and timing for behaviors to be implemented, the executive considers

temporal constraints on behaviors (e.g. whether two actions can be executed concurrently).

Planning Level: Finally, the planning level focuses on high-level decision making and planning for long-term behavior. This forward-thinking component is crucial to optimize the long-term behavior of the robot. However, the implementation of the decisions from the planner are deferred to the executive layer. In practice it might also be useful to have multiple planning levels, for example to split up mission level planning (very abstract planning) with shorter horizon planning[8].

[8] This split might be useful for computational performance reasons.

Example 23.1.1 (Office Mail Delivery Robot)**.** To further explore the components of the three-tiered robot system architecture, consider a robot whose primary task is to deliver mail within an office setting. In general, tasks that might be required of this robot include: the ability to move through hallways and rooms, avoid humans and other obstacles, open and close doors, announce a delivery, find a particular room, recharge its batteries, etc.

If a three-tiered architecture is used, the planner level would be in charge of high-level decision making tasks. For example the planner might specify the delivery order for each piece of mail to optimize the overall efficiency (i.e. by considering the relative locations of each delivery). The planner would also choose when to schedule time for recharging.

Given a task from the planner such as "Deliver package to Rm 009", the executive level would then coordinate how to accomplish the task. This might include sub-tasks such as move to the end of the hallway, open the door, enter Rm 009, announce delivery, and then wait and monitor to see if the package is retrieved. If the package is never retrieved within a specified amount of time the executive level could also choose to then carry on with the next set of tasks and send a message to the planner that the task was not completed.

Finally, the behavioral control layer would execute the tasks as specified by the executive level. This might include controlling the robot's wheels to move across the hallway, avoiding obstacles along the way. Or it could involve using a manipulator to open a door. If the current task specified by the executive was to open a door and the door was locked, the behavioral control level should eventually recognize failure and report back to the executive level.

23.2 *Architecture Styles*

In addition to choosing the robot system architecture, another very important task is to choose the architecture's *style*. An architecture's style refers to the computational structure that defines communication between components within the architecture. For example in the three-tiered architecture the style would define the method for communicating among the planning, executive, and behavioral control levels, or even between components of each individual level. The implementation of the connection style is typically referred to as

middleware, and two of the most common architecture styles are referred to as *client-server* and *publish-subscribe*.

23.2.1 Client-Server

Middleware based on the client-server style consists of message requests from clients that the server responds to (i.e. there is a request-response message pairing). This type of connection style can also be thought of as being *on-demand* messaging. One of the disadvantages of such a messaging style is that the client typically waits for the response from the server before continuing, leading to potential deadlocks (e.g. if the server crashes).

23.2.2 Publish-Subscribe

Middleware based on the publish-subscribe style uses asynchronous message broadcasting from publishers, which can then be subscribed to by other components of the system as needed. One disadvantage of this approach is that the interfaces are less well-defined (interactions are only one-way), but the main advantage is in reliability since deadlocks cannot occur (e.g. the system is robust to missing messages or messages arriving out of order). The middleware ROS (Robot Operating System) is a very popular publish-subscribe middleware used within the robotics community today.

24
The Robot Operating System

Introduction to the Robot Operating System (ROS)

This chapter introduces the fundamentals of the Robot Operating System (ROS)[1,2], a popular framework for creating robot software. Unlike what its name appears to suggest, ROS is not an operating system (OS). Rather, ROS is a "middleware" that encompasses tools, libraries and conventions to operate robots in a simplified and consistent manner across a wide variety of robotic platforms. ROS is a critical tool in the field of robotics today, and is widely used in both academia and industry.

This chapter begins by introducing specific challenges in robot programming that motivates the need for a middleware such as ROS. Afterwards, a brief history of ROS will be presented to shed some light on its development and motivations for its important features. Next, the fundamental operating structure of ROS will be discussed in further detail to provide insights into how ROS is operated on real robotic platforms. Lastly, specific features and tools of the ROS environment that greatly simplify robot software development will be presented.

[1] L. Joseph. *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*. Apress, 2018

[2] M. Quigley, B. Gerkey, and W. D. Smart. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. O'Reilly Media, 2015

24.1 Challenges in Robot Programming

Robot programming is a subset of computer programming, but it differs greatly from more classical software programming applications. One of the defining characteristics of robot programming is the need to manage many different individual hardware components that must operate in harmony (e.g. sensors and actuators on board the robot). In other words, robot software needs to not only run the "brain" of the robot to make decisions, but also to handle multiple input and output devices at the same time. Therefore, the following features are needed for robot programming:

- *Multitasking:* Given a number of sensors and actuators on a robot, robot software needs to multitask and work with different input/output devices in different threads at the same time. Each thread needs to be able to communi-

cate with other threads to exchange data.

- *Low level device control:* Robot software needs to be compatible with a wide variety of input and output devices: GPIO (general purpose input/output) pins, USB, SPI among others. C, C++ and Python all work well with low-level devices, so robot software needs to support either of these languages, if not all.
- *High level Object Oriented Programming (OOP):* In OOP, codes are encapsulated, inherited, and reused as multiple instances. Ability to reuse codes and develop programs in independent modules makes it easy to maintain code for complex robots.
- *Availability of 3rd party libraries and community support:* Ample third-party libraries and community support not only expedite software development, but also facilitate efficient software implementation.

24.2 *Brief History of ROS*

Until the advent of ROS, it was impossible for various robotics developers to collaborate or share work among different teams, projects or platforms. In 2007, early versions of ROS started to be conceived with the Stanford AI Robot (STAIR) project, which had the following vision:

- The new robot development environment should be free and open-source for everyone, and need to remain so to encourage collaboration of community members.
- The new platform should make core components of robotics – from its hardware to library packages – readily available for anyone who intends to launch a robotics project.
- The new software development platform should integrate seamlessly with existing frameworks (OpenCV for computer vision, SLAM for localization and mapping, Gazebo for simulation, etc).

Development of ROS started to gain traction when Scott Hassan, a software architect and entrepreneur, and his startup Willow Garage took over the project later that year to develop standardized robotics development platform. While mostly self-funded by Scott Hassan himself, ROS really satiated the dire needs for a standardized robot software development environment at the time. In 2009, ROS 0.4 was released, and a working ROS robot with a mobile manipulation platform called PR2 was developed. Eleven PR2 platforms were awarded to eleven universities across the country for further collaboration on ROS development, and in 2010 ROS 1.0 was released. Many of the original features from ROS 1.0 are still in use. In 2012, the Open Source Robotics Foundation (OSRF) started to supervise the future of ROS by supporting development, distribution,

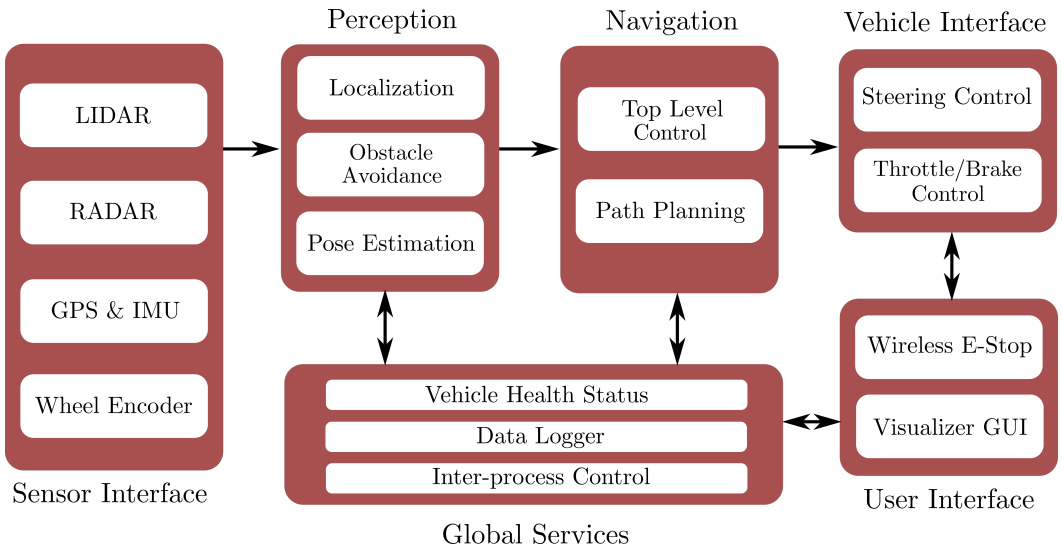


Figure 24.1: Modular software architecture designed to handle complexity of robot programming

and adoption of open software and hardware for use in robotics research, education, and product development. In 2014 the first long-term support (LTS) release, ROS Indigo Igloo, became available. Today, ROS has been around for 12 years, and the platform has become what is closest to the "industry standard" in robotics.

24.2.1 *Characteristics of ROS*

Building off of the initial needs first conceived by the STAIR project and the unique challenges persistent in robot programming, the ROS framework provides the following important capabilities:

- *Modularity:* ROS handles complexity of a robot through modularity: Each robot component that performs separate functions can be developed independently in units called *nodes* (Figure 24.1). Each node can share data with other nodes, and acts as the basic building blocks of ROS. Different functional capabilities on a robot can be developed in units called packages. Each package may contain a number of nodes that are defined from source code, configuration files, and data files. These packages can be distributed and installed on other computers.

- *Message passing:* ROS provides a message passing interface that allows nodes (i.e. programs) to communicate with each other. For example, one node might detect edges in a camera image, then send this information to an object recognition node, which in turn can send information about detected obstacles to a navigation module.

- *Built-in algorithms:* A lot of popular robotics algorithms are already built-in and available as off-the-shelf packages: PID[3], SLAM[4], and path planners such

[3] http://wiki.ros.org/pid
[4] http://wiki.ros.org/gmapping

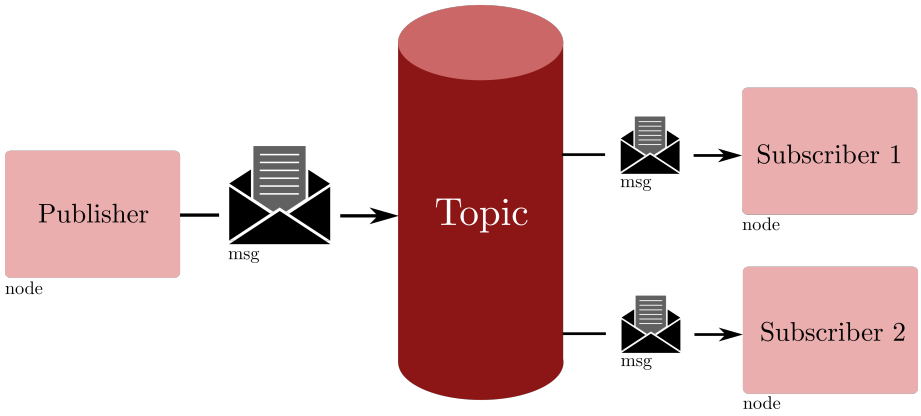


Figure 24.2: The ROS publish/-subscribe (pub/sub) model.

as A* and Dijkstra[5] are just a few examples. These built-in algorithms can significantly reduce time needed to prototype a robot.

- *Third-party libraries and community support:* The ROS framework is developed with pre-existing third-party libraries in mind, and most popular libraries such as OpenCV for computer vision[6] and PCL[7] integrate simply with a couple lines of code. In addition, ROS is supported by active developers all over the world to answer questions (ROS Answers[8] or to discuss various topics and public ROS-related news[9].

[5] http://wiki.ros.org/global_planner

[6] https://opencv.org

[7] http://pointclouds.org

[8] https://answers.ros.org/questions/

[9] https://discourse.ros.org

24.3 Robot Programming with ROS

Before jumping into using the functions and tools that ROS provides it is critical to understand a little more about how ROS operates. In particular, it is important to know that ROS uses a publish/subscribe (pub/sub) structure for communicating between different components or modules. This pub/sub structure (graphically shown in Figure 24.2) allows messages to be passed in between components or modules through a shared virtual "chat room". To support this structure there are four primary components of ROS:

1. Nodes: the universal name for the individual components or modules that need to send or receive information,
2. Messages: the object for holding information that needs communicated between nodes,
3. Topics: the virtual "chat rooms" where messages are shared,
4. Master: the "conductor" that organizes the nodes and topics.

24.3.1 Nodes

Definition 24.3.1 (Node). *A node*[10] *is a process that performs computation. Nodes*

[10] http://wiki.ros.org/Nodes

are combined together to communicate with one another using streaming topics, RPC services, and the Parameter Server.

Nodes are the basic building block of ROS that enables object-oriented robot software development. Each robot component is developed as an individual encapsulated unit of *nodes*, which are later reused and inherited, and a typical robot control system will be comprised of many nodes. The use of independent nodes, and their ability to be reused and inherited, greatly simplifies the complexity of the overall software stack.

For example, suppose a robot is equipped with a camera and you want to find an object in the environment and drive to it. Examples of nodes that might be developed for this task are: a `camera` node that takes the image and pre-processes it, an `edge_detection` node that takes the pre-processed image data and runs an edge detection algorithm, a `path_planning` node that plans a path between two points, and so on.

At the individual level, nodes are responsible for **pub**lishing or **sub**scribing to certain pieces of information that are shared among all other nodes. In ROS, the pieces of information are called “messages" and they are shared in virtual chat rooms called “topics".

24.3.2 *Messages*

Definition 24.3.2 (Messages)**.** *Nodes communicate with each other by publishing simple data structures to topics. These data structures are called messages*[11].

[11] `http://wiki.ros.org/Messages`

A message is defined by field types and field names. The field type defines the type of information the message stores and the name is how the nodes access the information. For example, suppose a node wants to publish two integers x and y, a message definition might look like:

```
int32 x
int32 y
```

where `int32` is the field type and `x/y` is the field name. While `int32` is a primitive field type, more complex field types can also be defined for specific applications. For example, suppose a sensor packet node publishes sensor data as an array of a user-defined `SensorData` object. This message, called `SensorPacket`, could have the following fields:

```
time            stamp
SensorData[]    sensors
uint32          length
```

In this case `SensorData` is a user-defined field type and the empty bracket `[]` is appended to indicate that field is an *array* of `SensorType` objects.

More generally, field types can be either the standard primitive types (integer, floating point, boolean, etc.), arrays of primitive types, or other user-defined types. Messages can also include arbitrarily nested structures and arrays as well.

Primitive message types available in ROS are listed below in Table 24.1. The first column contains the message type, the second column contains the serialization type of the data in the message and the third column contains the numeric type of the message in Python.

Primitive Type	Serialization	Python
bool (1)	unsigned 8-bit int	bool
int8	signed 8-bit int	int
uint8	unsigned 8-bit int	int (3)
int16	signed 16-bit int	int
uint16	unsigned 16-bit int	int
int32	signed 32-bit int	int
uint32	unsigned 32-bit int	int
int64	signed 64-bit int	long
uint64	unsigned 64-bit int	long
float32	32-bit IEEE float	float
float64	64-bit IEEE float	float
string	ascii string (4)	str
time	secs/nsecs unsigned 32-bit ints	rospy.Time

Table 24.1: Built-in ROS Messages

24.3.3 Topics

Definition 24.3.3 (Topics)**.** *Topics*[12] *are named units over which nodes exchange messages.*

[12] http://wiki.ros.org/Topics

A given topic will have a specific message type associated with it, and any node that either publishes or subscribes to the topic must be equipped to handle that type of message. The command `rostopic type <topic>` can be used to see what kind of message is associated with a particular topic. Any number of nodes can publish or subscribe to a given topic.

Fundamentally, topics are for unidirectional, streaming communication. This is perhaps not well suited for all types of communication, such as communication that demands a response (i.e. a service routine).

The `rostopic` command line tool can be used in several ways to monitor active topics/messages. Three of the most common rostopic commands are:

- `rostopic list`: lists all active topics
- `rostopic echo < topic >`: prints messages received on topic
- `rostopic hz < topic >`: measures topic publishing rate

The last command is particularly useful in debugging responsiveness of an application.

24.3.4 Master

Definition 24.3.4 (Master)**.** *The master is a process that can run on any piece of hardware to track publishers and subscribers to topics as well as services in the ROS system.*

Master is responsible for assigning network addresses and enabling individual ROS nodes to locate one another, even if they are running on different computers. Once these nodes have located each other, the communication will be peer-to-peer, i.e., the master will not send nor receive the messages.

In any given ROS system, there is exactly one master running at any time. A unique feature of the master is that master does not need to exist within the robot's hardware as long as a network connection exists. The master can be facilitated remotely, on a much larger and more powerful computer.

24.4 Writing a Simple Publisher Node and Subscriber Node

24.4.1 Publisher Node

A simple publisher node that publishes String messages ten times per second can be implemented in Python via the following code[13]:

[13] http://wiki.ros.org/rospy_tutorials/Tutorials/WritingPublisherSubscriber

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def talker():
    rospy.init_node('talker', anonymous=True)
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rate = rospy.get_param('~rate',1)
    ros_rate = rospy.Rate(rate)

    rospy.loginfo("Starting ROS node talker...")

    while not rospy.is_shutdown():
        msg= "Greetings humans!"

        pub.publish(msg)
        ros_rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

The first line:

```
#!/usr/bin/env python
```

will be included in every Python ROS Node at the top of the file. This line makes sure your script is executed as a Python script.

Next are the statements for importing specific Python libraries:

```
import rospy
from std_msgs.msg import String
```

Note that the library `rospy` must be imported when writing a ROS Node. The `std_msgs.msg` import enables the use of the `std_msgs/String` message type (a simple string container) for publishing string messages.

Next is the definition of the ROS publisher node:

```
rospy.init_node('talker', anonymous=True)
pub = rospy.Publisher('chatter', String, queue_size=10)
```

which creates a node called "talker" and defines the talker's interface to the rest of ROS. In particular:

- `pub = rospy.Publisher("chatter", String, queue_size=10)` declares that the node is publishing to the "chatter" topic using the String message type. String here is actually the ROS datatype (`std_msgs.msg.String`), and not Python's String datatype. The `queue_size` argument limits the amount of queued messages that are allowed, for situations where a subscriber is not receiving the published messages fast enough.

- `rospy.init_node(NAME, ...)` tells `rospy` the name of the node. Until `rospy` has this information, it cannot start communicating with the ROS Master. In this case, your node will take on the name talker. NOTE: the name must be a base name (i.e. it cannot contain any slashes "/").

- `anonymous=True` is a flag that tells `rospy` to generate a unique name for the node, since ROS requires that each node have a unique name. If a node with the same name comes up, it bumps the previous one so that malfunctioning nodes can easily be kicked off the network. This makes it easy to run multiple talker.py nodes.

- `anonymous = True` is another flag that ensures that the node has a unique name by adding random numbers to the end of NAME.

The next lines of code:

```
rate = rospy.get_param('~rate',1)
ros_rate = rospy.Rate(rate)
```

defines a ROS rate that can be used to time how often the node publishes. In particular, `rospy.get_param(param_name, default_value)` reads a private ROS parameter (indicated by '~') called `rate`. This rate value is then used to create a `Rate` object `ros_rate` in the second line. The `Rate` object's `sleep()` method offers a convenient way for looping at the desired frequency. For example, if `rate` is 10, ROS should go through the loop 10 times per second (as long as the processing time does not exceed 1/10th of a second!).

The line:

```
rospy.loginfo("Starting ROS node talker...")
```

performs three functions: it causes messages to get printed to screen, to be written to the Node's log file, and to be written to `rosout`. `rosout` is a handy tool for debugging that makes it possible to pull up messages using `rqt_console` instead of having to find the console window with your Node's output.

The loop:

```
while not rospy.is_shutdown():
    msg = "Greetings humans!"
    pub.publish(msg)
    ros_rate.sleep()
```

is a fairly standard rospy construct for first checking the `rospy.is_shutdown()` flag and then doing work. The `is_shutdown()` check is used to see if the program should exit (e.g. if there is a Ctrl-C interrupt). In this particular example, the "work" that is then performed inside of the loop is a call to `pub.publish(msg)`, which publishes a string to the "chatter" topic. The loop also calls `ros_rate.sleep()` so that it sleeps just long enough to maintain the desired rate through the loop.

24.4.2 Subscriber Node

A subscriber node called `listener` can now be created to subscribe to the published "chatter" topic:

```
!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(msg):
    rospy.loginfo("Received: %s", msg.data)

def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
```

```
    rospy.spin()

if __name__ == '__main__':
    listener()
```

The code for `listener.py` is similar to `talker.py`, except that a new callback-based mechanism for subscribing to messages is introduced.

First, the lines:

```
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
```

declare that the node subscribes to the "chatter" topic, which is of type `std_msgs.msgs.String`. When new messages are received, the function `callback` is invoked with the message as the first argument.

The line:

```
    rospy.spin()
```

then simply keeps the node from exiting until the node has been shutdown.

24.4.3 *Compiling and Running*

Once both the `talker.py` and `listener.py` nodes are ready, the `catkin` build system can be used to compile the code, and then both nodes can be run. Specifically, this is accomplished by running the following commands:

```
$ cd ~/catkin_ws
$ catkin_make
$ python talker.py
$ python listener.py
```

24.5 *Other Features in ROS Development Environment*

24.5.1 *Launch files*

As a robot project grows in scale, the number of nodes and configuration files grow very quickly. In practice, it could be very cumbersome to manually start up each individual node. A *launch file* provides a convenient way to start up multiple nodes and a master, as well as set up other configurations, all at the same time.

Definition 24.5.1. *Launch files are .launch files with a specific XML format that can be placed anywhere within a package directory to initialize multiple nodes, configuration files, and a master.*

While `.launch` files can be placed anywhere within a package directory, it is standard practice to create a `launch` folder inside the workspace directory to organize launch files. Launch files must start and end with a pair of launch tags: `<launch> ... </launch>`. To start a node using a launch file the following syntax should be used within the launch file:

```
<node name="..." pkg="..." type="..."/>
```

In this line, **pkg** points to the package associated with the node that is to be launched, **type** refers to the name of the node executable file, and the name of the node can be overwritten with the **name** argument (this will take priority over the name that is given to the node in the code). For example,

```
<node name="bar1" pkg="foo_pkg" type="bar" />
```

launches the `bar` node from the `foo_pkg` package with a new name, `bar1`. Alternatively,

```
<node name="listener1" pkg="rospy_tutorials" type="listener.py"
     args="--test" respawn="true" />
```

launches the `listener1` node using the `listener.py` executable from the `rospy_tutorials` package with the command-line argument `-test`. Additionally, if the node dies it will automatically be respawned.

There are other attributes that can be set when starting a node. While only `args` and `respawn` were introduced in this section, `http://wiki.ros.org/roslaunch/XML/node` is a great resource for additional parameters that can be used in the `<node>` tag.

24.5.2 Catkin Workspace

`catkin`[14] is a build system that compiles ROS packages. While `catkin`'s workflow[15] is very similar to CMake's, `catkin` adds support for automatic 'find package' infrastructure, for building multiple, dependent projects at the same time, and also supports both C and Python.

[14] `catkin` refers to the tail-shaped flower cluster on willow trees – a reference to Willow Garage where, `catkin` was created.

[15] `http://wiki.ros.org/catkin/conceptual_overview`

When developing with ROS, `catkin` should be run whenever a new project is started, or if there are any additions to packages. This is accomplished by creating a directory called `catkin_ws` and then running the compile command `catkin_make` in that directory:

```
mkdir -p ~/catkin_ws/src  # builds the catkin_ws in the home dir
cd ~/catkin_ws            # change current directory to catkin_ws
catkin_make               # run catkin
```

Once the `catkin` workspace is compiled, it will automatically contain the files `CMakeLists.txt` and `package.xml`. There are other sub-folders in `catkin_ws` as well, as shown in Figure 24.3, which can be changed as needed.

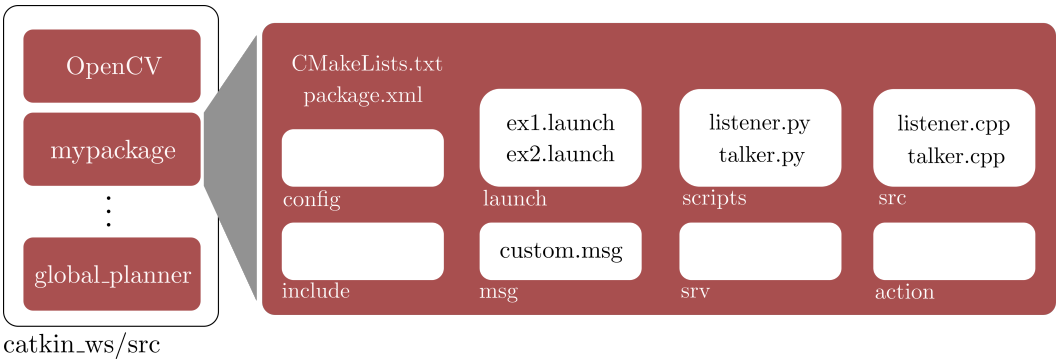


Figure 24.3: Components of an example ROS package named `mypackage` in a `catkin` workspace.

24.5.3 *Debugging*

Robot programming requires a lot of debugging. There are a few ways to debug your robot software, including (but not limited to):

- `rostopic`: a tool that monitors ROS topics in the command line,
- `rospy.loginfo()`: starts a background process that writes ROS messages to a ROS logger, viewable through a program such as `rqt_console`,
- `rosbag`: provides a convenient way to record a number of ROS topics for playback,
- `pdb`: provides a useful tool for debugging python scripts.

24.5.4 *Gazebo*

Gazebo[16] is a popular 3D dynamic simulator with the ability to accurately and efficiently simulate robots in complex environments (see Figure 24.4). While similar to game engines, Gazebo offers a higher fidelity physics simulation, a suite of sensors models, and interfaces for both users and programs. Gazebo can be used in any stage of robot development, from testing algorithms to running regression testing in realistic scenarios. Integration of Gazebo with ROS is possible via `gazebo_ros_pkgs` package.

[16] `http://gazebosim.org`

Figure 24.4: Screenshot of a scene in Gazebo.

Part VI

Advanced Topics in Robotics

25
Formal Methods

The safety and explainability of robotic systems has become increasingly important as applications for robotic systems transition to more unstructured and interactive environments. While one component to developing safe robots is to design robust and high-performing autonomy algorithms, another critical component is the analysis of the system's design. System analysis could come in several forms, including unit, component, or system-level testing, but one challenging aspect of testing is the determination of appropriate success criteria. It is also highly desirable to develop systems that are *provably correct* or *correct-by-construction* with respect to the stated success criteria.

This chapter introduces a set of rigorous mathematical tools and concepts for specifying desired behavior (i.e. requirements or specifications), proving that the system achieves the desired behavior, and synthesizing robot systems to be correct-by-construction. These mathematical tools are known as *formal methods*[1].

[1] E. M. Clarke et al. *Model Checking*. 2nd ed. MIT Press, 2018

Formal Methods

Formal methods provide a mathematical framework for reasoning about a system's specifications as well as analyzing whether the system's behavior guarantees their satisfaction. Approaches for synthesizing provably correct behavior can also be built on top of these tools and are commonly incorporated within the umbrella of formal methods. Historically these techniques have been developed within the computer science community, and have been used to study problems related to logic, automatically proving properties of algorithms, checking the correctness of properties of circuits, and more. However in this chapter formal methods will be explored within the context of autonomous systems.

Definition 25.0.1 (Formal Methods)**.** *Formal methods are mathematically based techniques for the specification, development, and verification of software and hardware systems.*

It is important to note that formal methods are not just particular solutions or algorithms but rather are a class of tools and formalisms. Accordingly, this chapter will not focus on a particular algorithm (solution) for applying formal

methods to problems in robotics, but will rather introduce several broadly used tools and techniques. Specifically, Section 25.1 will introduce *linear temporal logic*, which is a specialized language for writing specifications. Then the concept of the robotic system as a reactive model that can be *verified* to satisfy the stated specifications will be introduced in Section 25.2. Finally, the ability to *synthesize* robot systems to provably be able to satisfy a specification will be explored in Section 25.3.

25.1 Linear Temporal Logic

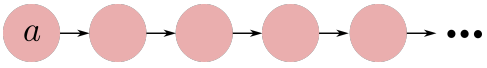
Linear temporal logic (LTL) is a mathematical *language* for formally expressing specifications or requirements on the system's behavior[2]. LTL is useful in robotics applications because it extends propositional logic[3] to handle *temporal* components (assuming discrete time steps), which are common in sequential decision making problems and other robotics tasks. For example propositional logic can be used to write a specification that "proposition a and proposition b must both be true", while LTL extends the possible specifications to include temporal constraints such as "proposition a must be true *until* proposition b is true".

[2] Similar to how ordinary differential equations provide a mathematical "language" that is useful for modeling the kinematics and dynamics of physical systems.

[3] A formalism for expressing logical operations including conjunction (and), disjunction (or), and negation (not).

The language of LTL can be expressed in terms of several atomic operators, the first few of which are inherited from propositional logic:

1. *true* or *false* (Boolean values)[4] represent Boolean constants.
2. $a, b, \ldots$ (propositional symbols) denote single variables that can either be true or false at the current time step.



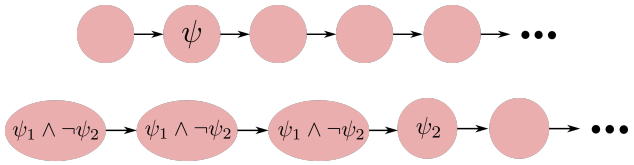
3. $\neg\psi$ (negation operator[5]) denotes the negation of ψ.
4. $\psi_1 \wedge \psi_2$ (conjunction "and" operator) which can be read as "ψ_1 and ψ_2".
5. $\psi_1 \vee \psi_2$ (disjunction "or" operator) which can be read as "ψ_1 or ψ_2". This operator can be expressed in terms of "and" and "not" as $\psi_1 \vee \psi_2 = \neg(\neg\psi_1 \wedge \neg\psi_2)$.
6. $\psi_1 \rightarrow \psi_2$ (implication operator) denotes that ψ_1 implies ψ_2. This operator can be expressed in terms of "not' and "or" as $\psi_1 \rightarrow \psi_2 = \neg\psi_1 \vee \psi_2$.

[4] Technically false can also be written as $\neg$true, where $\neg$ is the "not" operator.

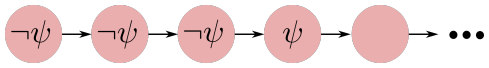
[5] Technically *false* can be written as $\neg$*true*.

Additional operators that are fundamental to LTL provide the capability for expressing temporal constraints:

7. $X\psi$ ("next" operator) denotes that ψ happens next (at the next time step).
8. $\psi_1 U \psi_2$ ("until" operator) denotes that ψ_1 should happen until ψ_2 happens.



9. $F\psi$ ("eventually" operator) denotes that ψ happens at some point in the *future*. This operator can be expressed in terms of the eventually operation as $F\psi = \text{true}\, U\, \psi$.



10. $G\psi$ ("always" operator) denotes that ψ happens globally (at all times). This operator can be expressed in terms of the negation and future operations as $G\psi = \neg F \neg \psi$.

From these atomic operators it is possible to define many new specifications through *composition*, and they can become arbitrarily complex as needed. A couple of common and useful compositions include:

11. $GF\psi$ ("infinitely often" composition) denotes that ψ will eventually happen an infinite number of times (i.e. globally, ψ will happen eventually). In other words there is always a ψ in the future.

12. $FG\psi$ ("stability"[6] composition) denotes that at some point in time ψ will be true for all time thereafter (i.e. eventually ψ will happen globally).

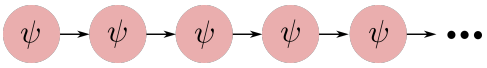
13. $G(\psi_1 \rightarrow F\psi_2)$ ("response" composition) denotes that for all time, whenever ψ_1 occurs then ψ_2 will occur sometime in the future. There are other useful variations on this composition, such as by replacing the F operator with X operator.

[6] This notion of stability is similar but not directly the same as the notions of stability from control theory.

Linear temporal logic provides a very powerful tool[7] for abstractly talking about time, and in general the specifications written using LTL can in a way be more "vague". For example the eventually operator F does not explicitly state *when* something must occur, just that at *some point* it will. It is also important to keep in mind that LTL is not an algorithm or technique for solving problems, but rather a language for *formulating* problems (i.e. for expressing properties of interest such as system specifications).

[7] There are also alternatives to LTL that provide even more powerful features, for example by not requiring a "linear" temporal structure but rather allowing for temporal "branching".

Example 25.1.1 (Coffee Machine Specification)**.** Consider a simple robot that makes coffee. This robot has a button that a user can press, and has two functions: grinding coffee beans and brewing coffee. The desired behavior of this robot could be expressed by the designer as: *if the start button is pressed, the robot will immediately start grinding beans for the next two cycles, and then brew the coffee for the next two cycles after that*. This specification, denoted as ϕ, could be



expressed via linear temporal logic as:

$$\phi : G\big(\text{button} \rightarrow \text{grind} \wedge X\text{grind} \wedge XX(\neg\text{grind} \wedge \text{brew}) \wedge XXX(\neg\text{grind} \wedge \text{brew})\big).$$

Note that the entire statement needs to be wrapped in the "always" operator to ensure this behavior can occur at any arbitrary time that a user presses the button.

25.2 *Verification*

System verification is the process of *proving* that the system's behavior will satisfy the stated requirements and specifications (often expressed using linear temporal logic). In the terminology of formal methods, this system is typically referred to as the *model*[8] and the output of the verification[9] procedure is a simple yes/no stating whether the model satisfies the specification. In this chapter the model will be denoted by P and the specification by ϕ, and the notation for stating that model P satisfies specification ϕ is $P \models \phi$ (which can be read as "P models ϕ").

[8] The model could refer to a piece of software, a hardware component, an individual algorithm, or even an entire robot.

[9] Also commonly referred to as *model checking*.

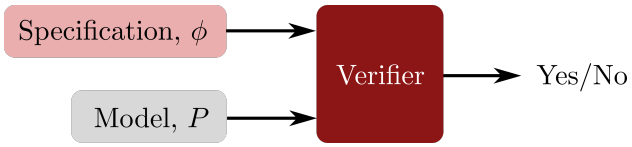


Figure 25.1: Given a system (model) and a specification, the process of verification proves whether the system satisfies the specification.

In this chapter it is assumed that the model P is a *reactive system*, meaning that its behavior is defined based on inputs i which effect the system's outputs o[10]. In contrast to robotic control problems where the "inputs" are generally the control inputs determined by the control algorithm, the inputs i within this context refer to signals coming from the environment. The outputs o of the model can then be thought of as the result of the system's decision making or underlying algorithm/process. As was mentioned previously, the model P can take on many forms depending on whether the system is a hardware component, software component, algorithm, finite state machine, or even simply a mathematical function such as a machine learning model or control law.

[10] The inputs and outputs occur at each time step, and the behavior is assumed to be non-terminating.

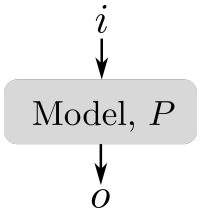


Figure 25.2: The model P is a system with inputs i and outputs o. These inputs and outputs are used to express the model's specification ϕ.

For a given model P the specification ϕ is assumed to be written in terms of the input and output sequences (i.e. the behavior is defined by the inputs and outputs of the system). Specifically, these sequences will be denoted as $\hat{i} = (i_0, i_1, \dots)$ and $\hat{o} = (o_0, o_1, \dots)$. With these definitions, the expression that P satisfies ϕ can equivalently be written as $P \models \phi$ or $\hat{i} \cup \hat{o} \models \phi$.

To summarize, the problem of model verification is to simply determine whether the input-output behavior of the model P guarantees that ϕ is satisfied

for all possible input sequences. There are several existing techniques that can perform system verification, and they may be tailored to the specific model form[11,12].

[11] M. Kwiatkowska, G. Normal, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. 2011, pp. 585–591

[12] G. Katz et al. "The Marabou Framework for Verification and Analysis of Deep Neural Networks". In: *Computer Aided Verification*. 2019, pp. 443–452

25.3 Reactive Synthesis

Given a reactive model P and a LTL specification ϕ, the problem of verification is to determine whether the behavior of P satisfies ϕ for all possible input sequences $\hat{i}$. But several important questions remain: how should the system be designed, and what should be changed if the verification step shows that $P \not\models \phi$? *Reactive synthesis* addresses these problems by *synthesizing* the system model P to be correct-by-construction. In other words, in reactive synthesis the specification ϕ is first defined and then a model is constructed from scratch to satisfy the specification.

Figure 25.3: Given a specification ϕ, the process of reactive synthesis generates a model P that realizes the specification under all possible environmental inputs.

25.3.1 Specification Satisfiability and Realizability

The first step in reactive synthesis is to determine whether a model[13] even exists which can satisfy the LTL specification ϕ for all possible input sequences. If no such model exists, then the system designer should reevaluate the specification itself.

[13] Technically speaking, a *finite state model*.

In the nomenclature of formal methods, the specification ϕ is said to be *realizable* if it can be satisfied for all possible input sequences, and it is *satisfiable* if there exists at least one input sequence leading to satisfaction. These properties can be more rigorously defined in terms of the input and output sequences that describe the system's behavior:

Definition 25.3.1 (Satisfiability). *A specification ϕ is satisfiable if for some input sequence there exists an output sequence that satisfies the specification. Mathematically:*

$$\exists\, \hat{i} = (i_0, i_1, \dots), \quad \exists\, \hat{o} = (o_0, o_1, \dots), \quad s.t.\ \hat{i} \cup \hat{o} \models \phi.$$

Definition 25.3.2 (Realizability). *A specification ϕ is realizable if for all possible input sequences there exists an output sequence that satisfies the specification. Mathematically:*

$$\forall\, \hat{i} = (i_0, i_1, \dots), \quad \exists\, \hat{o} = (o_0, o_1, \dots), \quad s.t.\ \hat{i} \cup \hat{o} \models \phi.$$

Obviously the property of satisfiability is weaker than realizability, and realizability is much more important in practice. For example in order to guarantee safety in a rigorous way it is not sufficient to show that the system will be safe

under a single scenario, but rather it should be shown for all scenarios. However, designing specifications that are realizable can be quite challenging, even in seemingly simple problems. As an example consider again the coffee machine robot from Example 25.1.1.

Example 25.3.1 (Coffee Machine Realizability)**.** The coffee machine robot from Example 25.1.1 had inputs $I = \{\text{button}\}$ and outputs $O = \{\text{grind}, \text{brew}\}$. For simplicity let $i_{\text{button}} = 1$ and $i_{\text{button}} = 0$ denote the button is pressed and not pressed, respectively. Additionally let $o_{\text{grind}} = 1$ and $o_{\text{brew}} = 1$ denote that the actions are occurring and let them be zero otherwise.

Recall that the linear temporal logic specification for the robot's behavior was defined as:

$$\phi : G\big(\text{button} \rightarrow \text{grind} \wedge X\text{grind} \wedge XX(\neg\text{grind} \wedge \text{brew}) \wedge XXX(\neg\text{grind} \wedge \text{brew})\big).$$

This specification can now be analyzed to determine whether it is realizable. First, notice that one possible sequence of inputs and outputs that satisfies this specification is:

$$\begin{aligned}(i_{\text{button}}, o_{\text{grind}}, o_{\text{brew}})_k = (0,\, 0,\, 0)_0,\\ (1,\, 1,\, 0)_1,\\ (0,\, 1,\, 0)_2,\\ (0,\, 0,\, 1)_3,\\ (0,\, 0,\, 1)_4,\\ (0,\, 0,\, 0)_5,\\ \vdots\end{aligned}$$

Because there exists a sequence that satisfies the specification ϕ it is by definition *satisfiable*. However, consider a second input sequence $\hat{i} = (0, 1, 1, 0, 0, \dots)$ where the coffee machine's button is pressed twice in a row:

$$\begin{aligned}(i_{\text{button}}, o_{\text{grind}}, o_{\text{brew}})_k = (0,\, 0,\, 0)_0,\\ (1,\, 1,\, 0)_1,\\ (1,\, 1,\, 0)_2,\\ (0,\, ?,\, 1)_3,\end{aligned}$$

At time step $k = 3$ there is no combination of outputs that will satisfy the specification, since the first button press requires that $o_{\text{grind},3} = 0$ but the second button press requires that $o_{\text{grind},3} = 1$! Therefore by definition this specification is not *realizable*[14].

[14] Does this mean it is impossible to automate a coffee maker? No! It just demonstrates that writing *specifications* can be challenging.

25.3.2 *Synthesis for Realizable LTL Specifications*

If a LTL specification ϕ is realizable, then the synthesis problem seeks to find a finite state system that satisfies ϕ under all possible inputs. This can be accomplished by formulating the problem as a two-player game where the objective

is for the system to generate "winning" outputs while the environment generates adversarial inputs. The two-player game formulation can be expressed mathematically by defining the following components:

1. With the inputs I and outputs O, at each time step the environment gets to choose from a set of $2^{|I|}$ actions and the system gets to choose from a set of $2^{|O|}$ actions.

2. The strategy of the system is expressed as a function $f : (2^{|I|})^* \to 2^{|O|}$, where f is a function from a finite sequence of environmental inputs to a specific output.

3. The linear temporal logic specification ϕ is defined by the input and output sequences.

4. The game is played for an infinitely long horizon, generating sequences $\hat{i} = (i_0, i_1, \dots)$ and $\hat{o} = (o_0, o_1, \dots)$.

5. The game is won if $\hat{i} \cup \hat{o} \models \phi$.

The process of converting a problem specification into this two-player game follows two main steps. First, the specification is converted into a *non-deterministic Büchi automaton* and then the automaton is determinized to yield the game. Once the game is appropriately formulated, it can be solved using existing algorithms to generate the policy f that defines the system's behavior. Unfortunately, converting the specification into the automaton is computationally very challenging! In fact the computational complexity is *doubly-exponential* in the size of the specification[15], which significantly limits the complexity of the problems that can be considered[16].

While the precise details for converting a specification into a two-player game and solving the game are beyond the scope of this chapter, the process can be explored visually through the following example.

[15] A doubly exponential function has the form $f(x) = a^{b^x}$.

[16] This is one of the most significant limitations of formal methods in practical robotic settings, and approaches to overcome this complexity are still a topic of research.

Example 25.3.2 (Simple Reactive Synthesis Problem)**.** Consider the LTL specification $\phi : G(r \to Xg)$ which states that whenever a request r is received the system should provide a grant g in the next time step. In this problem $I = \{r\}$ where r denotes a request or no request and $O = \{g\}$ where g specifies if a grant was made or not. The first step in transforming this specification into the two-player synthesis game is to generate the following Büchi automaton representation, as shown in Figure 25.4. In Figure 25.4 the variables q_0 and q_1 represent states of the automaton, and the transitions between the states are dependent on the environmental inputs and the system's behavior.

The two-player game is generated from this Büchi automaton[17] by introducing intermediate states as well as the unsafe "contradiction" state (denoted in Figure 25.5 as $\perp$). This game is represented graphically in Figure 25.5 where the $*$ denotes that any action could be taken by the model and the small grey circles represent the intermediate states. The system can "win" this two-player game

[17] This automaton is already deterministic, so no determinizing step is needed.

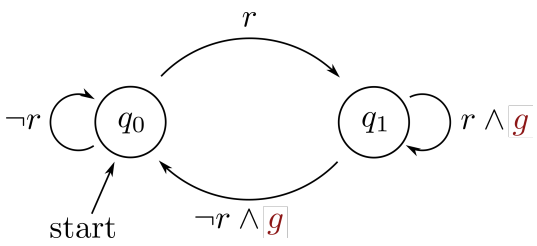


Figure 25.4: The Büchi automaton representation of the LTL specification $\phi : G(r \rightarrow Xg)$.

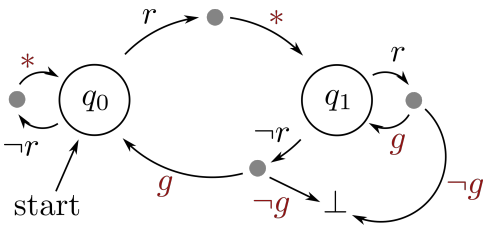


Figure 25.5: The two-player game representation derived from the Büchi automaton in Figure 25.4.

by ensuring that the contradiction state is never reached, which then defines the system's behavior! By analyzing Figure 25.5, it turns out that one "winning" strategy strategy (behavior) is for the system to *always* provide a grant! This strategy is shown graphically in Figure 25.6.

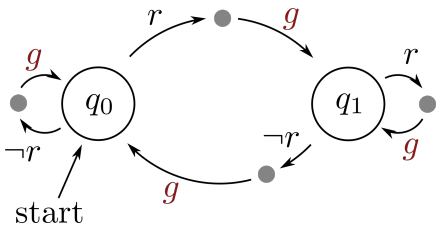


Figure 25.6: A strategy for the system in Example 25.3.2 to ensure the specification is met is to *always* provide grants, which is guaranteed to avoid the contradiction state in the two-player game shown in Figure 25.5.

26
Robotic Manipulation

Robotic manipulation, where a robot *physically* interacts and changes the environment, is one of the most challenging tasks in robot autonomy from the perspectives of perception, planning, and control. Consider a simple pick-and-place problem: the robot needs to identify the object, find a good place to grasp, stably pick up the object, and move it to a new location, all while ensuring no part of the robot collides with the environment. In practice even this simple task can become much harder, for example if other objects are in the way and must be moved first, if the object does not have particularly good grasping features, if the weight, size, and surface texture of the object is unknown, or if the lighting is poor[1]. Manipulation tasks are also commonly composed of sequences of interactions, such as making a sandwich or opening a locked door. This chapter focuses on *grasping*[2], which is a fundamental component to all manipulation tasks.

[1] Generally speaking the infinite variability of the real world makes *robust* manipulation extremely difficult.

[2] D. Prattichizzo and J. C. Trinkle. "Grasping". In: *Springer Handbook of Robotics*. Springer, 2016, pp. 955–988

Grasping

Grasping is a fundamental component of robotic manipulation that focuses on obtaining complete control of an object's motion (in contrast to other interactions such as pushing).

Definition 26.0.1 (Grasp)**.** *A grasp is an act of restraining an object's motion through application of forces and torques at a set of contact points.*

Grasping is challenging for several reasons:

1. The configuration of the gripper may be high-dimensional. For example the Allegro Hand (Figure 26.1) has 4 fingers with 3 joints each for a total of 12 dimensions. Plus there are an additional 6 degrees of freedom in the wrist posture (position and orientation), and all of these degrees of freedom vary continuously.

2. Choosing contact points can be difficult. An ideal choice of contact points would lead to a robust grasp, but the space of feasible contacts is restricted

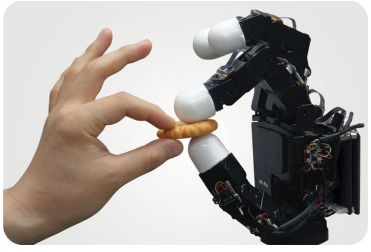


Figure 26.1: The Allegro Hand. Image retrieved from `wiki.wonikrobotics.com`.

by the gripper's geometry. A rigid body object also has 6 degrees freedom, which affects where the contact points are located in the robot's workspace.

3. While the robot is attempting the grasp it must be sure that its entire body does not come into collision with the environment.

4. Once a grasp has been performed it is important to evaluate how robust the grasp is. While the grasp quality would ideally be optimized during the planning step, it may be important to also check retroactively in case uncertainty led to a different grasp than planned.

To address each of these challenges, the grasp can be subdivided into parts: planning, acquisition/maintenance, and evaluation. This chapter will focus on the fundamentals of how a grasp can be *modeled* and *evaluated* from a mathematical perspective, as well as how grasps can be *planned*[3] using grasp force optimization. Learning-based approaches to grasping and manipulations will also be discussed at a high level in Section 26.4 and 26.5.

[3] Part of grasp planning also includes the motion planning of the entire robot, but the focus of this chapter is on the grasp itself.

26.1 Grasp Modeling

A grasp plan may be parameterized in several ways, including by the approach vector or wrist orientation of the gripper, by the initial finger configuration, or directly by points of contact with the object. However, regardless of the planning parameterization the resulting contacts between the gripper and the object will define the quality of the grasp. Therefore it is useful and convenient for grasp modeling to consider the contact points as the interaction interface between the gripper and object.

26.1.1 Contact Types

There are generally three types of contact that can occur in grasping scenarios:

1. *Point*: a point contact occurs when a single point comes in contact with either another point, a line, or a plane. A point contact is only stable if it is a point-on-plane contact[4], point-on-point or point-on-line contacts are unstable.

2. *Line*: line contacts occur when a line comes in contact with another line or a plane. Line-on-plane and line-on-nonparallel line contacts are stable, but line-on-parallel line contacts are unstable. Line contacts can also be represented as two point contacts.

3. *Plane*: plane-on-plane contacts are always stable. Plane contacts can also be represented as point contacts by converting a distribution of normal forces across a region into a weighted sum of point forces at the vertices of the region's convex hull.

[4] Point-on-plane contacts are by far the most commonly modeled contact types and will almost always be used in grasp analysis.

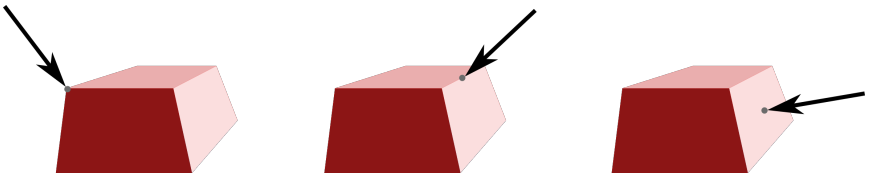


Figure 26.2: Grasping contacts are generally either point-on-point (left), point-on-line (middle), point-on-plane (right).

26.1.2 *Point-on-Plane Contact Models*

Point-on-plane contact models are by far the most commonly used for grasping since the possible contact points for most objects are almost always surface points (and not sharp edges or points). The purpose of the contact model is to specify the admissible forces and torques that can be transmitted through a particular contact. Considering a local reference frame defined at the contact point with the z direction pointing along the object's surface normal (with the positive direction defined as into the object), the force $\boldsymbol{f}$ can be written as:

$$\boldsymbol{f} = \boldsymbol{f}_{\text{normal}} + \boldsymbol{f}_{\text{tangent}},$$

where $\boldsymbol{f}_{\text{normal}} = [0,\ 0,\ f_z]^\top$ is the vector component along the normal direction (with magnitude f_z) and $\boldsymbol{f}_{\text{tangent}} = [f_x,\ f_y,\ 0]^\top$ is the vector component tangent to the surface. For all types of contact only an inward force can be applied, therefore $f_z \geq 0$. Three types of contact models are commonly used, and each defines a set $\mathcal{F}$ of admissible forces that can be applied through the contact:

1. Frictionless Point Contact: forces can only be applied along the surface normal, no torques or forces tangential with the surface are possible ($\boldsymbol{f}_{\text{tangent}} = 0$):

   $$\mathcal{F} = \{\boldsymbol{f}_{\text{normal}} \mid f_z \geq 0\}.$$

   These types of contact models are more common in form closure grasps.

2. Point Contact with Friction[5]: it is possible to apply forces in directions other than just the surface normal. The admissible forces (i.e. forces that don't lead to slipping) are typically defined by a *friction cone*:

   $$\mathcal{F} = \{\boldsymbol{f} \mid \|\boldsymbol{f}_{\text{tangent}}\| \leq \mu_s \|\boldsymbol{f}_{\text{normal}}\|, \quad f_z \geq 0\}.$$

   where μ_s is the static friction coefficient associated with the surface (see Figure 26.3).

   A pyramidal inner-approximation of the friction cone is often more useful from a computational standpoint, since its definition only requires a *finite* set of vectors (see Figure 26.4). The point contact with friction model is more common in force closure grasps.

3. Soft-finger Contact Model: allows for a torque τ_{normal} around the surface normal axis and also includes a friction cone for the forces as in the point

[5] Also referred to as the *hard finger* model.

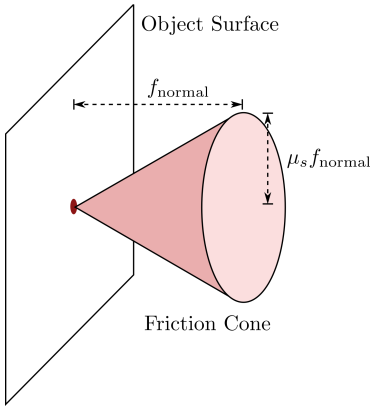


Figure 26.3: Friction cone defined by a static coefficient of friction μ_s.

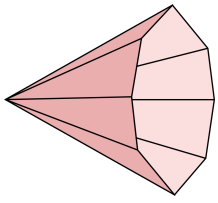


Figure 26.4: Linearized friction cone to *inner* approximate the true cone.

contact with friction model. The admissible torques are also constrained by friction:

$$\mathcal{F} = \{(\boldsymbol{f}, \tau_{\text{normal}}) \mid \|\boldsymbol{f}_{\text{tangent}}\| \leq \mu_s \|\boldsymbol{f}_{\text{normal}}\|, \quad f_z \geq 0, \quad |\tau_{\text{normal}}| \leq \gamma f_z\}.$$

where $\gamma > 0$ is the torsional friction coefficient.

26.1.3 Wrenches and Grasp Wrench Space

Under the assumption of a specific contact model, a grasp (defined by a set of contact points) can be quantified and evaluated by determining the *grasp wrench space*[6], which defines how the grasp can influence the object through an applied wrench.

[6] The grasp wrench space is a *subset* of the *wrench space* $\mathbb{R}^n$, where $n = 6$ in 3D settings and $n = 3$ in 2D settings.

Definition 26.1.1 (Wrench)**.** *A wrench is a vector valued quantity that describes the forces and torques being applied to an object. For a force $\boldsymbol{f} \in \mathbb{R}^3$ and torque $\boldsymbol{\tau} \in \mathbb{R}^3$ applied at the object's center of mass, the wrench is the stacked vector:*

$$\boldsymbol{w} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{\tau} \end{bmatrix} \in \mathbb{R}^6,$$

and is typically written with respect to a frame fixed in the body.

Each contact point i in a grasp applies a wrench to the object. Additionally the torque $\boldsymbol{\tau}_i$ can be computed by $\boldsymbol{\tau}_i = \boldsymbol{d}_i \times \boldsymbol{f}_i$ where $\boldsymbol{d}_i$ is the vector defining the position of the i-th contact point with respect to the object's center of mass. The wrench can then be written as:

$$\boldsymbol{w}_i = \begin{bmatrix} \boldsymbol{f}_i \\ \lambda(\boldsymbol{d}_i \times \boldsymbol{f}_i) \end{bmatrix}, \tag{26.1}$$

where the constant $\lambda \in \mathbb{R}$ is arbitrary but can be used to scale the torque magnitude if desired[7].

[7] If the forces $\boldsymbol{f}_{i,j}$ are dimensional a value of $\lambda = 1$ is common. When the forces are unit-dimension (i.e. scaled by their maximum magnitude), λ could be chosen to non-dimensionalize the entire wrench $\boldsymbol{w}_i$ by non-dimensionalizing the distance vector $\boldsymbol{d}_i$ (i.e. scale by an object size metric).

Using this definition of a wrench[8], a grasp can be defined as the set of all possible wrenches that can be achieved by the grasp's contact points. Mathematically, an admissible force $\boldsymbol{f}_i$ applied at the i-th contact point can be linearly mapped into the corresponding wrench on the object as $G_i \boldsymbol{f}_i$, where G_i is a wrench basis matrix that also includes a transformation from the local contact reference frame to an object-defined global reference frame. Therefore the total wrench on the object from all contacts is:

[8] For a soft-finger contact model the additional torque term must also be included.

$$\boldsymbol{w} = \sum_{i=1}^{k} G_i \boldsymbol{f}_i = G \begin{bmatrix} \boldsymbol{f}_1 \\ \vdots \\ \boldsymbol{f}_k \end{bmatrix}, \quad G = \begin{bmatrix} G_1 & \dots & G_k \end{bmatrix}, \tag{26.2}$$

where the combined matrix G is referred to as the *grasp map* (which varies depending on the type of contact model used).

The *grasp wrench space* can then be defined as:

Definition 26.1.2 (Grasp Wrench Space)**.** *The grasp wrench space $\mathcal{W}$ for a grasp with k contact points is the set of all possible wrenches $\boldsymbol{w}$ that can be applied to the object through admissible forces:*

$$\mathcal{W} := \{\boldsymbol{w} \mid \boldsymbol{w} = \sum_{i=1}^{k} G_i \boldsymbol{f}_i, \quad \boldsymbol{f}_i \in \mathcal{F}_i, \quad i = 1, \ldots, k\}. \tag{26.3}$$

In other words, the grasp wrench space is defined by the output of (26.2) over all possible applied force combinations $\{\boldsymbol{f}_i\}_{i=1}^{k}$. If the grasp wrench space is large the grasp can compensate for a bigger set of external wrenches that might be applied to the object, leading to a more robust grasp.

Example 26.1.1 (Computing a Grasp Wrench Space from Friction Cones)**.** Consider a grasping problem with k contact points with friction, and let contact point i be associated with a linearized friction cone $\mathcal{F}_i$ whose edges are defined by the set of m forces:

$$\{\boldsymbol{f}_{i,1}, \boldsymbol{f}_{i,2}, \ldots \boldsymbol{f}_{i,m}\},$$

such that any force $\boldsymbol{f}_i \in \mathcal{F}_i$ can be written as a positive combination of these vectors:

$$\boldsymbol{f}_i = \sum_{j=1}^{m} \alpha_{i,j} \boldsymbol{f}_{i,j}, \quad \alpha_{i,j} \geq 0.$$

The condition $\sum_{j=1}^{m} \alpha_{i,j} \leq 1$ will also be imposed to constrain the overall magnitude[9]. Geometrically, this means that the friction cone $\mathcal{F}_i$ is the convex hull of the points $\boldsymbol{f}_{i,j}$ and the origin of the local contact reference frame (see Figure 26.5).

[9] In practice the physical hardware has limitations on the magnitude of the normal forces that can be applied.

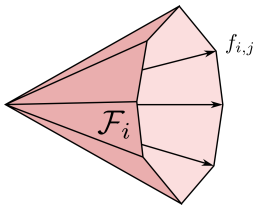


Figure 26.5: Any force $\boldsymbol{f}_i \in \mathcal{F}_i$ can be written as a convex combination of the forces along the edge vectors $\boldsymbol{f}_{i,j}$.

This friction cone can then be mapped into the wrench space using (26.1). Assuming the forces $\boldsymbol{f}_{i,j}$ and position vector $\boldsymbol{d}_i$ are already expressed in a reference frame fixed in the object that is common to all i contact points, the grasp wrench space $\mathcal{W}$ can be written as:

$$\mathcal{W} = \{\boldsymbol{w} \mid \boldsymbol{w} = \sum_{i=1}^{k} \boldsymbol{w}_i, \quad \boldsymbol{w}_i = \sum_{j=1}^{m} \alpha_{i,j} \boldsymbol{w}_{i,j}, \quad \boldsymbol{w}_{i,j} = \begin{bmatrix} \boldsymbol{f}_{i,j} \\ \lambda(\boldsymbol{d}_i \times \boldsymbol{f}_{i,j}) \end{bmatrix},$$
$$\sum_{j=1}^{m} \alpha_{i,j} \leq 1, \quad \alpha_{i,j} \geq 0\}.$$

In other words, the grasp wrench space is defined by taking the Minkowski sum over the sets of wrenches that can be generated from each individual contact!

For example, consider the 2D problem shown in Figure 26.6 where there are $k = 2$ contact points with friction. The friction cones are defined by the convex hull of the vectors $\{\boldsymbol{f}_{1,1}, \boldsymbol{f}_{1,2}\}$ and $\{\boldsymbol{f}_{2,1}, \boldsymbol{f}_{2,2}\}$ (and their origins) and the distance vectors from the center of mass to the contact points are $\boldsymbol{d}_1$ and $\boldsymbol{d}_2$. The force vectors $\boldsymbol{f}_{i,j}$ are then mapped into the wrenches $\boldsymbol{w}_{i,j}$ (shown on a 2D plot of vertical force f_y and torque τ in Figure 26.6, ignoring the horizontal force components f_x). The grasp wrench space $\mathcal{W}$ is then shown in the grey region of the wrench space, where the solid grey line is the boundary of $\mathcal{W}$.

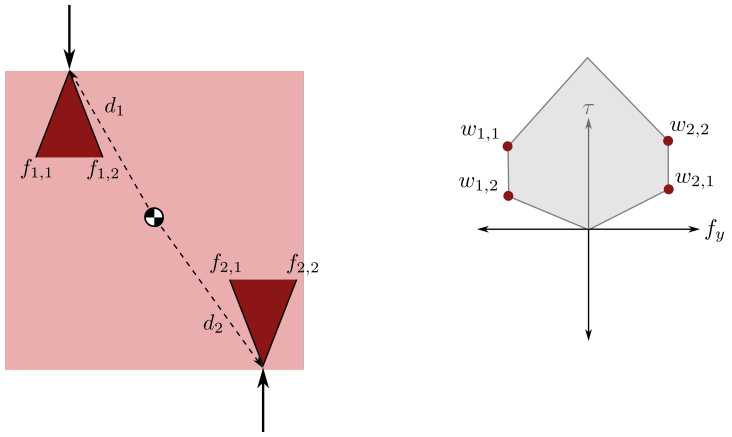


Figure 26.6: An example 2D grasp consisting of two point contacts with friction. The friction cones shown in the figure on the left yield the grasp wrench space in the figure on the right (showing only the vertical force and torque dimensions). Note that the grasp wrench space is bounded because it is assumed the magnitude of the contact forces are bounded. The solid grey line represents the boundary of $\mathcal{W}$.

26.2 Grasp Evaluation

Now that the basics of grasp modeling have been introduced[10] it is possible to explore techniques for evaluating whether a grasp is "good". In particular, an ideal grasp is one that has *closure*.

[10] contact types, contact models, grasp wrench spaces

Definition 26.2.1 (Grasp Closure)**.** *Grasp closure occurs when the grasp can be maintained for every possible disturbance load.*

For example having grasp closure on a book would enable the gripper to maintain its grasp even if the book was hit by another object or if another book was suddenly stacked on top of it. In practice it may not be reasonable to assume that every *magnitude* disturbance load could be accounted for, but the concept of closure is useful nonetheless.

[11] Also called power grasps or enveloping grasps. A grasp must have at least seven contacts to provide form closure for a 3D object.

It can also be helpful to distinguish between two types of grasp closure. A *form closure*[11] grasp typically has the gripper joint angles locked and there is no "wiggle" room for the object (i.e. the object is kinematically constrained). Alternatively, a *force closure*[12] grasp uses forces applied at contact points to be able to *resist* any external wrench. Force closure grasps typically rely on *friction* and generally require fewer contact points than are required for form closure, but may not be able to actually cancel all disturbance wrenches if the friction forces are too weak. This chapter will primarily focus on evaluating force closure grasps since these are most common in robotics.

[12] Also called a precision grasp. Under a point contact with friction model, a grasp must have at least three contacts to provide force closure for a 3D object.

26.2.1 Force Closure Grasps

The concept of force closure can be related to the grasp modeling concepts from Section 26.1:

Definition 26.2.2 (Force Closure Grasp)**.** *A grasp is a force closure grasp if for any*

Figure 26.7: Examples of grasps with form closure (left) and force closure under the soft-finger contact model (right).

external wrench $\boldsymbol{w}$ *there exist contact forces* $\{\boldsymbol{f}_i\}_{i=1}^k$ *such that:*

$$-\boldsymbol{w} = \sum_{i=1}^{k} G_i \boldsymbol{f}_i. \quad \boldsymbol{f}_i \in \mathcal{F}_i, \quad i = 1, \ldots, k,$$

or equivalently such that:

$$-\boldsymbol{w} \in \mathcal{W}.$$

This definition implies that the grasp wrench space must satisfy $\mathcal{W} = \mathbb{R}^n$ for a force closure grasp, which implicitly assumes that the contact forces can be infinite in magnitude.[13] Since real hardware has limitations on the magnitude of the applied contact forces, a more practical definition of force closure is to be able to *resist* any external wrenches. The conditions for force closure can be summarized by the following theorem:

[13] For 2D objects $n = 3$ and for 3D objects $n = 6$.

Theorem 26.2.3. *In an n-dimensional vector space with:*

$$\mathcal{W} := \{\boldsymbol{w} \mid \boldsymbol{w} = \sum_{k=1}^{N} \beta_k \boldsymbol{w}_k, \quad \beta_k \geq 0\},$$

$\mathcal{W} = \mathbb{R}^n$ *if and only if the set* $\{\boldsymbol{w}_k\}_{k=1}^N$ *contains at least* $n+1$ *vectors, n of the vectors are linearly independent, and there exists scalars* $\beta_k > 0$ *such that:*

$$\sum_{k=1}^{N} \beta_k \boldsymbol{w}_k = 0.$$

From a practical perspective this theorem specifies a minimum number of different wrenches that must be used as a basis for the grasp wrench space, and also states that it must be possible for the grasp to apply zero wrench *even when some of the contact forces are non-zero*. These conditions are equivalent to saying that grasp wrench space $\mathcal{W}$ must contain the origin in its *interior*[14].

[14] The grasp is not in force closure if the origin is on the *boundary* of $\mathcal{W}$.

Note that in the practical case where the applied contact forces are assumed to be bounded, the conditions of Theorem 26.2.3 must still hold to guarantee the origin is in the interior of $\mathcal{W}$, which is required to resist any external wrench. The implications of this theorem are explored further in the following examples.

Example 26.2.1 (2D Object (Forces Only))**.** Consider a simplified 2D problem where instead of complete force closure (i.e. ability to withstand any *wrench*) it is sufficient to only require the cancellation of external *forces*. In this case $n = 2$ and Theorem 26.2.3 states that it must be possible for the grasp to generate 3 force vectors where 2 are linearly independent and where:

$$\beta_1 \boldsymbol{f}_1 + \beta_2 \boldsymbol{f}_2 + \beta_3 \boldsymbol{f}_3 = 0, \quad \beta_1, \beta_2, \beta_3 > 0.$$

Two examples grasps are shown in Figures 26.8 and 26.9. In Figure 26.8 the three contacts are frictionless, but even though there are 3 possible force vectors with 2 linearly independent, there is no way to generate zero force with non-zero forces at each contact! Therefore this grasp does not have force closure.

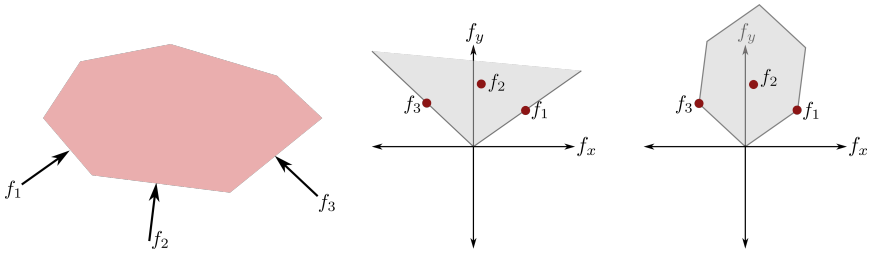


Figure 26.8: A 2D grasp with frictionless contacts that cannot compensate for all possible external forces on the object. The middle and right-side figures show the space of all possible applied forces for the cases of unbounded and bounded contact force magnitude, respectively.

Alternatively, Figure 26.9 shows a case where it is possible to have force closure using a point contact without friction and a point contact with friction (a hypothetical example). In this case all of the conditions in Theorem 26.2.3 are satisfied, and it can be seen that the origin is contained in the interior of the space of possible applied forces.

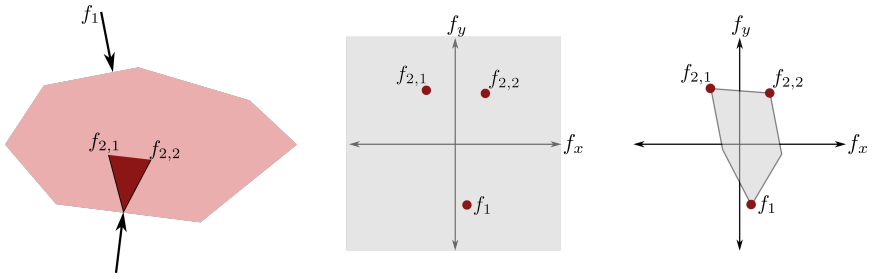


Figure 26.9: A 2D grasp consisting of a contact with friction and a contact without friction. The middle and right-side figures show the space of possible forces for the cases of unbounded and bounded magnitude, respectively. In the unbounded case it is possible to compensate for any external force, and in the bounded case it is possible to *resist* an arbitrary force.

Example 26.2.2 (2D Object)**.** In the more general case with 2D objects where the torque is also considered, the grasp wrench space is in 3D (i.e. $\mathcal{W} \subseteq \mathbb{R}^3$). Therefore, Theorem 26.2.3 states the grasp wrench space satisfies $\mathcal{W} = \mathbb{R}^3$ if and only if it is possible for the grasp to generate at least 4 different wrenches, with 3 being linearly independent, and where:

$$\beta_1 \boldsymbol{w}_1 + \beta_2 \boldsymbol{w}_2 + \beta_3 \boldsymbol{w}_3 + \beta_4 \boldsymbol{w}_4 = 0, \quad \beta_1, \beta_2, \beta_3, \beta_4 > 0.$$

If frictionless contacts are assumed these conditions require *at least* 4 contact points and in the friction case *at least* 2 contacts are required.

Consider again the grasp shown in Example 26.1.1 (Figure 26.6). The 4 edges of the friction cones create a set of 3 linearly independent wrenches, but there is no way to generate zero wrench with non-zero contact forces. This is evident in the fact that it is not possible to generate a negative torque, which means the grasp is not in force closure[15]. An alternative grasp that is in force closure is shown in Figure 26.10, which leverages a third contact point to ensure the grasp achieves stability.

[15] Notice again that the origin is not contained in the interior of the grasp wrench space!

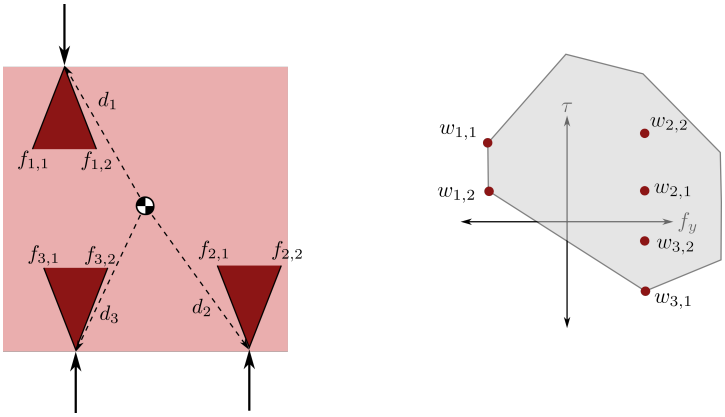


Figure 26.10: A 2D grasp consisting of three point contacts with friction. The friction cones shown in the figure on the left yield the grasp wrench space in the figure on the right (showing only the vertical force and torque dimensions and assuming bounded contact force magnitudes). This grasp is in force closure because it can resist any external wrench (the origin is contained in the interior of $\mathcal{W}$).

Example 26.2.3 (3D Object)**.** For 3D objects the grasp wrench space is in 6D (i.e. $\mathcal{W} \subseteq \mathbb{R}^6$). Theorem 26.2.3's conditions therefore require that the grasp to be able to generate at least 7 different wrenches, with 6 being linearly independent, and where:

$$\sum_{k=1}^{7} \beta_k \boldsymbol{w}_k = 0, \quad \beta_k > 0.$$

If frictionless contacts are assumed these conditions require *at least* 7 contact points and in the friction case *at least* 3 contacts are required.

26.2.2 *Grasp Wrench Hull*

The grasp wrench space $\mathcal{W}$ defines the set of all possible wrenches that can be applied to an object by a grasp, but unfortunately computing this set can be quite cumbersome in practice. One alternative approach for characterizing a grasp is through the definition of the *grasp wrench hull*, which can be efficiently computed. Given a set of linearized friction cones $\mathcal{F}_i$ defined by the set of bounded forces $\{\boldsymbol{f}_{i,1}, \boldsymbol{f}_{i,2}, \ldots, \boldsymbol{f}_{i,m}\}$ for each contact in the grasp, the wrench hull $\tilde{\mathcal{W}}$ is mathematically defined as:

$$\tilde{\mathcal{W}} = \{\boldsymbol{w} \mid \boldsymbol{w} = \sum_{i=1}^{k}\sum_{j=1}^{m} \alpha_{i,j}\boldsymbol{w}_{i,j}, \quad \boldsymbol{w}_{i,j} = \begin{bmatrix} \boldsymbol{f}_{i,j} \\ \lambda(\boldsymbol{d}_i \times \boldsymbol{f}_{i,j}) \end{bmatrix}, \quad \sum_{i=1}^{k}\sum_{j=1}^{m} \alpha_{i,j} = 1, \quad \alpha_{i,j} \geq 0\},$$

where $\boldsymbol{d}_i$ is again the vector from the object center of mass to contact point i. Note that this is almost identical to the grasp wrench space definition ex-

cept that the constraint $\sum_{j=1}^{m} \alpha_{i,j} \leq 1$ for all i has been replaced by the constraint $\sum_{i=1}^{k} \sum_{j=1}^{m} \alpha_{i,j} = 1$. Put simply, the wrench hull is the convex hull of the wrenches $\boldsymbol{w}_{i,j}$! The difference between the grasp wrench space and the wrench hull is shown in Figure 26.11 for the grasp from Example 26.2.2.

Importantly the property $\tilde{\mathcal{W}} \subseteq \mathcal{W}$ holds by definition. Therefore grasp force closure is also guaranteed when the origin is in the interior of the wrench hull space. This fact, coupled with the fact that $\tilde{\mathcal{W}}$ is easier to compute than $\mathcal{W}$, makes it a useful characterization of grasps for evaluating grasp quality.

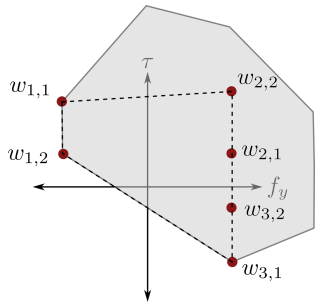


Figure 26.11: Difference between grasp wrench space $\mathcal{W}$ (grey area) and wrench hull $\tilde{\mathcal{W}}$ (area enclosed by black dashed line) for the grasp in Example 26.2.2.

26.2.3 *Grasp Quality*

If the gripper could apply contact forces with infinite magnitude then a "good" grasp could simply be defined as one that is in force closure. However a more practical definition of grasp quality should be based on the assumption that the magnitude of the contact forces is bounded. In other words, a metric for grasp quality should quantify *how well the grasp can resist external wrenches for a given bound on the contact force*.

To accomplish this, grasp quality metrics can be defined based on the definition of the grasp wrench hull $\tilde{\mathcal{W}}$. In particular, a useful metric is the radius of the largest ball centered at the origin that is completely contained in the grasp wrench hull (Figure 26.12). This metric is useful for the following reasons:

1. If the radius is zero, the origin is not contained in the interior of the wrench hull and therefore the grasp is not in force closure.

2. For a radius greater than zero, the metric represents the magnitude of the *smallest* external wrench that pushes the grasp to the limits. The direction from the origin to where the ball touches the boundary of $\mathcal{W}$ identifies the (opposite) direction in which the grasp is least able to resist external wrenches.

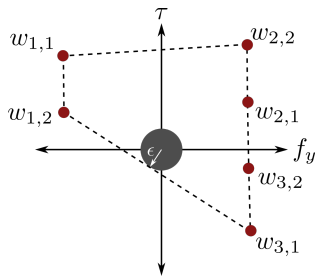


Figure 26.12: Grasp quality can be measured as the radius ϵ of the largest ball contained in the grasp wrench hull centered at the origin.

Another method for quantifying the grasp quality is to compute the volume of the grasp wrench hull $\tilde{\mathcal{W}}$. This approach provides more of an average-case metric rather than a worst-case metric, and can help differentiate between different grasp spaces that have the same worst-case metric. For example Figure 26.13 shows a grasp with the same worst-case metric as the grasp in Figure 26.12, but which would be considered worse with respect to the volumetric (average-case) metric.

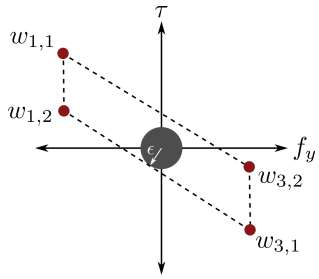


Figure 26.13: Using the volume of $\tilde{\mathcal{W}}$ as a metric for grasp quality can help differentiate between two grasps with the same worst-case performance. For example this grasp would be considered less robust than the grasp in Figure 26.12 since it has smaller volume.

26.3 *Grasp Force Optimization*

Recall from Section 26.1 that for a particular contact model a grasp map matrix G can be defined such that:

$$\boldsymbol{w} = G \begin{bmatrix} f_1 \\ \vdots \\ f_k \end{bmatrix}, \tag{26.4}$$

where $\boldsymbol{f}_i$ is the force vector associated with contact point i and $\boldsymbol{w}$ is the total wrench applied to the object. Additionally, recall that the matrix G also includes a rotational transformation such that the force vectors $\boldsymbol{f}_i$ are in local contact reference frames but $\boldsymbol{w}$ is represented in a common frame fixed in the object body (at the center of mass).

The next logical question to ask is how to compute force vectors $\{\boldsymbol{f}_i\}_{i=1}^k$ to achieve a desired wrench $\boldsymbol{w}$[16]. While one naive approach would be to just solve (26.4) using a least-squares method, this would fail to account for any constraints on the force vectors. In particular, this section will focus on the point contact with friction model where each force vector $\boldsymbol{f}_i = [f_x^{(i)}, f_y^{(i)}, f_z^{(i)}]^\top$ must satisfy the friction cone constraint:

$$\sqrt{{f_x^{(i)}}^2 + {f_y^{(i)}}^2} \leq \mu_{s,i} f_z^{(i)}, \quad f_z^{(i)} \geq 0,$$

where $\mu_{s,i}$ is the static friction coefficient for contact i. In this section the compact notation:

$$\boldsymbol{f}_i \in \mathcal{F}_i, \quad \mathcal{F}_i \coloneqq \{\boldsymbol{f} \in \mathbb{R}^3 \mid \sqrt{f_x^2 + f_y^2} \leq \mu_{s,i} f_z, \quad f_z \geq 0\},$$

for the friction cone constraint will again be used. It might also be desirable to include additional constraints on the force vectors, for example to account for hardware limitations (e.g. torque limits) or kinematic constraints. These additional constraints will be generally referred to by a convex constraint set C, such that $\boldsymbol{f}_i \in C$ is required for all $i = 1, \ldots, k$.

To summarize, the problem is to find a set of force vectors $\{\boldsymbol{f}_i\}_{i=1}^k$ such that (26.4) is satisfied[17] and such that $\boldsymbol{f}_i \in \mathcal{F}_i$ and $\boldsymbol{f}_i \in C$ for $i = 1, \ldots, k$. This problem can then be solved by formulating it as a convex optimization problem[18,19]:

$$\begin{aligned} \underset{\boldsymbol{f}_i,\ i \in \{1,\ldots,k\}}{\text{minimize}} \ & J(\boldsymbol{f}_1, \ldots, \boldsymbol{f}_k), \\ \text{s.t.} \ & \boldsymbol{f}_i \in \mathcal{F}_i, \quad i = 1, \ldots, k, \\ & \boldsymbol{f}_i \in C, \quad i = 1, \ldots, k, \\ & \boldsymbol{w} - G \begin{bmatrix} \boldsymbol{f}_1 & \ldots & \boldsymbol{f}_k \end{bmatrix}^\top = 0, \end{aligned} \tag{26.5}$$

where $J(\cdot)$ is the objective function that is convex in each $\boldsymbol{f}_i$. Simply choosing $J = 0$ would result in a convex feasibility problem, but a more common choice is:

$$J(\boldsymbol{f}_1, \ldots, \boldsymbol{f}_k) = \max\{\|\boldsymbol{f}_1\|, \ldots, \|\boldsymbol{f}_k\|\},$$

which is the maximum applied force magnitude among all contact points.

Note that the fundamental disadvantage of this optimization-based approach is that the positions of all contacts with respect to the object's center of mass and the object's friction coefficients are assumed to be known. It is also assumed that the desired wrench $\boldsymbol{w}$ is known!

[16] The desired wrench may be used to counter an external disturbance (to maintain equilibrium) or to *manipulate* the object.

[17] In the case that the desired wrench is used to counter an external disturbance, the condition (26.4) is referred to as the equilibrium constraint.

[18] S. Boyd and B. Wegbreit. "Fast Computation of Optimal Contact Forces". In: *IEEE Transactions on Robotics* 23.6 (2007), pp. 1117–1132

[19] The problem is technically a second-order cone program because of the friction cone constraints.

26.4 Learning-Based Approaches to Grasping

Model-based methods for grasp evaluation and optimization require several assumptions that may be either difficult to validate in practice, or may not even be valid in all scenarios. These assumptions include:

1. A coulomb (static) friction model defines the friction cone, and the coefficient μ_s is known.

2. The object's geometry and mass distribution is known[20], such that given a contact point the vector $\boldsymbol{d}_i$ from the object's center of mass to the contact is known.

3. The object is a rigid body.

4. The desired forces $\boldsymbol{f}_i$ can be applied perfectly.

[20] One option would be to build a database of known objects, but this may not be scalable to real world problems.

Learning-based methods for grasp analysis[21] can leverage data to decrease reliance on these assumptions, for example by not requiring explicit knowledge of the object's physical parameters. Learning-based methods can also combine the task of grasping with other parts of the manipulation pipeline, such as perception and motion planning.

This section will introduce some recent learning-based approaches to robotic grasping, which is still a very active area of research. Specifically, these examples will demonstrate several learning-based strategies including approaches that create synthetic training data from model-based simulators and approaches that use real hardware to generate data.

[21] J. Bohg et al. "Data-Driven Grasp Synthesis—A Survey". In: *IEEE Transactions on Robotics* 30.2 (2014), pp. 289–309

26.4.1 Choosing a Grasp Point from an RGB Image[22]

The objective of this supervised learning approach was to learn how to find a good grasp point in an RGB image of an object, and then generate a prediction of the point's 3D position. Since supervised learning techniques can require a lot of training data, this approach auto-generated training images *synthetically* using realistic rendering techniques (see Figure 26.14). The use of synthetic data also made it easier to collect a diverse training set including images with different lighting, object color, and object orientation and size.

Once a model was trained to produce good grasp point classifications, 3D predictions of the target grasp position were generated by *structure-from-motion*, where two images were used to triangulate the point in space. While there are certainly limitations to this approach, this work produced promising results, including good grasp success rates on novel objects (that weren't included in the training dataset). This work also had substantial influence on future learning-based grasping and manipulation approaches.

[22] A. Saxena, J. Driemeyer, and A. Ng. "Robotic Grasping of Novel Objects using Vision". In: *The International Journal of Robotics Research* 27.2 (2008), pp. 157–173

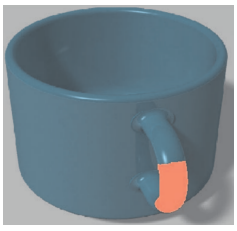


Figure 26.14: Synthetic image of a cup and its labeled grasp point from Saxena et al. (2008).

26.4.2 *Exploiting Simulation and a Database of Objects to Predict Grasp Success*[23]

The Dex-Net approach for learning to grasp is another supervised learning approach that relies on simulations to generate training data. Specifically this approach assumes a parallel jaw gripper and contains a large ($> 10,000$) database of 3D object models. For each model in the Dex-Net database, the simulator uses analytical (model-based) techniques to evaluate a large number of potential grasps for probability of success[24]. This is accomplished by empirically sampling a grasp some number of times and determining (through simulation) the percentage that result in force closure[25]. The database of objects and potential grasps can then be used to train a model to predict the probability of force closure for a new grasp/object.

In practice a number of candidate grasps are generated for a given (potentially novel) object, are evaluated by the learned model to predict probability of success, and then a *multi-armed bandit*[26] approach is used to select which grasp to take. The learned models are then updated based on the outcome of the action for continuous improvement. This work showed that leveraging the prior information from the object database can significantly improve grasping for new objects (even if they are not in the database), and later improvements have enhanced the approach even further.

26.4.3 *Learning to Grasp Through Real-World Trial-and-Error*[27]

Instead of leveraging simulators to generate synthetic data this work uses hardware experiments to generate real-world data. The resulting experiences are then used in a self-supervised approach to learn an *end-to-end* framework to grasp objects in cluttered environments. One of the reasons this work is significant is the lack of assumptions that are made: 3D object models are not needed, only RGB images are required, it does not use contact models or simulated data, no physical object information is used, and no hand-engineered path/grasp planning algorithms are used. Instead the system just learns through trial-and-error, exploring approaches to actuate the robot arm and gripper that eventually lead to robust grasps.

This approach showed impressive results over hand-designed or open-loop approaches, but at the cost that it took six months and a large number of robots to generate enough training data.

26.5 *Learning-Based Approaches to Manipulation*

The previous sections of this chapter have focused on the problem of grasping, but many robotic manipulation tasks involve more than simply grasping an object. For example it is possible to manipulate objects *without* force closure grasps, such as by pushing the object[28]. Many manipulation tasks that do involve grasping also involve other complex steps, such as using the grasped object to manipulate other objects (e.g. hitting a hammer with a nail) or placing

[23] J. Mahler et al. "Dex-Net 1.0: A cloud-based network of 3D objects for robust grasp planning using a Multi-Armed Bandit model with correlated rewards". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1957–1964

[24] The Dex-Net database consists of 2.5 million tested grasps.

[25] There is simulated uncertainty in object and gripper pose, as well as the surface friction.

[26] A fundamental reinforcement learning problem focused on uncertain decision making.

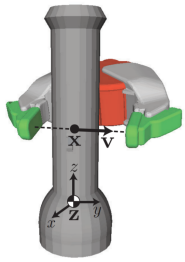


Figure 26.15: Dex-Net grasps are parameterized by the centroidal position of the gripper x and the approach direction v, Mahler et al. (2016).

[27] S. Levine et al. "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection". In: *The International Journal of Robotics Research* 37.4-5 (2018), pp. 421–436

[28] Not only is it possible, but may be necessary if the object is too large or heavy to grasp.

the object in a certain position (e.g. inserting a key into a lock). This section will introduce at a high-level some interesting and foundational problems in manipulation, and present the high-level ideas found in some recent research on learning-based approaches to solving them.

26.5.1 Planar Pushing

Planar pushing is a fundamental manipulation task where the goal is to control the pose of an object in a 2D setting by only using "pushing" contacts. While the contact point models used for grasping can also be applied in this problem, the interaction of the object and the surface must also be accounted for.

Similar to the physics-based contact models for grasping, physics-based models can also be developed to predict the sliding interactions between the object and the surface. In particular, the concept of a *friction limit surface*[29] can be used to model the interaction between the object and the surface. The friction limit surface is a boundary in wrench space that separates wrenches that the surface can apply to the object through friction and those it can't. The part of the wrench space enclosed by this surface will contain the origin (i.e. the zero wrench), and most importantly whenever the object is *slipping* the wrench applied on the object lies *on the friction limit surface*. This surface can be determined numerically if the coefficient of friction, the contact area, and the pressure distribution are known. For simplicity, it is common to approximate this surface as an ellipsoid. To summarize:

[29] I. Kao, K. Lynch, and J. Burdick. "Contact Modeling and Manipulation". In: *Springer Handbook of Robotics*. Springer, 2016, pp. 931–951

1. If an external wrench applied to the object is within the region of the wrench space enclosed by the friction limit surface, friction between the object and the surface will cause the object to remain motionless.

2. If the part slides quasistatically[30], the pushing wrench must lie on the friction limit surface and the motion (velocity) of the object can be determined.

[30] Assumption that the part moves slowly enough that inertial effects are negligible.

The friction limit surface provides the foundation for a physics-based model that predicts how an object will slide across a surface under external contact forces. Such a model could be used to design a controller (e.g. with model predictive control) for planar pushing tasks. However, these physics-based models are based on approximations and assumptions that may impact their accuracy or applicability to real problems. In fact some studies have been performed to evaluate the accuracy of physics-based pushing models[31].

Assumptions in physics-based pushing model: ellipsoidal friction limit surface, coulomb friction, perfectly planar object/surface, rigid body object, physical properties of object are known.

While physics-based controllers such as model predictive control can handle some uncertainty via feedback control mechanisms, it is still desirable to improve the modeling accuracy and eliminate assumptions requiring knowledge of the parameters that define the models[32]. Learning-based approaches are one possible solution to some of these challenges, where real-world data can be used to either completely replace or augment the physics-based models.

In fact, recent work[33] has compared the use of physics-based, hybrid (physics + learning), and learning-based models for planar pushing tasks. In this work

[31] K. Yu et al. "More than a million ways to be pushed. A high-fidelity experimental dataset of planar pushing". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 30–37

[32] For example the physical properties of the object and surface.

[33] A. Kloss, S. Schaal, and J. Bohg. "Combining learned and analytical models for predicting action effects from sensory data". In: *The International Journal of Robotics Research* (2020)

the hybrid model learned a mapping from sensor measurements (RGB images) into a set of parameters that were required for the physics-based motion model and the learning-based model just directly learned a single neural network for mapping sensor measurements to motion predictions directly. As might be expected the hybrid approach achieved better generalization by leveraging the physics-based model's structure, while the learning-based approach overfit to the training data[34].

[34] This is a classic example of bias-variance tradeoff in modeling.

26.5.2 Contact-Rich Manipulation Tasks

Many 6D manipulation problems involve grasping an object and then using it to physically interact with the environment. Classic everyday examples include hitting a nail with a hammer, inserting a key into a lock, and plugging a cord into an electrical outlet. These types of *contact-rich* tasks tasks typically rely on multiple different sensing modalities including haptic and visual feedback. Consider the task of inserting a key into a lock: without sight it would be challenging to correctly position the key and without tactile sensing (e.g. force/torque sensing) it would be challenging to know when the key is perfectly aligned and can be inserted.

However, it can be quite challenging to *integrate* multiple sensing modalities toward a common task, especially when the sensing modalities are so different and since manipulation tasks can be quite complex. One approach may be to individually develop systems for different subtasks and manually find a common interface to stitch them together, however this could be challenging from a system engineering perspective. An alternative is to use machine learning techniques to automatically integrate the sensing modalities.

One learning-based approach to this problem is to design an end-to-end system that takes as input all sensor data streams and outputs actions for the robot to execute the task. However, when implemented in a naive way (e.g. a single massive neural network architecture) end-to-end approaches can be data inefficient. An alternative is to add additional structure to the learning-based approach by leveraging some insights into the problem, similar to how the physics-based motion model was used in the learning-based planar pushing example discussed in the previous section.

A structured approach for manipulation tasks relying on multiple sensing modalities is introduced by Lee et al.[35]. In this work an end-to-end system that takes sensor data streams as input and outputs robot actions is split into two parts: first transforming the multi-modal sensor data streams into a low-dimensional feature representation that contains task relevant information[36], and then using these features as the input to a learned policy that generates robot actions. In other words, the insight is that the learning process can be made more efficient by first learning a way to compress and summarize all of the sensor data, and then learning how to use the summarized information to generate a good policy. Another benefit to this approach is that the sensor

[35] M. Lee et al. "Making Sense of Vision and Touch: Self-Supervised Learning of Multimodal Representations for Contact-Rich Tasks". In: *International Conference on Robotics and Automation (ICRA)*. 2019, pp. 8943–8950

[36] This is accomplished by training an autoencoder network.

data *encoder* can generalize more effectively to new tasks, meaning that only the policy portion needs to be retrained!

Part VII

Appendices

A
Machine Learning

Many algorithms and tools in robotic autonomy leverage models of the world that are often based on first-principles: physics-based kinematic models are used to design controllers, sensor models are used in localization algorithms, and geometric principles are used in understanding stereo vision. However, there are also many scenarios in robotics where these techniques may fail to capture the complexity of unstructured real-world environments. For example, how can a stop-sign be reliably detected in camera images when it could be rainy, foggy, or dark out, or when the stop-sign is partially occluded? Are there first-principles models that can accurately predict the behavior of a human driver, and distinguish between aggressive and defensive driving behavior? How can a robot be programmed to pick up objects with an infinite number of variations in size, shape, color, and texture? In the last few decades, advancements in *machine learning*[1] have led to start-of-the-art approaches for many of these challenging problems[2]. This chapter presents an introduction to machine learning to provide a knowledge of the fundamental tools that are used in learning-based algorithms for robotics, including computer vision, reinforcement learning, and more.

[1] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer, 2017

[2] Of course in many settings it is beneficial to use first-principles and machine learning techniques *in concert*.

Machine Learning

At their most fundamental level, machine learning techniques seek to extract useful patterns from *data*[3], and are typically classified as either *supervised* or *unsupervised*.

[3] In many cases the data will come from real world experiments, but in other cases may come from simulation.

Definition A.0.1 (Supervised Learning)**.** *Given a collection of n data points:*

$$\{(x_1, y_1), \ldots, (x_n, y_n)\},$$

where x_i is an input variable and y_i is an output, the supervised learning problem is to find a function $y = f(x)$ that fits the data and can be used to predict outputs y for new inputs x.

Definition A.0.2 (Unsupervised Learning)**.** *Given a collection of n data points $\{x_1, \ldots, x_n\}$, the unsupervised learning problem is to find patterns in the data.*

Supervised learning problems, such as regression and classification[4], are generally more common in robotics applications and will be the focus of this chapter. For example, robotic imitation learning-based controllers[5] can be expressed as a regression problem where the input x is the state of the robot and y is the action the robot should take. Classification problems also arise frequently in robotic computer vision, for example to identify whether the image x belongs to a particular class y (e.g. a dog or cat).

In both regression and classification problems, the learned function f is categorized as either *parametric* or *non-parametric*. Parametric functions are generally more structured and can be written down in an analytical form[6], while non-parametric functions are generally defined by the data points themselves[7]. The best choice between parametric or non-parametric functions is generally dependent on the particular problem and the type of data available. However, some of the most popular choices are parametric, such as polynomials and *neural networks*.

[4] In regression problems the output y is continuous and in classification problems the output y is discrete (categorical).

[5] Imitation learning refers to the process of learning to mimic a policy (e.g. from an expert) through example decisions.

[6] The most basic parametric function would be a linear function $f(x) = Wx$, parameterized by the "weight" matrix W.

[7] In the non-parametric k-nearest neighbors method, the value $f(x)$ is defined by the value of the data points y_i corresponding to the k closest points, x_i, to x.

A.1 Loss Functions

In supervised learning problems, a metric known as a *loss function* is used to evaluate and compare candidate models $f(x)$ that could be used to fit the data. Many loss functions for supervised learning problems exist, but some of the most common examples include the l_2 and l_1 loss (for regression) and the $0-1$ and cross entropy loss (for classification).

1. The l_2 loss function is defined by:

$$L = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2, \tag{A.1}$$

   where the summation is over some set of data points (x^i, y^i). From this loss function it can be seen that a penalty arises from the function f not perfectly matching the data at the sampled data points, but most importantly that the penalty is *quadratic* with respect to this residual. This loss function will therefore favor more small residuals over a few large residuals, which tends to make the model perform better "on average". However this also makes the l_2 loss sensitive to outliers in the data, making the training less robust.

2. The l_1 loss function is defined by:

$$L = \frac{1}{n}\sum_{i=1}^{n}|f(x_i) - y_i|. \tag{A.2}$$

   Unlike the l_2 loss, this loss function only penalizes the *absolute value* of the residual. Therefore this loss function will favor all residuals on a more equal footing and generally leads to a more robust training procedure that is less sensitive to outliers in the data.

3. The $0-1$ loss function is defined by:

$$L = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\{f(x_i) \neq y_i\}, \tag{A.3}$$

where $\mathbf{1}\{\cdot\}$ is the indicator function. This loss function can be used in classification problems and provides a loss of 1 whenever the classification is incorrect, and 0 otherwise. However, the use of this loss function introduces practical issues when training with gradient-based optimization, since this function is either flat or not differentiable at all points in the domain.

4. The cross entropy loss function[8] is defined by:

$$L = -\frac{1}{n}\sum_{i=1}^{n} y_i^\top \log f(x_i), \tag{A.4}$$

and is a common loss function in classification problems. To get an intuitive feeling for how the cross entropy loss works, consider a classification problem where the classes are $c \in \{1, 2, \ldots, C\}$ and where the function $f(x_i)$ outputs a *vector* of the probabilities of each class (which is normalized to sum to 1)[9]. Additionally, for each data point the vector y_i is a "one-hot" vector[10] specified by the class associated with x_i. Therefore the loss for a particular data point can be written as:

$$-y_i^\top \log f(x_i) = -\begin{bmatrix}0, \ldots, 1, \ldots, 0\end{bmatrix}\begin{bmatrix}\log f_1(x_i) \\ \vdots \\ \log f_C(x_i)\end{bmatrix} = -\log f_c(x_i),$$

where C is the number of classes, the 1 element in y_i is in the position corresponding to the correct class, and $f_c(x_i)$ is the probability of the correct class output by the model. Thus, to minimize the loss for this particular data point it is good to make $f_c(x_i) = 1$ (in fact as $f_c(x_i) \to 0$ the loss approaches infinity!). Cross entropy loss can also be derived from a statistical perspective, where it can be shown to be the same as maximizing the log-likelihood over all data points.

[8] Cross entropy loss is more practical than $0-1$ loss since it is a differentiable function.

[9] This can be accomplished by using the *softmax* function.

[10] A one-hot vector is a vector with all zeros and a single 1.

A.2 Model Training

In supervised learning problems with a predetermined parametric model (e.g. linear model or neural network), the *values* of the parameters can be optimized to best fit the data (i.e. minimize the specified loss function). This process of parameter optimization is referred to as *model training*. While in some special cases the optimal set of parameters can be computed analytically, it is more common to search for a good set of parameters in an iterative fashion using numerical optimization techniques.

Example A.2.1 (Linear Least Squares)**.** One of the most fundamental regression problems, linear least squares, can be solved analytically. In this problem, the parametric model is a linear model[11]:

$$f(x) = \theta^\top x,$$

where $x \in \mathbb{R}^p$ is the input and $\theta \in R^p$ is the set of model parameters, and the loss function is the l_2 loss (A.1). Given n data points (x_i, y_i), the loss function can be expressed in matrix form as:

$$L(\theta) = \frac{1}{n}\|Y - X\theta\|_2^2,$$

where the matrix $Y \in \mathbb{R}^n$ and $X \in \mathbb{R}^{n\times p}$ are defined by the data as:

$$Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}, \quad X = \begin{bmatrix} x_1^\top \\ \vdots \\ x_n^\top \end{bmatrix}.$$

The parameters θ are then chosen to minimize the loss function by taking the derivative:

$$\frac{dL}{d\theta} = \frac{2}{n}X^\top X\theta - \frac{2}{n}X^\top Y,$$

and setting it equal to zero, which gives $\theta^* = (X^\top X)^{-1}X^\top Y$.[12]

[11] This approach can also be extended to nonlinear settings through the use of basis functions. In particular the model becomes $f(x) = \theta^\top \phi(x)$, where $\phi(x)$ are nonlinear basis functions (sometimes referred to as *features*.

[12] Note that directly computing the inverse of $X^\top X$ may be challenging, but alternative numerical methods exist to compute the value of θ^* that satisfies the necessary condition of optimality.

A.2.1 Numerical Optimization

In many cases parameter optimization cannot be performed analytically and therefore numerical optimization algorithms are used. Two of the most fundamental algorithms for numerical optimization-based training of parametric models are *gradient descent* and *stochastic gradient descent*[13].

In gradient descent, the parameters $\theta \in \mathbb{R}^p$ of a model $f_\theta(x)$ are iteratively updated by:

$$\theta \leftarrow \theta - \eta\nabla_\theta L(\theta),$$

where $\nabla_\theta L(\theta)$ is the gradient of the loss function with respect to the parameters and the hyperparameter η is referred to as the *learning rate* or *step-size*. By leveraging the gradient, this update rule seeks to iteratively improve the parameters to incrementally decrease the loss.

Notice that the gradient of the loss can be written as:

$$\nabla_\theta L(\theta) = \frac{1}{n}\sum_{i=1}^{n}\nabla_\theta L_i(\theta),$$

where L_i is the term of the loss function associated with the i-th data point. Therefore computing the gradient of the loss function could be computationally intensive if the number of data points is very large. To address this issue, stochastic gradient descent uses an approximation of the gradient computed by randomly sampling the gradients over a smaller *batch* of data points S[14]:

[13] Gradient descent is referred to as a *first-order* method.

[14] The batch S is resampled at every iteration of the algorithm.

$$\nabla_\theta L(\theta) \approx \frac{1}{|S|} \sum_{i \in S \subset \{1,\dots,n\}} \nabla_\theta L_i(\theta),$$

where $|S|$ is the number of data points in the batch.

Beyond gradient descent approaches lie a broad set of additional numerical optimization algorithms that are commonly used in practice[15]. Often times these advanced methods may lead to faster learning rates or more robust learning, and some algorithms may also be more applicable to problems with larger amounts of data or larger numbers of model parameters.

[15] M. J. Kochenderfer and T. A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019

A.2.2 Training and Test Sets + Regularization

In supervised learning with parametric models, the goal is to train a model $f(x)$ that accurately predicts the output y for inputs x that are not seen in the data set. In other words, the goal is to find a model that *generalizes* to unseen data. It is important to note however that simply optimizing the loss function over a dataset *does not* guarantee that the model generalizes well, since it is possible to *overfit* the model to the data.

A model is overfit to a set of data if it predicts the set of data well (i.e. has a low loss) but fails to accurately predict new data. To counter this issue, one very common practice in machine learning is to split the full dataset into two parts: a training set and a test set[16]. As the names suggest, the model can be trained with the training data and then the test set can be used to verify whether overfitting has occured. To test for overfitting, the loss function can be evaluated over both sets of data. Overfitting has occured if the training loss is significantly lower than the test loss.

[16] There isn't an optimal way to split the data, but common splits range from 80/20 training/test to 50/50 training/test.

While splitting the data into training and test sets provides a good way to *verify* whether the learned model generalizes well, there are also techniques that can be employed in during the training process to avoid overfitting. In particular, the most common technique is known as *regularization*. One form of regularization is implemented by adding terms to the loss function to penalize "model complexity". For example, with a model $f_\theta(x)$ parameterized by the vector θ, two common forms of regularization include:

1. l_2 regularization, which consists of the addition of the term $\|\theta\|_2$ to the loss function,

2. l_1 regularization, which which consists of the addition of the term $\|\theta\|_1$ to the loss function.

A.3 Neural Networks

One very common parametric model used in machine learning is the *neural network*[17]. Neural networks are models with very specific structures, consisting of a hierarchical sequence of linear and nonlinear functions, which makes them

[17] Also known as the multi-layer perceptron.

very powerful function approximators. Mathematically, neural networks are typically described as a sequence of functions:

$$\begin{aligned} h_1 &= f_1(W_1 x + b_1), \\ h_2 &= f_2(W_2 h_1 + b_2), \\ &\vdots \\ \hat{y} &= f_K(W_K h_{K-1} + b_K), \end{aligned} \tag{A.5}$$

which is an easier notation than writing the equivalent composite function:

$$\hat{y} = f_K(W_K f_{K-1}(\dots) + b_K).$$

In this model, the parameters are the weights $W_1, \dots, W_K$ and biases $b_1, \dots, b_K$, and the structure of the model is predefined by the choice of the *activation functions* $f_1, \dots, f_K$ and the number of *layers* K. The intermediate variables $h_1, \dots, h_{K-1}$ are the outputs of the *hidden layers*, aptly named since they are not the input or the output of the model.

To fully specify the structure of the model, a practitioner needs to specify the number of hidden layers[18], the dimensionality of each of the intermediate variables h_i (usually chosen to be the same for all hidden layers), and the activation functions f_i.

[18] Neural networks with many layers are referred to as *deep neural networks*.

A.3.1 Activation Functions

Commonly used activation functions $f_1, \dots, f_K$ in neural networks include sigmoid functions, hyperbolic tangent functions, rectified linear units (ReLU), and leaky ReLU functions[19].

[19] It is typical for the same activation function to be used for all layers of the network.

1. Sigmoid function (also denoted as $\sigma(x)$):

$$f(x) = \frac{1}{(1 + e^{-x})},$$

2. Hyperbolic tangent function:

$$f(x) = \tanh(x),$$

3. ReLU function:

$$f(x) = \max\{0, x\},$$

4. Leaky ReLU function:

$$f(x) = \max\{0.1x, x\},$$

It is important to note that each of these activation functions share two important characteristics: they are *nonlinear* and they are easy to differentiate. It is critical that the activation function be nonlinear since a composition of linear functions will remain linear, and therefore no additional benefit is gained in modeling capability by adding more than a single layer to the network. Differentiability is also critical because the gradients must be easily computable during training[20].

[20] While ReLU and leaky ReLU are not strictly differentiable, this issue is easily mitigated in practice.

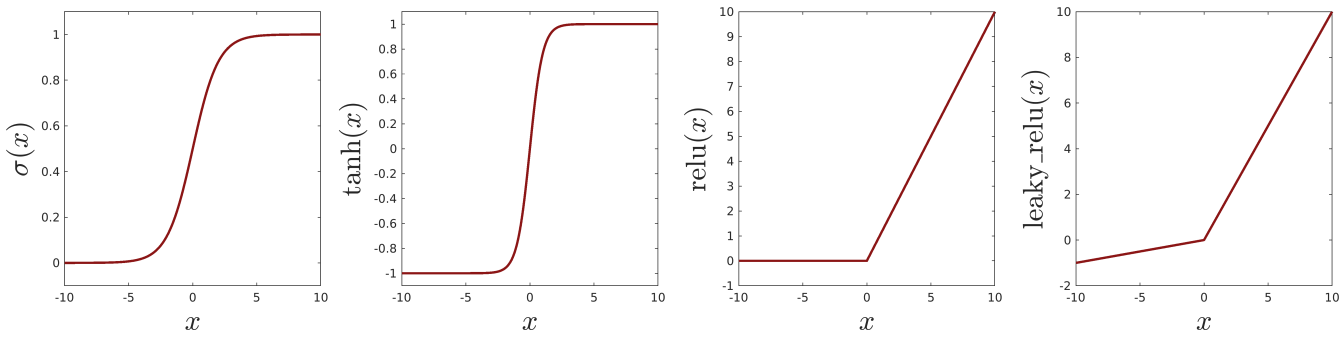


Figure A.1: Common activation functions used in neural networks.

A.3.2 *Training Neural Networks*

Neural networks are trained with gradient-based numerical optimization techniques, such as those mentioned in Section A.2.1 (e.g. stochastic gradient descent). Therefore once a particular loss function L has been chosen, the gradients $\frac{\partial L}{\partial \theta}$ must be computed for each parameter. Since neural networks can contain a large number of parameters, this gradient computation must be accomplished in a computationally efficient way. In particular, the gradients are computed using an algorithm referred to as *backpropagation*, which leverages the chain rule of differentiation and the layered structure of the network.

As with other parametric models, it is very important to avoid overfitting when training neural networks[21]. This can partially be accomplished using the division of the dataset into training and test sets, as well as by using regularization techniques as mentioned in Section A.2.2. Another technique for avoiding overfitting in neural networks is referred to as *dropout*, where some "connections" in the network are occasionally removed during the training process. This essentially forces the network to learn more redundant representations, which has been shown to improve generalization. Of course another useful technique to avoid overfitting is just to have an extremely large dataset, but in many cases this may not be very practical.

[21] It is quite easy to overfit when training neural networks since they have such a large number of parameters.

A.4 *Backpropagation and Computational Graphs*

From a theoretical standpoint, computing the gradients $\frac{dL}{d\theta}$ of the loss function with respect to the parameters is relatively straightforward. However, from a practical standpoint computing these gradients can be computationally expensive, especially for complex models such as neural networks. *Backpropagation*[22] is an algorithm that addresses this issue by computing all required gradients in an efficient way.

Backpropagation computes gradients by cleverly choosing the order in which operations required to compute the gradient are performed. By doing so it seeks to avoid redundant computations, and can in fact be viewed as an example of dynamic programming. While in some simple cases the backpropagation

[22] Sometimes also referred to as auto-differentiation.

Many software tools, such as PyTorch (`https://pytorch.org/`) and TensorFlow (`https://www.tensorflow.org/`) will automatically be able to perform backpropagation for a large class of functions.

algorithm may provide only a small advantage, in many cases (and in particular for neural network training) backprop can be orders of magnitude faster than naive approaches.

A *computational graph* is another practical tool that is useful when using the backpropagation algorithm to compute gradients. A computational graph provides a way to express a mathematical function using representations from graph theory. In particular the function is expressed as a directed graph where the nodes represent mathematical operations or function inputs and the edges represent intermediate quantities. Using a computational graph, a forward pass through the graph (starting at the root nodes, which are function inputs) is equivalent to evaluating the function.

This representation makes it very easy to see the structure of the mathematical operation that can be exploited by the backpropagation algorithm. As an example, consider the function $L(x,y) = g(f(x,y))$ and its associated computational graph shown in Figure A.2 (which includes the intermediate variable z). Using the chain rule, the gradient of L with respect to x is $\frac{\partial L}{\partial x} = \frac{dL}{dz}\frac{\partial z}{\partial x}$. The backpropagation algorithm uses this structure to convert the computation of the gradient $\frac{\partial L}{\partial x}$ into a sequence of *local* gradient computations $\frac{dL}{dz}$ and $\frac{\partial z}{\partial x}$, corresponding to each computation node in the graph. With this structure redundant computation can be be avoided. For example, when computing $\frac{\partial L}{\partial y}$ the partial gradient $\frac{dL}{dz}$ can be reused.

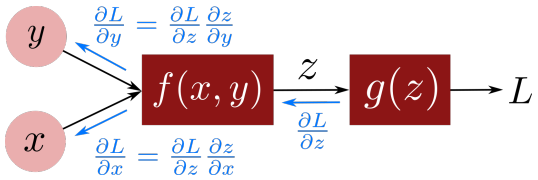


Figure A.2: Example computational graph for a function $L(x,y) = g(f(x,y))$.

To summarize, the backpropagation algorithm follows the following basic steps:

1. Perform a forward pass through the computational graph to compute any intermediate variables that may be needed for computing local gradients[23].
2. Starting from the graph output, perform a backwards pass over the graph where at each computation node the local gradient of the node with respect to its inputs and outputs is computed. Then, compute the gradient of the graph's output with respect to the inputs of the local computation node, leveraging the chain rule and previously calculated gradients. In Figure A.2, the first step of backprop would be to compute $\frac{\partial L}{\partial z} = \frac{dg}{dz}$, and the second step would use $\frac{\partial L}{\partial z}$ to compute the remaining gradients $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$.

[23] For example if $g(z) = z^2$ the gradient $\frac{dg}{dz} = 2z$ depends on the current value of z

Example A.4.1 (Training a Simple Model)**.** Consider a supervised learning problem with a parametric model defined as:

$$f(x) = (x + a)(x + b),$$

where a and b are parameters of the model, and a l_2 loss function (A.1) is used for training. A computational graph for computing the loss from a single data point with this model is shown in Figure A.3.

For this model and loss function the gradients required for training can be computed analytically as:

$$\frac{\partial L_i}{\partial a} = -2(y - (x+a)(x+b))(x+b),$$
$$\frac{\partial L_i}{\partial b} = -2(y - (x+a)(x+b))(x+a).$$

Computing the gradients in this way (the naive approach) would require 7 operations each (4 sums and 3 multiplications), for a total of 14 operations.

Alternatively the gradients can be computed in a more efficient way using backpropagation, which avoids redundant computations. This approach can be viewed as taking a *backward pass* over the computation graph. Starting at the output of the graph:

$$\frac{\partial L_i}{\partial z_1} = 2z_1.$$

Then moving on through the next operations and using the chain rule (and reusing the previous computations):

$$\frac{\partial L_i}{\partial \hat{y}} = \frac{\partial L_i}{\partial z_1}\frac{\partial z_1}{\partial \hat{y}} = -\frac{\partial L_i}{\partial z_1},$$

and:

$$\frac{\partial L_i}{\partial z_2} = \frac{\partial L_i}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z_2} = \frac{\partial L_i}{\partial \hat{y}}z_3,$$
$$\frac{\partial L_i}{\partial z_3} = \frac{\partial L_i}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z_3} = \frac{\partial L_i}{\partial \hat{y}}z_2.$$

Finally, the next step backward reaches the parameters a and b:

$$\frac{\partial L_i}{\partial a} = \frac{\partial L_i}{\partial z_2}\frac{\partial z_2}{\partial a} = \frac{\partial L_i}{\partial z_2},$$
$$\frac{\partial L_i}{\partial b} = \frac{\partial L_i}{\partial z_3}\frac{\partial z_3}{\partial b} = \frac{\partial L_i}{\partial z_3}.$$

To actually compute the numerical values of these gradients:

1. First perform a forward pass through the network to compute the values z_1, z_2, and z_3 (5 operations).
2. Then perform the backward pass computations to compute $\frac{\partial L_i}{\partial z_1}$, $\frac{\partial L_i}{\partial \hat{y}}$, $\frac{\partial L_i}{\partial z_2}$, $\frac{\partial L_i}{\partial z_3}$, $\frac{\partial L_i}{\partial a}$, and $\frac{\partial L_i}{\partial b}$ (4 operations).

Using backpropagation, only 9 operations are required to compute the gradients $\frac{\partial L_i}{\partial a}$, and $\frac{\partial L_i}{\partial b}$, which is a non-negligible reduction over the naive approach!

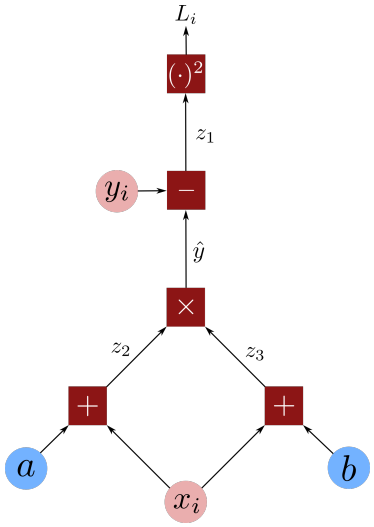


Figure A.3: Computational graph for computing the loss for a single data point for the model $f(x) = (x+a)(x+b)$ with l_2 loss (see Example A.4.1). The values (x_i, y_i) are the data point, the model output is $\hat{y}$, and z_1, z_2, z_3 are intermediate variables. The quantities a and b are parameters of the model.

Bibliography

[1] P. Abbeel and A. Ng. "Apprenticeship Learning via Inverse Reinforcement Learning". In: *Proceedings of the Twenty-First International Conference on Machine Learning*. 2004.

[2] M. Aicardi et al. "Closed loop steering of unicycle like vehicles via Lyapunov techniques". In: *IEEE Robotics & Automation Magazine* 2.1 (1995), pp. 27–35.

[3] U. Ascher and R. D. Russell. "Reformulation of boundary value problems into "standard" form". In: *SIAM Review* 23.2 (1981), pp. 238–254.

[4] A. Bajcsy et al. "Learning Robot Objectives from Physical Human Interaction". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 217–226.

[5] C. Basu et al. "Do You Want Your Autonomous Car to Drive Like You?" In: *12th ACM/IEEE International Conference on Human-Robot Interaction*. 2017, pp. 417–425.

[6] D. Bertsekas. *Reinforcement learning and optimal control*. Athena Scientific, 2019.

[7] J. Bohg et al. "Data-Driven Grasp Synthesis—A Survey". In: *IEEE Transactions on Robotics* 30.2 (2014), pp. 289–309.

[8] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge University Press, 2004.

[9] S. Boyd and B. Wegbreit. "Fast Computation of Optimal Contact Forces". In: *IEEE Transactions on Robotics* 23.6 (2007), pp. 1117–1132.

[10] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[11] R. Brooks. "A robust layered control system for a mobile robot". In: *IEEE Journal on Robotics and Automation* 2.1 (1986), pp. 14–23.

[12] E. M. Clarke et al. *Model Checking*. 2nd ed. MIT Press, 2018.

[13] G. Dudek and M. Jenkin. "Inertial Sensors, GPS, and Odometry". In: *Springer Handbook of Robotics*. Springer, 2008, pp. 477–490.

[14] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2011.

[15] A. Fusiello, E. Trucco, and A. Verri. "A compact algorithm for rectification of stereo pairs". In: *Machine Vision and Applications* 12.1 (2000), pp. 16–22.

[16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[17] F. Gustafsson. *Statistical Sensor Fusion*. Studentlitteratur, 2013, p. 554.

[18] C. Harris and M. Stephens. "A combined corner and edge detector". In: *4th Alvey Vision Conference*. 1988.

[19] R. Hartley and A. Zisserman. "Camera Models". In: *Multiple View Geometry in Computer Vision*. Academic Press, 2002.

[20] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer, 2017.

[21] L. Janson et al. "Fast Marching Tree: A Fast Marching Sampling-Based Method for Optimal Motion Planning in Many Dimensions". In: *Int. Journal of Robotics Research* 34.7 (2015), pp. 883–921.

[22] L. Joseph. *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*. Apress, 2018.

[23] L. Kaelbling et al. *6.01SC: Introduction to Electrical Engineering and Computer Science I*. MIT OpenCourseWare. 2011.

[24] I. Kao, K. Lynch, and J. Burdick. "Contact Modeling and Manipulation". In: *Springer Handbook of Robotics*. Springer, 2016, pp. 931–951.

[25] S. Karaman and E. Frazzoli. "Sampling-based Algorithms for Optimal Motion Planning". In: *Int. Journal of Robotics Research* 30.7 (2011), pp. 846–894.

[26] G. Katz et al. "The Marabou Framework for Verification and Analysis of Deep Neural Networks". In: *Computer Aided Verification*. 2019, pp. 443–452.

[27] L. E. Kavraki et al. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580.

[28] D. E. Kirk. *Optimal Control Theory: An Introduction*. Dover Publications, 2004.

[29] A. Kloss, S. Schaal, and J. Bohg. "Combining learned and analytical models for predicting action effects from sensory data". In: *The International Journal of Robotics Research* (2020).

[30] M. J. Kochenderfer and T. A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019.

[31] D. Kortenkamp, R. Simmons, and D. Brugali. "Robotic Systems Architectures and Programming". In: *Springer Handbook of Robotics*. Springer, 2008, pp. 283–302.

[32] M. Kwiatkowska, G. Normal, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. 2011, pp. 585–591.

[33] S. M. LaValle. *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006.

[34] S. M. LaValle. *Rapidly-Exploring Random Trees: A New Tool for Path Planning*. 1998.

[35] Y. LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[36] M. Lee et al. "Making Sense of Vision and Touch: Self-Supervised Learning of Multimodal Representations for Contact-Rich Tasks". In: *International Conference on Robotics and Automation (ICRA)*. 2019, pp. 8943–8950.

[37] J. Levine. *Analysis and Control of Nonlinear Systems: A Flatness-based Approach*. Springer, 2009.

[38] S. Levine et al. "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection". In: *The International Journal of Robotics Research* 37.4-5 (2018), pp. 421–436.

[39] C. Loop and Z. Zhang. "Computing rectifying homographies for stereo vision". In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Vol. 1. 1999, pp. 125–131.

[40] J. Mahler et al. "Dex-Net 1.0: A cloud-based network of 3D objects for robust grasp planning using a Multi-Armed Bandit model with correlated rewards". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1957–1964.

[41] H. P. Moravec. "Towards automatic visual obstacle avoidance". In: *5th International Joint Conference on Artificial Intelligence*. 1977.

[42] R. M. Murray. *Optimization-Based Control*. California Institute of Technology, 2009.

[43] A. Ng and S. Russell. "Algorithms for Inverse Reinforcement Learning". In: *Proceedings of the Seventeenth International Conference on Machine Learning*. 2000, pp. 663–670.

[44] N. Perveen, D. Kumar, and I. Bhardwaj. "An overview on template matching methodologies and its applications". In: *International Journal of Research in Computer and Communication Technology* 2.10 (2013), pp. 988–995.

[45] D. Prattichizzo and J. C. Trinkle. "Grasping". In: *Springer Handbook of Robotics*. Springer, 2016, pp. 955–988.

[46] M. Quigley, B. Gerkey, and W. D. Smart. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. O'Reilly Media, 2015.

[47] N. Ratliff, J. A. Bagnell, and M. Zinkevich. "Maximum Margin Planning". In: *Proceedings of the 23rd International Conference on Machine Learning*. 2006, pp. 729–736.

[48] S. Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.6 (2017), pp. 1137–1149.

[49] S. Ross, G. Gordon, and D. Bagnell. "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011, pp. 627–635.

[50] D. Sadigh et al. "Active Preference-Based Learning of Reward Functions". In: *Robotics: Science and System*. 2017.

[51] D. Sadigh et al. "Planning for cars that coordinate with people: leveraging effects on human actions for planning and active information gathering over human internal state". In: *Autonomous Robots* 42.7 (2018), pp. 1405–1426.

[52] A. Saxena, J. Driemeyer, and A. Ng. "Robotic Grasping of Novel Objects using Vision". In: *The International Journal of Robotics Research* 27.2 (2008), pp. 157–173.

[53] D. Scharstein and R. Szeliski. "High-accuracy stereo depth maps using structured light". In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Vol. 1. 2003.

[54] E. Schmerling, L. Janson, and M. Pavone. "Optimal sampling-based motion planning under differential constraints: the driftless case". In: *IEEE International Conference on Robotics and Automation*. 2015, pp. 2368–2375.

[55] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, 2011.

[56] D. Simon. *Optimal State Estimation: Kalman, H_∞, and Nonlinear Approaches*. John Wiley & Sons, 2006.

[57] J.-J. E. Slotine and W. Li. *Applied Nonlinear Control*. Pearson, 1991.

[58] R. Sutton and A. Barto. *Reinforcement learning: An introduction*. MIT Press, 2018.

[59] R. Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[60] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005.

[61] R. Tsai. "A Versatile Camera Calibration Technique for High-accuracy 3D Machine Vision Metrology Using Off-the-shelf TV Cameras and Lenses". In: *IEEE Journal on Robotics and Automation* 3.4 (1987), pp. 323–344.

[62] K. Yu et al. "More than a million ways to be pushed. A high-fidelity experimental dataset of planar pushing". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 30–37.

[63] M. D. Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks". In: *European Conference on Computer Vision (ECCV)*. Springer, 2014, pp. 818–833.

[64] Z. Zhang. "A Flexible New Technique for Camera Calibration". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22 (2000).

[65] B. D. Ziebart et al. "Maximum Entropy Inverse Reinforcement Learning". In: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*. 2008, pp. 1433–1438.