Daniele Gammelli, Joseph Lorenzetti,
Katie Luo, Gioele Zardini, Marco Pavone

Principles of Robot Autonomy

SEPTEMBER 23, 2025

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, we 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, we focus on providing a high-level understanding of the full *autonomy stack* to provide 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, we generally assume some prior knowledge of several topics. Specifically, familiarity with the basics of calculus, differential equations, linear algebra, probability and statistics, and computer 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* 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. We also reserve special acknowledgements for the course assistants who were instrumental in developing and refining the course material, and in particular Benoit Landry and Edward Schmerling, who were instrumental in developing the first iteration of the course. In large part, additional material for the course such as homework and lectures are also publicly available.

Contents

Part I

Robot Motion Planning and Control

1
Modeling Robot Dynamics

Robots can take on a wide variety of forms: they may have rigid or flexible structures, rely on different types of actuators for control, perceive their surroundings through diverse sensing modalities, or even exist as purely virtual agents. Despite this variety of embodiments, nearly all robotic systems share a fundamental characteristic—they are *dynamic* agents whose states evolve over time. The most immediate example of robot dynamics is physical motion, encompassing changes in position, velocity, joint configurations, and sensor orientations. A deep understanding of the dynamical properties of a robotic system is essential to its effective design and control. For instance, building a bipedal robot capable of walking or running requires detailed modeling of its motion to ensure that the mechanical structure and actuators can withstand the forces and torques involved. Accurate dynamic models are also critical for designing control strategies that achieve stable, efficient, and responsive locomotion.

This chapter introduces several foundational topics in modeling robotic systems. We will start with Section 1.1 by introducing the concept of a *state space model*, which provides a mathematical framework to describe the behavior of a robot's state over time. Next, in Section 1.2, we will detail how a robot's *kinematics and dynamics* are used to derive these state space models, focusing on the principles that govern physical motion and constraints. We will then explore specific motion models for wheeled robots in Section 1.3, including the *unicycle* and *differential drive* models, to illustrate practical applications of these concepts. Finally, in Section 1.4, we will discuss computational techniques for *simulating* robot dynamics, emphasizing numerical integration methods such as the Euler and Runge-Kutta methods to approximate and analyze the time evolution of robotic systems.

1.1 State Space Models

A state space model is a mathematical framework for describing the behavior of a dynamical system. Every state space model consists of two key components: a *state* and a *model*.

Definition 1.1.1 (State)**.** The *state* of a dynamical system at time t_0 is a minimal set of variables $x(t_0)$ such that, given the control input $u(t)$ for all $t \geq t_0$, the future evolution of the system's state $x(t)$ for all $t \geq t_0$ is uniquely determined, independently of the system's behavior for $t < t_0$.

Formally, the state is a sufficient statistic of the system's history: given the current state and the external inputs to the system, the system's future behavior is fully determined, independently of how that state was reached. The state of a robotic system can be finite or infinite-dimensional. For example, a simple mobile robot with a rigid body can be represented by a finite and low-dimensional state (e.g., the position and velocity of its center of mass), whereas a flexible robot might require an infinite-dimensional state to describe the continuous deformation of its body (e.g., a deflection function that describes the displacement at every point along the robot's length).

When modeling the dynamics of a robot, it is important to define the state in a way that aligns with the specific goals and requirements of the application. In practice, the complexity of the state representation can often be reduced by omitting parts of the robot's dynamics that are irrelevant to the problem at hand. For instance, in developing software to enable an autonomous car navigate urban environments, it may be sufficient to model only its position, orientation, and velocity, while abstracting away the internal mechanics of the engine, tires, or suspension. Conversely, if the car is being designed for high-performance racing, these specifics become critical to accurately capture and optimize its behavior. Throughout this book, we focus on applications where the state is finite-dimensional and represent it as a vector $\boldsymbol{x} \in \mathbb{R}^n$, referred to as the *state vector*.

The second key component of a state space model is the *model* itself, which defines the rules and equations governing the evolution of the state over time by relating it to a set of *inputs* and *outputs*.

The *inputs* to a model refer to external factors or control signals that influence the behavior of the system.[1] These can include forces, torques, voltages, commands, or any other external stimuli that drive the system's dynamics. As with the state, the dimensionality of the input may vary, but throughout this book we assume the input is represented by a finite-dimensional vector, $\boldsymbol{u} \in \mathbb{R}^m$.

[1] The term *input* is used interchangeably with *control* and *action* in different domains of robotics.

The *outputs* of a model are the observable variables or measurements derived from the system. These typically come from sensors or other measurement devices that capture data reflecting the system's state and behavior. For example, a robot equipped with a global navigation satellite system (GNSS) sensor may directly measure its position but not its heading. In some cases, the relationship between the state and the output is complex—for example, the connection between a robot's inertial pose and a red-green-blue-depth (RGB-D) camera image. The process of inferring the state from the outputs is referred to as *state estimation*, and it will be explored in detail in Chapters 13 - 17. We assume the output is finite-dimensional and denote it by the vector $\boldsymbol{y} \in \mathbb{R}^q$.

Definition 1.1.2 (Model)**.** A *model* describes the evolution of a dynamical system through two types of equations: *state equations* and *observation equations*. The state equations specify how the state evolves as a function of the current state and inputs, while the observation equations define how the state influences the measurable outputs.

State equations are typically modeled as differential equations[2] describing the changes in the state with respect to an independent variable, usually time:

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}(t), \boldsymbol{u}(t)), \tag{1.1}$$

where $\dot{\boldsymbol{x}} = \frac{d\boldsymbol{x}(t)}{dt}$ is the time derivative of the state vector $\boldsymbol{x}(t)$, and $f\colon \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ is the *dynamics* function of the system.

Observation equations take the form:

$$\boldsymbol{y}(t) = h(\boldsymbol{x}(t), \boldsymbol{u}(t)), \tag{1.2}$$

where $\boldsymbol{y}(t)$ denotes the output at time t, and $h\colon \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^q$ is a function that relates the state and inputs to the system's output.

Together, the definitions of the state $\boldsymbol{x}$, input $\boldsymbol{u}$, output $\boldsymbol{y}$, and Equations (1.1) and (1.2), form the *state space model*[3].

Throughout this book, we assume that time is the independent variable and simplify our notation by writing θ in place of $\theta(t)$ for time-dependent variables. Accordingly, we use $\dot{\theta}$ to represent time derivaties, $\ddot{\theta} = \frac{d^2\theta(t)}{dt^2}$ for second derivatives, and $\theta^{(m)} = \frac{d^m\theta(t)}{dt^m}$ for higher-order derivatives.

In summary, state space models provide a powerful formalism for modeling, analyzing, and controlling robotic systems. Mastering and effectively utilizing state space models is crucial for advancing robotic capabilities, enabling systems to autonomously navigate, interact, and adapt in dynamic and complex environments.

1.1.1 Types of State Space Models

State space models, as represented in Equation (1.1) and Equation (1.2), can be classified based on two key properties: linearity and time-invariance.

A model is said to be *time-invariant* if the functions f and h do not explicitly depend on time t; otherwise, it is *time-varying*. A model is said to be *linear*[4] if the functions f and h are linear functions of both the state $\boldsymbol{x}$ and control $\boldsymbol{u}$. More generally, a system is linear if it satisfies the *superposition principle*, which states that the response to a linear combination of inputs is the corresponding linear combination of the individual responses. Formally, if $x_1(t)$ and $x_2(t)$ are solutions corresponding to inputs $u_1(t)$ and $u_2(t)$, respectively, then for any scalars $\alpha, \beta \in \mathbb{R}$, the trajectory $\alpha x_1(t) + \beta x_2(t)$ is a solution corresponding to the input $\alpha u_1(t) + \beta u_2(t)$. If this property does not hold, the system is said to be *nonlinear*.

[2] Ordinary differential equations are most commonly used to model robotic systems. However, partial differential equations may be needed for more complex cases, such as systems with flexible or deformable structures, where dynamics vary over both space and time.

[3] In some contexts, it may be convenient to represent the evolution of a system using discrete-time difference equations. As we will discuss in Chapter 2, discrete-time models are particularly well-suited for systems that naturally evolve in discrete steps, or for approximating continuous-time systems within computational frameworks.

[4] Often referred to as a *linear system*.

Example 1.1.1 (Linear Time-Invariant Model)**.** Consider the system:

$$\dot{x} = x + u$$
$$y = x.$$

This model is linear and time-invariant because the functions describing the system depend linearly on $\boldsymbol{x}$ and $\boldsymbol{u}$ and there is no explicit time dependence.

Example 1.1.2 (Nonlinear Time-Varying Model)**.** Consider the system:

$$\dot{x} = tx + u$$
$$y = x^2.$$

This model is *nonlinear* due to the quadratic output term x^2, and it is also *time-varying* because the state equation explicitly depends on time through the term tx. Note, however, that the state equation $\dot{x} = tx + u$ is *linear*, as it satisfies the superposition principle with respect to x and u. Therefore, the nonlinearity in this system arises solely from the output equation, not from the state dynamics.

Linear Models and Their Standard Form. Linear models are particularly important due to their analytical tractability and widespread applicability. They are commonly expressed in a standard matrix form:

$$\begin{aligned} \dot{\boldsymbol{x}} &= A(t)\boldsymbol{x} + B(t)\boldsymbol{u}, \\ y &= C(t)\boldsymbol{x} + D(t)\boldsymbol{u}, \end{aligned} \tag{1.3}$$

where the matrices $A(t), B(t), C(t), D(t)$ are of appropriate dimensions and define the linear relationships for the state dynamics and the output. We refer to the matrix $A(t)$ as the *dynamics* matrix, $B(t)$ as the *control* matrix, $C(t)$ as the *output* or *sensor* matrix, and $D(t)$ as the *direct* or *feed-forward* matrix[5].

Despite the inherent nonlinear nature of many real-world systems, linear models are often employed due to their simplicity and ease of manipulation. A common practice is to approximate nonlinear systems with linear models around specific operating points using a technique known as *linearization*. As we will discuss in Chapter 3, this approach simplifies analysis and control design, making linear models a fundamental tool in engineering and control theory.

[5] If the matrices $A(t), B(t), C(t)$, and $D(t)$ are time-invariant, we have a Linear Time-Invariant (LTI) system, which is a cornerstone of control theory.

1.1.2 *Converting Higher-Order Models Into State Space Form*

The state space model, as described by Equation (1.1), is represented by first-order differential equations. However, many robotics applications involve higher-order differential equation models, such as those governing the dynamics of robotic arms or wheeled robots. To analyze and control these systems in a unified framework, we can convert higher-order models into first-order state space form by introducing additional state variables.

Consider a linear n-th order differential equation:

$$\theta^{(n)} + a_{n-1}\theta^{(n-1)} + \ldots + a_1\dot{\theta} + a_0\theta = u, \tag{1.4}$$

where $a_i \in \mathbb{R}$ are constants and θ is the state variable. To convert this differential equation into state space form, i.e., a set of first-order differential equations, we can define an n-dimensional state vector:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \coloneqq \begin{bmatrix} \theta^{(n-1)} \\ \theta^{(n-2)} \\ \vdots \\ \theta \end{bmatrix}.$$

Given that $\dot{x}_1 = \theta^{(n)}$, $\dot{x}_2 = x_1$, $\dot{x}_3 = x_2$ and so on, we can express the higher-order differential equation as a system of first-order state space equations:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_n \end{bmatrix} = \begin{bmatrix} -a_1x_1 - \ldots - a_nx_n \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} + \begin{bmatrix} u \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

In this particular case, the model is linear, so it can be represented in matrix form:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} -a_1 & -a_2 & \ldots & -a_{n-1} & -a_n \\ 1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & 1 & 0 \end{bmatrix} \boldsymbol{x}(t) + \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \boldsymbol{u}(t).$$

Let us consider a practical example of this technique:

Example 1.1.3 (Converting Newton's Second Law to State Space Form)**.** Many robotic systems, such as robotic arms and wheeled robots, involve dynamics that can be described by Newton's second law. Understanding how to convert these higher-order dynamics into state space form is crucial for controlling and analyzing numerous robotics systems.

Newton's second law describes the acceleration response of a mass m resulting from a force F:

$$F = m\ddot{s},$$

where s represents the one-dimensional positional displacement of the mass. Newton's second law, in this form, is a classic example of a *double integrator* system[6].

To convert this second-order differential equation into state space form, we first define the state vector:

$$\boldsymbol{x} \coloneqq \begin{bmatrix} s \\ \dot{s} \end{bmatrix}.$$

[6] In control theory, a double integrator describes a system in which the output (here, the position s) is obtained by integrating the input (acceleration) twice.

Then, we represent the dynamics in state space form:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} \boldsymbol{u},$$

where u represents the applied force F. By defining the state vector $\boldsymbol{x}$ and expressing the original second-order differential equation in this manner, we have effectively transformed it into a set of first-order differential equations.

1.2 *Kinematics and Dynamics*

Understanding a robot's physical motion is essential for enabling its autonomous operation. For instance, it is crucial to determine how an autonomous vehicle's actions, such as adjusting throttle and steering, affect its state and interaction with the environment. Similarly, it is important to understand how a robot manipulator's movements impact its ability to manipulate objects. A crucial aspect of this understanding involves the concepts of *kinematics* and *dynamics*, which govern the robot's physical motion and the constraints it must adhere to.

Definition 1.2.1 (Kinematics)**.** *Kinematics* is the study of the motion of physical systems, concerned with describing positions, velocities, and accelerations over time, without reference to the forces or torques that produce the motion.

A robot's kinematics outline limitations on its motion that are determined by its physical state or geometry. These constraints arise from the physical structure of the system, such as joint limits, actuator placement, linkage geometry, and the ways in which different components are mechanically connected. They determine how the system can move, independent of any external forces. For example, consider a wheeled robot. Static friction restricts the wheels from sliding laterally, meaning they cannot move in the direction parallel to the rotation axis. This kinematic constraint significantly limits the robot's ability to navigate and affects its overall maneuverability. By understanding these constraints, one can better grasp the feasible movements of the robot within its environment, which in turn informs how it can effectively interact with the world around it.

Definition 1.2.2 (Dynamics)**.** *Dynamics* is the study of the motion of physical systems as determined by the forces and torques acting upon them. It seeks to relate a system's motion to the underlying physical causes of that motion, such as gravity, friction, or applied inputs.

In the context of robotic or mechanical systems, dynamics are typically governed by Newton's Second Law, which states that the acceleration of a body is proportional to the net force acting on it. For example, the dynamics of an autonomous vehicle are described by the relationship between its acceleration and the external forces acting on it, including tire-road interaction, gravitational effects on slopes, and aerodynamic drag.

From the definitions above, kinematics describe constraints that arise from the robot's physical state or geometry, whereas dynamics explain how forces or inputs influence the robot's motion. In this section, we first introduce the concept of *generalized coordinates* for defining the physical configuration of a robot and discuss how the robot's kinematics and dynamics can be expressed in terms of these coordinates. Then, we will demonstrate how to use the system's kinematics and dynamics to define a state space model for the robot's physical motion. In the following chapters, we will explore how these models are applied to develop robust and high-performing algorithms for motion planning and control.

1.2.1 Generalized Coordinates

A robot's physical state, also referred to as its *configuration*, provides a complete specification of the position of every point on the robot at a given instant [7]. A configuration can be represented using a set of variables known as *generalized coordinates*, denoted by $\boldsymbol{q}(t) \in \mathbb{R}^{n_s}$. Generalized coordinates form a set of parameters that uniquely describe the robot's configuration relative to a reference frame. Depending on the system, they may include joint angles, Cartesian positions, orientations, or other parameters defining the robot's physical arrangement[8].

[7] T. Lozano Perez. "Spatial planning: a configuration space approach". In: *Autonomous Robot Vehicles*. 1990

Importantly, the configuration—i.e., the generalized coordinates—typically constitutes only a part of the full system state $\boldsymbol{x}$. As discussed in previous sections, in a state-space model, the state is a complete set of variables sufficient to determine the system's future evolution, given an external input. This usually includes both the generalized coordinates $\boldsymbol{q}$ and their time derivatives, called *generalized velocities*, denoted as $\dot{\boldsymbol{q}}$. In some systems, the state may also include additional internal variables such as actuator dynamics, sensor states, or environmental parameters.

[8] The terms "configuration" and "generalized coordinates" are often used interchangeably. However, while the configuration refers to the robot's physical arrangement in space, generalized coordinates are a specific mathematical representation of that arrangement. Multiple choices of generalized coordinates may represent the same configuration.

Example 1.2.1 (Rolling Wheel)**.** The configuration of a wheel rolling on a plane, as illustrated in Figure 1.1, can be represented by three parameters: the contact point position coordinates (x, y) and the heading angle θ relative to a fixed reference frame. This set of parameters, $\boldsymbol{q} = [x, y, \theta]^\top$, constitutes just one possible choice of generalized coordinates to define the wheel's configuration. Alternatively, the configuration could also be represented using the heading angle θ along with a polar coordinate representation of the contact point position.

1.2.2 Kinematic Constraints

Once a specific set of generalized coordinates, $\boldsymbol{q}$, is chosen to represent a robot's configuration, we can identify the relevant *kinematic constraints* for the robot. These kinematic constraints establish relationships between the generalized coordinates and the generalized velocities, thereby describing the limitations

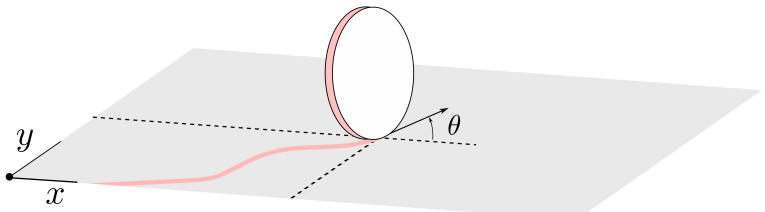


Figure 1.1: Generalized coordinates for a wheel rolling without slipping on a plane.

on the robot's motion. We refer the reader to Siciliano et al. [63] for a comprehensive treatment of robotic kinematics and the formulation of the associated constraint equations.

Definition 1.2.3 (Kinematic Constraints)**.** *Kinematic constraints* are a set of constraints imposed on the generalized coordinates, $\boldsymbol{q}$, and generalized velocities, $\dot{\boldsymbol{q}}$. We express kinematic constraints mathematically as:

$$\tilde{a}_i(\boldsymbol{q}, \dot{\boldsymbol{q}}) = 0, \qquad i = 1, \ldots, k < n_g, \tag{1.5}$$

where k is the number of constraints and n_g is the number of generalized coordinates.

In many robotics applications, kinematic constraints are linear with respect to the generalized velocities. These are known as *Pfaffian constraints*, and can be mathematically expressed as:

$$\boldsymbol{a}_i^\top(\boldsymbol{q})\dot{\boldsymbol{q}} = 0, \qquad i = 1, \ldots, k < n_g, \tag{1.6}$$

where $\boldsymbol{a}_i(\boldsymbol{q}) \in \mathbb{R}^{n_g}$. Pfaffian constraints can also be compactly represented in matrix form as:

$$A^\top(\boldsymbol{q})\dot{\boldsymbol{q}} = \boldsymbol{0}, \tag{1.7}$$

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

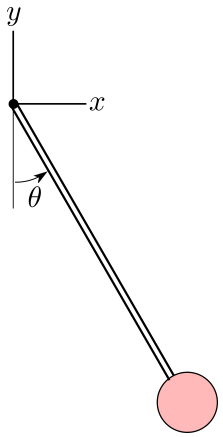


Figure 1.2: Generalized coordinates for a simple pendulum.

Example 1.2.2 (Pendulum)**.** Figure 1.2 shows a simple pendulum with a point mass and a rigid, massless, rod that rotates about a fixed pivot point. We can choose to represent the configuration of the pendulum by the Cartesian coordinate position of the mass, assuming the pivot point is the reference frame origin. The generalized coordinate vector for this choice is $\boldsymbol{q} = [x, y]^\top$, and the generalized velocity vector is $\dot{\boldsymbol{q}} = [\dot{x}, \dot{y}]^\top$. The fact that the rod connecting the pivot point to the mass is rigid introduces a restriction on the motion of this system, which we represent by the kinematic constraint:

$$\tilde{a}_1(\boldsymbol{q}, \dot{\boldsymbol{q}}) = x^2 + y^2 - L^2 = 0, \tag{1.8}$$

where L is the length of the rod. While this constraint is not in Pfaffian form, we can equivalently express it as a Pfaffian constraint by noting that:

$$\tilde{a}_1(\boldsymbol{q}, \dot{\boldsymbol{q}}) = 0 \implies \frac{\partial \tilde{a}_1(\boldsymbol{q}, \dot{\boldsymbol{q}})}{\partial t} = 0.$$

For the pendulum kinematic constraint in Equation (1.8), we have:

$$\frac{\partial \tilde{a}_1(\boldsymbol{q}, \dot{\boldsymbol{q}})}{\partial t} = 2x\dot{x} + 2y\dot{y},$$

and therefore we can write the constraint in the Pfaffian form of Equation (1.6) with:

$$\boldsymbol{a}_i^\top(\boldsymbol{q}) = \begin{bmatrix} 2x & 2y \end{bmatrix}. \tag{1.9}$$

The Pfaffian constraint in Equation (1.9) implies that Equation (1.8) holds as long as the pendulum starts in a state $\boldsymbol{q}(0)$ satisfying $\tilde{a}_1(\boldsymbol{q}(0)) = 0$.

An alternative choice of generalized coordinates to represent the pendulum's configuration is to consider the angle θ between the vertical and the pendulum's rod orientation, $\boldsymbol{q} = [\theta]$. This choice fully specifies the configuration without requiring us to define any kinematic constraints, making it a more natural choice for this system. Note that since $x = L\sin\theta$ and $y = -L\cos\theta$ the kinematic constraint in Equation (1.8) is trivially satisfied for all θ.

Example 1.2.3 (Rolling Wheel)**.** Consider the wheel illustrated in Figure 1.1, which we can represent with the generalized coordinates $\boldsymbol{q} = [x, y, \theta]^\top$. For this system, we can assume that the friction at the contact point between the wheel and the surface induces a no-slip condition. This no-slip condition is a constraint on the motion of the wheel that restricts the velocity component of the wheel in the lateral direction to always be zero. Since the unit vector $\boldsymbol{e}_v = [\cos\theta, \sin\theta]^\top$ describes the heading of the wheel, the lateral direction is given by the perpendicular unit vector $\boldsymbol{e}_{v,\perp} = [\sin\theta, -\cos\theta]^\top$. We can compute the lateral velocity from the dot product of the lateral direction unit vector and the velocity vector, $\boldsymbol{v} = [\dot{x}, \dot{y}]^\top$, which gives the no-slip kinematic constraint:

$$a_1(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \dot{x}\sin\theta - \dot{y}\cos\theta = 0. \tag{1.10}$$

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

1.2.3 *Holonomic and Nonholonomic Constraints*

Kinematic constraints often fall into two categories: *holonomic* or *nonholonomic*, depending on how they restrict the motion of the system. Holonomic constraints can be expressed solely as functions of the generalized coordinates, without involving generalized velocities. In contrast, nonholonomic constraints involve the generalized velocities and cannot be expressed solely in terms of the generalized coordinates.

Definition 1.2.4 (Holonomic Constraints)**.** Kinematic constraints that can be expressed in the form:

$$\tilde{a}_i(\boldsymbol{q}) = 0, \quad i = 1, \ldots, k < n_g, \tag{1.11}$$

are called *holonomic*.

In robotics applications, holonomic constraints generally arise due to mechanical interconnections, such as rigid links and joints of a robotic arm. We refer to a system that is only subject to holonomic constraints as a *holonomic system*. These constraints are a unique subclass of kinematic constraints that restrict the accessible configurations of the system. Specifically, for a system with n generalized coordinates under k holonomic constraints, the dimension of the space of accessible configurations is $n - k$. Holonomic constraints can always be equivalently expressed as Pfaffian constraints of the form Equation (1.6). This is because:

$$\tilde{a}_i(\boldsymbol{q}) = 0 \implies \frac{\partial \tilde{a}_i(\boldsymbol{q})}{\partial t} = 0,$$

and by differentiating the expression:

$$\frac{\partial \tilde{a}_i(\boldsymbol{q})}{\partial t} = \frac{\partial \tilde{a}_i(\boldsymbol{q})}{\partial \boldsymbol{q}} \dot{\boldsymbol{q}} = \boldsymbol{a}_i^\top(\boldsymbol{q}) \dot{\boldsymbol{q}}, \tag{1.12}$$

as we demonstrated in Example 1.2.2. However, it is important to note that not all Pfaffian constraints are holonomic. For a Pfaffian constraint to be holonomic, it must be integrable to the form in Equation (1.11). Specifically, there must exist a scalar function $\tilde{a}_i(\boldsymbol{q})$ such that:

$$\boldsymbol{a}_i^\top(\boldsymbol{q}) \dot{\boldsymbol{q}} = \frac{\partial \tilde{a}_i(\boldsymbol{q})}{\partial \boldsymbol{q}} \dot{\boldsymbol{q}} = 0. \tag{1.13}$$

This implies that the Pfaffian constraint, when integrated, yields a constraint solely dependent on the generalized coordinates $\boldsymbol{q}$, without any explicit dependence on their time derivatives $\dot{\boldsymbol{q}}$.

Example 1.2.4 (Pendulum)**.** Consider the pendulum from Example 1.2.2. The kinematic constraint in Equation (1.8) restricts the pendulum mass to lie on a circle of radius L, which is a subset of all possible generalized coordinates. This constraint is holonomic since we can express it as a function of only the generalized coordinates.

Example 1.2.5 (Rolling Wheel)**.** Consider the wheel from Example 1.2.3, where the kinematic constraint in Equation (1.10) restricts the direction of motion. In contrast to the pendulum, this constraint does not limit the wheel's ability to reach any configuration of generalized coordinates, namely the position and heading. Mathematically, we cannot integrate the constraint in Equation (1.10) to yield a constraint of the form $\tilde{a}_i(\boldsymbol{q}) = 0$, and thus this constraint is not holonomic.

While holonomic constraints restrict the system's accessible configurations, kinematic constraints can also limit the motion between configurations. We refer to these constraints as *nonholonomic* constraints. A system that is subject to at least one nonholonomic constraint is referred to as a *nonholonomic system*.

Definition 1.2.5 (Nonholonomic Constraints)**.** Constraints that can be described in Pfaffian form, $\boldsymbol{a}_i(\boldsymbol{q})^\top \dot{\boldsymbol{q}} = 0$, but cannot be integrated to the form $\tilde{a}_i(\boldsymbol{q}) = 0$ are called *nonholonomic*.

That is, the Pfaffian expressions cannot be written as the total time derivative of any scalar function that depends only on the generalized coordinates. In other words, there exists no scalar function $\tilde{a}_i(\boldsymbol{q})$ such that $\frac{d}{dt}\tilde{a}_i(\boldsymbol{q}) = \boldsymbol{a}_i(\boldsymbol{q})^\top \dot{\boldsymbol{q}}$. Geometrically, nonholonomic constraints restrict the instantaneous generalized velocities to lie in the null space of $A(\boldsymbol{q})^\top$, where $A(\boldsymbol{q})$ is the matrix whose rows are the vectors $\boldsymbol{a}_i(\boldsymbol{q})^\top$ corresponding to each nonholonomic constraint.

Example 1.2.6 (Rolling Wheel)**.** Consider the wheel example from Example 1.2.3, which has a nonholonomic constraint:

$$a_1(\boldsymbol{q})^\top \dot{\boldsymbol{q}} = \begin{bmatrix} \sin\theta & -\cos\theta & 0 \end{bmatrix} \dot{\boldsymbol{q}} = 0.$$

The null space of $a_1(\boldsymbol{q})^\top$ in this case is spanned by the vectors $[\cos\theta,\ \sin\theta,\ 0]$ and $[0,\ 0,\ 1]$, which suggests that all motion must be made up of a linear combination of these vectors. Intuitively, we expect this because $[\cos\theta,\ \sin\theta,\ 0]$ is the rolling direction and $[0,\ 0,\ 1]$ is the axis the wheel rotates about.

In summary, holonomic constraints restrict a system's motion by confining its configurations to lower-dimensional manifolds—specifically, level sets defined by scalar equations of the form $\tilde{a}_i(\boldsymbol{q}) = 0$. For a system with n generalized coordinates and k independent holonomic constraints, the configuration space is effectively reduced to a manifold of dimension $n - k$. This reduction reflects a true loss of accessibility in the configuration space: the system can only evolve along a restricted subset of configurations determined by the initial conditions and the constraint equations. In contrast, nonholonomic constraints act directly on the system's instantaneous velocities, typically expressed in Pfaffian form as $A(\boldsymbol{q})^\top \dot{\boldsymbol{q}} = 0$, where $A(\boldsymbol{q})^\top$ is a full-rank matrix of dimension $k \times n$. These constraints restrict the allowed directions of motion at each configuration by confining $\dot{\boldsymbol{q}}$ to lie in an $(n-k)$-dimensional subspace. However, since nonholonomic constraints are not integrable, they do not reduce the dimensionality of the configuration space itself. That is, although motion is constrained at each instant, the system may still be able to reach any configuration in the configuration space through admissible trajectories that respect the velocity constraints. Thus, while holonomic constraints reduce the number of independent configuration variables and confine the system to a lower-dimensional subset of the configuration space, nonholonomic constraints preserve full accessibility of the configuration space but restrict how that space can be traversed.

1.2.4 *Kinematic Models*

Once we have chosen an appropriate set of generalized coordinates $\boldsymbol{q}$ and have identified the relevant kinematic constraints, we can convert the kinematic constraints into a state space model of the form in Equation (1.1), which we refer to as a *kinematic model*.

Definition 1.2.6 (Kinematic Model)**.** Given a generalized coordinate vector $\boldsymbol{q} \in \mathbb{R}^{n_g}$, and k Pfaffian constraints[9], $A^\top(\boldsymbol{q})\dot{\boldsymbol{q}} = \boldsymbol{0}$, a *kinematic model* is a state space

[9] These Pfaffian constraints can come from a combination of holonomic and non-holonomic constraints.

model of the form:

$$\dot{\boldsymbol{q}} = G(\boldsymbol{q})\boldsymbol{u}, \tag{1.14}$$

where $\boldsymbol{u} \in \mathbb{R}^p$ is the input and where the column space of $G(\boldsymbol{q}) \in \mathbb{R}^{n_g \times n_g - k}$ spans the null space of $A^\top(\boldsymbol{q})$.

Each input in $\boldsymbol{u}$ corresponds to one degree of freedom of the system, and for any initial condition $\boldsymbol{q}(0)$ and sequence of inputs $\boldsymbol{u}(t)$ the solutions to the kinematic model are guaranteed to satisfy the Pfaffian constraints. We can prove that the trajectories of the kinematic model will satisfy the Pfaffian constraints by writing the model in the equivalent form:

$$\dot{\boldsymbol{q}} = G(\boldsymbol{q})\boldsymbol{u} = \sum_{i=1}^{n-k} \boldsymbol{g}_i(\boldsymbol{q})u_i,$$

where $\boldsymbol{g}_i \in \mathbb{R}^{n_g}$ is the i-th column of G and u_i is the i-th input. In this form, we can more easily see that each input acts on the generalized velocity $\dot{\boldsymbol{q}}$ through a particular mode that is defined by the vector $\boldsymbol{g}_i$. Since we have specified in Theorem 1.2.6 that the column space of G spans the null space of $A^\top(\boldsymbol{q})$, we know that by definition:

$$A^\top(\boldsymbol{q})\boldsymbol{g}_i(\boldsymbol{q})u_i = 0,$$

for any input $u_i \in \mathbb{R}$ and for all coordinates $\boldsymbol{q}$. Therefore, by definition each component of the input vector can only influence the generalized velocity in a way that satisfies the Pfaffian constraints. Another way to see this mathematically is by substituting the kinematic model into the Pfaffian constraint equation:

$$\begin{aligned} A^\top(\boldsymbol{q})\dot{\boldsymbol{q}} &= A^\top(\boldsymbol{q})G(\boldsymbol{q})\boldsymbol{u}, \\ &= A^\top(\boldsymbol{q})\big(\sum_{i=1}^{n-k} \boldsymbol{g}_i(\boldsymbol{q})u_i\big), \\ &= \sum_{i=1}^{n-k} A^\top(\boldsymbol{q})\boldsymbol{g}_i(\boldsymbol{q})u_i, \\ &= 0. \end{aligned}$$

Example 1.2.7 (Rolling Wheel)**.** Consider the rolling wheel example from Example 1.2.3, which has a single nonholonomic constraint:

$$a_1(\boldsymbol{q})^\top \dot{\boldsymbol{q}} = \begin{bmatrix} \sin\theta & -\cos\theta & 0 \end{bmatrix} \dot{\boldsymbol{q}} = 0,$$

where $\boldsymbol{q} = [x, y, \theta]^\top$. The null space of $a_1(\boldsymbol{q})^\top$ is spanned by the vectors $[\cos\theta, \sin\theta, 0]^\top$ and $[0, 0, 1]^\top$ and therefore the kinematic model is given by:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}. \tag{1.15}$$

In this case, the inputs u_1 and u_2 have an intuitive physical meaning: u_1 is the speed at which the wheel is moving, and u_2 is the wheel's angular rotation rate.

1.2.5 Dynamics Models

Kinematic models describe the geometric constraints that limit a robot's motion—for example, the fact that a robotic arm can only rotate about its joints. However, kinematics alone does not explain how motion is generated or resisted. For that, we turn to dynamics, which describe how forces and torques influence motion by producing accelerations.

Newton's second law of motion is the foundation of robot dynamics, relating the net force acting on a body to its acceleration. Applied to a single point mass, the law takes the form:

$$\boldsymbol{F}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = m\ddot{\boldsymbol{q}}, \tag{1.16}$$

where m is the mass of the particle, $\boldsymbol{q} = [x, y, z]^\top$ is its position vector, and $\boldsymbol{F}$ is the total force acting on the particle.

Example 1.2.8 (Mass-Spring-Damper System)**.** A fundamental example in the study of dynamics is the one-dimensional mass-spring-damper system. The system consists of a mass m attached to a spring and a damper, constrained to move along a line. The total force acting on the mass is typically composed of three terms:

1. an external input force F_{external},
2. a spring force $-kx$ that resists displacement from the equilibrium position, and
3. a damping force $-c\dot{x}$ that resists velocity.

The net force on the mass is:

$$F = F_{\text{external}} - kx - c\dot{x},$$

where $k > 0$ and $c > 0$ are the spring and damping coefficients, respectively, and x is the displacement from equilibrium. Substituting this expression into Newton's second law yields the following second-order differential equation:

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

This equation models oscillatory motion with damping, and it arises in many robotics applications—for example, when analyzing joint compliance, actuator dynamics, or contact interactions.

While the mass-spring-damper system illustrates the dynamics of a single particle in one dimension, real-world robotic systems are often more complex. To capture their behavior, we extend Newton's second law to systems of interconnected particles, typically modeled in robotics as *rigid bodies*. A rigid body is an idealized object in which the relative positions of all constituent particles remain fixed over time, regardless of external forces. This assumption implies that the body does not deform and allows us to reduce a complex system of

interacting particles to a simpler model governed by the motion of a finite set of parameters (e.g., position and orientation of a frame fixed to the body). As a result, rigid body dynamics provide a powerful and tractable framework for analyzing and simulating robotic systems[10].

The motion of a rigid body in three-dimensional space can be described in terms of its translational and rotational dynamics. Translational dynamics govern the motion of the body's center of mass and are described by Newton's second law, as expressed in Equation (1.16), where the position variable refers specifically to the center of mass. In three-dimensional space, a rigid body has three translational degrees of freedom, corresponding to movement along each of the Cartesian axes. Rotational dynamics, on the other hand, describe how the body's orientation evolves over time. A rigid body also has three degrees of freedom associated with its orientation in three-dimensional space, corresponding to rotation about each of its principal axes. Unlike translation, orientation cannot be represented by a single vector, and several parameterizations are commonly used. Notable examples include rotation matrices, which provide a full and unambiguous representation at the cost of redundancy; *Euler angles*, which use a sequence of three rotations to represent orientation, and *quaternions*, which offer a compact and singularity-free alternative well-suited for numerical applications.

[10] Interesting examples in robotics where the rigid body assumption may not hold include soft robots, robots with compliant end-effectors, or robots with lightweight flexible structures.

The rotational dynamics[11] of a rigid body are governed by the time evolution of its angular momentum. Specifically, they are described by:

$$\boldsymbol{M} = \dot{\boldsymbol{H}}, \tag{1.17}$$

where $\boldsymbol{M}$ denotes the total external moment (or torque) acting on the body, and $\boldsymbol{H}$ is the angular momentum, typically computed about the center of mass. This relationship is commonly referred to as *Euler's equation* for rotational dynamics and captures how applied torques influence changes in the body's rotational motion.

[11] We refer the reader to Shuster [59] for an in-depth treatment of rotational dynamics and attitude representations.

An alternative to the Newton-Euler method—defined by Equations (1.16)-(1.17)—for deriving the equations of motion for a rigid body is the *Lagrange Method*. This approach is closely tied to the notion of generalized coordinates, generalized velocities, and kinematic constraints, and takes an energy-based approach. In the Lagrange method, the dynamics of a rigid body are derived from a scalar quantity called the *Lagrangian*, defined as the difference between the kinetic and potential energies:

$$L(\boldsymbol{q}, \dot{\boldsymbol{q}}) = T(\boldsymbol{q}, \dot{\boldsymbol{q}}) - V(\boldsymbol{q}), \tag{1.18}$$

where $T(\boldsymbol{q}, \dot{\boldsymbol{q}})$ and $V(\boldsymbol{q})$ denote the kinetic and potential energies of the system, respectively. The evolution of the system is governed by *Lagrange's equations* [12,13], which incorporate both external influences and kinematic constraints. In the absence of constraints, the equations of motion are given by:

$$\frac{d}{dt}\Big(\frac{\partial L}{\partial \dot{\boldsymbol{q}}_j}\Big) - \frac{\partial L}{\partial \boldsymbol{q}_j} = Q_j, \quad j = 1, \ldots, n_g, \tag{1.19}$$

[12] B. Siciliano et al. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 2008. Chap. 7

[13] K. M. Lynch and K. C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017. Chap. 8

where $Q_j \in \mathbb{R}$ is a non-conservative *generalized force* associated with the generalized coordinate $\boldsymbol{q}_j$[14], and n_g corresponds to the system's degrees of freedom. Equations (1.19) describe how the generalized forces acting on the system relate to its position, velocity, and acceleration, providing a systematic way to derive the system's dynamic model from its kinetic and potential energies.

In the presence of Pfaffian constraints[15], Lagrange's equations take the form:

$$\begin{aligned} \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\boldsymbol{q}}_j}\right) - \frac{\partial L}{\partial \boldsymbol{q}_j} &= Q_j + \sum_{i=1}^{k} \lambda_i a_{ij}(\boldsymbol{q}), \quad j = 1, \ldots, n_g, \\ \boldsymbol{a}_i^\top(\boldsymbol{q})\dot{\boldsymbol{q}} &= 0, \qquad i = 1, \ldots, k, \end{aligned} \tag{1.20}$$

where a_{ij} is the j-th component of the i-th Pfaffian constraint vector $\boldsymbol{a}_i(\boldsymbol{q})$ and $\lambda_i \in \mathbb{R}$ is a *Lagrange multiplier*. The first n_g equations describe the dynamics of the generalized coordinates under the influence of both external and constraint forces, while the remaining k equations represent the kinematic constraints themselves. The complete system thus comprises $n_g + k$ equations in $n_g + k$ unknowns (the generalized coordinates and the Lagrange multipliers), and is commonly referred to as the *standard non-holonomic form*. If the system is holonomic and the generalized coordinates are chosen to be independent, the constraints are implicitly satisfied, and Lagrange's equations reduce to the simpler, unconstrained form introduced in Equation (1.19).

[14] Generalized forces are projections of physical forces and torques into the generalized coordinate space. Forces not derived from a potential—such as friction—are termed non-conservative. In contrast, forces like gravity are conservative.

[15] While Lagrange's method can accommodate general constraints, we focus here on Pfaffian constraints for simplicity.

Example 1.2.9 (Pendulum)**.** Consider again the pendulum depicted in Figure 1.2. To model its dynamics, we analyze how gravity drives the motion of the mass. Specifically, we will demonstrate four distinct approaches for deriving the equations of motion—using both Cartesian and polar coordinates, and applying both the Newton-Euler and Lagrange methods. This comparison will highlight how the choice of generalized coordinates can influence the complexity of the derivation.

We begin by using Newton's second law to derive the dynamics of the pendulum, focusing on the two forces acting on the mass: gravity and the force from the rod. We assume that the rod's force acts purely along its axis. Since the pendulum's length is fixed, this force must counteract the component of gravity along the rod and generate the required centripetal acceleration. The axial force exerted by the rod is given by:

$$F_r = mg\cos\theta + \frac{mv^2}{L},$$

where m is the mass of the pendulum, g is gravitational acceleration, L is the length of the rod, and v is the speed of the mass. The gravitational force is:

$$F_g = mg,$$

acting along the negative y-direction. To compute the net force in Cartesian coordinates, we project both the rod's force and the gravitational force onto

the x- and y-axes:

$$F_x = -\frac{mv^2}{L}\sin\theta - mg\sin\theta\cos\theta,$$
$$F_y = \frac{mv^2}{L}\cos\theta - mg\sin^2\theta.$$

Applying Newton's second law as defined in Equation (1.16) yields the equations of motion:

$$\ddot{x} = -\frac{v^2}{L}\sin\theta - g\sin\theta\cos\theta,$$
$$\ddot{y} = \frac{v^2}{L}\cos\theta - g\sin^2\theta.$$

To express these equations purely in terms of Cartesian coordinates, we substitute $x = L\sin\theta$ and $y = -L\cos\theta$, leading to:

$$\begin{aligned}\ddot{x} &= \frac{1}{L^2}(gxy - xv^2),\\ \ddot{y} &= -\frac{1}{L^2}(gx^2 + yv^2),\end{aligned} \tag{1.21}$$

with $v^2 = \dot{x}^2 + \dot{y}^2$. This method requires careful force analysis, as the kinematic constraint (fixed-length rod) is handled implicitly through the projected components of the rod's force.

As a second approach to deriving the equations of motion using Cartesian coordinates, we now apply the Lagrange method, which yields a slightly simpler formulation. We begin by defining the kinetic and potential energies:

$$T = \frac{1}{2}m(\dot{x}^2 + \dot{y}^2), \quad V = mgy,$$

and observe that there are no external non-conservative generalized forces[16]. As discussed in Example 1.2.2, we recall the system's single Pfaffian constraint:

$$x\dot{x} + y\dot{y} = 0.$$

[16] Gravity is a conservative force, so no generalized non-conservative forces appear.

Thus, using Lagrange's equations introduced in Equation (1.20), we obtain:

$$\begin{aligned}m\ddot{x} &= \lambda x,\\ m\ddot{y} + mg &= \lambda y,\\ x\dot{x} + y\dot{y} &= 0.\end{aligned} \tag{1.22}$$

We can solve for the Lagrange multiplier λ by differentiating the constraint with respect to time:

$$\frac{d}{dt}(x\dot{x} + y\dot{y}) = \dot{x}^2 + \dot{y}^2 + x\ddot{x} + y\ddot{y} = 0,$$

and by substituting the expressions for $\ddot{x}$ and $\ddot{y}$ from the first two Lagrange's equations in (1.22) to obtain:

$$\dot{x}^2 + \dot{y}^2 + \frac{1}{m}x^2\lambda + \frac{1}{m}y^2\lambda - gy = 0.$$

Solving for λ yields:

$$\lambda = \frac{m}{L^2}(gy - v^2),$$

where we used $L^2 = x^2 + y^2$ and $v^2 = \dot{x}^2 + \dot{y}^2$. Finally, substituting the expression for λ back into the equations of motion and simplifying, we find:

$$\begin{aligned} \ddot{x} &= \frac{1}{L^2}(gxy - xv^2), \\ \ddot{y} &= -\frac{1}{L^2}(gx^2 + yv^2), \end{aligned} \tag{1.23}$$

which matches the result previously obtained using Newton's method in Equation (1.21).

After applying both the Newton-Euler and Lagrange methods in Cartesian coordinates, we now replicate the derivation in polar coordinates. To apply Euler's equation for rotational dynamics as given in Equation (1.17), we adopt a coordinate frame fixed at the pivot point. The gravitational force acting on the mass generates a moment about the pivot:

$$M = -mgL\sin\theta,$$

while the angular momentum of the system about the same point is:

$$H = mL^2\dot{\theta},$$

where mL^2 is the *moment of inertia*[17] about the pivot. Substituting into Euler's equation yields the system dynamics:

[17] In Euler's equation, the moment of inertia plays a role analogous to mass in Newton's second law.

$$\ddot{\theta} = -\frac{g}{L}\sin\theta, \tag{1.24}$$

which are considerably more compact than the corresponding equations derived in Cartesian coordinates.

The Lagrange method also becomes significantly simpler when using the polar coordinate θ, as there is no need to handle Pfaffian constraints explicitly. In this formulation, the kinetic and potential energies of the system are:

$$T = \frac{1}{2}mL^2\dot{\theta}^2, \quad V = -mgL\cos\theta.$$

Applying Equation (1.20), and noting the absence of non-conservative generalized forces or constraints, we obtain the following equation of motion:

$$\ddot{\theta} = -\frac{g}{L}\sin\theta, \tag{1.25}$$

which, as expected, matches the result derived using Euler's equation in Equation (1.24).

1.3 *Wheeled Robot Motion Models*

Robots are developed in diverse forms, sizes, and configurations, each featuring distinct mobility solutions tailored to specific applications. Among these,

wheeled robots are particularly common because of their excellent mobility and simple design. In this section, we demonstrate how the concepts from the preceding sections can be applied to two classic and widely used motion models for simple wheeled robots: the *unicycle model* and the *differential drive model*.

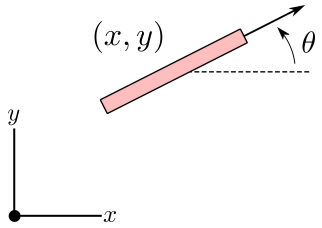


Figure 1.3: Generalized coordinates for a unicycle.

1.3.1 Unicycle Model

The unicycle model is one of the simplest kinematic models used for modeling robot motion. This model leverages the kinematics of the rolling wheel discussed in Example 1.2.3, essentially assuming the robot is constrained only by a no-slip constraint from a single wheel. Figure 1.3 illustrates a simplified diagram of the generalized coordinates for the unicycle model.

The kinematic model is identical to the one presented in Equation (1.15), namely:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}, \tag{1.26}$$

where v represents the forward speed and ω denotes the rotational rate.

While the unicycle model may be a simplified representation of the robot's true kinematics, it remains valuable in many contexts where detailed dynamics are unnecessary. Its main advantage lies in its simplicity, which often enables more computationally efficient algorithms. In practice, such lower-fidelity models are often used in the early stages of a system's design or decision-making process, and are later refined or supplemented with more accurate models when higher precision is required.

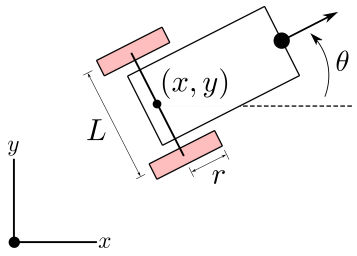


Figure 1.4: Generalized coordinates for a differential drive robot.

1.3.2 Differential Drive Model

The differential drive model is a variation on the unicycle model from the previous section, with two wheels fixed on a shared rear axle and a passive front wheel that induces no additional kinematic constraints. This model uses same generalized coordinates as the unicycle model, $\boldsymbol{q} = [x, y, \theta]^\top$, but also requires the definition of certain geometric parameters: the width of the rear axle, denoted by L, and the radius of the wheels, denoted by r, as illustrated in Figure 1.4.

The differential drive model assumes the wheels roll without slipping, making the derivation of its kinematic constraints similar to that of a single rolling wheel, as discussed in Example 1.2.3. The heading vector of each wheel is given by $\boldsymbol{e}_v = [\cos\theta, \sin\theta]^\top$, and the lateral direction is $\boldsymbol{e}_{v,\perp} = [\sin\theta, -\cos\theta]^\top$. Using the lateral direction vector, we define the no-slip kinematic constraints for the wheels as:

$$\dot{\boldsymbol{p}}_l^\top \boldsymbol{e}_{v,\perp} = 0, \quad \dot{\boldsymbol{p}}_r^\top \boldsymbol{e}_{v,\perp} = 0,$$

where $\dot{\boldsymbol{p}}_l$ and $\dot{\boldsymbol{p}}_r$ are the velocity vectors of the left and right wheels, respectively. Next, we express the wheel velocity vectors $\dot{p}_l$ and $\dot{p}_r$ as functions of the

generalized coordinates and velocities by leveraging the robot's geometry. The positions of the left and right wheel centers, denoted as $\boldsymbol{p}_l$ and $\boldsymbol{p}_r$, respectively, can be computed from the generalized coordinates by:

$$\boldsymbol{p}_l = \begin{bmatrix} x - \frac{L}{2}\sin\theta \\ y + \frac{L}{2}\cos\theta \end{bmatrix}, \quad \boldsymbol{p}_r = \begin{bmatrix} x + \frac{L}{2}\sin\theta \\ y - \frac{L}{2}\cos\theta \end{bmatrix}.$$

Taking the time derivative of these positions yields the velocity vectors:

$$\dot{\boldsymbol{p}}_l = \begin{bmatrix} \dot{x} - \dot{\theta}\frac{L}{2}\cos\theta \\ \dot{y} - \dot{\theta}\frac{L}{2}\sin\theta \end{bmatrix}, \quad \dot{\boldsymbol{p}}_r = \begin{bmatrix} \dot{x} + \dot{\theta}\frac{L}{2}\cos\theta \\ \dot{y} + \dot{\theta}\frac{L}{2}\sin\theta \end{bmatrix}.$$

After some algebraic manipulation, we find that the no-slip kinematic constraints for each wheel are equivalent:

$$\dot{\boldsymbol{p}}_l^\top \boldsymbol{e}_{v,\perp} = \dot{\boldsymbol{p}}_r^\top \boldsymbol{e}_{v,\perp} = \dot{x}\sin\theta - \dot{y}\cos\theta = 0,$$

indicating that the no-slip constraint for both wheels is redundant, and thus the constraint matches the single wheel constraint from Example 1.2.3. This is intuitive because the wheels are rigidly connected; hence, if one wheel cannot move laterally, neither can the other. The kinematic model for the differential drive model is also identical to the single wheel model in Equation (1.26), but the inputs can now be expressed in a more realistic form relative to the actual geometry of the robot.

In particular, instead of using the forward speed v and body rotation rate ω as inputs, as in Equation (1.26), the differential drive model uses the rotation rates of the left and right wheels, ω_l and ω_r. We can derive a relationship between these sets of inputs by considering the geometry of the robot and the no-slip wheel assumption. First, denote the position $\boldsymbol{p} = [x, y]^\top$ in terms of the wheel center positions by $\boldsymbol{p} = \frac{1}{2}(\boldsymbol{p}_l + \boldsymbol{p}_r)$, thus the velocity vector is $\dot{\boldsymbol{p}} = \frac{1}{2}(\dot{\boldsymbol{p}}_l + \dot{\boldsymbol{p}}_r)$. By the no-slip wheel assumption, the velocity v can be expressed as $v = \boldsymbol{e}_v^\top \dot{p}$, leading to:

$$\begin{aligned} v &= \boldsymbol{e}_v^\top \dot{\boldsymbol{p}}, \\ &= \frac{1}{2}\boldsymbol{e}_v^\top(\dot{\boldsymbol{p}}_l + \dot{\boldsymbol{p}}_r), \\ &= \frac{1}{2}(v_l + v_r), \\ &= \frac{r}{2}(\omega_l + \omega_r), \end{aligned}$$

where r is the radius of the wheel and v_l and v_r are the speeds of the left and right wheels, respectively. Additionally, the no-slip condition on each wheel is given by $v_l = \boldsymbol{e}_v^\top \dot{\boldsymbol{p}}_l$ and $v_r = \boldsymbol{e}_v^\top \dot{\boldsymbol{p}}_r$, expanded as:

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

Since $\dot{x}\cos\theta + \dot{y}\sin\theta = v$, we simplify these expressions to:

$$\frac{L}{2}\dot{\theta} = v_r - v,$$
$$\frac{L}{2}\dot{\theta} = v - v_l.$$

Combining these gives:

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

establishing the relationship between the generalized velocity $\dot{\theta}$ and the wheel rotational speeds.

In summary, the mapping between the inputs can be defined as:

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

which allows us to define the differential drive model:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \frac{r}{2}\cos\theta & \frac{r}{2}\cos\theta \\ \frac{r}{2}\sin\theta & \frac{r}{2}\sin\theta \\ \frac{r}{L} & -\frac{r}{L} \end{bmatrix} \begin{bmatrix} \omega_r \\ \omega_l \end{bmatrix}. \tag{1.27}$$

Despite the slight increase in complexity over the unicycle model, this model leverages the geometry of the robot to make the inputs more intuitive. This enhancement makes the differential drive model more suitable for certain motion planning and control tasks, as the robot's actuation typically originates from motors attached to the wheels' axles.

More generally, a kinematic state-space model should be interpreted only as a subsystem of a more comprehensive dynamical model. In particular, kinematic models typically assume direct control over certain motion variables—such as velocity or angular rate—without accounting for how these quantities are generated or constrained by the physical system. For more realistic modeling, it is often necessary to extend the kinematic model to include additional integrators in front of the control inputs.

Example 1.3.1 (Dynamic extension of the unicycle model)**.** The unicycle model introduced in Equation (1.26) assumes direct control over the forward velocity v and angular velocity ω, with the state defined by the variables (x, y, θ). To reflect the fact that velocity v is itself the result of integrating an acceleration input a, the model can be extended by treating v as an additional state, yielding the augmented state (x, y, θ, v) and input (ω, a). The dynamics become:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ \omega \\ a \end{bmatrix}. \tag{1.28}$$

This dynamic extension accounts for acceleration as a control input and enables the modeling of more realistic scenarios involving, e.g., actuation limits.

1.3.3 Bicycle/Simple Car Model

The bicycle model is a simplified kinematic model commonly used to approximate the motion of vehicles with two front-steered wheels and two rear-driven wheels, such as cars or mobile robots with similar geometry. The model captures key steering dynamics while assuming no slip at the contact points of the wheels. It is called a "bicycle" model because the two front wheels and two rear wheels are collapsed into a single front and rear wheel aligned on a common axis, forming a virtual two-wheeled vehicle. Compared to the unicycle and differential drive models introduced earlier, the bicycle model enforces more realistic kinematic constraints on how the system can turn. In particular, it captures the fact that the vehicle must steer to follow curved paths, and cannot rotate in place. As such, it provides a better approximation for many wheeled systems while still remaining relatively simple.

Figure 1.5 shows the simplified geometry of the bicycle model. This model can be derived by enforcing nonholonomic constraints on the rolling direction of each wheel and assuming ideal no-slip contact, following the discussion in previous sections. Figure 1.6 illustrates how the same kinematic model can be interpreted in the context of a four-wheeled vehicle. The key idea is that both front wheels steer with a common angle ϕ, and the vehicle moves forward with velocity v, subject to the no-slip constraints. These assumptions lead to the following differential equations characterizing the car model:

$$\begin{aligned} \dot{x} &= v\cos\theta, \\ \dot{y} &= v\sin\theta, \\ \dot{\theta} &= \frac{v}{L}\tan\phi, \end{aligned} \tag{1.29}$$

where (x, y) is the position of the rear axle center, θ is the heading angle, v is the forward speed, ϕ is the steering angle, and L is the length of the wheelbase (the distance between the front and rear axles). Therefore, we define the state as $\boldsymbol{x} = [x,\, y,\, \theta]^\top$ and the control input as $\boldsymbol{u} = [v,\, \phi]^\top$.

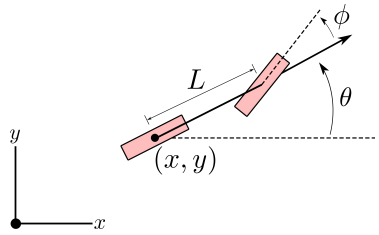


Figure 1.5: The bicycle model approximates the motion of a four-wheeled vehicle by collapsing each axle into a single wheel, aligned with the vehicle's centerline. The state consists of the position, (x, y), of the rear axle center and the heading angle, θ. The control inputs are the forward velocity, v, and the steering angle, ϕ.

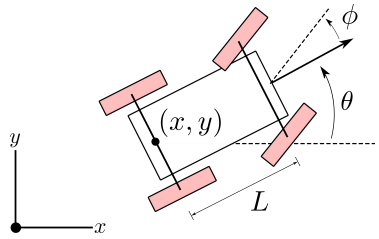


Figure 1.6: The same bicycle model applied to a car-like vehicle. The control and state definitions are identical to Figure 1.5, but the visualization makes explicit the mapping between the simplified model and a four-wheeled car.

1.4 Simulating Robot Dynamics

In Section 1.1, we introduced the concept of a *state space model* to mathematically describe the evolution of a robot's state over time. In Section 1.2 we demonstrated how a robot's *kinematics and dynamics* are used to derive a state space model that represents its physical motion. In this section, we present several computational techniques for *simulating* the changes in a robot's state over time.

The state space model in Equation (1.1) is a general system of ordinary differential equations, which in most cases cannot be solved analytically. Numerical simulation provides a practical approach to obtaining approximate solutions, allowing us to better understand a robot's dynamics and to test and validate algorithms for robot autonomy. Typically, when we refer to *simulating* a system,

we mean approximately solving an *initial value problem* (IVP) for a system of differential equations, as defined in Equation (1.1):

$$\dot{\boldsymbol{x}} = h(\boldsymbol{x}(t), t), \quad \boldsymbol{x}(t_0) = \boldsymbol{x}_0,$$

where $h(\boldsymbol{x}(t), t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t))$, and the input $\boldsymbol{u}(t)$ may either be explicitly defined as a known function of time (e.g., a predefined control sequence), or computed at each time step based on the current state.

The objective of this initial value problem is to find the trajectory $\boldsymbol{x}(t)$, starting from $\boldsymbol{x}(t_0)$, that satisfies the differential equation[18]. By the *Fundamental Theorem of Calculus*[19], the solution at time t can be written as:

$$\boldsymbol{x}(t) = \boldsymbol{x}(t_0) + \int_{t_0}^{t} h(\boldsymbol{x}(\tau), \tau) \mathrm{d}\tau.$$

[18] If h is Lipschitz continuous in $x(t)$ and continuous in t, the trajectory $x(t)$ exists and is unique.

[19] This expresses the inverse relationship between differentiation and integration: integrating the derivative $\dot{\boldsymbol{x}} = h(\boldsymbol{x}(t), t)$ over time recovers the original function $\boldsymbol{x}(t)$.

In general, evaluating this integral analytically for arbitrary functions $h(\boldsymbol{x}(t), t)$ is intractable. Therefore, we typically resort to numerical integration methods that involve a *discretization in time*:

$$\boldsymbol{x}(t) = \boldsymbol{x}(t_0) + \sum_{k=0}^{N-1} \int_{t_k}^{t_{k+1}} h(\boldsymbol{x}(\tau), \tau) \mathrm{d}\tau, \tag{1.30}$$

where $t_0 < t_1 < \cdots < t_N = t$ define a time grid, and each interval has width $\Delta t_k = t_{k+1} - t_k$. This decomposition breaks the continuous integration problem into a sum of smaller integrals over short intervals. Within each interval $[t_k, t_{k+1}]$, we can then approximate the integral using various numerical quadrature rules—such as the Euler method, the Midpoint method, or higher-order Runge-Kutta schemes—which trade off computational cost and accuracy. In this section, we provide a concise introduction to some of the most widely used methods.

Example 1.4.1 (Simple IVP)**.** To illustrate the different numerical integration methods, we will consider the initial value problem defined below and presented in Algorithm 1 as an example:

$$\dot{x}(t) = x(t) \sin^2(t), \quad x(0) = 1.$$

Our goal is to approximate the trajectory $x(t)$ over the interval $[0, 10]$ using various integration methods.

Concretely, we will explore different techniques to approximate the following analytical solution:

$$x(t) = x_0 \exp\left(\frac{t - t_0 - \sin(t - t_0)\cos(t + t_0)}{2}\right),$$

which, for our specific initial conditions $x_0 = 1$ and $t_0 = 0$, simplifies to:

$$x(t) = \exp\left(\frac{t - \sin(t)\cos(t)}{2}\right).$$

Simple IVP (Running Example)

```python
import numpy as np

# Define the derivative function h(x, t)
def h(x, t):
    return x * np.sin(t) ** 2

# Define initial conditions and final time
x0 = 1.0 # Initial state x(t0)
t0 = 0.0 # Initial time t0
tf = 10.0 # Final time tf

Δt = 0.5 # Discretization step
t = np.arange(t0, tf + Δt, Δt) # Array of timestamps

# Compute analytical solution
x_true = x0*np.exp(((t-t0) - np.sin(t-t0)*np.cos(t+t0))/2)

def integrate(h, x0, t, method):
    x = np.zeros((t.size, x0.size))
    x[0] = x0
    for i in range(t.size - 1):
        Δt = t[i + 1] - t[i]
        x[i + 1] = method(h, x[i], t[i], Δt)
    return x

def method(h, x, t, Δt):
    # Implment here numerical integreation method

# Test a specific numerical method
x_method = integrate(h, x0, t, method)
```

Algorithm 1: Definition of an illustrative initial value problem. The code for this example is available in the repository `github.com/StanfordASL/pora-exercises` in the notebook `ch01/simulation.ipynb`. In the following sections, we will explore different numerical integration methods and implement them in a custom `method` function.

1.4.1 *Euler method*

One of the simplest techniques for approximating the integral within each time interval of the discretized problem, as shown in Equation 1.30, is the Euler method[20]. This method approximates the integral over a short interval $[t_k, t_{k+1}]$

[20] Named after the Swiss mathematician Leonhard Euler and often referred to as the *forward Euler* method.

by evaluating the integrand at the beginning of the interval.

Given that the trajectory $\boldsymbol{x}(t)$ satisfies:

$$\boldsymbol{x}(t_{k+1}) = \boldsymbol{x}(t_k) + \int_{t_k}^{t_{k+1}} h(\boldsymbol{x}(\tau), \tau) d\tau,$$

the Euler method approximates this integral by assuming $h(\boldsymbol{x}(\tau), \tau)$ is constant within the interval, yielding:

$$\boldsymbol{x}(t_{k+1}) \approx \boldsymbol{x}(t_k) + \Delta t \cdot h(\boldsymbol{x}(t_k), t_k),$$

where $\Delta t = t_{k+1} - t_k$ is the time step.

This approximation corresponds to a first-order Taylor series expansion:

$$\begin{aligned} \boldsymbol{x}(t + \Delta t) &\approx \boldsymbol{x}(t) + \Delta t\, \dot{\boldsymbol{x}}(t), \\ &= \boldsymbol{x}(t) + \Delta t \cdot h(\boldsymbol{x}(t), t), \end{aligned} \tag{1.31}$$

which treats the rate of change $\dot{\boldsymbol{x}}(t)$ as constant across the interval.

Alternatively, the Euler method can be interpreted as a finite difference approximation of the time derivative:

$$\dot{\boldsymbol{x}}(t) \approx \frac{\boldsymbol{x}(t + \Delta t) - \boldsymbol{x}(t)}{\Delta t}.$$

While computationally inexpensive, the Euler method has limited accuracy due to its reliance on information from the beginning of each interval. The local truncation error[21] is of order $\mathcal{O}(\Delta t^2)$, and errors can accumulate significantly over long trajectories unless small time steps are used.

[21] That is, the error introduced in a single time step.

As a concrete illustration, consider the running example introduced in Example 1.4.1. A simple implementation of Euler's method is presented in Algorithm 2.

Euler Method

```
def euler(h, x, t, Δt):
    return x + Δt * h(x, t)

x_euler = integrate(h, x0, t, euler)
```

Algorithm 2: Python implementation of Euler's method.

1.4.2 *Midpoint method*

The Midpoint method is a refinement of the Euler method that improves accuracy by evaluating the derivative at the midpoint of the time interval, rather than at its beginning. Recall that Euler's method approximates the next state using the derivative $\dot{\boldsymbol{x}}$ evaluated at time t:

$$\boldsymbol{x}(t + \Delta t) \approx \boldsymbol{x}(t) + \Delta t \cdot h(\boldsymbol{x}(t), t).$$

In contrast, the Midpoint method approximates the integral by using the value of the derivative at $t + \frac{\Delta t}{2}$:

$$\boldsymbol{x}(t + \Delta t) \approx \boldsymbol{x}(t) + \Delta t \cdot h\left(\boldsymbol{x}\left(t + \frac{\Delta t}{2}\right), t + \frac{\Delta t}{2}\right). \tag{1.32}$$

While this yields a more accurate estimate, it is not yet explicit, since the value $\boldsymbol{x}(t + \frac{\Delta t}{2})$ is not known in advance.

To resolve this, we approximate the midpoint using a single Euler step of size $\frac{\Delta t}{2}$:

$$\boldsymbol{x}\left(t + \frac{\Delta t}{2}\right) \approx \boldsymbol{x}(t) + \frac{\Delta t}{2} \cdot h(\boldsymbol{x}(t), t). \tag{1.33}$$

Substituting this estimate into Equation (1.32), we arrive at the explicit form of the Midpoint method:

$$\boldsymbol{x}(t + \Delta t) \approx \boldsymbol{x}(t) + \Delta t \cdot h\left(\boldsymbol{x}(t) + \frac{\Delta t}{2} \cdot h(\boldsymbol{x}(t), t),\ t + \frac{\Delta t}{2}\right).$$

The Midpoint method improves the local truncation error with respect to the Euler method from $\mathcal{O}(\Delta t^2)$ to $\mathcal{O}(\Delta t^3)$, providing significantly better accuracy for small step sizes. The improvement comes at the cost of computing the derivative twice per step—once at the start of the interval and once at its midpoint.

As an illustration, a simple implementation for the running example introduced in Example 1.4.1 is presented in Algorithm 3.

Midpoint Method

```
def midpoint(h, x, t, Δt):
    t_mid = t + Δt / 2
    x_mid = x + (Δt / 2) * h(x, t)
    return x + Δt * h(x_mid, t_mid)

x_midpoint = integrate(h, x0, t, midpoint)
```

Algorithm 3: Python implementation of the Midpoint method.

1.4.3 *Runge-Kutta-4 method*

Both the Euler and Midpoint methods approximate the change in $\boldsymbol{x}$ over a time step interval by evaluating the derivative $\dot{\boldsymbol{x}}$ at one or two specific points within the interval. The Runge-Kutta family of methods generalizes this idea by using multiple evaluations of the derivative to achieve higher accuracy. One of the most widely used methods in this family is the *fourth-order Runge-Kutta method*[22], which computes four derivative estimates over the interval $[t, t + \Delta t]$:

[22] Often abbreviated as RK4.

$$\boldsymbol{x}(t+\Delta t) \approx \boldsymbol{x}(t) + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

where:

$$\begin{aligned} k_1 &= h(\boldsymbol{x}(t),\, t), \\ k_2 &= h(\boldsymbol{x}(t) + \frac{\Delta t}{2}k_1,\, t + \frac{\Delta t}{2}), \\ k_3 &= h(\boldsymbol{x}(t) + \frac{\Delta t}{2}k_2,\, t + \frac{\Delta t}{2}), \\ k_4 &= h(\boldsymbol{x}(t) + \Delta t k_3,\, t + \Delta t). \end{aligned}$$

RK4 improves the local truncation error to $\mathcal{O}(\Delta t^5)$, offering significantly better accuracy than the Euler or Midpoint methods. This comes at the cost of evaluating $h(\boldsymbol{x}, t)$ four times per step, but the method remains computationally efficient and stable for many practical applications.

Using the running example from Example 1.4.1, an implementation of RK4 is shown in Algorithm 4.

RK4 Method

```
def rk4(h, x, t, Δt):
    k1 = h(x, t)
    k2 = h(x + (Δt / 2) * k1, t + Δt / 2)
    k3 = h(x + (Δt / 2) * k2, t + Δt / 2)
    k4 = h(x + Δt * k3, t + Δt)
    return x + (Δt / 6) * (k1 + 2 * k2 + 2 * k3 + k4)

x_rk4 = integrate(h, x0, t, rk4)
```

Algorithm 4: Python implementation of the RK4 method.

To compare the performance of these methods, we can visualize their outputs against the analytical solution:

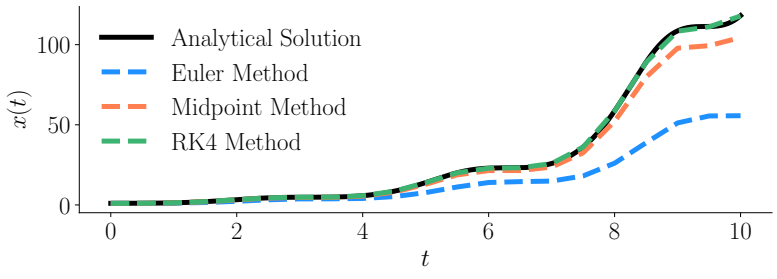


Figure 1.7: Visual comparison of various numerical integration methods and their approximations compared to the analytical solution for the initial value problem introduced in Example 1.4.1, with $\Delta t = 0.5$.

It is important to highlight that advanced simulation techniques extend far beyond these numerical integration methods, allowing for the high-fidelity simulation of robotic systems, including visualization in pixel space. These

techniques can incorporate detailed physical modeling, sensor data fusion, and learning-based approaches to create realistic and accurate simulations[23]. As we will discuss in later chapters on perception, these advancements are crucial for tasks such as robot training, planning under uncertainty, and evaluating the autonomy stack in novel and previously unseen scenarios.

[23] One notable example is the use of Neural Radiance Fields (NeRFs) for generating photorealistic scenes.

1.5 Summary

In this chapter, we introduced the fundamental principles underlying the modeling and simulation of robotic systems. We began by introducing state space models, which provide a mathematical framework to describe the evolution of a robot's state over time. Next, we discussed a robot's kinematics and dynamics, which characterize its motion and the constraints acting on it. This included a discussion of generalized coordinates and kinematic constraints—both holonomic and nonholonomic—along with the formulation of kinematic models using Pfaffian constraints. Throughout this chapter, we examined practical examples such as the rolling wheel, the pendulum, and wheeled robots like the unicycle and differential drive models. Finally, we introduced numerical integration techniques for simulating robot dynamics over time. We presented the Euler, Midpoint, and Runge-Kutta methods, highlighting their trade-offs and applications through concrete code examples.

To learn more. For readers interested in a deeper and more rigorous treatment of the concepts presented in this chapter, Siciliano and Khatib [60] and Siciliano et al. [62] offer comprehensive and widely adopted references. These texts cover the mathematical foundations of robot kinematics, dynamics, and control in greater depth, and provide additional examples, derivations, and exercises that complement and extend the material introduced here.

1.6 Exercises

The starter code for the exercises provided below is available online through GitHub. To get started, download the code by running in a terminal window:

```
git clone https://github.com/StanfordASL/pora-exercises.git
```

Problem 1: Numerical Integration Methods

In Example 1.4.1 we introduced a simple IVP and in Figure 1.7 we showed the differences between the Euler, midpoint, and fourth-order Runge-Kutta methods for solving it. In this exercise we will explore the use of a more advanced numerical integration scheme provided by the SciPy Python library called `odeint`.

Open the notebook `ch01/exercises/simulation.ipynb` and practice by implementing the dynamics models for a dampled pendulum and a bicycle, and then simulating them using `odeint`.

Problem 2: Nonholonomic Wheeled Robot Dynamics

The goal of this exercise is to familiarize yourself with some Python fundamentals that will be used throughout the book, such as Numpy and inheritance, as well as techniques for controlling nonholonomic wheeled robots. This exercise represents the start of an incremental journey to build your own robot autonomy stack.

Consider a simple robot with two wheels whose state is defined by the position of the center of the axle and the heading angle, shown in Figure 1.8. This robot's motion can be described by the simplest nonholonomic wheeled robot model, the unicycle model.

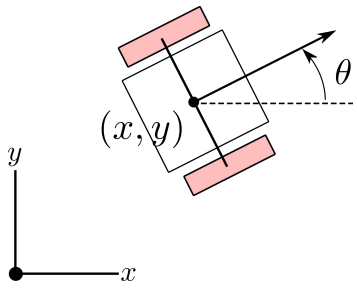


Figure 1.8: Generalized coordinates for a robot with unicycle kinematics.

The *kinematic* model we will use reflects the rolling without side-slip constraint, and is given below in Equation (1.34).

$$\begin{aligned} \dot{x}(t) &= v(t)\cos(\theta(t)), \\ \dot{y}(t) &= v(t)\sin(\theta(t)), \\ \dot{\theta}(t) &= \omega(t). \end{aligned} \tag{1.34}$$

In this model, the robot state is $\mathbf{x} = [x, y, \theta]^\top$, where $[x, y]^\top$ is the Cartesian location of the robot center and θ is its heading with respect to the x-axis. The robot control inputs are $\mathbf{u} = [v, \omega]^\top$, where v is the velocity along the main axis of the robot and ω is the angular velocity, subject to the control constraints:

$$|v(t)| \leq 0.75 \text{ m/s}, \qquad \text{and} \qquad |\omega(t)| \leq 1.0 \text{ rad/s}.$$

In this problem, we will demonstrate the use of class inheritance in Python classes and using Numpy for vectorized operations. The notebook associated with this problem is `ch01/exercises/nonholonomic_wheeled_robot_dynamics.ipynb`.

We will be using a `Dynamics` base class for two different dynamics models: the wheeled robot dynamics model in this exercise and a double integrator model in the next exercise. The base class contains two unimplemented functions: `feed_forward` and `rollout`. The `feed_forward` function will propagate the dynamics a single time step with disturbances, and the `rollout` function will apply the `feed_forward` function multiple times to retrieve a trajectory of states over multiple time steps. Because the feed-forward dynamics are subject to disturbances, the same control sequence will result in different trajectories. We will observe this by executing multiple rollouts of the dynamics using the same control sequence from the same initial state.

In the `ch01/exercises/nonholonomic_wheeled_robot_dynamics.ipynb` notebook, fill in the `RobotDynamics` class, in function `feed_forward` using discrete-time Euler integration, with the kinematic equations described in Equation (1.34). Then in the same class, fill in function `rollout` with two `for`-loops,

calling the `feed_forward` function. Run the cells that rollout the robot's dynamics and plot the control and state trajectories (this code has been written for you).

Problem 3: Double Integrator Dynamics

In this exercise, we consider the double integrator dynamics model:

$$\begin{aligned} \dot{x}(t) &= v_x(t), \\ \dot{y}(t) &= v_y(t), \\ \dot{v}_x(t) &= a_x(t), \\ \dot{v}_y(t) &= a_y(t). \end{aligned} \tag{1.35}$$

In this model, the robot state is $\mathbf{x} = [x, y, v_x, v_y]^\top$ and the robot control inputs are $\mathbf{u} = [a_x, a_y]^\top$.

Notice that in the previous problem, we used a `for`-loop to rollout several trajectories of the robot's dynamics. In this problem, we will use the same base dynamics class for a `DoubleIntegratorDynamics` class, and use vectorization to reduce the number of `for`-loops needed to perform multiple rollouts. The notebook associated with this problem is `ch01/exercises/double_integrator_dynamics.ipynb`.

i. To reduce the number of `for`-loops needed to perform multiple rollouts, we will vectorize the feed-forward dynamics equations applied in the function `feed_forward`. **Write down the discrete time vectorized equations for a single dynamics step for multiple rollouts** and fill in the function `feed_forward` in the `DoubleIntegratorDynamics` class. Use notation $\mathbf{X}_t$ to denote the stacked state vectors from each rollout at timestep t, $\bar{\mathbf{A}}$ as the constructed matrix in the notebook code `A_stack`, and $\bar{\mathbf{B}}$ as the constructed matrix in the notebook code `B_stack`.

ii. Fill in the code in function `rollout` in the `DoubleIntegratorDynamics` class using the `feed_forward` function you just wrote. Note that you should only need one `for`-loop!

References

[42] T. Lozano Perez. "Spatial planning: a configuration space approach". In: *Autonomous Robot Vehicles*. 1990.

[44] K. M. Lynch and K. C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017. Chap. 8.

[59] M. D. Shuster. "Survey of attitude representations". In: *Journal of the Astronautical Sciences* 41.4 (1993), pp. 439–517.

[60] B. Siciliano and O. Khatib. *Springer Handbook of Robotics*. Springer-Verlag, 2007.

[61] B. Siciliano et al. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 2008. Chap. 7.

[62] B. Siciliano et al. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 2008.

[63] Bruno Siciliano et al. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 2008. Chap. 2.

2
Open-Loop Control & Trajectory Optimization

In Chapter 1, we introduced the state space model as a mathematical formulation for representing a robot's dynamics. These models, typically expressed as systems of differential equations, provide a foundational framework that describes how the state of a robot evolves over time in response to control inputs. In particular, we saw how a state space model can be derived from the robot's kinematics and dynamics, providing a compact yet expressive description of its physical motion. In this chapter, we turn to a fundamental question: *given a state space model of the robot, how can we determine the control inputs that will drive it to execute a desired behavior?*

As we will see throughout Chapters 2- 5, robots must transform high-level goals into precise physical actions. This process is often described hierarchically, spanning *decision-making*, *motion planning*, *trajectory optimization*, and *control* (Figure 2.1). Each layer plays a distinct role, yet they remain deeply interconnected.

At the higher level of this hierarchy, decision-making governs what tasks the robot should perform to fulfill its objectives. It involves reasoning over goals, resources, and constraints, often under uncertainty. In a self-driving car, for example, this may correspond to deciding when to overtake, yield, or reroute. Because it involves strategic considerations rather than immediate actuation, decision-making typically unfolds on the order of seconds to minutes. Ultimately, decision-making defines the high-level objectives that guide subsequent layers of the hierarchy.

Once a high-level decision has been made, motion planning determines how to realize it in the robot's physical environment. This is typically expressed in terms of the robot's configuration space and involves finding a collision-free path that respects geometric and kinematic constraints. For instance, motion planning may compute a path for a mobile robot to navigate a cluttered warehouse without collisions. The timescale of motion planning is often on the order of hundreds of milliseconds to seconds.

Building on this path, trajectory optimization refines it into a time-parameterized trajectory that is dynamically feasible and optimized for performance criteria. This entails solving continuous optimization problems that incorporate dynamics, actuator limits, and objectives such as energy efficiency, comfort, or safety

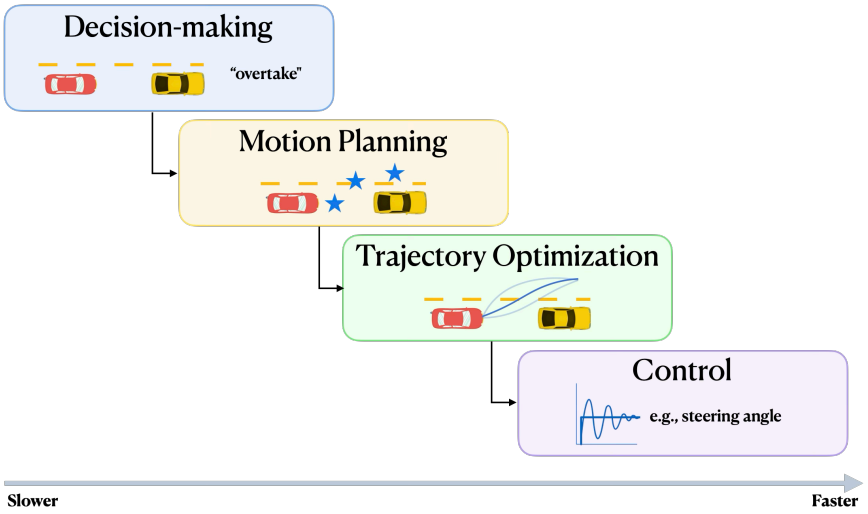


Figure 2.1: A hierarchical view of the relationship between decision-making, motion planning, trajectory optimization, and control.

margins. The result is a trajectory specifying both the robot's states and the control inputs needed to realize them over time, ensuring compatibility with the robot's actuation capabilities and dynamic constraints. Trajectory optimization typically operates on shorter timescales, from tens to hundreds of milliseconds.

Finally, low-level control ensures that the robot faithfully executes the desired trajectory in the physical world, by converting the trajectory into actuation commands. Controllers must operate at high frequency, applying feedback to correct deviations caused by disturbances, modeling errors, or sensor noise. Whether adjusting wheel torques, joint forces, or thrust vectors, control is what closes the loop between higher-level plans and physical reality.

Together, these layers form the backbone of an autonomous system: decision-making provides strategic guidance, motion planning translates that guidance into feasible paths, trajectory optimization refines those paths into feasible and optimal trajectories, and control ensures that the robot can follow those trajectories in the real world. In practice, the boundaries between these layers are often blurred. Trajectory optimization, for instance, may be tightly integrated with planning or even embedded within control loops, while high-level decisions may be informed by the lower-level processes. For the purposes of this book, we will adopt the hierarchical perspective outlined above, while acknowledging that real-world systems frequently combine or intertwine these processes.

In this chapter, we focus on trajectory optimization as a fundamental tool for computing trajectories that are both feasible and optimal. While control and motion planning will be revisited in Chapter 3 and Chapters 4- 5, respectively, our emphasis in this chapter is on the formulation and solution of the trajectory optimization problem. We begin in Section 2.1 by introducing the trajectory optimization problem and casting it as a continuous optimization problem. Building on this foundation, Sections 2.2 and 2.3 present two major classes of solution strategies—indirect methods, which derive optimality conditions

for the continuous problem using tools from the calculus of variations, and direct methods, which discretize and numerically solve the problem as a finite-dimensional nonlinear program. Finally, in Section 2.4, we explore specialized techniques tailored to certain problem structures, known as differentially flat systems.

2.1 The Optimal Control Problem

Optimal control theory aims to determine control inputs that drive a dynamical system to satisfy its physical constraints while optimizing a performance criterion. At a high level, formulating an optimal control problem requires three key components:

- A *mathematical model* of the system, typically expressed in state space form.
- A description of the *physical constraints* the system must satisfy.
- A specification of the *performance criterion* to be optimized.

Mathematical Model As discussed extensively in Chapter 1, the purpose of a mathematical model is to describe how the system's state evolves over time in response to control inputs. Using the notation from Chapter 1, the system dynamics can be expressed as a set of ordinary differential equations:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}\left(\boldsymbol{x}(t), \boldsymbol{u}(t)\right), \tag{2.1}$$

where $\boldsymbol{x}(t) \in \mathbb{R}^n$ is the state of the system at time t, $\boldsymbol{u}(t) \in \mathbb{R}^m$ is the control input, and $\boldsymbol{f} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ defines the state evolvution over time. Throughout this book, we will often refer to $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ as the state and control *sequences*, respectively, or equivalently as the state and control *trajectories*.

Physical Constraints Once we define the system dynamics, the next step is to specify the physical constraints that the system must satisfy. These constraints can take several forms, including:

- *Initial conditions*, which specify the state at the initial time t_0 as $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$.
- *Final conditions*, which specify the state at the final time t_f as $\boldsymbol{x}(t_f) = \boldsymbol{x}_f$ or $\boldsymbol{x}(t_f) \in \mathcal{X}_f$, where $\mathcal{X}_f$ denotes a set of allowable terminal states.
- *State constraints*, which require that the state remains within an allowable set $\mathcal{X}$ for all times $t \in [t_0, t_f]$, that is, $\boldsymbol{x}(t) \in \mathcal{X}$.
- *Control constraints*, which enforce that the control input remains within an allowable set $\mathcal{U}$ for all times $t \in [t_0, t_f]$, that is, $\boldsymbol{u}(t) \in \mathcal{U}$.

Depending on whether the constraints are satisfied or not, we can define the concept of *admissibility* for a control history and state trajectory.

Definition 2.1.1 (Admissible State and Control Sequences)**.** A state trajectory $\boldsymbol{x}(t)$ and a control sequence $\boldsymbol{u}(t)$ are admissible if they satisfy the state and control constraints at all times, that is:

$$\boldsymbol{x}(t) \in \mathcal{X} \quad \text{and} \quad \boldsymbol{u}(t) \in \mathcal{U}, \quad \forall t \in [t_0, t_f],$$

respectively.

Admissibility is a key concept in optimal control, as it restricts the set of realizable trajectories. In practice, this allows numerical methods to focus exclusively on admissible state and control sequences, rather than considering all possible solutions.

Performance Criterion The final component of an optimal control problem is the performance criterion to be optimized. An optimal control is defined as one that minimizes (or maximizes) this performance criterion. In some cases, the performance criterion may be implicitly defined by the problem statement (e.g., minimizing the time to reach a goal state), whereas in other cases, it must be explicitly designed (e.g., driving a car in a way that is comfortable for the passengers).

Throughout this book, we will focus on performance criteria that can be expressed as a cost functional[1] of the form:

$$J(\boldsymbol{x}(t), \boldsymbol{u}(t), t) = h(\boldsymbol{x}(t_f), t_f) + \int_{t_0}^{t_f} g(\boldsymbol{x}(t), \boldsymbol{u}(t), t)\, dt, \tag{2.2}$$

where $h : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ is the *terminal* cost and $g : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}$ is the *running* cost. The terminal cost h is evaluated at the final time t_f and typically represents a cost associated with the state of the system at that time, such as a penalty for being far from a desired goal state. The running cost g is integrated over the time interval $[t_0, t_f]$ and represents the instantaneous cost incurred by the system at each time step, for example, the cost of energy consumption, or any other cost associated with the system's operation. Depending on the problem, the final time t_f my be finite and fixed, finite and free, or infinite[2].

[1] A functional maps functions to real numbers; intuitively, we might say that a functional is a "function over functions". Here, the cost functional maps a state trajectory and control sequence to a real number representing the overall cost.

[2] In the case of an infinite final time, the terminal cost h is typically ignored and set to zero.

2.1.1 *Problem Formulation*

As a result, an optimal control problem can be formulated as follows:

Determine an admissible control sequence $\boldsymbol{u}^*(t)$ such that the system dynamics

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

generate a corresponding admissible state trajectory $\boldsymbol{x}^*(t)$ that minimizes the performance criterion:

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

where $\boldsymbol{u}^*(t)$ and $\boldsymbol{x}^*(t)$ are referred to as the *optimal control sequence* and *optimal state trajectory*, respectively.

This problem can be formally posed as the following optimization problem:

$$\begin{aligned} \underset{\boldsymbol{u}(t)}{\text{minimize}} \quad & J(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \\ \text{subject to} \quad & \dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \\ & \boldsymbol{x}(t_0) = \boldsymbol{x}_0, \quad \boldsymbol{x}(t_f) = \boldsymbol{x}_f \\ & \boldsymbol{u}(t) \in \mathcal{U}, \quad \boldsymbol{x}(t) \in \mathcal{X}, \quad t_0 < t < t_f. \end{aligned} \tag{2.3}$$

This general formulation serves as the starting point for the optimal control methods developed in the remainder of this book. There are also several important attributes related to the solution of this problem that are worth highlighting:

1. *Existence:* A solution to an optimal control problem is not guaranteed to exist; there may be no control history that is both admissible and optimal.

2. *Uniqueness:* Even when a solution exists, it may not be unique. Multiple admissible control inputs can yield the same performance. While this can pose challenges for numerical algorithms, it also provides flexibility in selecting among equally good solutions depending on the application.

3. *Optimality:* The objective of optimal control is to find a control sequence that outperforms *all* other admissible candidates. Thus, optimal control is interested in *global* optimality, as opposed to *local* optimality.

Example 2.1.1 (Autonomous Racing Optimal Control)**.** Consider an autonomous racing scenario in which the goal is to complete a lap of a known course in the shortest possible time. We can formulate this as a finite-horizon optimal control problem, where the objective is to minimize the final time t_f required to reach a designated goal position $(x_{\text{goal}}, y_{\text{goal}})$, subject to the constraint that the vehicle must remain on the track at all times. Let $\mathcal{X}$ denote the set of admissible states that correspond to positions on the course.

Suppose the vehicle is modeled using the simple kinematic car model from Equation (1.29), with state $\boldsymbol{x} = [x, y, \theta]^\top$ representing position and heading, and

control inputs $\boldsymbol{u} = [v, \phi]^\top$ representing forward speed and steering angle, respectively. The resulting optimal control problem is:

$$\begin{aligned} \underset{v(t),\phi(t)}{\text{minimize}} \quad & t_f \\ \text{s.t.} \quad & \dot{x} = v\cos\theta, \quad \dot{y} = v\sin\theta, \quad \dot{\theta} = \frac{v}{L}\tan\phi, \\ & \boldsymbol{x} \in \mathcal{X}_{\text{course}}, \quad \boldsymbol{u} \in \mathcal{U}, \\ & \boldsymbol{x}(t_0) = \boldsymbol{x}_0, \quad (x(t_f), y(t_f)) = (x_{\text{goal}}, y_{\text{goal}}). \end{aligned}$$

In this formulation, the cost depends only on the final time and not directly on the state or control at intermediate points. As a result, the time-optimal solution $(\boldsymbol{x}(t), \boldsymbol{u}(t))$ will lie on the boundaries of the admissible control and state sets. In practice, this means the vehicle will operate at full throttle and steer at the physical limits to cut the lap time, a strategy that achieves optimality mathematically—but may not make for a smooth or comfortable ride.

Throughout this book, we will see that the solution to the optimal control problem can take different forms, depending on whether and how it incorporates feedback from the current state of the system. At the most fundamental level, we distinguish between *open-loop* and *closed-loop* control. While closed-loop control will be the focus of Chapter 3, this chapter addresses open-loop control.

Open-Loop Control If the optimal control is computed purely as a function of time for a given initial state,

$$\boldsymbol{u}^*(t) = \ell(\boldsymbol{x}(t_0), t), \tag{2.4}$$

it is said to be in *open-loop* form. In the context of trajectory optimization, restricting the focus to open-loop strategies is natural, as they balance computational efficiency—computing open-loop sequences is faster than computing closed-loop policies—with effectivness, since robustness can be endowed through closed-loop tracking or by re-optimizing the trajectory in a receding-horizon fashion, as in Model Predictive Control[3].

[3] Discussed further in Chapter 3.

Having introduced the optimal control problem, we now turn to methods for computing optimal open-loop solutions. Fundamentally, Problem (2.3) is an *infinite-dimensional* optimization problem, where the optimization variables are functions of time. In practice, solution methods must rely on discretization stretegies, thereby approximating the infinite-dimensional problem with a finite-dimensional one. Different discretization approaches give rise to distinct families of methods.

Broadly speaking, two main classes of methods exist: *indirect methods* and *direct methods*. *Indirect methods* follow an "optimize-then-discretize" paradigm: they first derive necessary conditions for optimality—typically in the form of boundary-value problems involving adjoint variables—and then apply numerical techniques to solve these conditions. In contrast, *direct methods* take a

"discretize-then-optimize" approach: the state and control sequences are parameterized using finite-dimensional representations, and the resulting finite-dimensional optimization problem is solved numerically.

Beyond these two general classes, some systems admit further structural simplifications. In particular, *differentially flat systems* allow trajectories to be described in terms of a small set of variables that fully capture the system's evolution[4]. This property enables efficient trajectory generation and optimization, making these systems especially relevant in mobile robotics and aerospace applications.

[4] These variables are commonly known as *flat outputs*.

The remainder of this chapter examines these three classes of methods in detail: indirect methods in Section 2.2, direct methods in Section 2.3, and trajectory optimization for differentially flat systems in Section 2.4.

2.2 Indirect Methods

Indirect methods provide a principled framework for solving optimal control problems by drawing on ideas from calculus of variations (CoV)[5]. For a comprehensive treatment of the calculus of variations and its applications in optimal control theory, we refer the reader to Kirk [31]. At their core, indirect methods rely on the derivation of necessary optimality conditions (NOCs) that any solution must satisfy and then leverage numerical techniques to compute solutions consistent with these conditions. In this way, the NOCs serve as the bridge between the continuous-time formulation of an optimal control problem and its numerical resolution.

[5] Calculus of variations extends the principles of classical calculus from functions to functionals. The central idea is to study how small perturbations—called *variations*—of a candidate function influence the value of the functional. By analyzing the first- and higher-order effects of these variations, one can derive necessary conditions for optimality.

Before turning to the derivation of such conditions for *infinite*-dimensional optimization problems, let us first review key concepts from *finite*-dimensional optimization, which will serve as a foundation for the discussion ahead.

2.2.1 NOCs for Unconstrained Nonlinear Optimization Problems

Consider the following *finite*-dimensional optimization problem:

$$\min_{x \in \mathbb{R}^n} \quad f(x), \tag{2.5}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is assumed continuously differentiable, i.e., $f \in C^1$. We wish to identify the NOCs that any minimizer—local or global—must satisfy.

The key intuition is that at a local minimizer, no infinitesimal perturbation of the decision variable should decrease the objective. Formally, this requires analyzing how f changes under small variations around a candidate minimizer x^*.

First-order necessary condition. Let $x^* \in \mathbb{R}^n$ be a local minimizer. If $f \in C^1$, we can use gradients and Taylor series expansions to characterize the behavior of f near x^*. For a small perturbation Δx, the cost variation is, up to first order:

$$f(x^* + \Delta x) - f(x^*) \approx \nabla f(x^*)^\top \Delta x.$$

If $\boldsymbol{x}^*$ is a local minimizer, then for sufficiently small Δx, the first-order term must be non-negative[6]:

$$\nabla f(\boldsymbol{x}^*)^\top \Delta x = \sum_{i=1}^{n} \frac{\partial f(\boldsymbol{x}^*)}{\partial x_i} \Delta x_i \geq 0.$$

[6] This is because, if we were to decrease the cost by perturbing $\boldsymbol{x}^*$ by Δx, then $\boldsymbol{x}^*$ would not be a local minimizer.

In particular, by taking $\Delta \boldsymbol{x}$ to be positive and negative multiples of the coordinate unit vectors (i.e., vectors having all components equal to zero except for one component equal to one), we obtain simultaneously:

$$\frac{\partial f(\boldsymbol{x}^*)}{\partial x_i} \geq 0 \quad \text{and} \quad \frac{\partial f(\boldsymbol{x}^*)}{\partial x_i} \leq 0, \qquad i = 1, \ldots, n,$$

which forces

$$\frac{\partial f(\boldsymbol{x}^*)}{\partial x_i} = 0, \qquad i = 1, \ldots, n,$$

or, more compactly:

$$\nabla f(\boldsymbol{x}^*) = 0.$$

Thus, any local minimizer $\boldsymbol{x}^*$ must be a stationary point of f.

Second-order necessary condition. Assuming $f \in C^2$, consider again the Taylor expansion of f around a local minimizer $\boldsymbol{x}^*$, this time up to second order:

$$f(\boldsymbol{x}^* + \Delta x) - f(\boldsymbol{x}^*) \approx \nabla f(\boldsymbol{x}^*)^\top \Delta x + \frac{1}{2} \Delta x^\top \nabla^2 f(\boldsymbol{x}^*) \Delta x.$$

For $\boldsymbol{x}^*$ to be a local minimizer, the second-order variation must be nonnegative for all sufficiently small $\Delta \boldsymbol{x}$, that is:

$$\nabla f(\boldsymbol{x}^*)^\top \Delta x + \frac{1}{2} \Delta x^\top \nabla^2 f(\boldsymbol{x}^*) \Delta x \geq 0.$$

Using the first-order condition $\nabla f(\boldsymbol{x}^*) = 0$, the linear term vanishes, leaving:

$$\Delta x^\top \nabla^2 f(\boldsymbol{x}^*) \Delta x \geq 0.$$

Thus, the Hessian $\nabla^2 f(\boldsymbol{x}^*)$ must be positive semidefinite at any local minimizer.

Theorem 2.2.1 (Necessary Conditions for Unconstrained Local Minimizers)**.** *Let $\boldsymbol{x}^*$ be a local minimizer of $f : \mathbb{R}^n \to \mathbb{R}$. If $f \in C^1$ in an open set containing $\boldsymbol{x}^*$, then*

$$\nabla f(\boldsymbol{x}^*) = 0 \quad \textit{(first-order NOC)}. \tag{2.6}$$

If, in addition, $f \in C^2$, then

$$\nabla^2 f(\boldsymbol{x}^*) \succeq 0 \quad \textit{(second-order NOC)}. \tag{2.7}$$

2.2.2 NOCs for Constrained Nonlinear Optimization Problems

Having introduced the NOCs for unconstrained problems, this section extends the discussion to optimization problems subject to constraints. The definition of optimality conditions in the constrained setting requires the introduction of auxiliary variables, known as *Lagrange multipliers*. These variables are associated with the constraints and facilitate the characterization of optimal solutions while providing insights into the sensitivity of the optimal cost with respect to perturbations in the constraints. In this section, we limit our discussion on the theory of Lagrange multipliers to the case of equality constrained optimization. For a comprehensive treatment of optimality conditions in finite-dimensional optimization, the reader is referred to Bertsekas [5].

Consider the following constrained optimization problem:

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^n} \quad & f(\boldsymbol{x}) \\ \text{s.t.} \quad & h_i(\boldsymbol{x}) = 0, \quad i = 1, \dots, m, \end{aligned} \tag{2.8}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $h_i : \mathbb{R}^n \to \mathbb{R}$ are continuously differentiable.

For compactness, define the constraint function $h : \mathbb{R}^n \to \mathbb{R}^m$ as

$$h(\boldsymbol{x}) = (h_1(\boldsymbol{x}), \dots, h_m(\boldsymbol{x}))^\top, \tag{2.9}$$

so that the constraints can be written simply as $h(\boldsymbol{x}) = 0$.

The *Lagrange multiplier theorem* for equality-constrained optimization states that, if $\boldsymbol{x}^*$ is a local minimizer, then there exist scalars $\lambda_1^*, \dots, \lambda_m^*$, called *Lagrange multipliers*, such that:

$$\nabla f(\boldsymbol{x}^*) + \sum_{i=1}^{m} \lambda_i^* \nabla h_i(\boldsymbol{x}^*) = 0. \tag{2.10}$$

To interpret this condition, observe that the cost gradient $\nabla f(\boldsymbol{x}^*)$ must be orthogonal to the subspace of *first-order feasible variations*:

$$V(\boldsymbol{x}^*) \coloneqq \{\Delta\boldsymbol{x} \mid \nabla h_i(\boldsymbol{x}^*)^\top \Delta\boldsymbol{x} = 0, \quad i = 1, \dots, m\}.$$

This subspace consists of all variations $\Delta\boldsymbol{x}$ that preserve feasibility to first order [7]. Thus, condition (2.10) ensures that the first-order cost variation $\nabla f(\boldsymbol{x}^*)^\top \Delta\boldsymbol{x} = 0$ for all $\Delta\boldsymbol{x} \in V(\boldsymbol{x}^*)$. This statement is analogous to the $\nabla f(\boldsymbol{x}^*) = 0$ condition of unconstrained optimization.

[7] That is, variations for which $\boldsymbol{x} = \boldsymbol{x}^* + \Delta\boldsymbol{x}$ satisfies $h(\boldsymbol{x}) = 0$ to first order.

Formally, the necessary conditions for equality constrained optimality are summarized as follows:

Theorem 2.2.2 (Lagrange Multiplier Theorem — Necessary Conditions for Equality Constrained Local Minimizers)**.** *Let $\boldsymbol{x}^*$ be a local minimizer of $f : \mathbb{R}^n \to \mathbb{R}$ subject to the equality constraints $h_i(\boldsymbol{x}) = 0$, $i = 1, \dots, m$, and assume the constraint gradients $\nabla h_1(\boldsymbol{x}^*), \dots, \nabla h_m(\boldsymbol{x}^*)$ are linearly independent. Then there exist a unique vector $[\lambda_1^*, \dots, \lambda_m^*]^\top$, called the Lagrange multiplier vector, such that:*

$$\nabla f(\boldsymbol{x}^*) + \sum_{i=1}^{m} \lambda_i^* \nabla h_i(\boldsymbol{x}^*) = 0. \tag{2.11}$$

It is often convenient to express these conditions using the *Lagrangian function* $L : \mathbb{R}^{n+m} \to \mathbb{R}$ defined as:

$$L(\boldsymbol{x}, \boldsymbol{\lambda}) \coloneqq f(\boldsymbol{x}) + \sum_{i=1}^{m} \lambda_i h_i(\boldsymbol{x}). \tag{2.12}$$

The first-order NOCs for a local minimum $\boldsymbol{x}^*$ then take the compact form:

$$\begin{aligned} \nabla_{\boldsymbol{x}} L(\boldsymbol{x}^*, \boldsymbol{\lambda}^*) &= 0, \\ \nabla_{\boldsymbol{\lambda}} L(\boldsymbol{x}^*, \boldsymbol{\lambda}^*) &= 0, \end{aligned} \tag{2.13}$$

where $\nabla_{\boldsymbol{x}} L$ and $\nabla_{\boldsymbol{\lambda}} L$ denote the gradients with respect to $\boldsymbol{x}$ and $\boldsymbol{\lambda}$, respectively, and where the system in (2.13) consists of $n + m$ equations in $n + m$ unknowns—namely, the n components of $\boldsymbol{x}^*$ and the m components of $\boldsymbol{\lambda}^*$.

In practice, optimality conditions serve as a powerful tool to *filter* candidate solutions for global or local minima and often form the foundation of numerical optimization algorithms. For instance, in the unconstrained case of Problem (2.5), one might (i) find all stationary points by solving $\nabla f(\boldsymbol{x}) = 0$, and (ii) apply the second-order test by checking $\nabla^2 f(\boldsymbol{x}) \succeq 0$ at each candidate. This same philosophy extends naturally to infinite-dimensional optimal control problems, where any candidate solution must satisfy the corresponding NOCs. However, as we move to infinite-dimensional problems, the nature of the NOCs changes significantly: rather than yielding algebraic equations as in the finite-dimensional case, the NOCs for optimal control take the form of differential equations.

2.2.3 *Pontryagin's Minimum Principle*

Extending the concept of necessary optimality conditions to infinite-dimensional problems leads to Pontryagin's Minimum Principle (PMP), a cornerstone of optimal control theory. Specifically, the PMP generalizes the finite-dimensional NOCs to the infinite-dimensional setting.

Consider the problem of finding an admissible control $\boldsymbol{u}^*(t) \in \mathcal{U}$ that drives the system:

$$\dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t), t), \tag{2.14}$$

along a trajectory that minimizes the cost functional

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

To derive the NOCs, we define the *Hamiltonian*, the analog of the Lagrangian in finite-dimensional optimization:

$$H(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}(t), t) \coloneqq g(\boldsymbol{x}(t), \boldsymbol{u}(t), t) + \boldsymbol{p}(t)^\top f(\boldsymbol{x}(t), \boldsymbol{u}(t), t), \tag{2.15}$$

where $\boldsymbol{p}(t) \in \mathbb{R}^n$ is the *costate*[8] vector.

Similarly to the finite-dimensional case, where necessary conditions for optimality are derived by considering the cost increment $\Delta f = f(\boldsymbol{x} + \Delta \boldsymbol{x}) - f(\boldsymbol{x})$

[8] The term *costate* highlights that there is one costate associated with each state variable, analogous to Lagrange multipliers in finite-dimensional optimization.

in response to a perturbation $\Delta \boldsymbol{x}$, here we analyze the increment ΔJ under *variations* around a candidate function.

Theorem 2.2.3 (Pontryagin's Minimum Principle; for a comprehensive treatment, we refer the reader to Chapter 5 in Kirk [31]). *Let $\boldsymbol{u}^*(t)$ be an optimal control with associated state trajectory $\boldsymbol{x}(t)$ for the system in* (2.14) *over $[t_0, t_f]$. Then there exists a costate vector $\boldsymbol{p}^*(t)$ such that, for all $t \in [t_0, t_f]$, the following conditions hold:*

$$\begin{aligned} \dot{\boldsymbol{x}}^*(t) &= \frac{\partial H}{\partial \boldsymbol{p}}(\boldsymbol{x}^*(t), \boldsymbol{u}^*(t), \boldsymbol{p}^*(t), t), \\ \dot{\boldsymbol{p}}^*(t) &= -\frac{\partial H}{\partial \boldsymbol{x}}(\boldsymbol{x}^*(t), \boldsymbol{u}^*(t), \boldsymbol{p}^*(t), t), \\ \boldsymbol{u}^*(t) &= \arg\min_{\boldsymbol{u} \in \mathcal{U}} H(\boldsymbol{x}^*(t), \boldsymbol{u}, \boldsymbol{p}^*(t), t), \end{aligned} \tag{2.16}$$

along with the boundary conditions:

$$\begin{aligned} &\left[\frac{\partial h}{\partial \boldsymbol{x}}\left(\boldsymbol{x}^*(t_f), t_f\right) - \boldsymbol{p}^*(t_f)\right]^\top \delta \boldsymbol{x}_f \\ &+ \left[H(\boldsymbol{x}^*(t_f), \boldsymbol{u}^*(t_f), \boldsymbol{p}^*(t_f), t_f) + \frac{\partial h}{\partial t}\left(\boldsymbol{x}^*(t_f), t_f\right)\right] \delta t_f = 0, \end{aligned} \tag{2.17}$$

where $\delta \boldsymbol{x}_f$ and δt_f denote the variations of the final state and time, respectively [9].

[9] As we will discuss in the remainder of this section, the boundary conditions in Equation (2.17) depend on whether the final state and time are fixed (i.e., $\delta \boldsymbol{x}_f = 0$ or $\delta t_f = 0$) or free (i.e., $\delta \boldsymbol{x}_f$ or δt_f are arbitrary).

Equations (2.16) constitute the necessary conditions for optimality. They form a system of $2n$ first-order differential equations—n for the state and n for the costate—together with m algebraic equations defining the control input. Solving these equations produces $2n$ constants of integration. Half of these constants are determined by the initial conditions $\boldsymbol{x}^*(t_0) = \boldsymbol{x}_0$. The remaining n (or $n+1$, if the final time is free) are specified by the boundary conditions in Equation (2.17). This results in a two-point boundary value problem, which may be solved analytically in special cases, or numerically using methods such as shooting or collocation[10].

[10] J. Hertling. "Numerical Methods for Two-Point Boundary Value Problems (Herbert B. Keller)". In: *SIAM Review* 12.2 (1970), pp. 313–315

In practice, once the initial state is fixed, the boundary conditions are obtained by substituting the appropriate assumptions into Equation (2.17). Common cases include:

Fixed Final Time and Fixed Final State. If both t_f and $\boldsymbol{x}(t_f)$ are fixed, then $\delta t_f = 0$ and $\delta \boldsymbol{x}_f = 0$, leaving the sole boundary condition:

$$\boldsymbol{x}^*(t_f) = \boldsymbol{x}_f.$$

Fixed Final Time and Free Final State. If t_f is fixed but $\boldsymbol{x}(t_f)$ is free, then $\delta t_f = 0$ while $\delta \boldsymbol{x}_f$ is arbitrary. Hence, the boundary condition is:

$$\frac{\partial h}{\partial \boldsymbol{x}}\left(\boldsymbol{x}^*(t_f), t_f\right) - \boldsymbol{p}^*(t_f).$$

Free Final Time and Fixed Final State. If $\boldsymbol{x}(t_f)$ is fixed but t_f is free, then $\delta \boldsymbol{x}_f = 0$ while δt_f is arbitrary. Thus, the boundary condition is:

$$H(\boldsymbol{x}^*(t_f),\, \boldsymbol{u}^*(t_f),\, \boldsymbol{p}^*(t_f),\, t_f) + \frac{\partial h}{\partial t}(\boldsymbol{x}^*(t_f),\, t_f) = 0.$$

Free Final Time and Free Final State. If both $\boldsymbol{x}(t_f)$ and t_f are free, then $\delta \boldsymbol{x}_f$ and δt_f are arbitrary, and both coefficients in Equation (2.17) must be set to zero. That is:

$$\frac{\partial h}{\partial \boldsymbol{x}}(\boldsymbol{x}^*(t_f),\, t_f) - \boldsymbol{p}^*(t_f) = 0, \quad (n \text{ equations})$$
$$H(\boldsymbol{x}^*(t_f),\, \boldsymbol{u}^*(t_f),\, \boldsymbol{p}^*(t_f),\, t_f) + \frac{\partial h}{\partial t}(\boldsymbol{x}^*(t_f),\, t_f) = 0, \quad (1 \text{ equation}).$$

While these four cases cover many problems of practical interest, more general boundary conditions can be found in Kirk [31].

2.2.4 *Solving a Two-Point Boundary Value Problem*

Finding solutions that satisfy the necessary optimality conditions in Equation (2.16) is a nontrivial task, as these must simultaneously satisfy a system of $2n$ differential equations together with boundary conditions imposed at both t_0 and t_f. This type of problem, where conditions are specified at two distinct points in time, is known as a *two-point boundary value problem* (TPBVP).

Over the years, a number of numerical procedures have been developed for solving TPBVPs. Two broad classes of approaches are commonly used:

- **Shooting methods**, which reformulate the TPBVP as an initial value problem by guessing the unknown boundary conditions (e.g., the initial costate), simulating the system forward, and then iteratively adjusting the guess until the terminal boundary conditions are satisfied. Although conceptually straightforward, shooting methods may suffer from numerical instability, especially for long time horizons.
- **Collocation methods**, whereby the solution is approximated by a parametric function with unknown parameters at a set of discrete points (called collocation points). These methods turn the TPBVP into a large system of nonlinear algebraic equations that can be solved using computational techniques. Collocation methods are robust and widely used in practice because they avoid the instability issues of shooting methods.

Modern scientific computing environments provide high-level implementations of these ideas. For example, the `scikits.bvp_solver` package in Python or the function `bvp4c` in MATLAB implement numerical algorithms for solving TPBVPs with relatively little effort from the user.

Most solvers assume that the system of necessary conditions in Equation (2.16), along with its boundary conditions, can be expressed in the standard form:

$$\dot{\boldsymbol{z}} = g(\boldsymbol{z},\, t), \quad l(\boldsymbol{z}(t_0),\, \boldsymbol{z}(t_f)) = 0, \tag{2.18}$$

where $\boldsymbol{z}(t)$ collects the unknown functions (such as states and costates), g encodes their dynamics, and l encodes the two-point boundary constraints.

To illustrate how TPBVPs can be solved in practice, consider the toy dynamics:

$$\dot{\boldsymbol{z}}(t) = \begin{bmatrix} \dot{z}_1(t) \\ \dot{z}_2(t) \end{bmatrix} = \begin{bmatrix} z_2(t) \\ -z_1(t) \end{bmatrix},$$

with boundary conditions $z_1(t_0) = 0$ and $z_1(t_f) = -2$. The boundary conditions can equivalently be expressed in standard form as:

$$l(\boldsymbol{z}(t_0), \boldsymbol{z}(t_f)) = \begin{bmatrix} z_1(t_0) \\ z_1(t_f) + 2 \end{bmatrix} = 0.$$

In Python, the system dynamics $g(\boldsymbol{z}, t)$ and boundary conditions $l(\boldsymbol{z}(t_0), \boldsymbol{z}(t_f))$ can be passed directly to `solve_bvp` as shown in Algorithm 5.

Many optimal control problems can, in fact, be cast into the standard TPBVP form in Equation (2.18) and solved directly with off-the-shelf BVP solvers such as `solve_bvp`, sometimes after simple reformulations. Common cases include problems with conditions at special points, such as free-end problems, switching points, interface points, or discontinuities. Example 2.2.1 illustrates these ideas in the context of a free-final-time problem.

Example 2.2.1 (Free Final Time Optimal Control Problem; see Example 6.1 in How [23])**.** Consider the double integrator system

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

where $x \in \mathbb{R}$ is the state and $u \in \mathbb{R}$ is the control input. The control objective is to find a trajectory that minimizes the cost:

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

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 is a free final time problem with fixed boundary conditions on the state. The cost penalizes both the duration of the maneuver (through the αt_f^2 term) and the control effort (through the integral of u^2), with the trade-off governed by the weights α and β.

We can equivalently express the double integrator dynamics as a first-order system of differential equations by setting $x_1 = x$ and $x_2 = \dot{x}$:

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

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

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

Solving a TPBVP with `solve_bvp`

```python
from scipy.integrate import solve_bvp
import numpy as np

# Dynamics: ż = g(z,t)
def g(t, z):
    return np.vstack((z[1], -z[0]))

# Boundary conditions: l(z(t_0),z(t_f)) = 0
def l(z0, zf):
    return np.array([z0[0], zf[0] + 2])

# Time mesh and initial guess for z(t)
t_mesh = np.linspace(0, 4, 5)
z_guess = np.zeros((2, t_mesh.size))

# Solve TPBVP
sol = solve_bvp(g, l, t_mesh, z_guess)
z_sol = sol.sol(np.linspace(0, 4, 100))
```

Algorithm 5: Example usage of `solve_bvp` for a TPBVP in standard form. The code for this example is available in the repository `github.com/StanfordASL/pora-exercises` in the notebook `ch02/tpbvp.ipynb`.

The Hamiltonian is given by:

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

where p_1 and p_2 are the costate variables. Next, we construct the NOCs from Equation (2.16) by taking the partial derivatives of H with respect to p, x, and u:

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

Thus, from the last condition, the optimal control satisfies

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

Since this is a free-final-time problem with fixed terminal state, the boundary

conditions for the NOCs are given by:

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

However, the necessary conditions obtained above do not immediately match the "standard" form required by numerical TPBVP solvers in Equation (2.18), which assumes fixed final time. To cast the problem into standard form, one can apply the *time-scaling* strategy. First, the time horizon is rescaled to the fixed interval $[0, 1]$ by using the scaled time variable $\tau = t/t_f$. Next, the derivatives must be adjusted according to the new time variable. By the chain rule, differentiation with respect to τ introduces a scaling factor, i.e., $\frac{\partial}{\partial \tau} := \frac{\partial}{\partial t}\frac{\partial (\tau t_f)}{\partial \tau} = \frac{\partial}{\partial t} t_f$. Finally, the final time t_f is replaced by an auxiliary state variable r with trivial dynamics $\dot{r} = 0$. This results in a TPBVP with fixed final time (i.e., equal to 1) and an additional state variable r that encodes the original final time t_f.

For this example, the time-scaled cost becomes:

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

with corresponding Hamiltonian:

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

As a result, the necessary conditions for optimality become:

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

with boundary conditions:

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

After this reformulation, the problem adheres to the standard form and can be solved numerically.

For a systematic treatment of how nonstandard boundary value problems can be reformulated into standard form suitable for general-purpose solvers, see Ascher and Russell [3]. For a practical implementation of TPBVP solvers for free-final time optimal control problems, we refer the reader to the notebook `ch02/free_final_time_optimal_control.ipynb` in the repository `github.com/StanfordASL/pora-exercises`.

2.3 *Direct Methods*

So far, we introduced indirect methods, which involve deriving the necessary optimality conditions of the continuous-time optimal control problem and then discretizing them to numerically solve the resulting two-point boundary value problem. While indirect methods provide deep theoretical insights, they can be challenging to apply in practice due to the difficulty of solving boundary value problems, particularly for complex nonlinear dynamics or large-scale systems.

Direct methods take the opposite approach. Rather than deriving the optimality conditions analytically, the problem is discretized first. This reduces the continuous-time optimal control problem to a finite-dimensional nonlinear optimization problem, which can then be solved using general-purpose numerical optimization algorithms.

The process of converting the continuous-time optimal control problem into a discretized form amenable to numerical optimization is known as *transcription*. While there exist many different transcription methods, a simple and widely used approach is the *forward Euler discretization*. This method selects a discretization $0 = t_0 < t_1 < \ldots < t_N = t_f$ of the time interval $[0, t_f]$ and approximates the state and control sequences assuming a zero-order hold on both the states and control inputs, meaning that both the state and the control input are constant over each time interval $[t_i, t_{i+1})$. The system dynamics are then integrated forward using Euler integration:

$$\boldsymbol{x}_{i+1} \approx \boldsymbol{x}_i + h_i \boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{u}_i), \quad h_i = t_{i+1} - t_i. \tag{2.19}$$

Direct methods are typically grouped into two main families:

- *State and control parameterization methods (also known as direct collocation methods)*: here, both the control inputs and the state trajectories are discretized, and the dynamics are introduced explicitly as algebraic constraints linking the state and control variables at each discretization point (e.g., trapezoidal and Hermite-Simpson collocation, Gauss-Lobatto methods, etc.).
- *Control parameterization methods (also known as direct shooting methods)*: in this approach, only the control inputs are discretized, and the state trajectories are obtained by numerically integrating the system dynamics forward in time. As a result, the optimization variables are solely the discretized controls, while the states are implicitly defined by the integration of the dynamics (e.g., using single or multiple shooting techniques).

In what follows, we illustrate the fundamental concepts of both families of methods through a concrete example.

Example 2.3.1 (Zermelo's Problem - Continuous-time problem)**.** Consider the problem of steering a boat from a point $(0,0)$ to a point (M,ℓ) in a river with a current. The boat can be controlled by adjusting its direction and its speed is constant. The dynamics of the boat are described by the following differential equations:

$$\begin{aligned}\dot{x}(t) &= v\cos(u(t)) + \text{flow}(y(t)), \quad t \in [0, t_f],\\ \dot{y}(t) &= v\sin(u(t)), \quad t \in [0, t_f],\end{aligned}$$

where $(x(t), y(t))$ is the position of the boat with x defining the coordinate along the river and y the coordinate across the river, $u(t)$ is the control input (the direction of the boat), and v is the constant speed of the boat. For simplicity, assume that the river flow is described by an arbitrary function acting in the x-direction, with its intensity depending on the position $y(t)$, i.e., $\text{flow}(y(t))$.

The objective is to minimize the control effort over time, which can be formulated as an optimal control problem:

$$\begin{aligned}\underset{u(t)}{\text{minimize}} \quad & \int_0^{t_f} u(t)^2\, dt,\\ \text{subject to} \quad & \dot{x}(t) = v\cos(u(t)) + \text{flow}(y(t)), \quad t \in [0, t_f],\\ & \dot{y}(t) = v\sin(u(t)), \quad t \in [0, t_f],\\ & (x(0), y(0)) = (0,0),\\ & \Big(x(t_f), y(t_f)\Big) = (M,\ell),\\ & |u(t)| \le u_{\max}, \quad t \in [0, t_f].\end{aligned}$$

2.3.1 *Direct Collocation Methods*

Consider the Problem introduced in Example 2.3.1. Applying a state and control parametrization (i.e., collocation) method leads to the following finite-dimensional nonlinear program, equivalently described in Algorithm 6:

$$\begin{aligned}\underset{(x,y,u)}{\text{minimize}} \quad & \sum_{i=0}^{N-1} h_i u_i^2,\\ \text{subject to} \quad & x_{i+1} = x_i + h_i\left(v\cos(u_i) + \text{flow}(y_i)\right), \quad i = 0,\dots,N-1,\\ & y_{i+1} = y_i + h_i v\sin(u_i), \quad i = 0,\dots,N-1,\\ & (x_0, y_0) = (0,0),\\ & (x_N, y_N) = (M,\ell),\\ & |u_i| \le u_{\max}, \quad i = 0,\dots,N-1.\end{aligned}$$

In this formulation, both the state and control trajectories are discretized and treated as decision variables. The system dynamics are not enforced through numerical integration, but rather as algebraic equality constraints linking consecutive discretization points.

2.3.2 *Direct Shooting Methods*

Many of the concepts introduced for state and control parametrization carry over to control parametrization methods, with a key distinction in how the dynamics are handled. In control parametrization (shooting) methods, the optimization variables consist only of the control inputs at each discretization point. The state trajectory is not explicitly optimized but is instead computed recursively by forward simulation of the system dynamics. In other words, a candidate sequence of controls uniquely determines the corresponding states, which are then used to evaluate the cost and any state constraints.

Concretely, let us revisit Zermelo's problem from Example 2.3.1, this time using a shooting method transcription. In this formulation, the control inputs $\{u_i\}_{i=0}^{N-1}$ are treated as the optimization variables, while the states $\{(x_i, y_i)\}_{i=0}^{N-1}$ are computed recursively from the dynamics.

The resulting finite-dimensional optimization problem is:

$$
\begin{aligned}
\underset{u}{\text{minimize}} \quad & \sum_{i=0}^{N-1} h_i u_i^2, \\
\text{subject to} \quad & (x_N, y_N) = (M, \ell), \\
& |u_i| \leq u_{\max}, \quad i = 0, \ldots, N-1, \\
& \text{where, recursively,} \\
& x_{i+1} = x_i + h_i \left(v \cos(u_i) + \text{flow}(y_i) \right), \quad i = 0, \ldots, N-1, \\
& y_{i+1} = y_i + h_i v \sin(u_i), \quad i = 1, \ldots, N-1.
\end{aligned}
$$

Here, the dynamics are no longer constraints in the optimization problem, but rather equations that implicitly determine the state evolution given a candidate control sequence. Algorithm 7 provides a Python implementation of this shooting method approach to Zermelo's problem.

Both approaches come with their own advantages and limitations. Control parameterization methods generally result in smaller optimization problems, making the method computationally attractive. The dynamics are enforced exactly through integration (up to the accuracy of the chosen numerical integrator), which is especially useful when the dynamics are complex or only accessible via a black-box simulator. However, state constraints may be difficult to enforce, as the states are not explicit optimization variables but rather implicitly defined through the integration of the dynamics. This can lead to numerical instability or infeasibility when state constraints are critical. Moreover, errors from numerical integration may accumulate, potentially reducing the accuracy of the solution.

On the other hand, state and control parametrization methods treat both states and controls as optimization variables. This allows state constraints to be imposed directly, improving numerical stability and robustness, often leading to better-conditioned optimization problems when constraints play a central role. However, the resulting optimization problem generally grows significantly

in size, since all states and controls at every discretization point are treated as decision variables. This higher dimensionality increases computational cost and can make the solver more sensitive to initial guesses.

In practice, both methods are widely used, and the choice between them often depends on the problem structure, the availability of simulators or system models, and the importance of accurately handling state constraints.

2.4 *Differentially Flat Systems*

Computing open-loop control sequences by directly solving optimal control problems can often be computationally intensive. In many applications, it is useful to trade off strict optimality[11] for computational tractability by seeking "good" trajectories that are simpler to compute, even if slightly sub-optimal.

[11] That is, the theoretical best performance according to a given cost functional.

For a special class of systems known as *differentially flat systems*, generating such feasible trajectories is considerably simpler. A system is differentially flat if there exists a set of outputs, called *flat outputs*, such that all system states and inputs can be expressed as algebraic functions of these outputs and a finite number of their derivatives. This property allows trajectory generation to be performed in the space of the flat outputs, eliminating the need to solve differential equations as part of the optimization process and greatly reducing computational complexity.

Differentially flat models arise in several common robotics applications, including simple car models, quadrotors, and many other wheeled or aerial vehicles. Their relative simplicity and expressiveness make them particularly attractive for trajectory planning and open-loop control synthesis.

Example 2.4.1 (Differentially Flat Autonomous Vehicle Control)**.** Recall the motion planning task from Example 2.1.1 where the objective was to compute an open-loop control sequence to drive a vehicle through a course to a goal position in minimum time. If we relax the requirement of optimality and instead aim simply to find a feasible trajectory that follows the course, we can exploit the differential flatness of the kinematic car model. Specifically, consider the kinematic car model from Equation (1.29):

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

where (x, y) is the vehicle position, θ is the heading, v is the speed, ϕ is the steering angle, and L is the wheelbase.

This system is differentially flat with flat outputs $(x(t), y(t))$. Therefore, it is sufficient to specify any differentiable trajectory for $x(t)$ and $y(t)$ that respects the course constraints. From these trajectories, the remaining state and control variables—which are the quntities needed for practical implementation—can be

computed analytically. The heading is obtained from the velocity direction as

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

and the speed along the trajectory can be computed using either component of the velocity:

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

Finally, the steering angle is determined from the heading dynamics:

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

In this way, a feasible trajectory for the vehicle can be generated entirely by specifying smooth flat output trajectories, from which all states and, importantly, the control inputs can be derived directly.

We formalize this concept through the notion of differential flatness.

Definition 2.4.1 (Differential Flatness; for a comprehensive treatment, we refer the reader to Murray [49])**.** A nonlinear system with state $\boldsymbol{x} \in \mathbb{R}^n$ and control $\boldsymbol{u} \in \mathbb{R}^m$:

$$\dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t)), \tag{2.20}$$

is *differentially flat* if there exists a function α such that

$$\boldsymbol{z} = \alpha(\boldsymbol{x}, \boldsymbol{u}, \dot{\boldsymbol{u}}, \ldots, \boldsymbol{u}^{(a)}), \tag{2.21}$$

where $\boldsymbol{u}^{(i)}$ denotes the i-th time derivative of $\boldsymbol{u}$, and such that the system trajectories can be expressed as functions of the flat output $\boldsymbol{z} \in \mathbb{R}^m$ and a finite number of its derivatives:

$$\begin{aligned} \boldsymbol{x} &= \beta(\boldsymbol{z}, \dot{\boldsymbol{z}}, \ldots, \boldsymbol{z}^{(b)}) \\ \boldsymbol{u} &= \gamma(\boldsymbol{z}, \dot{\boldsymbol{z}}, \ldots, \boldsymbol{z}^{(c)}). \end{aligned} \tag{2.22}$$

In other words, a system is said to be differentially flat if there exists a set of outputs $\boldsymbol{z}$ (with the same dimension as the input vector $\boldsymbol{u}$) that completely determine both the states and the inputs, without requiring integration of the system dynamics. For trajectory optimization, this property is particularly advantageous: since the evolution of a flat system is fully characterized by its flat outputs, trajectories can be computed directly in the output space and then mapped to the corresponding inputs, thereby avoiding expensive integration of the dynamics.

In the following sections, we explore explore different techniques to exploit differential flatness for open-loop trajectory design, including how to parameterize trajectories in the flat output space, handle initial and terminal state constraints, and enforce control constraints.

2.4.1 *Trajectory Parameterization*

Our primary limitation when planning a trajectory in the flat output space is that it must be *differentiable*. A common approach is to parameterize each component of the flat output $\boldsymbol{z}$ using N smooth basis functions:

$$z_j(t) = \sum_{i=1}^{N} \alpha_i^{[j]} \psi_i(t), \tag{2.23}$$

where z_j is the j-th element of $\boldsymbol{z}$, $\alpha_i^{[j]} \in \mathbb{R}$ are parameters that define the trajectory, and $\psi_i(t)$ are smooth basis functions.

Polynomial basis functions are a natural choice, e.g., $\psi_1(t) = 1$, $\psi_2(t) = t$, $\psi_3(t) = t^2$, etc. A key advantage of this parameterization is that $z_j(t)$ is linear in the variables $\alpha_i^{[j]}$, which facilitates translating constraints on $\boldsymbol{z}$ and its derivatives directly into constraints on the coefficients $\alpha_i^{[j]}$.

2.4.2 *Equality Constraints*

A key component of any open-loop motion planning problem is the enforcement of boundary conditions. Typically, this means ensuring that the system begins at a prescribed initial state $\boldsymbol{x}(0) = \boldsymbol{x}_0$ and often that it reaches a desired terminal state, $\boldsymbol{x}(t_f) = \boldsymbol{x}_f$, at some final time t_f. When planning in the flat output space, these state conditions must be expressed as constraints on the flat output $\boldsymbol{z}(t)$ and its derivatives. Recalling the mapping in Equation (2.22), this leads to

$$\begin{aligned} \boldsymbol{x}_0 &= \beta(\boldsymbol{z}(0), \dot{\boldsymbol{z}}(0), \ldots, \boldsymbol{z}^{(q)}(0)), \\ \boldsymbol{x}_f &= \beta(\boldsymbol{z}(t_f), \dot{\boldsymbol{z}}(t_f), \ldots, \boldsymbol{z}^{(q)}(t_f)). \end{aligned} \tag{2.24}$$

In practice, this means that boundary conditions on $z_j(0), \dot{z}_j(0), \ldots, z_j^{(q)}(0)$ and $z_j(t_f), \dot{z}_j(t_f), \ldots, z_j^{(q)}(t_f)$ must be enforced. When using a smooth basis function parameterization of the form in Equation (2.23), these conditions translate directly into algebraic constraints on the coefficients $\alpha_i^{[j]}$. By differentiating Equation (2.23) q times, we obtain

$$\begin{aligned} \dot{z}_j(t) &= \sum_{i=1}^{N} \alpha_i^{[j]} \dot{\psi}_i(t), \\ &\vdots \\ z_j^{(q)}(t) &= \sum_{i=1}^{N} \alpha_i^{[j]} \psi_i^{(q)}(t). \end{aligned} \tag{2.25}$$

which allows us to express the boundary conditions as a system of linear equa-

tions:

$$\begin{bmatrix} \psi_1(0) & \psi_2(0) & \dots & \psi_N(0) \\ \dot{\psi}_1(0) & \dot{\psi}_2(0) & \dots & \dot{\psi}_N(0) \\ \vdots & \vdots & & \vdots \\ \psi_1^{(q)}(0) & \psi_2^{(q)}(0) & \dots & \psi_N^{(q)}(0) \\ \psi_1(t_f) & \psi_2(t_f) & \dots & \psi_N(t_f) \\ \dot{\psi}_1(t_f) & \dot{\psi}_2(t_f) & \dots & \dot{\psi}_N(t_f) \\ \vdots & \vdots & & \vdots \\ \psi_1^{(q)}(t_f) & \psi_2^{(q)}(t_f) & \dots & \psi_N^{(q)}(t_f) \end{bmatrix} \begin{bmatrix} \alpha_1^{[j]} \\ \alpha_2^{[j]} \\ \vdots \\ \alpha_N^{[j]} \end{bmatrix} = \begin{bmatrix} z_j(0) \\ \dot{z}_j(0) \\ \vdots \\ z_j^{(q)}(0) \\ z_j(T) \\ \dot{z}_j(t_f) \\ \vdots \\ z_j^{(q)}(t_f) \end{bmatrix}. \tag{2.26}$$

Assuming the matrix formed by the basis functions has a sufficient number of columns and that it is full column rank, we can solve for (possibly non-unique) $\alpha_i^{[j]}$ that solve the trajectory generation problem.

More generally, any equality constraint on the flat outputs or their derivatives—beyond just initial and terminal states—can be written in this linear form. For instance, waypoints can be added as additional equality constraints. However, if too many constraints are imposed, the system may become overdetermined, leaving no feasible solution. In such cases, one must increase the richness of the basis functions (e.g., higher-order polynomials or additional functions), which improves flexibility but also increases computational complexity.

2.4.3 *Inequality Constraints via Time Scaling*

Having addressed equality constraints in Section 2.4.2, we now turn to inequality constraints. These commonly arise in motion planning and control to enforce actuator limits or safety bounds on the state. For example, the simple car model from Example 2.4.1 may have a speed constraint of the form:

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

A useful tenchnique for handling such constraints in the flat-output space is *time scaling*. The idea is to first plan a trajectory that satisfies the equality constraints (e.g., by solving Equation (2.26)), and then adjust its temporal evolution—speeding up or slowing down along the path—to enforce the inequality constraints.

Formally, let $\boldsymbol{x}(t)$ be a trajectory satisfying the equality constraints. We can separate the *geometric path*[12] from its timing by introducing a path parameter $s(t)$:

$$\boldsymbol{x}(t) = \boldsymbol{x}(s(t)),$$

with $s(0) = s_0$, $s(t_f) = s_f$, and $\dot{s}(t) > 0$[13]. The geometric path $\boldsymbol{x}(s)$ captures the sequence of states, while the choice of $s(t)$ determines how quickly the system traverses that path. Varying $s(t)$ is referred to as *time scaling*.

[12] The geometric path of a trajectory is the sequence of states $\boldsymbol{x}$ of the trajectory, but not associated with a particular time

[13] The condition $\dot{s}(t) > 0$ ensures invertibility, so each t corresponds to a unique s.

Example 2.4.2 (Time Scaling for a Simple System)**.** Consider a scalar system with state $x \in \mathbb{R}$ and a straight-line path connecting an initial and terminal

state, x_0 and x_f:

$$x(s) = x_0 + s(x_f - x_0), \quad s \in [0, 1].$$

Choosing a cubic polynomial for $s(t)$,

$$s(t) = \frac{3}{T^2}t^2 - \frac{2}{T^3}t^3, \quad t \in [0, T],$$

which satisfies $s(0) = 0$, $s(T) = 1$, and $\dot{s}(t) > 0$, yields the temporal trajectory

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

Here, T controls the duration of the trajectory. If we impose a velocity bound,

$$|\dot{x}| \leq \dot{x}_{\max},$$

then

$$\dot{x} = 6\left(\frac{t}{T^2} - \frac{t^2}{T^3}\right)(x_f - x_0),$$
$$\ddot{x} = 6\left(\frac{1}{T^2} - \frac{2t}{T^3}\right)(x_f - x_0),$$

with the maximum velocity attained at $t = \frac{T}{2}$. We can then convert this into a constraint on T to ensure the inequality constraint is satisfied:

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

Example 2.4.2 illustrates the key idea: inequality constraints can often be transformed into conditions on the timing law $s(t)$, without altering the geometric path itself.

For general state-space systems, time scaling is often more complex. Given a feasible trajectory $(\boldsymbol{x}(t), \boldsymbol{u}(t))$ of the system dynamics in Equation (2.20), we can rewrite it as a geometric path $(\boldsymbol{x}(s), \boldsymbol{u}(s))$ using a path parameter $s(t)$:

$$\frac{d\boldsymbol{x}(s)}{ds}\frac{ds(t)}{dt} = f(\boldsymbol{x}(s), \boldsymbol{u}(s)). \tag{2.27}$$

For time scaling, we replace $s(t)$ with a new path parameter $\tilde{s}(t)$ over a possibly different interval $t \in [0, \tilde{t}_f]$, with $\tilde{s}(0) = s_0$ and $\tilde{s}(\tilde{t}_f) = s_f$[14]. The new scaling must still satisfy the dynamics:

$$\frac{d\boldsymbol{x}(\tilde{s})}{d\tilde{s}}\frac{d\tilde{s}(t)}{dt} = f(\boldsymbol{x}(\tilde{s}), \boldsymbol{u}(\tilde{s})). \tag{2.28}$$

Since the geometric path is fixed—as it was previously defined—the terms $\frac{d\boldsymbol{x}(\tilde{s})}{d\tilde{s}}$ and $\boldsymbol{x}(\tilde{s})$ are also fixed. Therefore, time scaling with a new path parameter $\tilde{s}(t)$, is only admissible if an appropriate $\tilde{\boldsymbol{u}}(\tilde{s})$ can be found. Fortunately, for many systems—including those commonly studied in motion planning—this is possible with the right choice of $\tilde{s}(t)$.

[14] The geometric path is still defined on the interval $[s_0, s_f]$, which must remain the same for any new time scaling law.

Example 2.4.3 (Time Scaling for the Simple Car Model)**.** Consider again the simple car model from Equation (1.29):

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

and suppose we have identified a candidate trajectory $\boldsymbol{x}_c(t)$ with control $\boldsymbol{u}_c(t)$ by leveraging the differential flatness of the model through Equation (2.26) and mapping the flat outputs $\boldsymbol{z}_c(t)$ into the state and control space.

For this model, a natural choice for the path parameter s is the arc-length, defined as:

$$s(t) = \int_0^t v(\tau)\mathrm{d}\tau.$$

such that $\dot{s}(t) = v(t) > 0$. With this choice, the geometric path $\boldsymbol{x}_c(s)$ is defined over $s \in [0, L_{\text{path}}]$, where L_{path} is the total length of the path. Rewriting the dynamics in terms of an arbitrary time scaling $\tilde{s}(t)$ gives

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

which must hold for any admissible time scaling $\tilde{s}(t)$[15].

[15] The trivial choice $\tilde{s}(t) = s(t)$ reproduces the original candidate trajectory with control inputs $\boldsymbol{u}_c(t)$.

By adopting the arc-length parameterization, we have $\dot{\tilde{s}} = v(\tilde{s})$, so these equations reduce to

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

The first two equations are automatically satisfied for any choice of $\tilde{s} \in [s_0, s_f]$, since the original candidate trajectory satisfies the dynamics. On the other hand, the third equation is satisfied provided we reuse the same steering input, $\phi(\tilde{s}) = \phi_c(\tilde{s})$. Therefore, the dynamics remain consistent for any choice of time scaling $\tilde{s}(t)$: the geometric path is preserved, while the temporal evolution along that path is left free. This observation is powerful as we may freely adjust the speed input $v(t)$, subject only to $\dot{\tilde{s}}(t) > 0$, without altering the geometry of the trajectory. In practice, this allows us to easily enforce inequality constraints on the speed $|v(t)| \leq v_{\max}$.

Example 2.4.3 shows a relatively straightforward application of time scaling to a model derived from kinematic constraints. This idea extends naturally to a

wide class of kinematic models of the form

$$\dot{\boldsymbol{x}}(t) = G(\boldsymbol{x}(t))\boldsymbol{u}(t). \tag{2.29}$$

Applying the chain rule, we obtain

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

which can be rewritten as

$$\frac{d\boldsymbol{x}(s)}{ds} = G(\boldsymbol{x}(s))\boldsymbol{u}_g(s), \tag{2.30}$$

where $\boldsymbol{u}_g(s) = \frac{\boldsymbol{u}(t)}{\dot{s}(t)}$ is the *geometric control*[16]. Equation (2.30) shows that the geometric path $\boldsymbol{x}(s)$ is fully determined by the geometric control $\boldsymbol{u}_g(s)$, independent of the time parametrization. Therefore, once the geometric control and geometric path are defined, we can temporally scale the trajectory $\boldsymbol{x}(t)$ using the path parameter $s(t)$ without changing the geometric path. The corresponding control inputs are recovered via $\boldsymbol{u}(t) = \dot{s}(t)\boldsymbol{u}_g(s)$.

[16] Since $s(t)$ must be strictly increasing, we require $\dot{s}(t) > 0$.

In summary, for models of the form (2.29), we can perform time scaling by:

1. Selecting a path parameter s (e.g., arc-length), computing $s(t)$ for the original trajectory $\boldsymbol{x}(t)$, and determining the interval $[s_0, s_f]$.
2. Re-parameterizing the control $\boldsymbol{u}(t)$ in terms of s.
3. Computing the geometric control $\boldsymbol{u}_g(s) = \boldsymbol{u}(s(t))/\dot{s}(t)$ for $s \in [s_0, s_f]$.
4. Defining a new path parameter function $\tilde{s}(t)$ over the interval $[0, \tilde{t}_f]$ with $\dot{\tilde{s}}(t) > 0$, $\tilde{s}(0) = s_0$, and $\tilde{s}(\tilde{t}_f) = s_f$.
5. Computing the new control inputs as $\tilde{\boldsymbol{u}}(t) = \boldsymbol{u}_g(\tilde{s}(t))\dot{\tilde{s}}(t)$ for all $t \in [0, \tilde{t}_f]$.

Example 2.4.4 (Time Scaling for the Unicycle Model)**.** Consider the kinematic unicycle model:

$$\begin{aligned} \dot{x} &= v\cos\theta, \\ \dot{y} &= v\sin\theta, \\ \dot{\theta} &= \omega, \end{aligned} \tag{2.31}$$

where (x, y) denote the position, θ the heading, v the forward velocity, and ω the rotation rate. We define the state as $\boldsymbol{x} = [x,\, y,\, \theta]^\top$ and the control as $\boldsymbol{u} = [v,\, \omega]^\top$.

A natural path parameter for this system is again the arc-length,

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

such that $\dot{s}(t) = v(t) > 0$. If the trajectory is defined over $t \in [0, T]$ with total length L_{path}, then $s(0) = 0$ and $s(T) = L_{\text{path}}$. The corresponding geometric

controls are

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

where the fact that $v_g(s) = 1$ follows directly from $\dot{s}(t) = v(s(t))$. Introducing a new timing law $\tilde{s}(t)$ generates a new velocity profile $\tilde{v}(\tilde{s}) = \dot{\tilde{s}}(t)$ along the path, which can use to solve for the new $\tilde{\omega}$ inputs by:

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

In practice, it is often simpler to directly prescribe a velocity profile $\tilde{v}(\tilde{s})$ along the path and compute the corresponding angular velocity $\tilde{\omega}(\tilde{s}) = \frac{\omega(\tilde{s})}{v(\tilde{s})}\tilde{v}(\tilde{s})$. Finally, to determine the new controls as functions of time, we note that:

$$\tau(s) = \int_0^s \frac{1}{\tilde{v}(s')} ds',$$

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

Example 2.4.5 (Planar Quadrotor Control)**.** Considers the control of a planar quadrotor system. The quadrotor is modeled with six state variables: horizontal position x, vertical position y, orientation angle ϕ, and their respective velocities. The control inputs are the thrusts T_1 and T_2 from the two rotors. The objective is to minimize the energy consumption, represented by the integral of the squared thrusts over time:

$$\min \int_0^{t_f} T_1(t)^2 + T_2(t)^2 dt$$

The system dynamics are given by the following differential equations:

$$\begin{bmatrix} \dot{x} \\ \dot{v}_x \\ \dot{y} \\ \dot{v}_y \\ \dot{\phi} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} v_x \\ -\frac{(T_1+T_2)}{m}\sin\phi \\ v_y \\ \frac{(T_1+T_2)}{m}\cos\phi - g \\ \omega \\ \frac{(T_2-T_1)\ell}{I_{zz}} \end{bmatrix}$$

where m is the mass, g is the gravitational acceleration, ℓ is the distance from the center of mass to each rotor, and I_{zz} is the moment of inertia about the z-axis.

This system is differentially flat, with flat outputs (x, y). For a practical implementation of differential flatness for trajectory generation applied to this system, refer to the notebook `ch02/differentially_flat_planar_quadrotor.ipynb` in the repository `github.com/StanfordASL/pora-exercises`.

2.5 Summary

In this chapter, we explored how trajectory optimization provides a fundamental framework for computing open-loop motions and establishing a foundation for autonomous decision-making in robotic systems.

We began by formalizing the optimal control problem—a mathematical formalization for the task of driving a system's state evolution through admissible control inputs while optimizing a performance criterion. This formulation involves three key components: the system's mathematical model, the physical constraints, and the performance criteria.

We then introduced two major families of methods for solving optimal control problems and compute optimal open-loop control sequences: direct methods and indirect methods. Indirect methods adopt an "optimize-then-discretize" approach, deriving analytical necessary conditions for optimality and solving the resulting two-point boundary value problem numerically. In contrast, direct methods follow a "discretize-then-optimize" strategy, transcribing the continuous problem into a finite-dimensional nonlinear program that can be solved with standard optimization solvers.

Lastly, we discussed differentially flat systems, a special class of systems for which trajectory generation is significantly simplified. For these systems, planning can be performed in a lower-dimensional "flat output" space using techniques like polynomial parameterization. We showed how initial, terminal, and waypoint constraints can be translated into linear algebraic equations, and how inequality constraints on state and control can be managed through time scaling.

To learn more. For comprehensive treatments of optimal control, we point the reader to several excellent references, including Murray [49], Kirk [31], Rao [54], and Kelly [30]. Kirk [31] offers a foundational perspective on indirect methods, including detailed derivations of the calculus of variations and Pontryagin's Minimum Principle. For a thorough exploration of direct methods and modern optimization-based approaches, Rao [54] and Kelly [30] provide in-depth coverage of transcription techniques, such as collocation and shooting, and their formulation as nonlinear programming problems. Finally, Murray [49] gives an extensive overview of differential flatness, illustrating how this property can be exploited for efficient trajectory generation.

2.6 Exercises

The starter code for the exercises provided below is available online through GitHub. To get started, download the code by running in a terminal window:

```
git clone https://github.com/StanfordASL/pora-exercises.git
```

We denote Problems requiring hand-written solutions and coding in Python with ✎ and 🖳, respectively.

✎ *Problem 1: Extremal Curves*

[This exercise is inspired by Kirk [31], Chapter 4, Problem 4.9]
Given the functional

$$J(x) = \int_0^1 \left(\frac{1}{2}\dot{x}(t)^2 + 5x(t)\dot{x}(t) + x(t)^2 + 5x(t) \right) dt,$$

find an extremal curve $x^* : [0, 1] \to \mathbb{R}$ that satisfies $x^*(0) = 1$ and $x^*(1) = 3$.

✎ *Problem 2: Minimum Control Effort*

Consider the dynamics

$$\dot{x}(t) = -2x(t) + u(t)$$

with the initial constraint $x(0) = 2$, terminal constraint $x(1) = 0$, and cost functional

$$J(u) = \int_0^1 u(t)^2 \, dt.$$

Write down the Hamiltonian and use the necessary optimality conditions to derive an optimal control $u^*(t)$ and corresponding state trajectory $x^*(t)$.

✎ *Problem 3: Zermelo's Ship*

Zermelo's ship must travel through a region of strong currents. The position of the ship is denoted by $(x(t), y(t)) \in \mathbb{R}^2$. The ship travels at a constant speed $v > 0$, yet its heading $\theta(t)$ can be controlled. The current moves in the positive x-direction with speed $w(y(t))$. The equations of motion for the ship are

$$\begin{aligned} \dot{x}(t) &= v \cos\theta(t) + w(y(t)) \\ \dot{y}(t) &= v \sin\theta(t) \end{aligned}.$$

We want to control the heading $\theta(t)$ such that the ship travels from a given initial position $(x(t_0), y(t_0)) = (x_0, y_0)$ to the origin $(0, 0)$ in minimum time.

1. Suppose $w(y(t)) = \frac{v}{h} y(t)$, where $h > 0$ is a known constant. Show that an optimal control law $\theta^*(t)$ must satisfy a linear tangent law of the form

   $$\tan\theta^*(t) = \alpha - \frac{v}{h} t$$

   for some constant $\alpha \in \mathbb{R}$.

2. Suppose $w(y(t)) \equiv \beta$ for some constant $\beta > 0$. Derive an expression for the optimal transfer time $t_1^* - t_0$.

Problem 4: Singular Arc for Dubins' Car

The kinematics of Dubins' car are described by

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

where $(x, y) \in \mathbb{R}^2$ is the car's position, $\theta \in \mathbb{R}$ is the car's heading, $v > 0$ is the car's constant known speed, and u is the controlled turn rate. The turn rate is bounded, i.e., $u \in [-\bar{\omega}, \bar{\omega}]$, where $\bar{\omega} > 0$ is a known constant.

The car starts at $(x, y) = (0, 0)$ with a heading of $\theta = 0$ at $t = 0$. We want the car to drive to $(x, y) = (0, c)$ in the least amount of time possible, where $c > 0$ is a given constant.

1. Use Pontryagin's maximum principle to express the optimal control input $u^*(t)$ as a function of the optimal co-state $p^*(t) := (p_x^*(t), p_y^*(t), p_\theta^*(t)) \in \mathbb{R}^3$.

   Hint: You should discover that the maximum condition for $u^*(t)$ is not informative whenever $p_\theta^*(t) \equiv \bar{p}_\theta$ for a particular fixed value $\bar{p}_\theta \in \mathbb{R}$. When such a lack of information persists over a non-trivial time interval, i.e., any time interval $[t_1, t_2]$ with $t_2 > t_1 \geq 0$, this is known as a *singular arc*. To compute $u^*(t)$ in this case, use the fact that $p_\theta^*(t) \equiv \bar{p}_\theta$ is constant in time along this singular arc.

2. Use boundary conditions to argue why $p^*(t)$ might end in a singular arc. Suppose we know $p^*(t)$ begins on a non-singular arc, then switches once to and ends on a singular arc. For this particular case, argue why $u^*(0) = \bar{\omega}$ and describe the optimal state trajectory $(x^*(t), y^*(t), \theta^*(t))$ and control trajectory $u^*(t)$ in words without explicitly deriving them.

Problem 5: Single Shooting for a Unicycle

Consider the kinematic model of a unicycle

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

where (x, y) is the planar position of the vehicle, θ is its heading angle, v is its forward velocity, and ω is its angular velocity. Overall, the state and control input for this system are $x := (x, y, \theta) \in \mathbb{R}^3$ and $u := (v, \omega) \in \mathbb{R}^2$, respectively. We have overloaded x to denote both horizontal position $x \in \mathbb{R}$ and the full state vector $x \in \mathbb{R}^3$.

Our task is to drive the vehicle from the starting configuration $x(0) = (0, 0, \pi/2)$ to the target configuration $x(T) = (5, 5, \pi/2)$ in minimum time with as little control effort as possible. To this end, we consider the objective

$$J(x, u) = \int_0^T \left(\alpha + v(t)^2 + \omega(t)^2\right) dt,$$

where $\alpha > 0$ is a chosen constant weighting factor and T is the free final time.

1. Derive the Hamiltonian and necessary optimality conditions, specifically
   (a) the ODE for the state and co-state,
   (b) the optimal control as a function of the state and co-state, and
   (c) the boundary conditions, including the additional condition for free final time T.

Hint: Since the control set is unbounded, use the weak maximum condition.

In practice, you might use a boundary value problem (BVP) solver from an existing computing library (e.g., `scipy.integrate.solve_bvp`), but in this problem we will use a bit of nonlinear optimization theory and JAX to write our own!

2. In the file `ch02/exercises/unicycle_optimal_control.ipynb`, complete the implementations of `dynamics`, `hamiltonian`, `optimal_control`, and `noc_ode`. Use $\alpha = 0.25$.

In the single shooting method, we need to initialize estimates of the initial co-state $p(0)$ and final time T. We then integrate the state and co-state dynamics forward in time from $t = 0$ to $t = T$, at which point we check whether the terminal boundary conditions are satisfied.

3. Use the ODE integration from `noc_trajectories` to complete `boundary_residual`, which should compute a measure of how far off each of your terminal boundary conditions is from satisfaction, given guesses for the initial co-state $p(0)$ and final time T.

4. Finally, in `newton_step` and `single_shooting`, implement the Newton-Raphson root-finding method for `boundary_residual`. Now, if you provide an appropriate guess for the initial costate and final time, you can solve the problem in `unicycle_optimal_control.ipynb` and see a plot of the optimal solution. You may find that whether or not your BVP solver converges to a solution is highly dependent on the quality of your initial guess—indeed, initialization is a major challenge when applying indirect methods for optimal control!

   Hint: For finding roots of a function $f : \mathbb{R}^n \to \mathbb{R}^n$, each iteration of the Newton-Raphson method entails improving a current best guess $x^{(k)}$ at iteration k using the update rule:

$$x^{(k+1)} = x^{(k)} - \frac{\partial f}{\partial x}(x^{(k)})^{-1} f(x^{(k)}).$$

References

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

[5] D. Bertsekas. *Nonlinear Programming*. Athena Scientific, 2016.

[22] J. Hertling. "Numerical Methods for Two-Point Boundary Value Problems (Herbert B. Keller)". In: *SIAM Review* 12.2 (1970), pp. 313–315.

[23] Jonathan P. How. *Lecture Notes for Principles of Optimal Control*. 2008.

[30] M. Kelly. "An Introduction to Trajectory Optimization: How to Do Your Own Direct Collocation". In: *SIAM Review* 59.4 (2017), pp. 849–904.

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

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

[54] A. Rao. "A Survey of Numerical Methods for Optimal Control". In: *Advances in the Astronautical Sciences* 135 (2010).

Collocation Formulation of Zermelo's Problem

```
# Decision variables: (x_i, y_i, u_i), i = 0,...,N
get_x = lambda z: z[:N + 1]
get_y = lambda z: z[N + 1:-N]
get_u = lambda z: z[-N:]
get_z = lambda x, y, u: np.concatenate([x, y, u])

# Cost: sum_{i=0}^{N-1} h_i u_i^2
cost = lambda z: np.sum(h * np.square(get_u(z)))

def constraints(z):
    x, y, u = get_x(z), get_y(z), get_u(z)
    constraints = []
    for i in range(N):
        # x_{i+1} = x_i + h_i (v cos(u_i) + flow(y_i))
        constraints.append(x[i+1] - x[i] - h*(v*np.cos(u[i])
                           + flow(y[i])))
        # y_{i+1} = y_i + h_i v sin(u_i)
        constraints.append(y[i+1] - y[i] - h*v*np.sin(u[i]))
    # Boundary conditions: (x_0, y_0) = (0,0), (x_N, y_N) = (M, l)
    constraints.extend([x[0], y[0], x[N] - M, y[N] - l])
    return np.array(constraints)

# State bounds
x_lower = np.zeros(N + 1)
x_upper = M * np.ones(N + 1)
y_lower = np.zeros(N + 1)
y_upper = l * np.ones(N + 1)

# Control bounds: u_i <= u_max;
u_lower = -u_max * np.ones(N) # control constraint
u_upper = u_max * np.ones(N)  # control constraint

bounds = Bounds(
        get_z(x_lower, y_lower, u_lower),
        get_z(x_upper, y_upper, u_upper))

# Solve the NLP
result = minimize(cost, z0, bounds=bounds,
                  constraints={'type': 'eq', 'fun': constraints})
```

Algorithm 6: Direct collocation approach to Zermelo's problem using forward Euler discretization. The code for this example is available in the repository github.com/StanfordASL/pora-exercises in the notebook ch02/zermelos_problem.ipynb.

Shooting Formulation for Zermelo's Problem

```
# Decision variables: (u_i), i = 0,...,N-1; Cost: sum_{i=0}^{N-1} h_i u_i^2
cost = lambda u: np.sum(h * np.square(u))

# States computed recursively from x_0 = 0, y_0 = 0
dynamics = lambda x, y, u: (
        x + h * (v * np.cos(u) + flow(y)),
        y + h * np.sin(u)
)

def inequality_constraints(u):
    x, y = 0, 0 # initial condition (x(0), y(0)) = (0, 0)
    constraints = []
    for ui in u:
        x, y = dynamics(x, y, ui)
        # (x_i, y_i) >= (0, 0) (box constraint with below)
        constraints.extend([x, y])
        # (x_i, y_i) <= [M, l]
        constraints.extend([M - x, l - y])
    # (x_N, y_N) >= [M, l] (enforcing equality with the above)
    constraints.extend([x - M, y - l])
    return constraints

bounds = Bounds(-u_max * np.ones(N),
                u_max * np.ones(N)) # |u_i| <= u_max

# Solve NLP
result = minimize(cost, u0, bounds=bounds,
                  constraints={'type': 'ineq',
                               'fun': inequality_constraints})
```

Algorithm 7: Direct shooting approach to Zermelo's problem using forward Euler discretization. The code for this example is available in the repository github.com/StanfordASL/pora-exercises in the notebook ch02/zermelos_problem.ipynb.

3
Closed-Loop Control & Trajectory Tracking

In Chapter 2, we introduced the concepts of *open-loop* and *closed-loop* control laws, and then explored techniques for designing open-loop control laws based on optimal control and differential flatness. Open-loop control laws are useful for determining nominal control inputs that accomplish different objectives, such as "move from point A to point B as quickly as possible", and are computationally less challenging that computing closed-loop control laws. However, open-loop control laws only leverage the initial state of the system and are therefore susceptible to unexpected or unmodeled disturbances. In contrast, closed-loop control laws are significantly more robust since they leverage real-time observations to compute the control input. We mathematically define a closed-loop control law[1] as a function of time and the current state.

[1] Closed-loop control laws are also sometimes referred to as *feedback controllers* or *control policies*.

Definition 3.0.1 (Closed-loop Control Law)**.** A closed-loop control law is a function that maps time and the current state[2] of the system to a control input:

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

[2] If the full system state is not directly measurable, we can define closed-loop control laws based on the current measured system outputs.

As an example, consider a wheeled robot trying to move from point to point. An open-loop control law could have poor performance in reaching the desired goal if the initial state is not perfectly known, if the dynamics model does not perfectly describe the robot's motion, or if external disturbances affect the system. Alternatively, a closed-loop control law will continuously attempt to correct for these errors by accounting for new information.

In this chapter, we introduce a few common approaches for designing closed-loop control laws. First, we introduce approaches for closed-loop feedback control for *linear dynamical systems*, including the *linear quadratic regulator (LQR)*, a well-known approach for closed-loop optimal control, and *proportional-integral-derivative (PID)* control, a classical control technique that is widely used across many domains of automation. Next, we discuss methods for nonlinear closed-loop control, including how techniques for linear systems can be applied to nonlinear systems through the process of *linearization*. Finally, we discuss how closed-loop controllers can be paired with open-loop controllers for *trajectory tracking*, which is a common robotics motion planning problem.

3.1 Linear Closed-loop Control

Consider the case where the system dynamics are linear:

$$\dot{\boldsymbol{x}}(t) = A\boldsymbol{x}(t) + B\boldsymbol{u}(t), \tag{3.2}$$

where $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times m}$ are time-invariant matrices. A fundamental closed-loop control task is to design a control law of the form in Equation (3.1) that will force the state, $\boldsymbol{x}$, to converge to the origin from some arbitrary initial state. We can accomplish[3] this by defining a *linear feedback controller* of the form:

$$\boldsymbol{u}(t) = K\boldsymbol{x}(t), \tag{3.3}$$

where $K \in \mathbb{R}^{m\times n}$ is a time-invariant *gain matrix*. The closed-loop dynamics under the linear feedback controller are:

$$\dot{\boldsymbol{x}}(t) = (A + BK)\boldsymbol{x}(t),$$

and therefore our objective is to choose the gain matrix K such that the matrix $A + BK$ is stable[4].

We can also use the linear feedback controller to force the system to converge to some constant setpoint, $(\boldsymbol{u}_d, \boldsymbol{x}_d)$, that is an equilibrium point[5] for the system. In this case, we can define a linear system for the *error dynamics*:

$$\delta\dot{\boldsymbol{x}}(t) = A\delta\boldsymbol{x}(t) + B\delta\boldsymbol{u}(t), \tag{3.4}$$

where $\delta\boldsymbol{x} = \boldsymbol{x} - \boldsymbol{x}_d$ and $\delta\boldsymbol{u} = \boldsymbol{u} - \boldsymbol{u}_d$. We then compute a controller gain matrix, K, to make the error dynamics converge to zero, and the resulting linear feedback controller is:

$$\boldsymbol{u}(t) = \boldsymbol{u}_d + K\delta\boldsymbol{x}(t). \tag{3.5}$$

[3] The linear system must be *controllable* or *stabilizable* in order to guarantee that convergence to the origin is achievable. A system can be determined to be controllable or stabilizable by a combined analysis of the system matrices A and B. Roughly speaking, a system is not controllable or stabilizable if the control matrix, B, doesn't give the control, $\boldsymbol{u}$, the authority to manipulate all unstable *modes* of the dynamics matrix, A.

[4] A matrix is called *stable* or *exponentially stable* if all of its eigenvalues have a negative real component.

[5] An equilibrium point for a linear system is a point that satisfies $0 = A\boldsymbol{x}_d + B\boldsymbol{u}_d$.

3.1.1 The Linear Quadratic Regulator (LQR)

Previously, we discussed how we can use a linear feedback controller of the form in Equation (3.3) to drive a linear system to converge to the origin. The *linear quadratic regulator (LQR)* is an optimal control technique and special case of the linear feedback control law that tries to drive the system to converge to the origin in an optimal way[6]. The finite-horizon LQR controller solves the closed-loop finite-horizon optimal control problem:

$$\begin{aligned} \min_{\boldsymbol{u}} \quad & \boldsymbol{x}(T)^\top F\boldsymbol{x}(T) + \int_0^T \boldsymbol{x}(t)^\top Q\boldsymbol{x}(t) + \boldsymbol{u}(t)^\top R\boldsymbol{u}(t)\mathrm{d}t \\ \text{s.t.} \quad & \dot{\boldsymbol{x}}(t) = A\boldsymbol{x}(t) + B\boldsymbol{u}(t), \end{aligned} \tag{3.6}$$

where T is a fixed final time, and the matrices $F \in \mathbb{R}^{n\times n}$, $Q \in \mathbb{R}^{n\times n}$, and $R \in \mathbb{R}^{m\times m}$ define the cost function[7]. We also require that F and Q are symmetric, positive semi-definite matrices and that R is a symmetric, positive definite matrix[8].

[6] Stabilizing the system about a particular state is referred to as *regulation*.

[7] The name linear quadratic regulator comes from the *linear* dynamics and the *quadratic* cost function.

[8] In practice, it is common for the matrices F, Q, and R to simply be diagonal matrices with positive diagonal elements.

The optimal solution to the finite-horizon LQR problem[9] is a linear feedback control law with a time-variant gain matrix:

$$\boldsymbol{u}(t) = K^*(t)\boldsymbol{x}(t),$$

where we compute $K^*(t)$ by:

$$K^*(t) = -R^{-1}B^\top P(t),$$

where $P(t)$ is a symmetric, positive definite matrix that solves the continuous time Riccati differential equation:

$$\dot{P}(t) = -A^\top P(t) - P(t)A + P(t)BR^{-1}B^\top P(t) - Q,$$

with the terminal condition $P(T) = F$.

Similarly, the *infinite-horizon* LQR controller solves the closed-loop optimal control problem:

$$\begin{aligned} \min_{\boldsymbol{u}} \quad & \int_0^\infty \boldsymbol{x}(t)^\top Q\boldsymbol{x}(t) + \boldsymbol{u}(t)^\top R\boldsymbol{u}(t)\mathrm{d}t \\ \text{s.t.} \quad & \dot{\boldsymbol{x}}(t) = A\boldsymbol{x}(t) + B\boldsymbol{u}(t), \end{aligned} \tag{3.7}$$

and the optimal feedback gain matrix, K^*, is *time-invariant* and is computed by:

$$K^* = -R^{-1}B^\top P,$$

where P is the time-invariant, symmetric, positive definite matrix that solves the continuous time alebraic Riccati equation[10]:

$$0 = A^\top P + PA - PBR^{-1}B^\top P + Q.$$

We can also formulate both the finite and infinite horizon LQR problems in discrete time and solve them using discrete versions of the Riccati equations.

3.1.2 *Proportional Integral Derivative (PID) Control*

Proportional integral derivative (PID) control is a classical control technique that is still widely used in various industries today. The core theory for PID control centers around the control of linear systems with a single input and a single output[11]. The control law takes the general form:

$$u(t) = k_p e(t) + k_i \int_0^t e(\tau)\mathrm{d}\tau + k_d \frac{de(t)}{dt}, \tag{3.8}$$

where $e(t) = r(t) - y(t)$ is the error between a target reference signal, $r(t)$, and the system output, $y(t)$. The values k_p, k_i, and k_d are proportional, integral, and derivative gains, respectively. We can see the relationship between the structure of the PID controller and the names we give to the gains: the proportional gain makes the control law proportional to the error, the integral gain scales the integral of the error, and the derivative gain scales the derivative of the error.

[9] Note that we can also apply these LQR results to the case where the linear dynamical system is time varying with system matrix $A(t)$ and control matrix $B(t)$.

[10] There are many open-source software tools that solve the continuous time Riccati differential equation and the algebraic Riccati equation.

[11] There are more advanced techniques for applying PID control to nonlinear system and systems with multiple inputs and outputs.

Double Integrator Infinite-horizon LQR

```python
import numpy as np
from scipy.linalg import solve_continuous_are

# System dynamics matrices for simple double integrator ẍ = u
A = np.array([[0, 1], [0, 0]])
B = np.array([[0], [1]])

# Define cost function matrices
Q = np.array([[1, 0], [0, 1]])
R = np.array([[1]])

# Solve the continuous algebraic Riccati equation
P = solve_continuous_are(A, B, Q, R)

# Compute optimal feedback gain matrix
K = -np.linalg.inv(R) @ B.T @ P

# Verify eigenvalues are negative (closed-loop system is stable)
eig_val, eig_vec = np.linalg.eig(A + B @ K)
print(eig_val)
```

Algorithm 8: Compute the optimal infinite-horizon LQR controller for a simple double integrator with dynamics $\ddot{x} = u$ in Python. The code for this example is available in the repository `github.com/StanfordASL/pora-exercises` in the notebook `ch03/lqr.ipynb`.

We can also see the structure of the controller in the block diagram shown in Figure 3.1. PID control is common in industry due to its simplicity, minimal computational needs, and tunability[12]. One disadvantage of PID control is that it doesn't directly leverage a model of the system and is therefore not the best approach to exploit the dynamics for optimal performance.

[12] Tuning PID controllers for desired performance is not a simple task, but some general approaches have been developed, such as the Ziegler-Nichols method.

Example 3.1.1 (PD Control of a Double-integrator System)**.** Consider a double-integrator system with dynamics:

$$\dot{\boldsymbol{x}}(t) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \boldsymbol{x}(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t).$$

where $x = [x_1, x_2]^\top$, which has two eigenvalues at the origin. Suppose we want to design a controller to drive the state x_1 to the origin, $x_1 = 0$. By using a PD controller of the form:

$$u(t) = k_p e(t) + k_d \frac{de(t)}{dt},$$

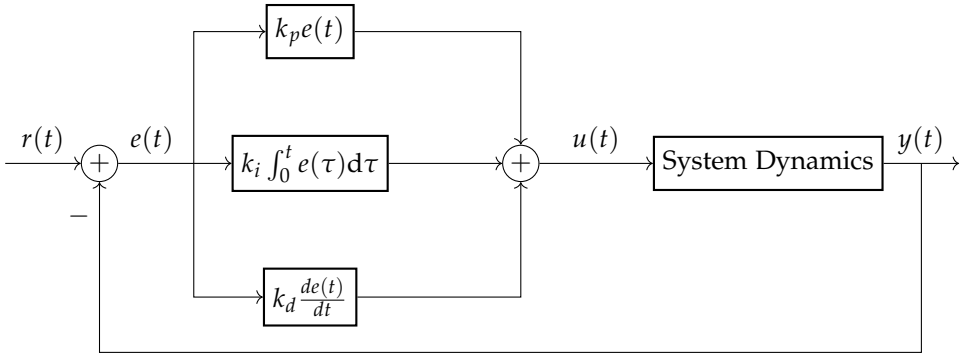


Figure 3.1: Block diagram for a PID controller in a feedback loop.

where $e(t) = x_1(t)$, the closed-loop dynamics of the system are:

$$\dot{\boldsymbol{x}}(t) = \begin{bmatrix} 0 & 1 \\ k_p & k_d \end{bmatrix} \boldsymbol{x}(t),$$

where we have used that $\dot{e} = \dot{x}_1 = x_2$. We can now choose the gains k_p and k_d to make the eigenvalues of the closed-loop dynamics matrix stable. The eigenvalues for the closed-loop system are:

$$\lambda = \frac{k_d}{2} \pm \frac{1}{2}\sqrt{k_d^2 - 4k_p},$$

so we should at least choose k_p and k_d to make the real part of the eigenvalues negative, which will ensure stability[13]. We could also consider tuning these values based on how much oscillation and overshoot is allowable, and how robust the controller should be to external disturbances. For example, in theory we could choose the magnitude of the gain k_p to be really large to drive fast convergence to the origin, but in practice this could amplify any noise and could make the controller behave poorly.

There are also many other important and useful tools from classical control theory beyond PID control. Classical control techniques typically perform their analysis in the frequency domain[14] rather than in the time domain, and will leverage tools such as *Bode plots* and the *Nyquist stability criterion*.

[13] Note that a proportional controller alone, with $k_d = 0$, will not make the system stable since at best the eigenvalues would be purely complex, and thus the system response would be a non-dampened oscillation.

[14] Frequency domain analysis uses the Laplace transform to model the linear system dynamics rather than a state space model. Controller design and stability analysis of linear systems can be simpler in the frequency domain because the time-domain ordinary differential equations simply become algebraic polynomial models, which are referred to as *transfer functions*.

3.2 *Nonlinear Closed-loop Control*

There are numerous approaches for designing closed-loop controllers for nonlinear dynamical systems. One general approach is to *linearize* the system's dynamics and then apply linear control techniques. Other approaches include nonlinear optimization-based approaches, Lyapunov theory-based methods, and geometric control.

3.2.1 *Linear Control of Nonlinear Dynamical Systems*

We can apply linear control techniques, such as the linear quadratic regulator, to systems with nonlinear dynamics through a process called *linearization.* Linearization is a technique that computes a linear approximation of a nonlinear function at a particular point via a first-order Taylor series expansion. Given a nonlinear dynamic system state space model of the form:

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

we linearize this system about the point[15] $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$ by computing the first order Taylor expansion:

$$f(\boldsymbol{x}, \boldsymbol{u}) \approx f(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}) + \underbrace{\frac{\partial f}{\partial \boldsymbol{x}}(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})}_{A} \delta \boldsymbol{x}(t) + \underbrace{\frac{\partial f}{\partial \boldsymbol{u}}(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})}_{B} \delta \boldsymbol{u}(t),$$

where $\delta \boldsymbol{x} = \boldsymbol{x} - \bar{\boldsymbol{x}}$ and $\delta \boldsymbol{u} = \boldsymbol{u} - \bar{\boldsymbol{u}}$ and the matrices A and B are the *Jacobian* matrices that are defined as the partial derivative of the dynamics with respect to the state and control vectors, evaluated at the linearization point. We therefore have the linear system:

$$\delta \dot{\boldsymbol{x}} = A \delta \boldsymbol{x} + B \delta \boldsymbol{u},$$

which, as a local approximation of the nonlinear dynamics, we can use to design a control law to compute the term $\delta \boldsymbol{u}$. The final closed-loop control law for the original nonlinear system is:

$$\boldsymbol{u}(t) = \bar{\boldsymbol{u}} + \delta \boldsymbol{u}(t),$$

which is a combination of the feedforward control, $\bar{\boldsymbol{u}}$, from the linearization point and the feedback term, $\delta \boldsymbol{u}(t)$. This approach can work well in practice as long as the controller can keep the system close enough to the linearization point that the linear approximation is good[16]. If there is enough divergence, the linear controller could fail and cause the system to become unstable. *Gain scheduling* is an extension of this concept, where linear controllers are designed based on a set of states and the specific controller gains are modified depending on the region the system is currently operating in.

[15] It is common to choose the linearization point to be an equilibrium point of the nonlinear system, such that $f(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}) = 0$. We can then use the resulting linear system to stabilize the nonlinear system about that equilibrium point.

[16] In practice, it might be useful to combine a linearization-based controller for equilibrium maintenance with a startup controller that can initially get the system "close enough" to the setpoint.

Example 3.2.1 (Inverted pendulum)**.** Consider the inverted pendulum depicted in Figure 3.2. Its dynamics are described by:

$$ml^2\ddot{\theta} = mgl \sin(\theta) + u,$$

where m is the mass, l is the length of the rod, g is the acceleration due to gravity, θ is the angle of rotation, and u is the control torque about the axis of rotation. We can express this model in state space form with state $\boldsymbol{x} := \left[\theta, \dot{\theta}\right]^\top$ as:

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}, u) = \begin{bmatrix} \dot{\theta} \\ \frac{g}{l} \sin(\theta) + \frac{1}{ml^2} u \end{bmatrix}.$$

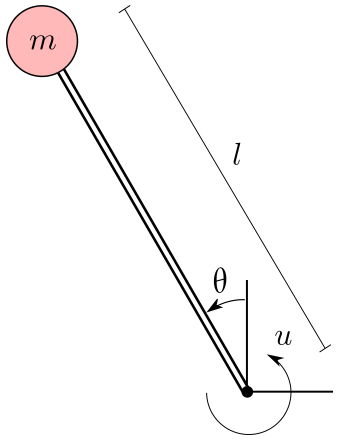


Figure 3.2: An inverted pendulum with a point mass, m, on a rigid rod of length l. The motion is described by the angle θ and u is the control torque about the axis of rotation.

The upright stationary position, $\boldsymbol{x}_e = [0, 0]^\top$, is an equilibrium point for this system with equilibrium control input $u_e = 0$. We can linearize the inverted pendulum dynamics about this equilibrium point to get the linear model:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} 0 & 1 \\ \frac{g}{l} & 0 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix} u,$$

which has an eigenvalue with a positive real part and is therefore unstable, as expected.

We could design a PD controller based on this linear model of the form:

$$u(t) = k_p \theta(t) + k_d \dot{\theta},$$

which would give the closed-loop dynamics:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} 0 & 1 \\ \frac{g}{l} + \frac{1}{ml^2} k_p & \frac{1}{ml^2} k_d \end{bmatrix} \boldsymbol{x}.$$

Similarly to the double integrator PD controller from Example 3.1.1, we can choose the controller gains k_p and k_d, to ensure the closed-loop dynamics are stable. Of course PD control is just one option, we could also compute an LQR controller based on the linear model.

3.2.2 *Nonlinear Control Methods*

Linear control methods are practical and useful in some contexts, but for controlling nonlinear systems they are not necessarily the highest-performing controllers and it can be challenging to obtain stability guarantees. There are several classes of nonlinear control methods that do not require linearization and can improve on the performance and stability of linear controllers, including optimization-based methods, Lyapunov theory-based methods, and geometric control.

Optimization-based approaches can follow a similar optimal control formulation to the open-loop control methods from the previous chapter, except the goal is to directly solve for an optimal closed-loop control law, $\boldsymbol{u}(t) = \pi^*(\boldsymbol{x}(t), t)$. Techniques for solving closed-loop optimal control problems are typically based on either the Hamilton-Jacobi-Bellman equation or dynamic programming. Another optimization-based approach for closed-loop control is *model predictive control (MPC)*[17], which is an approach that repeatedly solves open-loop finite-horizon optimal control problems at each time step, each time accounting for new information. MPC approaches strike a balance between performance and computational tractability: having better closed-loop performance than open-loop methods but less computational burden than closed-loop optimal control methods like dynamic programming.

[17] Also referred to as *receding horizon control*.

Lyapunov-theory based controllers[18] use the concept of Lyapunov functions to prove stability, even for nonlinear systems. Lyapunov functions are scalar

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

Inverted Pendulum Dynamics Linearization

```python
import jax
import jax.numpy as jnp

def inverted_pendulum_dynamics(x, u, g=9.81, m=1, l=1):
    """
    Evaluate the inverted pendulum dynamics.
    """
    θ, dθ_dt = x
    dx_dt = jnp.array([dθ_dt, (g/l)*jnp.sin(θ) + (1/m*l**2)*u])
    return dx_dt

# Linearize around the stationary upright position with zero
# control (i.e. the pendulum is perfectly balanced)
f_jac = jax.jacobian(inverted_pendulum_dynamics, argnums=(0, 1))
x = jnp.array([0., 0.])
u = 0.
A, B = f_jac(x, u) # Evaluate Jacobian at equilibrium point
```

Algorithm 9: Linearizing the inverted pendulum dynamics from Example 3.2.1 in Python using the JAX library. The code for this example is available in the repository `github.com/StanfordASL/pora-exercises` in the notebook `ch03/jax_linearization.ipynb`.

functions that can be thought of as "energy" functions[19]. If we can show that a Lyapunov function is decreasing along the "flow" of the dynamical system for a region around an equilibrium point, then we can guarantee that the equilibrium point is at least locally stable in that region. In the context of closed-loop control, we use the notion of a *control-Lyapunov function (CLF)*, which is essentially a Lyapunov function that can be shown to be decreasing along the flow of a dynamical system for at least some control input to the system. If a CLF exists for a nonlinear system, we can derive a closed-loop controller from the CLF by choosing the control input that minimizes the gradient of the CLF.

[19] Lyapunov functions must be positive definite.

3.3 *Trajectory Tracking Control*

Robotic closed-loop control problems are often broken down into a combination of two sequential steps. First, we use an open-loop method to generate a desirable trajectory, for example using optimal control or differential flatness methods, and then the second step is to *track* that trajectory using real-time observations in a closed-loop fashion. This approach, referred to as *trajectory tracking control*, is popular because it can take advantage of the computational

tractability of the open-loop trajectory generation methods while also gaining the performance and robustness advantages of closed-loop control.

Definition 3.3.1 (Trajectory Tracking Control Law)**.** We express the general *trajectory tracking* controller[20] in the form:

$$\boldsymbol{u}(t) = \boldsymbol{u}_d(t) + \pi(\boldsymbol{x}(t), \boldsymbol{x}_d(t), t), \tag{3.9}$$

where $\boldsymbol{x}_d(t)$ is the desired trajectory state and $\boldsymbol{u}_d(t)$ is the corresponding control[21], and $\pi(\boldsymbol{x}(t), \boldsymbol{x}_d(t), t)$ is a feedback function that attempts to correct for tracking error.

[20] As with Equation (3.1), we can also define this control law as a function of the measured system outputs if the full system state, $\boldsymbol{x}$, is not directly measurable.

[21] The control $\boldsymbol{u}_d(t)$ is also referred to as the *feedforward* control term.

We have already discussed open-loop methods for generating the desired trajectory, $\boldsymbol{x}_d(t)$, and control, $\boldsymbol{u}_d(t)$, in Chapter 2, and we can design the feedback component, $\pi(\boldsymbol{x}(t), \boldsymbol{x}_d(t), t)$ using various approaches from Section 3.1 and Section 3.2. For example, for a nonlinear system, we could design the feedback term using a linearization-based approach by linearizing at each point of the desired trajectory.

Example 3.3.1 (Trajectory Tracking via LQR)**.** Consider a linear system:

$$\dot{\boldsymbol{x}} = A\boldsymbol{x} + B\boldsymbol{u},$$

and a desired trajectory $(\boldsymbol{x}_d(t), \boldsymbol{u}_d(t))$ defined over the time interval $[0, T]$ computed using an open-loop method. The trajectory tracking error dynamics are:

$$\delta\dot{\boldsymbol{x}} = A\delta\boldsymbol{x} + B\delta\boldsymbol{u},$$

where $\delta\boldsymbol{x} = \boldsymbol{x} - \boldsymbol{x}_d$ and $\delta\boldsymbol{u} = \boldsymbol{u} - \boldsymbol{u}_d$. We can apply the finite-horizon LQR controller discussed in Section 3.1.1, which would yield a feedback controller of the form:

$$\delta\boldsymbol{u}(t) = K^*(t)\delta\boldsymbol{x}(t),$$

and therefore the trajectory tracking controller from Equation (3.9) would take the form:

$$\boldsymbol{u}(t) = \boldsymbol{u}_d(t) + K^*(t)(\boldsymbol{x}(t) - \boldsymbol{x}_d(t)).$$

3.3.1 Trajectory Tracking for Differentially Flat Systems

One useful fact about differentially flat systems is that they can be *feedback linearized* to yield a linear dynamical system of the form:

$$\boldsymbol{z}^{(q+1)} = \boldsymbol{w}, \tag{3.10}$$

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[22], and $\boldsymbol{w}$ is a modified "virtual" control input term[23].

Since the system in Equation (3.10) is linear, we can leverage techniques from linear control theory in Section 3.1 to design a closed-loop feedback controller

[22] The degree of the flat output space is the highest order of derivatives of the flat output that are needed to describe system dynamics.

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

for computing the virtual control, $\boldsymbol{w}$, which we can use to track an open-loop trajectory for the flat outputs. In particular, suppose we compute a reference flat output trajectory, $\boldsymbol{z}_d(t)$, and corresponding virtual input, $\boldsymbol{w}_d(t)$, using an open-loop method. 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:

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

where $\boldsymbol{e}^{(j)} = \boldsymbol{z}^{(j)} - \boldsymbol{z}_d^{(j)}$ is the j-th order derivative of the error and K_j is a diagonal matrix of controller parameters. Applying this control law to the system in Equation (3.10) will result in the closed-loop dynamics:

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

Since $\boldsymbol{z}_d^{(q+1)} = \boldsymbol{w}_d(t)$ from the reference trajectory, this can be simplified to give the closed-loop error dynamics:

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

We can now apply methods from linear control theory to choose the controller parameters in K_i that will make the error dynamics converge to zero, which will drive the system to track the flat output trajectory.

Example 3.3.2 (Extended Unicycle Trajectory Tracking)**.** Consider the dynamically extended unicycle model:

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

where the two control inputs are the acceleration, a, and the rotation rate, ω. This system is differentially flat with flat output $\boldsymbol{z} = [x, y]^\top$ and order $q = 1$. We can therefore express the system dynamics as:

$$\ddot{\boldsymbol{z}} = J(\theta, v)\boldsymbol{u}, \quad J(\theta, v) = \begin{bmatrix} \cos(\theta) & -v\sin(\theta) \\ \sin(\theta) & v\cos(\theta) \end{bmatrix},$$

where $\boldsymbol{u} = [a, \omega]^\top$, and define the virtual control inputs:

$$\boldsymbol{w} \coloneqq J(\theta, v)\boldsymbol{u},$$

so that we have a linear system of the form in Equation (3.10). We can then define a trajectory tracking controller for the feedback linearized system:

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

where $(\cdot)_d$ represents a term associated with the desired trajectory, and $k_{px}, k_{dx}, k_{py}, k_{dy} > 0$ are control gains. Next, we can retrieve the control inputs, $a(t)$ and $\omega(t)$, by solving the linear system:

$$J(\theta, v) \begin{bmatrix} a \\ \omega \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix},$$

assuming that the matrix $J(\theta, v)$ is full rank.

3.4 *Exercises*

The starter code for the exercises provided below is available online through GitHub. To get started, download the code by running in a terminal window:

```
git clone https://github.com/StanfordASL/pora-exercises.git
```

We denote Problems requiring hand-written solutions and coding in Python with ✎ and 🖳, respectively.

🖳 *Problem 1: Inverted Pendulum PD Control*

In the notebook `ch03/exercises/pid_control.ipynb`, implement the PD controller for the inverted pendulum dynamics. Then, play around with different values of the gains k_p and k_d and use JAX and NumPy to compute the Jacobian of the closed-loop dynamics. Verify the eigenvalues of the linearized matrix are stable. Finally, run the provided code to simulate the nonlinear closed-loop dynamics.

🖳 *Problem 2: Extended Unicycle Trajectory Tracking Control*

In the notebook `ch03/exercises/unicycle_trajectory_tracking.ipynb`, implement a differential flatness-based trajectory tracking controller for the extended unicycle robot described in Example 3.3.2.

References

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

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

4
Search-Based Motion Planning

In previous chapters, we approached the problem of robotic motion planning through the lens of control theory and optimal control. These techniques generate open and closed-loop control laws to accomplish specific tasks such as trajectory generation, trajectory tracking, and stabilization about a particular robot state. One common component of these methods is the use of a mathematical model of the robot's kinematics or dynamics to describe how the robot transitions from state to state given control inputs.

In this chapter, we explore another class of algorithms[1] for motion planning and trajectory generation that formulates the motion planning problem with respect to the robot's *configuration space* rather than the *state space* from previous chapters. This class of algorithms is well suited for higher-level motion planning tasks, such as motion planning in environments with obstacles. These approaches were developed in parallel with many of the techniques from previous chapters, and are still being researched today. Recall the general definition of the motion planning problem:

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

Definition 4.0.1 (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.

In the previous chapters, we approached this problem by formulating mathematical optimization problems that minimized a cost function subject to constraints on the motion including dynamics and kinematics, control limits, and conditions on the robot's initial state, or leveraged differential flatness properties of the robot's motion model. In these approaches, we parameterized the robot's trajectory by its state, $\boldsymbol{x}$, and the corresponding control inputs, $\boldsymbol{u}$, which satisfy a set of differential equations:

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

In this chapter, we address the motion planning problem with respect to a *configuration space*[2]. 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. In some cases, the state and configuration of the robot may be the same, but in other

[2] We sometimes use the short-hand notation $\mathcal{C}$-space to represent the configuration space.

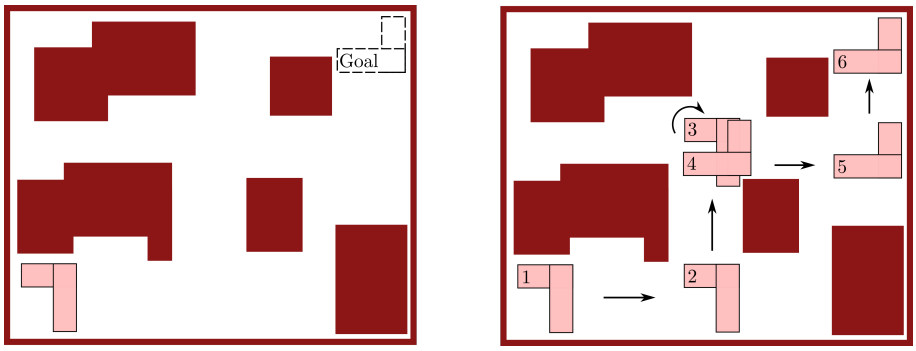


Figure 4.1: Motion planning in a two-dimensional workspace with obstacles.

cases we can tailor the definition of the configuration to simplify the motion planning problem. One important example of this is for geometric path planning, where we can plan paths in the configuration space without considering the robot's kinematic and dynamics model.

Example 4.0.1 (L-shaped Robot)**.** Consider the L-shaped robot in Figure 4.1 that lives in a two-dimensional world with obstacles, and is trying to get from one point to another. Suppose this robot has a state $\boldsymbol{x} = \left[x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}\right]^\top$, where x and y describe the robot's position and θ is its orientation, and consider a configuration space defined by $\boldsymbol{q} = [x, y, \theta]^\top$ which fully captures the robot's degrees of freedom. This motion planning problem involves obstacle avoidance, and, therefore, it can be easier to plan a sequence of collision-free configurations, $\boldsymbol{q}$, as we show in the right-side graphic of Figure 4.1, than try to compute a trajectory for the full state, $\boldsymbol{x}$. In this case, our use of the configuration space simplifies the motion planning problem by abstracting away the consideration of the robot's full dynamics. Once we define a geometric path in the configuration space, we can use other techniques, including those discussed in previous chapters, to define lower-level control laws for path tracking.

Note that in this example problem, the $\mathcal{C}$-space is $\mathbb{R}^2 \times \mathcal{S}^1$ which is a subset of $\mathbb{R}^3$. 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 because it encodes the robot's ability to move from one angle to another in two different ways[3]. In the context of this motion planning example problem, suppose the robot in Figure 4.1 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, as we show in Figure 4.2. If we don't consider the equivalence between the orientations 0 and 2π in the definition of the configuration space, the robot would not be able to traverse to the desired heading, since as Figure 4.2 shows, a clockwise rotation leads to a collision. By understanding that the configuration space is defined with respect to $\mathcal{S}^1$, the robot can achieve the desired heading by simply rotating counter-clockwise.

[3] For example, the robot can turn left or turn right to get to the same orientation.

Within the context of configuration space motion planning, we discuss two classes of *search-based* algorithms: *grid-based methods* and *combinatorial planners*.

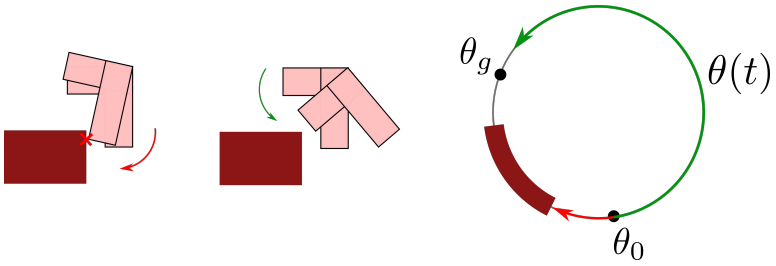


Figure 4.2: Motion planning example where the configuration space is defined by the manifold $\mathcal{S}^1$, which is crucial for getting from the initial orientation, θ_0, to the goal orientation, θ_g. In particular, rotating clockwise leads to collision but rotating counter-clockwise is a feasible path.

In both of these classes of search-based algorithms, we will look for several important properties. First, we desire algorithms that are *sound* in the sense that they either return a valid solution or no solution at all. Second, we will look for algorithms that are *complete*, which means that they terminate in finite time and return a valid solution if it exists. Finally, search-based algorithms should have good *time complexity* and *space complexity*, which refer to the number of steps before termination and maximum number of memory locations as a function of the input problem size, respectively.

4.1 Grid-based Motion Planners

In many robotics problems, the robot's configuration, $\boldsymbol{q}$, is a d dimensional vector and the $\mathcal{C}$-space is a subset of the continuous space $\mathbb{R}^d$. Planning in continuous spaces is challenging because there are an infinite number of potential configurations the robot take. Grid-based motion planners simplify this problem by using a grid to discretize the $\mathcal{C}$-space into a finite number of allowable configurations. For example, Figure 4.3 shows how we can discretize a simple $\mathcal{C}$-space in two dimensions into a grid.

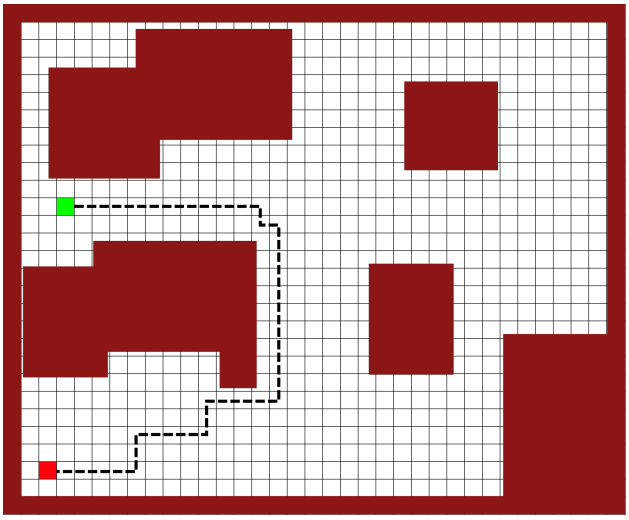


Figure 4.3: Discretizing the continuous configuration space using a grid.

In grid-based planners, we define undesirable configurations by simply iden-

tifying some cells of the grid to be forbidden, such as for obstacle collision avoidance. We also abstract away the kinematics and dynamics of the robot and assume that the robot has the ability to move freely between adjacent configurations cells. Following the $\mathcal{C}$-space discretization, the resulting motion planning problem is sometimes referred to as a *discrete planning* problem because only a finite number of neighboring configurations are available at each step, and the total number of possible configurations is finite. The motion planning planning problem therefore reduces to finding a way to traverse through the configuration cells from the initial configuration to a desired final configuration.

Mathematically, we can represent discrete planning problems using discrete *graphs*. We define a graph, denoted by $G = (V, E)$, by a set of vertices[4], V, and a set 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 configuration cells. With the graph representation, we cast the planning problem as finding a way to traverse through the graph to reach the desired vertex. Algorithms for solving these problems are referred to as *graph search methods*. The advantages of graph search approaches are that they are simple to use and for some problems can be very fast. The disadvantages primarily come from the result of the discretization procedure. For example, if the resolution of the grid is not fine enough the search algorithm may not be able to find a solution. Additionally, for a fixed resolution grid, the size of the graph grows exponentially with respect to the dimension of the configuration space. This computational complexity limits graph search methods to applications with simple robots with a low-dimensional configuration space.

[4] We also sometimes refer to the vertices as *nodes*.

4.1.1 Label Correcting Algorithms

A graph that represents the discrete planning problem is defined by a finite number of vertices and edges. Therefore, it should be theoretically possible to solve a graph search problem in finite time. In order to achieve this in practice, we must use several simple "accounting" tricks to keep track of how the search has progressed and avoid redundant exploration. We also often want to find an optimal path, and therefore we require a mechanism to keep track of the current best path that the algorithm has found. *Label correcting algorithms* are a general set of algorithms that employ these types of accounting techniques to guarantee satisfactory performance.

We typically define the notion of an optimal path in label correcting algorithms in terms of a cost-of-arrival.

Definition 4.1.1 (Cost-of-Arrival)**.** The *cost-of-arrival* associated with a vertex q, denoted $C(q)$, 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.

We also define the cost for traversing an edge from vertex q to vertex q' as $C(q, q')$[5]. To keep track of the nodes that have already been visited and which

[5] We can view this as a stage or running cost as defined in the previous chapters on optimal control.

still need further exploration, label correcting algorithms define a set of *frontier vertices*[6]. Tracking frontier vertices allows us to make guarantees 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, the algorithm will find it.

[6] Frontier vertices are also sometimes referred to as *alive*

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[7]:

[7] Note that we refer to graph vertices using the notation q to connect the graph abstraction to the fact that the node represents a physical robot configuration $\boldsymbol{q}$.

1. Initialize a set, Q, defining the set of frontier vertices as $Q = \{q_I\}$. Initialize the cost-of-arrival for the starting vertex as $C(q_I) = 0$ and for all other vertices, q', as $C(q') = \infty$.

2. Remove a vertex from the set Q and explore each of its connected[8] vertices, q'. For each connected vertex, 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 it 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 frontier vertex set, Q, if q' is not q_G.

[8] A vertex, q', is connected to q if there is an edge, (q, q'), in the graph.

3. Repeat step 2 until the set of frontier vertices, Q, is empty.

We also detail these steps in Algorithm 4.1.

Algorithm 4.1: General Label Correcting

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

Essentially, the label correcting algorithm iteratively searches connected neighbors, q', from a vertex, q, to see if moving from q to q' will lead to a lower overall cost than any previously found paths to q'. This is why we call these algorithms "label correcting", since they "correct" the cost-of-arrival as they find

better paths throughout the search process. Eventually, once the algorithm finds the best path from q_I to q, the vertex q will never again be added to the frontier set, Q, and therefore the algorithm is guaranteed to eventually terminate.

Theorem 4.1.2 (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, $C^*(q_G)$.*

Within the class of label correcting algorithms, individual algorithms primarily differ by how they select the next vertex from the set of frontier nodes, Q. In fact, we often refer to the frontier node set as a *priority queue* since the algorithm might assign priority values to the order in which vertices are selected. Different approaches for prioritizing the searched nodes include *depth-first search*, *breadth-first search*, and *best-first search*.

Depth-First Search: Depth-first search in a directed graph expands each node 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 frontier set, Q, is "last in/first out", where whenever a new vertex is selected from Q it chooses the vertex that was most recently added. This approach is *sound* in the sense that if it returns a path it is a valid path from the start to the goal. However, it is not *complete* on infinite graphs since if it starts in the wrong direction it will not converge.

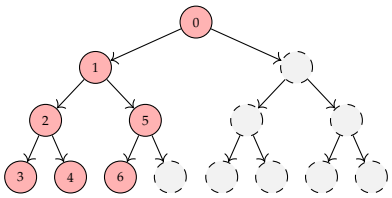


Figure 4.4: Depth-first search

Example 4.1.1 (Depth-first Search)**.** The general principle of depth-first search is that the successor nodes of previously explored nodes are added at the front of the queue. Starting at the node S in the graph shown in Figure 4.5, where the goal is to get to node G, the table below shows the changes to the queue, Q, and set of visited nodes, V, at each iteration:

Q	V
(S)	$\{S\}$
(A, B)	$\{S, A, B\}$
(C, D, B)	$\{S, A, B, C, D\}$
(D, B)	$\{S, A, B, C, D\}$
(G, B)	$\{S, A, B, C, D, G\}$
(B)	$\{S, A, B, C, D, G\}$

The depth-first search algorithm will first find the path (S, A, D, G).

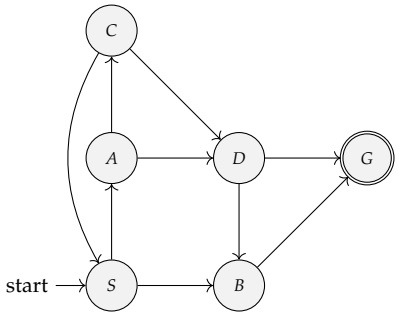


Figure 4.5: Example graph where the goal is to start from node S and find a path to the goal node G.

Breadth-First Search: In contrast to depth-first search, breadth-first search explores all of the unexplored neighboring nodes of the selected node before searching deeper. In terms of of the frontier set, Q, breadth-first search stores nodes as a queue where the first node added is the first node selected. This algorithm is *sound* in the sense that if it returns a path it is a valid path from the start to the goal. It is also *complete* on finite or countably infinite transition graphs.

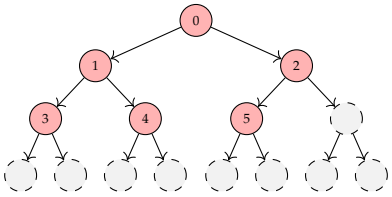


Figure 4.6: Breadth-first search

Example 4.1.2 (Breadth-first Search)**.** The general principle of breadth-first search is that the successor nodes of previously explored nodes are added at the back of the queue, Q. Given the graph in Figure 4.5, the changes to the queue, Q, and the set of visited nodes, V, at each iteration following breadth-first search are:

Q	V
(s)	$\{s\}$
(a, b)	$\{s, a, b\}$
(b, c, d)	$\{s, a, b, c, d\}$
(c, d, g)	$\{s, a, b, c, d, g\}$
(d, g)	$\{s, a, b, c, d, g\}$
(g)	$\{s, a, b, c, d, g\}$

This algorithm will first find the path (s, b, g).

Best-First Search: Best-first search[9] greedily selects vertices from the frontier set, Q, by looking at the current best cost-of-arrival. Mathematically we express this search priority as:

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

[9] Best-first search is also commonly known as *Dijkstra's algorithm*.

This approach is considered an optimistic approach since it is modeling the assumption that the best action based on current knowledge of the cost-of-arrivals 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.

4.1.2 A Algorithm*

A* is a label correcting algorithm that modifies the best-first search approach of Dijkstra's algorithm. Specifically, in Dijkstra's algorithm the goal vertex, q_G, is not taken into account by the search procedure. This can potentially lead to wasted effort in cases where the greedy choice makes no progress towards the goal. In the A* algorithm, we shift our focus from greedily optimizing for the cost-of-arrival to searching based on the *cost-to-go*.

Definition 4.1.3 (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.

We may not always know the cost-to-go in practice, and therefore we use *heuristics* to provide approximate cost-to-go values, which we denote by $h(q)$. 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, for a shortest distance traveled problem is to simply use the Euclidean 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 a total cost, $f(q)$, which we define as the

cost-of-arrival, $C(q)$, plus an approximate cost-to-go, $h(q)$[10]. This provides a better estimate of the total quality of a path than just using the cost-of-arrival alone. We define the A* algorithm more concretely in Algorithm 4.2.

[10] In the case that we choose the heuristic to be $h(q) = 0$ for all q, which is technically a valid heuristic, then A* is the same as Djikstra's algorithm.

Algorithm 4.2: 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*

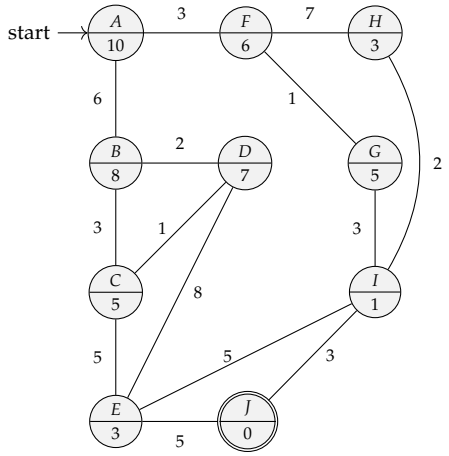


Figure 4.7: Graph for the A* example in Example 4.1.3. The numbers in the nodes represent the heuristic cost-to-go to reach the goal and the edge numbers show the edge cost to traverse between the nodes.

Example 4.1.3 (A* Algorithm)**.** Consider the graph in Figure 4.7, where we show the heuristic cost-to-go for each node along with the actual costs associated with each edge, and where the goal is to get from node A to node J. We show the iterations of A* in Table 4.1, where we highlight the stage costs, the cost-to-go heuristic, and the total cost considered by the algorithm. The algorithm begins with the starting node, A, which has a cost-of-arrival of $C = 0$, and a heuristic cost-to-go of $h = 10$, and therefore the total cost is $f = 10$. Next, we consider the paths from the edges of A, (A, F) and (A, B). The path (A, F) has a edge cost of $C(A, F) = 3$ and node F has a heuristic cost-to-go of $h = 6$, and therefore the total cost for node F is $f = 9$. Similarly, the path (A, B) has an edge cost $C(A, B) = 6$, $h = 8$, and $f = 14$. Since the total cost is lower for node F, we proceed by expanding (A, F) into (A, F, G) and (A, F, H), which end up having a total cost of $f = 9$ and $f = 13$, respectively. Next, we expand from node G due to the lower total cost to get (A, F, G, I), with $f = 8$, and continue with the next expansion to consider (A, F, G, I, J), (A, F, G, I, H), and (A, F, G, I, E), with total costs of 10, 12, and 15. At this point, the algorithm terminates since it has found a path to the goal node. Interestingly, in some cases, if we were to substitute the cost-to-go heuristic values for some of the nodes, for example if we changed

Path being considered	Stage cost C	Cost to go h	Total cost f
(A)	0	10	10
(A, B)	6	8	14
(A, F)	3	6	9
(A, F, G)	3+1	5	9
(A, F, H)	3+7	3	13
(A, F, G, I)	3+1+3	2	8
(A, F, G, I, H)	3+1+3+2	3	12
(A, F, G, I, E)	3+1+3+5	3	15
(A, F, G, I, J)	3+1+3+3	0	10

Table 4.1: Table summarizing the costs for the A* example in Example 4.1.3.

nodes B and D to have $h(B) = 2$ and $h(D) = 1$, it would take more steps for the algorithm to terminate. This is because these new heuristics are less accurate representations of the true total best cost-to-go, which highlights the importance of choosing good heuristics for computational efficiency.

4.2 *Combinatorial Motion Planning*

In contrast to grid-based planners, *combinatorial motion planners* find paths through the continuous configuration space without resorting to discretizations. Recall that in grid-based planners undesirable cells in the discretized configuration space, such as cells corresponding to collisions, are blocked out and simply not considered in the resulting path search. In combinatorial planners, we consider the structure of the free portion of the configuration space in a different way. First, we denote the subset of the configuration space, C, that is free, for

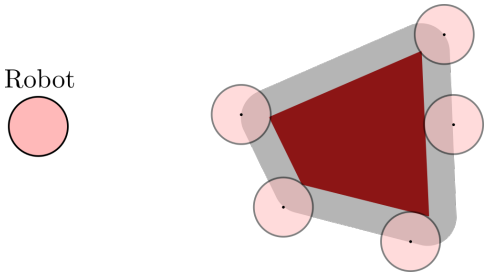


Figure 4.8: Free (white) and forbidden spaces (grey and red) of the configuration space for a simple circular robot in a two-dimensional world. Note that the forbidden space accounts for the physical dimensions of the robot.

example being collision free, as C_{free} and we refer to this set as the *free space*. For example, in Figure 4.8 and Figure 4.9 the free space is the areas not shaded with grey or red. Within the free space, C_{free}, combinatorial motion planners compute structures that we refer to as *roadmaps*. A roadmap is a graph where each vertex represents a point in the continuous subset of the configuration space, C_{free}, and each edge represents a path through C_{free} that connects a pair of vertices. This graph structure is similar to that used in grid-based planners with the important distinction that the vertices can potentially be any configuration $q \in C_{\text{free}}$, while in grid-based planners the vertices are defined ahead of time by discretization.

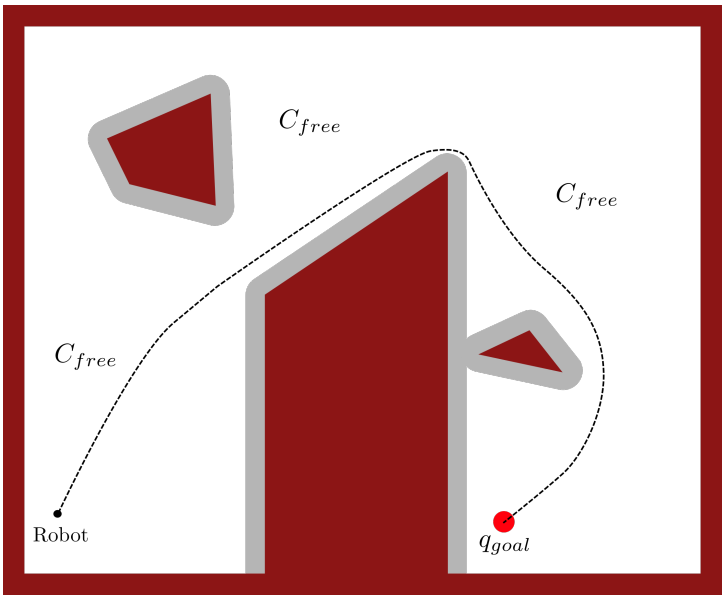


Figure 4.9: Once the free (white) and forbidden (grey and red) configurations have been identified, we can ignore the physical dimensions of the robot. This figure shows an example of a path planning problem in $\mathcal{C}$-space with obstacles.

This distinction is important because the flexibility of choosing the vertices does not result in any loss of information. Once we have defined the roadmap, we can compute a path from any point, $q_I \in C_{\text{free}}$, to a goal point, $q_G \in C_{\text{free}}$, 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 motion planners are *complete* in the sense that the algorithm will either find a solution or will correctly report that no solution exists, and in some cases these planners can even be optimal. However, in practice they are not always computationally feasible to implement except in problems with low-dimensional configuration spaces or with simple geometric representations of the environment. These motion planners also require that the free space be completely defined in advance, which is not necessarily a realistic requirement in practice.

4.2.1 Cell Decomposition

One common approach for deriving the roadmap for combinatorial motion planning is to use *cell decomposition* to decompose the configuration space to define C_{free}. Cell decomposition is the process of partitioning the free space, C_{free}, into a finite set of regions called cells. Each cell should be easy to traverse and should ideally be convex, the decomposition should be easy to compute, and adjacencies between cells should be straightforward to determine to aid in building the roadmap.

Example 4.2.1 (2D Cell Decomposition)**.** Consider a two-dimensional configuration space as shown in Figure 4.10. This space is decomposed into cells that are either lines or trapezoids by a process called vertical cell decomposition. Once

we have defined the cells, we generate the roadmap by placing a vertex in each cell, such as 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. There exist several approaches for performing cell decomposition, but in higher dimensions it becomes increasingly challenging.

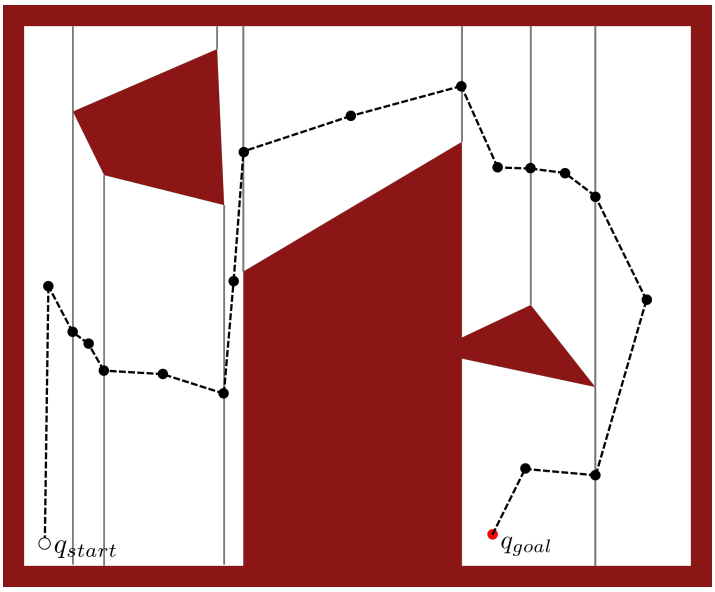


Figure 4.10: Example of two-dimensional cell decomposition with C_{free} colored white. We define a roadmap as the graph, G, with vertices shown as black dots and edges connecting them. To solve a planning problem with q_{start} and q_{goal}, we first connect these points to the roadmap and then the path is easily defined.

4.2.2 *Other Roadmaps*

Other approaches to define roadmaps, besides using cell decomposition, include maximum clearance or minimum distance methods. Maximum clearance roadmaps simply try to always keep as much distance from obstacles as possible, for instance 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, the use of minimum distance roadmaps is likely not desirable and therefore these approaches are less commonly used without additional modifications.

4.3 *Exercises*

The starter code for the exercises provided below is available online through GitHub. To get started, download the code by running in a terminal window:

```
git clone https://github.com/StanfordASL/pora-exercises.git
```

We denote Problems requiring hand-written solutions and coding in Python with ✎ and 🖳, respectively.

🖳 *Problem 1: A* Motion Planning*

In this exercise, you will implement the A* grid-based motion planning algorithm for some simple two-dimensional environments. In the file `ch04/exercises/a_star.ipynb`, you will implement the key parts of the A* algorithm, run the algorithm on some randomly generated path planning problems, and then will explore a way to smooth the resulting discrete paths , which could be useful for practical robot motion planning tasks.

References

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

5
Sampling-Based Motion Planning

In Chapter 4, we introduced motion planning problems that are formulated with respect to the robot's configuration space, or in short the C-space. Within the context of C-space planning, we also introduced two categories of approaches known as 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, avoiding the use of a rigid grid structure, and then plan with respect to the roadmap by connecting the initial configuration and desired configuration to the roadmap and performing a graph search to find a path along the roadmap. Generally speaking, grid-based methods suffer from the rigidity of the discretization. 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[1] in advance.

In this chapter, we present a class of motion planning algorithms which also build a roadmap, but do not require a full characterization of the free configuration space in advance. Instead, these algorithms build roadmaps one point at a time by sampling a point in the configuration space and querying an independent module to determine if the sample is admissible[2]. We refer to this class of planners as *sampling-based algorithms*[3].

Sampling-based algorithms are a common choice for practical applications since they are conceptually simple, flexible, relatively easy to implement, and can be extended beyond the geometric case to include things like differential motion constraints. The disadvantages of these approach are typically related to theoretical guarantees, for example these approaches cannot certify that a solution does not exist. In this chapter, we focus on two popular sampling-based methods, probabilistic roadmaps (PRM) and the rapidly-exploring random trees (RRT) algorithm, and will also briefly mention other methods such as the fast-marching tree algorithm (FMT*), kinodynamic planning, and deterministic sampling-based methods.

[1] The free configuration space is the set of all points in the configuration space that are allowable, such as ones that don't result in a collision with obstacles.

[2] In the context of robotics, an admissible configuration in motion planning problems is often one that is collision-free, and therefore we often refer to this module as a collision detection module, or simply a *collision checker*.

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

5.1 Probabilistic Roadmap (PRM)

The *probabilistic roadmap (PRM)* sampling-based method is conceptually similar to the combinatorial planners we discussed in Chapter 4 in that it constructs a topological graph, G, known as a *roadmap*. This graph includes vertices that represent robot configurations, q, within the unobstructed portion of the configuration space, C_{free}. These vertices are connected by edges that also reside entirely within C_{free}. After constructing the roadmap, we determine a motion plan for a specific initial configuration, q_I, and goal configuration, q_G, by first linking them to the roadmap and subsequently employing a graph-search method such as A* to navigate a path through the roadmap graph. The primary distinction between the PRM sampling-based method and combinatorial planners is related to the method of generating the roadmap.

The fundamental advantage of the PRM approach over combinatorial planners is that it avoids the need for a complete analysis of the free configuration space, which is a process that is typically resource-intensive. Instead, the PRM method involves randomly sampling configurations, q, and utilizing a black-box module to determine if q belongs to C_{free}. Concretely the outline of the general PRM approach is:

1. Randomly sample n configurations[4], q_i, from the configuration space, C.
2. Query a black-box module for each sampled configuration, q_i, to determine if $q_i \in C_{\text{free}}$. If $q_i \notin C_{\text{free}}$, then we remove it from the sample set.
3. Create a graph, $G = (V, E)$, with vertices from the sampled configurations, $q_i \in C_{\text{free}}$. Define a radius[5], r, and create edges for every pair of vertices, q and q', where the norm satisfies $\|q - q'\| \leq r$ and the straight line path between q and q' is also in the free configuration space[6].

[4] We sometimes refer to PRM as a *multi-query* method since it samples a batch of n samples at once.

[5] Referred to as the *connectivity radius*.

[6] We often perform edge validation by densely sampling the edge and querying the free space black-box module at each point.

Figure 5.1 shows an example of a PRM roadmap graph. Note that defining the

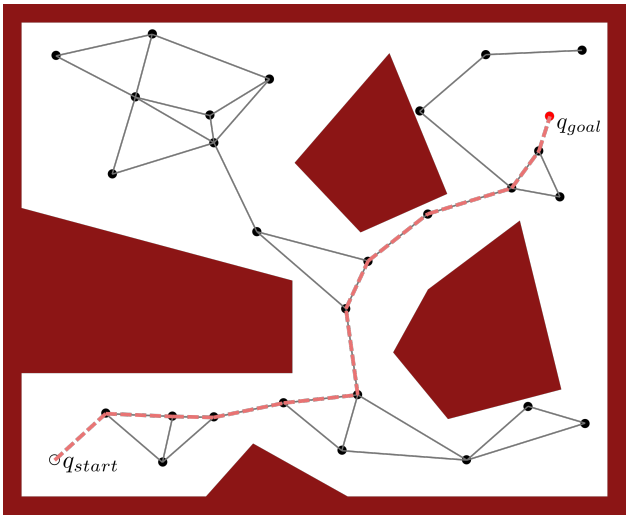


Figure 5.1: Example roadmap graph and path found by the PRM algorithm. The black dots represent the randomly sampled vertices of the graph and the grey lines represent the edges created between vertices within a predefined radius, r, of each other. The initial configuration, q_{start}, and goal configuration, q_{goal}, are connected through this roadmap along the pink line, which we find by using a graph-search algorithm.

connectivity radius, r, provides us with 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 free space module checks. On the flip side, making the radius too small could mean that we do not make enough connections.

One of the limitations of the probabilistic roadmap approach is that it often requires a large number of samples to adequately span the configuration space and find a satisfactory solution. A large sample size also leads to many queries of the free configuration space checking module, which can be an expensive operation in practice. However, in certain robotics applications, it might be acceptable or even advantageous to use an approach like PRM that thoroughly covers the space C_{free} with a roadmap. For example, in some applications we may have a need to solve the motion planning problem many times for various pairs of initial and goal configurations. In these scenarios, we only construct the PRM roadmap once, and then can reuse it for each problem as long as the environment remains unchanged.

5.2 *Rapidly-exploring Random Trees (RRT)*

Applications with static environments and a need for solving multiple planning problems can benefit from methods like PRM that front-load the work to build a detailed roadmap across the whole free configuration space. However, there are also many problems in robotics that are classified as *single-query* problems, where we will only compute a single motion plan for a given free configuration space[7]. Building a roadmap over the entire free configuration space in single-query contexts typically results in wasted effort. The *Rapidly-exploring Random Trees (RRT)* algorithm solves the single-query problem by incrementally sampling and building the roadmap, starting at the initial configuration, until the goal configuration is reached. The roadmap graph in this case is built as a *tree*, which is a special type of graph that has only one path between any two vertices.

[7] A common single-query planning scenario arises in contexts with a dynamic environment, such as if there is a moving obstacle.

In general, the RRT algorithm begins by initializing a tree[8], $T = (V, E)$, with a vertex at the initial configuration, $V = \{q_I\}$. At each iteration, the RRT algorithm performs the following steps:

[8] Due to its special structure, we refer to the roadmap graph as a tree and denote it as T rather than G.

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.

After each iteration, only a single point is sampled and potentially added to the tree. Occasionally, we also set the sampled point as the goal configuration, $q = q_G$, and if the nearest point, q_{near}, can be connected to q_G through the free configuration space, the search is terminated. Intuitively, this approach works because of a phenomenon we refer to as the *Voronoi bias*, which 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.

Variations on this standard algorithm exist, and in particular there are 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*[9]. RRT* introduces a notion of optimality into the algorithm and will 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 and the goal configuration and tries to connect them.

[9] RRT* is pronounced "RRT star".

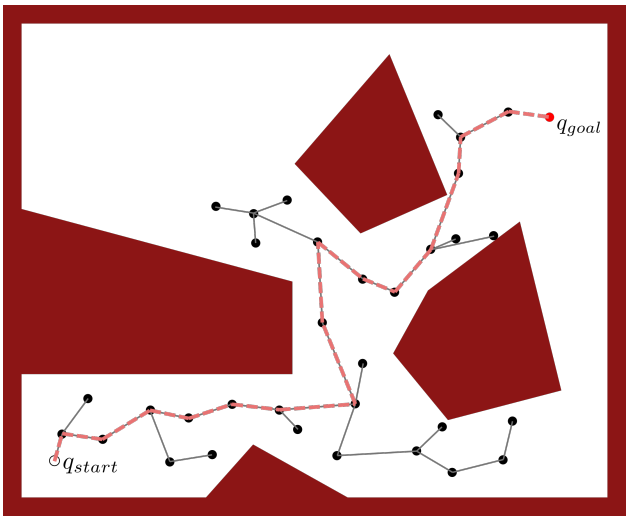


Figure 5.2: Example exploration tree by the RRT algorithm. The black dots represent points sampled at each iteration of the algorithm, which are connected to the nearest vertex that is currently part of the tree.

5.3 *Theoretical Results for PRM and RRT*

One of the main challenges of sampling-based motion planning is that it is unclear how many samples we need to find a solution. However, there are some theoretical guarantees for PRM and RRT regarding their asymptotic behavior as the number of samples approaches infinity, $n \to \infty$. In particular, PRM with a constant connectivity radius and RRT are guaranteed[10] to eventually find a solution if it exists[11,12]. With an appropriate choice of the connectivity radius, we can also show that PRM will find optimal paths as the number of samples approaches infinity. However, RRT can behave arbitrarily bad with non-negligible probability[13].

[10] These guarantees require an assumption that the configuration space is bounded.

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

[12] 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

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

5.4 Fast Marching Tree Algorithm (FMT*)

As we briefly mentioned in Section 5.3, PRM is an asymptotically optimal algorithm, meaning that it can find high-quality paths given a sufficient number of samples. However, using a high number of samples in PRM requires a high number of queries of the a black-box free configuration space checking module that can make the approach computationally expensive in practice. In contrast, the vanilla RRT algorithm is quick but typically fails to produce optimal paths.

The *Fast Marching Tree (FMT*)* algorithm[14], an advanced sampling-based motion planning technique, combines the benefits of both PRM and RRT style methods in that it is both rapid and asymptotically optimal. FMT* builds a tree structured graph in the same way RRT does, therefore maintaining the efficiency of RRT, but makes connections in a way that allows for asymptotic optimality. In particular, new connections are made using a technique referred to as *dynamic programming*. Dynamic programming is a technique that we can use to find the optimal path[15] with respect to a *cost-of-arrival*, denoted by $C(q)$, which represents the cost to traverse from the initial configuration, q_I, to the configuration q.

[14] 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

[15] An example of a common metric we may want to optimize is the Euclidean distance, which would result in computing a shortest distance path.

In the context of motion planning, dynamic programming leverages Bellman's *principle of optimality*, which states that the cost-of-arrival of the optimal paths satisfy the condition:

$$C^*(q) = \min_{q' : \|q' - q\| < r} C(q, q') + C^*(q'), \tag{5.1}$$

where q' are configuration nodes within radius r of node q, $C(q, q')$ is the cost to traverse the edge between q and q', and $C^*(q')$ is the optimal cost-to-arrive at q'. This relationship says that the cost-of-arrival at any configuration, q, 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 each time it needs to connect a new sample to the tree. However, in practice, using the condition in Equation (5.1) is complicated by the fact that the resulting edge may result in a violation of the non-free part of the configuration space. FMT* handles this by ignoring obstacles when leveraging the condition in Equation (5.1) to connect a new sample to the tree, and then if a violation of the non-free part of the C-space occurs from the resulting connection, it is simply skipped and the algorithm moves on to a new sample. This application of dynamic programming is referred to as *lazy* because it only checks for the constraint violation after the fact, and it turns out that this substantially reduces the total number of queries to the free-configuration space checker module and only leads to sub-optimality in rare cases.

5.5 Kinodynamic Planning

The geometric motion planning algorithms we previously introduced assume that the robot does not have any constraints on its motion and that we only

require the path to be contained within the free configuration space. This assumption makes the planning task easier because we can simply connect two configurations, q and q', with a straight line. However, as we discussed in Chapter 1, robots typically have kinematic and dynamic motion constraints, and for many 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*[16].

[16] 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

Similar to the previous chapters on open and closed-loop control and trajectory optimization, we assume that the robot operates in a state space, $\mathcal{X} \subseteq R^n$, with control inputs $\boldsymbol{u} \in \mathcal{U} \subseteq \mathbb{R}^m$, and that the motion constraints are defined by the differential model:

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u}), \tag{5.2}$$

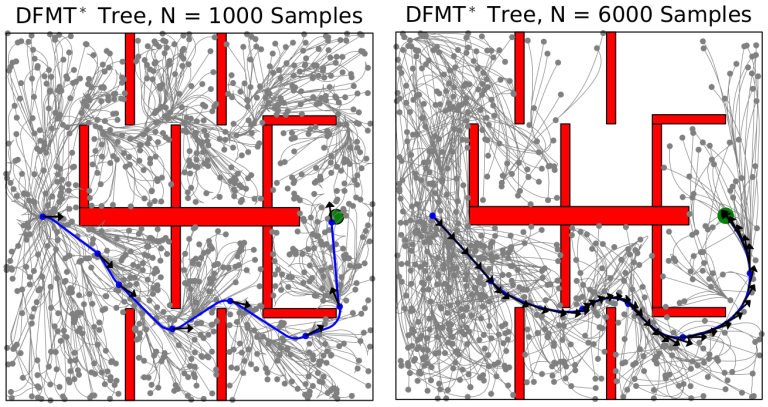
where $\boldsymbol{x}$ is the current robot state and $\boldsymbol{u}$ is the current control input. Note that the state space, $\mathcal{X}$, is not necessarily the same as the configuration space, C, but the configuration, q, is derivable from the state, $\boldsymbol{x}$. As we previously mentioned, we generally choose the configuration space to capture the information about the robot's state that is necessary for obstacle avoidance, but it may not be sufficient to completely capture the kinematics and dynamics.

We can extend the general RRT algorithm from Section 5.2 to handle the kinodynamic case by modifying how we add new nodes and edges to the tree. In particular, we start by sampling a random state, $\boldsymbol{x}$, from the state space, $\mathcal{X}$, and compute its nearest neighbor, $\boldsymbol{x}_{near}$, on the current tree. Instead of connecting $\boldsymbol{x}$ and $\boldsymbol{x}_{near}$ with a straight line, which is likely not a kinematically or dynamically feasible trajectory, we sample a random control, $\boldsymbol{u} \in \mathcal{U}$, and a random time, t. Then we propagate the state $\boldsymbol{x}_{near}$ forward by integrating the differential equation model in Equation (5.2) with the chosen control, $\boldsymbol{u}$, for the duration t to get a new state, $\boldsymbol{x}_{new}$. We add the resulting state, $\boldsymbol{x}_{new}$, to the tree if the path from $\boldsymbol{x}_{near}$ to $\boldsymbol{x}_{new}$ is within the free configuration space. This is referred to as a *forward-propagation-based* approach. Another approach to kinodynamic planning leverages *steering-based* algorithms. In these approaches, the planner selects two points in the state space, $\boldsymbol{x}$ and $\boldsymbol{x}'$, and then uses a steering subroutine to find a feasible trajectory to connect the states. Crucially, these approaches only work well in practice if the steering subroutine is computationally efficient, such as if the system is differentially flat. Figure 5.3 shows an example of a kinodynamic sampling-based planning algorithm called Differential FMT*[17] that is a varition of the FMT* algorithm we introduced in Section 5.4.

[17] 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

5.6 *Deterministic Sampling-Based Motion Planning*

Probabilistic sampling-based algorithms, such as PRM and RRT, have been quite successful in practice for robotic motion planning and often have nice theoretical properties with respect to probabilistic completeness or asymptotic optimality. These algorithms are *probabilistic* because they compute a path by connecting



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

Figure 5.3: Results from a kinodynamic planner called Differential FMT* (DFMT*). The figure on the left shows the results for a Reeds-Shepp car model, and on the right is a double integrator model.

independently and identically distributed random points in the configuration space. 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. Therefore, we can also explore *deterministic* approaches that can achieve similar or better theoretical guarantees and practical performance.

An important metric in the context of deterministic sampling-based motion planning is the l_2-dispersion.

Definition 5.6.1 (l_2-dispersion)**.** For a finite set of points, S, contained in $X \subset R^d$, its l_2-dispersion, $D(S)$, is defined as:

$$D(S) \coloneqq \sup_{x \in X} \min_{s \in S} \|s - x\|_2. \tag{5.3}$$

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 means that the points are more uniformly distributed since only a small radius ball can be fit among the points of S without touching or containing any of them.

In the context of deterministic sampling based motion planning, it is desirable to generate a set of samples, S, with low-dispersion. In fact, we can leverage low-dispersion sampling sequences 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 for us to create sequences of points, S, that *minimize* the l_2-dispersion. If the set S of n samples has l_2-dispersion that satisfies the inequality:

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

for some $\gamma > 0$, and if $\lim_{n \to \infty} n^{1/d} r = \infty$ for a connection radius, r, then the arc length of the path returned by the search algorithm can converge to the

optimal path as the number of samples approaches infinity, $n \to \infty$, just as for probabilistic planners[18].

[18] In fact, the deterministic approach can even take advantage of even smaller connection radii than probabilistic methods.

5.7 Exercises

The starter code for the exercises provided below is available online through GitHub. To get started, download the code by running in a terminal window:

```
git clone https://github.com/StanfordASL/pora-exercises.git
```

We denote Problems requiring hand-written solutions and coding in Python with ✎ and 🖳, respectively.

🖳 *Problem 1: Rapidly-Exploring Random Trees*

In this exercise, you will implement the RRT sample-based motion planning algorithm to plan paths in simple 2D environments. In the file `ch05/exercises/rrt.ipynb`, you will implement the key parts of the RRT algorithm and define a `GeometricRRT` planner that leverages simple straight line connections between nodes.

References

[24] 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.

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

[29] 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.

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

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

[58] 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.

Part II

Robot Perception

6
Introduction to Robot Sensors

The three main pillars of robotic autonomy are perception, planning, and control, which correspond to the see, think, and act stages of autonomy. The *perception* component consists of the numerous challenges associated with a robot sensing and understanding its environment, and a key element of perception is the sensors the robot uses to extract meaningful information about the world. In the next few chapters, we focus on the robot perception problem, and in particular we introduce common sensors utilized in robotics applications, discuss their key performance characteristics, and describe strategies for extracting useful information from the sensor measurements.

Robots operate in diverse environments which often require diverse sets of sensors for effective perception. For example, a self-driving car may utilize cameras, lidar, and radar for detecting objects in the environment. It also requires sensors for characterizing the physical state of the vehicle itself, such as inertial measurement units (IMU), 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

6.1 Sensor Classifications

We use the terms *proprioceptive* and *exteroceptive* to distinguish between sensors that measure the environment and sensors which measure quantities related the robot itself.

Definition 6.1.1 (Proprioceptive)**.** Proprioceptive sensors measure values internal to the robot. For example, a proprioceptive sensor might measure motor speed, wheel load, robot arm joint angles, or battery voltage.

Definition 6.1.2 (Exteroceptive)**.** Exteroceptive sensors acquire information from the robot's environment. For example, exteroceptive sensors measure distances to objects, light intensity, and sound amplitude.

Generally speaking, exteroceptive sensor measurements are more likely to require interpretation by the robot in order to extract meaningful environmental features. In addition to characterizing sensors based on what the measure, we also characterize sensors as *passive* or *active* based on how they operate.

Definition 6.1.3 (Passive Sensor)**.** Passive sensors, such as thermometers and cameras, measure ambient environmental energy entering the sensor.

Definition 6.1.4 (Active Sensor)**.** Active sensors, such as ultrasonic sensors, lidar and radar, emit energy into the environment and measure the reaction.

Classifying a sensor as active or passive is important because each exhibit unique characteristics and challenges. For example, passive sensors are heavily influenced by environmental conditions, such as a camera's reliance on good ambient lighting to take quality images.

6.2 Sensor Performance

Different types of sensors exhibit varying performance attributes. While some sensors maintain exceptional accuracy in controlled laboratory settings, their performance may suffer in natural real-world environments. Conversely, other sensors offer narrow, high-precision data across a variety of settings. We quantify and compare sensor performance characteristics by defining metrics related to *design specifications* and *in situ*[3] performance.

[3] In situ metrics quantify how well a sensor performs in the real environment.

6.2.1 Design Specification Metrics

A number of performance characteristics are specifically considered when designing a sensor, and which 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. We usually express this metric in decibels (dB), and compute it as:

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

   where r is the ratio between the upper and lower limits. In addition to the dynamic range ratio, the actual range is also an important sensor metric. For example, an optical rangefinder has a minimum operating range and gives 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. The lower limit of the dynamic range of a sensor is usually equal to its resolution[4].

   [4] 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. We usually express this metric in units of Hertz (Hz), which is measurements per second. High bandwidth sensors are

desirable so that downstream information can be updated at a high rate. For example, mobile robots may have to limit their maximum speed based on the bandwidth of their obstacle detection sensors.

6.2.2 *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, several important sensor metrics cannot be adequately characterized in lab 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 we define the error 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, for example we may state that a sensor has 97.5% accuracy. Therefore, small error corresponds to high accuracy and large error corresponds to low accuracy. For a measurement, m, and true value, v, we define the accuracy as $a := 1 - |m - v|/v$. Characterizing sensor accuracy is challenging since obtaining the true value, v, can be difficult or impossible.

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[5].

[5] A very precise sensor can still be highly inaccurate

6.2.3 *Sensor Errors*

When discussing in situ performance metrics such as accuracy and precision, it is important to be able to reason about the sources of sensor errors. In partic-

ular, 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 because they are deterministic and therefore reproducible and predictable. Calibration errors are a common source of systematic errors in sensors.

2. *Random errors* cannot be predicted using a sophisticated model since 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.

To reliably employ a sensor in practice, it is beneficial to characterize its systematic and random errors to allow for corrections that improve its accuracy and provide information about its precision. We refer to the process of quantifying sensor errors and identifying their origins as *error analysis*. This analysis often entails identifying all sources of systematic errors, modeling random errors[6], and assessing the cumulative effect of errors on the sensor's output.

[6] For example, using Gaussian distributions.

However, conducting a comprehensive error analysis can be difficult due to several factors. A significant challenge arises due to a *blurring* between systematic and random errors that is the result of changes to the operating environment. For instance, exteroceptive sensors on a mobile robot face varying measurement conditions as the robot navigates, with the sensor's performance potentially influenced by the robot's own movement. Therefore, an exteroceptive sensor's error profile may be heavily dependent on the particular environment and even the specific state of the robot[7]. This could cause errors to seem random as they occur unpredictably with the robot's movement, yet appear more systematic if the robot remains stationary. Therefore, while we can classify sensor errors as systematic or random in controlled environments, accurately characterizing these errors becomes substantially more complex in real-world settings.

[7] For example, active ranging sensors like lidar may fail based on the sensor's positioning relative to targets in the environment, and sonar sensors may return erroneous range readings due to specular reflections when aimed at certain angles against a smooth wall.

6.2.4 *Modeling Uncertainty*

If we could perfectly model and understand all systematic errors in sensor measurements we could theoretically correct for them. However, in practice, this is often not feasible. We therefore characterize uncertainty due to random errors by using *probability distributions*.

Given the practical challenge of identifying all sources of random error, we commonly make assumptions when modeling the error distribution. We commonly assume that random errors have a zero-mean, and that the distribution is symmetric and *unimodal*[8]. These assumptions can make mathematical analysis easier, but they also have limitations. For example, some assumptions, such as the unimodality of the distribution, may not hold true in real-world applications.

[8] A very common distribution that fits these properties is the Gaussian distribution.

Example 6.2.1 (Sensor Uncertainty Assumptions)**.** Consider a sonar sensor, an active sensor that uses acoustic pulses to measure distance. Suppose the sonar's accuracy is high, with random errors mainly stemming from noise from internal timing circuits. We could reasonably assume this noise is unimodal and possibly Gaussian. However, in scenarios where the sonar encounters materials causing coherent reflections, distance over-estimations become likely and could result in a bias towards positive errors. A comprehensive distribution that also captures this effect should be bimodal and asymmetric.

6.3 Common Sensors on Mobile Robots

6.3.1 Encoders

Encoders are proprioceptive electro-mechanical sensors that convert mechanical motion into a series of digital signals that can be interpreted to measure relative or absolute position measurements[9]. One common application of encoders in robotics is for sensing the rotation angle and speed of wheels or motors. This is important for being able to design good control laws for wheel speed control and motor-driven joints.

[9] Thanks to their extensive use across many domains, significant advancements have been made in developing affordable encoders that provide high resolution.

One common type of encoder is the *optical encoder*. Optical encoders work by directing light through slits in a rotating metal or glass disc onto a photodiode, creating sine or square wave pulses corresponding to the disc's rotation. We can then integrate the number of wave peaks to determine how much the disk has rotated. The encoder's resolution, expressed in cycles per revolution (CPR), determines its minimum angular resolution. In terms of bandwidth, it is critical that the encoder is sufficiently fast to handle the expected shaft rotation rates[10]. Quadrature encoders are also common in robotics applications to additionally sense the direction of rotation.

[10] Encoder bandwidth is generally not a concern in mobile robot applications.

As with most proprioceptive sensors, encoders typically operate in a very predictable and controlled environment and we can account for their systematic errors and cross-sensitivities. In practice, we often assume perfect accuracy of optical encoders since their errors are typically dwarfed by errors in downstream components.

6.3.2 Heading Sensors

Heading sensors can be proprioceptive or exteroceptive[11] and are used to determine the robot's orientation in space. They can also be used to obtain position estimates by fusing the orientation and velocity information and integrating through a process known as *dead reckoning*.

[11] For example, gyroscopes and inclinometers are proprioceptive and compasses are exteroceptive.

Compasses: Compasses are exteroceptive sensors that measure the earth's magnetic field to provide an estimate of direction. In mobile robotics, digital compasses using the Hall effect are popular and 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. The inertial stability increases with the spinning speed, ω, the precession speed, Ω, and the wheel's inertia, I, since we can express the reactive torque, τ, by:

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

Mechanical gyroscopes are built with an inner and outer gimbal that isolates the rotor axis from external torques to keep it space-stable. 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. We can therefore estimate angular velocity since it is proportional to the frequency difference. In modern optical gyroscopes, bandwidth can easily exceed 100 kHz, while resolution can be smaller than 0.0001 degrees/hr.

6.3.3 *Accelerometers*

An accelerometer is a device used to measure net acceleration due to external forces, including gravity. Mechanical accelerometers are essentially spring-mass-damper systems that we model by the second order differential equation[12]:

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

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 is:

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

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

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

cantilevered beam and a proof mass. The deflection of the proof mass from its neutral position is measured using capacitive or piezoelectric effects.

6.3.4 *Inertial Measurement Unit (IMU)*

Inertial measurement units (IMU) are devices that use gyroscopes and accelerometers to estimate relative position, orientation, velocity, and acceleration with respect to an inertial reference frame. We show their general working principle in Figure 6.1.

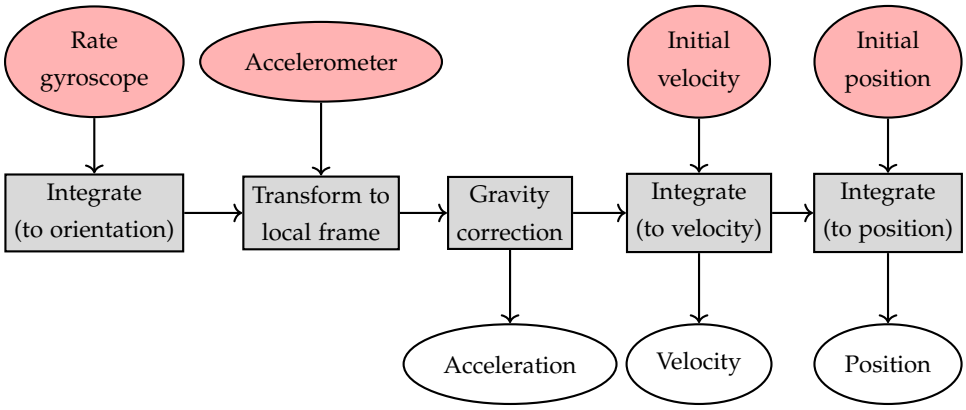


Figure 6.1: Inertial measurement unit (IMU) block diagram.

First, we integrate gyroscope data to estimate the vehicle orientation while the three accelerometers estimate the instantaneous acceleration along each axis. We then transform the acceleration into the local navigation frame using the current estimate of the vehicle orientation relative to gravity and subtract the gravity vector from the measurement. Next, we integrate the resulting acceleration to obtain the velocity and integrate again to compute the position, provided that we know both the initial velocity and position.

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 component will also effect the downstream components. For example, drift in the gyroscope leads to errors in 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. We can account for drift by using periodic references to some external measurement, such as GNSS position measurements, cameras, or other sensors.

6.3.5 *Beacons*

Beacons are signaling devices with precisely known positions[13]. We can determine the position of a mobile robot by knowing the position of the beacon and

[13] Stars and lighthouses are classic examples.

by having access to relative position measurements. The GNSS positioning system and indoor camera-based motion capture systems are advanced examples of beacons. GNSS based positioning[14], which is extremely popular in robotics and other fields, works by processing synchronized signals from at least four satellites. We need signals from at least four satellites to estimate the four unknown quantities: the three position coordinates and a clock correction variable.

[14] We can also use modified GNSS-based methods, such as *differential GPS*, to increase positioning accuracy.

6.3.6 *Active Ranging*

Active ranging sensors provide direct distance measurements to objects in the vicinity of the sensor. These sensors are important in robotics for localization and environment reconstruction. There are two main types of active ranging sensors, time-of-flight active ranging sensors and geometric active ranging sensors[15].

[15] Examples of time-of-flight sensors include ultrasonic, laser rangefinder, and time-of-flight cameras, and examples of geometric sensors include optical triangulation and structured light sensors.

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:

$$d = ct,$$

where d is the distance traveled, c is the speed of wave propagation, and t is the time of flight. Note that the time of flight is significantly smaller when using electromagnetic signals, on the order of nanoseconds for distance on the order of meters, which can make these types of sensors more challenging to develop in an affordable and robust way. In general, the quality of different time-of-flight range sensors depends on several factors including uncertainties in determining the exact time of arrival of the reflected signal, inaccuracies in the time of flight measurement, the dispersal cone of the transmitted beam[16], interaction with the target[17], and the speed of the mobile robot and dynamics targets.

[16] Mainly with ultrasonic range sensors.

[17] For example, surface absorption, specular reflections.

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 we can use geometric properties to analyze the reflection and estimate range via triangulation. Optical triangulation sensors (1D) transmit a collimated 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 such as a point, line, or texture, onto the environment. The reflection is captured by a receiver and then, together with known geometric values, we can estimate range via triangulation.

6.3.7 *Other Sensors*

Some classical examples of other sensors include radar, tactile sensors, and vision based sensors like cameras. Radar sensors leverage the Doppler effect to

produce relative velocity measurements. Tactile sensors are particularly useful for robots that interact physically with their environment.

6.4 Computer Vision

Vision sensors have become crucial 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[18]. 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. We generally refer to the analysis and processing of images 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, we will focus on cameras and camera models.

[18] The human eye provides millions of bits of information per second.

6.4.1 Digital Cameras

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[19], 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 called pixels, and can contain between 20,000 and several million pixels total. We can think of each pixel 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.

[19] The duration of exposure is usually expressed as the shutter speed, such as 1/125, 1/60, or 1/30 of a second.

6.4.2 Image Formation

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 reflected light rays on a photoreceptive surface such as a CCD or a CMOS sensor. Since light reflecting off an object is generally scattered in many directions, exposing a planar photoreceptive surface to these reflected rays would result in many rays being captured at each pixel, which would lead to blurry images. A solution to this issue is to add a barrier in front of the photoreceptive surface that only lets some of the rays pass through an aperture, as we show in Figure 6.2. The earliest approach to filtering light rays in this way was to have a small hole in the barrier surface. We refer to cameras with this type of filter as *pinhole* cameras.

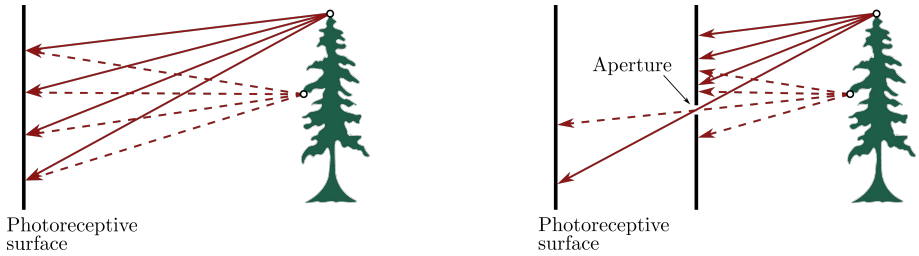


Figure 6.2: Light rays on a photoreceptive surface, referred to as the image plane. On the left, numerous rays being reflected and scattered by the object leads to blurry images whereas, on the right, a barrier has been added so that the scattered light rays can be distinguished.

6.4.3 *Pinhole Camera Model*

A pinhole camera has no lens but rather a single small aperture. Light from the scene passes through this pinhole aperture and projects an inverted image onto the image plane, as we show in Figure 6.3. While modern cameras do not operate in this way, we can use the principles of the pinhole camera to derive useful mathematical models.

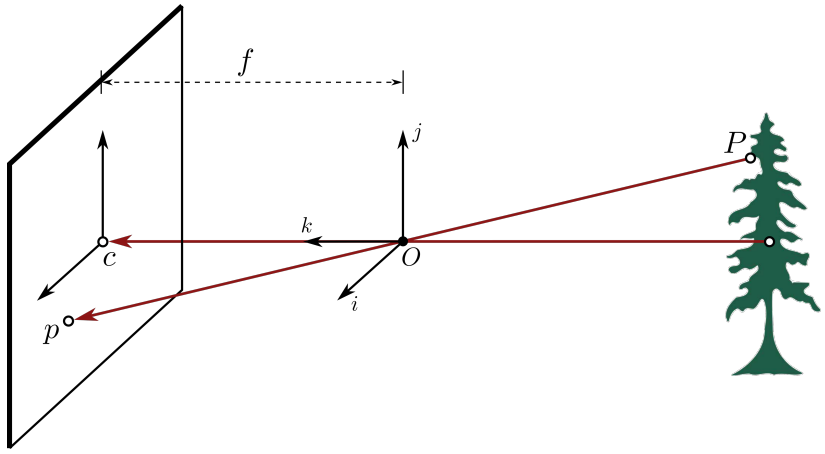


Figure 6.3: Pinhole camera model. Due to the geometry of the pinhole camera system, the object's image is inverted on the image plane. In this figure, O is the camera center, c is the image center, and p the principal point.

We start by defining several useful references to help develop the mathematical pinhole camera model. First, the *camera reference frame* is centered at a point, O, that is at a focal length, f, in front of the image plane, as we show in Figure 6.3. We define this reference frame, with directions (i, j, k), such that the k axis is coincident with the *optical axis* that points toward the image plane. We denote the coordinates of a point in the camera frame by $P = (X, Y, Z)$. 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, we can deduce the following relationships between the coordinates $P = (X, Y, Z)$ and $p = (x, y, z)$:

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

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

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

From the geometry of the camera, we can see that $z = f$ where f is the focal length, such that we can rewrite these expressions as:

$$x = f\frac{X}{Z}, \quad y = f\frac{Y}{Z}. \tag{6.1}$$

Therefore, we can compute the position of the pixel on the image plane that captures a ray of light from the point P.

6.4.4 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 image blurring. A small aperture lets through fewer light rays, but the resulting image is darker. As a solution, lenses focus light by refraction and can be used to replace the aperture, avoiding the need for these trade-offs.

We can develop a mathematical model for lenses similar to the pinhole model by using properties from Snell's law. Figure 6.4 shows a diagram of the most basic lens model, which is the *thin lens model*[20]. 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. We develop a mathematical model similar to Equation (6.1) from the geometry of similar triangles:

$$\frac{y}{Y} = \frac{z}{Z}, \quad \frac{y}{Y} = \frac{z - f}{f} = \frac{z}{f} - 1, \tag{6.2}$$

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

[20] The *thin lens model* assumes no optical distortion due to the curvature of the lens.

these two equations yields the *thin lens equation*:

$$\frac{1}{z} + \frac{1}{Z} = \frac{1}{f}. \tag{6.3}$$

Note that in this model, and 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. In practice, an acceptable focus is possible within some range of distances referred to as 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. We can use this formula to estimate the distance to an object if we know the focal length, f, and the current distance of the image plane to the lens, z. This technique is called *depth from focus*.

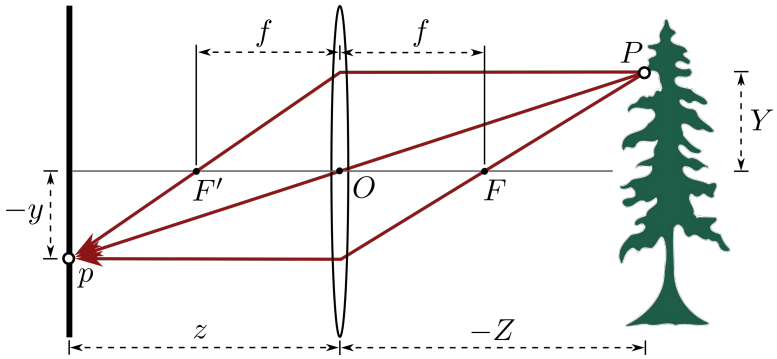


Figure 6.4: The thin lens model.

References

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

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

7
Camera Models and Calibration

We introduced the problem of robot perception in Chapter 6, and described various types of sensors that robots use to understand their environment[1]. Cameras are one sensing modality that is particularly important for many robotics applications due to their ability to capture an enormous amount of information in every image. Extracting information from an image that is relevant to the robot, a process we refer to as *image processing* or *computer vision*, is another aspect of robot perception that is quite challenging.

In this chapter, we begin to explore the problem of computer vision by focusing on some of the fundamental mathematical tools for calibrating cameras and processing their images to extract useful information about the scene[2,3]. Specifically, we leverage the *pinhole camera model* from Chapter 6 to explore the fundamental image processing task of *perspective projection*, which is the task of determining how a particular point in the scene maps to a point in the camera image.

[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

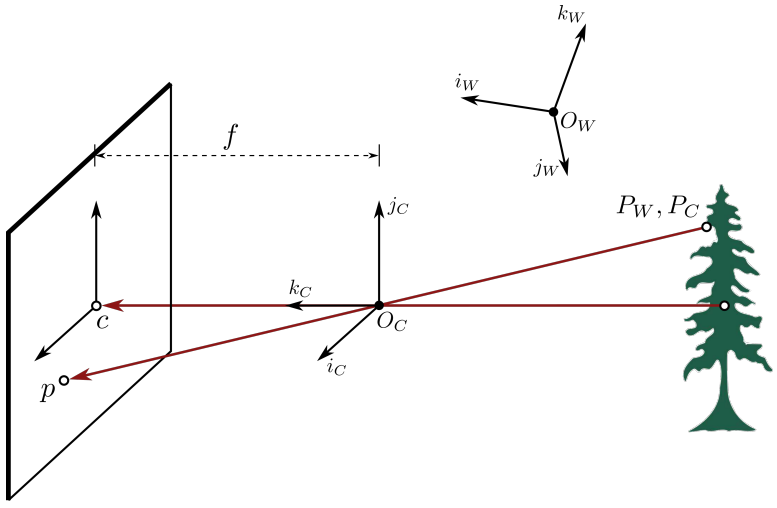


Figure 7.1: Graphical representation of the pinhole camera model. In this model, the point O_C is the camera center, c is the image center, and f is the focal length of the camera. We assume that all light rays from point P in the scene pass through point O_C and are captured on the image plane at point p.

7.1 Perspective Projection

The pinhole camera model, which we show graphically in Figure 7.1, can be used to mathematically define relationships between points, P, in the scene and points, p, on the image plane. Notice that we can represent any point P in the scene in two ways: in camera frame coordinates, denoted as P_C, or in world frame coordinates, denoted as P_W. Our overall objective of this section is to derive a mathematical model that we can use to map a point, P_W, expressed in world frame coordinates to a point, p, on the image plane. We accomplish this by combining two transformations together, a transformation of P from 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.

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

The first step we consider is how to map 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. In Chapter 6, we presented the pinhole camera equations:

$$x = f\frac{X_C}{Z_C}, \quad y = f\frac{Y_C}{Z_C}, \tag{7.1}$$

where $P_C = (X_C, Y_C, Z_C)$, $p = (x, y)$, and f is the focal length of the pinhole camera[4].

[4] We generally do not include the z term of p simply because $z = f$ is a fixed value.

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*. Pixel coordinates are generally defined with respect to a reference frame in the lower corner of the image plane to avoid negative coordinates. We show this new reference frame in Figure 7.2, where we define the image center, c, with coordinates $(\tilde{x}_0, \tilde{y}_0)$, where $\tilde{(\cdot)}$ is the notation we use to denote a coordinate with respect to this new reference frame. In this new reference frame, we map the

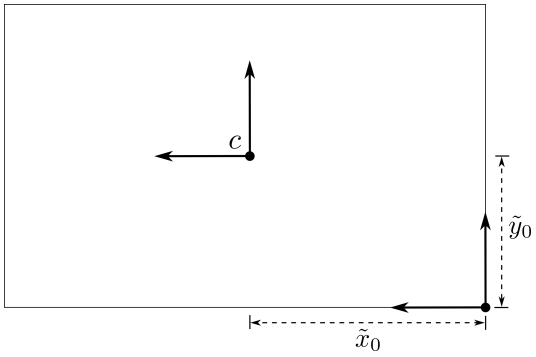


Figure 7.2: We define a new reference frame with coordinates denoted by $\tilde{(\cdot)}$ with its origin in the lower corner of the image plane. The image center coordinates in this new frame are $(\tilde{x}_0, \tilde{y}_0)$.

point P_C to the coordinates $(\tilde{x}, \tilde{y})$ by:

$$\tilde{x} = f\frac{X_C}{Z_C} + \tilde{x}_0, \quad \tilde{y} = f\frac{Y_C}{Z_C} + \tilde{y}_0. \tag{7.2}$$

Finally, given the number of pixels per unit distance, we can map these new coordinates to pixel coordinates (u, v) by:

$$u = \alpha \frac{X_C}{Z_C} + u_0, \quad v = \beta \frac{Y_C}{Z_C} + v_0, \tag{7.3}$$

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.

Note that the transformation from the point P_C in camera frame coordinates to p in pixel coordinates given by Equation (7.3) is not linear. However, we can represent this transformation as a linear mapping[5] through an additional change of coordinates. In particular, we will express the points P_C and p in *homogeneous coordinates*.

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

For a two-dimensional point (x_1, x_2) or a three-dimensional point (x_1, x_2, x_3) in Euclidean space, we represent the point in homogeneous coordinates by the transformation:

$$(x_1,\, x_2) \to (\alpha x_1,\, \alpha x_2,\, \alpha)\,, \quad \text{and} \quad (x_1,\, x_2,\, x_3) \to (\alpha x_1,\, \alpha x_2,\, \alpha x_3,\, \alpha)\,, \tag{7.4}$$

for any $\alpha \neq 0$. These new coordinates are called homogeneous coordinates because we can choose the scaling factor, α, arbitrarily as long as $\alpha \neq 0$. We transform a set of homogeneous coordinates back by:

$$(y_1,\, y_2,\, y_3) \to \left(\frac{y_1}{y_3},\, \frac{y_2}{y_3}\right), \quad \text{and} \quad (y_1,\, y_2,\, y_3,\, y_4) \to \left(\frac{y_1}{y_4},\, \frac{y_2}{y_4},\, \frac{y_3}{y_4}\right). \tag{7.5}$$

We will denote when a point is described in homogeneous coordinates using the superscript h. For example, we express the point $P_C = (X_C, Y_C, Z_C)$ in camera frame coordinates with $\alpha = 1$ in homogeneous coordinates by:

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

and we can express the pixel coordinate $p = (u, v)$ 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 from Equation (7.3). With the expression of these points in homogeneous coordinates, we can see that their relationship is transformed from the nonlinear relationship in Equation (7.3) to the *linear* relationship:

$$\begin{bmatrix} \alpha & 0 & u_0 & 0 \\ 0 & \beta & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha X_c + u_0 Z_c \\ \beta Y_c + v_0 Z_c \\ Z_c \end{bmatrix}. \tag{7.6}$$

Often, in practice, we also add a skewness parameter, γ[6], and we can write

[6] The skewness parameter generally ends up being close to zero.

this linear relationship in the more compact form:

$$\begin{bmatrix} K & 0_{3\times 1} \end{bmatrix} P_C^h = p^h, \quad K \coloneqq \begin{bmatrix} \alpha & \gamma & u_0 \\ 0 & \beta & v_0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{7.7}$$

We refer the matrix K in Equation (7.7) as the *camera matrix* or *matrix of intrinsic parameters* 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, we often estimate them in practice by performing a camera calibration.

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

Recall from Figure 7.1 that we can express a point, P, in the scene either in terms of camera frame coordinates, P_C, or world frame coordinates, P_W. While we discussed the use of the pinhole model to map P_C coordinates to pixel coordinates, p, in the previous section, in this section we discuss the mapping between the camera and world frame coordinates of the point P, as we show in Figure 7.3.

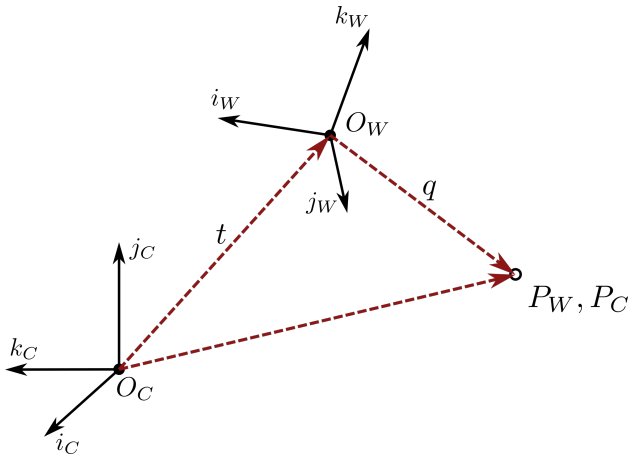


Figure 7.3: A depiction of the point P expressed either in camera coordinates, P_C, or in world frame coordinates, P_W. We denote the world frame origin by O_W and the camera frame origin by O_C.

From Figure 7.3, we can write P_C as:

$$P_C = t + q, \tag{7.8}$$

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 the same vector as P_W, just expressed with respect to a different coordinate frame. The coordinates are related by a rotation:

$$q = RP_W, \tag{7.9}$$

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

as:

$$R := \begin{bmatrix} i_w \cdot i & j_w \cdot i & k_w \cdot i \\ i_w \cdot j & j_w \cdot j & k_w \cdot j \\ i_w \cdot k & j_w \cdot k & k_w \cdot k \end{bmatrix}, \tag{7.10}$$

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, we can map the point P_W to camera frame coordinates P_C by:

$$P_C = t + RP_W, \tag{7.11}$$

where t is the vector in camera frame coordinates from O_C to O_W and R is the rotation matrix defined in Equation (7.10). Similar to the previous section, we can equivalently express this transformation for the case where the points P_W and P_C are expressed in homogeneous coordinates:

$$\begin{bmatrix} P_C \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0_{1\times 3} & 1 \end{bmatrix} \begin{bmatrix} P_W \\ 1 \end{bmatrix}. \tag{7.12}$$

7.1.3 *Mapping World Frame Coordinates to Image Coordinates ($P_W \to p$)*

The objective of the perspective projection task is to find a way to mathematically relate the position of a point in world frame coordinates, denoted P_W, to the corresponding pixel coordinates, p, on the image plane. With the relationship from Equation (7.12) that we developed for mapping P_W to the camera frame coordinates, P_C, and the relationship in Equation (7.7) for mapping P_C to pixel coordinates, p, we can now define the direct mapping from P_W to p. In particular, combining the two transformations 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 we can simplify to:

$$p^h = K \begin{bmatrix} R & t \end{bmatrix} P_W^h. \tag{7.13}$$

In Equation (7.13), P_W^h is the homogeneous coordinate representation of P_W and p^h is the homogeneous coordinate representation of p. Recall that the matrix $K \in \mathbb{R}^{3\times 3}$ is the matrix of intrinsic camera parameters, and the matrix $[R \;\; t] \in \mathbb{R}^{3\times 4}$ contains *extrinsic* parameters[7]. 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.

[7] Extrinsic parameters describe the camera's position and orientation relative the points in the scene.

7.2 *Camera Calibration: Direct Linear Method*

Before we can use the expression in Equation (7.13) in practice, we need to determine the camera's intrinsic and extrinsic parameters, denoted by the parameters K, R, and t. One approach is to use the direct linear transformation method for camera calibration[8], which requires a set of known correspondences, $p_i \leftrightarrow P_{W,i}$ for $i = 1, \ldots, n$.

[8] 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

7.2.1 Direct Linear Calibration: Step 1

For direct linear calibration, the first step is to write each corresponding pair of points, $p_i = (u_i, v_i)$ and $P_{W,i} = (X_{W,i}, Y_{W,i}, Z_{W,i})$, in homogeneous coordinates and then use the expression in Equation (7.13) to write:

$$p_i^h = MP_{W,i}^h, \quad i = 1, \ldots, n, \tag{7.14}$$

where we refer to $M = K[R \;\; t]$ as the *homography*. Next, we use the n correspondences to estimate the homography, M, and then later we can extract the intrinsic and extrinsic parameters from M. A useful first step to determine M is to rewrite it in terms of its rows:

$$M = \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix}, \tag{7.15}$$

where $m_i \in \mathbb{R}^{1\times 4}$ is the i-th row of M. By considering the rows of M individually, we can write the relationship in Equation (7.14) as:

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

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, by some algebraic manipulation, yields the expressions:

$$\begin{aligned} u_i(m_3 \cdot P_{W,i}^h) - (m_1 \cdot P_{W,i}^h) = 0, \quad i = 1, \ldots, n \\ v_i(m_3 \cdot P_{W,i}^h) - (m_2 \cdot P_{W,i}^h) = 0, \quad i = 1, \ldots, n. \end{aligned} \tag{7.16}$$

We can now combine these $2n$ equations together in one large matrix equation:

$$\tilde{P}m = 0, \quad m \coloneqq \begin{bmatrix} m_1^\top \\ m_2^\top \\ m_3^\top \end{bmatrix}, \tag{7.17}$$

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 $P_{W,i}^h$. For a more concrete representation of how we define $\tilde{P}$, the first couple rows are given by:

$$\tilde{P} = \begin{bmatrix} -(P_{W,1}^h)^\top & 0_{1\times 4} & u_1(P_{W,1}^h)^\top \\ 0_{1\times 4} & -(P_{W,1}^h)^\top & v_1(P_{W,1}^h)^\top \\ -(P_{W,2}^h)^\top & 0_{1\times 4} & u_2(P_{W,2}^h)^\top \\ \vdots & \vdots & \vdots \end{bmatrix}. \tag{7.18}$$

Note that we must have at least six correspondences, $n \geq 6$, to ensure that m is uniquely defined. With this sufficient number of correspondences, we could ideally directly solve Equation (7.17). However, in practice, a more robust procedure is to build $\tilde{P}$ with more than 6 points, which gives an overdetermined set of equations that may not have a solution[9]. Therefore, to compute m, we formulate an optimization problem:

$$\begin{aligned} \min_{m} \ & \|\tilde{P}m\|^2, \\ \text{s.t.} \ & \|m\|^2 = 1, \end{aligned} \tag{7.19}$$

where the constraint $\|m\|^2 = 1$ is required to ensure that the optimization problem cannot be solved by trivially choosing $m_i = 0$ for each $i = 1, \ldots, 12$. We call this optimization problem a *constrained least-squares* problem.

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

Example 7.2.1 (Constrained Least-Squares Optimization)**.** 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:

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

and the necessary optimality conditions:

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

We can write the first necessary optimality condition as $A^\top A x = \lambda x$, 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 optimum is the eigenvector associated with the smallest eigenvalue. We can efficiently compute this eigenvector by using a singular value decomposition of $A = U\Sigma V^\top$ and then choosing x to be the column of V associated with the smallest singular value, since $A^\top A = V\Sigma^2 V^\top$.

7.2.2 Direct Linear Calibration: Step 2

Once we have solved the optimization problem in Equation (7.19) to compute the vector 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 step, we will express the matrix M 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. We can factorize M as:

$$M = K \begin{bmatrix} R & t \end{bmatrix}, \tag{7.20}$$

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

$$\begin{bmatrix} c_1 & c_2 & c_3 \end{bmatrix} = KR, \tag{7.21}$$

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

7.2.3 A Flexible Camera Calibration Method

The homography, M, is defined for a specific set of extrinsic parameters R and t. In practice, however, it might be desirable for us to estimate the camera's intrinsic parameters from N different images from different perspectives, and therefore with N different homographies due to the varying extrinsic parameters. In this case, we can apply an alternative procedure[10] to the direct linear calibration method to extract the intrinsic parameters, K.

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

We begin 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, such as a checkerboard pattern, on a planar surface. In this case, we can assume that the world frame origin lies on the plane such that $Z_W = 0$ for all points on the plane. Since $Z_W = 0$, we can simplify the relationship between p^h and P_W^h given by Equation (7.13) to:

$$p^h = \tilde{M}\tilde{P}_W^h, \tag{7.22}$$

with:

$$\tilde{M} = K \begin{bmatrix} r_1 & r_2 & t \end{bmatrix}, \quad \tilde{P}_W^h = \begin{bmatrix} X_W & Y_W & 1 \end{bmatrix}^\top, \tag{7.23}$$

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 we can still estimate the homography matrix, $\tilde{M}$, using the same procedure discussed earlier.

Next, we identify a set of constraints on the intrinsic parameter matrix, K, by writing the homography, $\tilde{M}$, as:

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

and noting that since r_1 and r_2 are orthonormal we have:

$$\tilde{c}_1^\top B \tilde{c}_2 = 0, \quad \tilde{c}_1^\top B \tilde{c}_1 = \tilde{c}_2^\top B \tilde{c}_2, \tag{7.24}$$

where $B = K^{-\top}K^{-1} \in \mathbb{R}^{3\times 3}$ is a symmetric matrix. We can therefore solve for the intrinsic camera parameters, K, by using the constraints in Equation (7.24) to solve for the symmetric matrix B and then backing out the parameters that define K. To compute the matrix B from the constraints in Equation (7.24), we can employ several useful tricks. The main trick is to notice that even though

B consists of nine parameters, it is symmetric, and, therefore, we only need six parameters to specify it fully. Therefore, we reparameterize the matrix $B \in \mathbb{R}^{3\times 3}$ as a vector $b \in \mathbb{R}^6$ as:

$$b = \begin{bmatrix} B_{11} & B_{12} & B_{22} & B_{13} & B_{23} & B_{33} \end{bmatrix}^\top. \tag{7.25}$$

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

$$\tilde{c}_i^\top B \tilde{c}_j = v_{ij}^\top b, \tag{7.26}$$

where:

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

and 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, we can rewrite the constraints in Equation (7.24) 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:

$$\begin{bmatrix} v_{12}^\top \\ (v_{11} - v_{22})^\top \end{bmatrix} b = 0, \tag{7.27}$$

which is linear with respect to the unknowns vector b. Importantly, while the homographies, M, are different for each image due to the different extrinsic parameters, the intrinsic camera parameters represented by the vector b are the same. Therefore, with N images from the same camera, even with potentially different perspectives, we can stack the constraints in Equation (7.27) to give:

$$Vb = 0, \tag{7.28}$$

where $V \in \mathbb{R}^{2N\times 6}$. In the case where we include the skewness parameter, γ, in K, there must be $N \geq 3$ images in order to specify B uniquely. Similarly to the approach for computing the homography in the previous section, we can compute the vector b as the solution to the constrained least squares problem:

$$\begin{aligned} &\min_b \ \|Vb\|^2, \\ &\text{s.t. } \|b\|^2 = 1. \end{aligned} \tag{7.29}$$

Once we have computed b, we can solve for the intrinsic camera parameters, K, by leveraging the definition of $B = K^{-T}K^{-1}$. In particular, we compute the

intrinsic parameters by:

$$\begin{aligned} v_0 &= \frac{B_{12}B_{13} - B_{11}B_{23}}{B_{11}B_{22} - B_{12}^2}, \\ \lambda &= B_{33} - \frac{B_{13}^2 + v_0(B_{12}B_{13} - B_{11}B_{23})}{B_{11}}, \\ \alpha &= \sqrt{\frac{\lambda}{B_{11}}}, \\ \beta &= \sqrt{\frac{\lambda B_{11}}{B_{11}B_{22} - B_{12}^2}}, \\ \gamma &= \frac{-B_{12}\alpha^2\beta}{\lambda}, \\ u_0 &= \frac{\gamma v_0}{\beta} - \frac{B_{13}\alpha^2}{\lambda}, \end{aligned} \tag{7.30}$$

where we can think of λ 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 we have extracted the camera intrinsic parameters, K, from this procedure, given any new homography, $\tilde{M}$, we can compute the extrinsic parameters by:

$$\begin{aligned} r_1 &= \frac{K^{-1}\tilde{c}_1}{\|K^{-1}\tilde{c}_1\|}, \\ r_2 &= \frac{K^{-1}\tilde{c}_2}{\|K^{-1}\tilde{c}_2\|}, \\ r_3 &= r_1 \times r_2, \\ t &= \frac{K^{-1}\tilde{c}_3}{\|K^{-1}\tilde{c}_1\|}. \end{aligned} \tag{7.31}$$

As one final step, we note that the matrix R defined with column vectors r_1, r_2, and r_3 will not generally satisfy the orthonormality property of a rotation matrix. We can correct this issue by again using optimization methods to compute a valid rotation matrix that best corresponds to these column vectors:

$$\begin{aligned} &\min_R \ \|R - Q\|^2, \\ &\text{s.t. } R^\top R = I, \end{aligned} \tag{7.32}$$

where:

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

We solve this problem by choosing $R = UV^\top$, where U and V are defined by the singular value decomposition of $Q = U\Sigma V^\top$.

7.3 Camera Auto-Calibration

The direct linear transformation from Section 7.2 and Zhang's flexible calibration method from Section 7.2.3 require point correspondences to calculate the intrinsic and extrinsic parameters. Auto-calibration offers an alternative approach that does not make this assumption by utilizing multiple views of a static scene to determine the camera's intrinsic and extrinsic parameters. This approach leverages the fact that a camera's intrinsic parameters remain constant across different views of the same scene. By identifying correspondences between points in multiple images using an algorithm like SIFT[11], we can estimate the intrinsic matrix, K, and extrinsic matrices, R and t, by bundle adjustment[12] based on the static scene geometry. In total, the camera auto-calibration process consists of five key steps.

[11] David G Lowe. "Object recognition from local scale-invariant features". In: *Proceedings of the seventh IEEE international conference on computer vision*. Vol. 2. Ieee. 1999, pp. 1150–1157

[12] Bill Triggs et al. "Bundle adjustment—a modern synthesis". In: *Vision Algorithms: Theory and Practice: International Workshop on Vision Algorithms Corfu, Greece, September 21–22, 1999 Proceedings*. Springer. 2000, pp. 298–372

First, we use an algorithm such as SIFT to identify key points in the scene and their correspondence points across multiple images from different views. The SIFT algorithm is robust in detecting and describing local features in images, making it effective for finding correspondences under varying conditions.

Second, we use the correspondences between a pair of images to compute the *fundamental matrix*. The fundamental matrix, $F \in \mathbb{R}^{3\times 3}$, relates corresponding points between a pair of images of the same scene[13]. For a pair of images (I, I') of the same scene, the fundamental matrix, F, satisfies:

[13] The fundamental matrix describes the *epipolar geometry* between two image views.

$$p^\top F p' = 0, \tag{7.33}$$

where p and p' are corresponding points in images I and I', respectively. We will discuss the fundamental matrix more in the context of stereo vision in Chapter 8.

To compute the fundamental matrix for the image pair, we first construct a point correspondence matrix, W, where each row represents a correspondence between points in the two images. Let (u_i, v_i) and (u_i', v_i') be the coordinates of one set of matched points, p_i and p_i'. We form each row of the matrix W using one set of corresponding coordinates as:

$$W = \begin{bmatrix} u_1u_1' & u_1v_1' & u_1 & v_1u_1' & v_1v_1' & v_1 & u_1' & v_1' & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ u_nu_n' & u_nv_n' & u_n & v_nu_n' & v_nv_n' & v_n & u_n' & v_n' & 1 \end{bmatrix} \tag{7.34}$$

From this point correspondence matrix, we then compute the fundamental matrix by solving the linear system $Wf = 0$ using a singular value decomposition (SVD), where f is the vectorized form of the fundamental matrix. We describe the specifics of this procedure in more detail in Section 8.1.1.

Third, we compute the *essential matrix*, which is similar to the fundamental matrix in that it relates corresponding points in two images based on scene geometry. For two images, I and I', we define the essential matrix by the rotation matrix, R, and translation vector, t, relating the coordinate frames of the two

images by:

$$E = [t]_\times R, \tag{7.35}$$

where $[t]_\times$ is the matrix representation of the cross product. Assuming the intrinsic matrix, K, remains the same between the images, we can also define the essential matrix, E, with respect to the fundamental matrix as:

$$E = K^\top F K. \tag{7.36}$$

Therefore, we can first compute the essential matrix from the previously computed fundamental matrix and the intrinsic parameter matrix, K, and then we can compute the camera extrinsic rotation and translation parameters, R and t, by the singular value decomposition:

$$E = U \Sigma V^\top, \tag{7.37}$$

where $\Sigma = \text{diag}(1, 1, 0)$. From this decomposition, there are two possible solutions for the camera extrinsic parameters:

$$\begin{aligned} R_1 &= UWV^\top, & R_2 &= UW^\top V^\top, \\ t_1 &= U[:, 2], & t_2 &= -U[:, 2], \end{aligned} \tag{7.38}$$

where:

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{7.39}$$

We can identify the correct solution by using each to compute the three-dimensional points from the two-dimensional matched correspondence points, as we describe in the next step. The extrinsic parameters, R and t, we should use are the ones that ensure that most of the three-dimensional points lie in front of both cameras, meaning they will have positive depth values.

The fourth step is to perform triangulation to compute the 3D points in the scene from the 2D image correspondence points. We define the projection matrix, P, for an image as:

$$P = K \begin{bmatrix} R & t \end{bmatrix}, \tag{7.40}$$

where again K is the camera intrinsic matrix, R is the rotation matrix, and t is the translation vector. This projection matrix maps a 3D point in homogeneous coordinates into the 2D camera frame coordinates by:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = P \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}. \tag{7.41}$$

Therefore, for matched points across two images, we can compute the 3D coordinates by solving the linear system:

$$\begin{bmatrix} uP_3^\top - P_1^\top \\ vP_3^\top - P_2^\top \\ u'P_3'^\top - P_1'^\top \\ v'P_3'^\top - P_2'^\top \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = 0, \tag{7.42}$$

where (u, v) are the coordinates and P_1, P_2, and P_3 are the first, second, and third rows of the projection matrix, P, for image I, respectively, and (u', v') are the coordinates and P_1', P_2', and P_3' are the rows of the projection P' for image I'. We can solve Equation (7.42) using a least squares method.

So far, we have used correspondences in pairs of images from the same camera of the same scene to estimate the image extrinsics, R and t, and computed estimates of the 3D scene points by triangulation. These computations require knowledge of the camera intrinsic matrix, K, which is the quantity we are trying to estimate. We can leverage the previous steps to compute K by using an iterative optimization-based procedure, where we begin with an estimate[14] of K and refine it until convergence. In particular, we refine the estimate of the camera intrinsics and extrinsics by solving the optimization:

[14] For example, we could start with an estimate by referencing the camera manufacturer's data, or using some other simpler method.

$$\sum_{i=1}^{N} \sum_{j=1}^{M} \| p_{ij} - P_i(K, R_i, t_i) X_j \|^2, \tag{7.43}$$

where p_{ij} is an observed 2D point in image i that corresponds to the j-th 3D point, X_j is the j-th estimated 3D point, $P_i(K, R_i, t_i)$ is the projection matrix for image i, M is the total number of triangulated 3D points, and N is the total number of images. Note that the each projection matrix is a function of the intrinsic parameters, which are constant across all images, as well as the extrinsic parameters for the image. We can optimize this cost function by applying a nonlinear method, such as Levenberg-Marquardt, to compute a new set of parameters. We then repeat the steps listed above, computing new extrinsics, triangulation points, and optimizing, until convergence.

7.4 Challenges

7.4.1 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 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. We show images showing barrel and pincushion distortion in Figure 7.4.

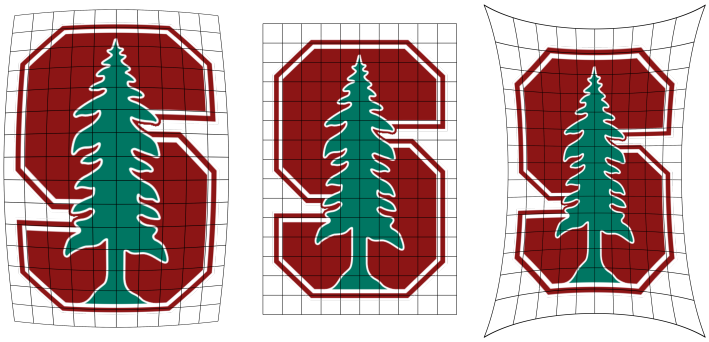


Figure 7.4: Different kinds of radial distortions that are seen in real lenses, which may affect the accuracy of the pinhole camera model.

There are methods we can use 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:

$$\begin{bmatrix} u_d \\ v_d \end{bmatrix} = \begin{bmatrix} u_d \\ v_d \end{bmatrix} (1 + kr^2) \begin{bmatrix} u - u_{cd} \\ v - v_{cd} \end{bmatrix} + \begin{bmatrix} u_{cd} \\ v_{cd} \end{bmatrix}, \tag{7.44}$$

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.

7.4.2 *Measuring Depth*

Once we know the camera intrinsic and extrinsic parameters, K, R, and t, 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 Equation (7.14) not being invertible, but intuitively this is because we can not determine the distance along the line of sight from p to P in Figure 7.1.

However, there are some techniques we can use to compute depth estimates from 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 we have:

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

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. Since f and z are known, we can use this relationship to compute the depth, Z. If we have two cameras, we can estimate depth by using *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, similar to the approach we used for camera auto-calibration in Equation (7.42).

7.5 Exercises

The starter code for the exercises provided below is available online through GitHub. To get started, download the code by running in a terminal window:

```
git clone https://github.com/StanfordASL/pora-exercises.git
```

We denote Problems requiring hand-written solutions and coding in Python with ✎ and 🖳, respectively.

🖳 *Problem 1: Camera Calibration: Extrinsics*

In the file `ch07/exercises/camera_extrinsics.ipynb`, you will implement the key parts of the method to compute the camera extrinsic parameters R and t. Specifically, you will use the steps in Section 7.2.1 to compute the homography matrix M given a calibration image. Then, given the camera intrinsic matrix K, you will use the method at the end of Section 7.2.3 to compute the extrinisics, R and t, for the image.

🖳 *Problem 2: Camera Calibration: Intrinsics*

In Problem 1, we provided the camera intrinsic matrix, K. In this exercise, you will use the flexible calibration method described in Section 7.2.3 to compute the intrinsic matrix yourself. Complete the exercise located at `ch07/exercises/camera_intrinsics.ipynb`.

🖳 *Problem 3: Manipulating Point Clouds From The KITTI Dataset*

In this exercise we will explore the KITTI dataset, and look at point cloud and RGB image data, and look at how these two sources of data can be aligned. The code for this exercise can be found at `ch07/exercises/lidar_rgb_alignment.ipynb`.

References

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

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

[41] David G Lowe. "Object recognition from local scale-invariant features". In: *Proceedings of the seventh IEEE international conference on computer vision*. Vol. 2. Ieee. 1999, pp. 1150–1157.

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

[73] Bill Triggs et al. "Bundle adjustment—a modern synthesis". In: *Vision Algorithms: Theory and Practice: International Workshop on Vision Algorithms Corfu, Greece, September 21–22, 1999 Proceedings*. Springer. 2000, pp. 298–372.

[74] 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.

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

8
Stereo Vision and Structure From Motion

In Chapter 7, we introduced the 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 a camera. This relationship is based on the pinhole camera model, and requires knowledge about the camera's intrinsic and extrinsic parameters. We also presented methods for camera calibration, which allows us to determine the intrinsic and extrinsic camera parameters.

Given a calibrated camera with known parameters, another fundamental problem in robotic perception is how to leverage images to recover three-dimensional information about the structure of the environment [1]. The camera projection model alone does not provide us with enough information to fully determine the 3D position of a point from a single image, specifically because we cannot determine the point's *depth*[2]. In this chapter, we introduce *stereo vision* and *structure from motion*[3,4], two approaches for extracting 3D information about a scene from camera images.

Stereo vision and structure from motion both leverage multiple images of a scene in order to determine three-dimensional structure. Stereo vision leverages images from different viewpoints, typically from two cameras, and structure from motion typically considers images captured from a single camera that moves through the scene.

[1] While we could also use a number of other sensors to recover three-dimensional scene information, such as ultrasonic sensors or laser rangefinders, cameras capture a broad range of information that goes beyond depth sensing and are also attractive based on their cost and size.

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

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

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

8.1 Stereo Vision

Stereopsis[5] 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 our eyes' different positions in the head. This disparity makes our brain fuse the two retinal images into a single image, allowing us to perceive objects as single solid entities. 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 from the disparity between your

[5] From *stereo*, meaning solidity, and *opsis*, meaning vision or sight.

eyes.

Computational stereopsis, or *stereo vision*, is the process of obtaining depth information of a 3D scene given 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, identifying correlations between each point in the 3D environment and their corresponding pixels in each camera. Reconstruction is the problem of *triangulation*, which uses the pixel correspondences to determine the full position of the source point in the scene, including depth.

8.1.1 Epipolar Constraints

The first step in the stereo vision process is to fuse the two or more images and generate point correspondences[6]. This task can be quite challenging, and erroneously matching features can lead to large errors in the reconstruction step. We leverage several techniques to simplify this task, including the use of *epipolar constraints*.

[6] We generally assume that the perspective of each image is only a slight variation from the other, such that the features appear similarly in image.

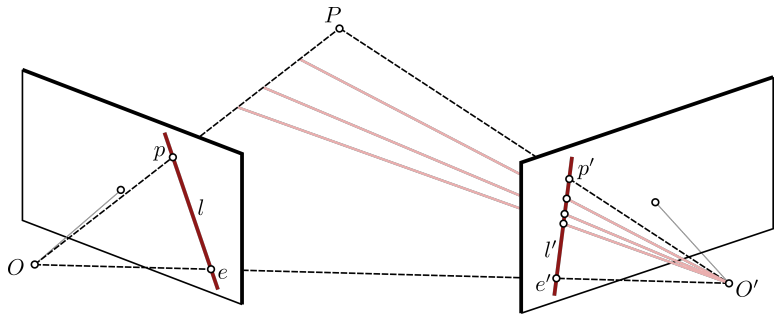


Figure 8.1: The point, P, in the scene, the optical centers, O and O', of the two cameras, and the two images, p and p', of P all lie in the same plane, which we refer to as the epipolar plane. The lines l and l' are the epipolar lines of the points p and p', respectively. Note that if the point p is observed in one image, the corresponding point in the second image must lie on the epipolar line l'!

Consider the image pixel coordinates, p and p', of a point, P, observed by two cameras with optical centers, O and O', as we show in Figure 8.1. 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. We refer to the line l as the *epipolar line* associated with the point p, and it passes through the point e, which we refer 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. We refer to this as an *epipolar constraint*, and it greatly simplifies the correspondence problem by restricting the possible candidate points to a line rather than the entire image[7]. Mathematically, we write the epipolar constraint as:

$$\overline{Op} \cdot [\overline{OO'} \times \overline{O'p'}] = 0, \tag{8.1}$$

since $\overline{Op}$, $\overline{O'p'}$, and $\overline{OO'}$ are coplanar. Assuming the world reference frame is

[7] In other words, the correspondence problem becomes a one-dimensional problem rather than a two-dimensional one.

collocated with the camera with an origin at point O, we can write this constraint as:

$$p^\top F p' = 0, \tag{8.2}$$

where we refer to $F \in \mathbb{R}^{3\times 3}$ as the *fundamental matrix*. The fundamental matrix, which we already introduced in Chapter 7 in the context of camera calibration, has seven degrees of freedom and is singular. Additionally, the fundamental matrix is only dependent on the intrinsic camera parameters for each camera and the geometry that defines their relative positioning, and we generally assume it to be constant. The expression for the fundamental matrix in terms of the camera intrinsic parameters is:

$$F = K^{-\top} E K'^{-1}, \quad E = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix} R, \tag{8.3}$$

where K and K' are the intrinsic parameter matrices for the two stereo cameras, and R and $t = [t_1, t_2, t_3]^\top$ are the rotation matrix and translation vector that map the second camera frame coordinates into the first camera frame coordinates. The matrix E is the *essential matrix*, which we also introduced in Chapter 7.

Note that with the epipolar constraint defined by the fundamental matrix in Equation (8.2), we can express the epipolar lines l and l' by $l = Fp'$ and $l' = F^\top p$. Additionally, we can show that $F^\top e = Fe' = 0$, where e and e' are the epipoles in the image frames of cameras, since by definition the translation vector, t, is parallel to the coordinate vectors of the epipoles in the camera frames. This fact guarantees that the fundamental matrix, F, is singular.

If the parameters K, K', R, and t are not already known, we can compute the fundamental matrix, F, in a manner similar to how we computed the intrinsic parameter matrix, K, in the previous chapter. Suppose we have a number of corresponding points $p = [u, v, 1]^\top$ and $p' = [u', v', 1]^\top$ that are expressed as homogeneous coordinates. Each pair of points has to satisfy the epipolar constraint in Equation (8.2), which we can write as:

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

We can equivalently express this constraint by reparameterizing the matrix F in vector form, which we denote as f, as:

$$\begin{bmatrix} uu' & uv' & u & vu' & vv' & v & u' & v' & 1 \end{bmatrix} f = 0 \tag{8.4}$$

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'), we can stack these constraints into a matrix, W, such that:

$$Wf = 0, \tag{8.5}$$

where $W \in \mathbb{R}^{n\times 9}$.

Given $n \geq 8$ correspondences, we compute an estimate, $\tilde{F}$, of the fundamental matrix by solving the optimization problem:

$$\begin{aligned} &\min_{f} \ \|Wf\|^2, \\ &\text{s.t. } \|f\|^2 = 1, \end{aligned} \tag{8.6}$$

which is a constrained least squares problem, which we discussed in Example 7.2.1. However, note that the estimate, $\tilde{F}$, computed using Equation (8.6) is not guaranteed to be singular. We can compute a singular fundamental matrix from this estimate through a second optimization step:

$$\begin{aligned} &\min_{F} \ \|F - \tilde{F}\|^2, \\ &\text{s.t. } \det(F) = 0, \end{aligned} \tag{8.7}$$

which in practice we solve by computing a singular value decomposition of the matrix $\tilde{F}$.

8.1.2 Image Rectification

Given a pair of stereo images, epipolar rectification[8,9] is a transformation of each image plane such that all corresponding epipolar lines become collinear and parallel to one of the image axes[10]. We can think of the resulting rectified images as acquired by a new stereo camera obtained by rotating the original cameras about their optical centers. The advantage of epipolar rectification is that the correspondence search becomes simpler and computationally less expensive because the search is done only along the horizontal lines of the rectified images. We illustrate the steps of the epipolar rectification algorithm in Figure 8.2. Observe that after the rectification, all of the epipolar lines in the left and right image are collinear and horizontal.

[8] 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

[9] 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

[10] For convenience, usually the horizontal axis.

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

8.1.4 Reconstruction Problem

In a stereo vision setup, once a correspondence between the two images is identified, we can reconstruct the 3D scene point based on the process of *triangulation*. Triangulation has already been covered by our discussion on epipolar

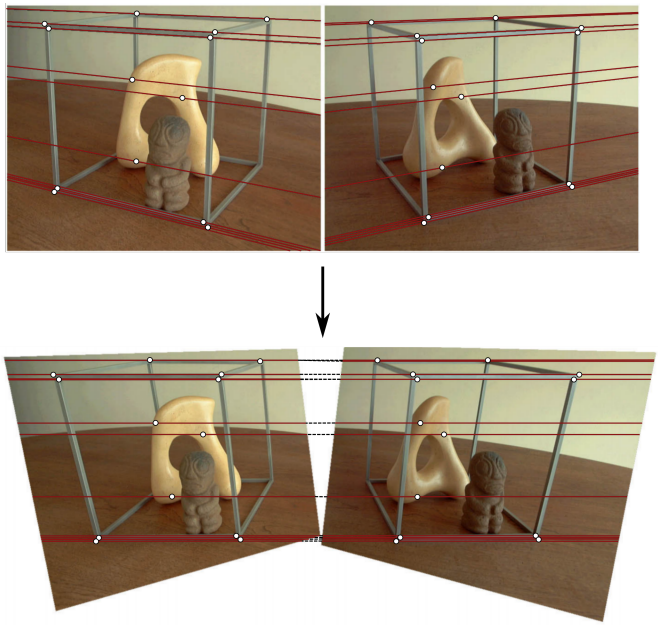


Figure 8.2: Epipolar rectification example from Loop et al. (1999).

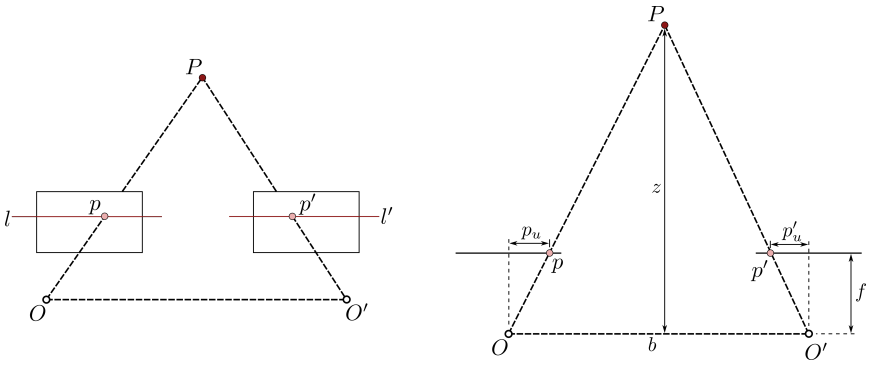


Figure 8.3: Triangulation with rectified images (horizontal view on the left, top-down view on the right).

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*, as we show in Figure 8.3.

In Figure 8.3, a point, P, in the scene is located at coordinate (x, y, z) with respect to the origin located in the left camera at point O. We denote the horizontal pixel coordinate in the left and right image by p_u and p'_u, respectively. Based on the geometry, we can compute the depth of point P from the properties of similar triangles:

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

which we can algebraically simplify to:

$$z = \frac{bf}{p_u - p'_u}, \tag{8.8}$$

where f is the focal length. Generally a small baseline, b, will lead to larger depth errors, but a large baseline may cause features to be visible from one camera and not the other. We refer to the difference in the image coordinates, $p_u - p'_u$, as *disparity*. This is an important term in stereo vision because it is only by measuring disparity that we can recover depth information. The disparity can also be visually represented in a *disparity map*, which is simply a map of the disparity values for each pixel in an image. We show an example of a disparity map in Figure 8.4[11]. Note that the largest disparities occur from nearby objects, since disparity is inversely proportional to the depth, z.

[11] 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

Figure 8.4: 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).

8.2 Structure From Motion (SFM)

The *structure from motion (SFM)* method uses a similar principle as stereo vision, but uses a single 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 across images, but the extrinsic parameters consisting of the rotation matrix, R, and relative position vector, t, will be different for each image. Consider a case where we take m images of n fixed 3D points from

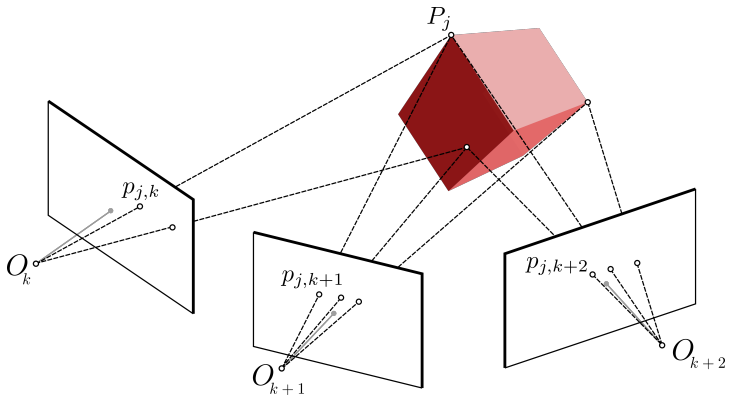


Figure 8.5: A depiction of the structure from motion (SFM) method. A single camera is used to take multiple images from different perspectives, which provides enough information to reconstruct the 3D scene.

different perspectives. This would lead to m homography matrices, M_k, and

n 3D points, P_j, that we would need to determine by leveraging the projection relationships:

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

Notice that there is quite a bit of similarity between this problem and the camera auto-calibration problem discussed in Chapter 7, except here we assume we already know the camera intrinsic parameters.

Structure from motion methods also have some unique disadvantages, such as an ambiguity in the absolute scale of the scene that cannot be determined. For example, a bigger object at a longer distance and a smaller object at a closer distance can yield the same projections. One application of the structure from motion concept is known as *visual odometry*. Visual odometry estimates the motion of a robot in part by using visual inputs. This approach is commonly used in practice, for example by rovers on Mars, and is useful because it not only allows us to recover the motion of the robot but we can also generate a 3D scene reconstruction that we can use for planning.

References

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

[17] 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.

[40] 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.

[57] 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.

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

9
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 images. In this chapter, we introduce methods for extracting other types of information, such as semantic information, through *image processing*. For example, we might want to identify what type of objects are in the scene in addition to understanding where they are located. Extracting this type of visual content from raw images is important for mobile robots to be able to intelligently interpret their surroundings[1].

At its core, 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, in this chapter, we focus on some of the more fundamental methods that are relevant for robotics[2,3]. In particular, the methods we focus on are related to image filtering, feature detection, and feature description.

[1] Information extracted through image processing can have a significant impact on a robot's ability perform fundamental tasks including localization, mapping, and decision making.

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

[3] H. P. Moravec. "Towards automatic visual obstacle avoidance". In: *5th International Joint Conference on Artificial Intelligence*. 1977

9.1 Image Filtering

Image filtering is one of the principal tasks in image processing. The term *filter* comes from frequency domain signal processing and refers to the process of accepting or rejecting certain frequency components of a signal[4]. 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 on the pixels in the immediate spatial neighborhood, as we show in Figure 9.1.

[4] For example, eliminating high-frequency noise is a classic filtering problem.

Mathematically, we describe an image as a function, $I(x, y)$, that maps a pixel at coordinate (x, y) in the domain $[a, b] \times [c, d]$ to either a scalar for grayscale images or a three-dimensional vector corresponding to red, green, and blue values for color images. A spatial filter for an image, $I(x, y)$, consists of a neighborhood of pixels around a particular point, (x, y), under examination, which we denote as S_{xy}[5], and a predefined operation, F, that is performed on the image pixels encompassed by the neighborhood S_{xy}. We define a new image, $I'(x, y)$, by ap-

[5] This region is typically rectangular.

plying the spatial filter operation F to all pixels, (x, y), in the original image, I.

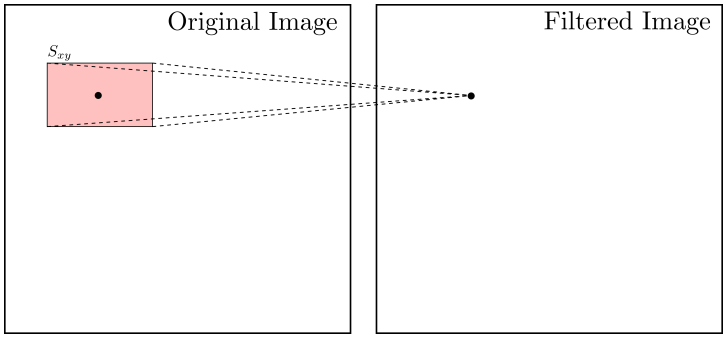


Figure 9.1: Illustration of the concept of spatial filtering. The spatial filter operates on a neighborhood, S_{xy}, of each point in the original image to produce a new pixel in the filtered image.

In general, filters can leverage linear or nonlinear operations, but many of the most fundamental filters are linear and we can express them mathematically as:

$$I'(x,y) = F \circ I = \sum_{i=-N}^{N} \sum_{j=-M}^{M} F(i,j) I(x+i, y+j), \tag{9.1}$$

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, we say that this filter is of size $(2N+1) \times (2M+1)$. We generally refer to the filter operation F as a *mask* or *kernel*. Broadly speaking, we refer to filters expressed by Equation (9.1) as *correlation filters*.

Convolution filters are another class of linear filters that we commonly use. Convolution filters are similar to correlation filters, but use reverse image indices[6]. In particular, we express convolution filters mathematically by:

$$I'(x,y) = F * I = \sum_{i=-N}^{N} \sum_{j=-M}^{M} F(i,j) I(x-i, y-j). \tag{9.2}$$

[6] In fact, correlation and convolution filters are identical when the filter mask is symmetric in both the horizontal and vertical directions.

Convolution filters are associative, meaning that for two different filter masks, F and G, it is true that $F * (G * I) = (F * G) * I$. This associative property is useful for tasks such as smoothing an image before applying a differentiation filter. Suppose the mask F implements a derivative filter and G implements a smoothing filter, then sequentially applying these filters would result in $F * (G * I)$. However, because of the associative property, we can convolve the masks together first such that only the single filter $(F * G) * I$ needs to be applied to the image.

Note that in both correlation and convolution filters, the boundaries of the image need some special care because of the width and height of the mask. For example, in Figure 9.2 we show 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 relative to the mask size, the exact approach likely won't vary the final result significantly.

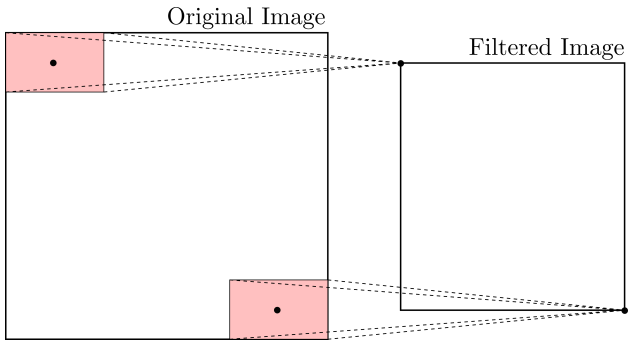


Figure 9.2: Due to the width and height of the mask, the filtered image may be smaller than the original. This can be fixed with several techniques, such as padding.

Example 9.1.1 (Practical Tricks for Image Filtering)**.** When implementing correlation and convolution filters, we can leverage special tricks to simplify the process. In this example, we introduce two simplification tricks: a change in indexing and zero-padding.

First, to accommodate varying sizes of filters, including even and odd sized filters, we can change the indexing such that the coordinate of interest is associated with the top-left element in the window rather than the center. For a correlation filter, this would correspond to:

$$I'(x,y) = F \circ I = \sum_{i=1}^{K} \sum_{j=1}^{L} F(x,y) I(x+i-1, y+j-1), \tag{9.3}$$

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. Note that this formulation results in an output image, I', that is shifted up and to the left. To see this shift, consider the top-left pixel at $x = 1$ and $y = 1$ in the new image, I'. We generate this new pixel value by applying the filter, F, over the pixels in the original image at rows $\{1, \ldots, K\}$ and columns $\{1, \ldots, L\}$, which is not centered at $(1,1)$ in the original image, I. In practice, this shifting is not an issue as long as we always index with respect to the top-left corner. We show an example of top-left indexing in Figure 9.3.

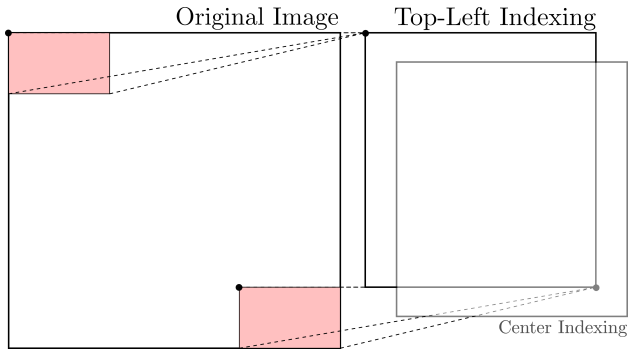


Figure 9.3: Top-left indexing is typically easier to implement than center indexing. Notice that when top-left indexing, it appears as if the filtered image has shifted with respect to when we use center indexing.

Zero-padding[7] is another simple trick that we can use to ensure that the output filtered image, I', has the same dimension as the input image, I. In this

[7] Also commonly referred to as *same padding*.

approach, we pad the left and right boundaries of the image by $\lfloor K/2 \rfloor$ columns of zeros, and pad the top and bottom boundaries by $\lfloor L/2 \rfloor$ rows of zeros, where $\lfloor \cdot \rfloor$ denotes the *floor* operation. For example, the image:

$$I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix},$$

becomes:

$$I_{\text{padded}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ 0 & 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

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 a correlation filter from Equation (9.3) and a filter, F, with $K = 2, 3$ and $L = 2, 3$, we can define the new image, I', 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. We show an example use of padding combined with top-left indexing graphically in Figure 9.4.

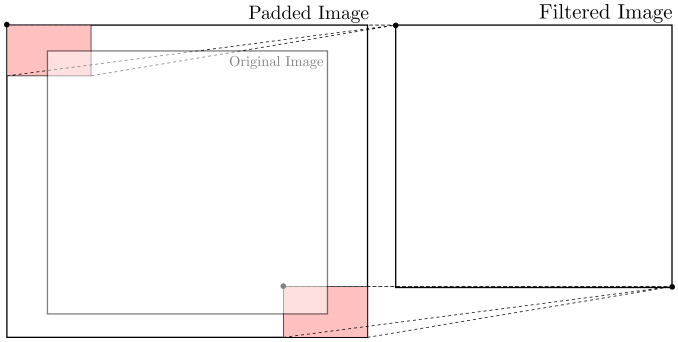


Figure 9.4: Image padding is a commonly used technique to ensure that the size of the filtered image is the same size as the original.

9.1.1 *Moving Average Filter*

The moving average filter returns the average of the pixels in the mask, which achieves a smoothing effect[8]. For example, we can define a moving average filter with a normalized[9] 3×3 mask with F from Equation (9.1) defined as:

$$F = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Due to the symmetry of the mask, the correlation filter from Equation (9.1) and convolution filter from Equation (9.2) will be identical.

[8] Smoothing removes sharp features in the image.

[9] The normalization is used to maintain the overall brightness of the image.

9.1.2 *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:

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(- \frac{x^2 + y^2}{2\sigma^2} \right).$$

We use this function to obtain the mask operation, F, by sampling the function about the center pixel. For example, for the center pixel with $i = j = 0$ in Equation (9.1), we sample $G_\sigma(0, 0)$. 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 or convolution filters. The advantage of the Gaussian filter is that it provides more weight to the neighboring pixels that are closer. We show an example of this filter in Figure 9.5.

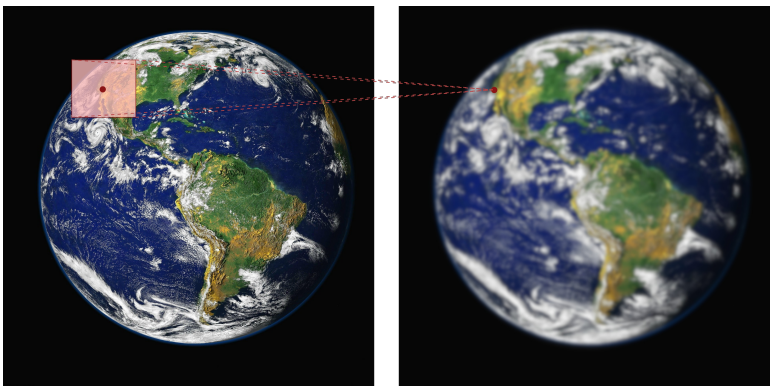


Figure 9.5: Example of a Gaussian smoothing filter, which produces a smoothing (blurring) effect on the filtered image.

9.1.3 *Separable Masks*

We call a mask *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 we can represent the mask as an outer product of two vectors, meaning it is equivalent to the 2D convolution of those two vectors. If a separable mask has shape $M \times M$ and the input image has size $w \times h$, then the computational complexity of directly performing the convolution is $O(M^2wh)$. By separating the masks, the computational cost is $O(2Mwh)$, which is 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, we note that the Gaussian smoothing filter mask is also separable. To see why this is, note that we can decompose the Gaussian weighting function as:

$$\begin{aligned} G_\sigma(x,y) &= \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right),\\ &= \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{x^2}{2\sigma^2}\right)\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{y^2}{2\sigma^2}\right),\\ &= g_\sigma(x)\cdot g_\sigma(y). \end{aligned}$$

9.1.4 *Image Differentiation Filters*

We can identify some image features, such as edges, by looking at the spatial derivatives in the pixel intensity values in both the vertical and horizontal directions. Since we represent images as functions defined over a discrete domain, the traditional method for differentiating continuous functions is not applicable. Instead, we can compute differences between pixels using techniques like the central difference method:

$$\begin{aligned} \frac{\partial I}{\partial x} &= \frac{I(x+1,y)-I(x-1,y)}{2},\\ \frac{\partial I}{\partial y} &= \frac{I(x,y+1)-I(x,y-1)}{2}. \end{aligned} \tag{9.4}$$

where $\partial I/\partial x$ is the derivative in the horizontal direction and $\partial I/\partial y$ is the derivative in the vertical direction. We can also define the derivatives using just one side instead of a central difference, for example $\frac{\partial I}{\partial x} = I(x+1,y) - I(x,y)$.

We can also differentiate an image using convolution filters. In particular, one common approach is to use a convolution filter of the form Equation (9.2) defined with a mask, F, called a *Sobel mask*[10]. We denote this mask as S_x for the x direction and S_y for the y direction:

[10] Also referred to as simply a *Sobel operator*.

$$S_x = \begin{bmatrix}1 & 0 & -1\\ 2 & 0 & -2\\ 1 & 0 & -1\end{bmatrix}, \quad S_y = \begin{bmatrix}1 & 2 & 1\\ 0 & 0 & 0\\ -1 & -2 & -1\end{bmatrix} \tag{9.5}$$

Sobel masks are similar to the central difference method but use more neighboring pixels when calculating the derivative[11]. Note that Sobel masks are *separable*.

[11] Specifically, they also consider the rows above and below to compute the difference.

9.1.5 *Similarity Measures*

We can also use filtering 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, we can compute the similarity between

the pixel (x, y) in image I_1 and pixel (x', y') in image I_2 by:

$$\begin{aligned} SAD &= \sum_{i=-N}^{N} \sum_{j=-M}^{M} |I_1(x+i, y+j) - I_2(x'+i, y'+j)|, \\ SSD &= \sum_{i=-N}^{N} \sum_{j=-M}^{M} [I_1(x+i, y+j) - I_2(x'+i, y'+j)]^2, \end{aligned} \tag{9.6}$$

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 we consider.

9.2 Image Feature Detection

A local feature[12] in an image is a pattern that differs from its immediate neighborhood in terms of intensity, color, or texture. We can generally categorize local features in several ways, for example by whether or not they provide semantic content. For example, features that may provide semantic content include edges or other geometric shapes, such as lanes of a road or blobs corresponding to blood cells in medical images. 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, we could recognize a scene 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.

[12] Also sometimes referred to as interest points, interest regions, or keypoints.

In this section, we discuss several feature detection strategies. While many strategies exist for different types of features, our focus will be on two common features that are often useful in robotics: edges and corners.

9.2.1 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. We can evaluate an edge detector based on several criteria for robustness and performance, including accuracy, localization, and single response. Good accuracy implies few false positives or negatives[13], good lo-

[13] In this case, a false positive is a detection of an edge that isn't real, and a false negative is a missed edge.

calization implies that the detected edge should be exactly where the true edge is in the image, and a single response implies that only one edge is detected for each real edge. Noise and discretization effects can make edge detection challenging in practice.

Most edge detection methods rely on two key steps: smoothing and differentiation. We perform differentiation 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.

Example 9.2.1 (Edge Detection in 1D)**.** In Figure 9.6, we show an example of how noise can corrupt image differentiation. Notice that in this case it is impos-

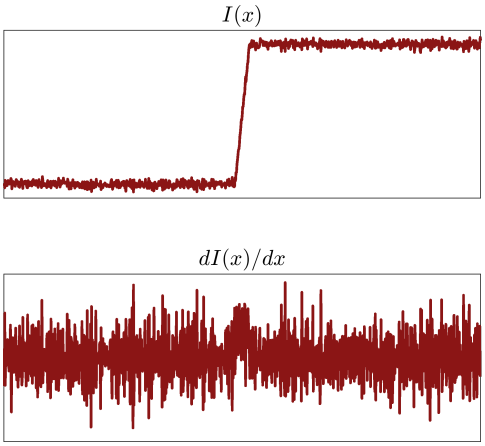


Figure 9.6: Differentiation of a signal with noise can be particularly challenging. We can address this by first smoothing the signal.

sible 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 9.6 is defined by $I(x)$. We can compute a smoothed version 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:

$$s'(x) = \frac{d}{dx} * s(x).$$

We show this process in Figure 9.7. Note that since these filters are convolutions, we can leverage the associativity property to combine them into a single filter:

$$s' = (\frac{d}{dx} * g_\sigma) * I.$$

Example 9.2.2 (Edge Detection in 2D)**.** Edge detection in a two-dimensional image is quite similar to the example previously discussed for one-dimension. Let the smoothing filter be the Gaussian smoothing filter from before, and consider

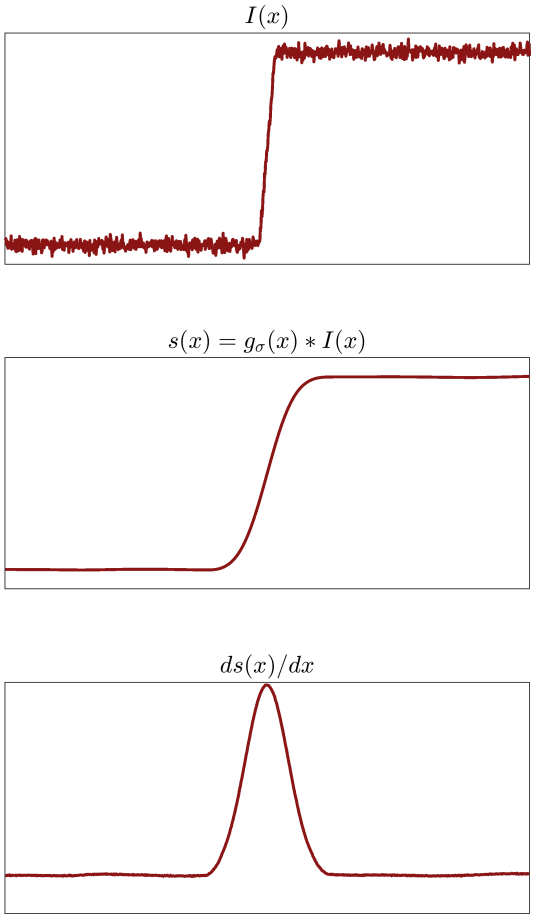


Figure 9.7: Edge detection through convolution with a Gaussian smoothing filter, followed by a differentiation filter.

a differentiation filter such as the Sobel filter. We can write the gradient of the smoothed image in both the x and y directions 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 we use the associativity property of the smoothing and differentiation convolution filters to define the combined filters $G_{\sigma,x}$ and $G_{\sigma,y}$. We can then compute the magnitude of the gradient by:

$$|\nabla S| = \sqrt{S_x^2 + S_y^2},$$

which we can use to compare against a predefined threshold value for edge detection. To guarantee that we define thin edges, it is also possible to filter out points whose gradient magnitude are above the threshold but are not local maxima. We show an example of this process in Figure 9.8.

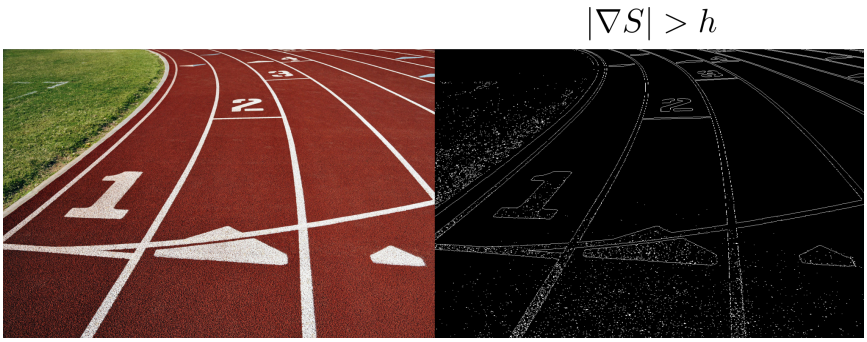


Figure 9.8: Edge detection using the Sobel edge detector.

9.2.2 *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. The repeatability of a corner detector quantifies how well we can find the same features 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 we can use to reliably produce correspondences. Both of these properties are particularly important in applications such as panorama stitching and 3D reconstruction.

We can generally think of corner detection 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[14], which has the useful property that the detection is invariant to rotations and linear intensity changes, such as 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.

[14] C. Harris and M. Stephens. "A combined corner and edge detector". In: *4th Alvey Vision Conference*. 1988

9.3 *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 desirable for image descriptors to be repeatable[15] and distinct. Perhaps the simplest example of a descriptor is an $n \times m$ window of pixel intensities centered at the feature, which we can normalize 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.

[15] For example, invariant with respect to pose, scale, and illumination.

9.4 *Exercises*

The starter code for the exercises provided below is available online through GitHub. To get started, download the code by running in a terminal window:

```
git clone https://github.com/StanfordASL/pora-exercises.git
```

We denote Problems requiring hand-written solutions and coding in Python with ✎ and 🖳, respectively.

🖳 *Problem 1: Correlation and Gaussian Smoothing Filters*

In the file `ch09/exercises/correlation_filter.ipynb`, you will implement the correlation filter and Gaussian smoothing filter from Section 9.1.2.

References

[20] C. Harris and M. Stephens. "A combined corner and edge detector". In: *4th Alvey Vision Conference*. 1988.

[47] H. P. Moravec. "Towards automatic visual obstacle avoidance". In: *5th International Joint Conference on Artificial Intelligence*. 1977.

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

10
Information Extraction

Chapter 9 introduced some fundamental topics related to image processing, including filtering, feature detection, and feature description. While these techniques are useful for a large number of computer vision applications, they are not sufficient to extract higher-level information. For example, we discussed methods for identifying local features of the image, such as corners and edges, but these may be too localized to understand higher-level features or semantic content.

In this chapter, we introduce methods for higher-level feature extraction, including geometric feature extraction and image object recognition[1]. Geometric feature extraction is useful to identify structure, including geometric primitives such as lines, circles, and planes, which is useful in robotics applications for localization and mapping[2]. Object recognition from images is also important in robotics applications to ensure robots operating in real environments can complete their tasks safely and effectively.

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

[2] Geometric feature extraction methods are generally applicable to different types of sensor data, including images or data collected from laser rangefinders or radar.

10.1 Geometric Feature Extraction

It is common in robotic localization and mapping to represent the environment using simple geometric primitives[3] that we can efficiently extract from sensor data. In this section, we present some techniques for line extraction from range data[4]. Lines are one of the most fundamental geometric primitives that we would want to extract from data, and techniques for extracting other primitives are conceptually similar.

There are two main challenges with extracting lines from range data. The first is *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, in this chapter, we consider line extraction problems based on two-dimensional range data.

[3] Common geometric primitives include lines, circles, corners, and planes.

[4] Range data can generally come from a variety of sources, including laser rangefinders, radar, or even computer vision.

10.1.1 Line Segmentation

The line segmentation problem is to determine how many lines exist in a given set of data and which data points correspond to each line. We will discuss three popular algorithms for line segmentation: 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 a popular line extraction algorithm that is fast but not very robust to outliers. The split-and-merge algorithm repeatedly fits lines to sets of points and then splits the set of points into two sets if any point lies more than a specified distance, d, from the line. By repeating this process until no more splits occur, we are guaranteed that all points will lie less than the distance, d, to a line. After this splitting process is complete, a second step merges any of the newly formed lines that are collinear. We present this algorithm in more detail in Algorithm 10.1. A popular variant

Algorithm 10.1: 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 ← [S]
i ← 1
while i ≤ length(L) do
    Fit a line (α, r) to the set L[i]
    Detect the point P ∈ L[i] with maximum distance, D, to the line (α, r)
    if D < d then
        i ← i + 1
    else
        Split L[i] at P into new sets, S1 and S2
        L[i] ← S1
        L[i + 1] ← S2
Merge collinear sets in L
```

of the split-and-merge algorithm is known as the iterative-end-point-fit algorithm. This algorithm is the split-and-merge algorithm in Algorithm 10.1 where the line is constructed by simply connecting the first and the last points of the set. We show this approach graphically in Figure 10.1.

Random Sample Consensus (RANSAC): Random Sample Consensus (RANSAC)[5] is an algorithm to estimate the parameters of a model from a set of data that may contain outliers[6]. 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 laser scan of an indoor environment may contain distinct lines from the surrounding walls but also points from other static and dynamic objects

[5] Martin A. Fischler and Robert C. Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". In: *Commun. ACM* 24.6 (1981), pp. 381–395

[6] This problem is sometimes referred to as *robust* model parameter estimation.

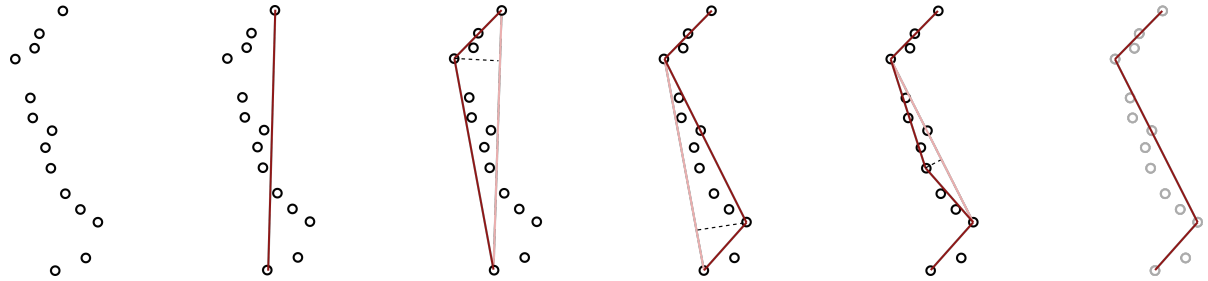


Figure 10.1: Iterative-end-point-fit variation of the split-and-merge algorithm for extracting lines from data.

such as chairs or humans. In this case, if the goal is to extract lines to represent the walls, then any data point corresponding to other objects would be an outlier. In general, we can apply RANSAC 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[7]. In this section, we focus on using RANSAC for line extraction from two-dimensional data.

RANSAC is an iterative method and is non-deterministic[8]. Given a dataset, S, of N points, we start by randomly selecting a sample of two points from S. Next, we construct a line from the two sampled points and compute the distance of all other points to this line. We then define the set of *inliers*, which is comprised of all points whose distance to the line is within a predefined threshold, d. By repeating this process k times, we generate k inlier sets and their associated lines and return the inlier set with the most points. We detail this procedure in Algorithm 10.2 and illustrate the process in Figure 10.2.

[7] Where the goal in structure-from-motion problems is to identify image correspondences which satisfy a rigid body transformation.

[8] In other words, it is stochastic or random. Running the algorithm twice on the same inputs will not necessarily produce the same results.

Algorithm 10.2: 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 two points from S.
  Fit line, l_i, through the two 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.

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, we can perform a simple statistical analysis of RANSAC. Let p be the *desired* probability of finding a set of points free of outliers and let w be the

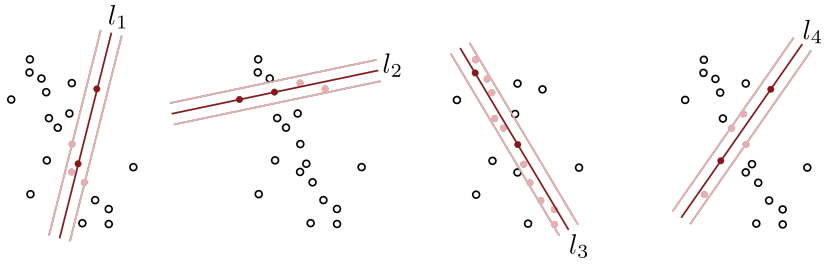


Figure 10.2: Example of the RANSAC algorithm, showing four iterations of the algorithm. If the algorithm was terminated after these four iterations, line l_3 would be returned since it contains the maximum number of points.

probability of selecting an inlier from the dataset, S, of N points, which we can express as:

$$w = \frac{\text{\# inliers}}{N}.$$

Assuming we draw point samples 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 $(1 - w^2)^k$. We can therefore find the minimum number of iterations, $\bar{k}$, needed to find an outlier-free set with probability p by solving:

$$1 - p = (1 - w^2)^k,$$

for k. In other words, we can compute $\bar{k}$ as:

$$\bar{k} = \frac{\log(1 - p)}{\log(1 - w^2)}.$$

While the value of w may not be known exactly[9], we can still use this expression to get a good estimate of the number of iterations, k, that we need for good results. It is important to note that this probabilistic approach often leads to a much smaller number of iterations than a brute force search through all combinations. We can attribute this to the fact that $\bar{k}$ is only a function of w and not the total number of samples, N, in the dataset.

[9] There are advanced versions of RANSAC that can estimate w in an adaptive online fashion.

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 to data outliers. The main disadvantages are that the algorithm needs to run multiple times to extract multiple features, 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 dataset, S, votes for a *set* of possible line parameters, (m, b), where m is the slope and b is the intercept point. 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 we can also write as:

$$b = -mx_i + y_i.$$

Therefore, each point, (x, y), in the original space space maps to a *line*, (m, b), in the Hough space, as we show in Figure 10.3). 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, as we show in Figure 10.4.

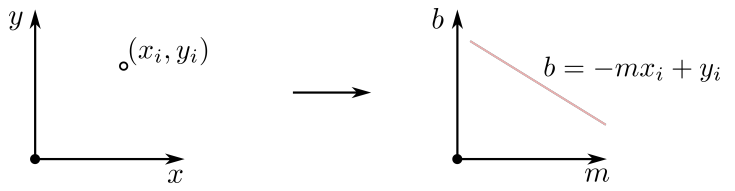


Figure 10.3: Each point, (x_i, y_i), in the original space maps to a *line* in the Hough space which describes all possible parameters m and b that would generate a line passing through the point (x_i, y_i).

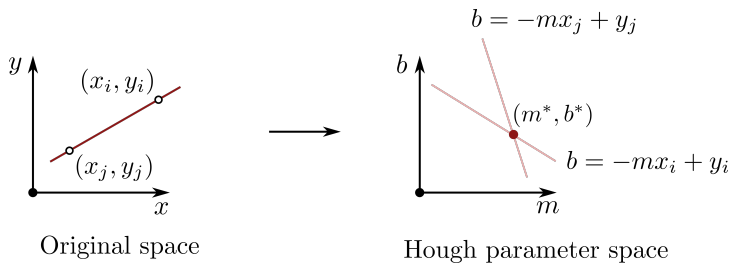


Figure 10.4: All points on a line in the original space yield lines in the Hough space that intersect at a common point.

We can apply this concept to the line segmentation problem by searching in the Hough space for intersections among the lines that correspond to each point, (x, y), in the set, S. In practice, we do this 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. We choose local maxima among the cells 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 the slope, m, can range from $-\infty$ to ∞. Alternatively, we can 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, we map a point, (x_i, y_i), from the original space to the polar Hough space, (α, r), as a sinusoidal curve, as we show in Figure 10.5). We provide an example of the Hough transform using the polar representation in Figure 10.6.

10.1.2 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

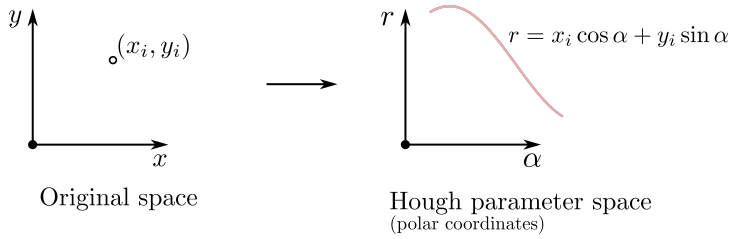


Figure 10.5: Representation of a point, (x_i, y_i), in the Hough space when using a polar coordinate representation of a line with parameters α and r.

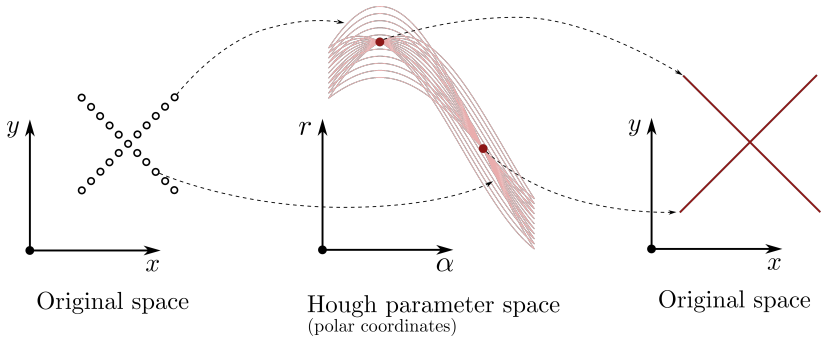


Figure 10.6: Example of the Hough transformation using a polar coordinate representation of lines.

corresponding data points. For the split-and-merge, RANSAC, and Hough-transform line segmentation algorithms we previously discussed, we also implicitly defined a line associated with the segmented data points. However, these implicitly defined lines 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, meaning that there are more data points than parameters to choose, and noise in the data means that there is not a 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:

$$\rho \cos(\theta - \alpha) = r, \quad \text{or} \quad x \cos\alpha + y \sin\alpha = r, \tag{10.1}$$

where α and r are the parameters that define the line. We provide a visual representation of these definitions in Figure 10.7.

For a collection, S, of N points, which we denote by (ρ_i, θ_i), we compute the error corresponding to the perpendicular distance from a point to a line defined by parameters α and r by:

$$d_i = \rho_i \cos(\theta_i - \alpha) - r, \tag{10.2}$$

where d_i is the error. We can therefore formulate the line fitting task as an optimization problem over the parameters α and r to minimize the combined errors

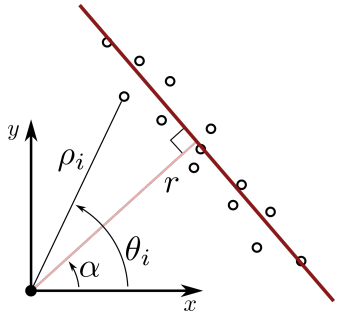


Figure 10.7: Representation of a line in polar coordinates, defined by the parameters r and α, which are the distance and angle to the closest point on the line to the origin.

d_i for $i = 1, \ldots, N$. In particular, we aggregate the combined errors using a sum of the squared errors:

$$S(r, \alpha) = \sum_{i=1}^{N} d_i^2 = \sum_{i=1}^{N} (\rho_i \cos(\theta_i - \alpha) - r)^2. \tag{10.3}$$

This is a classic least squares optimization problem that we can efficiently solve. However, this cost function generally assumes that each of the data points is equally affected by noise[10]. 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.

[10] In other words, the uncertainty of each measurement is the same

Accounting for unique uncertainties in each data point leads to a *weighted least squares estimation* problem. In particular, we assume that the variance of each range measurement, ρ_i, is given by σ_i. We then modify the cost function from Equation (10.3) to be:

$$S_w(r, \alpha) = \sum_{i=1}^{N} w_i d_i^2 = \sum_{i=1}^{N} w_i (\rho_i \cos(\theta_i - \alpha) - r)^2, \tag{10.4}$$

where the weights, w_i, are given by:

$$w_i = \frac{1}{\sigma_i^2}.$$

We can show that the solution to the optimization problem defined by the weighted cost function in Equation (10.4) is:

$$\begin{aligned} r &= \frac{\sum_{i=1}^{N} w_i \rho_i \cos(\theta_i - \alpha)}{\sum_{i=1}^{N} w_i}, \\ \alpha &= \frac{1}{2} \text{atan2} \left(\frac{\sum_{i=1}^{N} w_i \rho_i^2 \sin(2\theta_i) - \frac{2}{\sum_{i=1}^{N} w_i} \sum_{i=1}^{N} \sum_{j=1}^{N} w_i w_j \rho_i \rho_j \cos\theta_i \sin\theta_j}{\sum_{i=1}^{N} w_i \rho_i^2 \cos(2\theta_i) - \frac{1}{\sum_{i=1}^{N} w_i} \sum_{i=1}^{N} \sum_{j=1}^{N} w_i w_j \rho_i \rho_j \cos(\theta_i + \theta_j)} \right) + \frac{\pi}{2} \end{aligned} \tag{10.5}$$

10.2 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 is 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, for example breeds of dogs or car models. In this section, we introduce three common methods for object recognition, namely *template matching*, *bag of visual words*, and *neural network methods*.

10.2.1 Template Matching

Template matching[11] is a machine vision technique for identifying parts of

[11] 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

an image that match a given image pattern[12]. 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 Chapter 9. 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 Chapter 9 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 they do not handle rotations or scale changes, which are quite common in real world applications.

[12] Advanced template matching algorithms enable finding pattern occurrences regardless of their orientation and local brightness.

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. We can use this concept, referred to as using *image pyramids*[13], to accelerate object search by first using a coarser resolution image to localize the object, and then using finer resolution images for actual detection. We can build image pyramids 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. We refer to the sequence of images resulting from this approach as a *Gaussian pyramid*.

[13] R. Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010

10.2.2 Bag of Visual Words

The key idea behind the bag of visual words[14] approach is that we can simplify object representations by considering them as a collection of their subparts[15], and we refer to the subparts as *visual words*. In this approach, we search a source image for *visual words*, and we create a distribution of visual words that we find in the image in the form of a histogram. We can then perform object detection 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, we would likely determine that the training images that also have eyes and a nose are also images of faces.

[14] The model originated in natural language processing, where we consider texts such as documents, paragraphs, and sentences as collections, or "bags", of words.

[15] For example, a bike is an object with wheels, a frame, and handlebars.

10.2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) represent a very powerful paradigm in object recognition[16]. These approaches were first introduced in the field of computer vision for image recognition in 1989, and since then have significantly

[16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016

boosted performance in image recognition and classification tasks. Research in this field is still active.

References

[15] Martin A. Fischler and Robert C. Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". In: *Commun. ACM* 24.6 (1981), pp. 381–395.

[18] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[52] 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.

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

[71] R. Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

11
Deep Learning Architectures for Perception

11.1 Convolutional Neural Networks (CNNs)

Convolutional neural networks are a type of deep learning architecture that is very common in the fields of computer vision and image processing. The architecture of a CNN contains a special structure that leverages convolution filters similar those we discussed in Chapter 9. However, in the context of machine learning, the convolution filters embedded in the structure of a CNN are optimized for the desired task and do not require human specification, giving higher performance and reducing the amount of required manual engineering. CNN architectures are comprised of several main components, convolutional layers, nonlinear activations, pooling layers, and fully-connected layers. We will go over each of these core building blocks in this section:

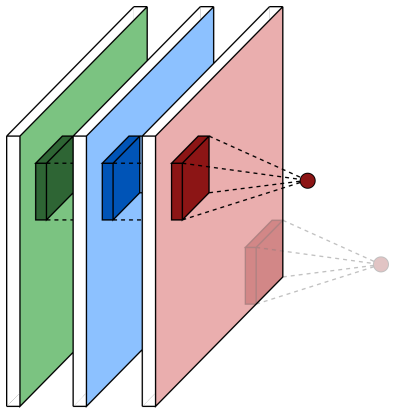


Figure 11.1: A convolution filter being applied to a 3-channel RGB image.

Convolution Layers. One of the main structural concepts that is unique to the architecture of a CNN is the use of convolution layers. Convolution 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 we can express them 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 11.1, we show how a filter is applied over an image with three color channels (red, green, and blue). In this case, the filter has dimension $m \times n \times 3$, which is vectorized to a weight vector, w, with $3mn$ elements. The *stride* of the filter describes how many positions it shifts by when sliding over the input. The output of the filter is then passed through a nonlinear activation[1].

[1] Typically a ReLU function.

Once we have applied the filter to the entire image, the collection of outputs from the nonlinear activation function create a new filtered image, which we typically refer to as an *activation map*. In practice, a number of different filters

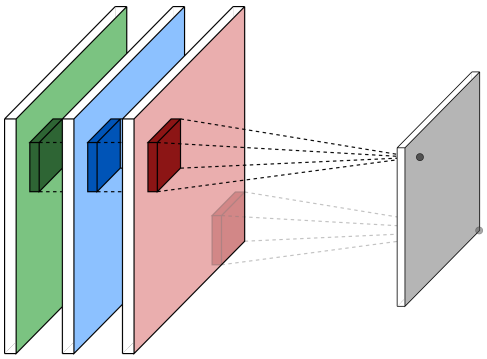


Figure 11.2: The outputs of a convolution filter and activation function applied across an image make up a new image, called an *activation* map.

are usually learned in each convolution layer, which produces a corresponding number of activation maps as the output[2]. This is crucial such that each filter can focus on learning one specific relevant feature. We show examples of different filters that might be learned in different convolution layers of a CNN in Figure 11.3[3]. Notice that the low-level features which are learned in earlier convolution layers look a lot like edge detectors, which are more basic and fundamental features, while later convolution layers have filters that look more like actual objects.

[2] 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.

[3] M. D. Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks". In: *European Conference on Computer Vision (ECCV)*. Springer, 2014, pp. 818–833

In general, using convolution layers to exploit the spatial locality of images provides several benefits. One benefit is parameter sharing, where the (small) filter's parameters are applied at all points on the image. This keeps the total number of learned parameters in the model much smaller than if we used a fully-connected layer. Another benefit is that the sparse interactions from 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. The convolution layer is also equivariant to translation, meaning that the convolution of a shifted image is equivalent to

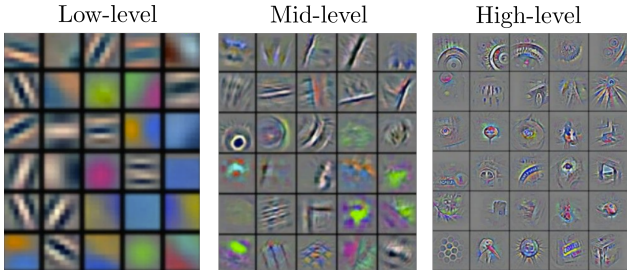


Figure 11.3: Low-level, mid-level, and high-level feature visualizations in a convolutional neural network from Zeiler and Fergus (2014).

the shifted convolution of the original image[4]. This allows convolution layers to detect features regardless of their position in the image. Finally, the use of convolution layers gives us the ability to work with images of varying size, if needed.

[4] However, convolution is not equivariant to changes in scale or rotation.

Pooling Layers *Pooling* is the second major structural component in CNNs. Pooling layers typically come after convolution layers and their nonlinear activation functions. The primary function of a pooling layer 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[5], and is useful in enabling the input images to vary in size[6]. The most common type of pooling is *max pooling*, but other types also exist, such as *mean pooling*.

[5] This occurs because it lowers the resolution.

[6] The size of the pooling can be modified to keep the size of the pooling layer output constant.

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.

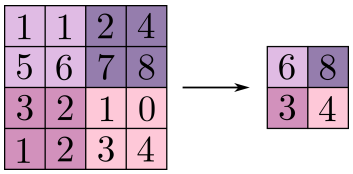


Figure 11.4: Max pooling example with 2 $\times$ 2 filter and stride of 2.

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 we can think of as a highly condensed representation of the image, and 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.

CNN Performance We can say that a CNN learns how to process images *end-to-end* because it essentially learns how to perform two steps simultaneously: fea-

ture extraction and classification or regression[7]. 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 with more parameters, and also by combining CNN architectures with other techniques such as Transformers.

[7] In other words, it learns the entire process from image input to the desired output.

[8] In some specific applications, hand-engineered features may still be better. For example, we might use engineering insight to identify a structure to the problem that a CNN could not easily learn.

Notable Architectures Several landmark CNN architectures have significantly advanced the field of computer vision and demonstrated the power of deep learning. AlexNet (2012) was the first deep CNN to achieve breakthrough performance on ImageNet, popularizing the use of ReLU activations and dropout regularization while demonstrating that deeper networks could dramatically outperform traditional methods. ResNet (2015) introduced residual connections that allow information to skip layers, enabling the training of much deeper networks (up to 152 layers) without suffering from vanishing gradients, and showed that depth itself could be a key factor in improving performance. YOLO (You Only Look Once) represents a different approach to CNN design, pioneering real-time object detection by treating detection as a single regression problem rather than a classification task, demonstrating how CNN architectures can be adapted for various computer vision tasks beyond simple image classification. These architectures have not only achieved state-of-the-art results in their respective domains but have also influenced countless subsequent designs and established important principles for CNN development.

11.2 *Vision Transformers (ViTs)*

A vision transformer[9] leverages a transformer architecture for computer vision applications by segmenting an image into a set of square patch tokens that are embedded by a linear operation. Since this architecture does not contain some of the structural constraints of CNN-based models, it requires larger training datasets to achieve similar performance with similar model size. However, empirical results have shown that with sufficiently large datasets it is possible for vanilla Transformer architectures to perform well. In practice, we can also pre-train a vision transformer on a large scale image dataset, followed by fine-tuning on smaller datasets for specific applications. Below, we will discuss each component in more detail.

[9] A. Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *International Conference on Learning Representations*. 2021

Transformers Transformers are a deep learning architecture that have been widely applied in a variety of domains, including machine translation, natural language processing, computer vision, and more. Compared to CNNs, Transformers enforce fewer structural constraints, and as long as we can organize the input data into a set or sequence, it can be processed by a Transformer-based model. Since the internal structure of the Transformer is purely learned from data, it requires less domain knowledge, which makes it easier to generalize to

different data modalities. Transformers are also quite computationally efficient and scalable due to their ability to be parallelized, allowing for extremely large models with hundreds of billions of parameters to be trained.

Transformer architectures have several main components: tokenizers, embedding layers, transformer layers, and unembedding layers. In this section, we briefly introduce each of these main components.

Tokens and Embeddings Transformers take inputs in the form of a set or sequence of tokens. A *token* is a numeric representation of the raw input data[10], expressed by a vector. For example, for a language task, a token could be a numeric representation of a word, or set of words or characters. We can generate language tokens through a word-to-token mapping defined by a learned dictionary lookup. In the context of computer vision, a token is often a numeric representation of a *patch*[11] of an input image. For example, we can convert a patch of size $P \times P$ with C color channels into a token vector of size CP^2 by flattening the patch.

[10] A *tokenizer* refers to the method for generating tokens from the raw inputs.

[11] A patch is usually a square subset of the full image.

We then transform tokens into a new vector, which we refer to as a token *embedding* vector. A token embedding vector is a high-dimensional latent space representation of the token. In computer vision tasks, we can also append a special "class" token embedding that the model can use as a workspace to reason about the classification of the image. We also typically add an embedding of the token's position to the embedding vector, since this information is often also relevant[12]. The embeddings are learned, for example as a linear operation, and we can think of the embedding latent space as a space where "closeness" of two embedding vectors implies there is some similarity between the two tokens.

[12] The position embedding could give information about the position of a word in a sequence or position of a patch in an image. For example, in the sentence, "The dog chased the cat.", it is important to know that "dog" comes before "cat" to capture the sentence's correct meaning.

Once we have transformed the raw input into a set of embedding vectors, we aggregate them as rows of an input matrix, X^0, of dimension $N \times D$, where N is the *context size*[13] and D is the size of the embedding space.

[13] In general, larger context sizes will give better performance because we can capture more unique information. However, this comes at the cost of increased computational requirements.

Transformer Layer One of the key components of the Transformer architecture is the *self-attention layer*. Each embedding vector from the input, which is a row of X^0, represents only a single token. Therefore, they don't yet contain contextual information captured by the other tokens. For example, in the sentence, "The child played with the toy car." the embedding vector for the word "car" doesn't have awareness of the adjective "toy". Our goal with self-attention is to allow the model to figure out how to modify each embedding vector from the input in a way that is relevant, given the entire set of inputs. In the previous example, we want the model to learn that the "car" embedding vector can be modified by the "toy" adjective so that it becomes a new vector in the high-dimensional embedding space that corresponds to "toy car" rather than some other type of car, like a delivery van.

Mathematically, *scaled dot-product* self-attention layers have the form:

$$O = \text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^\top}{\sqrt{d_k}})V,$$

where we call $Q \in \mathbb{R}^{m \times d_k}$ the *query* matrix, $K \in \mathbb{R}^{n \times d_k}$ the *key* matrix, $V \in \mathbb{R}^{n \times d_v}$ the *value* matrix, and $O \in \mathbb{R}^{m \times d_v}$ the *output* matrix. The intuition of this mathematical formulation is that the term $QK^\top$ is taking the dot product between all pairs of queries and keys, represented by the rows of Q and K, and a larger dot product suggests that a particular pair are similar to each other. We then apply the softmax function to each column of the resulting matrix to normalize the columns so that their elements sum to one[14]. Finally, the multiplication by the value matrix, V, makes it such that each row of the output matrix, O, is the weighted sum of the rows of the value matrix, where the weights correspond to the normalized dot products.

[14] The division by $\sqrt{d_k}$, where d_k is the size of the keys and queries, is another trick to help with the stability of the training process.

In the context of the transformer, we typically define the matrices as $Q = XW_Q$, $K = XW_K$, and $V = XW_V$, where X is the input matrix with each row corresponding to one embedding vector from the input sequence and $W_{(\cdot)}$ are matrices of learnable parameters. We then add the output of the self-attention function to the the input in what we refer to as a *residual connection*:

$$\bar{X} = X + \text{Attention}(XW_Q, XW_K, XW_V).$$

Here we present the mathematical form using matrix notation, which is how these models are implemented in practice. However, for intuition, it is easier to reason about the transformation of a single input embedding vector, corresponding to a single row of X. For example, going back to the toy car sentence example, let's consider the embedding vector for the word "car". The attention function modifies this vector into a new vector in the embedding space that captures more information from the rest of the sentence. Specifically, the value matrix, V, would give a set of potential modifications corresponding to the other words of the sentence, and then we pick which modifications to actually apply by the dot-product relevance weighting. The relevance weightings, which specify which other words in the sentence we should *attend to*, would likely show there is a strong match between "car" and "toy".

A single attention mechanism is limited in the ways it can define relevance among the various tokens, specifically by the finite matrices W_Q, W_K, and W_V. We can increase the model's capability through *multi-head attention*, which simply defines a number of attention mechanisms with unique parameters, W_Q^i, W_K^i, and W_V^i, concatenates their results, and multiplies by another learned matrix, W_O.

Next, we pass the output of the multi-head self-attention layer into a feed-forward layer, such as a *multi-layer perceptron*. A multi-layer perceptron is a sequence of layers that alternate between a fully connected linear function and a non-linear activation function[15]. Specifically, we apply a multi-layer perceptron to each individual token of the output of the attention layer.

[15] For example, a rectified linear unit (ReLU).

Each transformer layer consists of the two main components of a multi-head attention layer followed by the multi-layer perceptron. After each of these components, there is a residual connection that adds the output of the component to the input[16]. We then stack some number of these Transformer layers in se-

[16] We also usually apply a normalization operation to help stabilize training, and also leverage other common training tricks such as *dropout*.

quence[17].

[17] Note that the dimensions of the inputs and outputs of each Transformer layer are typically the same, $N \times D$.

Unembedding Layer Once we have converted the raw input into a sequence of tokens, converted them into embedding vectors, and passed them through the Transformer layers, our next step is to convert the output from the embedding space back into the space of tokens. Specifically, we take an embedding vector, x, and convert it into a probability distribution over tokens by using a softmax function:

$$o = \text{softmax}(xW + b),$$

where W and b are learned parameters. The output, o, is a vector whose size matches the size of the token vocabulary, and whose elements sum to one.

Practical Considerations Vision Transformers represent a paradigm shift in computer vision by adapting the flexible transformer architecture to image processing tasks through patch-based tokenization and self-attention mechanisms. Unlike CNNs, which rely on built-in spatial inductive biases through convolution operations, ViTs learn spatial relationships purely from data via positional embeddings and attention patterns, requiring larger datasets but offering greater flexibility across different visual tasks. The self-attention mechanism allows each image patch to attend to all other patches globally, enabling the model to capture long-range dependencies that might be difficult for CNNs with their local receptive fields. While ViTs typically require more training data than CNNs to achieve comparable performance on smaller datasets, they demonstrate superior scalability and can be effectively pre-trained on large-scale datasets before fine-tuning for specific applications. This approach has proven particularly successful in scenarios where large amounts of training data are available, establishing ViTs as a complementary and sometimes superior alternative to traditional convolutional approaches in computer vision.

11.3 PointNet and Point Cloud Processing

Point Cloud Characteristics Point clouds represent 3D data as collections of points in three-dimensional space, where each point is typically defined by its position, $\mathbf{p}$, in space[18] and may include additional attributes such as color, intensity, or surface normals. This representation is particularly prevalent in autonomous driving systems, where LiDAR (Light Detection and Ranging) sensors capture millions of points per second to create detailed 3D maps of the vehicle's surroundings, enabling real-time obstacle detection and navigation. Unlike images, which have a regular 2D grid structure with fixed dimensions, point clouds possess several unique characteristics that make them challenging to process with traditional deep learning architectures.

[18] The exact representation can vary: Cartesian coordinates (x, y, z), spherical coordinates (r, θ, ϕ), or cylindrical coordinates (r, θ, z) depending on the application and sensor type.

Point clouds are fundamentally **unordered sets** of points, meaning there is no inherent sequential or spatial ordering like the pixel arrangement in images.

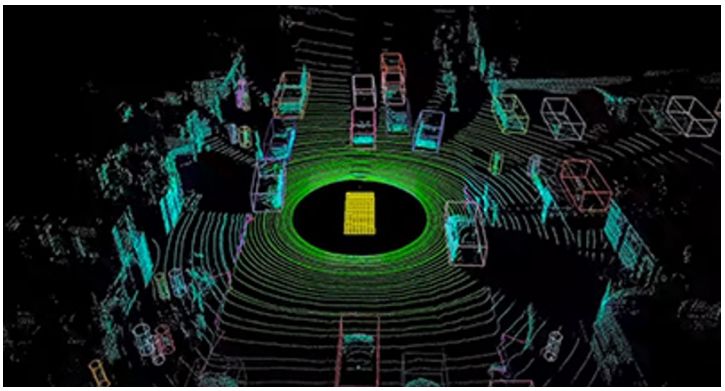


Figure 11.5: LiDAR point cloud data from an autonomous vehicle showing detected objects including cars, pedestrians, and road infrastructure.

A point cloud containing N points can be represented as a set $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_N$ where each point $\mathbf{p}_i \in \mathbb{R}^d$ (typically $d = 3$ for spatial coordinates). Crucially, any permutation of these points represents the same geometric object, which requires our neural network architectures to be **permutation invariant**.

Point clouds also exhibit **variable cardinality**, meaning different point clouds can contain vastly different numbers of points. While an image always has a fixed resolution (e.g., 224×224 pixels), one point cloud might contain 1,000 points while another contains 100,000 points, depending on the scanning resolution, distance from the sensor, and object complexity. This variability poses challenges for batch processing and network design. The **sparse and irregular**

Figure 11.6: Point clouds of different objects showing variable cardinality: a simple table (more points), a detailed chair (more points), and a toy building facade (most points).

nature of point cloud data differs from the dense, regular grid of image pixels. Points are distributed non-uniformly throughout 3D space, with varying local densities that depend on the scanning angle, distance, and surface properties of the objects being captured. This sparsity means that most of the 3D space contains no points, and the local neighborhood structure around each point is highly variable and irregular. These fundamental differences necessitate specialized neural network architectures that can handle the unique properties of point cloud data.

PointNet Architecture PointNet approaches point cloud processing through several key architectural innovations that ensure permutation invariance while extracting meaningful geometric features. The architecture consists of point-wise

feature extraction, symmetric aggregation functions, and spatial transformation components that work together to process unordered point sets effectively.

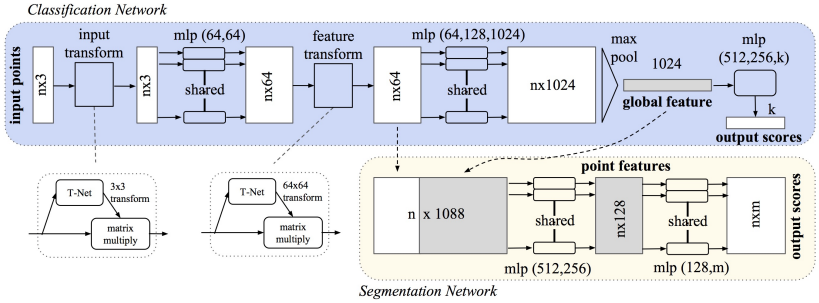


Figure 11.7: PointNet architecture showing point-wise MLPs, transformation networks (T-Net), and symmetric aggregation for classification and segmentation tasks.

Point-wise Multi-Layer Perceptrons The foundation of PointNet[19] is the application of multi-layer perceptrons (MLPs) to individual points in the point cloud. Given a point cloud with N points, where each point $p_i \in \mathbb{R}^3$ represents spatial coordinates, PointNet applies the same MLP function f to each point independently:

$$h_i = f(p_i) = \text{MLP}(p_i)$$

where $h_i \in \mathbb{R}^k$ is the learned feature representation for point p_i. This point-wise processing is crucial for maintaining permutation invariance because the same transformation is applied to each point regardless of its position in the input sequence. The MLP typically consists of several fully connected layers with ReLU activations, progressively increasing the feature dimensionality from the input coordinates (usually 3D) to higher-dimensional feature spaces (e.g., 64, 128, 1024 dimensions). Importantly, the parameters of this MLP are shared across all points, similar to how convolutional filters share parameters across spatial locations in CNNs, but without the spatial locality constraints.

[19] details on where to find more

Symmetric Aggregation Functions After extracting point-wise features, PointNet must aggregate these features into a single global representation while preserving permutation invariance. This is achieved through symmetric functions that produce the same output regardless of input ordering. The most commonly used symmetric function in PointNet is the element-wise maximum:

$$g = \max_{i=1,\ldots,N} h_i$$

where the max operation is applied element-wise across all feature vectors h_i. Mathematically, this can be proven to be permutation invariant: if σ is any permutation of the points, then $\max_{i=1,\ldots,N} h_{\sigma(i)} = \max_{i=1,\ldots,N} h_i$. While alternative symmetric functions like summation or mean could be used, max pooling has the advantage of being selective, allowing the network to focus on the most

discriminative features across all points. However, this global aggregation approach means that PointNet captures only global features and may miss important local geometric structures, which motivates the hierarchical extensions in PointNet++.

Transformation Networks (T-Net) To achieve invariance to geometric transformations such as rotation and translation, PointNet incorporates transformation networks (T-Net) that learn to align point clouds to a canonical orientation. The T-Net is itself a mini-PointNet that predicts a transformation matrix $T \in \mathbb{R}^{k \times k}$:

$$T = \text{T-Net}(\{p_1, p_2, \ldots, p_N\})$$

This transformation matrix is then applied to either the input coordinates (input transform) or intermediate features (feature transform). For the input transform, $T \in \mathbb{R}^{3 \times 3}$ aligns the spatial coordinates, while for the feature transform, $T \in \mathbb{R}^{64 \times 64}$ normalizes the feature space. To ensure the stability of optimization, a regularization term is added to the loss function that encourages the transformation matrix to be close to orthogonal:

$$L_{\text{reg}} = ||I - TT^T||_F^2$$

where $|| \cdot ||_F$ denotes the Frobenius norm. This regularization prevents the transformation from becoming degenerate and helps maintain the geometric properties of the point cloud.

Hierarchical Feature Learning (PointNet++) While PointNet effectively captures global features, it struggles to learn local geometric patterns due to its reliance on global max pooling. PointNet++[20] addresses this limitation by introducing a hierarchical architecture that learns features at multiple scales, similar to how CNNs build hierarchical representations through multiple convolutional layers. The core innovation in PointNet++ is the set abstraction layer, which recursively applies the PointNet architecture to local regions of the point cloud. Given a point cloud with N points, each set abstraction layer samples N' representative points (where $N' < N$), groups nearby points around each representative point, and applies a PointNet to extract local features. This process creates a hierarchical pyramid of features, where early layers capture fine-grained local details and later layers capture broader geometric patterns. Set abstraction in PointNet++ consists of three key operations: sampling, grouping, and feature extraction. **Sampling** uses farthest point sampling (FPS) to select representative points that provide good coverage of the entire point cloud. Given a set of points, FPS iteratively selects the point that is farthest from all previously selected points, ensuring diverse spatial coverage. **Grouping** then defines local regions around each selected point using either ball query (all points within radius r) or k-nearest neighbors. This creates local point sets of varying sizes that capture the local geometry around each representative point. **Feature extraction** refers to feature propagation layers that upsample features from coarser

[20] details on where to find more

to finer resolutions. These layers use inverse distance weighted interpolation to propagate features from subsampled points back to the original point cloud:

$$f^{(j)}(x) = \frac{\sum_{i=1}^{k} w_i(x) f_i^{(j-1)}}{\sum_{i=1}^{k} w_i(x)}, \quad w_i(x) = \frac{1}{d(x, x_i)^p}$$

where $f^{(j)}$ represents features at layer j, $d(x, x_i)$ is the distance between points, and p is typically set to 2. Skip connections between corresponding abstraction and propagation layers help preserve fine-grained details, similar to U-Net architectures in image segmentation.

Graph Neural Networks for 3D Data An alternative approach to processing point clouds treats them as graph structures, where points serve as nodes and edges are defined based on spatial proximity or learned relationships. Dynamic Graph Convolutional Neural Networks (DGCNN)[21] exemplify this approach by constructing graphs dynamically in feature space rather than just coordinate space. DGCNN applies edge convolution operations that aggregate information from neighboring points:

[21] details on where to find more

$$x_i' = \max_{j:(i,j)\in\mathcal{E}} h_\theta(x_i, x_j - x_i)$$

where x_i and x_j are feature vectors of connected points, h_θ is a learnable function (typically an MLP), and the edge set $\mathcal{E}$ is dynamically updated based on feature similarity after each layer. This dynamic graph construction allows the network to capture both geometric and semantic relationships that evolve as features are learned. The edge convolution operation differs from standard graph convolutions by explicitly modeling the edge information $(x_j - x_i)$, which captures the relative geometric relationships between neighboring points. This approach has shown success in tasks like point cloud classification and part segmentation, often outperforming PointNet on datasets where local geometric structure is crucial.

Point-based Feature Learning for Downstream Tasks After processing through PointNet's point-wise MLPs and symmetric aggregation, or PointNet++'s hierarchical set abstraction layers, these architectures produce rich point-wise feature representations that encode both local geometric patterns and global shape information. These learned features serve as powerful inputs for various downstream tasks including 3D object detection, semantic segmentation where each point is classified into categories like road, building, or vegetation, and instance segmentation for identifying individual object instances within the point cloud. We will explore the details of training point-based detection and segmentation networks in a subsequent section on object detection architectures.

Computational Limitations and Alternative Approaches Despite their intuitive design, point-based methods including PointNet, PointNet++, and DGCNN face significant computational challenges when processing large-scale point clouds.

Real-world applications like autonomous driving can generate point clouds with millions of points per frame, making the $O(N^2)$ complexity of neighborhood search in DGCNN or the recursive sampling in PointNet++ computationally prohibitive. Memory requirements also scale poorly, as each point must be processed individually, leading to irregular memory access patterns that are inefficient on modern GPU architectures. In the following section, we will discuss some voxel-based and pillar-based approaches that leverage regular grid structures for efficient 3D convolutions. By discretizing the 3D space into regular voxels or vertical pillars, these methods can apply standard convolutional operations while maintaining spatial locality and enabling efficient parallel processing. This structured representation trades some geometric precision for computational efficiency and scalability, making it particularly suitable for real-time applications in autonomous driving and robotics where processing speed is critical.

11.4 3D Convolutions: VoxelNet and PointPillars

In the previous section, we discussed that the computational limitations of point-based methods have motivated the development of grid-based approaches that discretize 3D space into regular structures, enabling the application of efficient convolutional operations. Rather than processing individual points with irregular neighborhoods, voxel-based and pillar-based methods transform point clouds into structured representations where standard CNNs can be applied. This paradigm shift trades some geometric precision for substantial computational advantages, making real-time processing of large-scale point clouds feasible for applications like autonomous driving. By leveraging the regularity of grid structures, these methods can utilize optimized convolution implementations and parallel processing capabilities of modern hardware.

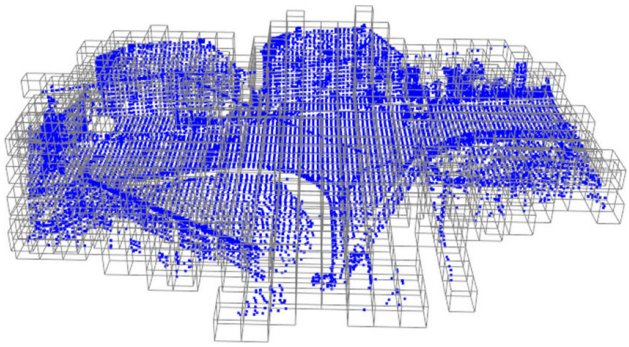


Figure 11.8: Comparison of point cloud representations: (blue) original point cloud overlaid on the (grey) voxel-based discretization into 3D cubic cells.

Voxel and Pillar-based Representations Grid-based methods transform irregular point clouds into structured representations by discretizing 3D space into regular units. Two primary approaches have emerged: voxel-based representations

that divide space into cubic voxels, and pillar-based representations that use vertical columns extending through the entire height of the scene[22]. The **voxel-based representation** creates a full 3D regular grid structure of size $L \times W \times H$ by partitioning space into cubic cells of size $v_l \times v_w \times v_h$[23], where each voxel can contain zero or more points from the original point cloud. This approach preserves complete spatial relationships in all three dimensions, enabling rich 3D feature learning through volumetric convolutions. Points within each voxel are aggregated into a single feature representation, typically through operations like mean pooling, max pooling, or learned aggregation functions. In contrast, the **pillar-based representation** adopts a 2.5D approach, looking at the scene from a birds-eye view, and treats vertical columns ("pillars") as the fundamental processing unit. The space is transformed into a 2D grid structure of size $L \times W$. Each pillar extends vertically through the entire height range of the point cloud, effectively collapsing the height dimension during initial processing. This approach is particularly useful in self-driving settings, where the scene processed is often very large and the reduction to 2D significantly improves computational efficiency. Points within each pillar are aggregated while preserving some height information through encoding strategies, but the primary spatial reasoning occurs in the horizontal, birds-eye view plane.

[22] These are frequently used in autonomous vehicle or navigation domain, where the scene can be viewed as a 2D "map" instead of a true 3D scene, for computational efficiency

[23] In practice, these dimensions are often set to be equal, creating cubic voxels.

The choice between these representations involves significant trade-offs in computational complexity and spatial information preservation. Voxel-based methods provide richer spatial context by maintaining full 3D neighborhood relationships with memory scaling as $O(L \times W \times H)$, enabling detection of complex 3D geometric patterns but requiring computationally expensive 3D convolutions. Pillar-based approaches reduce complexity by projecting the problem into 2D with memory scaling as $O(L \times W)$, enabling the use of mature 2D CNN architectures and optimized implementations, but potentially losing important vertical structure information crucial for multi-level feature detection. Both representations face spatial resolution trade-offs, where finer grids capture more geometric detail at exponentially higher computational cost, and must address sparsity challenges where most grid cells remain empty, motivating the development of sparse convolution techniques.

Sparse Convolutions for Efficiency Real-world point clouds exhibit extreme sparsity when discretized into voxel grids. In many applications such as autonomous driving and indoor scene processing, the majority of 3D space consists of empty air or unoccupied regions. Standard dense 3D convolutions waste significant computation on empty space, making them impractical for large-scale applications. Sparse convolutions address this by computing only on occupied voxels and their neighborhoods, using efficient data structures to maintain compact representations of non-empty regions[24]

[24] Popular implementations include `spconv` for PyTorch/TensorFlow and Minkowski Engine for general sparse tensor operations.

Example 11.4.1 (Memory Savings in Sparse Convolutions)**.** A typical autonomous-vehicle LiDAR scene discretized at 10cm resolution over a $100m \times 100m \times 10m$

volume would require:

$$\text{Grid size} = \frac{100m}{0.1m} \times \frac{100m}{0.1m} \times \frac{10m}{0.1m} = 1000 \times 1000 \times 100$$

$$\text{Total voxels} = 1000 \times 1000 \times 100 = 100 \text{ million voxel features per layer}$$

In contrast, sparse representations store only the occupied voxels, making memory usage proportional to the number of non-empty voxels rather than total grid size. The sparser the scene, the greater the memory savings.

3D Convolution Operations 3D Convolutional Neural Networks extend the successful principles of 2D CNNs to volumetric data by operating directly on 3D grids of voxels. While 2D convolutions slide filters across height and width dimensions of images, 3D convolutions add depth as a third spatial dimension, enabling the network to capture spatial relationships to understand the 3D scene. Specifically, a 3D convolution applies a filter of size (k_x, k_y, k_z) across all three spatial dimensions of the input volume. Mathematically, for an input volume X and filter W, the 3D convolution operation produces:

$$Y_{i,j,k} = \sum_{u=0}^{k_x-1} \sum_{v=0}^{k_y-1} \sum_{w=0}^{k_z-1} X_{i+u,j+v,k+w} \cdot W_{u,v,w} + b$$

where (i, j, k) represents the spatial position in the output volume and b is the bias term.

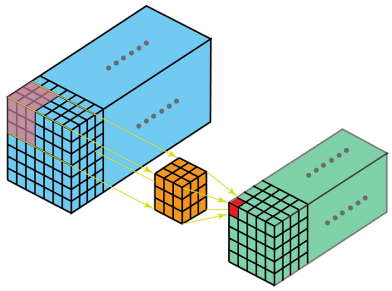


Figure 11.9: A 3D convolution filter sliding across a volumetric input, showing how the (k_x, k_y, k_z) filter operates in all three spatial dimensions.

Common kernel sizes include $3 \times 3 \times 3$ for capturing local 3D patterns and $1 \times 1 \times 1$ for channel-wise feature mixing without spatial aggregation[25]. The key advantage of 3D convolutions over approaches that process 2D slices independently is their ability to learn features that span multiple depths, such as the full 3D shape of objects or volumetric textures. The receptive field in 3D grows cubically with network depth, allowing deeper layers to capture increasingly global context, though this rapid growth must be balanced against increased computational cost.

[25] Larger kernels like $5 \times 5 \times 5$ can capture broader spatial context but significantly increase computational cost due to the cubic scaling of operations.

VoxelNet One of the pioneering architectures developed for processing voxel-based representations is VoxelNet[26]. The architecture addresses the challenge of

[26] Proposed by Zhou and Tuzel (2018), it has since developed into many variants, such as Voxel R-CNN and VoxelNeXt.

processing irregular point clouds by first voxelizing them into a 3D voxel grid, then leverages 3D convolutions that we described above to process them and detect objects.

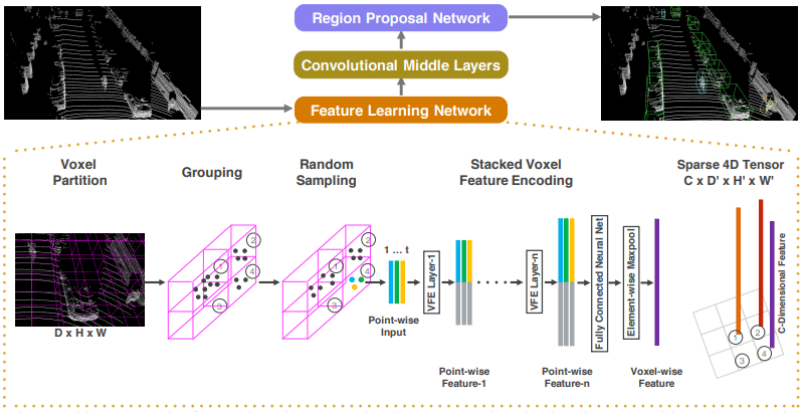


Figure 11.10: VoxelNet architecture showing the complete pipeline from point cloud voxelization through VFE layers, 3D convolutional middle layers, to the Region Proposal Network for 3D object detection.

Voxel Feature Encoding (VFE) Layers A core component of VoxelNet is its Voxel Feature Encoding layers, which process the variable number of points within each voxel to produce fixed-size feature representations. Given a voxel containing points $\{p_1, p_2, \ldots, p_n\}$, where each point $p_i = (x_i, y_i, z_i, r_i)$ includes spatial coordinates and optional reflectance intensity, the VFE layers apply point-wise multi-layer perceptrons to each point independently:

$$f_i = \text{MLP}(p_i)$$

To capture contextual information within each voxel, VoxelNet augments each point with the centroid of all points in the same voxel. For a voxel containing n points, the centroid is computed as $\bar{p} = \frac{1}{n}\sum_{i=1}^{n} p_i$, and each point is then represented as the concatenation $[p_i, p_i - \bar{p}]$, providing both absolute and relative spatial information.

The VFE layers then aggregate features across all points in the voxel using element-wise max pooling, ensuring permutation invariance:

$$v = \max_{i=1,\ldots,n} f_i$$

where v is the final voxel-level feature representation. This aggregation step converts the variable-sized point sets within each voxel into fixed-size feature vectors suitable for subsequent 3D convolution operations.

3D Convolutional Middle Layers After voxel feature encoding, VoxelNet applies a series of 3D convolutional layers to build hierarchical representations of the scene. These layers follow standard 3D CNN design principles, progressively increasing receptive field size while extracting increasingly abstract features.

The sparse nature of voxel occupancy makes this stage well-suited for sparse convolution implementations to improve computational efficiency. The 3D convolutional layers serve multiple purposes: they aggregate information across neighboring voxels to capture larger geometric structures, they build multi-scale representations through progressive downsampling, and they prepare features for the final object detection stage. The output of these layers is a dense feature map that encodes rich 3D spatial information about the scene.

PointPillars Architecture While VoxelNet processes full 3D voxels, PointPillars takes a different approach by using vertical pillars that extend through the entire height of the scene. The key difference lies in the Pillar Feature Network (PFN), which encodes points within each pillar and then converts the resulting pillar features into a 2D "pseudo-image" representation. This transformation allows PointPillars to leverage mature 2D CNN architectures for subsequent processing, rather than computationally expensive 3D convolutions. The pillar-based approach offers significant computational advantages by reducing the problem from 3D to 2.5D, enabling the use of optimized 2D convolution operations and existing hardware accelerations designed for image processing. This design choice makes PointPillars particularly suitable for real-time applications where computational efficiency is crucial. A notable variant that builds on similar principles is SECOND[27], which combines voxel-based processing with sparse convolution techniques.

[27] SECOND (Sparsely Embedded Convolutional Detection) combines voxel-based processing with sparse convolution techniques for improved efficiency.

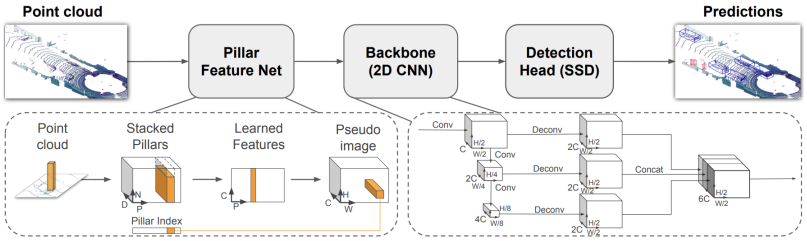


Figure 11.11: PointPillar architecture.

Object Detection with Learned Features Often, the desired use case for these point-cloud network architectures is to detect relevant scene objects, useful for planning a robot's next movement or to navigate in the space. To accomplish this detection task, networks must transform raw point cloud data into structured representations that can identify and localize objects such as predicting 3D bounding boxes. After processing through either VoxelNet's VFE layers and 3D convolutional middle layers, or PointPillars' PFN and 2D convolutional backbone, both architectures produce rich feature representations that encode geometric patterns and spatial relationships across the scene. VoxelNet's features capture full 3D spatial context through volumetric processing, while PointPillars' features efficiently encode vertical structure information within 2D feature maps. These learned features serve as input to a Region Proposal

Network (RPN) that generates 3D bounding box proposals for object detection. The key contribution of both approaches lies in demonstrating that the entire pipeline—from raw point cloud processing to 3D object detection—can be trained end-to-end, allowing the networks to learn optimal feature representations specifically for the detection task rather than relying on hand-crafted features. We will explore the details of training object detection networks and designing appropriate loss functions in a subsequent section on object detection.

11.5 Multi-modal Fusion Approaches

Modern robotic systems, particularly autonomous vehicles, rely on multiple sensors to achieve robust perception in diverse environmental conditions. This creates the challenge of designing architectures to effectively combining features from different network types: point-based features from architectures like PointNet or voxel-based features from networks like VoxelNet for processing LiDAR data, and 2D CNN features for processing RGB camera images. The architectural design choices for fusing these heterogeneous feature representations are a consideration in trade-offs between system performance and computational efficiency.

Fusion Strategies Multi-sensor fusion architectures can be categorized based on where in the network pipeline different modalities are combined. *Early fusion* architectures concatenate features from different modalities at the input or early feature extraction stages. For example, RGB images and LiDAR point clouds can be projected into a common bird's-eye view representation and concatenated channel-wise, feeding into a unified network architecture that processes joint representations from the earliest stages.

- *Late fusion* architectures maintain separate processing pipelines for each modality, with independent networks producing separate predictions that are combined at the output level. This design allows each branch to use specialized architectures optimized for its specific data type—2D CNNs for images and point-based or voxel-based networks for LiDAR—but may miss important cross-modal feature interactions during the learning process.
- *Intermediate fusion* architectures represent a hybrid approach, combining features at multiple stages throughout the network hierarchy. This design enables modality-specific feature extraction in early layers while allowing cross-modal feature interactions in deeper layers. Features from different sensor streams are aligned and fused at corresponding spatial scales, requiring careful architectural design to handle the different feature dimensions and spatial resolutions.

Technical Challenges Effective multi-sensor fusion architectures must address several technical challenges. *Spatial alignment* between features from different

modalities requires careful architectural design to ensure that features correspond to the same physical locations in 3D space. This often involves learnable projection and transformation layers that can handle calibration uncertainties. *Temporal synchronization* presents additional architectural challenges due to the fundamental differences in how these sensors operate. LiDAR sensors are typically rotational and continuously scan the environment in 360 degrees, meaning different portions of the surroundings are captured at slightly different times during a full rotation. In contrast, cameras capture entire images instantaneously, creating temporal misalignment between the two data streams. This often requires architectural components that can handle asynchronous data streams and compensate for the temporal offset between sensor modalities.

Performance Benefits Well-designed fusion architectures leverage the complementary strengths of different feature types. Features from 2D CNNs excel at capturing semantic and appearance information, while point-based or voxel-based features provide precise geometric and spatial relationships. The architectural challenge lies in designing fusion modules that effectively combine these different feature types while maintaining computational efficiency. Successful fusion architectures typically result in improved detection accuracy and enhanced robustness across diverse environmental conditions, making them essential for safety-critical robotic applications.

References

[12] A. Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *International Conference on Learning Representations*. 2021.

[77] M. D. Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks". In: *European Conference on Computer Vision (ECCV)*. Springer, 2014, pp. 818–833.

12
Object Detection and Recognition

For a robot to safely navigate and interact with the world around them, they need visual understanding capabilities beyond simple image classification—not only identifying what objects are present in an image, but also determining where they are located. Consider an autonomous vehicle navigating a busy intersection that must detect multiple pedestrians, vehicles, and cyclists while simultaneously understanding which pixels belong to the drivable road surface versus sidewalks or building facades. A household robot organizing a cluttered kitchen must not only detect individual objects like cups and plates, but also understand their precise boundaries to enable careful grasping and placement. An agricultural robot in an orchard must distinguish between individual fruits on overlapping branches while understanding the 3D structure of the scene to plan safe navigation paths.

These scenarios require three visual understanding tasks essential for robotics. *Object detection* identifies what objects are present and where they are located using bounding boxes. *Semantic segmentation* classifies every pixel into scene categories like "road" or "vegetation." *Instance segmentation* combines both capabilities, identifying individual object instances and their precise boundaries. Each serves different robotics needs: detection for obstacle avoidance and tracking, semantic segmentation for navigation and scene understanding, and instance segmentation for manipulation. Furthermore, many robotics applications require reasoning about full 3D structure. Autonomous vehicles need depth and relative positions of obstacles for path planning. Robotic arms need precise 3D object poses for manipulation. Search and rescue robots need detailed 3D scene understanding for navigation in complex environments. This motivates extending detection and segmentation to 3D sensor data, using the point cloud and voxel processing architectures from the previous chapter.

In this chapter, we will explore methodological developments that tackle these robotics perception tasks. The progression from expensive two-stage detectors to real-time one-stage approaches addresses the need for low-latency decisions in dynamic environments, while efficient 3D processing methods address computational challenges of real-time LiDAR processing. We will demonstrate how the CNN, PointNet, and voxel-based architectures from the previous

chapter can be extended to enable robust visual understanding for autonomous robotic systems.

12.1 2D Object Detection Foundations

Object detection extends beyond image classification by not only identifying what objects are present in an image, but also determining where they are located. This dual requirement—classification and localization—fundamentally shapes the architectural design of detection systems. Unlike classification networks that output a single prediction per image, detection networks must handle varying numbers of objects at different scales and positions, requiring specialized architectures that can efficiently process these challenges.

Definition 12.1.1 (Object Detection)**.** Given an input image I, object detection aims to identify all instances of objects from a predefined set of classes $\mathcal{C} = \{c_1, c_2, \ldots, c_K\}$ and localize each instance with a bounding box. Formally, the output is a set of detections $\mathcal{D} = \{(b_i, c_i, s_i)\}_{i=1}^{N}$ where $b_i = (x_i, y_i, w_i, h_i)$ represents the bounding box coordinates, $c_i \in \mathcal{C}$ is the predicted class, and $s_i \in [0, 1]$ is the confidence score.

The evolution of object detection architectures can be broadly categorized into two paradigms: two-stage detectors that separate object localization from classification, and one-stage detectors that perform both tasks simultaneously. Additionally, the choice between anchor-based and anchor-free methods represents a fundamental design decision that affects both training dynamics and inference efficiency. Understanding these foundational concepts provides the framework for extending detection principles to 3D scenarios and multi-modal sensor fusion, which we will explore in subsequent sections.

12.1.1 Two-Stage Detection: R-CNN to Fast R-CNN

“mention this in context of 2-stage detectors" *** de-emphasize anything prior faster rcnn

The two-stage detection paradigm emerged as a natural approach to the object detection problem by decomposing it into two sequential sub-problems: first generating a set of object proposals that likely contain objects, then classifying these proposals while refining their locations. This divide-and-conquer strategy proved highly effective, establishing the foundation for many subsequent detection architectures.

R-CNN: Establishing the Two-Stage Paradigm The original R-CNN (Regions with CNN features) architecture[1] established the two-stage paradigm through a three-step process that combined classical computer vision techniques with modern deep learning. The architecture begins with selective search[2], which generates approximately 2,000 region proposals per image by identifying regions likely to contain objects based on low-level visual cues.

[1] Proposed by Girshick et al. (2014), R-CNN demonstrated that CNN features could dramatically improve object detection performance when combined with traditional region proposal methods.

[2] Selective search is a graph-based segmentation algorithm that generates object proposals by hierarchically grouping superpixels based on color, texture, size, and shape compatibility.

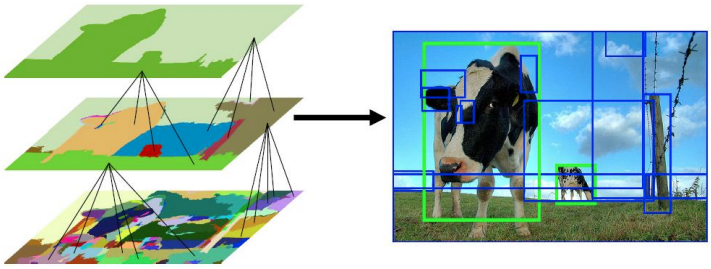


Figure 12.1: Selective search hierarchical grouping process. Starting from initial superpixel segmentation (left), regions are progressively merged based on color, texture, size, and shape compatibility, ultimately generating diverse object proposals of varying scales and locations (right).

Each proposed region is then warped to a fixed size of 227×227 pixels and processed independently through a pre-trained CNN (originally AlexNet) to extract a 4096-dimensional feature vector. This feature extraction step leverages the powerful representations learned by CNNs on large-scale image classification datasets, transferring this knowledge to the detection task. Finally, these CNN features are fed to class-specific Support Vector Machine (SVM) classifiers for object classification and linear regressors for bounding box refinement.

Mathematically, for a region proposal r with extracted CNN features $\phi(r)$, the classification score for class c is computed as:

$$s_c(r) = w_c^T \phi(r) + b_c$$

where w_c and b_c are the learned SVM parameters for class c. The bounding box regression predicts corrections $(\Delta x, \Delta y, \Delta w, \Delta h)$ to transform the proposal coordinates (x, y, w, h) to better align with the ground truth:

$$\begin{aligned}
\Delta x &= w_x^T \phi(r) + b_x \\
\Delta y &= w_y^T \phi(r) + b_y \\
\Delta w &= w_w^T \phi(r) + b_w \\
\Delta h &= w_h^T \phi(r) + b_h
\end{aligned}$$

While R-CNN achieved breakthrough detection performance on benchmark datasets, it suffered from significant computational inefficiencies and training complexity. Each of the 2,000 proposals required a separate forward pass through the CNN, making both training and inference extremely slow—processing a single image could take minutes. The multi-stage training process required pre-training the CNN on ImageNet, training SVMs for classification, and training linear regressors for bounding box refinement, making the pipeline complex and difficult to optimize end-to-end.

Fast R-CNN: Shared Computation Breakthrough Fast R-CNN[3] addressed these limitations through a key architectural innovation: shared computation across all proposals. Instead of processing each proposal independently through the CNN, Fast R-CNN processes the entire input image once through a convolutional backbone to generate a feature map, then extracts proposal-specific features from this shared representation.

[3] Introduced by Girshick (2015), Fast R-CNN addressed the computational bottlenecks of R-CNN while maintaining the two-stage paradigm and improving detection accuracy.

The central innovation is the Region of Interest (RoI) pooling layer, which extracts fixed-size feature representations from variable-sized proposal regions on the shared feature map. For a proposal with coordinates (x, y, w, h) on a feature map F with spatial dimensions $H \times W$, RoI pooling divides the proposal region into a regular $k \times k$ grid (typically 7×7) and applies max pooling within each grid cell:

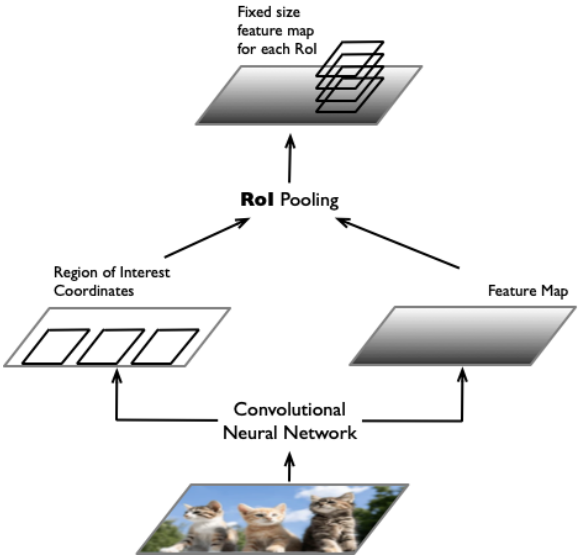


Figure 12.2: RoI pooling mechanism. A region of interest proposal of arbitrary size (left) is used to apply max pooling within the feature map (left) to produce a fixed-size feature representation per region (top) for subsequent classification and regression layers.

$$\text{RoI}_{i,j} = \max_{(x',y') \in \text{bin}_{i,j}} F_{x',y'}$$

where $\text{bin}_{i,j}$ represents the spatial extent of the (i, j)-th grid cell. This operation ensures that regardless of the input proposal size, the output is always a fixed $k \times k \times d$ feature tensor, where d is the number of feature channels.

Fast R-CNN also unified the training process through a multi-task loss function that jointly optimizes classification and bounding box regression:

$$L = L_{\text{cls}}(p, u) + \lambda[u \geq 1] L_{\text{bbox}}(t^u, v)$$

where $L_{\text{cls}}(p, u) = -\log p_u$ is the log loss for true class u with predicted class probabilities p, and L_{bbox} is the smooth L1 loss for bounding box regression. The indicator function $[u \geq 1]$ ensures that bounding box loss is only computed for positive examples (background class has $u = 0$), and λ balances the two loss terms.

This architectural change provided substantial improvements in both computational efficiency and detection accuracy. By sharing CNN computation across all proposals, Fast R-CNN reduced training time by an order of magnitude while achieving higher mean Average Precision (mAP) on standard benchmarks. The end-to-end training also eliminated the complex multi-stage opti-

mization procedure, making the system more practical for real-world deployment. However, Fast R-CNN still relied on external region proposal algorithms like selective search, which remained a computational bottleneck and prevented the entire detection pipeline from being truly end-to-end learnable. This limitation motivated the development of learnable region proposal methods, which we will explore in the next section.

12.1.2 *Learnable Proposals: RPN and Faster R-CNN*

While Fast R-CNN significantly improved computational efficiency through shared CNN computation, it still relied on external region proposal algorithms like selective search. These traditional methods suffered from several fundamental limitations: they were computationally expensive, requiring seconds per image; they were not learned from data and thus could not adapt to specific datasets or tasks; and they created a bottleneck that prevented the entire detection pipeline from being optimized end-to-end. The Region Proposal Network (RPN) innovation addressed these limitations by making region proposal generation a learnable component within the detection framework.

Limitations of Selective Search Selective search and similar traditional proposal methods operate using hand-crafted features and heuristics that remain fixed regardless of the detection task or dataset. These algorithms typically generate thousands of proposals per image using expensive graph-based operations, with processing times often exceeding the CNN inference itself. More critically, since these methods are not learnable, they cannot benefit from the supervision available during detection training—they cannot learn which types of regions are most likely to contain objects for a specific application domain.

Region Proposal Network Innovation The Region Proposal Network represents a paradigm shift by treating proposal generation as a learned prediction task. The RPN is essentially a fully convolutional network that slides a small network over the convolutional feature map produced by the backbone CNN. At each sliding window position, the RPN simultaneously predicts multiple region proposals using a set of reference boxes called anchors.

Anchor Box Design Principles Anchors serve as reference templates that cover different scales and aspect ratios at each spatial location in the feature map. For a feature map of size $H \times W$, the RPN generates $H \times W \times k$ potential proposals, where k is the number of anchor templates per location. Common anchor designs use 3 scales (e.g., 128^2, 256^2, 512^2 pixels) and 3 aspect ratios (e.g., 1:1, 1:2, 2:1), resulting in $k = 9$ anchors per location.

Mathematically, for an anchor centered at position (x_a, y_a) with width w_a and height h_a, the RPN predicts refinements $(\Delta x, \Delta y, \Delta w, \Delta h)$ to produce a final

proposal:

$$x = \Delta x \cdot w_a + x_a$$
$$y = \Delta y \cdot h_a + y_a$$
$$w = w_a \cdot \exp(\Delta w)$$
$$h = h_a \cdot \exp(\Delta h)$$

The exponential transformation for width and height ensures positive values and provides scale-invariant parameterization.

Objectness Scoring Unlike traditional proposal methods that use complex heuristics, the RPN performs binary classification to determine "objectness"—whether each anchor location contains an object of any class versus background. This objectness score p^* is simpler than full multi-class classification but captures the essential information needed for proposal generation. The RPN learns to distinguish object-like regions from background using the same convolutional features that will later be used for detailed classification.

RPN Loss Function The RPN is trained using a multi-task loss that combines objectness classification and bounding box regression:

$$L_{\text{RPN}} = \frac{1}{N_{\text{cls}}} \sum_i L_{\text{cls}}(p_i, p_i^*) + \lambda \frac{1}{N_{\text{box}}} \sum_i p_i^* L_{\text{box}}(t_i, t_i^*)$$

where L_{cls} is the log loss for binary classification, L_{box} is the smooth L1 loss for box regression, N_{cls} and N_{box} are normalization terms, and λ balances the two losses. The box regression loss is only computed for positive anchors (those with $p_i^* = 1$), indicated by the multiplication with p_i^*.

During training, anchors are assigned positive labels if they have Intersection over Union (IoU) > 0.7 with any ground truth box, or if they are the highest IoU anchor for a ground truth box. Anchors with IoU < 0.3 are assigned negative labels, while those with intermediate IoU values are ignored to avoid ambiguous supervision.

Faster R-CNN: Integration with Fast R-CNN Faster R-CNN combines the RPN with Fast R-CNN into a single, unified network that shares convolutional features between proposal generation and detection. The architecture consists of a shared CNN backbone (e.g., VGG-16 or ResNet), followed by two sibling branches: the RPN for generating proposals and the Fast R-CNN detection head for classifying proposals and refining their locations.

The shared backbone is crucial for computational efficiency—rather than running separate CNNs for proposal generation and detection, both tasks operate on the same feature representation. This sharing also enables the network to learn features that are beneficial for both tasks simultaneously.

Training Strategies Training Faster R-CNN requires careful coordination between the RPN and detection components. Initial approaches alternated between training the RPN and the detection network. First, the RPN is trained using ImageNet-pretrained features. Then, the detection network is trained using proposals from the trained RPN, fine-tuning the shared convolutional layers. This process can be repeated, though diminishing returns are typically observed after the first iteration.

Today, most training of both components is done simultaneously, using a combined loss function:

$$L_{\text{total}} = L_{\text{RPN}} + L_{\text{Fast R-CNN}}$$

Joint training is more efficient and often achieves better performance, as it allows the RPN and detection network to adapt to each other during learning.

Non-Maximum Suppression and Post-Processing After the RPN generates proposals, Non-Maximum Suppression (NMS) removes redundant detections. The algorithm sorts proposals by objectness score and iteratively removes proposals that have high IoU (typically > 0.7) with higher-scored proposals. This reduces the number of proposals fed to the detection stage from thousands to hundreds, improving computational efficiency while maintaining detection quality.

The complete Faster R-CNN pipeline processes an image through the shared backbone, generates scored proposals via RPN with NMS post-processing, extracts RoI features for the top proposals, and produces final classifications and refined bounding boxes. This end-to-end learnable system achieved significant improvements in both speed and accuracy over previous two-stage methods, establishing the foundation for modern object detection architectures.

12.1.3 One-Stage Detection: YOLO

The Region Proposal Network represented a major breakthrough by making proposal generation learnable, but it still required a two-stage pipeline where proposals were generated first and then classified separately. This sequential approach, while effective, created computational bottlenecks that limited real-time performance in robotics applications. As autonomous vehicles, drones, and mobile robots demanded faster detection systems for dynamic environments, a fundamental question emerged: could object detection be reformulated to predict bounding boxes and classes directly from image features in a single forward pass?

You Only Look Once (YOLO)[4] provided a radical answer to this question. Rather than decomposing detection into proposal generation followed by classification, YOLO treats object detection as a single regression problem, directly predicting bounding box coordinates and class probabilities from image pixels in one evaluation of the network. This paradigm shift eliminated the computational overhead of generating and processing thousands of proposals, enabling

[4] Introduced by Redmon et al. (2016), YOLO revolutionized object detection by demonstrating that competitive detection performance could be achieved through direct single-stage prediction, enabling real-time performance for robotics applications.

genuine real-time object detection suitable for robotics systems operating in dynamic environments.

Core YOLO Innovation: Grid-Based Direct Detection YOLO's central innovation lies in its spatial decomposition of the detection problem through a grid-based approach. The method divides the input image into an $S \times S$ grid (typically 7×7 for the original YOLO), where each grid cell becomes responsible for detecting objects whose center points fall within that cell's spatial region. This responsibility assignment creates a natural spatial organization that eliminates the need for separate proposal generation.

Each grid cell simultaneously predicts multiple bounding boxes along with their associated confidence scores and class probabilities. The key insight is that this grid-based spatial division provides sufficient spatial coverage while maintaining computational tractability—rather than evaluating thousands of potential object locations as in proposal-based methods, YOLO evaluates a fixed number of predictions per grid cell, resulting in a manageable total number of predictions regardless of scene complexity. The elimination of the proposal generation stage represents more than just a computational optimization; it fundamentally changes how the network approaches object detection. Rather than learning to generate good proposals and then classify them, the network must learn to directly map from image features to final detection outputs. This end-to-end learning enables the network to optimize the entire detection pipeline jointly, potentially leading to better coordination between localization and classification components.

YOLO Architecture and Predictions The YOLO architecture consists of a single CNN backbone followed by fully connected layers that produce the final detection tensor. The original implementation used a modified GoogLeNet architecture as the backbone, processing input images of size 448×448 pixels through convolutional layers that progressively reduce spatial resolution while increasing feature depth. The final convolutional features are flattened and processed through fully connected layers to produce a structured output tensor.

The network's output is a tensor of size $S \times S \times (B \times 5 + C)$, where S is the grid size, B is the number of bounding boxes predicted per cell, and C is the number of object classes. For the original YOLO trained on PASCAL VOC, this results in a $7 \times 7 \times 30$ tensor, with $B = 2$ bounding boxes and $C = 20$ classes. Each bounding box prediction consists of five values: $(x, y, w, h, \text{confidence})$, where (x, y) represents the box center relative to the grid cell boundaries, (w, h) represents the box dimensions relative to the entire image, and confidence represents the model's certainty that the box contains an object. The class predictions are formulated as conditional probabilities $P(\text{Class}_i|\text{Object})$, representing the probability of each class given that an object is present in the cell. This conditional formulation is crucial—each grid cell predicts only one set of class probabilities regardless of the number of bounding boxes, reflecting the as-

sumption that each cell is responsible for at most one object class.

The final detection confidence for each bounding box is computed by multiplying the conditional class probabilities with the bounding box confidence scores:

$$\text{Detection Confidence} = P(\text{Class}_i|\text{Object}) \times P(\text{Object}) \times \text{IoU}(\text{pred}, \text{truth})$$

This formulation ensures that high detection scores require both confident object presence and accurate localization.

Loss Function and Training YOLO's loss function addresses the multi-task nature of the detection problem by combining coordinate regression, confidence prediction, and classification into a unified objective. The loss function consists of several components with different weights to balance their relative importance:

$$\begin{aligned}
L = & \underbrace{\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}ij^{\text{obj}} [(x_i - \hat{x}i)^2 + (y_i - \hat{y}i)^2]}_{\text{bounding box center coordinates}} \\
& + \underbrace{\lambda\text{coord} \sum i = 0^{S^2} \sum_{j=0}^{B} \mathbb{1}ij^{\text{obj}} [(\sqrt{w_i} - \sqrt{\hat{w}i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}i})^2]}_{\text{bounding box dimensions}} \\
& + \underbrace{\sum i = 0^{S^2} \sum j = 0^{B} \mathbb{1}ij^{\text{obj}} (C_i - \hat{C}i)^2}_{\text{confidence for cells with objects}} \\
& + \underbrace{\lambda\text{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}ij^{\text{noobj}} (C_i - \hat{C}i)^2}_{\text{confidence for cells without objects}} \\
& + \underbrace{\sum i = 0^{S^2} \mathbb{1}i^{\text{obj}} \sum c \in \text{classes}(p_i(c) - \hat{p}i(c))^2}_{\text{class probabilities}}
\end{aligned}$$

The loss function uses different weights for different components: $\lambda_{\text{coord}} = 5$ increases the importance of coordinate predictions, while $\lambda_{\text{noobj}} = 0.5$ decreases the weight of confidence predictions for cells without objects. The square root transformation for width and height helps the loss function treat errors in small and large boxes more equally, since a small absolute error in a small box represents a larger relative error than the same absolute error in a large box.

The indicator function $\mathbb{1}_{ij}^{\text{obj}}$ denotes whether cell i contains an object and bounding box j is responsible for that prediction (determined by which predicted box has the highest IoU with the ground truth). This responsibility assignment is crucial for training stability, as it ensures each ground truth object is associated with exactly one predicted bounding box. Training YOLO follows a two-stage approach: the convolutional layers are first pre-trained on ImageNet for classification, then the entire network is fine-tuned on detection data. The

classification pre-training provides the network with strong feature representations that are then adapted for the detection task. During detection training, the learning rate is carefully adjusted to balance the different loss components and ensure stable convergence.

Speed vs. Accuracy Trade-offs and Robotics Impact YOLO's architectural design prioritizes computational efficiency, achieving detection speeds that were unprecedented at the time of its introduction. The original YOLO processes images at 45 frames per second (FPS) on contemporary GPU hardware, while a faster variant (Fast YOLO) achieved 155 FPS by using a smaller network architecture. These speeds represent order-of-magnitude improvements over contemporary two-stage methods like Fast R-CNN, which operated at approximately 7 FPS. However, this speed comes with accuracy trade-offs. YOLO's grid-based approach struggles with small objects, since multiple small objects within the same grid cell cannot be detected independently. The method also has difficulty with objects that appear in unusual aspect ratios, as the fixed number of bounding box predictors per cell limits the diversity of detectable shapes. Additionally, the coarse spatial quantization imposed by the grid structure can lead to less precise localization compared to methods that can place proposals at arbitrary locations.

For robotics applications, these trade-offs often represent acceptable compromises. Autonomous vehicles operating in real-time require detection systems that can process sensor data fast enough to support control decisions, even if absolute detection accuracy is somewhat reduced. The "good enough" detection philosophy embodied by YOLO aligns well with robotics applications where timely decisions often matter more than perfect perception. The impact of YOLO on the robotics field extends beyond its specific technical contributions. By demonstrating that real-time object detection was achievable with modest computational resources, YOLO democratized object detection for resource-constrained robotics platforms. Mobile robots, drones, and embedded systems could now incorporate sophisticated visual understanding capabilities without requiring expensive computational hardware.

The evolution of YOLO through subsequent versions (YOLOv2, YOLOv3, YOLOv4, YOLOv5, and beyond) has addressed many of the original accuracy limitations while maintaining the core computational advantages. Modern YOLO variants incorporate multi-scale feature processing, improved loss functions, and architectural refinements that close much of the accuracy gap with two-stage methods while preserving real-time performance. This progression demonstrates the enduring value of the single-stage detection paradigm for robotics applications where speed and efficiency are paramount.

12.1.4 *Transformer-based Object Detection*

Transformers have recently been adapted to tackle object detection, and their performance shows several benefits over CNN-based models. Detection Transformers (DETR)[5] propose to formulate the object detection problem as a direct set prediction problem, largely streamlining the detection pipeline through its end-to-end structure. In its most basic form, DETR is an end-to-end Transformer model that takes in images as inputs and predicts a fixed set of potential bounding boxes. DETR removes many hand-designed components, including "region proposal" and "non-maximum suppression" that are commonly used in CNN-based models.

[5] Nicolas Carion et al. "End-to-End Object Detection with Transformers". In: *Computer Vision – ECCV 2020*. Springer International Publishing, 2020, pp. 213–229

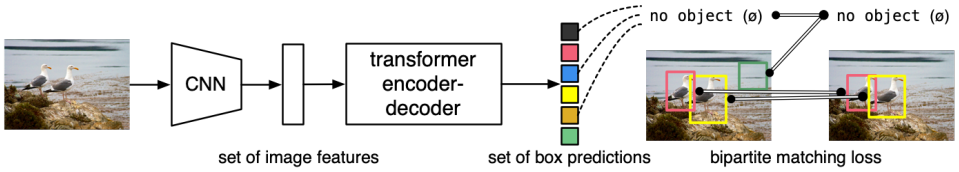


Figure 12.3: Detection Transformers (DETR) from Carion, Massa, et al. (2020)

Specifically, DETR uses a conventional CNN backbone to learn 2D feature maps from an input image, as we show on the left side of Figure 12.4. The model then converts the 2D feature maps into a sequence of feature tokens, similarly to Vision Transformers. These tokenized features are further fed into a Transformer encoder, comprised of a stack of self-attention mechanisms and multi-layer perceptrons, for further feature encoding. The encoded sequence is processed by a Transformer decoder that relates the feature sequence with a set of "learnable object queries". These object queries encode the distribution of object information, including size, location, and category, over an image.

Note that Transformer decoders have some key differences from encoders. For example, they can use what we refer to as *cross-attention layers* and *masked attention layers*. We use cross-attention layers to allow a sequence to get contextual information from another sequence, unlike self-attention layers which gather contextual information from within a single sequence. In the context of DETR, this allows the decoder to relate the encoder's feature embeddings to the object queries, which are two different input sets.

Finally, each object query, after absorbing image features, is processed by shared fully connected layers to predict class labels, bounding box centers and bounding box sizes. A "no object" label is assigned to queries without true objects detected, allowing the model to handle a variable number of objects in an image. In contrast to CNN-based object detectors, Transformer-based object detectors do not have one-to-one matching between the prediction set and the ground-truth set. Therefore, a set-based loss is used to produce an optimal bipartite matching between predicted and ground-truth objects, followed by optimizing object-centric (bounding box) losses.

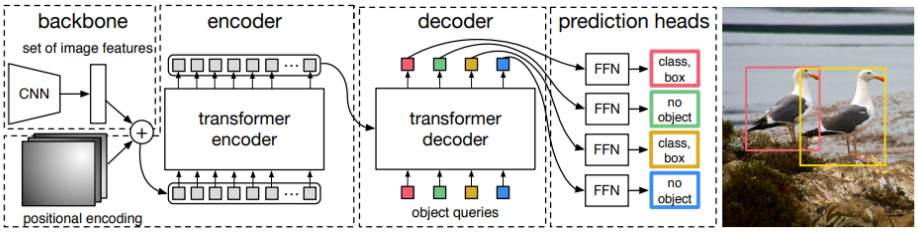


Figure 12.4: Detection Transformers (DETR) architecture, from Carion, Massa, et al. (2020)

12.2 3D Object Detection

While 2D object detection provides valuable information about what objects are present and their approximate locations in images, many robotics applications require understanding the full 3D structure and pose of objects in the physical world. Consider a robotic arm performing pick-and-place operations—knowing that a cup appears in a specific region of an image is insufficient for grasping; the robot needs the cup's precise 3D location, orientation, and dimensions to plan a successful grasp trajectory. Similarly, autonomous vehicles must understand the 3D positions and velocities of surrounding cars, pedestrians, and obstacles to make safe navigation decisions in real-world coordinates rather than image pixels. The transition from 2D to 3D detection introduces changes in problem formulation, data representation, and evaluation metrics while preserving many of the core architectural principles developed for 2D detection. Understanding these extensions provides the foundation for building robust 3D detection systems using the point cloud and voxel processing architectures from the previous chapter.

12.2.1 Extending Object Detection to 3D

3D object detection extends the 2D formulation by instead predicting 3D bounding boxes to represent objects in three-dimensional space. While 2D detection outputs bounding boxes parameterized by (x, y, w, h) in image coordinates, 3D detection requires additional parameters to specify the object's full pose and extent.

Definition 12.2.1 (3D Object Detection)**.** Given 3D sensor data (point cloud, voxel grid, or RGB-D), 3D object detection aims to identify all instances of objects from a predefined set of classes and localize each instance with a 3D bounding box. The output is a set of 3D detections $\mathcal{D}_{3D} = \{(b_i^{3D}, c_i, s_i)\}_{i=1}^{N}$ where $b_i^{3D} = (x, y, z, l, w, h, \theta)$ represents the 3D bounding box with center coordinates (x, y, z), dimensions (l, w, h) for length, width, and height, and orientation θ.

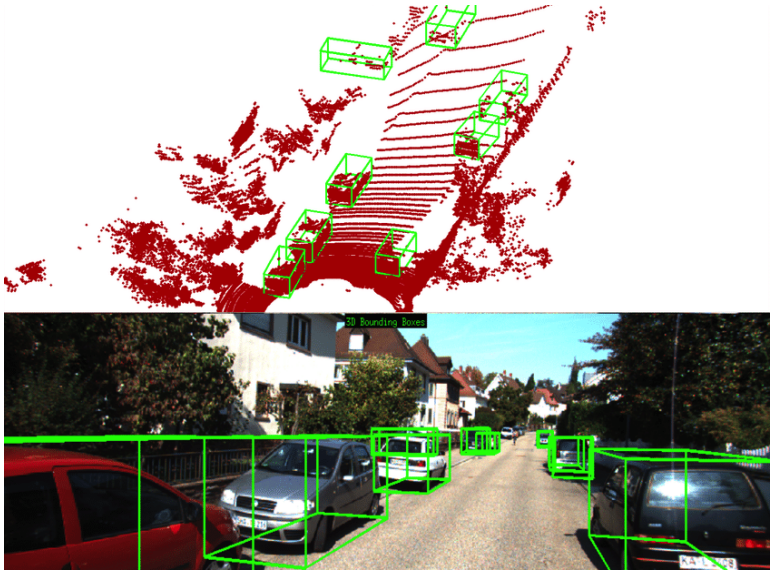


Figure 12.5: 3D Object Detection. A self-driving scene with 3D object detections on LiDAR point cloud inputs and camera inputs.

Extending Anchor Design to 3D Space For 3D anchor design, each anchor is parameterized by seven values: $(x_a, y_a, z_a, l_a, w_a, h_a, \theta_a)$ representing the center coordinates, dimensions, and orientation. Common 3D anchor designs use aspect ratios appropriate for the target object classes, and discrete orientation bins (e.g., 0°, 90°, 180°) to handle rotation invariance.

The anchor refinement process follows similar principles to 2D detection, with the network predicting corrections $(\Delta x, \Delta y, \Delta z, \Delta l, \Delta w, \Delta h, \Delta\theta)$ to transform anchor parameters into final detections:

$$
\begin{aligned}
x &= \Delta x \cdot l_a + x_a \\
y &= \Delta y \cdot w_a + y_a \\
z &= \Delta z \cdot h_a + z_a \\
l &= l_a \cdot \exp(\Delta l) \\
w &= w_a \cdot \exp(\Delta w) \\
h &= h_a \cdot \exp(\Delta h) \\
\theta &= \theta_a + \Delta\theta
\end{aligned}
$$

Similar to 2D setting, the exponential transformation ensures positive dimensions, while orientation is handled through additive corrections with appropriate normalization to handle angle wraparound.

Two-Stage vs. One-Stage Paradigms in 3D The two-stage and one-stage detection paradigms from 2D systems transfer directly to 3D detection, with each approach offering distinct advantages for different 3D data modalities and applications. *Two-stage 3D detectors* follow the proposal-then-classification paradigm, first generating 3D object proposals from point clouds or voxel grids, then re-

fining these proposals through dedicated classification and regression heads. This approach works particularly well with point-based representations, where the first stage can identify promising object centers using techniques like Hough voting, and the second stage can perform detailed classification using local point features. *One-stage 3D detectors* perform classification and localization simultaneously, making them better suited for real-time robotics applications where latency is critical. These methods work well with regular voxel or pillar representations that enable efficient convolutional processing across the entire 3D space. The choice between paradigms often depends on the input data modality: point-based methods naturally lend themselves to two-stage approaches due to the irregular nature of point clouds, while voxel-based methods can efficiently implement one-stage detection using 3D CNNs.

Non-Maximum Suppression in 3D Non-Maximum Suppression extends to 3D by replacing 2D IoU calculations with 3D IoU or Bird's Eye View (BEV) IoU metrics. 3D IoU computes the overlap between two 3D bounding boxes in full 3D space, accounting for differences in position, size, and orientation:

$$\text{IoU}_{3D}(b_1, b_2) = \frac{\text{Volume}(b_1 \cap b_2)}{\text{Volume}(b_1 \cup b_2)}$$

Computing 3D IoU requires determining the intersection volume between two oriented 3D boxes, which is more complex than the 2D case but essential for accurate duplicate removal.

For autonomous driving applications, BEV IoU is often preferred as it focuses on the ground plane where most objects interact:

$$\text{IoU}_{BEV}(b_1, b_2) = \frac{\text{Area}(b_1^{BEV} \cap b_2^{BEV})}{\text{Area}(b_1^{BEV} \cup b_2^{BEV})}$$

where b^{BEV} represents the projection of the 3D bounding box onto the ground plane. BEV IoU is computationally simpler and often more relevant for navigation tasks.

Evaluation Metrics for 3D Detection 3D object detection uses specialized metrics that account for spatial dimensions and orientation accuracy. The standard metric is 3D Average Precision (AP) computed using 3D IoU thresholds (typically 0.5 and 0.7). For autonomous driving, evaluation often focuses on Bird's Eye View (BEV) metrics that emphasize horizontal plane accuracy, with benchmarks like KITTI providing difficulty-based analysis. Orientation accuracy is measured through angular error between predicted and ground truth orientations, with some metrics requiring joint spatial and angular tolerance for correct detections.

12.2.2 3D Detection from Point Clouds and Voxel Representations

Building effective 3D object detection systems requires leveraging the specialized architectures for 3D data processing developed in the previous chapter.

The choice between point-based and voxel-based representations fundamentally shapes the detection architecture, with each approach offering distinct advantages for different robotics applications. Point-based methods preserve the geometric precision of the original sensor data and handle irregular point distributions naturally, making them well-suited for applications requiring precise object localization. Voxel-based methods trade some geometric precision for computational efficiency by imposing regular grid structures that enable optimized convolutional operations, making them preferred for real-time robotics applications.

Rather than being mutually exclusive, these representations often complement each other within detection pipelines. Many successful 3D detection systems combine the efficiency of voxel processing for initial feature extraction with the precision of point-based refinement for final object localization. Understanding how different detection paradigms—two-stage, one-stage, and transformer-based—can be adapted to work with these 3D representations provides the foundation for building robust detection systems.

Leveraging 3D Feature Representations The 3D detection architectures we will explore build directly upon the feature extraction capabilities of PointNet, VoxelNet, and PointPillars discussed in the previous chapter. These architectures provide learned feature representations that encode geometric patterns, spatial relationships, and semantic information from 3D sensor data. The key insight is that these features can serve as input to detection heads that predict object classifications and 3D bounding box parameters.

Point-based representations excel at preserving fine-grained geometric details and handling the irregular structure of sensor data. PointNet++ features capture multi-scale geometric patterns through hierarchical set abstraction, enabling detection of objects at different scales and levels of detail. These features are particularly valuable for detecting small objects or distinguishing between closely spaced instances where geometric precision is critical. Voxel-based representations provide computational advantages through regular grid structures that enable efficient 3D convolutions and parallel processing. VoxelNet features encode local geometric patterns within voxels while maintaining spatial relationships across the scene. PointPillars features offer a hybrid approach, encoding vertical structure within pillars while enabling efficient 2D processing for large-scale scenes. The choice between these representations often depends on the computational constraints and accuracy requirements of the specific robotics application.

12.2.3 Two-Stage 3D Detection: Extending Faster R-CNN

The two-stage detection paradigm extends naturally to 3D by first generating object proposals in 3D space, then refining these proposals through dedicated classification and regression networks. This approach works particularly well

when combined with the hierarchical feature representations from PointNet++ or the structured features from VoxelNet.

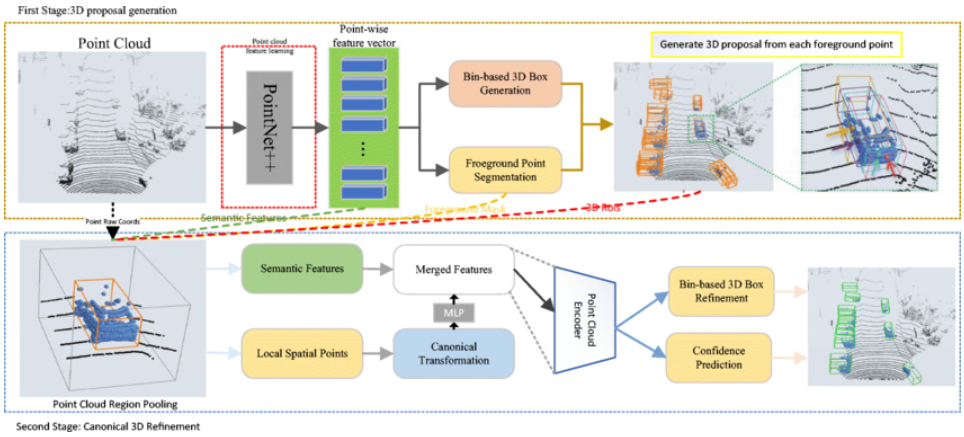


Figure 12.6: PointRCNN architecture showing bottom-up 3D proposal generation from point-wise features, followed by canonical coordinate refinement and final detection heads.

PointRCNN: Point-based Two-Stage Detection PointRCNN demonstrates how the Faster R-CNN paradigm can be adapted to work directly with point cloud data using PointNet++ features. The architecture follows a bottom-up approach where object proposals are generated directly from point-wise features rather than through dense sliding window approaches used in image detection.

The first stage leverages PointNet++ hierarchical features to perform point-wise binary classification, identifying points that likely belong to foreground objects versus background. Rather than generating proposals at regular grid locations, PointRCNN generates 3D proposals centered at high-confidence foreground points. This approach is computationally efficient because it only considers a subset of points for proposal generation, and it preserves the geometric precision of the original point cloud. For each proposal, the second stage extracts local point features within the proposed 3D region and applies canonical coordinate transformation to normalize the local geometry. This transformation aligns the object coordinate system with a canonical orientation, making the subsequent classification and regression tasks more robust to object orientation variations. The canonical transformation is particularly important for 3D detection because objects can appear in arbitrary orientations in the sensor coordinate system.

The mathematical formulation for canonical transformation involves rotating and translating the local point coordinates so that the object's principal axes align with canonical directions:

$$\mathbf{p}_{\text{canonical}} = \mathbf{R}^{-1}(\mathbf{p}_{\text{local}} - \mathbf{t})$$

where $\mathbf{R}$ is the estimated object rotation and $\mathbf{t}$ is the estimated object center.

This transformation enables the network to learn object-centric features that are invariant to the object's pose in the world coordinate system.

VoxelNet: Voxel-based Two-Stage Detection VoxelNet adapts the two-stage paradigm to work with voxel-based representations by integrating Voxel Feature Encoding with Region Proposal Network concepts. The architecture processes point clouds through VFE layers to generate voxel-wise features, then applies 3D convolutional layers to build hierarchical representations across the voxelized space. The proposal generation stage adapts the RPN concept to 3D by sliding 3D anchor templates across the feature volume. At each spatial location in the 3D feature map, the network predicts objectness scores and 3D bounding box refinements for multiple anchor templates covering different object sizes and orientations. The 3D RPN loss combines objectness classification with 3D bounding box regression:

$$L_{\text{3D-RPN}} = \frac{1}{N_{\text{cls}}} \sum_i L_{\text{cls}}(p_i, p_i^*) + \lambda \frac{1}{N_{\text{box}}} \sum_i p_i^* L_{\text{3D-box}}(b_i, b_i^*)$$

where $L_{\text{3D-box}}$ incorporates losses for all seven parameters of the 3D bounding box: center coordinates, dimensions, and orientation.

The second stage performs 3D RoI pooling to extract fixed-size features for each proposal, followed by classification and bounding box refinement. The 3D RoI pooling operation extends the 2D concept by pooling features from 3D regions of the feature volume, maintaining spatial relationships in all three dimensions.

12.2.4 One-Stage 3D Detection: Direct Prediction

One-stage 3D detection methods perform object classification and localization simultaneously, eliminating the separate proposal generation stage. These approaches are particularly well-suited for real-time robotics applications where detection latency must be minimized.

CenterPoint: Treating 3D Objects as Points CenterPoint represents objects as points in Bird's Eye View (BEV) space and performs detection through keypoint estimation, similar to 2D anchor-free methods like CenterNet. The approach builds on PointPillars pillar-based representation to efficiently process large-scale point clouds while maintaining real-time performance.

The key insight is that 3D objects can be effectively represented by their center points when projected into BEV space, particularly for autonomous driving scenarios where objects primarily move on the ground plane. CenterPoint predicts a heatmap in BEV coordinates where peaks correspond to object centers, along with regression maps that predict 3D bounding box parameters for each detected center. The detection pipeline processes point clouds through PointPillars to generate BEV feature maps, then applies 2D convolutional networks to

predict center heatmaps and regression targets:

$$\mathbf{Y}_{\text{heatmap}} = \sigma(\text{Conv}_{2D}(\mathbf{F}_{\text{BEV}}))$$
$$\mathbf{Y}_{\text{regression}} = \text{Conv}_{2D}(\mathbf{F}_{\text{BEV}})$$

where $\mathbf{F}_{\text{BEV}}$ represents the BEV feature map from PointPillars processing, and σ is the sigmoid activation for heatmap prediction. The regression targets include 3D center offsets, object dimensions, and orientation angles.

The loss function combines center point detection with regression objectives:

$$L_{\text{CenterPoint}} = L_{\text{heatmap}} + \lambda_{\text{reg}} L_{\text{regression}}$$

where L_{heatmap} uses focal loss to handle the extreme imbalance between center points and background, and $L_{\text{regression}}$ uses smooth L1 loss for the continuous regression targets. CenterPoint extends beyond basic detection by incorporating velocity estimation for tracking applications. By processing consecutive frames, the network can predict object velocities directly as part of the regression targets, enabling seamless integration with multi-object tracking systems essential for autonomous navigation.

12.2.5 Transformer-based 3D Detection

Transformer architectures have been successfully adapted to 3D detection by treating object detection as a set prediction problem, eliminating the need for hand-designed anchors and complex post-processing steps like non-maximum suppression.

3DETR: Set-to-Set Prediction in 3D 3DETR extends the DETR paradigm to 3D object detection by using transformer architectures to directly predict sets of 3D bounding boxes from point cloud or voxel features. The approach uses learnable object queries that attend to 3D scene features through cross-attention mechanisms, enabling end-to-end learning from raw 3D data to final detections. The architecture processes 3D input data through feature extraction networks (PointNet++ for point clouds or 3D CNNs for voxel grids) to generate scene feature representations. These features are then processed by a transformer encoder to build contextual representations that capture long-range dependencies across the 3D scene. The transformer decoder uses a fixed set of learnable object queries to attend to the encoded scene features and predict object detections.

Each object query learns to specialize in detecting objects with particular characteristics or in specific spatial regions. The cross-attention mechanism allows queries to gather relevant information from across the entire scene, enabling detection of partially occluded objects or objects that extend across multiple local regions. The self-attention within the decoder enables queries to coordinate with each other, reducing duplicate detections without explicit post-processing. The final prediction heads convert each object query's repre-

sentation into 3D bounding box parameters and class predictions:

$$\mathbf{b}_i = \text{MLP}_{\text{box}}(\mathbf{q}_i)$$
$$\mathbf{c}_i = \text{MLP}_{\text{class}}(\mathbf{q}_i)$$

where $\mathbf{q}_i$ is the i-th object query after transformer processing. The training uses Hungarian matching to establish optimal assignment between predicted and ground truth objects, followed by standard detection losses.

The set-based prediction eliminates the need for anchor design, anchor assignment strategies, and non-maximum suppression, simplifying the detection pipeline while achieving competitive performance. This approach is particularly attractive for complex 3D scenes where traditional anchor-based methods struggle with the high-dimensional anchor space and complex object interactions.

12.3 Semantic and Instance Segmentation

Segmentation extends object detection by providing pixel-level or point-level understanding of scenes, enabling robots to understand not just where objects are located, but precisely which pixels belong to each object or scene category. While object detection provides coarse spatial understanding through bounding boxes, segmentation offers fine-grained spatial reasoning essential for tasks like autonomous navigation on complex terrain, precise robotic manipulation, and detailed scene understanding.

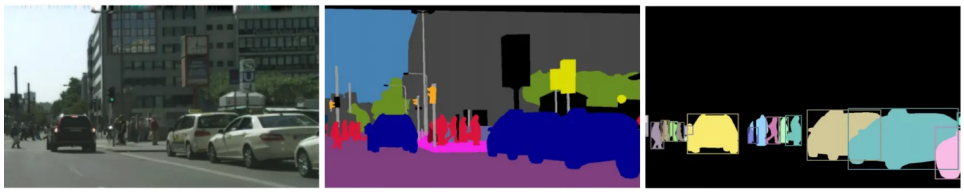


Figure 12.7: An input image and the corresponding semantic and instance segmentation.

12.3.1 Semantic Segmentation

Semantic segmentation extends image understanding beyond object detection by classifying every pixel in an image into predefined semantic categories, providing dense spatial understanding of the scene. Unlike object detection which outputs sparse bounding boxes, semantic segmentation produces pixel-level predictions that preserve the precise boundaries and spatial extent of different scene elements. This fine-grained understanding is essential for robotics applications where precise spatial reasoning is required.

Problem Definition and Robotics Applications Semantic segmentation performs pixel-level classification without distinguishing between different instances of

the same class. For an input image I of size $H \times W$, the output is a segmentation map S of the same spatial dimensions, where each pixel (i, j) is assigned a class label $S_{i,j} \in \mathcal{C}$ from the predefined set of semantic categories. For robotics systems, autonomous vehicles use segmentation to identify drivable road surfaces, distinguish between different types of terrain, and understand scene layout for path planning. Mobile robots navigating indoor environments use segmentation to identify floors, walls, furniture, and obstacles, enabling more sophisticated spatial reasoning for navigation planning. In each case, the pixel-level precision enables robots to make more informed decisions about how to interact with their environment.

Fully Convolutional Networks (FCNs) Semantic segmentation can be viewed as dense classification where standard CNN architectures are adapted to produce spatial output maps rather than single classification scores. Fully Convolutional Networks (FCNs) build on the convolutional feature extraction capabilities of CNNs while replacing the fully connected classification layers with convolutional layers that preserve spatial structure. These models replacing the fully connected layers typically used for classification with convolutional layers that can accept images of arbitrary size and produce correspondingly sized output maps. For a CNN backbone that produces feature maps of size $H/32 \times W/32$ (due to pooling operations), FCN applies 1×1 convolutions to produce class score maps, then upsamples these maps back to the original image resolution. The upsampling process uses transposed convolutions (also called deconvolutions) to increase spatial resolution:

$$y_{i,j} = \sum_{m,n} x_{\lfloor i/s \rfloor + m, \lfloor j/s \rfloor + n} \cdot w_{m,n}$$

where s is the upsampling stride and w represents the learned transposed convolution weights. This operation is the mathematical inverse of convolution with stride s, enabling learnable upsampling that can recover spatial details.

FCN introduces skip connections that combine features from different layers of the encoder to recover fine-grained spatial information lost during downsampling. These connections add feature maps from earlier layers (with higher spatial resolution) to upsampled feature maps from deeper layers (with richer semantic information):

$$F_{\text{fused}} = \text{Upsample}(F_{\text{deep}}) + F_{\text{shallow}}$$

This fusion enables the network to combine high-level semantic understanding with low-level spatial precision, crucial for accurate boundary delineation in robotics applications.

U-Net and Encoder-Decoder Architectures U-Net represents a systematic approach to encoder-decoder architectures that has become foundational for semantic segmentation across many domains. The U-Net architecture consists of

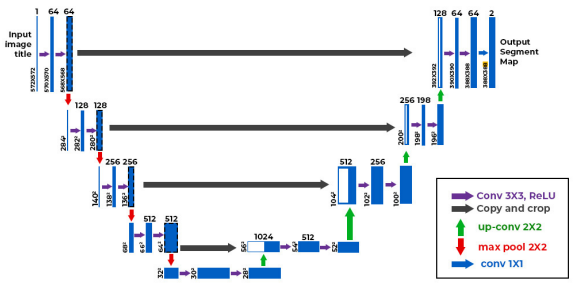


Figure 12.8: U-Net architecture showing the symmetric encoder-decoder design with skip connections at multiple scales, enabling combination of high-resolution spatial information with high-level semantic features.

a contracting path (encoder) that progressively reduces spatial resolution while increasing feature depth, followed by an expansive path (decoder) that gradually recovers spatial resolution while combining features across scales. For each decoder layer, skip connections concatenate features from corresponding encoder layer:

$$F^{(i)}_{\text{decoder}} = \text{UpConv}(F^{(i-1)}_{\text{decoder}}) \oplus F^{(i)}_{\text{encoder}}$$

where $\oplus$ denotes concatenation and UpConv represents upsampling convolution operations. These skip connections preserve fine-grained spatial details that would otherwise be lost during the encoding process.

The symmetric design ensures that the decoder has access to features at multiple scales, enabling accurate segmentation of both large objects (captured by deep, low-resolution features) and fine details (preserved through skip connections from high-resolution features). This multi-scale feature combination is particularly important for robotics applications where accurate boundary detection affects safety and task performance.

Training Loss: Classification Cross-Entropy Per Pixel Semantic segmentation networks are trained using pixel-wise classification loss, treating each pixel as an independent classification problem. The standard loss function is cross-entropy computed across all pixels:

$$L_{\text{seg}} = -\frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} \sum_{c=1}^{C} y_{i,j,c} \log(\hat{y}_{i,j,c})$$

where $y_{i,j,c}$ is the ground truth one-hot encoding for pixel (i, j) and class c, and $\hat{y}_{i,j,c}$ is the predicted probability. This formulation treats each pixel independently, enabling efficient batch processing and straightforward optimization.

However, pixel-wise cross-entropy can struggle with class imbalance, which is common in robotics scenarios where background pixels often dominate the scene. Various modifications address this challenge, including weighted cross-entropy that assigns different weights to different classes based on their frequency, and focal loss that emphasizes hard examples by down-weighting well-classified pixels.

12.3.2 *Instance Segmentation*

Instance segmentation combines object detection and semantic segmentation by identifying individual object instances and their precise pixel-level boundaries. Unlike semantic segmentation which treats all objects of the same class identically, instance segmentation distinguishes between separate instances—for example, identifying three individual cars rather than just "car pixels." This capability is critical for robotics applications where understanding individual objects enables targeted interaction and manipulation.

Problem Definition and Distinction from Semantic Segmentation Instance segmentation extends semantic segmentation by assigning unique instance identifiers to pixels belonging to distinct objects. For an input image, the output includes both semantic labels and instance masks, where each instance mask M_k defines the pixel-level extent of the k-th detected object instance. The key distinction is that semantic segmentation answers "what is this pixel?" while instance segmentation answers "what is this pixel and which specific object does it belong to?" For a robotic arm grasping objects from a bin, semantic segmentation might identify all pixels as "tool," but instance segmentation identifies individual wrenches, screwdrivers, and hammers, enabling the robot to select and grasp specific items.

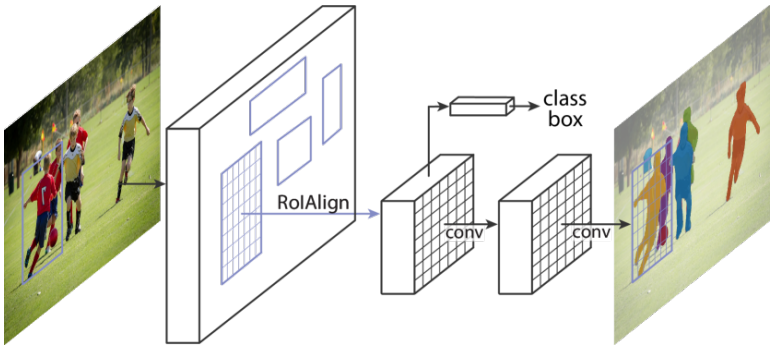


Figure 12.9: Mask R-CNN architecture showing the addition of a mask prediction branch to Faster R-CNN, with RoI Align replacing RoI pooling for improved spatial alignment.

Mask R-CNN: Extending Detection with Segmentation Mask R-CNN extends Faster R-CNN by adding a segmentation branch that predicts pixel-level masks for each detected object. The architecture maintains the two-stage paradigm: the RPN generates object proposals, and the detection head performs classification, bounding box regression, and mask prediction. The key innovation is RoI Align, which replaces RoI pooling to address spatial misalignment issues. While RoI pooling quantizes proposal coordinates to discrete feature map positions, RoI Align uses bilinear interpolation to sample features at exact locations:

$$F_{\text{aligned}}(x, y) = \sum_{i,j} I(i, j) \cdot \max(0, 1 - |x - i|) \cdot \max(0, 1 - |y - j|)$$

This precise alignment is essential for accurate mask prediction, as small spatial misalignments can significantly degrade segmentation quality.

The mask prediction branch applies a small FCN to each RoI-aligned feature to produce a binary mask for the predicted object class. The mask loss is computed only for the predicted class to avoid competition between classes:

$$L_{\text{mask}} = -\frac{1}{m^2}\sum_{i,j}[y_{i,j}\log(\hat{y}_{i,j}^{k^*}) + (1-y_{i,j})\log(1-\hat{y}_{i,j}^{k^*})]$$

where k^* is the predicted class, $y_{i,j}$ is the ground truth mask, and $\hat{y}_{i,j}^{k^*}$ is the predicted mask for class k^*.

Panoptic Segmentation Panoptic segmentation unifies semantic and instance segmentation by providing complete scene understanding. The task divides semantic categories into "things" (countable objects like cars, people) and "stuff" (amorphous regions like sky, road), performing instance segmentation for things and semantic segmentation for stuff. For robotics applications, panoptic segmentation provides comprehensive scene understanding. An autonomous vehicle can simultaneously understand the road surface (stuff), individual vehicles and pedestrians (thing instances), and background elements like buildings and vegetation (stuff), enabling holistic reasoning about the driving environment.

Bottom-Up Approaches Bottom-up instance segmentation methods first perform pixel-level feature learning, then group pixels into instances based on learned embeddings. These approaches contrast with top-down methods like Mask R-CNN that first detect objects then segment them. Associative embedding learns pixel-level features where pixels belonging to the same instance have similar embedding vectors, while pixels from different instances have dissimilar embeddings. Instance masks are then generated by clustering pixels in the embedding space:

$$d(e_i, e_j) = ||e_i - e_j||_2$$

where e_i and e_j are embedding vectors for pixels i and j. Pixels with distances below a threshold are grouped into the same instance.

These methods can handle arbitrary numbers of instances without predefined proposals but require robust clustering algorithms to separate instances reliably. They are particularly useful for robotics scenarios with dense object arrangements where proposal-based methods might struggle.

12.3.3 3D Segmentation

3D segmentation extends pixel-level understanding to volumetric data, providing precise spatial reasoning for robotics applications that require detailed 3D scene understanding. While 2D segmentation enables robots to understand image content, 3D segmentation allows reasoning about the full spatial extent and

structure of objects in the physical world. This capability is essential for manipulation tasks requiring grasp planning, navigation in complex 3D environments, and understanding object affordances based on geometric structure.

Point Cloud Segmentation Point cloud segmentation assigns semantic labels or instance identifiers to individual points in 3D space. The formulation extends 2D segmentation concepts to irregular point data, where each point $\mathbf{p}_i = (x_i, y_i, z_i)$ receives a label $l_i \in \mathcal{C}$ for semantic segmentation or instance identifier I_i for instance segmentation. Semantic segmentation of point clouds using PointNet++ leverages the hierarchical set abstraction layers from the previous chapter. The architecture processes points through multiple scales of local feature extraction and aggregation, then applies classification heads to predict semantic labels for each point:

$$l_i = \text{MLP}_{\text{seg}}(\mathbf{f}_i^{(L)})$$

where $\mathbf{f}_i^{(L)}$ represents the final point-wise feature after L layers of hierarchical processing. The multi-scale feature extraction enables accurate segmentation of objects at different sizes and levels of detail.

Instance segmentation in point clouds requires additional mechanisms to group points into distinct object instances. Methods like PointGroup combine semantic segmentation with learned offset vectors that point toward instance centers, enabling clustering of points belonging to the same object. The training loss combines semantic classification with offset regression:

$$L_{\text{point-instance}} = L_{\text{semantic}} + \lambda L_{\text{offset}} + \gamma L_{\text{clustering}}$$

where L_{offset} encourages points to predict vectors pointing toward their instance centers, and $L_{\text{clustering}}$ promotes tight clustering within instances and separation between instances.

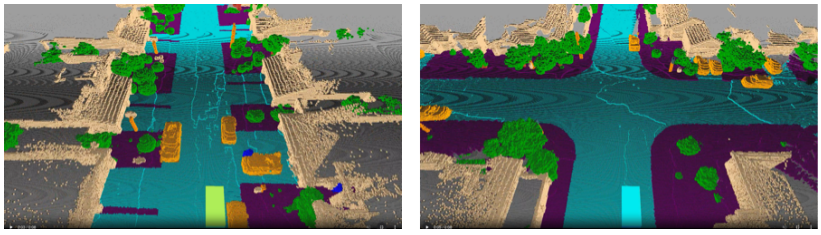


Figure 12.10: 3D voxel-based segmentation showing example from the Occ3D dataset, for self-driving scenes. This task specifies what object is in each of the 3D locations.

Voxel-based Segmentation Voxel-based segmentation processes regular 3D grids where each voxel represents a volumetric unit in 3D space. The formulation treats segmentation as 3D dense classification, where each voxel $v_{i,j,k}$ receives a semantic label or occupancy prediction. Occupancy grids represent a fundamental approach where each voxel indicates whether that region of space is

occupied by an object. This binary classification provides essential information for navigation and collision avoidance:

$$o_{i,j,k} = \sigma(\text{MLP}(\mathbf{f}_{i,j,k}))$$

where $o_{i,j,k} \in [0,1]$ represents the occupancy probability for voxel (i,j,k).

3D U-Net architectures extend the encoder-decoder paradigm to volumetric data for detailed semantic segmentation. The architecture applies 3D convolutions throughout the encoding and decoding paths, with 3D skip connections preserving spatial details:

$$\mathbf{F}^{(i)}_{\text{decoder}} = \text{UpConv3D}(\mathbf{F}^{(i-1)}_{\text{decoder}}) \oplus \mathbf{F}^{(i)}_{\text{encoder}}$$

The 3D convolutions capture volumetric patterns and spatial relationships essential for accurate 3D segmentation, while skip connections ensure fine-grained geometric details are preserved in the final predictions.

12.3.4 Robotics-Specific Applications

Segmentation enables several critical robotics capabilities that require detailed geometric understanding of objects and environments. We explore a few examples below.

Example 12.3.1. Grasp point prediction through part segmentation identifies functional regions of objects that are suitable for robotic grasping. By segmenting objects into semantic parts (handles, graspable surfaces, fragile regions), robots can plan grasps that are both mechanically sound and functionally appropriate. For example, segmenting a mug into handle, rim, and body regions enables the robot to choose appropriate grasp locations based on the intended manipulation task.

Example 12.3.2. Terrain traversability analysis uses 3D segmentation to classify different terrain types and their suitability for robot navigation. Outdoor mobile robots use segmentation to distinguish between solid ground, obstacles, vegetation, and hazardous terrain, enabling safe path planning in complex outdoor environments. The 3D understanding allows reasoning about terrain slope, roughness, and stability that would be impossible with 2D analysis alone.

Example 12.3.3. Object affordance understanding through part-based analysis enables robots to reason about how objects can be used based on their geometric structure. By segmenting objects into functional parts and understanding the spatial relationships between parts, robots can infer possible interactions and manipulation strategies. A segmented chair with identified seat, backrest, and legs enables the robot to understand both the object's function and how to manipulate it safely.

These applications demonstrate how 3D segmentation provides the detailed spatial understanding necessary for robots to interact effectively with complex 3D environments, going beyond simple object detection to enable sophisticated reasoning about object structure, function, and manipulation possibilities.

12.4 Exercise: Exploring YOLO Object Detector

The starter code for the exercises provided below is available online through GitHub. To get started, download the code by running in a terminal window:

```
git clone https://github.com/StanfordASL/pora-exercises.git
```

YOLO Model Analysis

Using the provided Jupyter notebook `yolo_exploration.ipynb`, load a pre-trained YOLOv5 model and analyze its performance on the provideds sample images. Inspect the prediction outputs at various score thresholds and measure inference timing to understand the speed advantages that make YOLO suitable for real-time robotics applications.

References

[8] Nicolas Carion et al. "End-to-End Object Detection with Transformers". In: *Computer Vision – ECCV 2020*. Springer International Publishing, 2020, pp. 213–229.

Part III

Robot Localization

13
Introduction to Localization and Filtering

We have already discussed the robot motion planning problem and surveyed common algorithms for it, ranging from optimal control to sampling-based methods. All of these approaches implicitly assume access to the robot's current state, for initializing trajectory optimization or for closing the loop in feedback control. In practice, however, this state cannot be read directly; it must be *estimate* from noisy, partial sensor data.

Robot perception, as introduced in previous chapters, tackles the challenge of extracting semantic and geometric information from raw sensor streams. Such methods are indispensable for local, instantaneous awareness. For instance, detecting nearby obstacles with a laser scanner or identifying objects in view with a camera. Yet this information is inherently *local* and relative to the robot's current position. It suffices for collision avoidance, but not for the global reasoning required by full planning and control schemes.

This gap is addressed by *robot localization and mapping*, one of the core components of the "think" stage in the classical "see-think-act" cycle. The goal is to synthesize local sensor data into a coherent *global estimate* of the robot's state, and, in mapping, of the surrounding environment. In this chapter, we focus on *localization*: the ability to infer the robot's current state with respect to a global frame or map [1]. For instance, before a robot can navigate to a target room on the floor plan shown in Figure 13.1, it must first establish where in the building it is located.

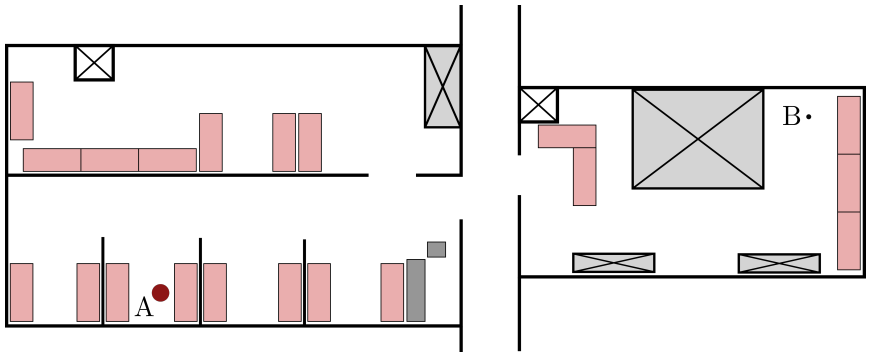


Figure 13.1: Localization is crucial for autonomy: to move from A to B, the robot must know which room it occupies, and that the only path to B runs through the hallway. Inferring such global information from local range measurements requires specialized algorithms.

[1] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005, R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, 2011

A central challenge in localization is *uncertainty*. Sensor data is noisy, incomplete, and sometimes ambiguous. To handle this, localization is cast in a *probabilistic* framework: instead of maintaining a single guess of the robot's state, we maintain a *belief distribution* over possible states. This representation allows both point estimates and measures of uncertainty to be extracted. Such uncertainty quantification is vital for downstream tasks: a planner may avoid high-risk trajectories under localization uncertainty, or even select actions that deliberately reduce uncertainty through information gathering.

The rest of this chapter introduces the probabilistic foundations underlying localization and filtering: random variables, probabilistic distributions, conditional probabilities, Bayes' rule, and Markov models. We then present the canonical recursive estimator, the Bayes filter, which formalizes how ot maintain and update a belief distribution as controls are applied and new measurements arrive.

13.1 Preliminary Concepts in Probability

To reason systematically about uncertainty in robotics, we rely on basic tools from probability theory. These tools provide the language to model noisy sensor measurements, uncertain robot states, and stochastic environments. In this section we review the key building blocks:random variables, probability distributions, conditional probabilities, and Bayes' rule—that form the foundation for probabilistic filtering algorithms such as the Bayes filter.

13.1.1 Random Variables

Uncertain quantities in robotics, such as sensor measurements, robot states, or environmental properties—are modeled as *random variables*. Depending on the domain of possible values, random variables are classified as discrete or continuous.

Definition 13.1.1 (Discrete Random Variable)**.** A *discrete random variable* X takes values from a countable set. The probability that X takes on a specific value x is denoted by $p(X = x)$, or more compactly $p(x)$. The function $p(x)$ is called the *probability mass function* (PMF), and it must satisfy

$$\sum_x p(x) = 1,$$

where the sum is over all possible values of X.

Definition 13.1.2 (Continuous Random Variable)**.** A *continuous random variable* X takes values in an uncountable set, typically a subset of $\mathbb{R}^n$. Its distribution is characterized by a *probability density function* (PDF) $p(x)$,[2] which satisfies

$$\int_{-\infty}^{\infty} p(x)\, dx = 1.$$

[2] For continuous variables the probability of any single value is zero; only intervals have nonzero probability mass.

The probability that X lies in the interval $[a, b]$ is

$$P(a \leq X \leq b) = \int_a^b p(x)\, dx.$$

Example 13.1.1 (Discrete vs. Continuous Random Variables)**.** A coin flip is a discrete random variable: $X \in \{\text{heads}, \text{tails}\}$, with PMF

$$p(\text{heads}) = \tfrac{1}{2}, \qquad p(\text{tails}) = \tfrac{1}{2}.$$

In robotics, the robot's pose $\boldsymbol{x} \in SE(2)$ or $SE(3)$ is typically modeled as a continuous random variable, since it can take on infinitely many values.

13.1.2 Probability Distributions

We will often refer to the probability mass function for discrete random variables and probability density function for continuous random variables as simply *probability distributions*. There are many ways to parameterize a probability distribution, such as a discrete set of probability masses or as a continuous function defined by some number of parameters. One of the most common continuous probability distributions is the Gaussian distribution[3], which is parameterized by a mean and variance.

[3] The Gaussian distribution is also known as the Normal distribution.

Some probability distribution representations are more expressive than others, but there is usually a trade-off with computational complexity of the algorithms that use the representation.

13.1.3 Joint Distributions, Independence, and Conditioning

Many robotics problems involve more than one uncertain quantity at a time. For example, a robot might simultaneously reason about its pose, the position of an obstacle, and a sensor reading. In such cases, it is useful to describe the probabilities of multiple random variables together using a *joint distribution*.

Definition 13.1.3 (Joint Distribution)**.** The *joint distribution* of two random variables X and Y specifies the probability that both take on specific values simultaneously. It is denoted by $p(X = x, Y = y)$, or more compactly $p(x, y)$.

Independence. Random variables can be related to each other in important ways. For example, the random variables X = "today is cloudy" and Y = "today it is raining" are correlated: if there are no clouds, it is unlikely to rain. By contrast, two random variables are *probabilistically independent* if the value of one does not provide any information about the other.

Definition 13.1.4 (Probabilistic Independence)**.** Two random variables X and Y are probabilistically *independent* if and only if

$$p(x, y) = p(x)\, p(y). \tag{13.1}$$

Example 13.1.2 (Independent Sensor Measurements)**.** Suppose a robot uses a proximity sensor and a temperature sensor, modeled by random variables X and Y. Let $X \in \{\text{close}, \text{medium}, \text{far}\}$ and $Y \in \{\text{low}, \text{med}, \text{high}\}$, with $p(X = \text{close}) = 1/3$ and $p(Y = \text{med}) = 1/5$. If the two sensors are independent, the joint probability of observing "close" and "med" is

$$p(X = \text{close}, Y = \text{med}) = \tfrac{1}{3} \cdot \tfrac{1}{5} = \tfrac{1}{15}.$$

Conditional probability. Another key concept is the probability of one random variable given that another has already been observed.

Definition 13.1.5 (Conditional Probability)**.** The *conditional probability* of a random variable X taking value x, given that Y took value y, is

$$p(x \mid y) := \frac{p(x, y)}{p(y)}. \tag{13.2}$$

Conditional probabilities allow us to update beliefs when new information becomes available. If X and Y are independent, then $p(x \mid y) = p(x)$: knowing Y provides no additional information about X.

Example 13.1.3 (Sensor Conditional Probabilities)**.** Building on Example 13.1.2, consider an obstacle detection variable $Z \in \{\text{detected}, \text{not detected}\}$. Assume that the detection probability depends on the proximity sensor value:

$$p(Z = \text{detected} \mid X = \text{close}) = \tfrac{5}{6}, \quad p(Z = \text{detected} \mid X = \text{medium}) = \tfrac{1}{3}, \quad p(Z = \text{detected} \mid X = \text{far}) = \tfrac{1}{5}.$$

If $p(X = \text{close}) = 1/3$, then the probability that the robot both detects an obstacle and registers "close" is

$$p(Z = \text{detected}, X = \text{close}) = p(Z = \text{detected} \mid X = \text{close})\, p(X = \text{close}) = \tfrac{5}{18}.$$

Conditional independence. Finally, independence can also hold *given* the outcome of another variable. This arises frequently in robotics when different sensor readings become independent once the true state is specified.

Definition 13.1.6 (Conditional Independence)**.** Two random variables X and Y are *conditionally independent*[4] given a third random variable Z if and only if

$$p(x, y \mid z) = p(x \mid z)\, p(y \mid z). \tag{13.3}$$

[4] Note that conditional independence does not imply unconditional independence, and vice versa.

Example 13.1.4 (Conditional Independence in Robotics)**.** A mobile robot equipped with two wheel encoders produces measurements of traveled distance: one from the left wheel (X) and one from the right wheel (Y). At first glance, these two measurements may seem correlated, since the robot's motion affects both. However, if we condition on the underlying hidden variable Z = "true distance traveled," the two encoder readings are independent:

$$p(x, y \mid z) = p(x \mid z)\, p(y \mid z).$$

That is, once the actual distance traveled is fixed, the left and right encoders provide independent noisy measurements of that same quantity. This is a typical use of conditional independence in probabilistic sensor models.

13.1.4 Law of Total Probability

The *law of total probability* links marginal, joint, and conditional probabilities. It provides a systematic way to compute the probability of one random variable by accounting for all possible outcomes of another.

Definition 13.1.7 (Law of Total Probability)**.** For discrete random variables X and Y:

$$p(x) = \sum_y p(x, y) = \sum_y p(x \mid y)\, p(y).$$

For continuous random variables:

$$p(x) = \int p(x, y)\, dy = \int p(x \mid y)\, p(y)\, dy.$$

In words: the probability of $X = x$ is obtained by summing (or integrating) over all possible values of Y, weighting the conditional probability $p(x \mid y)$ by how likely each y is.

This process is known as *marginalization*, and $p(x)$ is called the *marginal probability* of X.

Example 13.1.5 (Robot Localization via Marginalization)**.** Suppose a robot's position X depends on which hallway Y it is currently in. The probability of being at a particular location x can be computed by considering every possible hallway y:

$$p(x) = \sum_y p(x \mid y)\, p(y).$$

In practice, this means we marginalize over the possible hallways, combining both the likelihood of being in each hallway and the probability of observing x given that hallway.

13.1.5 Bayes' Rule

The joint probability, $p(x, y)$, between two random variables, X and Y, is related to the conditional probabilities, $p(x \mid y)$ and $p(y \mid x)$, from the definition of a conditional probability in Equation (13.2). Since we can express the joint probability using either conditional probability, we have:

$$p(x, y) = p(x \mid y)p(y) = p(y \mid x)p(x).$$

This relationship is commonly referred to as *Bayes' rule*[5]:

[5] Sometimes also referred to as *Bayes' theorem*.

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 because 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* problems where we need to infer the value of one random

variable from another. For example, suppose we have a good initial guess of the probability distribution[6], $p(x)$, for a random variable, X. Given new information about the outcome of a second random variable, Y, that is related to X, we can use Bayes' rule to update our belief about the probability distribution of X by computing $p(x \mid y)$[7]. Bayes' rule also extends to cases with additional random variables. For example, with three random variables, X, Y, and Z, Bayes' rule is:

$$p(x \mid y, z) = \frac{p(y \mid x, z)p(x \mid z)}{p(y \mid z)}.$$

[6] When we have an estimate of the probability distribution $p(x)$ before any new information is used to update it, we will refer to it as the *prior* probability.

[7] This new distribution, which we obtained by updating the prior distribution $p(x)$ with the new information about Y, is commonly referred to as the *posterior* probability.

Example 13.1.6 (Bayes' Rule)**.** Consider a scenario where a robot is trying to figure out if it is in room A or room B inside of a building. The robot has an initial guess that the probability it is in room A is $p(A) = \frac{3}{4}$, and the robot has a camera that can be used to improve the estimate. Suppose that a single image, I, is captured and the features extracted from the image are compared to the known room features which gives the conditional probabilities:

$$p(I \mid A) = \frac{3}{4}, \quad p(I \mid B) = \frac{1}{2}.$$

We can use Bayes' rule to compute the posterior probability:

$$p(A \mid I) = \frac{p(I \mid A)p(A)}{p(I)},$$

where we use the law of total probability to compute:

$$p(I) = p(I, A) + p(I, B) = p(I \mid A)p(A) + p(I \mid B)p(B),$$

and using $p(B) = 1 - p(A)$.

13.1.6 Expectation and Covariance

Probability distributions describe uncertainty in full detail by assigning probabilities to every possible outcome of a random variable. In practice, however, we often summarize a distribution using more compact statistics. Two of the most commonly used summaries are the *expected value* and the *covariance*.

Definition 13.1.9 (Expected Value)**.** The *expected value*[8] of a random variable X is denoted by $\mathbb{E}[X]$. For discrete random variables:

$$\mathbb{E}[X] = \sum_x x\, p(x),$$

[8] Also referred to as the *mean* or the *first moment* of a distribution.

where the sum is over all outcomes of X. For continuous random variables:

$$\mathbb{E}[X] = \int x\, p(x)\, dx.$$

The expectation can be interpreted as the long-run average outcome over infinitely many samples. It also has a useful property of *linearity*:

$$\mathbb{E}[aX + b] = a\mathbb{E}[X] + b,$$

for any $a, b \in \mathbb{R}$. For vector-valued random variables, $\mathbb{E}[X]$ is simply the vector of expectations of each component.

Definition 13.1.10 (Covariance)**.** The *covariance* between two random variables X and Y is denoted $\text{cov}(X, Y)$ and defined as

$$\begin{aligned}\text{cov}(X, Y) &= \mathbb{E}\left[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])^\top\right] \\ &= \mathbb{E}\left[XY^\top\right] - \mathbb{E}[X]\ \mathbb{E}[Y]^\top.\end{aligned}$$

Covariance describes how two random variables vary together. $\text{cov}(X, Y) > 0$: large values of X tend to coincide with large values of Y, and small with small. $\text{cov}(X, Y) < 0$: large values of X tend to coincide with small values of Y (and vice versa). $\text{cov}(X, Y) = 0$: the variables show no linear relationship (e.g., independent variables).

Example 13.1.7 (Robot Motion Uncertainty)**.** Suppose X is the forward displacement of a robot and Y is its lateral displacement during one motion step. If the wheels slip more when the robot moves farther forward, then X and Y will have a positive covariance. If forward motion tends to reduce sideways drift, the covariance will be negative. If the two types of motion noise are unrelated, the covariance will be close to zero.

13.2 Markov Models

In Chapter 1, we modeled robot motion using kinematics and dynamics, obtaining a set of first-order differential equations (see Equation (1.1)) that deterministically describe how the state $\boldsymbol{x}$ evolves in time given the current state and control input $\boldsymbol{u}$. In this section, we generalize this view to a *probabilistic* setting by introducing *Markov models*, which describe how the state evolves under uncertainty. Markov models are fundamental to robotics, appearing in localization, mapping, planning, and decision-making under uncertainty.

State, controls, and measurements. As in Chapter 1, the state $\boldsymbol{x} \in \mathbb{R}^n$ collects all variables relevant to the task at hand. In motion planning and control, this typically includes the robot's physical state (pose, velocity, etc.), while in localization or higher-level planning it may also include environment variables such as landmark positions or object features. We work in discrete time, writing $\boldsymbol{x}_t$ for the state at time t. We also use the shorthand $\boldsymbol{x}_{t_1:t_n} := \boldsymbol{x}_{t_1}, \boldsymbol{x}_{t_2}, \ldots, \boldsymbol{x}_{t_n}$ for sequences of states, with analogous notation for control inputs $\boldsymbol{u}_{t_1:t_n}$ and measurements $\boldsymbol{z}_{t_1:t_n}$.[9]

[9] Measurements can come from any of the sensors introduced earlier, such as cameras, lidar, or inertial units.

Unlike deterministic dynamics, Markov models specify probability distributions over possible states and observations. In full generality, the state evolution is modeled as

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{0:t-1}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}), \tag{13.5}$$

the distribution of the current state $\boldsymbol{x}_t$ conditioned on the entire history of past states, controls, and measurements. Following the convention used throughout this chapter, the robot first executes the control $\boldsymbol{u}_t$, then receives the measurement $\boldsymbol{z}_t$ based on the resulting state $\boldsymbol{x}_t$. The corresponding probabilistic

measurement model is

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_{0:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}). \tag{13.6}$$

The Markov property. In many applications, we define the state $\boldsymbol{x}_t$ to be *complete*, meaning it contains all the information necessary to predict future states. Formally, this assumption implies that past states and measurements provide no additional predictive power beyond $\boldsymbol{x}_{t-1}$ and $\boldsymbol{u}_t$. This is known as the *Markov property*, under which the models simplify to

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t), \tag{13.7}$$

for the state transition, and

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_t), \tag{13.8}$$

for the measurement model.

Markov models in robotics A *Markov model* thus consists of a state transition distribution (13.7) and a measurement distribution (13.8). Intuitively, the transition model captures process uncertainty (e.g., wheel slip when applying a control), while the measurement model captures sensor noise (e.g., rangefinder variability). Together, these models form the backbone of probabilistic state estimation (Figure 13.2).

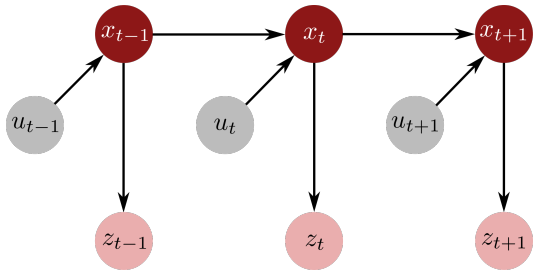


Figure 13.2: Graphical representation of a Markov model. At each time step, the control $\boldsymbol{u}_t$ influences the new state $\boldsymbol{x}_t$, and the resulting state generates a measurement $\boldsymbol{z}_t$.

Markov models are sometimes referred to as *partially observable Markov models*, since the state is not necessarily fully observable from measurements. If there are no controls, i.e., no $\boldsymbol{u}_t$, the model reduces to a *hidden Markov model (HMM)*, where the term "hidden" reflects that the underlying state $\boldsymbol{x}_t$ cannot be observed directly.

13.3 Bayes Filter

Robot localization is a classic instance of a filtering problem[10] where our goal is to compute a probability distribution over the current state, $\boldsymbol{x}_t$, given the history of control inputs, $\boldsymbol{u}_{1:t}$, and measurements, $\boldsymbol{z}_{1:t}$. One of the canonical approaches to this filtering problem is known as the *Bayes filter* or *recursive Bayesian estimation*. The Bayes filter leverages a Markov model to recursively update a *belief*

[10] Localization is one instance of the more general filtering problem referred to as *state estimation*.

distribution, which is a probability distribution over $\boldsymbol{x}_t$. Mathematically, we denote the belief distribution as $\text{bel}(\boldsymbol{x}_t)$ and define it as:

$$\text{bel}(\boldsymbol{x}_t) \coloneqq p(\boldsymbol{x}_t \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}). \tag{13.9}$$

In other words, the belief, $\text{bel}(\boldsymbol{x}_t)$, is a posterior probability distribution over the state conditioned on the available history information. We also define a distribution called the *prediction* distribution as:

$$\overline{\text{bel}}(\boldsymbol{x}_t) \coloneqq p(\boldsymbol{x}_t \mid \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}), \tag{13.10}$$

which does not include the most recent measurement, $\boldsymbol{z}_t$. We call the process of using the new measurement, $\boldsymbol{z}_t$, to compute the belief, $\text{bel}(\boldsymbol{x}_t)$, from the predicted belief, $\overline{\text{bel}}(\boldsymbol{x}_t)$, a *correction* or *measurement update*. The Bayes filter consists of a prediction step for computing $\overline{\text{bel}}(\boldsymbol{x}_t)$ from the prior belief followed by a correction step for computing $\text{bel}(\boldsymbol{x}_t)$ given the new measurement, $\boldsymbol{z}_t$.

13.3.1 Algorithm

The recursive structure of the Bayes filter is summarized in Algorithm 13.1. The inputs for each step are the belief from the previous state[11] and the current control input and measurement. For each state, $\boldsymbol{x}_t$, this algorithms performs a *prediction* step to compute $\overline{\text{bel}}(\boldsymbol{x}_t)$ and then a *correction* step based on the measurement, $\boldsymbol{z}_t$. The prediction step is essentially just using the state transition model from Equation (13.7) to predict what might happen to each state for the given control, $\boldsymbol{u}_t$. The correction step then modifies the prediction to account for the measurement that was actually observed in the real world.

[11] In practice, we typically initialize the prior distribution, $\text{bel}(\boldsymbol{x}_0)$, using a best guess or simply a uniform distribution.

Algorithm 13.1: Bayes Filter

Data: $\text{bel}(\boldsymbol{x}_{t-1}), \boldsymbol{u}_t, \boldsymbol{z}_t$
Result: $\text{bel}(\boldsymbol{x}_t)$
foreach $\boldsymbol{x}_t$ **do**
  `// Prediction (motion update)`
  $\overline{\text{bel}}(\boldsymbol{x}_t) = \int p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)\, \text{bel}(\boldsymbol{x}_{t-1})\, d\boldsymbol{x}_{t-1}$
  `// Correction (measurement update)`
  $\text{bel}(\boldsymbol{x}_t) = \eta\, p(\boldsymbol{z}_t \mid \boldsymbol{x}_t)\, \overline{\text{bel}}(\boldsymbol{x}_t)$
return $\text{bel}(\boldsymbol{x}_t)$

Here η is a normalization constant ensuring $\text{bel}(\boldsymbol{x}_t)$ integrates (or sums) to one.[12] Conceptually, the filter performs a *predict-correct* cycle: it forecasts possible states using the motion model, then sharpens or shifts this forecast based on the new sensor reading.

[12] In practice, $\eta = 1/p(\boldsymbol{z}_t \mid \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t})$ comes directly from Bayes' rule.

13.3.2 Derivation

We begin deriving the Bayes filter by expanding the definition of the belief distribution from Equation (13.9) using Bayes' rule:

$$\begin{aligned}\text{bel}(\boldsymbol{x}_t) &\coloneqq p(\boldsymbol{x}_t \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}) \\ &= \eta p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) p(\boldsymbol{x}_t \mid \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}),\end{aligned}$$

where η is the normalization constant:

$$\eta = \frac{1}{p(\boldsymbol{z}_t \mid \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t})}.$$

We then leverage the Markov assumption to simplify $p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) = p(\boldsymbol{z}_t \mid \boldsymbol{x}_t)$ and, combined with the definition of the prediction belief, we have:

$$\text{bel}(\boldsymbol{x}_t) = \eta p(\boldsymbol{z}_t \mid \boldsymbol{x}_t) \overline{\text{bel}}(\boldsymbol{x}_t),$$

which is the measurement update step of the Bayes filter algorithm. For the prediction step, we start from the definition of the prediction belief and leverage the law of total probability to marginalize out the previous state, $\boldsymbol{x}_{t-1}$:

$$\begin{aligned}\overline{\text{bel}}(\boldsymbol{x}_t) &\coloneqq p(\boldsymbol{x}_t \mid \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}), \\ &= \int p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) p(\boldsymbol{x}_{t-1} \mid \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) d\boldsymbol{x}_{t-1}.\end{aligned}$$

Using the Markov assumption again, we can 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 from the structure of the Markov model in Figure 13.2, the control, $\boldsymbol{u}_t$, does not have any influence on the previous state, $\boldsymbol{x}_{t-1}$, so we can also simplify the prior distribution $p(\boldsymbol{x}_{t-1} \mid \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) = p(\boldsymbol{x}_{t-1} \mid \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t-1})$. Using these simplifications, we can compute the prediction belief as:

$$\overline{\text{bel}}(\boldsymbol{x}_t) = \int p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) \text{bel}(\boldsymbol{x}_{t-1}) d\boldsymbol{x}_{t-1},$$

since by definition $\text{bel}(\boldsymbol{x}_{t-1}) = p(\boldsymbol{x}_{t-1} \mid \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t-1})$.

13.3.3 Discrete Bayes Filter

When the state space is finite, we represent the belief as a probability mass function over states $\{x_k\}$. Let $p_{k,t}$ denote the probability of being in state x_k at time t.

We can apply a discrete version of the Bayes filter to problems where the state space is finite and we represent the belief, $\text{bel}(\boldsymbol{x}_t)$, by a probability mass function rather than a probability density function. The probability mass function that defines the belief is a finite collection of probabilities, $\{p_{k,t}\}$, where $p_{k,t}$ is the probability associated with state k at timestep t. We define the discrete Bayes filter algorithm in Algorithm 13.2. Note that it follows the same procedure as the continuous Bayes filter in Algorithm 13.1, but with summations replacing the integrals.

Algorithm 13.2: Discrete Bayes Filter

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 $p_{k,t}$

13.3.4 Practical Considerations

The Bayes filter provides a general, principled foundation for state estimation, but direct application can be challenging:

- **Continuous state spaces:** the integrals in the prediction step are often intractable.
- **Discrete state spaces:** the summations are exact but quickly become computationally expensive as the number of states grows.

Despite these challenges, the Bayes filter serves as the starting point for practical algorithms. Specialized filters such as the Kalman filter, extended Kalman filter (EKF), unscented Kalman filter (UKF), and particle filter all derive from this framework by making approximations or exploiting structure.

Example 13.3.1 (1D Robot on a Line)**.** Consider a robot moving along a straight hallway represented by a one-dimensional line. The hidden state $\boldsymbol{x}_t$ is the robot's position along this line. At each time step t the robot receives:

- a *control input* $\boldsymbol{u}_t$, representing a commanded forward displacement, and
- a *measurement* $\boldsymbol{z}_t$, representing the noisy distance to the nearest wall detected by a range sensor.

Prediction. Suppose the robot starts at $\boldsymbol{x}_{t-1} = 2\,\text{m}$ with a belief concentrated around that position. It issues a command $\boldsymbol{u}_t = +1\,\text{m}$ forward. Because of wheel slip and actuator noise, the actual displacement may vary, e.g., $\Delta x \sim \mathcal{N}(1, 0.1^2)$. The prediction step therefore spreads the belief forward, producing $\overline{\text{bel}}(\boldsymbol{x}_t)$ centered at 3 m but with increased variance.

Correction. At the same step, the robot's range sensor reports $\boldsymbol{z}_t = 2.9\,\text{m}$ to the wall. The sensor is noisy, modeled by $p(\boldsymbol{z}_t \mid \boldsymbol{x}_t) = \mathcal{N}(\boldsymbol{z}_t; \text{true distance}(\boldsymbol{x}_t), 0.05^2)$. If the predicted belief places probability mass around $\boldsymbol{x}_t = 3\,\text{m}$, this measurement is highly consistent and the correction step sharpens the belief around 3 m. If instead the prediction had significant mass at $\boldsymbol{x}_t = 5\,\text{m}$, the measurement likelihood would downweight those hypotheses.

Recursive operation. Over time, the filter alternates prediction (broadening due to motion uncertainty) and correction (sharpening when distinctive sensor information arrives). If the hallway contains identical-looking sections, the belief may remain multimodal until the robot observes a unique feature (e.g., the hallway's end), at which point the Bayes filter collapses the ambiguity and localizes the robot with high confidence.

References

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

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

14
Approximate Filters for State Estimation

In Chapter 13 we introduced the Bayes filter as the canonical framework for state estimation: a recursive procedure that alternates between *prediction*, using a probabilistic state transition model, and *correction*, using a probabilistic measurement model. While conceptually elegant, the Bayes filter is rarely tractable to implement in its full generality. The integrals in the prediction step and the normalization in the correction step can be computationally intractable for continuous, high-dimensional systems, and exact enumeration is impossible except in very small discrete domains.

To make the Bayes filter practically useful, we therefore need to approximate the belief distribution. Over the years, two broad families of approximations have emerged:

- **Parametric filters:** These assume that the belief distribution belongs to a specific parametric family (most commonly Gaussian), characterized by a fixed set of parameters such as mean and covariance [1]. Exploiting the structure of the chosen family allows for efficient recursive updates. The Kalman filter and its variants (e.g., EKF, UKF) are prime examples.
- **Non-parametric filters:** These make no strong assumptions on the form of the belief distribution. Instead, they approximate it directly, either by discretization (histogram filters) or by sampling (particle filters). This flexibility allows non-parametric filters to represent multimodal and highly nonlinear beliefs, at the expense of higher computational cost.

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

Viewed together, parametric and non-parametric filters represent two ends of a spectrum. Parametric filters trade representational flexibility for computational efficiency, while non-parametric filters trade efficiency for expressiveness. In robotics practice, both families play a critical role: parametric filters often suffice when the problem structure is close to Gaussian and unimodal, while non-parametric filters are indispensable when ambiguity, multimodality, or strong nonlinearities are present.

In the remainder of this chapter, we develop both approaches within a unified narrative. We begin with parametric filters, introducing the *Kalman filter*

family and highlighting how Gaussian assumptions simplify the Bayes filter equations. We then turn to non-parametric filters, where we relax these assumptions and represent beliefs more directly through discretization or sampling. By presenting them side-by-side, we will emphasize their shared foundation in the Bayes filter and their complementary strengths for real-world state estimation.

14.1 The Gaussian Distribution

The Gaussian (or *normal*) distribution is one of the most widely used probability distributions in science and engineering, and plays a central role in robotics state estimation. Its importance arises not only from its frequent appearance in natural noise processes, but also from its favorable mathematical properties that make recursive filtering tractable.

Univariate case. The probability density function (PDF) of a one-dimensional[2] Gaussian random variable X with mean μ and variance σ^2 is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\tfrac{1}{2}\tfrac{(x-\mu)^2}{\sigma^2}\right). \tag{14.1}$$

[2] We refer to a one-dimensional Gaussian as *univariate* and to higher-dimensional cases as *multivariate*.

We write this compactly as $X \sim \mathcal{N}(\mu, \sigma^2)$.

Multivariate case. For an n-dimensional random vector $\boldsymbol{x} \in \mathbb{R}^n$ with mean $\boldsymbol{\mu} \in \mathbb{R}^n$ and covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{n\times n}$, the multivariate Gaussian distribution is defined by

$$p(\boldsymbol{x}) = \frac{1}{\sqrt{\det(2\pi\boldsymbol{\Sigma})}} \exp\left(-\tfrac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right), \tag{14.2}$$

denoted $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. The covariance $\boldsymbol{\Sigma}$ captures both the spread of each component of $\boldsymbol{x}$ and their pairwise correlations.

The Gaussian distribution exhibits several important mathematical properties related to affine transformations, addition, and multiplication, which make it particularly attractive for use in filtering algorithms.

Affine transformations: The first useful property of the Gaussian distribution is that an affine transformation of a Gaussian random variable is also a Gaussian random variable. If the random vector X has a multivariate Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$, then the random variable Y computed from an affine transformation:

$$Y = AX + b,$$

also has a multivariate Gaussian distribution with mean $A\boldsymbol{\mu} + b$ and covariance $A\boldsymbol{\Sigma}A^\top$. In other words, if $X \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, then $Y \sim \mathcal{N}(A\boldsymbol{\mu} + b, A\boldsymbol{\Sigma}A^\top)$.

Sum: The next useful property of Gaussians is that the sum of two independent Gaussian random variables is also a Gaussian random variable. 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 computed by the sum:

$$Y = X_1 + X_2,$$

also has a multivariate Gaussian distribution with mean $\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2$ 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 $Y \sim \mathcal{N}(\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2)$.

Product: The product of two Gaussian probability density functions is also a Gaussian probability density function. Consider two Gaussian probability density functions:

$$p_1(x) = \frac{1}{\sqrt{\det(2\pi\boldsymbol{\Sigma}_1)}} \exp\big(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_1)^\top \boldsymbol{\Sigma}_1^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_1)\big)$$
$$p_2(x) = \frac{1}{\sqrt{\det(2\pi\boldsymbol{\Sigma}_2)}} \exp\big(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_2)^\top \boldsymbol{\Sigma}_2^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_2)\big).$$

Their product is:

$$\begin{aligned} p(x) &= p_1(x) \cdot p_2(x) \\ &= \frac{\exp\big(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_1)^\top \boldsymbol{\Sigma}_1^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_1) - \frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_2)^\top \boldsymbol{\Sigma}_2^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_2)\big)}{2\pi^d \sqrt{\det(\boldsymbol{\Sigma}_1)}\sqrt{\det(\boldsymbol{\Sigma}_2)}} \\ &= \frac{\exp\big(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu})\big)\exp\big(-\frac{1}{2}(\boldsymbol{\mu}_1^\top \boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2^\top \boldsymbol{\Sigma}_2^{-1}\boldsymbol{\mu}_2 - \boldsymbol{\mu}^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu})\big)}{2\pi^d \sqrt{\det(\boldsymbol{\Sigma}_1)}\sqrt{\det(\boldsymbol{\Sigma}_2)}}, \end{aligned}$$

where d is the dimension of the covariance matrices, and we can see that the second exponential is constant with respect to $\boldsymbol{x}$. Therefore, the product is a Gaussian probability density function with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$:

$$\boldsymbol{\Sigma} = (\boldsymbol{\Sigma}_1^{-1} + \boldsymbol{\Sigma}_2^{-1})^{-1},$$
$$\boldsymbol{\mu} = \boldsymbol{\Sigma}(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_2^{-1}\boldsymbol{\mu}_2).$$

Why Gaussians in filtering? These properties ensure that when both the transition and measurement models are linear with Gaussian noise, the Bayes filter reduces to simple recursive updates of the mean and covariance. This leads directly to the family of *Kalman filters*, which we will introduce in the next section.

14.2 Kalman Filter

The Kalman filter is the canonical *parametric* realization of the Bayes filter for linear-Gaussian models. Unlike the discrete Bayes filter from Chapter 13, we can efficiently apply this filter to problems with *continuous* states. Specifically, the Kalman filter uses a multivariate Gaussian distribution 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 therefore:

$$\text{bel}(\boldsymbol{x}_t) = \frac{1}{\sqrt{\det(2\pi\boldsymbol{\Sigma}_t)}}\exp\big(-\frac{1}{2}(\boldsymbol{x}_t - \boldsymbol{\mu}_t)^\top \boldsymbol{\Sigma}_t^{-1}(\boldsymbol{x}_t - \boldsymbol{\mu}_t)\big).$$

Like the Bayes filter, the Kalman filter is split up into two steps: a prediction step and measurement update step. Both of these steps update the mean, $\boldsymbol{\mu}$, and covariance, $\boldsymbol{\Sigma}$, parameters, which requires us to make several assumptions about the problem setup. First, we must assume that the initial belief, $\text{bel}(\boldsymbol{x}_0)$, is Gaussian with $\boldsymbol{x}_0 \sim \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$. We also assume that the state transition model is linear and of the form:

$$\boldsymbol{x}_t = A_t\boldsymbol{x}_{t-1} + B_t\boldsymbol{u}_t + \boldsymbol{\epsilon}_t, \tag{14.3}$$

where $\boldsymbol{x}_{t-1}$ is the previous state, $\boldsymbol{u}_t$ is the most recent control input, $\boldsymbol{\epsilon}_t$ is an independent *process noise* that is normally distributed according to $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{R}_t)$, and A_t and B_t are time varying matrices that define the dynamics. Due to the mathematical properties discussed earlier, this affine transition model preserves the structure of the Gaussian distribution such that if $\boldsymbol{x}_{t-1}$ is normally distributed, then so is $\boldsymbol{x}_t$. Specifically, the probabilistic state transition model based on Equation (14.3) is:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) = \frac{1}{\sqrt{\det(2\pi\boldsymbol{R}_t)}}\exp\big(-\frac{1}{2}(\boldsymbol{x}_t - A_t\boldsymbol{x}_{t-1} - B_t\boldsymbol{u}_t)^\top \boldsymbol{R}_t^{-1}(\boldsymbol{x}_t - A_t\boldsymbol{x}_{t-1} - B_t\boldsymbol{u}_t)\big).$$

and therefore the next state is normally distributed with:

$$\boldsymbol{x}_t \sim \mathcal{N}(A_t\boldsymbol{x}_{t-1} + B_t\boldsymbol{u}_t,\ \boldsymbol{R}_t).$$

Our next assumption is that the measurement model is also linear, which is needed to preserve the Gaussian structure in the measurement update step. We assume the measurement model is of the form:

$$\boldsymbol{z}_t = C_t\boldsymbol{x}_t + \boldsymbol{\delta}_t, \tag{14.4}$$

where $\boldsymbol{\delta}_t$ is an independent measurement noise that is normally distributed with $\mathcal{N}(\mathbf{0}, \boldsymbol{Q}_t)$ and C_t is a time varying measurement model matrix. Again leveraging the mathematical properties of Gaussian distributions, we can write the probabilistic measurement model as:

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_t) = \frac{1}{\sqrt{\det(2\pi\boldsymbol{Q}_t)}}\exp\big(-\frac{1}{2}(\boldsymbol{z}_t - C_t\boldsymbol{x}_t)^\top \boldsymbol{Q}_t^{-1}(\boldsymbol{z}_t - C_t\boldsymbol{x}_t)\big),$$

such that $\boldsymbol{z}_t \sim \mathcal{N}(C_t\boldsymbol{x}_t,\ \boldsymbol{Q}_t)$.

To summarize, we assume that the initial belief is normally distributed and that both the state transition and measurement models are linear with Gaussian noise. These assumptions ensure that the prediction and measurement update steps from the Bayes filter will maintain the structure of the Gaussian belief, which makes the Kalman filter a practically efficient algorithm since we only need to update the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. The tradeoff of making these assumptions is that the Kalman filter is now limited to a more restricted class of problems.

14.2.1 *Algorithm (Predict–Correct Form)*

Let $\bar{\boldsymbol{\mu}}_t$ and $\bar{\boldsymbol{\Sigma}}_t$ denote the *predicted* mean and covariance, i.e., the belief after applying the control but before incorporating the new measurement.

Algorithm 14.1: Kalman Filter (linear-Gaussian)

Data: $\boldsymbol{\mu}_{t-1}$, $\boldsymbol{\Sigma}_{t-1}$, $\boldsymbol{u}_t$, $\boldsymbol{z}_t$
Result: $\boldsymbol{\mu}_t$, $\boldsymbol{\Sigma}_t$
`// Prediction (motion update)`
$\bar{\boldsymbol{\mu}}_t \leftarrow A_t \boldsymbol{\mu}_{t-1} + B_t \boldsymbol{u}_t$;
$\bar{\boldsymbol{\Sigma}}_t \leftarrow A_t \boldsymbol{\Sigma}_{t-1} A_t^\top + G_t \boldsymbol{R}_t G_t^\top$;
`// Innovation (measurement residual)`
$\tilde{\boldsymbol{z}}_t \leftarrow \boldsymbol{z}_t - C_t \bar{\boldsymbol{\mu}}_t$;
$S_t \leftarrow C_t \bar{\boldsymbol{\Sigma}}_t C_t^\top + \boldsymbol{Q}_t$;
`// Kalman gain`
$K_t \leftarrow \bar{\boldsymbol{\Sigma}}_t C_t^\top S_t^{-1}$;
`// Correction (measurement update)`
$\boldsymbol{\mu}_t \leftarrow \bar{\boldsymbol{\mu}}_t + K_t \tilde{\boldsymbol{z}}_t$;
`// Covariance update (Joseph form for numerical stability)`
$\boldsymbol{\Sigma}_t \leftarrow (\mathbf{I} - K_t C_t) \bar{\boldsymbol{\Sigma}}_t (\mathbf{I} - K_t C_t)^\top + K_t \boldsymbol{Q}_t K_t^\top$;
return $\boldsymbol{\mu}_t$, $\boldsymbol{\Sigma}_t$

One-line intuition: predict the state forward with process uncertainty; then "pull" the prediction toward the measurement proportionally to its reliability. The matrix S_t is the *innovation covariance*; the residual $\tilde{\boldsymbol{z}}_t$ is the *innovation*. The *Kalman gain* K_t balances model and measurement confidence.

14.2.2 *Derivation*

One way to derive the Kalman filter algorithm is by evaluating the Bayes filter updates from Chapter 13 with the Gaussian belief structure and probabilistic transition and measurement models. This would involve explicitly computing an integral of $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) p(\boldsymbol{x}_{t-1})$ for the prediction step, which is a little tedious. Instead, we consider a more intuitive approach that directly leverages the properties of Gaussians presented in Section 14.1. First, from the prior belief distribution, $\text{bel}(\boldsymbol{x}_{t-1}) \sim \mathcal{N}(\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1})$, we compute the predicted belief, $\overline{\text{bel}}(\boldsymbol{x}_{t-1})$, by using the affine transformation property of Gaussian random variables and the sum of two independent Gaussian random variables property. Specifically, we apply these properties to the linear state transition model in Equation (14.3) to give the predicted mean:

$$\bar{\boldsymbol{\mu}}_t = A_t \boldsymbol{\mu}_{t-1} + B_t \boldsymbol{u}_t + \mathbf{0},$$

where the $\mathbf{0}$ comes from the mean of the independent Gaussian process noise, $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{R}_t)$. The predicted covariance is then:

$$\bar{\boldsymbol{\Sigma}}_t = A_t \boldsymbol{\Sigma}_{t-1} A_t^\top + \boldsymbol{R}_t.$$

For the measurement update step of the Bayes filter, we have:

$$\text{bel}(\boldsymbol{x}_t) \propto p(\boldsymbol{z}_t \mid \boldsymbol{x}_t) \overline{\text{bel}}(\boldsymbol{x}_t),$$

where $p(\boldsymbol{z}_t \mid \boldsymbol{x}_t) \sim \mathcal{N}(C_t \boldsymbol{x}_t, \boldsymbol{Q}_t)$ and $\overline{\text{bel}}(\boldsymbol{x}_t) \sim \mathcal{N}(\bar{\boldsymbol{\mu}}_t, \bar{\boldsymbol{\Sigma}}_t)$. We can therefore use the fact that the product of two Gaussians probability density functions is also a Gaussian probability density function to compute:

$$\text{bel}(\boldsymbol{x}_t) = \eta \exp\big(-\frac{1}{2} J_t\big),$$

where η is a normalization constant and:

$$J_t = (\boldsymbol{z}_t - C_t \boldsymbol{x}_t)^\top \boldsymbol{Q}_t^{-1} (\boldsymbol{z}_t - C_t \boldsymbol{x}_t) + (\boldsymbol{x}_t - \bar{\boldsymbol{\mu}}_t)^\top \bar{\boldsymbol{\Sigma}}_t^{-1} (\boldsymbol{x}_t - \bar{\boldsymbol{\mu}}_t).$$

We compute the mean, $\boldsymbol{\mu}_t$, for this new probability density function by finding where the first derivative of $\text{bel}(\boldsymbol{x}_t)$ with respect to $\boldsymbol{x}_t$ is zero, which occurs when the derivative of J_t with respect to $\boldsymbol{x}_t$ is zero. Similarly, we compute the new covariance, $\boldsymbol{\Sigma}_t$, as the inverse of the second derivative of J_t with respect to $\boldsymbol{x}_t$. Therefore, we have the conditions:

$$\begin{aligned} 0 &= -C_t^\top \boldsymbol{Q}_t^{-1} (\boldsymbol{z}_t - C_t \boldsymbol{\mu}_t) + \bar{\boldsymbol{\Sigma}}_t^{-1} (\boldsymbol{\mu}_t - \bar{\boldsymbol{\mu}}_t), \\ \boldsymbol{\Sigma}_t^{-1} &= C_t^\top \boldsymbol{Q}_t^{-1} C_t + \bar{\boldsymbol{\Sigma}}_t^{-1}, \end{aligned}$$

which gives:

$$\boldsymbol{\Sigma}_t = (C_t^\top \boldsymbol{Q}_t^{-1} C_t + \bar{\boldsymbol{\Sigma}}_t^{-1})^{-1}.$$

and through algebraic manipulation, we write the mean in terms of the covariance $\boldsymbol{\Sigma}_t$:

$$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t + \boldsymbol{\Sigma}_t C_t^\top \boldsymbol{Q}_t^{-1} (\boldsymbol{z}_t - C_t \bar{\boldsymbol{\mu}}_t),$$

With a few additional algebraic steps, we now transform these equations in the form of the Kalman filter equations in Algorithm 14.1. From the matrix inversion lemma, we write:

$$(C_t^\top \boldsymbol{Q}_t^{-1} C_t + \bar{\boldsymbol{\Sigma}}_t^{-1})^{-1} = \bar{\boldsymbol{\Sigma}}_t - \bar{\boldsymbol{\Sigma}}_t C_t^\top (C_t \bar{\boldsymbol{\Sigma}}_t C_t^\top + \boldsymbol{Q}_t)^{-1} C_t \bar{\boldsymbol{\Sigma}}_t,$$

and then we define the Kalman gain as $K_t := \bar{\boldsymbol{\Sigma}}_t C_t^T (C_t \bar{\boldsymbol{\Sigma}}_t C_t^\top + \boldsymbol{Q}_t)^{-1}$ so that the covariance is given by:

$$\boldsymbol{\Sigma}_t = \bar{\boldsymbol{\Sigma}}_t - K_t C_t \bar{\boldsymbol{\Sigma}}_t,$$

Through some additional algebra, we also express the mean in terms of the Kalman gain to get:

$$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t + K_t (\boldsymbol{z}_t - C_t \bar{\boldsymbol{\mu}}_t).$$

Further details on this derivation and the algebraic steps involved can be found in Thrun et al.[3].

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

14.2.3 Practical Considerations

The Kalman filter exploits the structure of the Gaussian distribution, which makes it a computationally efficient algorithm for filtering in a continuous state space. However, the use of Gaussian beliefs also restricts the flexibility of the probabilistic model, since we have to assume the sufficiency of linear state transition and measurement models. In practice, this linearity assumption may not be very accurate with respect to the real world behavior of the robot and sensors. The structure also limits the belief distribution to be unimodal, which may limit performance in some applications. For example, in robot localization tasks, a multimodal distribution can better capture the global distribution. In the next section, we take a look at a variant of the Kalman filter that we can apply to problems with nonlinear state transition and measurement models, which is much more commonly applied in practical robotics settings.

14.3 Extended Kalman Filter (EKF)

The *extended Kalman filter (EKF)* generalizes the Kalman filter to nonlinear process and measurement models while retaining a Gaussian belief parameterization. It does so by locally *linearizing* the nonlinear models and then applying KF-style predict–correct updates to the mean and covariance. This makes the EKF a practical default for many robotics state estimation tasks with smooth nonlinear dynamics and sensing.

Instead of the linear models in Equation (14.3) and Equation (14.4) used by the Kalman filter, the EKF considers general nonlinear state transition and measurement models of the form:

$$\begin{aligned} \boldsymbol{x}_t &= f(\boldsymbol{x}_{t-1}, \boldsymbol{u}_t) + \boldsymbol{\epsilon}_t, \\ \boldsymbol{z}_t &= h(\boldsymbol{x}_t) + \boldsymbol{\delta}_t, \end{aligned} \tag{14.5}$$

where $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{R}_t)$ and $\boldsymbol{\delta}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{Q}_t)$ are normally distributed process and measurement noise terms.

The EKF incorporates these nonlinear models into the prediction and measurement update steps of the filter in two ways: first by using the nonlinear models directly and second by using their linear approximation from a first order Taylor series expansion. We perform the first order Taylor series expansion of the state transition model, $f(\boldsymbol{x}_{t-1}, \boldsymbol{u}_t)$, about the *most likely state* from the current belief distribution, which is the expected value, $\boldsymbol{\mu}_{t-1}$:

$$f(\boldsymbol{x}_{t-1}, \boldsymbol{u}_t) \approx f(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t) + G_t(\boldsymbol{x}_{t-1} - \boldsymbol{\mu}_{t-1}),$$

where $G_t = \nabla_x f(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t)$ is the Jacobian of $f(\boldsymbol{x}_{t-1}, \boldsymbol{u}_t)$ evaluated at $\boldsymbol{\mu}_{t-1}$. Using this linear approximation, we write the probabilistic state transition model 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:

$$\Delta \boldsymbol{x}_t = \boldsymbol{x}_t - f(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t) - G_t(\boldsymbol{x}_{t-1} - \boldsymbol{\mu}_{t-1}).$$

The prediction step of the EKF leverages the nonlinear state transition model and the linearized model to update the mean and covariance as:

$$\begin{aligned}\bar{\boldsymbol{\mu}}_t &= f(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t),\\ \bar{\boldsymbol{\Sigma}}_t &= G_t \boldsymbol{\Sigma}_{t-1} G_t^\top + \boldsymbol{R}_t,\end{aligned}$$

which exhibits a strong similarity to the Kalman filter prediction step.

We use a similar procedure for the measurement corrections. Specifically, we approximate the measurement model using a first order Taylor series expansion about the *predicted* point, $\bar{\boldsymbol{\mu}}_t$, to yield:

$$h(\boldsymbol{x}_t) \approx h(\bar{\boldsymbol{\mu}}_t) + H_t(\boldsymbol{x}_t - \bar{\boldsymbol{\mu}}_t),$$

where $H_t = \nabla_x h(\bar{\boldsymbol{\mu}}_t)$ is the Jacobian of $h(\boldsymbol{x}_t)$ evaluated at $\bar{\boldsymbol{\mu}}_t$. We then write the probabilistic measurement model 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 $\Delta \boldsymbol{z}_t = \boldsymbol{z}_t - h(\bar{\boldsymbol{\mu}}_t) - H_t(\boldsymbol{x}_t - \bar{\boldsymbol{\mu}}_t)$. The measurement update step of the EKF uses the nonlinear measurement model and the linear approximation to compute:

$$\begin{aligned}\boldsymbol{\mu}_t &= \bar{\boldsymbol{\mu}}_t + K_t(\boldsymbol{z}_t - h(\bar{\boldsymbol{\mu}}_t)),\\ \boldsymbol{\Sigma}_t &= (I - K_t H_t)\bar{\boldsymbol{\Sigma}}_t,\end{aligned}$$

where the Kalman gain is $K_t = \bar{\boldsymbol{\Sigma}}_t H_t^\top (H_t \bar{\boldsymbol{\Sigma}}_t H_t^\top + \boldsymbol{Q}_t)^{-1}$. Again, we can see that this is very similar to the Kalman filter measurement update step.

We combine the EKF prediction and measurement update steps together in the overall EKF algorithm definition in Algorithm 14.2. Compare this to Algorithm 14.1 and you will notice only the small difference that the EKF uses a combination of the nonlinear models and linear approximations from their Jacobians. Let $\bar{\boldsymbol{\mu}}_t, \bar{\boldsymbol{\Sigma}}_t$ denote the predicted mean and covariance.

Intuition. The EKF carries out a first-order approximation of the nonlinear models around the most plausible operating point (the mean). The KF formulas then apply to this locally linear surrogate. The quality of the update hinges on the local linearity of f and h around the chosen linearization points and on the fidelity of the noise covariances.

14.3.1 Practical Considerations

The extended Kalman filter provides more accurate results than the Kalman filter in many applications due to its ability to consider more general nonlinear models. However, the linear approximation of the nonlinear models by a Taylor

Algorithm 14.2: Extended Kalman Filter (EKF)

Data: $\boldsymbol{\mu}_{t-1}$, $\boldsymbol{\Sigma}_{t-1}$, $\boldsymbol{u}_t$, $\boldsymbol{z}_t$
Result: $\boldsymbol{\mu}_t$, $\boldsymbol{\Sigma}_t$
// Linearize dynamics at $(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t)$
$F_t \leftarrow \partial f / \partial \boldsymbol{x} \,\big|_{(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t)}$;
$L_t \leftarrow \partial f / \partial \boldsymbol{\epsilon} \,\big|_{(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t)}$;
// Prediction
$\bar{\boldsymbol{\mu}}_t \leftarrow f(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t)$;
$\bar{\boldsymbol{\Sigma}}_t \leftarrow F_t \, \boldsymbol{\Sigma}_{t-1} \, F_t^\top \; + \; L_t \, \boldsymbol{R}_t \, L_t^\top$;
// Linearize measurement at $\bar{\boldsymbol{\mu}}_t$
$H_t \leftarrow \partial h / \partial \boldsymbol{x} \,\big|_{\bar{\boldsymbol{\mu}}_t}$;
// Innovation and gain
$\tilde{\boldsymbol{z}}_t \leftarrow \boldsymbol{z}_t - h(\bar{\boldsymbol{\mu}}_t)$;
$S_t \leftarrow H_t \, \bar{\boldsymbol{\Sigma}}_t \, H_t^\top + \boldsymbol{Q}_t$;
$K_t \leftarrow \bar{\boldsymbol{\Sigma}}_t \, H_t^\top \, S_t^{-1}$;
// Correction
$\boldsymbol{\mu}_t \leftarrow \bar{\boldsymbol{\mu}}_t + K_t \, \tilde{\boldsymbol{z}}_t$;
// Joseph-form covariance update (numerically robust)
$\boldsymbol{\Sigma}_t \leftarrow (\mathbf{I} - K_t H_t) \, \bar{\boldsymbol{\Sigma}}_t \, (\mathbf{I} - K_t H_t)^\top + K_t \, \boldsymbol{Q}_t \, K_t^\top$;
return $\boldsymbol{\mu}_t$, $\boldsymbol{\Sigma}_t$

series expansion can lead to issues where the filter does not perform well, or even diverges if the approximation is not accurate enough[4]. The EKF also still suffers from the same unimodal modeling assumption as the Kalman filter, since the belief distribution is still represented by a single Gaussian distribution.

[4] Another extension of the Kalman filter, known as the *Unscented Kalman filter*, also uses general nonlinear state transition and measurement models, but avoids linearization by representing the Gaussian distribution by a set of samples points called *sigma-points*.

14.3.2 Unscented Kalman Filter (UKF)

The *unscented Kalman filter (UKF)* addresses a core limitation of EKF: first-order linearization can poorly capture the effect of strong nonlinearities on the mean and covariance. Instead of linearizing f and h, the UKF *propagates carefully chosen deterministically sampled points* (*sigma points*) through the true nonlinear functions and then recomputes the mean/covariance from the transformed samples.

Unscented transform (UT). For an n-dimensional Gaussian $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, construct $2n+1$ sigma points

$$\mathcal{X}^{(0)} = \boldsymbol{\mu}, \qquad \mathcal{X}^{(i)} = \boldsymbol{\mu} + \mathbf{c}_i, \qquad \mathcal{X}^{(i+n)} = \boldsymbol{\mu} - \mathbf{c}_i, \;\; i = 1, \ldots, n,$$

with $\mathbf{c}_i$ columns from a matrix square root of $(n+\lambda)\boldsymbol{\Sigma}$ (e.g., Cholesky). Associate weights $\{W_i^{(m)}, W_i^{(c)}\}$ for mean and covariance using parameters (α, β, κ) via $\lambda = \alpha^2(n+\kappa) - n$.[5] Pushing the sigma points through a nonlinear $g(\cdot)$ gives

[5] $\alpha \in (0, 1]$ scales the spread (e.g., 10^{-3}–0.3), β reflects prior knowledge of Gaussianity (often $\beta = 2$), κ is secondary scaling (e.g., 0 or $3 - n$).

$\mathcal{Y}^{(i)} = g(\mathcal{X}^{(i)})$; recompute the mean/covariance by the weighted average:

$$\bar{\mathbf{y}} = \sum_i W_i^{(m)} \mathcal{Y}^{(i)}, \qquad \mathbf{P}_{yy} = \sum_i W_i^{(c)} \big(\mathcal{Y}^{(i)} - \bar{\mathbf{y}}\big)\big(\mathcal{Y}^{(i)} - \bar{\mathbf{y}}\big)^\top.$$

The cross-covariance $\mathbf{P}_{xy}$ is formed similarly. For Gaussian priors, the UT matches mean/covariance up to at least second order (often third), without Jacobians.

UKF recursion (additive noise, sketch).

1. **Sigma set from prior:** build sigma points from $(\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1})$.
2. **Propagation:** pass them through $f(\cdot, \boldsymbol{u}_t)$ to obtain predicted points; compute $\bar{\boldsymbol{\mu}}_t$, $\bar{\boldsymbol{\Sigma}}_t \leftarrow \mathbf{P}_{ff} + \boldsymbol{R}_t$.
3. **Measurement transform:** pass predicted points through $h(\cdot)$ to get predicted measurements; compute $\hat{\boldsymbol{z}}_t$, $S_t \leftarrow \mathbf{P}_{hh} + \boldsymbol{Q}_t$, and cross-covariance $\mathbf{P}_{xz}$.
4. **Update:** $K_t = \mathbf{P}_{xz} S_t^{-1}$, $\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t + K_t(\boldsymbol{z}_t - \hat{\boldsymbol{z}}_t)$, $\boldsymbol{\Sigma}_t = \bar{\boldsymbol{\Sigma}}_t - K_t S_t K_t^\top$.

For *non-additive* noise, augment the state with noise variables and build sigma points in the augmented space.

EKF vs. UKF (when to use which).

- **EKF** is appropriate when f, h are mildly nonlinear, Jacobians are easy/accurate, and computational budget is tight. First-order accurate; may bias under strong curvature.
- **UKF** excels when nonlinearities are substantial, derivatives are hard/unreliable, or measurements are highly nonlinear (e.g., bearings). Captures mean/covariance more accurately; cost scales with $2n + 1$ transformations per step.
- **Square-root UKF** improves numerical stability by propagating Cholesky factors rather than covariances.

14.3.3 *Worked Example (Nonlinear Sensing): Range–Bearing Update*

Let $\boldsymbol{x}_t = [p_x,\, p_y,\, \theta]^\top$ be a robot pose, and a landmark at $\ell = [\ell_x, \ell_y]^\top$ is observed with range–bearing

$$h(\boldsymbol{x}_t) = \begin{bmatrix} \sqrt{(\ell_x - p_x)^2 + (\ell_y - p_y)^2} \\ \operatorname{atan2}(\ell_y - p_y, \ell_x - p_x) - \theta \end{bmatrix}.$$

The EKF uses $H_t = \partial h / \partial \boldsymbol{x}|_{\bar{\boldsymbol{\mu}}_t}$ and wraps the bearing residual to $(-\pi, \pi]$. The UKF bypasses H_t entirely by transforming sigma points through $h(\cdot)$; this often reduces bias for large range/bearing nonlinearities or when the prior is elongated (high covariance).

Summary. EKF and UKF are two sides of the same *Gaussian belief* coin: EKF linearizes models; UKF transports the distribution. Both rest on the Bayes filter and complement each other in robotics practice.

14.4 Non-parametric Filters: From Grids to Particles

Parametric filters gain efficiency by committing to a fixed belief shape (usually Gaussian). This can be too rigid when uncertainty is multimodal, the dynamics or sensing are strongly nonlinear, or data association is ambiguous. *Non-parametric* filters avoid a fixed functional form and instead approximate the belief either by (i) discretizing the state space into bins (histogram filters), or (ii) representing it with samples (particle filters). Both are direct approximations of the Bayes filter and trade additional computation for representational flexibility.

In Chapter 13, we introduced the robot localization and state estimation problem and how we can address it through a probabilistic framework. We also discussed the Bayes filter, a canonical algorithm that uses a probabilistic state transition and measurement model to recursively update a belief probability distribution over the state space. Then, we discussed the practical limitations of the Bayes filter, namely how the prediction update and measurement correction steps can sometimes be computationally intractable. We also showed how we can mitigate these practical issues by modeling the belief distribution in a *parametric* way, such as by using a Gaussian distribution in the Kalman filter and extended Kalman filter. The advantage of using a parametric belief distribution is that its structure can be exploited to reduce the complexity of the update steps, but a disadvantage is that it may not be able to effectively approximate the true belief distribution, which could lead to degraded performance. For example, when we use a Gaussian distribution to model the belief distribution in a continuous state space, we only have to update a finite set of mean and covariance parameters instead of the entire infinite-dimensional probability density function, but we are also limited to a unimodal belief distribution that may not be representative of the true state uncertainty.

In contrast to parametric filters, *non-parametric* filters do not make assumptions on the structure of the belief distribution[6]. This can be desirable for robotics applications where a fixed structure of the belief distribution may result in poor performance. For example, consider a robot localization problem where the robot may not initially know what room of a building it is in. In this case, we cannot effectively model the robot's position uncertainty by a unimodal Gaussian distribution since it could have distinct high probability modes of being in multiple rooms. In this chapter, we introduce two non-parametric filters that are based off of the Bayes filter approach from Chapter 13. The first is the *histogram filter*, which is an extension of the discrete Bayes filter to continuous state spaces. The second is the *particle filter*, which is a sample-based filter that is more computationally tractable for practical applications.

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

14.5 Histogram Filter

The *histogram filter* is a non-parametric filter that we can view as an extension of the discrete Bayes filter from Chapter 13 to continuous state spaces. This is accomplished by discretizing the continuous state space into a finite number of regions, and then we represent the belief distribution over the discretized state space by a finite set of probability masses. Mathematically, for the random state vector X, we discretize the continuous state space, $\mathcal{X}$, into a finite set of regions[7]:

$$\mathcal{X} = b_1 \cup b_2 \cup \ldots \cup b_K,$$

where b_k is the k-th "bin". For example, if the one-dimensional random variable, X, can have outcomes in the interval $[0, 1]$, then one possible decomposition is to split the interval $[0, 1]$ into a finite number of sub-intervals with equal width. We then define the belief distribution, $\text{bel}(\boldsymbol{x}_t)$, over the state, $\boldsymbol{x}_t$, at time t in a non-parametric way by specifying a probability mass, $p_{k,t}$, to each bin, b_k. We can also define a probability density function in a piecewise manner:

$$p(\boldsymbol{x}_t) = \frac{p_{k,t}}{|b_k|}, \quad \boldsymbol{x}_t \in b_k,$$

where $|b_k|$ denotes the "area" or "volume" of the bin.[8]

It is often useful to make explicit the binwise transition and likelihood that the filter approximates. Let the continuous transition and measurement models be $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)$ and $p(\boldsymbol{z}_t \mid \boldsymbol{x}_t)$. Then the *binwise* quantities are

$$T_{ki}^{(t)} := \mathbb{P}(\boldsymbol{x}_t \in b_k \mid \boldsymbol{x}_{t-1} \in b_i, \boldsymbol{u}_t) = \frac{1}{|b_i|} \int_{\boldsymbol{x}_{t-1} \in b_i} \int_{\boldsymbol{x}_t \in b_k} p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)\, d\boldsymbol{x}_t\, d\boldsymbol{x}_{t-1}, \tag{14.6}$$

$$L_k^{(t)} := p(\boldsymbol{z}_t \mid \boldsymbol{x}_t \in b_k) = \frac{1}{|b_k|} \int_{\boldsymbol{x}_t \in b_k} p(\boldsymbol{z}_t \mid \boldsymbol{x}_t)\, d\boldsymbol{x}_t. \tag{14.7}$$

In practice, we approximate these integrals. A common choice is to use a representative "mean" state for each bin:

$$\hat{\boldsymbol{x}}_k := \frac{1}{|b_k|} \int_{b_k} \boldsymbol{x}\, d\boldsymbol{x}. \tag{14.8}$$

Using these mean states, we approximate the probabilistic state transition model $p(b_{k,t} \mid b_{i,t-1}, \boldsymbol{u}_t)$ for transitioning from one bin to another by:

$$p(b_{k,t} \mid b_{i,t-1}, \boldsymbol{u}_t) \approx \eta |b_k|\, p(\hat{\boldsymbol{x}}_{k,t} \mid \hat{\boldsymbol{x}}_{i,t-1}, \boldsymbol{u}_t), \tag{14.9}$$

where $p(\hat{\boldsymbol{x}}_{k,t} \mid \hat{\boldsymbol{x}}_{i,t-1}, \boldsymbol{u}_t)$ is the original, non-discretized, state transition model evaluated at the mean bin states, $\hat{\boldsymbol{x}}_k$, and η is a normalization constant[9]. We discretize the probabilistic measurement model in a similar manner:

$$p(\boldsymbol{z}_t \mid b_{k,t}) \approx p(\boldsymbol{z}_t \mid \hat{\boldsymbol{x}}_{k,t}), \tag{14.10}$$

such that we approximate the measurement probability associated with a bin, b_k, by the measurement probability associated with the mean bin state, $\hat{\boldsymbol{x}}_k$.[10]

[7] In the context of the histogram filter, we often refer to the the discretized regions as *bins*.

[8] By construction $\sum_k p_{k,t} = 1$. In practice, always renormalize after the correction step to maintain this invariant.

[9] If the bin areas $|b_k|$ are equal, we can absorb this term into the normalization constant η. For higher fidelity, sample multiple points per bin (simple quadrature) and average.

[10] For angular variables (e.g., headings), ensure the representative state respects periodicity and wrap residuals appropriately.

Once we have discretized the state space with the bins, b_k, and the state transition and measurement models using the bin mean states, $\hat{x}_k$, the histogram filter mimics the discrete Bayes filter in Algorithm 13.2, which we detail in Algorithm 14.3.

Algorithm 14.3: Histogram Filter

Data: $\{p_{k,t-1}\}, \boldsymbol{u}_t, \boldsymbol{z}_t$
Result: $\{p_{k,t}\}$
foreach k **do**
  $\overline{p}_{k,t} = \sum_i p(b_{k,t} \mid b_{i,t-1}, \boldsymbol{u}_t)\, p_{i,t-1}$
  $p_{k,t} = p(\boldsymbol{z}_t \mid b_{k,t})\, \overline{p}_{k,t}$
// Normalization (to enforce $\sum_k p_{k,t} = 1$; use log-weights to avoid underflow if needed)
$\eta \leftarrow (\sum_k p_{k,t})^{-1}$; **foreach** k **do**
  $p_{k,t} \leftarrow \eta\, p_{k,t}$
return $\{p_{k,t}\}$

When implementing this algorithm, one can consider the following insights. First, the transition matrix is typically *sparse*: most motion models move mass only to nearby bins—iterate over neighbors of each b_i ("push") to reduce cost. Second, if $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)$ is shift-invariant (e.g., additive Gaussian motion in a grid), the prediction is a discrete *convolution* that can be accelerated with separable kernels or FFTs in 1D/2D. Third, at domain boundaries, use reflecting, absorbing, or wrap-around conditions consistent with the task (e.g., wrap headings on S^1).

Like the discrete Bayes filter, the main disadvantage of the histogram filter is that it can become computationally intractable if the number of values representing the belief distribution is too large. This can happen if the discretization of the continuous state space is high-resolution, which could be important for accuracy, or if the state space is high-dimensional. For example, in a robot localization problem where we are trying to estimate a two-dimensional pose of the robot in a building, we would need to discretize along three dimensions: the two-dimensional position and the heading. If we discretized with a relatively coarse resolution of one meter and one degree for heading, we could easily have hundreds of thousands of bins.[11]

[11] Cost scales roughly as $O(K\, n_{\text{nbr}})$ per step, where n_{nbr} is the number of motion-neighbor bins (often small). Memory is $O(K)$. This "curse of dimensionality" motivates particle filters for higher-dimensional continuous states.

Worked example (1D line, Gaussian motion + range). Consider a robot moving along a line segment $[0, L]$. We discretize this interval into K equal bins of width $\Delta = L/K$, with bin k spanning $[x_k^-, x_k^+]$ and centroid $\hat{x}_k = (k - \frac{1}{2})\Delta$. The robot follows a simple additive motion model and receives a noisy range measurement to the wall at the origin:

$$x_t = x_{t-1} + u_t + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, \sigma_u^2), \qquad z_t = x_t + \nu_t, \quad \nu_t \sim \mathcal{N}(0, \sigma_z^2).$$

Prediction. After applying control u_t, the robot's position distribution is shifted and blurred according to motion noise. For each pair of bins (i, k), the probability of moving from bin i to bin k is given by the Gaussian mass over b_k centered at $\hat{x}_i + u_t$:

$$T_{ki}^{(t)} \approx \Phi\Big(\tfrac{x_k^+ - (\hat{x}_i + u_t)}{\sigma_u}\Big) - \Phi\Big(\tfrac{x_k^- - (\hat{x}_i + u_t)}{\sigma_u}\Big),$$

where Φ is the standard normal CDF. The predicted belief $\overline{p}_{k,t}$ is obtained by summing these contributions over all bins i.

Correction. Given a measurement z_t, each bin is reweighted according to how likely its centroid $\hat{x}_k$ is under the measurement model:

$$L_k^{(t)} \approx \mathcal{N}\big(z_t;\, \hat{x}_k, \sigma_z^2\big).$$

The posterior is then $p_{k,t} \propto L_k^{(t)}\, \overline{p}_{k,t}$, followed by normalization so that $\sum_k p_{k,t} = 1$.

Discussion. This process naturally captures both unimodal and multimodal beliefs. For example, starting from a uniform prior, the filter may maintain several possible peaks if the measurement is ambiguous (e.g., the environment has repeated structures). As more controls and measurements accumulate, the inconsistent modes gradually vanish, and the posterior collapses to a single sharp peak at the true location. In this way, the histogram filter provides a simple yet powerful tool for global localization in low-dimensional settings.

14.6 Particle Filter

The *particle filter* is a non-parametric filter that we can apply to continuous state spaces in a more computationally tractable way than the histogram filter. Rather than discretizing the state space *a priori* like the histogram filter, this filter represents the belief distribution by a finite set of samples from the state space called *particles*[12]. We define the set of particles at time t mathematically as $\mathcal{P}_t$:

$$\mathcal{P}_t \coloneqq \{\boldsymbol{x}_t^{[1]}, \boldsymbol{x}_t^{[2]}, \ldots, \boldsymbol{x}_t^{[K]}\}, \tag{14.11}$$

where $\boldsymbol{x}_t^{[k]}$ is the k-th particle. Each particle, $\boldsymbol{x}_t^{[k]}$, represents a hypothesis about the true state, $\boldsymbol{x}_t$, and therefore regions of the state space with more particles correspond to regions of higher probability. The particles are ideally always distributed according to the current belief:

$$\boldsymbol{x}_t^{[k]} \sim \text{bel}(\boldsymbol{x}_t),$$

but theoretically this only occurs as the number of samples, K, approaches infinity[13].

Following the Bayes filter paradigm, the particle filter updates the prior belief distribution, represented by the set of particles, $\mathcal{P}_{t-1}$, with a prediction and

[12] The particle filter is sometimes referred to as a *Monte Carlo* algorithm due to its sampling approach.

[13] In practice, the set of particles only *approximately* represents the belief distribution, and in many common applications, around $K \approx 1000$ samples tends to be sufficient

measurement update step. The prediction step considers each particle, $\boldsymbol{x}_{t-1}^{[k]}$, in the prior set, $\mathcal{P}_{t-1}$, and samples a new predicted sample from the state transition model:

$$\bar{\boldsymbol{x}}_t^{[k]} \sim p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}^{[m]}, \boldsymbol{u}_t).$$

The filter then computes an importance factor, $w_t^{[k]}$, for each predicted sample, $\bar{\boldsymbol{x}}_t^{[k]}$, based on how well the observed measurement, $\boldsymbol{z}_t$, matches the prediction:

$$w_t^{[k]} = p(\boldsymbol{z}_t \mid \bar{\boldsymbol{x}}_t^{[k]}).$$

We then collect the predicted particles, $\bar{\boldsymbol{x}}_t^{[k]}$, and their associated weights, $w_t^{[k]}$, in a new particle set, $\bar{\mathcal{P}}_t$, that represents the predicted belief distribution, $\overline{\text{bel}}(\boldsymbol{x}_t)$. The particle filter's measurement update step consists of *resampling* (with replacement) a new set of K particles from the predicted set, $\bar{\mathcal{P}}_t$, with a probability proportional to the weights, $w_t^{[k]}$. This step gives preference in the new sample set to the predicted particles that showed higher correlation to the measurement, $\boldsymbol{z}_t$. Finally, we collect the resampled points to define the posterior belief distribution particle set, $\mathcal{P}_t$. We outline the particle filter algorithm in Algorithm 14.4, and we show a few iterations of the algorithm for a simple robot localization problem in Figure 14.1.

Algorithm 14.4: Particle Filter

Data: $\mathcal{P}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t$
Result: $\mathcal{P}_t$
$\bar{\mathcal{P}}_t = \mathcal{P}_t = \emptyset$
for $k = 1$ **to** K **do**
  Sample $\bar{\boldsymbol{x}}_t^{[k]} \sim p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}^{[k]}, \boldsymbol{u}_t)$
  $w_t^{[k]} = p(\boldsymbol{z}_t \mid \bar{\boldsymbol{x}}_t^{[k]})$
  $\bar{\mathcal{P}}_t = \bar{\mathcal{P}}_t \cup \left(\bar{\boldsymbol{x}}_t^{[k]}, w_t^{[k]}\right)$
for $k = 1$ **to** K **do**
  Draw i with probability $\propto w_t^{[i]}$
  $\mathcal{P}_t = \mathcal{P}_t \cup \{\bar{\boldsymbol{x}}_t^{[i]}\}$
return $\mathcal{P}_t$

The resampling step of the particle filter is important for practical reasons beyond just updating the belief for the measurement corrections. In particular, without the resampling step, the particles would drift over time to regions of low probability and there would be fewer particles to represent the regions of high probability. This could lead to a loss of accuracy and overall divergence of the filter from a good representation of the belief. We can therefore view the resampling step as a probabilistic implementation of the Darwinian idea of survival of the fittest, since it refocuses the particle set to regions in the state space with high posterior probability. This also helps make the algorithm computa-

tional efficient since it reduces the number of particles that we need by focusing them on the regions of the state space that have higher probability.

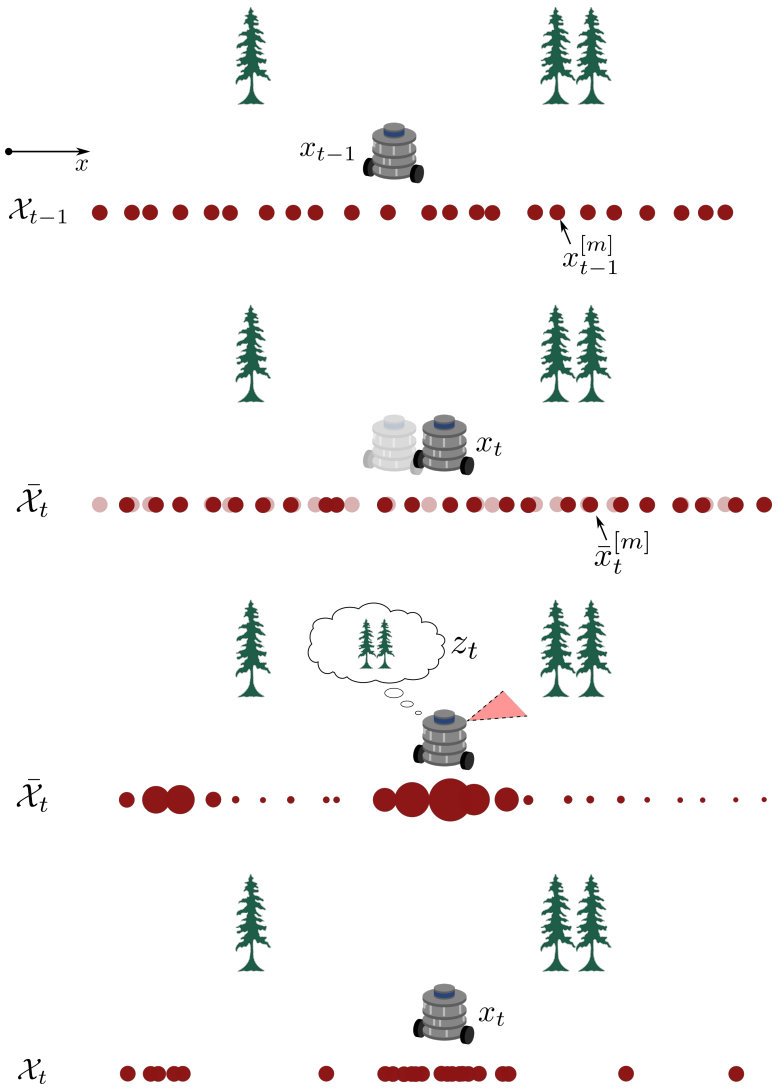


Figure 14.1: A particle filter used for robot localization. We first update the initial set of particles according to the transition model, and then weight them according to the observation. Finally, we generate a new set of particles through weighted resampling.

Decision guide.

- If dynamics/sensing are mildly nonlinear and beliefs stay near unimodal ⇒ **EKF**.
- If Jacobians are hard or curvature matters ⇒ **UKF** (or square-root UKF).
- If global/multimodal uncertainty or severe non-Gaussian noise ⇒ **Particle filter**.
- If state is low-dimensional and a map/grid is available ⇒ **Histogram filter**.

Table 14.1: Parametric vs. Non-parametric Filters (at a glance)

	Parametric (KF / EKF / UKF)	**Non-parametric (Histogram / Particle)**
Belief	Single Gaussian (mean, covariance)	Grid masses or weighted samples
Nonlinearity	EKF (linearize), UKF (sigma points)	Native; no Jacobians needed
Multimodality	Poor (unimodal)	Natural (multi-peak)
Dimensionality	Scales well with state dim	Histogram: suffers; Particle: scalable with K
Computation	Cheap per step	Histogram: grows with bins; Particle: $O(K)$
When it shines	Smooth models, near-Gaussian noise, good observability	Ambiguity, strong nonlinearities, non-Gaussian noise, global localization
Pitfalls	Linearization bias (EKF), covariance inconsistency	Degeneracy without resampling; sample impoverishment

14.7 *Exercises*

The starter code for the exercises provided below is available online through GitHub. To get started, download the code by running in a terminal window:

```
git clone https://github.com/StanfordASL/pora-exercises.git
```

We denote Problems requiring hand-written solutions and coding in Python with ✎ and 🖳, respectively.

🖳 *Problem 1: Kalman Filter*

In the file `ch14/exercises/kalman_filter.ipynb`, implement the Kalman filter predict and update steps and then apply the algorithm to a landmark localization problem.

References

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

15
Robot Localization

Chapter 13 introduced a probabilistic framework for the localization and state estimation problem along with the canonical Bayes filter algorithm. The previous chapters also introduced some widely used filtering algorithms based on the Bayes filter, including the Kalman filter, extended Kalman filter, histogram filter, and particle filter. Despite their foundational significance, these algorithms typically need further refinements and extensions before we can effectively apply them to the task of robot localization. For instance, in their standard form, these algorithms do not incorporate a local *map* of the environment. Local maps may be very important for localization if the robot relies on sensors like laser rangefinders or cameras that identify local features rather than global sensors like GNSS. Therefore, we now consider a more specific definition of robot localization, which is to determine the pose of a robot relative to a *map* of the environment[1]. In this chapter, we give an overview of the taxonomy of robot localization problems, and then introduce several map-based localization algorithms[2] that are based on the framework of the Bayes filter.

[1] When using a map for localization, we can view our goal as finding the coordinate transformation between the map's global coordinate frame and the robot's local coordinate frame.

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

15.1 A Taxonomy of Robot Localization Problems

We can better understand the broad scope of challenges related to robot localization by developing a compact taxonomy of localization problems. This categorization will divide localization problems along a number of important dimensions, such as how much initial knowledge the robot has, whether the environment is static or dynamic, if information is gathered passively or actively, and whether information about the environment is gathered by one robot, or collaboratively among several robots.

Pose Tracking and Global Pose Localization: The first way we categorize robot localization problems is by how much knowledge is initially available to the robot, which can have a significant impact on what type of localization algorithm is most appropriate for the problem. *Pose tracking* problems generally assume that the initial global pose of the robot is at least roughly known, and the task is to incrementally update or refine the pose as the robot navigates through the en-

vironment. In these types of problems, the localization error is generally small and it is often reasonable to model the belief distribution as unimodal, and therefore one option is to use a Kalman filter or EKF from Chapter 14. *Global localization* problems do not make any assumption regarding knowledge of the initial pose, and unimodal belief distributions are likely not going to effectively capture the global uncertainty. For global localization problems, we might prefer non-parametric filters, such as the particle filter from Chapter 14, that are better suited for modeling multiple hypotheses. An even more challenging variant of the global localization problem is referred to as the *kidnapped robot problem*, where we consider scenarios where the robot could be "teleported" to a new location at any time. In this problem, we have to deal with sudden large changes of the robot's pose, which means the localization algorithm has to be more robust and be able to recover from potentially large divergences.

Static and Dynamic Environment Localization: Environmental changes are another important aspect that we must consider for mobile robot localization. In a *static* environment, the robot is the only object that moves, and in a *dynamic* environment, other objects may change locations or configurations over time. Localization is certainly easier in static environments. Dynamic environments may require augmentation of the localization algorithm, such as by including the state of changing environmental elements in the filter algorithm, in addition to the robot's state.

Active and Passive Localization: Gathering information using the robot' sensors is crucial for localization, and localization is crucial for motion planning so the robot can complete its task. We therefore should consider closing the loop by also including information gathering actions during downstream motion planning tasks. In our taxonomy, *active localization* problems consider the ability of the robot to explicitly choose its actions to improve its localization and overall understanding of the environment, while *passive localization* problems assume the robot's motion is unrelated to localization. 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 more environmental information as it moves along the wall. We can also consider hybrid approaches that only use active localization when strictly needed, which would help avoid inefficiencies if more sensor information doesn't significantly improve the robot's ability to complete the task.

Single and Multi-Robot Localization: Finally, we also categorize localization problems based on whether there is only a single robot or multiple that gather independent information and cooperatively share it. *Single-robot* problems are the most commonly studied and utilized approach and are simpler, but *multi-robot localization* can lead to better system level performance. In multi-robot problems, the robots aren't limited to sharing sensor information, they can also share their belief distributions.

15.2 Robot Localization via Bayesian Filtering

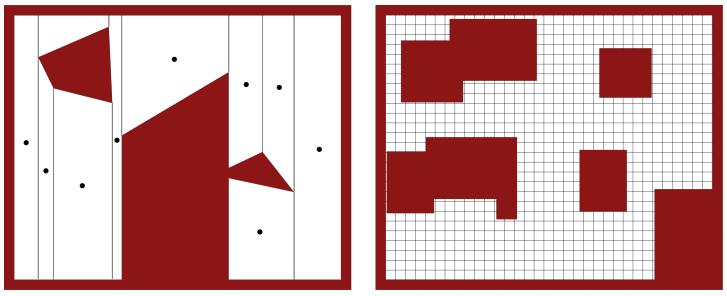
In previous chapters, we introduced several well-known variations of the Bayes filter, including the parametric extended Kalman filter in Chapter 14 and the nonparametric particle filter in Chapter 14. These algorithms rely on a probabilistic Markov state transition model and measurement model to update the robot's belief distribution over time, and in this section, we show how we can also incorporate the notion of an environment *map* into this framework.

The three main variables of the models in the general filtering context are the state, $\boldsymbol{x}_t$, the control input, $\boldsymbol{u}_t$, and the measurements, $\boldsymbol{z}_t$. For map-based localization, we introduce a new vector, $\boldsymbol{m}$, that is a collection of properties about objects in the environment which we define as:

$$\boldsymbol{m} = \{m_1, m_2, \ldots, m_N\}, \tag{15.1}$$

where m_i represents the set of properties of a specific object and N is the number of objects being tracked. We generally consider two types of maps: location-based maps and feature-based maps. The choice of map type can lead to differences in both computational efficiency and expressiveness of the resulting algorithm. In location-based maps, the index i associated with object m_i corresponds to a specific location. For example, the object m_i in a location-based map might represent cells in a cell decomposition or grid representation[3] of a map, such as we show in Figure 15.1. One potential disadvantage of cell-based maps

[3] In three dimensions, the grid representation is usually referred to as a *voxel* representation.



Cell Decomposition Map Grid-based Map

Figure 15.1: Two examples of location-based maps. Both represent the map as a set of volumetric objects, which in these examples are cells.

is that their resolution is dependent on the size of the cells, and therefore there is a tradeoff between computational complexity and accuracy. However, their advantage is that they can explicitly encode information about the presence of objects in specific locations.

In feature-based maps, the index i is a feature index, and m_i contains information about the properties of that feature, such as its Cartesian location[4]. Figure 15.2 shows two examples of feature-based maps, one is represented by a set of lines and another is represented by nodes and edges of a graph[5]. Feature-based maps can be computationally efficient and can be finely tuned to specific environments, for example, a line-based map might make sense to use in highly structured environments such as buildings. However, their main disadvantage is

[4] We can also think of feature-based maps as a collection of landmarks.

[5] Generally referred to as a *topological map*.

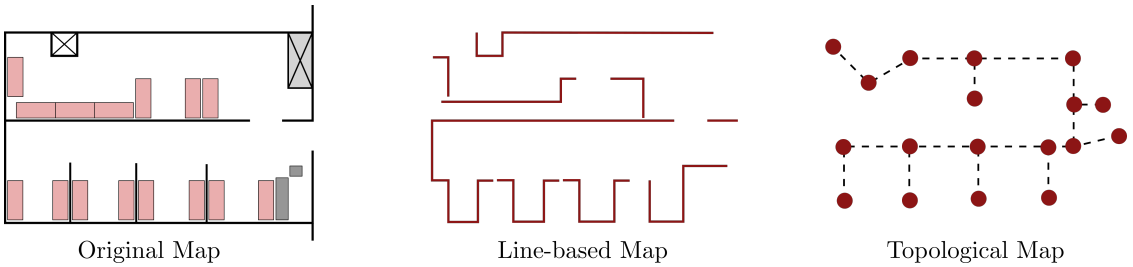


Figure 15.2: Two examples of feature-based maps. One uses a collection of lines and the other uses a graph representation of the empty spaces.

that they may have lower resolution and might not capture spatial information about all potential obstacles.

15.2.1 *Map-Aware State Transition Model*

In the previous chapters on Bayesian filtering, the probabilistic state transition model was given by $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)$, but in map-based robot localization problems we must also account for how the map, $\boldsymbol{m}$, affects the state transition, since in general:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) \neq p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t, \boldsymbol{m}).$$

For example, the model $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)$ does not account for the fact that a robot cannot move through walls in the environment. To include the map in the transition model, we make an assumption that:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t, \boldsymbol{m}) \approx \eta \frac{p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) p(\boldsymbol{x}_t \mid \boldsymbol{m})}{p(\boldsymbol{x}_t)}, \tag{15.2}$$

where η is a normalization constant. This approximation is derived from Bayes' rule by assuming that $p(\boldsymbol{m} \mid \boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) \approx p(\boldsymbol{m} \mid \boldsymbol{x}_t)$[6]:

$$\begin{aligned}
p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t, \boldsymbol{m}) &= \frac{p(\boldsymbol{m} | \boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)}{p(\boldsymbol{m} \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t)}, \\
&= \eta' p(\boldsymbol{m} | \boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t), \\
&\approx \eta' p(\boldsymbol{m} | \boldsymbol{x}_t) p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t), \\
&= \eta \frac{p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) p(\boldsymbol{x}_t \mid \boldsymbol{m})}{p(\boldsymbol{x}_t)},
\end{aligned}$$

where η' and η are normalization constants. In this approximation, the term $p(\boldsymbol{x}_t \mid \boldsymbol{m})$ is the state probability conditioned on the map, which we can think of as describing the "consistency" of the state with respect to the map. For example, $p(\boldsymbol{x}_t \mid \boldsymbol{m}) = 0$ for a state, $\boldsymbol{x}_t$, that cannot be physically realized by the robot, such as being inside of a wall of a building. We can therefore view the approximation in Equation (15.2) as taking an initial probabilistic guess using the original state transition model without map knowledge, and then using the map consistency term, $p(\boldsymbol{x}_t \mid \boldsymbol{m})$, to update the likelihood of the new state, $\boldsymbol{x}_t$, given the map.

[6] Note that this assumption improves as the transition time between $t-1$ and t is reduced.

15.2.2 Map-Aware Measurement Model

We also need to modify the probabilistic measurement model model, $p(\boldsymbol{z}_t \mid \boldsymbol{x}_t)$, to account for map information since local measurements are significantly influenced by the environment. For example, a range measurement is clearly dependent on what object is currently in the line of sight, such as a wall or some other feature in the map. We express the new map-aware measurement model as $p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m})$, where the measurement is now also conditioned on the map.

Another simplifying assumption we sometimes make for the measurement model is that each of the measurements of the vector $\boldsymbol{z}_t \in \mathbb{R}^p$ are conditionally independent of each other, given the state and map. With this assumption, we can express the measurement model as:

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m}) = \prod_{i=1}^{p} p(z_t^i \mid \boldsymbol{x}_t, \boldsymbol{m}). \tag{15.3}$$

15.3 Markov Localization

The first map-based localization algorithm we introduce is referred to as the *Markov localization* algorithm. This algorithm is conceptually the same as the Bayes filter, except for the inclusion of the map in the probabilistic state transition model, $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t, \boldsymbol{m})$, and measurement model, $p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m})$. The algorithm, which we outline in Algorithm 15.1, provides a general framework that we can use to address many of the localization problems discussed in our taxonomy in Section 15.1, including pose tracking and global pose localization problems, but it does assume the map is already known. Similarly to the Bayes

Algorithm 15.1: Markov Localization

Data: $\text{bel}(\boldsymbol{x}_{t-1}), \boldsymbol{u}_t, \boldsymbol{z}_t, \boldsymbol{m}$
Result: $\text{bel}(\boldsymbol{x}_t)$
foreach $\boldsymbol{x}_t$ **do**
  $\overline{\text{bel}}(\boldsymbol{x}_t) = \int p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t, \boldsymbol{m}) \text{bel}(\boldsymbol{x}_{t-1}) d\boldsymbol{x}_{t-1}$
  $\text{bel}(\boldsymbol{x}_t) = \eta p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m}) \overline{\text{bel}}(\boldsymbol{x}_t)$
return $\text{bel}(\boldsymbol{x}_t)$

filter from Chapter 13, the Markov localization algorithm in Algorithm 15.1 is generally not computationally tractable to implement in practice. Therefore, like the Bayes filter, we use this algorithm as a foundation to design more practical algorithms. In Section 15.4, we again use a Gaussian belief distribution, like in the EKF, to design an *extended Kalman filter localization* algorithm. Then, in Section 15.5, we again use a sampling-based approach, like in the particle filter, to design the *Monte Carlo localization* algorithm.

15.4 Extended Kalman Filter (EKF) Localization

The *extended Kalman filter (EKF) localization* algorithm is a map-based localization algorithm that leverages the Markov localization framework and is very similar to the general EKF algorithm from Chapter 14. Specifically, this algorithm models the belief distribution as a Guassian such that $\text{bel}(\boldsymbol{x}_t) \sim \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$. This modeling choice adds structure to the filtering problem, which improves the computational efficiency relative to the general Markov localization algorithm[7].

[7] As with the EKF, one tradeoff of this choice is that we are restricted to a unimodal distribution that is not expressive enough to solve global pose localization problems.

As for the EKF in Section 14.3, we assume the state transition model is:

$$\boldsymbol{x}_t = f(\boldsymbol{x}_{t-1}, \boldsymbol{u}_t) + \boldsymbol{\epsilon}_t,$$

where $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{R}_t)$ is a zero-mean Gaussian process noise. We also compute the state transition Jacobian, G_t, as:

$$G_t = \nabla_{\boldsymbol{x}} f(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t), \tag{15.4}$$

where $\boldsymbol{\mu}_{t-1}$ is the mean of the previous belief distribution, $\text{bel}(\boldsymbol{x}_{t-1})$. The main difference between the general EKF algorithm and the EKF localization algorithm is the assumption that a feature-based map is available. Specifically, we assume the map, $\boldsymbol{m}$, consists of N point landmarks:

$$\boldsymbol{m} = \{m_1, m_2, \ldots, m_N\}, \quad m_j = (m_{j,x}, m_{j,y}),$$

where we define each landmark, m_j, by its two-dimensional location, $(m_{j,x}, m_{j,y})$, in the global coordinate frame. The measurements, $\boldsymbol{z}_t$, associated with these point landmarks at time t are:

$$\boldsymbol{z}_t = \{\boldsymbol{z}_t^1, \boldsymbol{z}_t^2, \ldots\},$$

where $\boldsymbol{z}_t^i$ is the measurement associated with a particular landmark. We assume that each landmark measurement is generated by the model:

$$\boldsymbol{z}_t^i = h(\boldsymbol{x}_t, j, \boldsymbol{m}) + \boldsymbol{\delta}_t,$$

where $\boldsymbol{\delta}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{Q}_t)$ models zero-mean Gaussian sensor noise and j is the index of the map feature, $m_j \in \boldsymbol{m}$, that is associated with measurement i.

One new fundamental problem that we now need to consider is the *data association problem*, which arises due to uncertainty in which measurements are associated with which landmark. We mathematically denote the measurement-feature correspondences by the variable $c_t^i \in \{1, \ldots, N+1\}$, which takes on the value $c_t^i = j$ if measurement i corresponds to landmark j, and $c_t^i = N+1$ if measurement i has no corresponding landmark. Given the feature correspondence, c_t^i, for measurement i, we can compute the Jacobian H_t^i that we use in the EKF measurement update step. Specifically, for the i-th measurement, we compute the Jacobian of the new measurement model by:

$$H_t^{c_t^i} = \nabla_{\boldsymbol{x}} h(\bar{\boldsymbol{\mu}}_t, c_t^i, \boldsymbol{m}), \tag{15.5}$$

where $\bar{\boldsymbol{\mu}}_t$ is the predicted mean from the EKF prediction step.

15.4.1 EKF Localization with Known Correspondences

In practice, we generally will not know the correspondences, c_t^i, between measurements, $\boldsymbol{z}_t^i$, and landmarks, m_j. However, for pedagogical purposes, it is useful for us to first consider the form of the EKF localization algorithm for the case where we assume the correspondences are known. For known correspondences, we give the EKF localization algorithm in Algorithm 15.2. Compared to the general EKF algorithm in Algorithm 14.2, we can see that the main difference is that we process multiple measurements at the same time. This is possible because of our assumption in Equation (15.3) that the measurements are conditionally independent. Specifically, with this conditional independence assumption and some special properties of Gaussian distributions, we can perform the multi-measurement update by sequentially looping over each measurement and applying the standard EKF measurement update steps.

Algorithm 15.2: EKF Localization, Known Correspondences

Data: $\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t, \boldsymbol{c}_t, \boldsymbol{m}$
Result: $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$
$\bar{\boldsymbol{\mu}}_t = f(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t)$
$\bar{\boldsymbol{\Sigma}}_t = G_t \boldsymbol{\Sigma}_{t-1} G_t^\top + \boldsymbol{R}_t$
foreach $\boldsymbol{z}_t^i$ **do**
  $j = c_t^i$
  $S_t^i = H_t^j \bar{\boldsymbol{\Sigma}}_t [H_t^j]^\top + \boldsymbol{Q}_t$
  $K_t^i = \bar{\boldsymbol{\Sigma}}_t [H_t^j]^\top [S_t^i]^{-1}$
  $\bar{\boldsymbol{\mu}}_t = \bar{\boldsymbol{\mu}}_t + K_t^i (\boldsymbol{z}_t^i - h(\bar{\boldsymbol{\mu}}_t, j, \boldsymbol{m}))$
  $\bar{\boldsymbol{\Sigma}}_t = (I - K_t^i H_t^j) \bar{\boldsymbol{\Sigma}}_t$
$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t$
$\boldsymbol{\Sigma}_t = \bar{\boldsymbol{\Sigma}}_t$
return $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$

15.4.2 EKF Localization with Unknown Correspondences

Algorithm 15.2 shows what the EKF localization algorithm looks like if the correspondences are known, but in practice they will be uncertain and we will need to estimate them along with the robot state. One approach is to use maximum likelihood estimation, where we calculate the most likely value of the correspondences, $\boldsymbol{c}_t$, by maximizing the measurement 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, we choose the correspondence variables to maximize the probability of getting the current measurement given the map and the history of the correspondence variables, the measurements, and the controls. By marginalizing

over the current pose, $\boldsymbol{x}_t$, we can write this probability distribution as:

$$\begin{aligned} p(\boldsymbol{z}_t \mid \boldsymbol{c}_{1:t}, \boldsymbol{m}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) &= \int p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{c}_{1:t}, \boldsymbol{m}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) p(\boldsymbol{x}_t \mid \boldsymbol{c}_{1:t}, \boldsymbol{m}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) d\boldsymbol{x}_t, \\ &= \int p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{c}_t, \boldsymbol{m}) \overline{\text{bel}}(\boldsymbol{x}_t) d\boldsymbol{x}_t, \end{aligned}$$

which leverages the Markov assumption to simplify $p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{c}_{1:t}, \boldsymbol{m}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) = p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{c}_t, \boldsymbol{m})$, and uses the definition of the predicted belief:

$$\overline{\text{bel}}(\boldsymbol{x}_t) \coloneqq p(\boldsymbol{x}_t \mid \boldsymbol{c}_{1:t}, \boldsymbol{m}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}),$$

which now includes the map and correspondences. Note that the term $p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{c}_t, \boldsymbol{m})$ is the measurement model given *known* correspondences, which we can factor using the conditional independence assumption from Equation (15.3) so that:

$$p(\boldsymbol{z}_t \mid \boldsymbol{c}_{1:t}, \boldsymbol{m}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) = \int \overline{\text{bel}}(\boldsymbol{x}_t) \prod_i p(\boldsymbol{z}_t^i \mid \boldsymbol{x}_t, c_t^i, \boldsymbol{m}) d\boldsymbol{x}_t.$$

Since each correspondence variable, c_t^i, in the integral shows up in a separate term of the product, we can simplify the optimization by performing an independent optimization over each correspondence parameter:

$$\hat{c}_t^i = \arg\max_{c_t^i} \int p(\boldsymbol{z}_t^i \mid \boldsymbol{x}_t, c_t^i, \boldsymbol{m}) \overline{\text{bel}}(\boldsymbol{x}_t) d\boldsymbol{x}_t.$$

We can solve this problem efficiently under the assumption that the measurement model and belief distribution are Gaussian[8]. In particular, the probability distribution resulting from the integral is a Gaussian with mean and covariance:

$$\int p(\boldsymbol{z}_t^i \mid \boldsymbol{x}_t, c_t^i, \boldsymbol{m}) \overline{\text{bel}}(\boldsymbol{x}_t) d\boldsymbol{x}_t \sim \mathcal{N}(h(\bar{\boldsymbol{\mu}}_t, c_t^i, \boldsymbol{m}),\ H_t^{c_t^i} \bar{\boldsymbol{\Sigma}}_t [H_t^{c_t^i}]^\top + \boldsymbol{Q}_t).$$

[8] Similar to the previous chapters, in this case, the product of terms inside the integral will be Gaussian since both terms are Gaussian.

We can therefore express the maximum likelihood optimization problem 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 $S_t^j = H_t^j \bar{\boldsymbol{\Sigma}}_t [H_t^j]^\top + \boldsymbol{Q}_t$. This maximization is also equivalent to:

$$\hat{c}_t^i = \arg\min_{c_t^i}\ d_t^{i,c_t^i}, \tag{15.6}$$

where:

$$d_t^{ij} = (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^j)^\top [S_t^j]^{-1} (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^j), \tag{15.7}$$

is referred to as the *Mahalanobis distance*. We can show the equivalence by noting that in the definition of the Gaussian probability density function:

$$\mathcal{N}(\boldsymbol{z}_t^i \mid \hat{\boldsymbol{z}}_t^j,\ S_t^j) = \eta \exp\Big(-\frac{1}{2} (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^j)^\top [S_t^j]^{-1} (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^j) \Big),$$

that the exponential function is monotonically increasing and that the normalization constant η is positive, therefore we achieve the maximum by maximizing the quadratic term in the exponential.

Adding the maximum likelihood estimation step for the correspondences into the previous EKF localization algorithm in Algorithm 15.2 gives the new EKF location algorithm in Algorithm 15.3.

Algorithm 15.3: EKF Localization, Unknown Correspondences

Data: $\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t, \boldsymbol{m}$
Result: $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$
$\bar{\boldsymbol{\mu}}_t = f(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t)$
$\bar{\boldsymbol{\Sigma}}_t = G_t \boldsymbol{\Sigma}_{t-1} G_t^\top + \boldsymbol{R}_t$
foreach $\boldsymbol{z}_t^i$ **do**
  foreach *landmark k in the map* **do**
    $\hat{\boldsymbol{z}}_t^k = h(\bar{\boldsymbol{\mu}}_t, k, \boldsymbol{m})$
    $S_t^k = H_t^k \bar{\boldsymbol{\Sigma}}_t [H_t^k]^\top + \boldsymbol{Q}_t$
  $j = \arg\min_k \ (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^k)^\top [S_t^k]^{-1} (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^k)$
  $K_t^i = \bar{\boldsymbol{\Sigma}}_t [H_t^j]^\top [S_t^j]^{-1}$
  $\bar{\boldsymbol{\mu}}_t = \bar{\boldsymbol{\mu}}_t + K_t^i (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^j)$
  $\bar{\boldsymbol{\Sigma}}_t = (I - K_t^i H_t^j) \bar{\boldsymbol{\Sigma}}_t$
$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t$
$\boldsymbol{\Sigma}_t = \bar{\boldsymbol{\Sigma}}_t$
return $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$

One of the disadvantages of using the maximum likelihood estimation approach for determining the correspondences is that it can be brittle with respect to outliers and in cases where there are equally likely hypotheses for the correspondence. An alternative approach for estimating correspondences that is more robust to outliers is to use a *validation gate*. In this approach, the smallest Mahalanobis distance, d_t^{ij}, must also pass a *thresholding* test:

$$(\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^j)^\top [S_t^j]^{-1} (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^j) \leq \gamma,$$

in order for a correspondence to be created.

Example 15.4.1 (Differential Drive Robot with Range and Bearing Measurements)**.** Consider a differential drive robot with state $\boldsymbol{x} = [x, y, \theta]^\top$ and a sensor that measures the range, r, and bearing, ϕ, of landmarks $m_j \in \boldsymbol{m}$ relative to the robot's local coordinate frame. Additionally, we assume that we collect multiple measurements corresponding to different features at each time step such that:

$$\boldsymbol{z}_t = \{[r_t^1, \phi_t^1]^\top, [r_t^2, \phi_t^2]^\top, \dots\},$$

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:

$$h(\boldsymbol{x}_t, j, \boldsymbol{m}) = \begin{bmatrix} \sqrt{(m_{j,x} - x)^2 + (m_{j,y} - y)^2} \\ \text{atan2}(m_{j,y} - y, m_{j,x} - x) - \theta \end{bmatrix}. \tag{15.8}$$

The measurement Jacobian, H_t^j, corresponding to a measurement from landmark j is therefore:

$$H_t^j = \begin{bmatrix} -\frac{m_{j,x} - \bar{\mu}_{t,x}}{\sqrt{(m_{j,x} - \bar{\mu}_{t,x})^2 + (m_{j,y} - \bar{\mu}_{t,y})^2}} & -\frac{m_{j,y} - \bar{\mu}_{t,y}}{\sqrt{(m_{j,x} - \bar{\mu}_{t,x})^2 + (m_{j,y} - \bar{\mu}_{t,y})^2}} & 0 \\ \frac{m_{j,y} - \bar{\mu}_{t,y}}{(m_{j,x} - \bar{\mu}_{t,x})^2 + (m_{j,y} - \bar{\mu}_{t,y})^2} & -\frac{m_{j,x} - \bar{\mu}_{t,x}}{(m_{j,x} - \bar{\mu}_{t,x})^2 + (m_{j,y} - \bar{\mu}_{t,y})^2} & -1 \end{bmatrix}. \tag{15.9}$$

It is also common for us 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. In this case, we typically use a diagonal covariance matrix since we can assume these two measurements are uncorrelated.

15.5 Monte Carlo Localization (MCL)

Another Bayesian estimation approach to Markov localization is the Monte Carlo localization (MCL) algorithm. This algorithm leverages the framework of the non-parametric particle filter algorithm from Chapter 14, and is therefore better suited for solving *global pose* localization problems[9] than the EKF localization algorithm.

[9] We can also use MCL to solve the kidnapped robot problem by injecting new randomly sampled particles at each step to ensure the samples don't get too concentrated.

Like the particle filter, the MCL algorithm represents the belief distribution, $\text{bel}(\boldsymbol{x}_t)$, by a set of K particles:

$$\mathcal{P}_t := \{\boldsymbol{x}_t^{[1]}, \boldsymbol{x}_t^{[2]}, ..., \boldsymbol{x}_t^{[K]}\},$$

where each particle, $\boldsymbol{x}_t^{[k]}$, represents a hypothesis about the true state, $\boldsymbol{x}_t$. At each step of the algorithm, we use the state transition model to propagate the particles forward, and then we use the measurement model to resample a new set of particles based on the measurement likelihood. We describe the MCL algorithm in detail in Algorithm 15.4, which we can see is nearly identical to the particle filter algorithm in Algorithm 14.4 except that we use the map, $\boldsymbol{m}$, in the probabilistic state transition and measurement models.

15.6 Exercises

The starter code for the exercises provided below is available online through GitHub. To get started, download the code by running in a terminal window:

Algorithm 15.4: Monte Carlo Localization

Data: $\mathcal{P}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t, \boldsymbol{m}$
Result: $\mathcal{P}_t$
$\bar{\mathcal{P}}_t = \mathcal{P}_t = \emptyset$
for $k = 1$ **to** K **do**
  Sample $\bar{\boldsymbol{x}}_t^{[k]} \sim p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}^{[k]}, \boldsymbol{u}_t, \boldsymbol{m})$
  $w_t^{[k]} = p(\boldsymbol{z}_t \mid \bar{\boldsymbol{x}}_t^{[k]}, \boldsymbol{m})$
  $\bar{\mathcal{P}}_t = \bar{\mathcal{P}}_t \cup (\bar{\boldsymbol{x}}_t^{[k]}, w_t^{[k]})$
for $k = 1$ **to** K **do**
  Draw i with probability $\propto w_t^{[i]}$
  Add $\bar{\boldsymbol{x}}_t^{[i]}$ to $\mathcal{P}_t$
return $particleset_t$

```
git clone https://github.com/StanfordASL/pora-exercises.git
```

We denote Problems requiring hand-written solutions and coding in Python with ✎ and 🖳, respectively.

🖳 *Problem 1: Extended Kalman Filter Localization*

In the file `ch15/exercises/ekf_localization.ipynb`, implement the extended Kalman filter algorithm and then apply the algorithm to a robot localization problem.

🖳 *Problem 2: Particle Filter Localization*

In the file `ch15/exercises/particle_filter_localization.ipynb`, implement the particle filter algorithm and then apply the algorithm to a robot localization problem.

References

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

16
Simultaneous Localization and Mapping (SLAM)

In Chapter 15, we studied robot *localization* under the assumption that a map of the environment, $\boldsymbol{m}$, was known. While this assumption simplifies the estimation process, it is rarely met in practice: robots often operate in *a priori unknown* or partially known environments. Examples include autonomous search-and-rescue in collapsed buildings, planetary exploration, and mapping of underwater structures. In such cases, a robot must concurrently infer its own state and build a map from noisy sensor data.

This joint estimation problem is known as *simultaneous localization and mapping (SLAM)* [1]. SLAM plays a central role in the perception and navigation layers of an autonomy stack. By providing a consistent spatial frame of reference, it enables downstream planning, control, and decision-making components to operate effectively in previously unseen environments.

The SLAM problem emerged in the late 1980s, with seminal contributions by Hugh Durrant-Whyte, John Leonard, and others, who framed it as a probabilistic joint estimation of pose and map. Early approaches treated SLAM as a large filtering problem. The Extended Kalman Filter (EKF) [2] provided a conceptually elegant way to maintai a joint Gaussian distribution over all robot poses and landmarks. While influential, EKF-SLAM quickly revealed two limitations: i) the cost of updating a dense covariance matrix grew quadratically with map size, making it difficult to scale, and ii) repeated linearization of nonlinear dynamics introduced inconsistencies.

In the early 2000s, a breakthrough came with particle-filter methods, most notably FastSLAM [3]. By combining a particle-based representation of the robot's trajectory with separate Kalman filters for landmarks (a technique known as Rao-Blackwellization), FastSLAM improved scalability and handled data association more flexibly. Here, data association refers to the problem of determining which previously mapped landmark, if any, corresponds to each new sensor measurement – an ambiguity that arises when multiple landmarks look similar or when sensing is noisy. However, FastSLAM also struggled with "particle depletion" over long trajectories, which limited robustness in practice.

The field shifted again in the mid-2000s toward an optimization-based view. Rather than updating a filter step by step, researchers formulated SLAM as a

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

[2] Randall Smith, Matthew Self, and Peter Cheeseman. "Estimating uncertain spatial relationships in robotics". In: *Autonomous robot vehicles*. Springer, 1990, pp. 167–193, John J Leonard and Hugh F Durrant-Whyte. "Simultaneous map building and localization for an autonomous mobile robot." In: *IROS*. vol. 3. 1991, pp. 1442–1447

[3] Michael Montemerlo et al. "FastSLAM: A factored solution to the simultaneous localization and mapping problem". In: *Aaai/iaai* 593598.2 (2002), pp. 593–598

sparse nonlinear least-squares problem over a pose graph. This "graph-based SLAM" perspective made it possible to exploit sparsity in the problem, leverage robust cost functions to manage outliers, and apply incremental solvers for real-time operation. The factor graph formalism later generalized this idea, providing a unified way to represent diverse sources of constraints (from odometry to loop closures to landmark observations) in a single probabilistic model.

Recent years have seen multiple extensions in various areas, including Visual(-inertial) SLAM (e.g., PTAM [4]), learning-based SLAM (learned features, differentiable bundle adjustment), Neural implicit maps (e.g., NeRF, Gaussian splatting), and semantic and dynamic SLAM, handling object-level constraints and non-rigid scenes. Modern SLAM combines probabilistic estimation, large-scale optimization, and learned representations, advancing toward the broader vision of spatial AI.

In the remainder of this chapter, we develop the mathematical framework underlying SLAM, beginning with its formalization as a Bayesian state estimation problem. We introduce the state-space models that capture robot motion and sensor measurements, explain how these lead to the online and full SLAM formulations above, and establish the probabilistic factorization that underpins both classical (EKF, FastSLAM) and modern (graph- and factor-based) approaches.

Before delving into mathematical details, it is useful to step back and consider the main algorithmic paradigms that have emerged for SLAM. These provide the conceptual framing for the methods we will develop in the rest of the chapter.

[4] Feng Lu and Evangelos Milios. "Robot pose estimation in unknown environments by matching 2d range scans". In: *Journal of Intelligent and Robotic systems* 18.3 (1997), pp. 249–275, Frank Dellaert and Michael Kaess. "Square root SAM: Simultaneous localization and mapping via square root information smoothing". In: *The International Journal of Robotics Research* 25.12 (2006), pp. 1181–1203, Michael Kaess, Ananth Ranganathan, and Frank Dellaert. "iSAM: Incremental smoothing and mapping". In: *IEEE Transactions on Robotics* 24.6 (2008), pp. 1365–1378, Rainer Kümmerle et al. "g 2 o: A general framework for graph optimization". In: *2011 IEEE international conference on robotics and automation*. IEEE. 2011, pp. 3607–3613

16.1 SLAM Paradigms

A modern SLAM pipeline can be conceptually divided into two parts. The *front-end* extracts constraints from raw sensor data (features, correspondences, loop closures). The *back-end* maintains a consistent estimate of trajectory and map by solving an optimization or filtering problem over these constraints. The next sections will focus on back-end paradigms before returning to front-end design.

The SLAM back-end problem can be approached through several algorithmic paradigms, each shaped by different fidelity and computational priorities. At the highest level, modern SLAM algorithms fall into two broad families: *filter-based* and *smoothing-based* methods. This distinction reflects whether estimation is formulated as a sequential state update problem or as a global optimization over the entire robot trajectory and map.

16.1.1 Filter-based Approaches

Filter-based SLAM maintains a compact state estimate that is updated online as new sensor measurements arrive. The key idea is to propagate a belief over the robot's current state (and possibly the map) using a recursive prediction-

correction cycle. During prediction, the motion model propagates the state distribution forward in time. During correction, the measurement model incorporates the latest observations to refine the estimate. Two classical instances dominate this family:

- *Extended Kalman Filter (EKF) SLAM*, which linearizes motion and measurement models around the current estimate and maintains a joint Gaussian over poses and landmarks.
- *Particle filter-based methods*, such as *FastSLAM*, which factorize the posterior into a distribution over trajectories (represented by particles) and independent landmark estimates.

Filtering naturally supports real-time, online operation with bounded memory requirements, making it attractive for embedded and resource-limited platforms. However, its sequential nature can cause accumulated linearization errors and drift to persist unless additional mechanisms for loop closure are integrated.

16.1.2 Smoothing-based Approaches

Smoothing-based SLAM treats the entire robot trajectory and map as variables in a global optimization problem, leveraging all available measurements simultaneously. The estimation is framed as a nonlinear least-squares problem or equivalently as a maximum a posteriori (MAP) problem, often represented via graph structures:

- *Pose Graph SLAM* models only robot poses as variables connected by relative pose constraints.
- *Factor Graph SLAM* generalizes this by allowing heterogeneous variables (poses, landmarks, calibration parameters) and constraints of arbitrary form.

This global perspective offers several advantages. First, robustness to closures, since relinearization over the whole trajectory can remove drift. Second, flexibility to incorporate heterogeneous sensing modalities. Third, improved accuracy at the cost of higher computational demands.

16.1.3 Choosing a Paradigm

In practice, the choice between filtering and smoothing depends on several factors. First, application constraints, such as computational resources, sensor bandwidth, and latency requirements. Second, environmental characteristics, such as density of loop closures, degree of nonlinearity, and scale. Finally, desired estimation quality, i.e., tolerance for drift vs. need for global consistency. Many contemporary SLAM systems adopt hybrid strategies, leveraging filtering for short-term, high-rate state estimation and periodic smoothing for global consistency.

With this framing in place, we next describe how raw sensor streams are transformed into constraints: the *front-end* of a SLAM system. We then develop the probabilistic foundations that link those constraints to estimation, and finally present representative *back-end* algorithms from each family: EKF-SLAM and FastSLAM (filtering) and pose-/factor-graph SLAM (smoothing), highlighting their strengths, limitations, and typical use cases.

16.2 Front-End

So far, we have focused on back-end formulations: how to structure and solve the estimation problem once constraints are given. In a SLAM system, the *front-end* is the part of the pipeline that takes the raw sensor observations and converts them into constraints, the mathematical relationships that the back-end will later use to optimize poses and map estimates. If we think of the back-end as the brain that reasons about the entire history of the robot's motion and the structure of the environment, the front-end is the perceptual and interpretative apparatus: it decides *what information* the brain receives, *how* that information is structured, and *with what level of certainty*.

16.2.1 Feature Extraction

At its most basic level, the front-end begins by extracting salient information from sensor streams. In visual SLAM, this may involve detecting corners, edges, or learned descriptors that can serve as repeatable landmarks across multiple viewpoints. in LiDAR-based SLAM, geometric primitives such as planes, edges, or surface patches are identified. The quality and stability of these features directly impact the accuracy of the subsequent estimation: poor features lead to weak or drifting contraints, while rich and robust ones enable consistent localization and mapping.

add reference to Katie's section on the above

16.2.2 Data Association

Once features are extracted, the next critical step is data association: deciding whether a feature observed at time t corresponds to a feature observed previously. This step is notoriously challenging, as prceptual aliasing, sensor noise, and environmental changes can all induce incorrect matches. A correct association introduces a valid constraint between poses and landmarks; an incorrect one can catastrophically corrupt the entire estimate. Probabilistic methods, gating strategies, and robust descriptors are employed to mitigate these risks, but the problem remains one of the hardest in SLAM.

16.2.3 Outlier Rejection

Because incorrect associations are inevitable in practice, the front-end must incorporate mechanisms for detecting and rejecting outliers before passing constraints to the back-end. Techniques range from simple geometric consistency checks, to RANSAC-based hypothesis testing, to modern approaches that leverage semantic priors or consistency with the current map. Outlier rejection is not merely a preprocessing step, it is an essential safeguard against divergence of the estimator.

16.2.4 Loop Closure Detection

A defining capability of SLAM systems is loop closure detection: recognizing that the robot has returned to a previously visited location. Loop closures serve to eliminate accumulated drift, effectively "tying together" distant parts of the trajectory. Methods for loop closure range from bag-of-words visual matching, to learned place recognition networks, to scan-context descriptors for LiDAR. Each approach trades off between robustness to environmental change and computational efficiency. Successful loop closure detection is one of the most important contributions of the front-end, as it directly enables globally consistent maps.

16.2.5 Scan Alignment and ICP

In range-based SLAM, and often as a refinement step in vision-based systems, scan alignment plays a central role. ICP and its many variants align overlapping pointclouds to produce relative pose constraints. Although conceptually simple, these approaches are sensitive to initialization and prone to local minima, so modern systems typically use them in combination with feature-based methods or as part of a multi-stage front-end pipeline.

add references to Katie's parts

16.3 Examples by Sensing Modality

Up to this point we have examined the two major components of a SLAM system: the *front-end*, which converts raw sensor observations into constraints, and the *back-end*, which estimates trajectories and maps from those constraints. In practice, the concrete design of each stage is strongly shaped by the *sensing modality*. Although the abstract SLAM framework is sensor-agnostic, the type of sensor determines what features can be extracted, how reliably data can be associated, and what form of constraints are passed to the optimizer. This section surveys representative SLAM pipelines organized by sensing modality, highlighting how sensor characteristics—ranging from appearance-rich images to geometric point clouds and weather-robust radar signals—have driven the

evolution of both front-end and back-end techniques.

16.3.1 Visual SLAM

Cameras are among the most widely used sensors in SLAM due to their low cost, small form factor, and ability to provide rich appearance information. Visual SLAM approaches can be broadly categorized into *monocular*, *stereo*, and *RGB-D* systems, depending on the available input. Each provides a different balance between depth recovery, robustness, and computational cost. From a front-end perspective, visual pipelines rely heavily on feature detection and matching. Local features such as ORB, SIFT, or SURF are extracted from successive frames and associated across time to establish correspondences. These correspondences give rise to geometric constraints such as epipolar relationships, which are used to infer relative pose. To increase robustness, outlier rejection (e.g., RANSAC) and loop closure detection via visual place recognition are essential. Bundle adjustment, which jointly optimizes poses and landmark positions, naturally serves as the back-end optimization engine in this setting.

Representative systems illustrate this progression. ORB-SLAM and its variants have become canonical examples, combining feature-based tracking, loop closure detection, and pose graph optimization into a complete pipeline. VINS-Mono demonstrates how integration with inertial measurements can greatly improve robustness, especially in texture-poor or high-dynamic environments. More recent approaches incorporate direct methods, which minimize photometric error directly on pixel intensities rather than relying on sparse features, thereby exploiting more of the available image information.

The strengths of visual SLAM are clear: high spatial resolution, semantic richness, and ubiquity of sensors. However, limitations arise in challenging lighting conditions, in scenes with strong dynamics, or when depth cannot be reliably inferred (particularly in monocular settings). These challenges have motivated hybrid pipelines, such as visual-inertial SLAM, and learning-based front-ends that attempt to improve data association and depth prediction.

16.3.2 LiDAR SLAM

LiDAR sensors provide direct geometric measurements of the environment in the form of dense or semi-dense point clouds. This modality has become especially important for autonomous vehicles and robotics platforms where accurate metric localization is required in diverse conditions. The front-end of LiDAR SLAM often begins with scan preprocessing (e.g., deskewing, motion compensation) followed by registration. Point cloud registration is frequently performed via ICP and its many robust variants. Local features such as geometric primitives (planes, edges, and corners) are also extracted to create stable correspondences across scans. Loop closure detection can be accomplished by comparing global descriptors of scans, or by embedding geometric signatures into place-recognition networks. On the back-end, pose graph optimization is

the dominant formulation: each scan is treated as a node, with edges encoding relative transformations from ICP or feature-based registration. Robust optimization is critical here, as even small registration errors can accumulate into significant drift over long trajectories. Incremental solvers enable efficient updates in real-time systems, while mapping components fuse successive point clouds into a consistent global map.

Classic systems such as LOAM (LiDAR Odometry and Mapping) demonstrated how tightly-coupled front-end registration and back-end optimization can yield centimeter-level accuracy in real time. More recent work integrates LiDAR with inertial sensing (LIO-SAM, VINS-LIO), leveraging complementary strengths for robustness in fast-motion scenarios.

LiDAR SLAM excels in conditions where cameras struggle, such as low light or high dynamic range scenes. However, limitations include sensor cost, power consumption, and difficulty handling large dynamic environments. Data sparsity at long ranges and sensitivity to weather (e.g., rain, fog) further motivate hybrid pipelines.

16.3.3 Radar SLAM

Radar sensors, once primarily used for automotive collision avoidance, are emerging as valuable tools for SLAM. Their unique strengths lie in robustness to adverse weather and lighting, and long-range detection capability. However, radar signals are noisy, have lower angular resolution, and often include significant multipath effects, making SLAM challenging. The radar SLAM front-end typically relies on feature extraction from radar scans, such as identifying prominent reflectors or learning robust descriptors. Data association is complicated by the high rate of false positives, requiring strong outlier rejection mechanisms. Loop closure detection can be achieved via global scan descriptors or by exploiting the periodicity of radar signatures along revisited paths.

The back-end formulations often resemble those of LiDAR SLAM, with pose graphs or factor graphs constructed from radar-based relative pose estimates. Recent advances include direct radar odometry via learned representations and joint radar-visual pipelines that compensate for radar's low spatial resolution with camera cues.

Systems such as RadarSLAM and more recent radar-inertial odometry frameworks show that robust localization and mapping is possible even in degraded conditions where cameras and LiDAR fail. While still maturing, radar SLAM represents a critical modality for robust autonomy in all-weather, safety-critical applications.

16.3.4 Discussion

Each sensing modality emphasizes a different balance between richness of data, robustness, and environmental adaptability. Cameras provide dense appearance cues but are fragile under lighting changes. LiDAR yields precise geometry at

the cost of expense and sensitivity to adverse weather. Radar offers robustness and range but with sparse, noisy measurements. Increasingly, practical SLAM systems integrate multiple modalities, combining complementary strengths into robust, real-time pipelines. This sensor fusion trend underscores the modularity of the SLAM problem and the importance of a unified view of front-end and back-end design.

Having described how constraints are produced from raw sensor data, we now turn to the probabilistic models that underpin SLAM estimation. These mathematical foundations provide the link between sensor-derived constraints and the algorithms introduced later in the chapter.

16.4 Mathematical foundations of SLAM

Equipped with an understanding of the front-end, we can now formalize the SLAM problem as a Bayesian state estimation problem. This formulation captures the joint evolution of robot state and map, and sets the stage for the algorithmic approaches discussed in the following sections. Formally, given a sequence of control inputs $\boldsymbol{u}_{1:t}$ and sensor measurements $\boldsymbol{z}_{1:t}$, the SLAM problem consists of estimating both a) the robot trajectory $\boldsymbol{x}_{1:t}$ or its current pose $\boldsymbol{x}_t$, and b) the map $\boldsymbol{m}$, which may be represented parametrically (e.g., via landmarks) or non-parametrically (e.g., via occupancy grids). Let $\boldsymbol{x}_t \in \mathbb{R}^n$ denote the robot's state at time t. This typically features its pose (i.e., position and orientation) and may include velocity or other variables related to the motion model. Further, let $\boldsymbol{m}$ denote the map of the environment. Maps may take different forms:

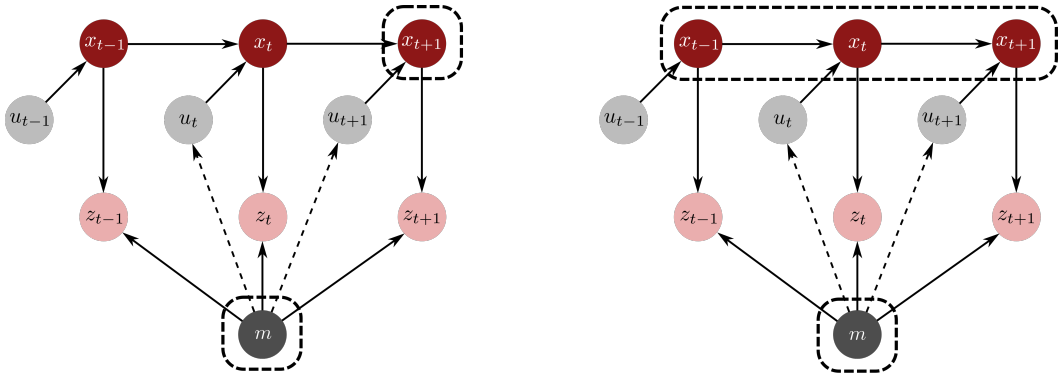
- *Feature-based maps*: $\{m_1, \ldots, m_K\}$, a finite set of landmarks m_k. Each element m_k encodes information that can include position, but may also incorporate additional attributes such as orientation, shape, or appearance descriptors, depending on sensing modality and map representation.
- *Dense or grid maps*: for instance, occupancy grids, which discretize the environment into cells with associated occupancy probabilities, or signed distance fields (often discretized for practical implementation).
- *Hybrid maps*: combining features and dense structures (common in modern SLAM).

In practice, two common formulations of the SLAM problem are distinguished:

- **Online SLAM**, where the goal is to estimate the current robot state together with the map;
- **Full SLAM**, where the goal is to estimate the entire trajectory of the robot together with the map.

To unify notation, we define the *joint state* as:

$$\boldsymbol{y}_t \triangleq (\boldsymbol{x}_t, \boldsymbol{m}) \quad \text{(online SLAM)},$$



Online SLAM

Full SLAM

Figure 16.1: Online SLAM problems estimate only the current robot state together with the map, whereas full SLAM problems estimate the entire trajectory of past robot states (the state history) along with the map.

$$\boldsymbol{y}_{1:t} \triangleq (\boldsymbol{x}_{1:t}, \boldsymbol{m}) \quad \text{(full SLAM)},$$

where $\boldsymbol{x}_{1:t} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t)$ denotes the full state trajectory. In this context, two problem formulations are

- **Online SLAM**: Estimate the belief

$$\text{bel}(\boldsymbol{x}_t, \boldsymbol{m}) = p(\boldsymbol{x}_t, \boldsymbol{m} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}), \tag{16.1}$$

  which focuses on the *current* robot state and the map.

- **Full SLAM**: Estimate the belief

$$\text{bel}(\boldsymbol{x}_{1:t}, \boldsymbol{m}) = p(\boldsymbol{x}_{1:t}, \boldsymbol{m} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}), \tag{16.2}$$

  which recovers the *entire* robot trajectory alongside the map.

The distinction between the two is illustrated in Figure 16.1. In both cases, there is a fundamental coupling: accurate localization depends on map quality, and accurate mapping depends on localization quality. This interplay leads to sensitivity to drift, data association errors, and outliers—issues addressed in modern SLAM via loop closure detection, robust estimation, and global optimization.

16.4.1 Motion and measurement models

At each time step, the robot receives a control input $\boldsymbol{u}_t$ (e.g., wheel velocities, steering commands, thrust forces), and sensor observations $\boldsymbol{z}_t$ (e.g., lidar scans, camera images, range-bearing measurements). The history of data up to time t is $\boldsymbol{u}_{1:t} = (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_t)$ and $\boldsymbol{z}_t = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_t)$, respectively. The robot's motion is modeled by a state transition model

$$\boldsymbol{x}_{t+1} = f(\boldsymbol{x}_t, \boldsymbol{u}_t) + \boldsymbol{w}_t,$$

where $f(\cdot)$ captures the deterministic motion dynamics and $\boldsymbol{w}_t$ is stochastic process noise with distribution $p(\boldsymbol{w}_t)$. Sensor observations follow the measurement model

$$\boldsymbol{z}_t = h(\boldsymbol{x}_t, \boldsymbol{m}) + \boldsymbol{v}_t,$$

where $h(\cdot)$ maps the robot state and map to predicted measurements, and $\boldsymbol{v}_t$ is stochastic measurement noise with distribution $p(\boldsymbol{v}_t)$. Both models may be *linear* or *nonlinear*, depending on the sensing and actuation setup. For instance:

- Linear motion model (odometry): a simple differential-drive robot with small wheel slippage has dynamics that can be approximated as $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + B\boldsymbol{u}_t + \boldsymbol{w}_t$, where B is a constant matrix mapping wheel velocities $\boldsymbol{u}_t$ to pose increments.

- Nonlinear motion model (unicycle): more realistic kinematics can be given by:
$$\begin{aligned} \boldsymbol{x}_{t+1} &= \boldsymbol{x}_t + \boldsymbol{v}_t \cos(\theta_t) + \boldsymbol{w}_t^x, \\ \boldsymbol{y}_{t+1} &= \boldsymbol{y}_t + \boldsymbol{v}_t \sin(\theta_t) + \boldsymbol{w}_t^y, \\ \theta_{t+1} &= \theta_t + \boldsymbol{w}_t + \boldsymbol{w}_t^\theta, \end{aligned}$$
where $\boldsymbol{v}_t, \boldsymbol{w}_t$ are commanded linear and angular velocities.

- Linear measurement model (range sensor in 1D): if a robot moves along a line and measures the distance to a fixed landmark at position m, then $\boldsymbol{z}_t = m - \boldsymbol{x}_t + \boldsymbol{v}_t$, which is linear in $\boldsymbol{x}_t$.

- Nonlinear measurement model (range-bearing in 2D): for a landmark at position (m_x, m_y), one has:
$$\begin{aligned} r_t &= \sqrt{(m_x - \boldsymbol{x}_t)^2 + (m_y - \boldsymbol{y}_t)^2} + \boldsymbol{v}_t^r, \\ \theta_t &= \arctan\left(\frac{m_y - \boldsymbol{y}_t}{m_x - \boldsymbol{x}_t}\right) - \theta_t + \boldsymbol{v}_t^\theta, \end{aligned}$$
which is nonlinear in the robot pose and landmark coordinates.

- Nonlinear measurement model (camera projection): a 3D landmark (X, Y, Z) projects to image coordinates as
$$\begin{bmatrix} u \\ v \end{bmatrix} = \frac{1}{Z} \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + \boldsymbol{v}_t,$$
which is nonlinear due to the division by Z.

16.4.2 Bayesian formulation of SLAM

We can formalize the SLAM problem as a Bayesian state estimation problem, in which both the robot's trajectory and the environment map are unknown and must be estimated jointly from noisy sensor data. Specifically, leveraging Bayes' rule, one has:

$$p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}) \propto p(\boldsymbol{z}_{1:t} \mid \boldsymbol{y}_{1:t}, \boldsymbol{u}_{1:t})\; p(\boldsymbol{y}_{1:t} \mid \boldsymbol{u}_{1:t}),$$

where:

- $p(\boldsymbol{z}_{1:t} \mid \boldsymbol{y}_{1:t}, \boldsymbol{u}_{1:t})$ is the *likelihood*, encoding information from the sensor models;
- $p(\boldsymbol{y}_{1:t} \mid \boldsymbol{u}_{1:t})$ is the *prior*, capturing motion dynamics knowledge and map assumptions.

The difference between online and full SLAM is reflected in whether $\boldsymbol{y}_{1:t}$ includes the entire trajectory or only the current state. In the remainder of this chapter, we present classical solutions, *EKF-SLAM* for online SLAM and *FastSLAM* for full SLAM, before moving on to modern graph-based approaches, robust optimization, and registration methods such as *ICP* and bundle adjustment.

16.4.3 State-space factorization for SLAM

The Bayesian formulation introduced above becomes tractable by leveraging the Markov property of motion and measurement processes: conditioned on the current state, the future is independent of the past. For SLAM, this turns the motion model into

$$p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_{1:t}, \boldsymbol{m}, \boldsymbol{u}_{1:t}) = p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t),$$

meaning that the next state depends only on the current state and the control input. Furthermore, the measurement model adapts to

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{m}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) = p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m})$$

meaning that the current measurement depends only onn the current state and the map. Leveraging these properties, the joint posterior for the full SLAM can be expressed as:

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{m}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}) = p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m}), \tag{16.3}$$

where $p(\boldsymbol{x}_1)$ is the prior over the initial state, and $p(\boldsymbol{m})$ is the prior over the map. This factorization underlies both filtering-based methods (which marginalize past poses to keep $\boldsymbol{x}_t$ only) and smoothing-based methods (which maintain the entire trajectory). In the filtering case, the goal is to maintain a belief distribution over the current state and map,

$$\text{bel}(\boldsymbol{x}_t, \boldsymbol{m}) \triangleq p(\boldsymbol{x}_t, \boldsymbol{m} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}),$$

and update it as new controls and measurements arrive. Applying Bayes' rule together with the Markov properties yields the standard recursion, referred to as *recursive estimation*:

1. *Prediction* (motion update): propagate the belief forward in time using the motion model,

$$\overline{\text{bel}}(\boldsymbol{x}_{t+1}, \boldsymbol{m}) = \int p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t)\, \text{bel}(\boldsymbol{x}_t, \boldsymbol{m})\, d\boldsymbol{x}_t.$$

2. *Correction* (measurement update): refine the belief using the new measurement,

$$\text{bel}(\boldsymbol{x}_{t+1}, \boldsymbol{m}) = \eta p(\boldsymbol{z}_{t+1} \mid \boldsymbol{x}_{t+1}, \boldsymbol{m}) \overline{\text{bel}}(\boldsymbol{x}_{t+1}, \boldsymbol{m}),$$

where η is a normalizing constant ensuring the belief integrates to 1.

This recursion is *exact* in general, but computing it is only tractable for small, discrete problems. In practice, SLAM algorithms rely on approximations (e.g., EKF linearization, particle filters, sparse optimization) to perform the updates efficiently.

This state-space perspective provides the unifying foundation for SLAM algorithms:

- EKF-SLAM implements the recursion directly with Gaussian assumptions and first-order linearization.
- FastSLAM factors the posterior leveraging Rao-Blackwellization, estimating the trajectory with particles and the landmarks with independent EKFs.
- Graph-based SLAM encodes the same factorization in a graphical model, solved as a sparse nonlinear least-squares problem.

16.5 Extended Kalman Filter SLAM

We have already seen two use cases of the extended Kalman filter, first as a general state estimation algorithm in Chapter 14, and then for robot localization given a known map in Chapter 15. In this section, we introduce the *EKF SLAM* algorithm, a natural starting point for implementing the online formulation introduced in Equation (16.1) [5]. In EKF-SLAM, the map is treated as part of an augmented state vector and the joint posterior over robot pose and map is recursively estimated under Gaussian noise assumptions and first-order linearizations of the process and measurement models. This approach generalizes the use of the EKF from the earlier cases of localization and generic recursive state estimation to the simultaneous estimation of a static feature-based map and the evolving robot trajectory.

[5] Randall Smith, Matthew Self, and Peter Cheeseman. "Estimating uncertain spatial relationships in robotics". In: *Autonomous robot vehicles*. Springer, 1990, pp. 167–193, John J Leonard and Hugh F Durrant-Whyte. "Simultaneous map building and localization for an autonomous mobile robot." In: *IROS*. vol. 3. 1991, pp. 1442–1447

As in Chapter 15, we assume that the map is feature-based:

$$\boldsymbol{m} = \{m_1, m_2, \ldots, m_N\},$$

where m_i is the i-th feature with coordinates $\left(m_{i,x}, m_{i,y}\right)$. In this formulation, the joint state vector is

$$\boldsymbol{y}_t := \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{m}, \end{bmatrix}, \tag{16.4}$$

and the goal of the online SLAM problem is now to compute the posterior belief distribution:

$$\text{bel}(\boldsymbol{y}_t) = p(\boldsymbol{x}_t, \boldsymbol{m} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}).$$

We consider a state transition model for the augmented state vector, $\boldsymbol{y}$, of the form

$$\boldsymbol{y}_{t+1} = g(\boldsymbol{y}_t, \boldsymbol{u}_{t+1}) + \boldsymbol{w}_t,$$

with additive Gaussian process noise $\boldsymbol{w}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{R}_t)$. The nonlinear map g is defined as:

$$g(\boldsymbol{y}_{t-1}, \boldsymbol{u}_t) = \begin{bmatrix} f(\boldsymbol{x}_{t-1}, \boldsymbol{u}_t) \\ \boldsymbol{m}_{1,t-1} \\ \vdots \\ \boldsymbol{m}_{N,t-1} \end{bmatrix},$$

where f denotes the robot motion model, and we assume that map feature $m_i \in \boldsymbol{m}$ is static, so its process model is the identity. The process noise covariance is

$$\boldsymbol{R}_t = \begin{bmatrix} \tilde{\boldsymbol{R}}_t & 0 \\ 0 & 0 \end{bmatrix},$$

where $\tilde{\boldsymbol{R}}_t$ is motion model noise covariance for the robot, and the zero blocks reflect the assumption of no process noise for the map features. The Jacobian of the augmented motion model is

$$G_t = \nabla_{\boldsymbol{y}} g(\boldsymbol{y}, \boldsymbol{u}_t)\big|_{\boldsymbol{y}=\boldsymbol{\mu}_{t-1}},$$

that is, the derivative of g with respect to the augmented state, evaluated at the current mean estimate $\boldsymbol{\mu}_{t-1}$ of the posterior. We adopt the same measurement model as in Chapter 15:

$$\boldsymbol{z}_t^i = h(\boldsymbol{y}_t, j) + \boldsymbol{\delta}_t,$$

where $\boldsymbol{\delta}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}_t)$ is zero-mean Gaussian noise and j is the index of the map feature $m_j \in \boldsymbol{m}$ associated with measurement i. The Jacobian of the measurement model is

$$H_t^j = \nabla_{\boldsymbol{y}} h(\boldsymbol{y}, j)\big|_{\boldsymbol{y}=\bar{\boldsymbol{\mu}}_t},$$

that is, the derivative of the measurement function with respect to the augmented state, evaluated at the predicted mean $\bar{\boldsymbol{\mu}}_t$ from the EKF prediction step.

16.5.1 EKF SLAM with Known Correspondences

As in the EKF localization algorithm, we first consider the case where data associations are known. Let $\boldsymbol{c}_t = [c_t^1, \dots]^\top$ denote the correspondence vector, where c_t^i is the index of the map feature associated with measurement $\boldsymbol{z}_t^i$. With known correspondences, the EKF SLAM algorithm in Algorithm 16.1 (see annotations for each block) is nearly identical to the EKF localization algorithm in Algorithm 15.2, except that it operates on the augmented state vector $\boldsymbol{y}$

A typical initialization for the belief $\text{bel}(\boldsymbol{y}_0)$ is:

$$\boldsymbol{\mu}_0 = \begin{bmatrix} \boldsymbol{x}_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \boldsymbol{\Sigma}_0 = \begin{bmatrix} \tilde{\boldsymbol{\Sigma}}_0 & 0 & \cdots & 0 \\ 0 & \infty & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \infty \end{bmatrix},$$

Algorithm 16.1: EKF Online SLAM, Known Correspondences

Data: $\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t, \boldsymbol{c}_t$
Result: $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$
// Prediction step: propagate mean and covariance with motion model
$\bar{\boldsymbol{\mu}}_t = g(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t)$
$\bar{\boldsymbol{\Sigma}}_t = G_t \boldsymbol{\Sigma}_{t-1} G_t^T + \boldsymbol{R}_t$
// Correction step: for each measurement, update state and covariance
foreach $\boldsymbol{z}_t^i$ **do**
  $j = c_t^i$; // index of associated map feature
  if *feature j never seen before* **then**
    Initialize $\begin{bmatrix} \bar{\mu}_{j,x} \\ \bar{\mu}_{j,y} \end{bmatrix}$ as expected position based on $\boldsymbol{z}_t^i$
  // Innovation covariance
  $S_t^i = H_t^j \bar{\boldsymbol{\Sigma}}_t [H_t^j]^T + \boldsymbol{Q}_t$
  // Kalman gain
  $K_t^i = \bar{\boldsymbol{\Sigma}}_t [H_t^j]^T [S_t^i]^{-1}$
  // State update using measurement residual
  $\bar{\boldsymbol{\mu}}_t = \bar{\boldsymbol{\mu}}_t + K_t^i (\boldsymbol{z}_t^i - h(\bar{\boldsymbol{\mu}}_t, j))$
  // Covariance update
  $\bar{\boldsymbol{\Sigma}}_t = (I - K_t^i H_t^j) \bar{\boldsymbol{\Sigma}}_t$
// Final posterior belief
$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t$
$\boldsymbol{\Sigma}_t = \bar{\boldsymbol{\Sigma}}_t$
return $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$

where:

$$\boldsymbol{x}_0 = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \tilde{\boldsymbol{\Sigma}}_0 = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix},$$

and $\boldsymbol{x}_0$ and $\tilde{\boldsymbol{\Sigma}}$ are the initial robot state and associated covariance. The origin is chosen as the map's arbitrary reference frame, placing the robot at the origin with certainty. Feature covariances are initialized with very large values (conceptually "infinite") to represent complete lack of prior knowledge about their locations.

When a feature is first observed, the algorithm re-initializes its mean using the corresponding measurement, thus avoiding linearizing the measurement function about an arbitrary (and potentially poor) initial guess, such as the origin. For instance, with a range-bearing sensor (e.g., lidar or a laser rangefinder), a single measurement (r, φ) provides enough information to triangulate the

landmark's Cartesian coordinates relative to the robot:

$$m_x = \boldsymbol{x}_t + r\cos(\theta_t + \varphi), \quad m_y = \boldsymbol{y}_t + r\sin(\theta_t + \varphi).$$

This makes single-observation initialization valid in such cases. However, the same assumption does not hold for pure-bearing sensors (e.g., monocular cameras without depth estimation), where multiple observations or motion are required to infer landmark positions. A more detailed discussion of these distinctions is provided in Section 16.3.

16.5.2 EKF SLAM with Unknown Correspondences

Handling unknown correspondences in EKF SLAM is considerably more challenging than in the known-map localization problem of Chapter 13. In localization, where the map is given, we can determine measurement-to-feature correspondences via a maximum likelihood procedure and then apply the EKF update to the belief state. In SLAM, the map itself is estimated, so the the maximum likelihood association must rely on *estimated* feature positions. Moreover, the algorithm must detect and incorporate previously unseen features.

What changes compared to the known-correspondence case? In the known-correspondence setting, each measurement $\boldsymbol{z}_t^i$ comes with a fixed index c_t^i identifying the associated map feature. In the unknown-correspondence case, this association must be inferred online. Thus, the algorithm must decide whether a measurement should be linked to an *existing* feature (data association) or treated as a *new* feature (map expansion). This additional decision layer is what fundamentally distinguishes this case from the known-correspondence formulation.

The general approach is:

1. For each measurement $\boldsymbol{z}_t^i$, hypothesize a potential new feature position, increasing the feature count from N_{t-1} to $N_t = N_{t-1} + 1$.

2. For all existing features $k = 1, \ldots, N_t$, we compute the Mahalanobis distance [6]:

   $$d_t^{ik} = (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^k)^\top [\boldsymbol{S}_t^k]^{-1} (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^k),$$

   where $\hat{\boldsymbol{z}}_t^k = h(\bar{\boldsymbol{u}}_t, k)$ is the predicted measurement for feature k and $\boldsymbol{S}_t^k$ is the corresponding innovation covariance. Intuitively, the Mahalanobis distance measures how many "standard deviations" the actual measurement $\boldsymbol{z}_t^i$ lies from the predicted measurement $\hat{\boldsymbol{z}}_t^k$, taking into account correlations in the uncertainty. Unlike the plain Euclidean distance, it automatically down-weights directions of high uncertainty and upweights directions of low uncertainty, making it a natural measure of statistical consistency.

3. If $d_t^{ik} > \alpha$ for all existing features, accept the hypothesized feature as new. The threshold α controls the trade-off between false positives (too small) and missed detections (too large)[7].

[6] Roy De Maesschalck, Delphine Jouan-Rimbaud, and Désiré L Massart. "The mahalanobis distance". In: *Chemometrics and intelligent laboratory systems* 50.1 (2000), pp. 1–18

[7] Choosing α too small may lead to an excessive number of spurious features, increasing computational load; choosing it too large risks missing actual features.

The complete EKF SLAM algorithm for unknown correspondences is summarized in Algorithm 16.2.

Algorithm 16.2: EKF Online SLAM, Unknown Correspondences

Data: $\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t, N_{t-1}$
Result: $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$
$N_t = N_{t-1}$
$\bar{\boldsymbol{\mu}}_t = g(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t)$
$\bar{\boldsymbol{\Sigma}}_t = G_t \boldsymbol{\Sigma}_{t-1} G_t^T + \boldsymbol{R}_t$
foreach $\boldsymbol{z}_t^i$ **do**
  Estimate 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 $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$

Although conceptually straightforward, EKF online SLAM with unknown correspondences is rarely robust in practice. Sensor noise can introduce false features that persist indefinitely. Mitigation strategies include robust outlier rejection and using more distinctive feature descriptors. A further drawback is the quadratic computational complexity in the number of features N, making it challenging to balance runtime and localization accuracy.

Example 16.5.1 (Differential Drive Robot with Range and Bearing Measurements)**.** Consider a differential drive robot with a state consisting of the two-dimensional position and heading, $\boldsymbol{x} = [x, y, \theta]^\top$. Suppose a sensor is available which measures the range, r, and bearing, ϕ to features $m_j \in \boldsymbol{m}$, relative to the robot's local coordinate frame. At each timestep, multiple measurements are collected:

$$\boldsymbol{z}_t = \{[r_t^1, \phi_t^1]^\top, [r_t^2, \phi_t^2]^\top, \dots\},$$

where each measurement $\boldsymbol{z}_t^i = [r_t^i, \phi_t^i]^\top$. For SLAM, define the augmented state:

$$\boldsymbol{y}_t \coloneqq \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{m}_1 \\ \vdots \\ \boldsymbol{m}_N \end{bmatrix} = \begin{bmatrix} x & y & \theta & m_{1,x} & m_{1,y} & \dots & m_{N,x} & m_{N,y} \end{bmatrix}^\top.$$

With known correspondences, the measurement model for feature j is:

$$h(\boldsymbol{y}_t, j) = \begin{bmatrix} \sqrt{(m_{j,x} - x)^2 + (m_{j,y} - y)^2} \\ \text{atan2}(m_{j,y} - y, m_{j,x} - x) - \theta \end{bmatrix}.$$

The associated Jacobian H_t^j, corresponding to a measurement from feature j is:

$$H_t^j = \begin{bmatrix} -\frac{\bar{\mu}_{j,x} - \bar{\mu}_{t,x}}{\sqrt{q_{t,j}}} & -\frac{\bar{\mu}_{j,y} - \bar{\mu}_{t,y}}{\sqrt{q_{t,j}}} & 0 & 0 & \dots & 0 & \frac{\bar{\mu}_{j,x} - \bar{\mu}_{t,x}}{\sqrt{q_{t,j}}} & \frac{\bar{\mu}_{j,y} - \bar{\mu}_{t,y}}{\sqrt{q_{t,j}}} & 0 & \dots \\ \frac{\bar{\mu}_{j,y} - \bar{\mu}_{t,y}}{q_{t,j}} & -\frac{\bar{\mu}_{j,x} - \bar{\mu}_{t,x}}{q_{t,j}} & -1 & 0 & \dots & 0 & -\frac{\bar{\mu}_{j,y} - \bar{\mu}_{t,y}}{q_{t,j}} & \frac{\bar{\mu}_{j,x} - \bar{\mu}_{t,x}}{q_{t,j}} & 0 & \dots \end{bmatrix},$$

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 feature m_j from $\bar{\boldsymbol{\mu}}_t$. With both a range and bearing measurement, we compute the *estimated* position of feature m_j by:

$$\begin{bmatrix} \bar{\mu}_{j,x} \\ \bar{\mu}_{j,y} \end{bmatrix} = \begin{bmatrix} \bar{\mu}_{t,x} \\ \bar{\mu}_{j,y} \end{bmatrix} + \begin{bmatrix} r_t^i \cos(\phi_t^i + \bar{\mu}_{t,\theta}) \\ r_t^i \sin(\phi_t^i + \bar{\mu}_{t,\theta}) \end{bmatrix},$$

which we can use in the known-correspondence EKF SLAM algorithm in Algorithm 16.1 to initialize the feature position, and in the unknown-correspondence case in Algorithm 16.2 to hypothesize the position of new features.

While EKF-SLAM provides a principled probabilistic framework for joint pose and map estimation, its computational and memory requirements scale quadratically with the number of features, and its reliance on Gaussian assumptions and linearization can limit performance in large or highly nonlinear environments. Such limitations have motivated the development of more scalable and flexible approaches, such as particle filter-based methods, which we discuss next.

16.6 Particle Filter-Based SLAM

The SLAM problem can also be addressed within the framework of nonparametric particle filters. A major advantage of this approach is that it can be extended to solve the full SLAM problem, estimating the joint posterior $p(\boldsymbol{x}_{1:t}, \boldsymbol{m} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t})$, which includes both the map $\boldsymbol{m}$ and the full trajectory $\boldsymbol{x}_{1:t}$ of the robot. This contrasts with EKF-SLAM, which addresses only the online SLAM problem (estimating the current pose and map).

A naive implementation would resemble Monte Carlo localization from Chapter 15, but with the augmented state vector $\boldsymbol{y}$ from Equation (16.4) including the entire map. However, such an approach is computationally infeasible: the number of particles required to represent the belief grows exponentially with the dimensionality of the state space, and the map may contain a large number of features. A key insight avoids this curse of dimensionality: given the full robot path and known correspondences, the locations of individual map features are *conditionally independent*. Formally, the SLAM posterior over the augmented state $\boldsymbol{y}_{1:t} = (\boldsymbol{x}_{1:t}, \boldsymbol{m})$ can be factorized as

$$p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) = p(\boldsymbol{x}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) \prod_{i=1}^{N} p(m_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}), \qquad (16.5)$$

which we derive in more detail in Equation (16.6).

This factorization separates the posterior into

- a *path posterior* $p(\boldsymbol{x}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t})$ over robot trajectories, handled by a particle filter, and
- *feature posteriors* $p(m_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t})$ for each map element, handled independently (e.g., via EKFs).

This decomposition drastically reduces the effective state dimension that particles must represent, enabling efficient particle-based SLAM in large maps. As with other particle-based approaches, Particle SLAM can a) handle nonlinear process and measurement models without linearization, b) represent multi-modal belief distributions, and c) avoid explicit Jacobian derivations. Its drawbacks mirror those discussed earlier: particle methods may require large numbers of samples to avoid degeneracy in high-dimensional spaces, and performance depends on careful proposal design and resampling strategies.

16.6.1 Factoring the Posterior

Let the full augmented state be $\boldsymbol{y}_{1:t} = (\boldsymbol{x}_{1:t}, \boldsymbol{m})$ and assume known correspondences $c_{1:t}$ and a single measurement per time step. The key insight is that we can factor the posterior as:

$$p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) = p(\boldsymbol{x}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) \prod_{i=1}^{N} p(m_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}), \qquad (16.6)$$

where m_i is the i-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 $p(m_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t})$ are referred to as the *feature posteriors*. We derive this factored form by first using Bayes' rule to express the posterior, $p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t})$, as:

$$p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) = p(\boldsymbol{x}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}).$$

Next, we note that since the feature posterior, $p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t})$, is conditioned on $\boldsymbol{x}_{1:t}$, the dependence on $\boldsymbol{u}_{1:t}$ is redundant such that we can simplify:

$$p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) = p(\boldsymbol{x}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}).$$

Next, we turn our attention to the feature posterior, $p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t})$, and consider whether or not the feature m_i is observed in the latest measurement, $\boldsymbol{z}_t$, which we denote as $i = c_t$ if it was observed and $i \neq c_t$ if it was not observed. Under these two cases, we write the feature posterior for m_i as:

$$p(m_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}) = \begin{cases} p(m_i \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}), & i \neq c_t, \\ \frac{p(\boldsymbol{z}_t | m_i, \boldsymbol{x}_t, c_t) p(\boldsymbol{m}_i | \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1})}{p(\boldsymbol{z}_t | \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t-1}, c_{1:t})}, & i = c_t, \end{cases}$$

where the first case follows from the most recent measurement having no effect on unrelated features, and the second case is obtained by applying Bayes' rule together with the conditional independence of features given the trajectory (so that other features can be factored out of the likelihood). For the case of $i = c_t$, we can therefore write the probability of the observed feature as:

$$p(m_{c_t} \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}) = \frac{p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t-1}, c_{1:t}) p(m_{c_t} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t})}{p(\boldsymbol{z}_t \mid m_{c_t}, \boldsymbol{x}_t, c_t)}.$$

We can now show that the factorization in Equation (16.6) holds by induction. First, suppose that the feature posterior at the previous time, $t-1$, is factored as[8]:

$$p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}) = \prod_{i=1}^{N} p(m_i \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}).$$

[8] This assumption is true at the first time step because there is not yet any information about any features.

Then, using Bayes' rule and the conditional independence of features given the robot path, the posterior at time t can be written as

$$\begin{aligned} p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}) &= \frac{p(\boldsymbol{z}_t \mid \boldsymbol{m}, \boldsymbol{x}_t, c_t) p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1})}{p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t-1}, c_{1:t})}, \\ &= \frac{p(\boldsymbol{z}_t \mid m_{c_t}, \boldsymbol{x}_t, c_t)}{p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t-1}, c_{1:t})} \prod_{i=1}^{N} p(m_i \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}). \end{aligned}$$

Next, we apply the analysis of $p(m_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{c}_{1:t})$ above for the cases where $i \neq c_t$ and $i = c_t$ to get:

$$\begin{aligned} p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}) &= \frac{p(\boldsymbol{z}_t \mid m_{c_t}, \boldsymbol{x}_t, c_t)}{p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t-1}, c_{1:t})} p(m_{c_t} \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}) \prod_{i \neq c_t} p(m_i \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}), \\ &= p(m_{c_t} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{c}_{1:t}) \prod_{i \neq c_t} p(m_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{c}_{1:t}), \\ &= \prod_{n=1}^{N} p(m_n \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{c}_{1:t}), \end{aligned}$$

which shows the correctness of the factorization.

16.6.2 *FastSLAM with Known Correspondences*

Now that we have shown that we can factor the posterior distribution, $p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t})$, according to Equation (16.6), we introduce *FastSLAM*, the particle SLAM algorithm which exploits this factorization for computational efficiency. Specifically, FastSLAM estimates the path posterior, $p(\boldsymbol{x}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t})$, using a particle filter and estimates each feature posterior, $p(m_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t})$, by an EKF conditioned on the robot path, $\boldsymbol{x}_{1:t}$. This reduces the size of the state space that the particles must represent to only the size of the robot's state, which is generally significantly smaller than the number of potential features in the map, and therefore helps avoid the curse of dimensionality that can plague the particle filter approach.

For this factorization, the set of particles that FastSLAM uses is:

$$\mathcal{P}_t := \{P_t^{[1]}, P_t^{[2]}, \ldots, P_t^{[K]}\},$$

where the k-th particle is:

$$P_t^{[k]} := \{\boldsymbol{x}_t^{[k]}, \boldsymbol{\mu}_{1,t}^{[k]}, \boldsymbol{\Sigma}_{1,t}^{[k]}, \ldots, \boldsymbol{\mu}_{N,t}^{[k]}, \boldsymbol{\Sigma}_{N,t}^{[k]}\},$$

with $\boldsymbol{x}_t^{[k]}$ a hypothesis of the robot state, and $(\boldsymbol{\mu}_{i,t}^{[k]}, \boldsymbol{\Sigma}_{i,t}^{[k]})$ the EKF mean and covariance for feature m_i under that trajectory hypothesis. Note that each particle contains the parameters for an EKF for each feature since, from the factorization, the posterior over the map feature is conditioned on the state. Thus, each particle carries its own local map estimate, and with K particles there are NK EKFs in total.

Algorithmic structure. The FastSLAM recursion proceeds in blocks, blending elements of particle filters with EKF updates:

1. **Prediction (motion update).** For each particle, sample a new robot pose $\boldsymbol{x}_t^{[k]}$ from the state transition model given control input $\boldsymbol{u}_t$:

$$\boldsymbol{x}_t^{[k]} \sim p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}^{[k]}, \boldsymbol{u}_t).$$

2. **Feature update (measurement correction).** For the observed feature $j = c_t$, update the EKF mean and covariance in each particle:

$$\hat{\boldsymbol{z}}^{[k]} = h(\boldsymbol{\mu}_{j,t-1}^{[k]}, \boldsymbol{x}_t^{[k]}),$$

$$S = H^j \boldsymbol{\Sigma}_{j,t-1}^{[k]} [H^j]^\top + Q_t,$$

$$K = \boldsymbol{\Sigma}_{j,t-1}^{[k]} [H^j]^\top S^{-1},$$

$$\boldsymbol{\mu}_{j,t}^{[k]} = \boldsymbol{\mu}_{j,t-1}^{[k]} + K(\boldsymbol{z}_t - \hat{\boldsymbol{z}}^{[k]}), \quad \boldsymbol{\Sigma}_{j,t}^{[k]} = (I - KH^j)\boldsymbol{\Sigma}_{j,t-1}^{[k]}.$$

3. **Weighting.** Each particle receives a weight $w^{[k]}$ proportional to the likelihood of the observation under its map estimate:

$$w^{[k]} \propto \exp\Big(-\tfrac{1}{2}(z_t - \hat{z}^{[k]})^\top S^{-1}(z_t - \hat{z}^{[k]})\Big).$$

4. **Copying unchanged features.** For all $n \neq c_t$, propagate map features unchanged:

$$\boldsymbol{\mu}_{n,t}^{[k]} = \boldsymbol{\mu}_{n,t-1}^{[k]}, \quad \boldsymbol{\Sigma}_{n,t}^{[k]} = \boldsymbol{\Sigma}_{n,t-1}^{[k]}.$$

5. **Resampling.** Draw a new particle set $\mathcal{P}_t$ by resampling from the weighted particles, favoring those that explain the measurements well.

We outline this procedure in Algorithm 16.3. It is important to see how FastSLAM combines two ideas: (1) the *particle filter* is used to track the robot trajectory distribution, while (2) the *EKF updates* are used to maintain Gaussian feature posteriors inside each particle.

Unknown correspondences. So far, we have assumed that correspondences c_t are given. In practice, FastSLAM can also be extended to handle unknown correspondences: within each particle, one can evaluate the likelihood of the observation under multiple feature hypotheses (e.g., using the Mahalanobis distance as in EKF-SLAM) and update the feature set accordingly. This increases robustness but also raises complexity, since each particle must solve its own data association problem.

16.7 Graph SLAM

A particularly important specialization of the general SLAM formulation arises when the environment is represented implicitly through relative pose measurements between robot states. In *pose graph SLAM*, the variables are restricted to the sequence of robot poses $\boldsymbol{x}_{0:T}$, while landmarks (if any) are marginalized out. Each measurement constrains a pair of poses, typically corresponding to odometry or loop closure observations, and can be written as:

$$\boldsymbol{z}_{ij} = h_{ij}\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) + \boldsymbol{v}_{ij},$$

where $\boldsymbol{z}_{ij}$ is the relative pose measurement from pose i to pose j, h_{ij} is the measurement function (often composition of relative transformations), and $\boldsymbol{v}_{ij}$ is zero-mean noise with known covariance. The maximum a posteriori (MAP) estimate of the poses is obtained by minimizing the sum of squared, information-weighted residuals:

$$\boldsymbol{x}_{0:T}^\star = \arg\min_{\boldsymbol{x}_{0:T}} \sum_{(i,j)\in\mathcal{E}} \left\|\boldsymbol{r}_{ij}(\boldsymbol{x}_i, \boldsymbol{x}_j)\right\|_{\blacksquare_{ij}}^2,$$

where $\mathcal{E}$ is the set of edges in the graph, $\blacksquare_{ij}$ is the *information matrix* of $\boldsymbol{z}_{ij}$, and

$$\boldsymbol{r}_{ij}(\boldsymbol{x}_i, \boldsymbol{x}_j) \triangleq \boldsymbol{z}_{ij} \ominus h_{ij}(\boldsymbol{x}_i, \boldsymbol{x}_j),$$

is the residual. The information matrix $\blacksquare_{ij}$ is defined as the inverse of the measurement covariance, $\blacksquare_{ij} = \boldsymbol{Q}_{ij}^{-1}$. It encodes the confidence we have in the measurement: directions of high variance in $\boldsymbol{Q}_{ij}$ correspond to low confidence (small entries in $\blacksquare_{ij}$), while directions of low variance correspond to high confidence (large entries). The weighted norm $\|\boldsymbol{r}\|_{\blacksquare}^2 = \boldsymbol{r}^\top \blacksquare \boldsymbol{r}$ therefore penalizes residuals more strongly in directions where the sensor is reliable.

The operator $\ominus$ denotes the relative pose difference on $SE(2)$ or $SE(3)$. Given two poses $\boldsymbol{x}_a, \boldsymbol{x}_b \in SE(3)$, the expression

$$\boldsymbol{x}_a \ominus \boldsymbol{x}_b = \mathrm{Log}\big(\boldsymbol{x}_a^{-1} \boldsymbol{x}_b\big),$$

maps the transformation from $\boldsymbol{x}_a$ to $\boldsymbol{x}_b$ into the tangent space (a vector in $\mathbb{R}^3$ for $SE(2)$ or $\mathbb{R}^6$ for $SE(3)$). This ensures that residuals are computed in a linear vector space, while still respecting the underlying geometry of rotations and translations.

Linearization and Jacobians. Because the residuals are nonlinear in $\boldsymbol{x}_i, \boldsymbol{x}_j$, iterative optimization is required. At each iteration, the residuals are linearized around the current estimate. Denote the Jacobians of $\boldsymbol{r}_{ij}$ as

$$\boldsymbol{A}_i = \frac{\partial \boldsymbol{r}_{ij}}{\partial \boldsymbol{x}_i}, \quad \boldsymbol{A}_j = \frac{\partial \boldsymbol{r}_{ij}}{\partial \boldsymbol{x}_j}.$$

These Jacobians encode how the residual changes with respect to perturbations in the connected poses.

Normal equations and sparsity. The linearized least-squares problem leads to the *sparse* normal equations:

$$\boldsymbol{H}\, \Delta \boldsymbol{x} = \boldsymbol{b},$$

where $\Delta \boldsymbol{x}$ is the pose increment. Here $\boldsymbol{H}$ is the *global information (Hessian) matrix*, obtained by summing Jacobian and information contributions from all edges:

$$\boldsymbol{H} = \sum_{(i,j)\in\mathcal{E}} J_{ij}^\top\, \Omega_{ij}\, J_{ij},$$

and $\boldsymbol{b}$ is the corresponding gradient vector,

$$\boldsymbol{b} = \sum_{(i,j)\in\mathcal{E}} J_{ij}^\top\, \Omega_{ij}\, r_{ij}.$$

Only the blocks of $\boldsymbol{H}$ corresponding to poses directly connected by a measurement are nonzero, which reflects the locality of sensing and allows the use of sparse linear algebra.

Robust kernels and priors. In practice, two modifications are typically required. *Robust kernels.* Not all measurements are reliable: for instance, an incorrect loop closure may introduce a residual that is inconsistent with the rest of the

graph. To reduce their impact, the squared error term $\|r_{ij}\|^2_{\Omega_{ij}}$ can be replaced with a robust loss $\rho(\|r_{ij}\|^2_{\Omega_{ij}})$, where $\rho(\cdot)$ grows more slowly than the quadratic function (e.g., Huber or Tukey kernels). This means that small residuals are treated normally, while very large residuals are "downweighted," preventing a single outlier from dominating the solution.

Priors / gauge constraints. The optimization problem is also underdetermined: without additional information, the entire solution can be shifted or rotated without changing the relative errors. To remove this ambiguity, we add a prior factor on the first pose (e.g., fixing it at the origin with small covariance), or equivalently introduce a gauge constraint. This anchors the graph in a global reference frame, ensuring a unique and well-posed solution.

Retraction on the manifold. The linear system provides increments $\Delta \boldsymbol{x}_i$ that live in the *tangent space* of the pose manifold, i.e., a local linear approximation of $SE(2)$ or $SE(3)$. Since robot poses themselves must remain valid rigid-body transformations, we cannot update them by simple vector addition. Instead, each pose is updated using the $\oplus$ operator:

$$\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i \oplus \Delta \boldsymbol{x}_i,$$

where $\oplus$ denotes a *retraction*: it maps the tangent increment $\Delta \boldsymbol{x}_i \in \mathbb{R}^3$ (for $SE(2)$) or $\mathbb{R}^6$ (for $SE(3)$) back onto the manifold.

Concretely, this is often implemented via the exponential map of the Lie group:

$$\boldsymbol{x}_i \oplus \Delta \boldsymbol{x}_i \;=\; \boldsymbol{x}_i \cdot \mathrm{Exp}(\Delta \boldsymbol{x}_i),$$

so that translations and rotations remain properly represented as rigid transformations. This guarantees that the updated poses remain on the manifold $SE(2)$ or $SE(3)$, rather than drifting off into the linearized space.

Summary. The resulting optimization procedure is a batch Gauss-Newton or Levenberg-Marquardt algorithm applied to the pose graph. Loop closures appear as edges that connect non-consecutive nodes, providing the essential constraints to correct accumulated drift. The sparsity of the information matrix enables scalable solvers that can handle large-scale graphs with thousands of poses.

While pose graph SLAM is most commonly used for robot-centric mapping when only relative pose constraints are needed, the same formulation can be extended to multi-robot scenarios (with inter-robot relative pose edges) or to hybrid representations where selected landmarks are retained as additional variables.

The next section generalizes this concept to *factor graphs*, which provide a more flexible and modular representation for SLAM, accommodating heterogeneous measurements and variables beyond robot poses.

16.8 Factor Graph SLAM

While pose graph SLAM models only robot poses connected by relative-pose constraints, many real-world scenarios involve additional unknowns: explicit landmark positions, extrinsic calibration between sensors, time-varying biases, or even semantic object labels. *Factor graph SLAM* generalizes the pose graph formulation by casting SLAM as a probabilistic graphical model in which arbitrary variables can be constrained by heterogeneous measurements. This perspective has become the standard abstraction for modern SLAM back-ends.

16.8.1 Motivation

Pose graph SLAM is elegant but restrictive: it assumes every measurement can be reduced to a relative pose between two robot states. In practice, this abstraction hides important structure:

- Landmarks cannot be explicitly represented and refined.
- Sensor calibration parameters (e.g., camera intrinsics, lidar-IMU extrinsics) must be estimated separately.
- Multi-modal sensing introduces higher-order constraints involving more than two variables at once.

Factor graphs overcome these limitations by embedding SLAM into the broader framework of graphical models. Each measurement becomes a *factor*, i.e., a local probabilistic relation between the subset of variables it touches. This modularity allows heterogeneous information to be fused consistently within one optimization problem.

16.8.2 Formulation

In a factor graph, nodes represent *variables* to be estimated:

$$\begin{aligned} X &= \{\boldsymbol{x}_0, \boldsymbol{x}_1, \dots, \boldsymbol{x}_T\} && \text{robot poses,} \\ L &= \{\mathbf{l}_1, \dots, \mathbf{l}_M\} && \text{landmarks,} \\ \Theta &= \{\boldsymbol{\theta}_1, \dots\} && \text{sensor parameters, biases, etc.} \end{aligned}$$

Edges represent *factors*, each encoding the likelihood of a measurement $\boldsymbol{z}_k$ given the variables Y_k it depends on:

$$p(X, L, \Theta \mid Z) \propto \prod_k \phi_k(Y_k).$$

Example. Suppose a stereo camera at pose x_t observes a landmark l_j. The measurement depends not only on the robot pose and landmark coordinates, but also on the stereo calibration parameters θ. This induces a ternary factor

$$\phi(x_t, l_j, \theta) \propto p(z_{t,j} \mid x_t, l_j, \theta),$$

linking three variables simultaneously. Such higher-order constraints cannot be represented in a pure pose graph, but arise naturally in the factor graph formulation.

16.8.3 Residuals and Linearization

For each factor we define a residual

$$\boldsymbol{r}_k(Y_k) = \boldsymbol{z}_k \ominus h_k(Y_k),$$

where $h_k(\cdot)$ is the measurement function and $\ominus$ denotes subtraction in the appropriate tangent space. Linearization around the current estimate introduces the Jacobian

$$\boldsymbol{J}_k = \nabla_{Y_k} \boldsymbol{r}_k(Y_k).$$

Linearized system. The MAP estimation problem reduces to nonlinear least squares over all residuals. At each iteration, linearization yields the sparse normal equations

$$\boldsymbol{H}\,\Delta Y = \boldsymbol{b},$$

with contributions from each factor,

$$\boldsymbol{H} \mathrel{+}= \boldsymbol{J}_k^\top \boldsymbol{\Omega}_k \boldsymbol{J}_k, \qquad \boldsymbol{b} \mathrel{+}= \boldsymbol{J}_k^\top \boldsymbol{\Omega}_k \boldsymbol{r}_k,$$

where $\boldsymbol{\Omega}_k$ is the information matrix of measurement k. Sparsity arises because each factor touches only a small subset of variables, which is the key to scalability.

Robust kernels and priors. As in pose graph SLAM, robust kernels $\rho(\cdot)$ can be applied to downweight large residuals from outliers. Gauge freedom is removed by including priors as additional factors, e.g., anchoring the first pose.

Schur complement. When landmarks are numerous, it is common to marginalize them analytically via the Schur complement. This reduces the system to one over poses and calibration parameters while preserving the information conveyed by landmark observations.

Manifold retraction. After solving for increments ΔY, variables are updated using the $\oplus$ operator:

$$y \leftarrow y \oplus \Delta y,$$

which retracts from the tangent space back onto the appropriate manifold (e.g., $\mathrm{Exp}(\cdot)$ on $SE(3)$ for poses, direct addition in $\mathbb{R}^3$ for landmarks).

16.8.4 Algorithmic structure

16.8.5 Alternatives and Variants

While Gauss-Newton and Levenberg-Marquardt are the standard batch solvers, large-scale problems often rely on iterative linear methods such as preconditioned conjugate gradient. For online operation, incremental solvers such as iSAM and iSAM2 update only a subset of variables upon receiving new factors, achieving real-time performance. Other approaches explore convex relaxations, stochastic gradient methods, or sampling-based inference, though these remain less common in mainstream SLAM systems.

16.8.6 Practical Impact

The factor graph formalism has three defining advantages:

- **Modularity**: new sensor models or constraints can be added as factors without altering the solver.
- **Scalability**: sparsity in the factor graph yields sparse Jacobians and Hessians, enabling efficient large-scale optimization.
- **Incrementality**: compatibility with incremental solvers allows real-time updates.

For these reasons, factor graphs underpin nearly all modern SLAM back-ends. Widely used libraries such as `GTSAM`, `g2o`, and `Ceres Solver` implement this framework and form the backbone of current research and deployed robotic systems.

16.9 Advanced and Emerging Methods

While classical SLAM pipelines rely on geometric features and optimization-based back-ends, the field has rapidly expanded to incorporate learning-based techniques, richer scene representations, and integration with broader AI systems. These methods aim not only to improve accuracy and robustness, but also to enable SLAM to function in environments and applications beyond the reach of purely geometric methods.

16.9.1 Deep Learning in SLAM

Learning has been used to improve both the front-end and back-end of SLAM. On the front-end, convolutional and transformer-based networks provide robust feature detection, semantic segmentation, and depth estimation, even in challenging lighting or texture-poor settings. On the back-end, learned priors can regularize optimization, improve loop-closure detection, or predict uncertainty

in sensor data. End-to-end "neural SLAM" systems attempt to replace hand-engineered pipelines entirely, although their generalization and interpretability remain active research challenges.

16.9.2 Neural Implicit Maps

Traditional mapping approaches (occupancy grids, point clouds, mesh reconstructions) scale poorly or lack continuity. Neural implict representations, such as signed distance functions or neural radiance fields, provide compact, continuous encodings of geometry and appearance. These maps can be queried at arbitrary resolution, fused across time, and potentially shared among multiple agents. While computationally demanding, implicit maps point to a new paradigm where SLAM outputs not just geometry, but a photorealistic and semantically enriched digital twin of the environment.

16.9.3 Semantic and Dynamic SLAM

Classic SLAM often assumes static scenes, but real-world environments are dynamic and cluttered with moving agents. Semantic SLAM augments the map with object-level labels, enabling robots to recognize and reason about doors, vehicles, or furniture rather anonymous landmarks. Dynamic SLAM explicitly models moving objects, separating them from the static background and in some cases tracking them jointly with the ego-motion. Such capabilities open the door to task-driven autonomy, where maps support higher-level reasoning and interaction.

16.9.4 Toward Spatial AI

Looking forward, SLAM is evolving beyond trajectory estimation and mapping toward a broader concept sometimes referred to as *Spatial AI*. Here, geometry, semantics, and temporal dynamics are tightly integrated into a coherent representation that supports decision-making, planning, and interaction. Rather than being a self-contained module, SLAM becomes part of a larger perception-and-action loop, one that enables robots to act intelligently in complex, dynamic worlds.

Connect with Katie

16.10 Exercises

The starter code for the exercises provided below is available online through GitHub. To get started, download the code by running in a terminal window:

```
git clone https://github.com/StanfordASL/pora-exercises.git
```

We denote Problems requiring hand-written solutions and coding in Python with ✎ and 🖳, respectively.

🖳 *Problem 1: EKF SLAM*

In the file `ch16/exercises/ekf_slam.ipynb`, implement the extended Kalman filter SLAM algorithm and then apply the algorithm to a robot SLAM problem.

References

[10] Roy De Maesschalck, Delphine Jouan-Rimbaud, and Désiré L Massart. "The mahalanobis distance". In: *Chemometrics and intelligent laboratory systems* 50.1 (2000), pp. 1–18.

[11] Frank Dellaert and Michael Kaess. "Square root SAM: Simultaneous localization and mapping via square root information smoothing". In: *The International Journal of Robotics Research* 25.12 (2006), pp. 1181–1203.

[27] Michael Kaess, Ananth Ranganathan, and Frank Dellaert. "iSAM: Incremental smoothing and mapping". In: *IEEE Transactions on Robotics* 24.6 (2008), pp. 1365–1378.

[34] Rainer Kümmerle et al. "g 2 o: A general framework for graph optimization". In: *2011 IEEE international conference on robotics and automation*. IEEE. 2011, pp. 3607–3613.

[37] John J Leonard and Hugh F Durrant-Whyte. "Simultaneous map building and localization for an autonomous mobile robot." In: *IROS*. Vol. 3. 1991, pp. 1442–1447.

[43] Feng Lu and Evangelos Milios. "Robot pose estimation in unknown environments by matching 2d range scans". In: *Journal of Intelligent and Robotic systems* 18.3 (1997), pp. 249–275.

[46] Michael Montemerlo et al. "FastSLAM: A factored solution to the simultaneous localization and mapping problem". In: *Aaai/iaai* 593598.2 (2002), pp. 593–598.

[68] Randall Smith, Matthew Self, and Peter Cheeseman. "Estimating uncertain spatial relationships in robotics". In: *Autonomous robot vehicles*. Springer, 1990, pp. 167–193.

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

Algorithm 16.3: FastSLAM

Data: $\mathcal{P}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t, c_t$
Result: $\mathcal{P}_t$
for $k = 1$ **to** K **do**
  `// Prediction: sample new robot pose`
  Sample $\boldsymbol{x}_t^{[k]} \sim p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}^{[k]}, \boldsymbol{u}_t)$
  `// Measurement update for observed feature`
  $j = c_t$
  if *feature j never seen before* **then**
    Initialize feature: $(\boldsymbol{\mu}_{j,t-1}^{[k]}, \boldsymbol{\Sigma}_{j,t-1}^{[k]})$
  else
    $\hat{\boldsymbol{z}}^{[k]} = h(\boldsymbol{\mu}_{j,t-1}^{[k]}, \boldsymbol{x}_t^{[k]})$
    $S = H^j \boldsymbol{\Sigma}_{j,t-1}^{[k]} [H^j]^T + \boldsymbol{Q}_t$
    $K = \boldsymbol{\Sigma}_{j,t-1}^{[k]} [H^j]^T [S]^{-1}$
    $\boldsymbol{\mu}_{j,t}^{[k]} = \boldsymbol{\mu}_{j,t-1}^{[k]} + K(\boldsymbol{z}_t - \hat{\boldsymbol{z}}^{[k]})$
    $\boldsymbol{\Sigma}_{j,t}^{[k]} = (I - KH^j)\boldsymbol{\Sigma}_{j,t-1}^{[k]}$
    `// Weighting: compute importance weight`
    $w^{[k]} = \det(2\pi S)^{-1/2} \exp\Big(-\frac{1}{2}(\boldsymbol{z}_t - \hat{\boldsymbol{z}}^{[k]})^\top S^{-1} (\boldsymbol{z}_t - \hat{\boldsymbol{z}}^{[k]})\Big)$
  `// Copy unchanged features`
  for $n \in \{1, \ldots, N\}$, $n \neq c_t$ **do**
    $\boldsymbol{\mu}_{n,t}^{[k]} = \boldsymbol{\mu}_{n,t-1}^{[k]}$
    $\boldsymbol{\Sigma}_{n,t}^{[k]} = \boldsymbol{\Sigma}_{n,t-1}^{[k]}$
`// Resampling: select new particle set according to weights`
$\mathcal{P}_t = \emptyset$
for $i = 1$ **to** K **do**
  Draw k with probability $\propto w_t^{[i]}$
  $\mathcal{P}_t = \mathcal{P}_t \cup (\boldsymbol{x}_t^{[k]}, \boldsymbol{\mu}_{1,t}^{[k]}, \boldsymbol{\Sigma}_{1,t}^{[k]}, \ldots, \boldsymbol{\mu}_{N,t}^{[k]}, \boldsymbol{\Sigma}_{N,t}^{[k]})$
return $\mathcal{P}_t$

Algorithm 16.4: Pose-Graph (GraphSLAM) – Batch Gauss–Newton / Levenberg–Marquardt

Data: Initial poses $\boldsymbol{x}_{0:T}^{(0)}$ (e.g., odometry), edge set $\mathcal{E}$ with measurements $\{\boldsymbol{z}_{ij}\}$ and information matrices $\{\boldsymbol{\Omega}_{ij}\}$, prior on root pose $\left(\boldsymbol{x}_0^{\text{prior}}, \boldsymbol{\Omega}_{\text{prior}}\right)$, max iterations K, damping $\lambda \geq 0$ (LM), robust kernel ρ (optional)

Result: Optimized poses $\boldsymbol{x}_{0:T}^{\star}$

```
// Batch Pose-Graph SLAM (Gauss-Newton / Levenberg-Marquardt)
```

$\boldsymbol{x} \leftarrow \boldsymbol{x}_{0:T}^{(0)}$

for $k = 1$ **to** K **do**

  Initialize normal equations: $\boldsymbol{H} \leftarrow \boldsymbol{0}$, $\boldsymbol{b} \leftarrow \boldsymbol{0}$

  foreach $(i, j) \in \mathcal{E}$ **do**

    $\hat{\boldsymbol{z}}_{ij} \leftarrow h_{ij}(\boldsymbol{x}_i, \boldsymbol{x}_j)$

    $\boldsymbol{r}_{ij} \leftarrow \boldsymbol{z}_{ij} \ominus \hat{\boldsymbol{z}}_{ij}$ `// pose residual on` $SE(2/3)$

    $\boldsymbol{A}_i, \boldsymbol{A}_j \leftarrow \nabla_{\boldsymbol{x}_i, \boldsymbol{x}_j} \boldsymbol{r}_{ij}$

    $w_{ij} \leftarrow$ robust weight from $\rho(\|\boldsymbol{r}_{ij}\|_{\boldsymbol{\Omega}_{ij}})$ `// set` $w_{ij} = 1$ `if no robust kernel`

    $\tilde{\boldsymbol{\Omega}}_{ij} \leftarrow w_{ij} \boldsymbol{\Omega}_{ij}$

    `// Scatter-add into sparse` $\boldsymbol{H}, \boldsymbol{b}$

    $\boldsymbol{H}_{ii} \mathrel{+}= \boldsymbol{A}_i^{\top} \tilde{\boldsymbol{\Omega}}_{ij} \boldsymbol{A}_i, \quad \boldsymbol{H}_{ij} \mathrel{+}= \boldsymbol{A}_i^{\top} \tilde{\boldsymbol{\Omega}}_{ij} \boldsymbol{A}_j, \quad \boldsymbol{H}_{jj} \mathrel{+}= \boldsymbol{A}_j^{\top} \tilde{\boldsymbol{\Omega}}_{ij} \boldsymbol{A}_j$

    $\boldsymbol{b}_i \mathrel{+}= \boldsymbol{A}_i^{\top} \tilde{\boldsymbol{\Omega}}_{ij} \boldsymbol{r}_{ij}, \quad \boldsymbol{b}_j \mathrel{+}= \boldsymbol{A}_j^{\top} \tilde{\boldsymbol{\Omega}}_{ij} \boldsymbol{r}_{ij}$

  `// Anchor to fix gauge`

  $\boldsymbol{H}_{00} \mathrel{+}= \boldsymbol{\Omega}_{\text{prior}}, \quad \boldsymbol{b}_0 \mathrel{+}= \boldsymbol{\Omega}_{\text{prior}} \left(\boldsymbol{x}_0^{\text{prior}} \ominus \boldsymbol{x}_0\right)$

  `// Solve for increment`

  Solve $(\boldsymbol{H} + \lambda \boldsymbol{I})\, \Delta \boldsymbol{x} = \boldsymbol{b}$ with sparse Cholesky/QR

  `// Retract on the manifold`

  for $i = 0$ **to** T **do**

    $\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i \oplus \Delta \boldsymbol{x}_i$ `// retraction via` $\mathrm{Exp}(\cdot)$ `on` $SE(2/3)$

  if $\|\Delta \boldsymbol{x}\|_{\infty} < \varepsilon$ ***or*** *relative cost decrease* $< \tau$ **then**

    break

return $\boldsymbol{x}$

Algorithm 16.5: Factor-Graph SLAM: batch Gauss–Newton / Levenberg–Marquardt

Data: Variables $Y = \{X, L, \Theta, \dots\}$ with initial guess $Y^{(0)}$;
factors $\mathcal{F} = \{\phi_k\}$, each with measurement z_k, information $\mathbf{\Omega}_k$, and model $z_k \approx h_k(Y_k)$;
optional ordering π;
max iters K, damping $\lambda \geq 0$, robust kernel ρ (optional)
Result: MAP estimate $Y^\star$
$Y \leftarrow Y^{(0)}$
for $t = 1$ **to** K **do**
  $\boldsymbol{H} \leftarrow \mathbf{0}$, $\boldsymbol{b} \leftarrow \mathbf{0}$
  foreach $\phi_k \in \mathcal{F}$ **do**
    $\hat{\boldsymbol{z}}_k \leftarrow h_k(Y_k)$
    $\boldsymbol{r}_k \leftarrow \boldsymbol{z}_k \ominus \hat{\boldsymbol{z}}_k$ `// on tangent space`
    $\boldsymbol{J}_k \leftarrow \nabla_{Y_k} \boldsymbol{r}_k$
    $w_k \leftarrow$ robust weight from $\rho(\|\boldsymbol{r}_k\|_{\mathbf{\Omega}_k})$
    $\tilde{\mathbf{\Omega}}_k \leftarrow w_k \mathbf{\Omega}_k$
    $\boldsymbol{H} \mathrel{+}= \boldsymbol{J}_k^\top \tilde{\mathbf{\Omega}}_k \boldsymbol{J}_k, \quad \boldsymbol{b} \mathrel{+}= \boldsymbol{J}_k^\top \tilde{\mathbf{\Omega}}_k \boldsymbol{r}_k$
  if use Schur complement **then**
    Partition $\boldsymbol{H}, \boldsymbol{b}$ into pose vs. landmark blocks and eliminate L
  Solve $(\boldsymbol{H} + \lambda \boldsymbol{I}) \Delta Y = \boldsymbol{b}$ with sparse Cholesky/QR
  foreach *variable* $y \in Y$ **do**
    $y \leftarrow y \oplus \Delta y$
  if $\|\Delta Y\|_\infty < \varepsilon$ *or relative cost decrease* $< \tau$ **then**
    break
return Y

17
Sensor Fusion and Object Tracking

Individual sensors have inherent design limitations, such as limited range, limited field of view, or performance degradation under certain environmental conditions. In practice, it is also common for individual sensors to malfunction, either through a degradation of measurement accuracy over time or even from catastrophic sensor failure. We can design robots to be robust to these weaknesses and failure modes by incorporating multiple sensors, including multiple types of sensors, which can effectively reduce uncertainty for perception and localization tasks.

For example, a self-driving car might use a combination of lidar and radar for measuring distances, since lidar can provide short range high resolution data but radar is more robust at longer ranges[1]. The car might also leverage cameras to compliment distance sensors, which could give bearing measurements to an obstacle and are also much better suited to help with object recognition.

[1] Radar is also generally more robust than lidar in poor weather conditions such as fog, rain, or snow.

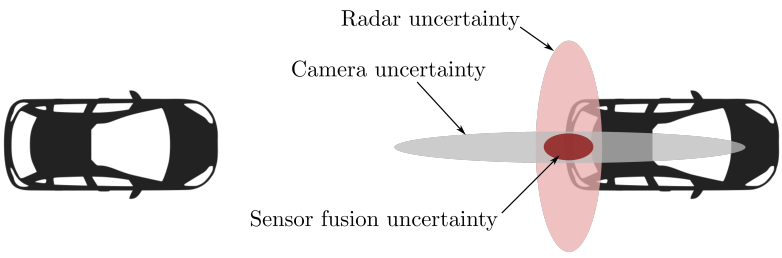


Figure 17.1: Sensor fusion can reduce uncertainty by providing more well-rounded data. For example, a radar sensor may provide good longitudinal distance accuracy but slightly less lateral accuracy, and a camera may provide poor range estimation but good lateral position estimation. By fusing these two signals, the resulting position estimate can be accurate longitudinally and laterally.

As another example, a wheeled robot may use GNSS sensors as well as wheel encoders to estimate position. The GNSS sensors can help ensure the global position error is small, but the wheel encoders might give a higher resolution signal for small movements, or if GNSS signals are lost.

This chapter covers the topic of *sensor fusion*[2,3], which provides methods for combining signals from multiple sensors and multiple sensor modalities to support a common goal, such as state estimation.

[2] F. Gustafsson. *Statistical Sensor Fusion*. Studentlitteratur, 2013, p. 554

[3] D. Simon. *Optimal State Estimation: Kalman, H_∞, and Nonlinear Approaches*. John Wiley & Sons, 2006

17.1 A Taxonomy of Sensor Fusion

We begin our discussion on sensor fusion by presenting a taxonomy of concepts and challenges that put the sensor fusion problem into a broader perspective. This includes a taxonomy for challenges related to the raw data signals that sensors provide, a taxonomy for types of signal fusion problems, and a taxonomy for the overall architectural designs of multi-sensor systems.

Data-related Taxonomy: One of the primary challenges with data fusion is the inherent imperfection in measurement data, including uncertainty from sensor noise, imprecision from sensor bias, and granularity from the sensor's resolution limits. Other important data-related aspects of sensor fusion include data correlation, disparity, and inconsistency, such as from data conflicts, outliers, or disorder. Broadly speaking, sensor data can experience multiple types of imperfection at the same time, and therefore we must develop data fusion algorithms with robustness in mind.

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, such as time-series data, intermediate-level problems fuse features and characteristics, and high-level fusion problems consider decisions. We also categorize fusion problems 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* problems consider sensors that provide complementary information about the environment, such as 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[4]. Generally speaking, competitive fusion increases reliability and accuracy of fused information, complementary fusion increases the completeness of information, and cooperative fusion broadens the type of information that we can gather.

[4] For example, GNSS localization and stereo vision can be cooperatively fused because they measure fundamentally different environmental quantities.

Architectural Taxonomy: We also classify fusion algorithms based on their type of architecture, namely whether they are *centralized*, *decentralized*, or *distributed*. Centralized architectures first collect all sensor data and then perform computations on the entire collection of data. This approach is theoretically optimal since all information is gathered and operated on at once, but it requires high levels of communication and processing which may be practically challenging. Decentralized architectures are essentially collections of centralized systems, and generally still suffer from the same high communication and processing requirements as centralized architectures. On the other hand, distributed architectures do not collect all sensor information ahead of time, but rather perform computations directly on local sensor data before potentially passing information on for further fusion tasks. Distributed architectures scale better, but can

lead to suboptimal performance because each sensor is performing local processing and therefore does not have all available information.

17.2 Bayesian Approach to Sensor Fusion

The previous chapters presented several algorithms for robot state estimation and localization based on Bayes filter. We can also view these algorithms as methods to solve the sensor fusion problem. In this section, we explore in more detail the Bayesian approach to sensor fusion, and show exactly how these approaches can blend measurement data to reduce uncertainty.

Recall that the Bayesian approach is probabilistic and models unknowns as random variables and quantifies knowledge and uncertainty in the form of probability distributions over these variables. This principled approach is also useful for sensor fusion problems for several reasons. First, it provides a *unified* framework for representing knowledge that is compatible with any quantity and type of sensor and is interpretable. Second, probability distributions implicitly provide information about uncertainty[5]. Third, Bayes' rule provides a theoretically principled approach for updating the probability distributions given data. Finally, we can use Bayesian approaches to deal with missing information and classification of new observations.

[5] For example, the variance of a Gaussian distribution is a statistic that quantifies the uncertainty in the distribution.

Example 17.2.1 (Probabilistic Competitive Fusion Example)**.** As an example to show how we can use a probabilistic approach to reduce uncertainty through sensor fusion, consider a case where two sensors are fused to estimate a single quantity, $x \in \mathbb{R}$. 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 $\sigma_1^2 < \sigma_2^2$, which means the first sensor has a higher precision than the second sensor. Assuming conditional independence, the joint measurement probability is:

$$p(y_1, y_2 \mid x) = p(y_1 \mid x)p(y_2 \mid x),$$

and by exploiting the property that the product of two Gaussian density functions is a Gaussian density function:

$$p(y_1, y_2 \mid x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}},$$

where:

$$\mu = \frac{y_1\sigma_2^2 + y_2\sigma_1^2}{\sigma_1^2 + \sigma_2^2}, \quad \sigma^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2}.$$

Given two measurements y_1 and y_2, we can see that the best estimate of the quantity x is given by μ, which is a weighted average of the two measurements where more influence is given to the measurement with higher precision. Since $\frac{\sigma_1^2}{\sigma_1^2+\sigma_2^2} < 1$ and $\frac{\sigma_2^2}{\sigma_1^2+\sigma_2^2} < 1$, we can also see that $\sigma^2 < \sigma_1^2 < \sigma_2^2$, and therefore the overall uncertainty is smaller.

17.2.1 Kalman Filter Sensor Fusion

The Kalman filter from **??** is a common parametric state estimation technique for solving sensor fusion problems. The Kalman filter assumes a linear state transition model:

$$\boldsymbol{x}_t = A_t \boldsymbol{x}_{t-1} + B_t \boldsymbol{u}_t + \boldsymbol{\epsilon}_t,$$

and a linear measurement model:

$$\boldsymbol{z}_t = C_t \boldsymbol{x}_t + \boldsymbol{\delta}_t,$$

where $\boldsymbol{x}$ is the state and $\boldsymbol{z}$ is the measurement. The Kalman filter also models the belief probability distribution over $\boldsymbol{x}$ and the noise terms, $\boldsymbol{\epsilon}$, $\boldsymbol{\delta}$, as Gaussian distributions:

$$\text{bel}(\boldsymbol{x}_t) \sim \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t), \quad \boldsymbol{\epsilon}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{R}_t), \quad \boldsymbol{\delta}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}_t),$$

where $\boldsymbol{R}_t$ and $\boldsymbol{Q}_t$ are the covariance matrices for the state transition and measurement noise models, respectively.

We can use this algorithm for sensor fusion because the measurement vector, $\boldsymbol{z}$, can include measurements from multiple different types of sensors at the same time, as long as the relationship between the sensor measurement and the state, $\boldsymbol{x}$, follows the linear model assumption. Each iteration of the Kalman filter corrects the predicted state estimate using each component of the measurement vector at time t simultaneously, and takes into account the covariance $\boldsymbol{R}_t$ that includes the covariance of each individual sensor. In fact, the Kalman filter will implicitly favor measurements with lower covariance when performing the correction step[6].

[6] Specifically, this occurs during the computation of the Kalman gain.

A useful trick for applying the Kalman filter to sensor fusion problems is to note that we can define the state, $\boldsymbol{x}$, to include 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, sensor offsets, or variables that define sensor and actuator health.

Example 17.2.2 (Kalman Filter Multi-Sensor Fusion)**.** Consider a self-driving car that has an inertial measurement unit (IMU), a GNSS receiver, and a lidar sensor, and suppose the goal is to leverage all of these sensors to estimate the longitudinal position, velocity, and acceleration of the vehicle. This suite of sensors provides noisy position estimates through the lidar and GNSS sensors,

as well as noisy acceleration measurements from the IMU. In this example, we perform sensor fusion by using the Kalman filter algorithm.

We start by defining a very simple kinematics model that models the longitudinal motion of the vehicle:

$$\dot{p} = v,$$
$$\dot{v} = a,$$

where p is the longitudinal position, v is the longitudinal velocity, and a is the longitudinal acceleration. The analytical solution of these differential equations assuming a constant acceleration is:

$$p(t) = p(0) + v(0)t + \frac{1}{2}at^2,$$
$$v(t) = v(0) + at,$$

and therefore we can discretize the differential equation model in time by choosing a sampling time, T, and assuming a constant acceleration over the interval to get:

$$\begin{bmatrix} p_{t+1} \\ v_{t+1} \\ a_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & T_s & \frac{T^2}{2} \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_t \\ v_t \\ a_t \end{bmatrix} + \boldsymbol{\epsilon}_t,$$

where we have also added the Gaussian process noise, $\boldsymbol{\epsilon}$. The state of this system is $\boldsymbol{x} := [p, v, a]^\top$. We assume 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 $\boldsymbol{\delta} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{Q}_t)$ is Gaussian measurement noise with zero mean and covariance:

$$\boldsymbol{Q}_t = \begin{bmatrix} \sigma^2_{\text{lidar}} & 0 & 0 \\ 0 & \sigma^2_{\text{gnss}} & 0 \\ 0 & 0 & \sigma^2_{\text{imu}} \end{bmatrix},$$

with assumed parameters $\sigma_{\text{lidar}} = 0.5$, $\sigma_{\text{gnss}} = 0.1$, and $\sigma_{\text{imu}} = 0.2$.

Figure 17.2 shows results of the application of the Kalman filter algorithm for fusing these sensor measurements into position estimates. The top plot shows a case where the GNSS sensor is not used, and we can see the noisy high-variance lidar measurements result in a noisy estimate of the ground truth position of the car. With the addition of the lower-variance GNSS sensor in the bottom figure, the estimate of the position is much less noisy[7].

[7] Generally speaking, the estimate would also be more accurate even with the addition of a sensor that was noisier than the lidar sensor, but the impact would not be as significant.

17.3 Practical Challenges in Sensor Fusion

Sensor fusion problems are generally quite challenging and can vary significantly from application to application. Some practical problems in the context

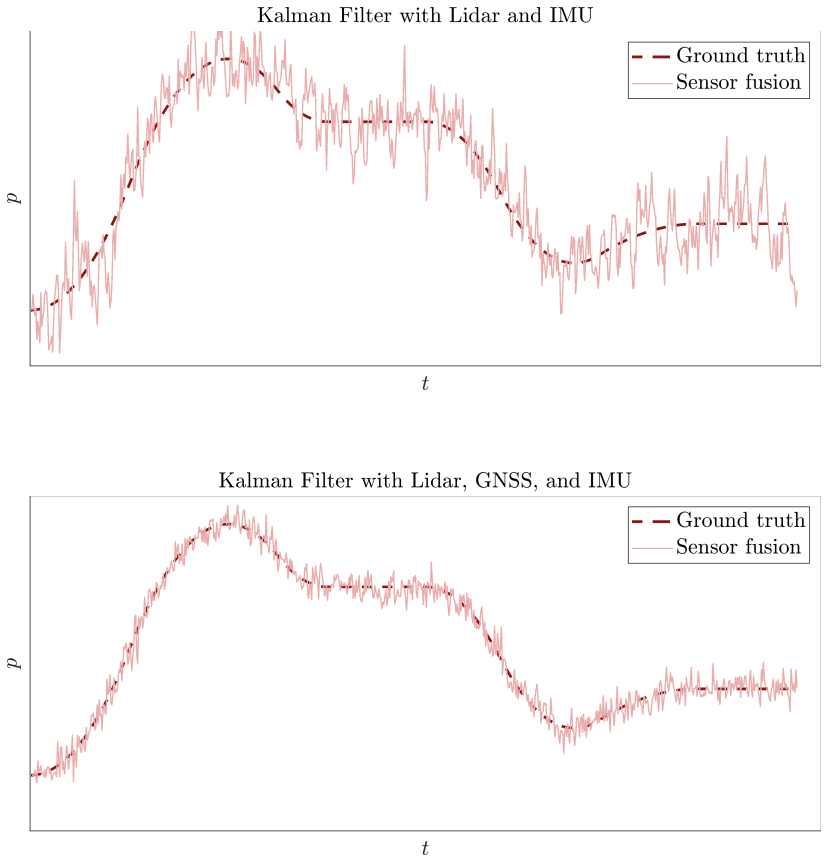


Figure 17.2: Kalman filter sensor fusion for Example 17.2.2. The position of a vehicle is estimated using noisy lidar, GNSS, and IMU data, and the resulting estimate tracks the ground truth. We can see that the addition of the GNSS sensor improves the estimate through sensor fusion.

of sensor fusion are referred to as *registration*, *bias*, *correlation*, *data association*, and *out-of-sequence measurements*. The registration problem is that coordinate frames, both in time and space, of different sensors may not always be aligned, which is necessary to ensure we appropriately combine their signals. 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 any knowledge of correlation between sensors can have an impact on the best way to fuse the information. In some robotics applications, such as in multi-target tracking problems, data association[8] can also be a challenge. Finally, out-of-sequence measurements[9] also pose a logistical challenge in practical sensor fusion applications. Out-of-sequence measurements might lead to an incorrect temporal order, which causes a negative time measurement update during data fusion, such as in the Kalman filter algorithm.

[8] The data association problem is similar to the correspondence problem in the context of SLAM, which we discussed in Chapter 16.

[9] One potential cause of out-of-sequence measurements is due to communication limitations among agents in multi-agent settings.

17.4 *Object Tracking*

Robot systems rely on an accurate perception of their dynamic environment to safely and effectively operate autonomously. Therefore, tracking other objects by predicting their state over time given noisy and sometimes ambiguous mea-

surements, occlusions, false signals, and inherent prediction uncertainty[10] is an important task. The *single-object tracking*[11] problem is well-understood and typically easy to solve. For example, we can leverage algorithms that we have already studied, such as the EKF, to track the object's state. *Multi-object tracking* algorithms typically maintain a set of estimation filters, with one filter for each object being tracked. Like in the single-object tracking problem, we can leverage single-hypothesis algorithms like the Kalman filter or EKF for each individual filter. However, multi-object tracking is a more challenging problem due to additional factors, such as the data association problem for assigning observations to each target and the track maintenance problem to know when to create or delete tracks.

[10] In other words, uncertainty in the dynamics of the object.

[11] Also sometimes referred to as *single hypothesis tracking*.

17.4.1 Gating

For the data association problem in multi-object tracking, we must consider how likely each observation comes from a particular track. One approach to help solve this problem is to use a *gating or screening mechanism*. This process generally consists of simply ignoring measurements that occur outside of a specific region for each track, which can speed up the data assignment process by reducing the number of measurements that need to be processed. The gating regions can take on simple geometric forms, such as rectangular or ellipsoidal areas that represent the level set of a multivariate Gaussian distribution.

17.4.2 Data Association

Data association or data assignment is the process of linking an observation to a track. This is particularly difficult if we have a large number of target tracks, many detections, or conflicting hypotheses. We categorize the assignment problems as either 2-D or S-D. The 2-D assignment problem assigns n targets to m measurements, for example where the m measurements all come from the same sensor. The S-D assignment problem assigns n targets to a set of measurements $\{m_1, m_2, \dots\}$, for example where each set of measurements, m_i, comes from a different sensor. We focus on two solution methods for the 2-D assignment problem, the *global nearest neighbor (GNN)* approach and the *joint probabilistic data association (JPDA)* approach. The global nearest neighbor approach is a single hypothesis approach that assigns the nearest observations to existing tracks and creates new track hypotheses for unassigned observations. The assignment of observations to an existing track is straightforward if there is no conflict where an observation falls in the gating region of multiple targets, or if multiple observations fall within the gating region of a single target. If there is a conflict, the GNN approach defines a cost based on a generalized statistical distance and makes the assignments that minimize the cost. Alternatively, the joint probabilistic data association (JPDA) approach is a Bayesian technique that fuses measurements weighted by the probability of the observation-to-track association[12].

[12] Unlike the GNN, which makes a hard assignment, the JPDA approach performs a soft assignment of observations to tracks.

17.4.3 Track Maintenance

The track maintenance problem is to determine when to create new tracks and when to remove old tracks. A simple approach for track removal is to keep track of how many times observations are assigned to the track, and to remove the track if it has not been assigned an observation in some fraction of the most recent updates. Similarly, for track creation, a simple approach is to create a virtual track when there is a single unassigned observation, and then promote this virtual track into a new track if it is assigned an observation in some fraction of the following updates.

17.4.4 Extended Object Tracking

One potential failure mode of standard multi-object tracking algorithms is when a single target generates multiple observations[13], such as due to observation reflections or due to high-resolution sensor modalities like lidar that might generate a point cloud for a single object. We refer to this new problem, of handling multiple observations from a single sensor for each track, as an *extended object tracking* problem. Extended object tracking algorithms estimate position and velocity like standard multi-object tracking algorithm, but they also estimate the dimensions and the orientation of the object. Prominent extended object tracking algorithms include the Gamma-Gaussian inverse Wishart probability hypothesis density (PHD) tracker and the Gaussian-mixture PHD tracker.

[13] Standard multi-object tracking algorithms typically assume a single object detection per sensor.

References

[19] F. Gustafsson. *Statistical Sensor Fusion*. Studentlitteratur, 2013, p. 554.

[66] D. Simon. *Optimal State Estimation: Kalman, H_∞, and Nonlinear Approaches*. John Wiley & Sons, 2006.

Part IV

Robot Decision Making

18
Finite State Machines

Algorithms for solving problems in robot control, trajectory optimization, motion planning, perception, localization, and state estimation often involve modeling, manipulation, and observation of *continuous* variables. This is a natural consequence of the fact that robots are often physical entities that operate in physical environments. For example, motion planning and control algorithms manipulate the robot's physical state, consisting of continuously varying positions, velocities, and orientations, and perception and localization tasks observe continuously valued information from the environment to estimate the robot's or the environment's physical state.

However, there are also problems in robot autonomy where we represent the state of the robot or environment using discrete variables, such as when planning higher-level tasks that involve discrete logic or actions. For example, consider a planning task to go from point A to point B, pick up a package, and then deliver it to point C. While the robot's continuous physical state is crucial for the lower-level motion planning and control task to get the robot to drive from point to point, it is also important to keep track of the stage of the overall plan that the robot is currently performing, such as what location the robot is currently headed to and if the package has been successfully picked up. We might also want to keep track of other discrete valued states of the robot, such as if a sensor is functioning properly. Analogously to dynamics and kinematics models for the robot's continuous physical state, *finite state machines*[1] are one useful modeling framework[2] for discrete states and transitions of the robot and its environment. In this chapter, we provide a mathematical definition of a finite state machine, discuss some architecture options and computational challenges, and then discuss a practical implementation approach.

[1] L. Kaelbling et al. *6.01SC: Introduction to Electrical Engineering and Computer Science I*. MIT OpenCourseWare. 2011

[2] It is important to note that this is a *framework* for modeling and is not a particular *algorithm*.

18.1 Finite State Machines

Finite state machines (FSMs) are a computational modeling framework for systems with a *finite* number of states whose output depends on the entire history of their inputs. This framework is used in a wide variety of disciplines, including electrical engineering, linguistics, computer science, philosophy, biology, and

more. We can use finite state machines in several different ways, including to *specify* a desired program or behavior, to *model and analyze* a system's behavior, or to *predict future behavior*.

One important practical disadvantage of finite state machines is that their complexity does not scale well with system complexity, and, generally speaking, it can be time consuming and challenging to design FSMs for practical robotic systems. To reduce complexity as much as possible, we must carefully choose the appropriate set of states to represent the system, and even with a well defined set of states the interactions and transitions between states can be complex and hard to specify. For example, Figure 18.1 shows a graphical representation of the finite state machine for the popular open source flight software PX4. Specifying the full behavior for a system like this can lead to a complex FSM, even if there are not very many states.

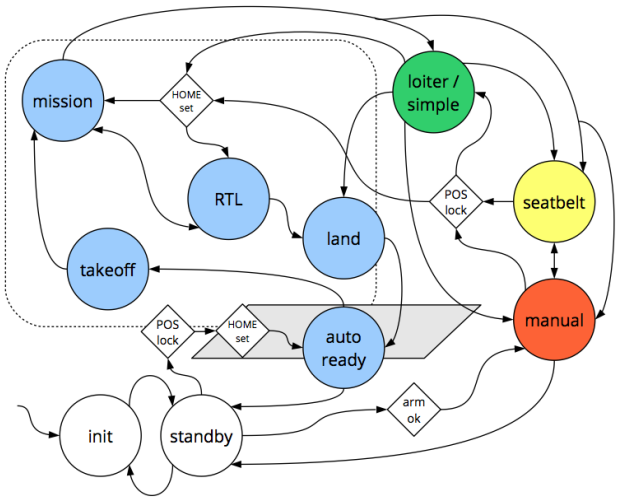


Figure 18.1: A graphical representation of the finite state machine for the open source flight software PX4, `https://px4.io/`. We can see that even for a relatively small number of states, the FSM is quite complex in order to model the full behavior of the system. Image retrieved from diydrones.com.

We define a finite state machine mathematically by a *finite* set of states, S, a set of inputs, I, a set of outputs, O, a next-state function, $n(i_t, s_t) \rightarrow s_{t+1}$, that maps the input, i_t, and current state, s_t, at time t to the next state, s_{t+1}, an output function, $o(i_t, s_t) \rightarrow o_t$, and an initial state, s_0. We also represent FSMs graphically, which often gives a more intuitive understanding of how the system will behave. To define the graph representation, we use nodes to represent each state in the set S and each directed edge of the graph corresponds to a possible transition between states that is defined by a particular input. We also typically include the outputs for a particular state-input pair, (s, i), along each directed edge. Figure 18.2 shows an example graphical representation of a simple finite state machine with three states.

Example 18.1.1 (Parking Gate Control)**.** Consider a parking gate control problem where the goal is to raise the gate when a car arrives, and then lower the gate when the car has passed. We assume sensors are available to tell if a car is at the gate, when the car has passed through the gate, and the current posi-

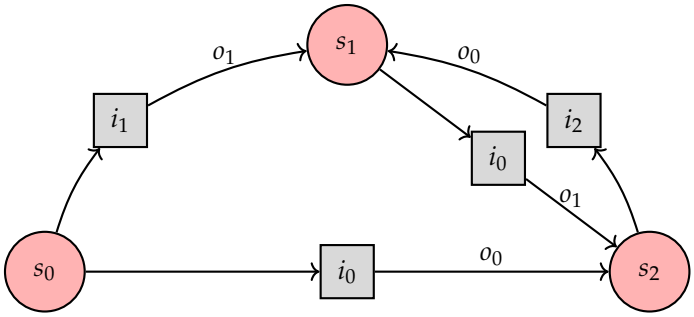


Figure 18.2: A graphical representation of a finite state machine with states $S = \{s_0, s_1, s_2\}$, inputs $I = \{i_0, i_1, i_2\}$ and outputs $O = \{o_0, o_1\}$. The directed edges correspond to the next-state functions and the output associated with each edge is defined by the output function. For example, in this FSM, we show the transition $n(i_1, s_0) \rightarrow s_1$ in the top left, along with the corresponding output $o(i_1, s_0) \rightarrow o_1$.

tion of the gate. The control actions are raising, lowering, or holding the gate position fixed. Note that in the real world, the position and velocity of the gate can vary continuously between the *down* and *up* positions. However, we use a higher level abstraction for designing a finite state machine that defines the overall logic for the parking gate. In our FSM model, we choose the states to be:

$$S := \{\text{down, raising, up, lowering}\}.$$

We choose the set of inputs, which correspond to the available sensors, as:

$$I := \{\text{car waiting, no car waiting, car past, car not past,} \\ \text{gate up, gate down, gate moving}\}.$$

Finally, we define the output of the finite state machine, which specify the actions for the gate, as:

$$O := \{\text{lower, raise, hold}\}.$$

We now choose the next-state function and output function to define the desired behavior for the parking gate. For example, if the gate is currently down and the sensor measures that a car is waiting, the desired behavior is to output the command to raise the gate. This desired behavior is transcribed in the next-state and output functions by:

$$n(\text{car waiting}, \text{down}) \rightarrow \text{raising},$$
$$o(\text{car waiting}, \text{down}) \rightarrow \text{raise},$$

Similarly, if the gate was just raised for the car and the sensor shows that the car is not yet fully past, the desired output is to hold the gate up, and we define the next-state and output functions as:

$$n(\text{up}, \text{car not past}) \rightarrow \text{up},$$
$$o(\text{up}, \text{car not past}) \rightarrow \text{hold},$$

Figure 18.3 shows a graphical representation of the full car parking gate finite state machine.

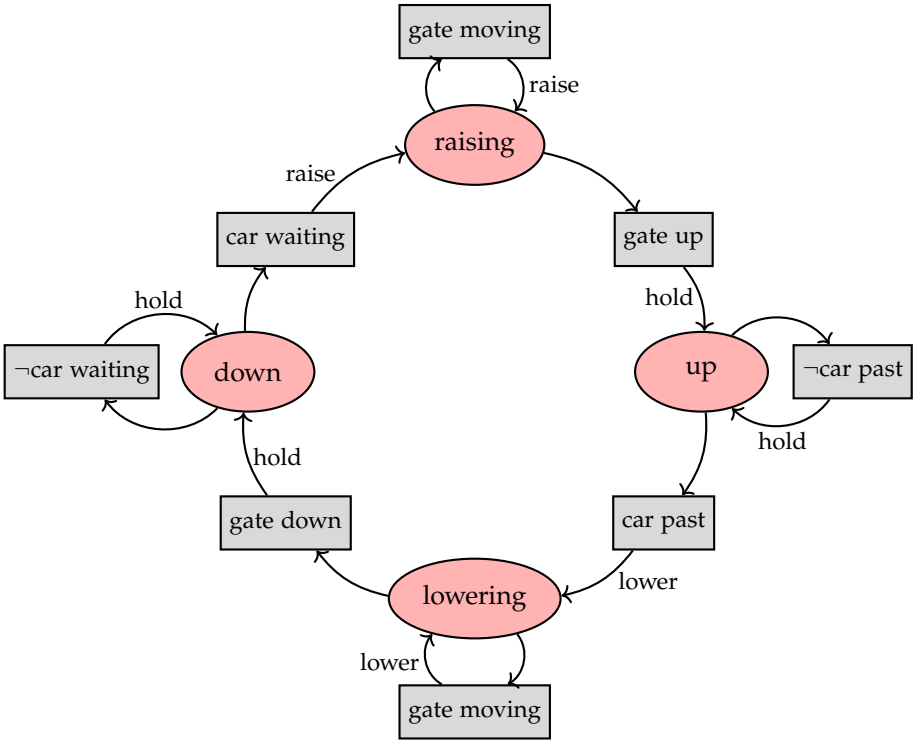


Figure 18.3: A graphical representation of the finite state machine for the parking gate controller discussed in Example 18.1.1.

18.2 Finite State Machine Architectures

Finite state machines can become quite complex since increasing the number of states by one from N to $N + 1$ increases the number of possible transitions by N. We can keep the complexity of FSMs lower by analyzing for and removing redundant states, using hierarchical FSMs, and using compositions based on common patterns.

18.2.1 Reducing the Number of States

Algorithms exist to identify and combine states in FSMs that would yield the same overall behavior. In particular, we say that two states are equivalent, and therefore can be combined, if they have the same output and transition to the same or equivalent states for all input combinations. One possible generic algorithmic approach for reducing states in an FSM is to first place all states into a single set, then create a single partition based on the output behavior, and then repeatedly partition further based on next state transitions until no further partitions are possible. We provide an example of this procedure in Example 18.2.1.

Example 18.2.1 (Finite State Machine State Reduction)**.** Consider a finite state machine that detects the input sequences 010 or 110. We show the states, next-state function values, and output function values for this FSM in Table 18.1. We can see that the states are the partial sequences and a *reset* state, $S :=$

$\{0, 1, 00, 01, 10, 11, \text{reset}\}$, the inputs are $I := \{0, 1\}$, and the outputs are the booleans $O := \{\text{True}, \text{False}\}$ that indicate if the sequence 010 or 110 has been created. For example, if the current partial sequence is 01 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 18.1: Finite state machine for a sequence detector that accepts digits 0 and 1 and outputs True if the sequences 010 or 110 are generated.

We can now simplify this FSM by removing redundant states. To identify redundant states, we first place all of the states into a single set, $\{\text{reset}, 0, 1, 00, 01, 10, 11\}$, and create a partition based on the output behavior:

$$\{\text{reset}, 0, 1, 00, 10\} : \text{always leads to a False output},$$
$$\{01, 11\} : \text{does not always lead to False output}.$$

We then further partition these sets based on the next-state function until we cannot make any further partitions. In the first step, we partition the set $\{\text{reset}, 0, 1, 00, 10\}$ into:

$$\{\text{reset}, 00, 10\} : \text{cannot transition to } \{01,11\},$$
$$\{0, 1\} : \text{can transition to } \{01,11\},$$

and then partition $\{\text{reset}, 00, 10\}$ into:

$$\{\text{reset}\} : \text{can transition to } \{0, 1\},$$
$$\{00, 10\} : \text{cannot transition to } \{0, 1\}.$$

Therefore, instead of the original seven states, $\{0, 1, 00, 01, 10, 11, \text{reset}\}$, there are now only four states, $S_{\text{new}} = \{\{01, 11\}, \{0, 1\}, \{00, 10\}, \text{reset}\}$. We can therefore now define the equivalent[3] and reduced finite state machine shown in Table 18.2.

[3] Equivalent here meaning it has the same input-output behavior.

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 18.2: 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.

18.2.2 Hierarchical FSMs

In some cases, there might be states that are not truly equivalent but we might still find it to be beneficial to group them closely together. For these cases, we

can define FSMs based on the concepts of *super-states*, which are groups of closely related states, and *generalized transitions* that define transitions between super-states. This approach is analogous to graph clustering.

18.2.3 Compositions

We can also compose individual state machines 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, such as we show in Figure 18.4. Parallel compositions run two FSMs side by side using the same input. In this case, we define both the state and output by the concatenation of the two individual FSMs' state and output. Feedback compositions use a single FSM, but only require a partial input and then also reuse the output as input[4].

[4] Note that this composition requires the input and output vocabularies to be the same.

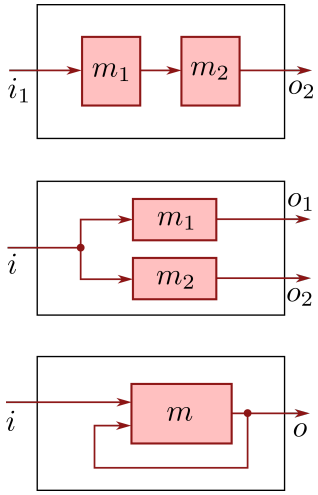


Figure 18.4: Cascade, parallel, and feedback compositions of finite state machines.

18.3 Implementation Details

There are numerous ways to implement finite state machines in practice. One common approach is to exploit Object Oriented Programming (OOP) by building the finite state machine as a class. This approach keeps track of the state of the FSM in a class member variable. We then implement the state update process and definition of the FSM outputs through the use of if-else statements in class methods. We show an example implementation in Python of the parking gate controller finite state machine from Example 18.1.1 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()
```

References

[26] L. Kaelbling et al. *6.01SC: Introduction to Electrical Engineering and Computer Science I*. MIT OpenCourseWare. 2011.

19
Sequential Decision Making

Decision making is a fundamental task in robot autonomy. Specifically, we are typically interested in the problem of *sequential decision making*[1,2], which allows us to decompose longer horizon planning tasks into incremental actions, incorporate new information about the world over time, and also reason about how future actions and observations can influence the current decision. Robot sequential decision making tasks are very diverse. They range from low-level control to high-level task planning, they can involve discrete or continuous actions and states, and they might need to be made at high or low frequencies. For example, the low-level trajectory tracking task that we introduced in Chapter 3 on closed-loop motion planning and control is a sequential decision making task that involves continuous physical robot dynamics and operates at high frequency. On the other side of the spectrum are high-level decision making tasks, such as the parking gate operation problem from Chapter 18 on finite state machines, which involve discrete state and control spaces and can operate at low frequencies. Many other important autonomous sequential decision making tasks live somewhere in between these extremes. For example, an autonomous vehicle navigating an intersection needs to reason about continuous physical motions as well as discrete actions like activating a turn signal, or a warehouse robot may need to decide the order to pickup and drop off packages while accounting for physical constraints.

While we have already discussed some concepts for sequential decision making problems, including the closed-loop control framework and the finite state machine modeling framework, in this chapter, we introduce a new general optimization-based problem formulation[3] that is commonly applied to a large variety of decision making problems. We also extend the problem formulation to consider problems with *uncertainty*[4], which is a fundamental aspect of practical robot autonomy, in the forms of the *stochastic decision making problem* and the related and commonly used *Markov decision process* framework. In addition to the problem formulation, we also introduce *dynamic programming*, a foundational approach for solving these problems that leverages the *principle of optimality*.

[1] D. Bertsekas. *Reinforcement learning and optimal control*. Athena Scientific, 2019

[2] M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray. *Algorithms for Decision Making*. MIT Press, 2022

[3] Like some of the methods we discussed in Chapter 3, this new approach is considered a *closed-loop optimal control* method.

[4] These problems are referred to as *stochastic* decision making problems, and can include uncertainty in the robot's state and environment.

19.1 Deterministic Decision Making Problem

Sequential decision making problems are commonly formulated as optimization problems because the objective function can naturally encode the *goal* or *task* and the constraints encode other relevant limitations, like physical motion constraints and control or resource constraints. The mathematical formulation for these problems includes several components, including a state transition model describing the robot's behavior, a set of admissible control inputs, and a cost function. This formulation is quite similar to the optimization problems we discussed in Chapter 2 and Chapter 3, except we now express the problem in *discrete-time* rather than in *continuous-time*[5]. In practice, discrete time formulations are often more convenient for higher level decision making problems and it is also generally easier to design and implement practical computational algorithms to solve them[6].

[5] The continuous-time formulation is known as the Hamilton–Jacobi–Bellman formulation.

[6] Recall the discussions on the advantages of *direct methods* from Chapter 2.

In the deterministic decision making problem, we model robot's state transition model[7] in *discrete-time* as:

$$\boldsymbol{x}_{t+1} = f_t(\boldsymbol{x}_t, \boldsymbol{u}_t), \quad t = 0, \ldots, T-1, \tag{19.1}$$

[7] In the context of sequential decision making, we typically refer to this model as a *state transition* model rather than a *dynamics* model, which was the terminology we used in the context of motion planning and control.

where $\boldsymbol{x} \in \mathbb{R}^n$ is the robot's state, $\boldsymbol{u} \in \mathbb{R}^m$ is the control input, the function f_t defines how the robot's state changes at time step t, and T is an integer that defines a finite planning horizon for the decision making problem. We also assume that only some control actions are admissible at a given state, which we denote by the set $\mathcal{U}(\boldsymbol{x}_t)$. For example, in a high-level routing problem, a car may only have an option to turn left or right when it is at an intersection. Therefore, we express the control constraints at time step t by:

$$\boldsymbol{u}_t \in \mathcal{U}(\boldsymbol{x}_t). \tag{19.2}$$

Note that there are generally no restrictions on how the set of admissible controls is defined. For example, $\mathcal{U}(\boldsymbol{x}_t)$ could be a finite set of actions or a convex region over a continuous action space.

We assume the cost function that defines the goal of the decision making problem to be *additive* and defined over a finite horizon as:

$$J(\boldsymbol{x}_0, \boldsymbol{u}_0, \ldots, \boldsymbol{u}_{T-1}) = g_T(\boldsymbol{x}_T) + \sum_{t=0}^{T-1} g_t(\boldsymbol{x}_t, \boldsymbol{u}_t), \tag{19.3}$$

where g_T is a terminal state cost function and g_t for $t = 0, \ldots, T-1$ are stage cost functions[8]. We also don't place any particular restrictions on these cost functions, for example related to convexity or differentiability.

[8] In practice, it is common for the stage cost to be constant over time.

Definition 19.1.1 (Deterministic Decision Making Problem)**.** The deterministic decision making problem for the state transition model in Equation (19.1), control constraints in Equation (19.2), and cost function in Equation (19.3) is to compute the finite horizon control sequence that is the solution to the optimiza-

tion problem:

$$\begin{aligned} J^*(\boldsymbol{x}_0) = \underset{\boldsymbol{u}_t,\ t=0,\ldots,T-1}{\text{minimize}}\ & J(\boldsymbol{x}_0, \boldsymbol{u}_0, \ldots, \boldsymbol{u}_{T-1}), \\ \text{s.t. } & \boldsymbol{x}_{t+1} = f_t(\boldsymbol{x}_t, \boldsymbol{u}_t), \quad t = 0, \ldots, T-1, \\ & \boldsymbol{u}_t \in \mathcal{U}(\boldsymbol{x}_t),\ t = 0, \ldots, T-1 \end{aligned} \tag{19.4}$$

The solution to Equation (19.4) is the optimal *open-loop* control sequence, $\{\boldsymbol{u}_0^*, \ldots, \boldsymbol{u}_{T-1}^*\}$, given the initial condition, $\boldsymbol{x}_0$, which is similar to the trajectory optimization problem in Chapter 2. This problem is generally challenging to solve in practice since there is no guarantee that the state transition model in Equation (19.1) and cost function in Equation (19.3) have any particular structure that we can leverage to make the optimization problem amenable to numerical optimization algorithms. In theory, we could solve the problem through a brute force search over all possible combinations of sequences, $\{\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{T-1}\}$, but this leads to a combinatorial explosion of options and is therefore not practical except for very small problems.

19.1.1 Principle of Optimality (Deterministic Case)

Fortunately, the deterministic decision making problem possesses an underlying structure that we can leverage to solve the problem more efficiently than with brute force search. We refer to this underlying problem structure as the *principle of optimality*[9].

[9] Also referred to as *Bellman's principle of optimality*.

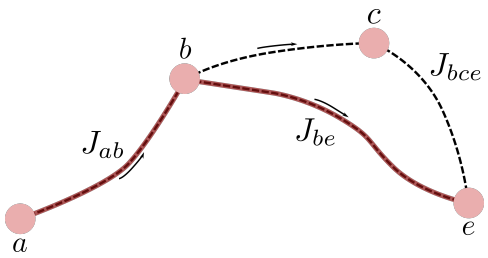


Figure 19.1: Starting from point a, let the path $a \to b \to e$ be the optimal path from a to e, with a total cost of $J_{ae}^* = J_{ab} + J_{be}$. The principle of optimality says that the path $b \to e$ must therefore be the optimal path when starting from point b. We can prove this by contradiction, since if the path $b \to c \to e$ had a lower cost than path $b \to e$, such that $J_{bce} < J_{be}$, then the original path, $a \to b \to e$, cannot be optimal.

The principle of optimality for deterministic systems states 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 19.1. This property greatly simplifies the overall problem, since we can *reuse* optimal paths for different scenarios. We define the principle of optimality more formally in Theorem 19.1.2.

Theorem 19.1.2 (Principle of Optimality (Deterministic Case)). *Let* $\{\boldsymbol{u}_0^*, \boldsymbol{u}_1^* \ldots, \boldsymbol{u}_{T-1}^*\}$ *be an optimal control sequence to the deterministic decision making problem in Equation* (19.4) *with a given initial condition,* $\boldsymbol{x}_0^*$*, and let the resulting optimal state sequence be* $\{\boldsymbol{x}_0^*, \boldsymbol{x}_1^* \ldots, \boldsymbol{x}_T^*\}$*. The tail sequence,* $\{\boldsymbol{u}_t^*, \ldots, \boldsymbol{u}_{T-1}^*\}$*, is then an optimal control sequence when starting from* $\boldsymbol{x}_t^*$ *and minimizing the cost:*

$$J_{tail}(\boldsymbol{x}_t, \boldsymbol{u}_t, \ldots, \boldsymbol{u}_{T-1}) = g_T(\boldsymbol{x}_T) + \sum_{i=t}^{T-1} g_i(\boldsymbol{x}_i, \boldsymbol{u}_i),$$

from time t to time T

We now demonstrate how to apply the principle of optimality to simplify the decision making problem for the scenario in Figure 19.2. In this case, our goal is to find an optimal path from point b to point f, and we assume that we already know optimal paths from c, d, and e to f. A brute force search over all possible paths in this problem would require us to evaluate nine paths:

$$\{b \to c \to f, \quad b \to c \to d \to f, \quad b \to c \to d \to e \to f, \quad b \to d \to c \to f, \quad b \to d \to f, \\ b \to d \to e \to f, \quad b \to e \to d \to c \to f, \quad b \to e \to d \to f, \quad b \to e \to f\}.$$

However, by leveraging the principle of optimality, the number of candidate paths is reduced to just three:

$$b \to c \to f, \quad b \to d \to f, \quad b \to e \to f.$$

By leveraging the principle of optimality, we can perform a search over just the *immediate* decisions by concatenating the optimal tail decisions. We generally implement this procedure backward in time, for example, in Figure 19.2, we evaluate the goal point f first, then the points c, d, and e, and then finally the point b.

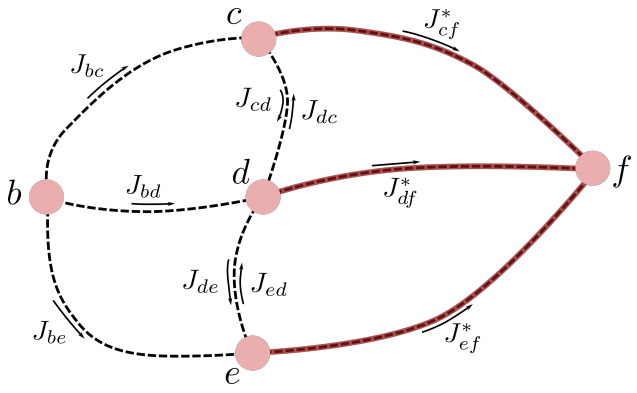


Figure 19.2: Suppose we know the optimal paths from points c, d and e to f. Using the principle of optimality, we can find the optimal path from point b to f by only searching over paths from b to c, d, and e, and determining the lowest cost from the candidates $\{J_{bc} + J^*_{cf}, J_{bd} + J^*_{df}, J_{be} + J^*_{ef}\}$. In other words, we can leverage the *optimal tails* to reduce the total number of paths that we need to be consider when finding the optimal path from b to f.

19.1.2 *Dynamic Programming (Deterministic Case)*

Dynamic programming is a foundational technique that leverages the principle of optimality[10] to *globally* solve the deterministic decision making problem in Equation (19.4). The dynamic programming algorithm, which we detail in Algorithm 19.1, performs a backward-in-time recursion where each step performs a *local* optimization[11] that leverages the optimal tail costs from the previous iteration. The output of the dynamic programming algorithm is a set of costs, $J^*_t(\boldsymbol{x}_t)$, for each time step, $t = 0, \ldots, T$, and states, $\boldsymbol{x}_t$, which provide the optimal tail costs for the tail subproblems.

Then, given an initial condition, $\boldsymbol{x}_0$, we can compute the optimal control sequence, $\{\boldsymbol{u}^*_0, \ldots, \boldsymbol{u}^*_{T-1}\}$, that solves the deterministic decision making problem

[10] The principle of optimality is a fundamental property that is actually leveraged in almost all decision making algorithms, not just dynamic programming.

[11] The local optimization equation is often referred to as the *Bellman* equation.

Algorithm 19.1: Dynamic Programming (Deterministic)

$J_T^*(\boldsymbol{x}) = g_T(\boldsymbol{x})$, for all $\boldsymbol{x} \in \mathcal{X}$
for $t = T - 1$ **to** 0 **do**
  $J_t^*(\boldsymbol{x}) = \underset{\boldsymbol{u} \in \mathcal{U}(\boldsymbol{x})}{\text{minimize}}\ g_t(\boldsymbol{x}, \boldsymbol{u}) + J_{t+1}^*(f_t(\boldsymbol{x}, \boldsymbol{u}))$, for all $\boldsymbol{x} \in \mathcal{X}$
return $J_0^*(\cdot), \ldots, J_T^*(\cdot)$

with a "forward pass" procedure. For this procedure, we start by computing the first control input:

$$\boldsymbol{u}_0^* = \underset{\boldsymbol{u}_0 \in \mathcal{U}(\boldsymbol{x}_0)}{\arg\min}\ g_0(\boldsymbol{x}_0, \boldsymbol{u}_0) + J_1^*(f_0(\boldsymbol{x}_0, \boldsymbol{u}_0)).$$

We then compute the next state, $\boldsymbol{x}_1^* = f_0(\boldsymbol{x}_0, \boldsymbol{u}_0^*)$, and repeat the process:

$$\boldsymbol{u}_1^* = \underset{\boldsymbol{u}_1 \in \mathcal{U}(\boldsymbol{x}_1^*)}{\arg\min}\ g_1(\boldsymbol{x}_1^*, \boldsymbol{u}_1) + J_2^*(f_1(\boldsymbol{x}_1^*, \boldsymbol{u}_1)),$$

until the full trajectory and optimal control is specified.

In practice, the dynamic programming algorithm in Algorithm 19.1 is not practical to implement when the state space is continuous, since it would have to iterate over an infinite number of states. One possible modification for Algorithm 19.1 to handle continuously valued states is to discretize the state space into a finite set of states. However, even a finite but large set of states can be computationally challenging to handle in practice. These challenges have led to the development of more practical algorithms that are based on dynamic programming and the principle of optimality, but make various simplifying approximations. Another interesting thing to note about the dynamic programming algorithm is that control constraints can actually simplify the procedure, since they restrict the number of possible state transitions that we need to consider.

Example 19.1.1 (Deterministic Dynamic Programming)**.** Consider the environment shown in Figure 19.3, where the goal is to start at point a and reach point h while incurring the smallest cost. In this problem, we represent the state as the current location and we encode the control constraints by the arrows indicating possible travel directions. For example, at point c, it is possible to either go right or up but not down or left. We also define the cost of traversing between two points in Figure 19.3.

For this problem, we start the dynamic programming recursion at the goal point h with:

$$J_T^*(h) = 0,$$

since we assume there is no cost to stay at point h. Moving backward in time, we can see that the possible states $\boldsymbol{x}_{T-1}$ that can transition to $\boldsymbol{x}_T = h$ are the points h, e, and g, again assuming it is possible to stay at h with no cost. There-

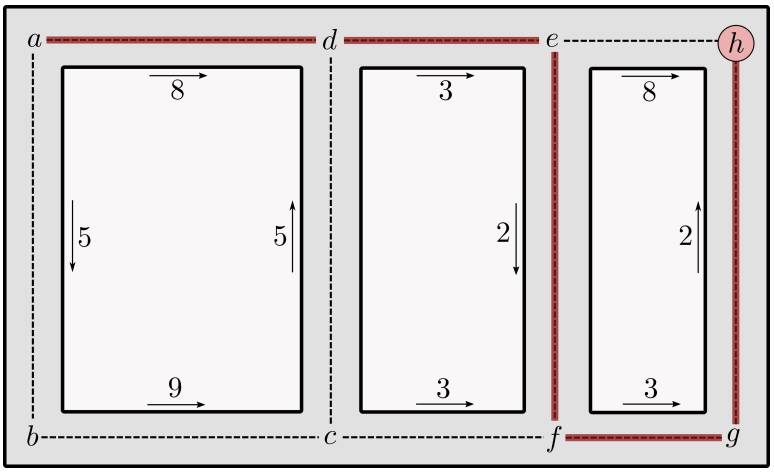


Figure 19.3: A deterministic decision making problem where the goal is to move from point a to point h while incurring the minimal amount of cost. The path $a \to d \to e \to f \to g \to h$ is the optimal path. We solve this problem by dynamic programming in Example 19.1.1.

fore, in the first step of the dynamic programming recursion we compute:

$$\begin{aligned} J^*_{T-1}(h) &= 0 + J^*_T(h) = 0, \quad u^*_{T-1}(h) = \text{stay}. \\ J^*_{T-1}(e) &= 8 + J^*_T(h) = 8, \quad u^*_{T-1}(e) = \text{right}, \\ J^*_{T-1}(g) &= 2 + J^*_T(h) = 2, \quad u^*_{T-1}(g) = \text{up}, \end{aligned}$$

Note that $J^*_t(h) = 0$ for all $t \leq T$, and therefore we will not explicitly include it in the following steps. In the next step:

$$\begin{aligned} J^*_{T-2}(e) &= 8 + J^*_{T-1}(h) = 8, \quad u^*_{T-2}(e) = \text{right}, \\ J^*_{T-2}(g) &= 2, \quad u^*_{T-2}(g) = \text{up}, \\ J^*_{T-2}(d) &= 3 + J^*_{T-1}(e) = 11, \quad u^*_{T-2}(d) = \text{right}, \\ J^*_{T-2}(f) &= 3 + J^*_{T-1}(g) = 5, \quad u^*_{T-2}(f) = \text{right}, \end{aligned}$$

At this point, these optimal tail costs are the optimal costs associated with control actions that lead from e, g, d, or f to the end point, h, in *two* steps. Continuing the recursion for the third step:

$$\begin{aligned} J^*_{T-3}(e) &= \min\{8 + J^*_{T-2}(h),\, 2 + J^*_{T-2}(f)\} = 7, \quad u^*_{T-3}(e) = \text{down}, \\ J^*_{T-3}(g) &= 2, \quad u^*_{T-3}(g) = \text{up}, \\ J^*_{T-3}(d) &= 3 + J^*_{T-2}(e) = 11, \quad u^*_{T-3}(d) = \text{right}, \\ J^*_{T-3}(f) &= 5, \quad u^*_{T-3}(f) = \text{right}, \\ J^*_{T-3}(a) &= 8 + J^*_{T-2}(d) = 19, \quad u^*_{T-3}(a) = \text{right}, \\ J^*_{T-3}(c) &= \min\{5 + J^*_{T-2}(d),\, 3 + J^*_{T-2}(f)\} = 8, \quad u^*_{T-3}(c) = \text{right}. \end{aligned}$$

We can now see that it is possible to accomplish the objective of going from point a to h in three time steps on path $a \to drightarrowerightarrow f$), and that we would 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 dynamic programming recursion:

$$\begin{aligned}
J^*_{T-4}(e) &= 7, & u^*_{T-4}(e) &= \text{down},\\
J^*_{T-4}(g) &= 2, & u^*_{T-4}(g) &= \text{up},\\
J^*_{T-4}(d) &= 3 + J^*_{T-3}(e) = 10, & u^*_{T-4}(d) &= \text{right},\\
J^*_{T-4}(f) &= 5, & u^*_{T-4}(f) &= \text{right},\\
J^*_{T-4}(a) &= 8 + J^*_{T-3}(d) = 19, & u^*_{T-4}(a) &= \text{right}\\
J^*_{T-4}(c) &= \min\{5 + J^*_{T-3}(d),\, 3 + J^*_{T-3}(f)\} = 8, & u^*_{T-4}(c) &= \text{right},\\
J^*_{T-4}(b) &= 9 + J^*_{T-3}(c) = 17, & u^*_{T-4}(b) &= \text{right},
\end{aligned}$$

and finally with one more iteration:

$$\begin{aligned}
J^*_{T-5}(e) &= 7, & u^*_{T-5}(e) &= \text{down},\\
J^*_{T-5}(g) &= 2, & u^*_{T-5}(g) &= \text{up},\\
J^*_{T-5}(d) &= 10, & u^*_{T-5}(d) &= \text{right},\\
J^*_{T-5}(f) &= 5, & u^*_{T-5}(f) &= \text{right},\\
J^*_{T-5}(a) &= \min\{8 + J^*_{T-4}(d),\, 5 + J^*_{T-4}(b)\} = 18, & u^*_{T-5}(a) &= \text{right}\\
J^*_{T-5}(c) &= \min\{5 + J^*_{T-4}(d),\, 3 + J^*_{T-4}(f)\} = 8, & u^*_{T-5}(c) &= \text{right},\\
J^*_{T-5}(b) &= 9 + J^*_{T-4}(c) = 17, & u^*_{T-5}(b) &= \text{right}.
\end{aligned}$$

Further iterations would no longer change the costs and optimal decisions, so the algorithm has converged. We can finally see that with a sufficiently long horizon, in this case $T \geq 5$, the optimal path from a to h is $a \to d \to e \to f \to g \to h$ and incurs a cost of 18. Note that the dynamic programming algorithm has actually given us a lot more information than what we specifically needed, which was just the optimal sequence from a to h. In particular, given *any* starting point and *any* horizon, we can now easily generate an optimal control sequence to h. For example, if we wanted to start at point c and get to h in $T = 3$ steps, we can immediately see that the optimal path is $c \to f \to g \to h$ and the optimal cost is 8.

19.2 Stochastic Decision Making Problem

Many interesting real-world robotics problems involve uncertainty in how the state changes over time, and therefore designing algorithms based on the deterministic state transition model in Equation (19.1) may not be sufficient to achieve robust autonomy. Instead of leveraging the deterministic problem in Definition 19.1.1, we can instead consider a *stochastic* decision making problem that leverages a stochastic discrete-time state transition model:

$$\boldsymbol{x}_{t+1} = f_t(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{w}_t), \quad t = 0, \ldots, T-1, \tag{19.5}$$

where $\boldsymbol{w}_t$ represents a stochastic disturbance. We also assume that the disturbance, $\boldsymbol{w}_t$, has a known conditional probability distribution, $p_t(\boldsymbol{w}_t \mid \boldsymbol{x}_t, \boldsymbol{u}_t)$,

which can change over time. Note that we assume the disturbance is only dependent on the current state, $\boldsymbol{x}_t$, and control, $\boldsymbol{u}_t$, which is another example of the Markov assumption that we used to develop Bayesian algorithms for probabilistic filtering. Considering a stochastic state transition model means we also modify the cost function to account for the uncertainty in the future state trajectory. It is common practice to define the cost in the stochastic problem as the expected value[12]:

$$J^{\pi}(\boldsymbol{x}_0) = \mathbb{E}_{\boldsymbol{w}}\left[g_T(\boldsymbol{x}_T) + \sum_{t=0}^{T-1} g_t(\boldsymbol{x}_t, \pi_t(\boldsymbol{x}_t), \boldsymbol{w}_t)\right], \tag{19.6}$$

[12] Using the expected value for the cost, which minimizes the cost *on average*, is often referred to as a *risk-neutral* formulation.

where the expectation is over the stochastic variables, $\boldsymbol{w}$.

Another significant difference between the stochastic and deterministic decision making problems is that the stochastic problem solves for a control *policy* rather than an open-loop control sequence. Control policies, which we usually denote $\boldsymbol{u} = \pi(\boldsymbol{x})$, are functions that map the state, $\boldsymbol{x}$, to a control, $\boldsymbol{u}$, and therefore define a closed-loop control law. The search for optimal control *policies* makes the stochastic problem more difficult to solve but is required to have robust closed-loop behavior. We now define the stochastic decision making problem in Definition 19.2.1.

Definition 19.2.1 (Stochastic Decision Making Problem)**.** The stochastic decision making problem for the stochastic state transition model in Equation (19.5), control constraints in Equation (19.2), and cost function in Equation (19.6) is to compute the finite horizon sequence of policies, $\pi := \{\pi_0, \ldots, \pi_{T-1}\}$, that solves:

$$\begin{aligned} J^*(\boldsymbol{x}_0) = \underset{\pi}{\text{minimize}}\ & J_\pi(\boldsymbol{x}_0), \\ \text{s.t. } & \boldsymbol{x}_{t+1} = f_t(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{w}_t), \quad k = 0, \ldots, T-1, \\ & \pi_t(\boldsymbol{x}_t) \in \mathcal{U}(\boldsymbol{x}_t), \ t = 0, \ldots, T-1 \end{aligned} \tag{19.7}$$

19.2.1 *Principle of Optimality (Stochastic Case)*

The principle of optimality also applies to the stochastic setting, and while the proof is slightly different due to having to reason about probability distributions, the intuition is identical to the deterministic case. In the stochastic setting, the principle of optimality is:

Theorem 19.2.2 (Principle of Optimality (Stochastic Case))**.** *Let $\pi^* = \{\pi_0^*, \pi_1^* \ldots, \pi_{T-1}^*\}$ be an optimal policy for the stochastic decision making problem in 19.7, and assume the state, $\boldsymbol{x}_t$, is reachable. Then, the tail policy sequence, $\{\pi_t^*, \ldots, \pi_{T-1}^*\}$, is an optimal policy sequence when starting from $\boldsymbol{x}_t$ to minimize the cost from time t to time T.*

As in the deterministic case, we can again leverage this principle to simplify algorithms for solving the decision making problem by optimizing over immediate decisions based on known optimal tail policies.

19.2.2 *Dynamic Programming (Stochastic Case)*

The dynamic programming algorithm for the stochastic setting, defined in Algorithm 19.2, is similar to Algorithm 19.1 for the deterministic case. This algorithm

Algorithm 19.2: Dynamic Programming (Stochastic Case)

$J_T^*(\boldsymbol{x}) = g_T(\boldsymbol{x})$, for all $\boldsymbol{x} \in \mathcal{X}$
for $t = T-1$ **to** 0 **do**
  $J_t^*(\boldsymbol{x}) = \min_{\boldsymbol{u} \in \mathcal{U}(\boldsymbol{x})} \mathbb{E}_{\boldsymbol{w}} \left[g_t(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}) + J_{t+1}^*(f_t(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w})) \right]$, for all $\boldsymbol{x} \in \mathcal{X}$
return $J_0^*(\cdot), \ldots, J_T^*(\cdot)$

computes the optimal costs, $J_t^*(\boldsymbol{x})$, for each time step, t, and for all states, $\boldsymbol{x}$. Once we have computed these values, we can extract the optimal policy by:

$$\pi_t^*(\boldsymbol{x}_t) = \underset{\boldsymbol{u}_t \in \mathcal{U}(\boldsymbol{x}_t)}{\arg\min}\ \mathbb{E}_{\boldsymbol{w}_t} \left[g_t(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{w}_t) + J_{t+1}^*(f_t(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{w}_t)) \right]. \tag{19.8}$$

Example 19.2.1 (Stochastic Dynamic Programming)**.** Consider an inventory control problem where the available stock of a particular item is the state, $x_t \in \mathbb{N}$, the ability to add to the inventory is the control, $u_t \in \mathbb{N}$, and the demand for the item is a stochastic variable, $w_t \in \mathbb{N}$. We model the dynamics of the available stock as:

$$x_{t+1} = \max\{0, x_t + u_t - w_t\},$$

which models the fact that demand reduces available stock but the stock can also never be negative. We also consider the control constraints:

$$x_t + u_t \leq 2,$$

which limits the amount of additional inventory that we can add based on the current available stock to ensure that $x_t \leq 2$. We model the stochastic demand, w_t, by the probability distribution:

$$p(w_t = 0) = 0.1, \quad p(w_t = 1) = 0.7, \quad p(w_t = 2) = 0.2.$$

Finally, we define the cost for a horizon of $T = 3$ as:

$$\mathbb{E}\left[\sum_{t=0}^{2} u_t + (x_t + u_t - w_t)^2 \right],$$

which penalizes ordering new stock at each time step and also having available stock at the next time step, since the extra would have to be stored.

We apply the stochastic dynamic programming algorithm by starting with the terminal costs:

$$J_3(x_3) = 0,$$

and then recursively computing:

$$J_2(0) = \underset{u_2 \in \{0,1,2\}}{\text{minimize}}\ \mathbb{E}\left[u_2 + (u_2 - w_2)^2\right] = \underset{u_2 \in \{0,1,2\}}{\text{minimize}}\ u_2 + 0.1u_2^2 + 0.7(u_2 - 1)^2 + 0.2(u_2 - 2)^2 = 1.3,$$
$$J_2(1) = \underset{u_2 \in \{0,1\}}{\text{minimize}}\ \mathbb{E}\left[u_2 + (1 + u_2 - w_2)^2\right] = 0.3,$$
$$J_2(2) = \mathbb{E}\left[(2 - w_2)^2\right] = 1.1,$$

where the last cost is easily evaluated since the constraint makes the control $u_2 = 0$ 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:

$$J_1(0) = \underset{u_1 \in \{0,1,2\}}{\text{minimize}}\ \mathbb{E}\left[u_1 + (u_1 - w_1)^2 + J_2(\max\{0, u_1 - w_1\})\right] = 2.5,$$
$$J_1(1) = \underset{u_1 \in \{0,1,\}}{\text{minimize}}\ \mathbb{E}\left[u_1 + (1 + u_1 - w_1)^2 + J_2(\max\{0, 1 + u_1 - w_1\})\right] = 1.5,$$
$$J_1(2) = \mathbb{E}\left[(2 - w_1)^2 + J_2(\max\{0, 2 - w_1\})\right] = 1.68,$$

with optimal stage policies:

$$\pi_1^*(0) = 1,$$
$$\pi_1^*(1) = 0,$$
$$\pi_1^*(2) = 0.$$

Finally, in the last step:

$$J_0(0) = \underset{u_0 \in \{0,1,2\}}{\text{minimize}}\ \mathbb{E}\left[u_0 + (u_0 - w_0)^2 + J_1(\max\{0, u_0 - w_0\})\right] = 3.7,$$
$$J_0(1) = \underset{u_0 \in \{0,1,\}}{\text{minimize}}\ \mathbb{E}\left[u_0 + (1 + u_0 - w_0)^2 + J_1(\max\{0, 1 + u_0 - w_0\})\right] = 2.7,$$
$$J_0(2) = \mathbb{E}\left[(2 - w_0)^2 + J_1(\max\{0, 2 - w_0\})\right] = 2.818,$$

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 having no stock or too much stock. We can also note that the optimal policy ends up being the same at all time steps: if you have no stock you add one item, otherwise you do nothing.

19.3 Markov Decision Processes

In our introduction of the deterministic and stochastic sequential decision making problems in Section 19.1 and Section 19.2, we use a problem formulation based on common notation from the field of optimal control. An equivalent problem formulation for *stochastic* sequential decision problems is the *Markov decision process (MDP)*, which is widely used in robotics and, in particular, the field of reinforcement learning.

A *Markov decision process* is defined for a system by a state space, a control or action space, a stochastic state transition model, and a reward function[13]. Following our notational convention from previous chapters, we denote the state space as $\mathcal{X}$ and the control space as $\mathcal{U}$. Note that this notation differs from the standard in the reinforcement learning literature, which typically denotes the state space as $\mathcal{S}$ and refers to controls as *actions* and denotes the action space as $\mathcal{A}$.

[13] The MDP formulation uses a *reward* function, which is just a negative expression of a *cost* function.

In contrast to the stochastic decision making problem discussed in Section 19.2, which uses the model in Equation (19.5), the MDP formulation equivalently represents the stochastic state transition model as the probability distribution:

$$p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t), \tag{19.9}$$

which is the conditional probability distribution over the next state, $\boldsymbol{x}_{t+1}$, given the current state and control. Note that by modeling the state transition probability as a function of only the previous state and control, and not the full state and control histories, we are again employing the *Markov assumption* that we also leveraged in Chapter 13 on Bayesian filtering. We also assume the state transition model is *stationary* and therefore does not change in time, which is a common assumption in MDP problem formulations. We encode the decision making objective in a MDP by the reward function:

$$R(\boldsymbol{x}_t, \boldsymbol{u}_t). \tag{19.10}$$

which defines the reward for taking the control $\boldsymbol{u}_t$ from state $\boldsymbol{x}_t$, and is equivalent to the negative of the stage cost, g_t, from Equation (19.3). For finite horizon problems with horizon T, the goal is to compute a sequence of policies, $\pi := \{\pi_0, \ldots, \pi_{T-1}\}$, that define the controls $\boldsymbol{u}_t = \pi_t(\boldsymbol{x}_t)$ that maximizes the risk-neutral *expected* value:

$$V_T^{\pi}(\boldsymbol{x}) = \mathbb{E}\left[\sum_{t=0}^{T-1} R(\boldsymbol{x}_t, \pi_t(\boldsymbol{x}_t)) \mid \boldsymbol{x}_0 = \boldsymbol{x}\right], \tag{19.11}$$

where $V_T^{\pi}(\boldsymbol{x})$ is the *value function*[14] for applying the policy π to the system with stochastic dynamics defined by Equation (19.9) for T steps. With each of these components, we can define problem formulation for the finite horizon Markov decision process:

[14] This is analogous to the *cost-to-go*, J^{π}, defined earlier.

Definition 19.3.1 (Finite Horizon MDP Problem)**.** The finite horizon MDP problem with states $\boldsymbol{x} \in \mathcal{X}$, controls $\boldsymbol{u} \in \mathcal{U}$, the probabilistic state transition model defined by Equation (19.9), and the finite horizon value function in Equation (19.11) is to compute the finite horizon sequence of policies, $\pi := \{\pi_0, \ldots, \pi_{T-1}\}$, that solves:

$$\pi^*(\boldsymbol{x}) = \arg\max_{\pi} \; V_T^{\pi}(\boldsymbol{x}), \tag{19.12}$$

It is also common to consider the *infinite-horizon* problem. In this case, our goal is to compute a *stationary* policy, $\pi(\boldsymbol{x})$, that maximizes an infinite horizon value function defined as the *discounted*[15] cumulative expected value:

$$V^{\pi}(\boldsymbol{x}) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t)) \mid \boldsymbol{x}_0 = \boldsymbol{x}\right], \tag{19.13}$$

where we refer to $\gamma \in (0, 1)$ as the *discount factor*. We therefore define the infinite horizon problem as:

[15] We can view the discount as modeling the fact that we are more confident about the short-term impacts of our actions, but it is also required mathematically to ensure the value function is bounded, assuming the reward function, $R(\boldsymbol{x}, \boldsymbol{u})$, is bounded.

Definition 19.3.2 (Infinite Horizon MDP Problem)**.** The infinite horizon MDP problem with states $\boldsymbol{x} \in \mathcal{X}$, controls $\boldsymbol{u} \in \mathcal{U}$, the probabilistic state transition model defined by Equation (19.9), and the finite horizon value function in Equation (19.11) is to compute the stationary policy, π, that solves:

$$\pi^*(\boldsymbol{x}) = \arg\max_{\pi} \; V^{\pi}(\boldsymbol{x}), \tag{19.14}$$

19.3.1 Dynamic Programming (Markov Decision Processes)

Recall that the MDP problem formulation is conceptually the same as the stochastic decision making problem from Section 19.2 and, importantly, the principle of optimality and the dynamic programming technique are equally applicable to this formulation. In the finite horizon MDP problem formulation, we can express the value function from Equation (19.11) recursively as:

$$V_{k+1}^{\pi}(\boldsymbol{x}) = R(\boldsymbol{x}, \pi(\boldsymbol{x})) + \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \pi(\boldsymbol{x})) V_k^{\pi}(\boldsymbol{x}'). \tag{19.15}$$

Then, we again leverage the principle of optimality to express the optimal value function as:

$$V_{k+1}^{*}(\boldsymbol{x}) = \max_{\boldsymbol{u} \in \mathcal{U}} R(\boldsymbol{x}, \boldsymbol{u}) + \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_k^{*}(\boldsymbol{x}') \tag{19.16}$$

where $V_k^*(\boldsymbol{x}) = V_k^{\pi^*}(\boldsymbol{x})$ is the optimal value function for the k-step horizon. We then leverage this recursion for the dynamic programming algorithm for the finite horizon case in Algorithm 19.3. We can then extract the optimal policy, π_t^*, by computing:

$$\pi_t^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u} \in \mathcal{U}} \; R(\boldsymbol{x}, \boldsymbol{u}) + \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_{T-1-t}^{*}(\boldsymbol{x}'). \tag{19.17}$$

Algorithm 19.3: Dynamic Programming (Finite Horizon MDP)

$V_0^*(\boldsymbol{x}) = 0$, for all $\boldsymbol{x} \in \mathcal{X}$
for $k = 0$ **to** $T - 1$ **do**
  $V_{k+1}^*(\boldsymbol{x}) = \max_{\boldsymbol{u} \in \mathcal{U}} R(\boldsymbol{x}, \boldsymbol{u}) + \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_k^*(\boldsymbol{x}')$, for all $\boldsymbol{x} \in \mathcal{X}$
return $V_0^*(\cdot), \ldots, V_T^*(\cdot)$

In the infinite horizon MDP problem formulation, we can similarly express the value function from Equation (19.13) recursively as:

$$V^\pi(\boldsymbol{x}) = R(\boldsymbol{x}, \pi(\boldsymbol{x})) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \pi(\boldsymbol{x})) V^\pi(\boldsymbol{x}'), \tag{19.18}$$

and again leverage the principle of optimality to express the optimal value function as:

$$V^*(\boldsymbol{x}) = \max_{\boldsymbol{u} \in \mathcal{U}} R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^*(\boldsymbol{x}') \tag{19.19}$$

where $V^*(\boldsymbol{x}) = V^{\pi^*}(\boldsymbol{x})$ is the optimal value function and where we commonly refer to the equation for $V^*(\boldsymbol{x})$ as the *Bellman optimality equation*. The dynamic programming-based algorithm for the infinite horizon case[16] is shown in Algorithm 19.4. From the optimal value function, $V^*(\boldsymbol{x})$, from Algorithm 19.4, we

[16] In the context of reinforcement learning, this algorithm is commonly referred to as *value iteration*.

Algorithm 19.4: Dynamic Programming (Infinite Horizon MDP)

$V_0^*(\boldsymbol{x}) = 0$, for all $\boldsymbol{x} \in \mathcal{X}$
for $k = 0$ **to** ∞ **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 $\boldsymbol{x} \in \mathcal{X}$
  if V^* *converged* **then**
    return $V^*(\cdot)$

can extract the optimal policy by computing:

$$\pi^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u} \in \mathcal{U}} R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^*(\boldsymbol{x}'). \tag{19.20}$$

Note that in the infinite horizon algorithm, we initialize the value function to zero, but in practice we can initialize this to any bounded value, and initializing by a guess of the optimal value would help speed up convergence.

Example 19.3.1 (Finite Horizon MDP Dynamic Programming)**.** Consider again the inventory control problem from Example 19.2.1, where the state, $x_t \in \mathbb{N}$, is the available stock of a particular item, the control, $u_t \in \mathbb{N}$, adds items to the inventory, the demand, w_t, is uncertain, and the state transition and constraints are:

$$x_{t+1} = \max\{0, x_t + u_t - w_t\},$$
$$p(w = 0) = 0.1, \quad p(w = 1) = 0.7, \quad p(w = 2) = 0.2.$$

and:

$$x_t + u_t \leq 2.$$

Based on the state transition model, we can define the probabilistic state transition model of the form in Equation (19.9) as:

$$\begin{aligned}
p(x_{t+1} = \{0,1,2\} \mid x_t = 0, u_t = 0) &= \{1,0,0\},\\
p(x_{t+1} = \{0,1,2\} \mid x_t = 0, u_t = 1) &= \{0.9,0.1,0\},\\
p(x_{t+1} = \{0,1,2\} \mid x_t = 0, u_t = 2) &= \{0.2,0.7,0.1\},\\
p(x_{t+1} = \{0,1,2\} \mid x_t = 1, u_t = 0) &= \{0.9,0.1,0\},\\
p(x_{t+1} = \{0,1,2\} \mid x_t = 1, u_t = 1) &= \{0.2,0.7,0.1\},\\
p(x_{t+1} = \{0,1,2\} \mid x_t = 2, u_t = 0) &= \{0.2,0.7,0.1\},
\end{aligned}$$

where we have omitted some transition values due to the control constraints. Next, we define the reward function as:

$$\begin{aligned}
R(x_t, u_t) &= -\mathbb{E}\left[u_t + (x_t + u_t - w_t)^2\right],\\
&= -\big(u_t + (x_t + u_t - \mathbb{E}\left[w_t\right])^2 + Var[w_t]\big).
\end{aligned}$$

We then apply Algorithm 19.3 starting with the value function with no steps to go:

$$V_0^*(x) = 0,$$

and then recursively compute:

$$\begin{aligned}
V_1^*(0) &= \max_{u \in \{0,1,2\}} -\big(u + (u - 1.1)^2 + 0.29\big) = -1.3,\\
V_1^*(1) &= \max_{u_2 \in \{0,1\}} -\big(u + (1 + u - 1.1)^2 + 0.29\big) = -0.3,\\
V_1^*(2) &= -\big((2 - 1.1)^2 + 0.29\big) = -1.1,
\end{aligned}$$

where $\mathbb{E}\left[w\right] = 1.1$ and $Var(w) = 0.29$. The optimal stage policies associated with this step are:

$$\pi_{T-1}^*(0) = 1, \quad \pi_{T-1}^*(1) = 0, \quad \pi_{T-1}^*(2) = 0.$$

In the next step:

$$\begin{aligned}
V_2^*(0) &= \max_{u \in \{0,1,2\}} -\big(u + (u - 1.1)^2 + 0.29\big) + \sum_{x'} p(x' \mid x = 0, u) V_1^*(x') = -2.5,\\
V_2^*(1) &= \max_{u \in \{0,1,\}} -\big(u + (1 + u - 1.1)^2 + 0.29\big) + \sum_{x'} p(x' \mid x = 1, u) V_1^*(x') = -1.5,\\
V_2^*(2) &= -\big((2 - 1.1)^2 + 0.29\big) + \sum_{x'} p(x' \mid x = 2, u = 0) V_1^*(x') = -1.68,
\end{aligned}$$

with optimal stage policies:

$$\pi_{T-2}^*(0) = 1, \quad \pi_{T-2}^*(1) = 0, \quad \pi_{T-2}^*(2) = 0.$$

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:

$$\pi_{T-3}^*(0) = 1, \quad \pi_{T-3}^*(1) = 0, \quad \pi_{T-3}^*(2) = 0.$$

As expected, these results are identical to the results from Example 19.2.1 since the problem is the same, we have just formulated it using the MDP framework.

19.4 Limitations of Dynamic Programming

Dynamic programming is a powerful foundational algorithm that is the backbone for a collection of methods for solving sequential decision-making problems, but it also suffers from several practical limitations. First, in their standard form, dynamic programming-based algorithms require *perfect knowledge* of the environment, including the deterministic or stochastic state transition model and the reward function. In many practical scenarios, this requirement may be limiting if the system's dynamics are too complex to model accurately, such as for applications involving friction, contact forces, and other non-linear interactions with the environment, or if the dynamics are stochastic. The dynamic programming-based algorithms presented in this chapter also assume that the full state of the system is known and observable, which is not always the case in practice since it maybe not be possible or economical to have sensors that measure every single state component directly. In settings with state uncertainty[17], the problem becomes even harder to solve from a computational perspective since we have to reason about policies defined over *belief* probability distributions over the state space, which effectively increases the dimension of the problem.

[17] In contexts where we express stochastic decision making problems as Markov decision processes (MDP), we refer to the extension to handle state uncertainty as a *partially observable Markov decision process (POMDP)*.

Another significant practical limitation of dynamic programming is referred to as the *curse of dimensionality*. The curse of dimensionality is that the computational and storage requirements grow exponentially with the dimension of the state space. Specifically, if the state is n-dimensional and each state variable can take on M different discrete values, then at each step of the algorithm the Bellman equation must be solved M^n times. While this may be practically possible to implement for small problems, it can grow out of hand very quickly. This is particularly relevant in robotics, where the state space can be very high-dimensional. For example, the state of a robot can include joint angles and their velocities in three-dimensions, actuator states, and more.

These challenges related to dynamic programming motivate the development of approximate dynamic programming approaches that are more practical for

specific settings, such as with high-dimensional states, when the model is not known, and more. In the next chapter, we introduce methods from the field of *reinforcement learning* that address the sequential decision making through the lens of *learning from interaction* to remove the requirement that the environment is fully modeled.

References

[6] D. Bertsekas. *Reinforcement learning and optimal control*. Athena Scientific, 2019.

[32] M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray. *Algorithms for Decision Making*. MIT Press, 2022.

20 Reinforcement Learning

In Chapter 19, we introduced the deterministic and stochastic sequential decision-making problems and the Markov decision process (MDP) framework, and demonstrated how these problems can be solved by dynamic programming. However, these approaches are limited by their assumption of knowledge of the system dynamics and their exponential growth in computational requirements with the dimensionality of the state space. In this chapter, we provide an introduction and overview of the field of *Reinforcement Learning*[1,2]. At a high level, reinforcement learning is the process of *learning what to do*[3] from interaction with the environment. In other words, reinforcement learning is the process of learning from interaction, whereby, rather than being told what to do, the learner must discover an effective strategy through trial and error and by receiving feedback from the environment. Unlike the methods covered in the previous chapter, in the context of reinforcement learning, we do not make assumptions regarding the knowledge or observability of the environment. This makes reinforcement learning a practical and general framework for autonomous decision-making.

[1] D. Bertsekas. *Reinforcement learning and optimal control*. Athena Scientific, 2019

[2] R. Sutton and A. Barto. *Reinforcement learning: An introduction*. MIT Press, 2018

[3] Specifically, learning how to map states to controls with the objective of maximizing a numerical reward signal.

In this chapter, we begin in Section 20.1 by introducing key concepts and theoretical foundations of the reinforcement learning problem. Next, in Section 20.2, we introduce reinforcement learning algorithms based on exact dynamic programming that leverage ideas from Chapter 19. Motivated by practical limitations of dynamic programming, we introduce two foundational model-free learning paradigms known as Monte Carlo methods and temporal-difference learning in Section 20.3. Finally, in Section 20.5 and Section 20.6, we discuss widely used model-free and model-based reinforcement learning algorithms.

20.1 The Reinforcement Learning Problem

The reinforcement learning problem is a mathematical formalism for learning-based decision making that captures the essential aspects of a learning agent interacting over time with its environment to achieve a goal or objective. In this section, we present the mathematical formulation, which takes the form of an

optimal control problem within the context of a partially unknown Markov decision process (MDP). Within the landscape of machine learning paradigms, there are a number of key distinctions and specific challenges that are unique to reinforcement learning.

First, in supervised learning contexts[4], the learner is provided with a dataset of labeled examples from a knowledgeable supervisor, and the goal is to learn to imitate the supervisor's behavior. While this is a powerful form of learning, it is not suitable for learning from interaction, where obtaining labeled examples is impractical. Reinforcement learning is instead characterized by a process of *learning without a teacher*, whereby the agent is not told which actions to take, but instead must learn from environment feedback in the form of rewards or penalties.

[4] Supervised learning is a very well-studied category of machine learning.

The second unique and challenging aspect of reinforcement learning problems is that the agent must learn to balance the intrinsic trade-off between *exploration and exploitation*. In this trade-off, the agent must *explore* the environment to discover the best actions while also making sure to *exploit* the knowledge it has already acquired to maximize its reward. If an agent were to exclusively pursue only one of the two strategies, it would either fail to discover the optimal policy or fail to exploit it effectively. In other words, the agent must learn to try a variety of actions and progressively focus on the most rewarding ones.

Third, the feedback from the environment that drives the learning process may be *delayed or sparse*. This is a common feature of many real-world problems[5]. Delays in the reward signal can make it difficult for the agent to correctly perform *credit assignment*[6].

[5] For example, the reward signal may only be received after a sequence of actions has been taken, such as in a game of chess, where feedback is only received at the end of the game when it is either won or lost.

[6] Attributing the reward to the actions that led to it, even if the responsible actions were taken well before the reward was received.

Lastly, in reinforcement learning contexts, the data used for learning is not independent and identically distributed, which is a fundamental assumption underlying the majority of statistical learning theory. Instead, the data is highly correlated since the agent's actions influence the subsequent data it receives.

20.1.1 Elements of Reinforcement Learning

In this and previous chapters, we have encountered and briefly discussed several key elements of a reinforcement learning system: a *policy*, an *environment*, a *reward signal*, a *value function*, and, optionally, a *model* of the environment. In this section, we provide a more formal definition of these elements and discuss how they interact in the context of the reinforcement learning problem.

A *policy*, $\pi(\boldsymbol{u}_t \mid \boldsymbol{x}_t)$, is a mapping from states to actions that defines the agent's behavior. The policy can take the form of a simple function, such as a lookup table or a parametric function, or we can define it by a complex decision-making scheme, such as an explicit search process. Policies are either deterministic, where each state maps to a single action, or stochastic, where a state maps to a distribution over actions[7].

[7] We commonly denote stochastic policies as $\pi(\boldsymbol{u}_t \mid \boldsymbol{x}_t)$ and deterministic policies as $\pi(\boldsymbol{x}_t)$.

The *environment* is the system the agent interacts with. We mathematically represent the environment by a transition model, $p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t)$, which defines

the probability of transitioning to a new state, $\boldsymbol{x}_{t+1}$, given the current state, $\boldsymbol{x}_t$, and action, $\boldsymbol{u}_t$. The transition model may also be either deterministic or stochastic, depending on the nature of the environment.

A *reward signal* defines the goal of the agent. At each time step, we assume the agent receives a scalar *reward*, $r_t \in \mathbb{R}$, from the environment that indicates how well the agent is performing. Reward signals are deterministic or stochastic functions of the state of the environment and the action taken, and we denote the function that produces the reward as $R(\boldsymbol{x}_t, \boldsymbol{u}_t)$.

While the reward signal represents an immediate measure of performance, the *value function* represents performance in the long run. Specifically, the value of a state defines how much reward the agent can expect to accumulate from that state onwards. For example, a state might have a low immediate reward but a high value if it usually leads to states with high rewards, and vice versa. An effective agent chooses actions by considering the value of the action rather than just the immediate reward[8].

[8] Therefore, many methods in reinforcement learning are centered around accurately estimating the value of an action.

Lastly, a *model* of the environment is an optional component of the reinforcement learning problem that represents the agent's understanding of the environment. The model's goal is to mimic the behavior of the environment, and we can use it to make hypotheses about how the environment will evolve[9]. In this chapter, we explore reinforcement learning algorithms that use models for learning, referred to as *model-based* algorithms, as well as more direct *model-free* algorithms that do not attempt to learn a model of the environment and solely focus on discovering optimal policies by trial-and-error learning.

[9] For example, we can use models to evaluate different actions before executing them.

At a high level, most reinforcement learning algorithms follow the same basic learning cycle. First, the agent interacts with the environment by observing the state, $\boldsymbol{x}$, applying an action, $\boldsymbol{u}$, from a chosen *behavior policy*[10], and then observing the next state, $\boldsymbol{x}'$, and scalar reward, $r = R(\boldsymbol{x}, \boldsymbol{u})$. This procedure, which we show in Figure 20.1, may repeat for multiple steps, during which the agent uses the observed transitions, $(\boldsymbol{x}, \boldsymbol{u}, r, \boldsymbol{x}')$, to update its policy.

[10] The behavior policy does not necessarily have to match the learned policy.

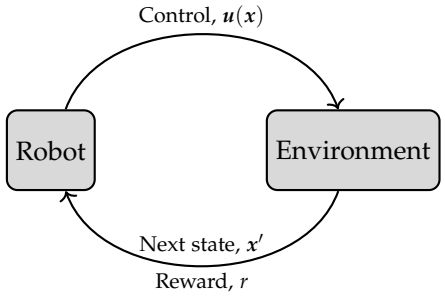


Figure 20.1: The reinforcement learning problems consists of a robot (agent) that learns how to make decisions by interacting with the environment.

20.1.2 *Problem Formulation*

In Section 19.3, we introduced the Markov decision process (MDP) framework in the context of sequential decision making problems where we assume the

environment's dynamics are known. The mathematical framework for modeling the reinforcement learning problem is also built around the Markov decision process, and so in this section we reintroduce the key concepts of MDPs. Note that there will be some small differences in the MDP formulation that we use in this chapter which are more common in the reinforcement learning community. For example, it is common in the context of reinforcement learning to use stochastic policies, $\pi(\boldsymbol{u} \mid \boldsymbol{x})$, while in Section 19.3 we presented the MDP formulation using a deterministic policy, $\pi(\boldsymbol{x})$. At a high level, MDPs represent a formalization of the sequential decision-making problem, where the agent's actions influence not only the immediate reward but also the subsequent states, and through those the future rewards.

We formally refer to an MDP as a tuple of elements $\mathcal{M} = (\mathcal{X}, \mathcal{U}, p, R, \gamma)$, where $\mathcal{X}$ is the discrete or continuous *state space*, $\mathcal{U}$ is the discrete or continuous *action space*, $p(\cdot)$ describes the dynamics of the system through a conditional probability distribution of the form $p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t)$, $R : \mathcal{X} \times \mathcal{U} \to \mathbb{R}$ defines a reward function, and $\gamma \in (0, 1]$ is a scalar discount factor. From a reinforcement learning perspective, our goal is to learn a policy defined as a probability distribution over actions to take from a given state, $\pi(\boldsymbol{u} \mid \boldsymbol{x})$. A *trajectory* is a sequence of states and actions of length T, given by:

$$\tau \coloneqq (\boldsymbol{x}_0, \boldsymbol{u}_0, \ldots, \boldsymbol{x}_T),$$

where T may be infinite. The *trajectory distribution*, p_π, for a policy, π, is:

$$p_\pi(\tau) = \prod_{t=0}^{T-1} \pi(\boldsymbol{u}_t \mid \boldsymbol{x}_t) p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t). \tag{20.1}$$

The reinforcement learning objective, V_T^π, is the expectation of future discounted cumulative reward under this trajectory distribution:

$$V_T^\pi \coloneqq \mathbb{E}_{\tau \sim p_\pi(\tau)} \left[\sum_{t=0}^{T-1} \gamma^t R(\boldsymbol{x}_t, \boldsymbol{u}_t) \right]. \tag{20.2}$$

An additional concept required to fully characterize V_T^π is that of *discounting*. In particular, the discount factor, γ, is a scalar value in the range $[0, 1]$ that determines the relative importance of future rewards, whereby a smaller γ will make the agent focus more on immediate rewards, while a larger γ will make the agent give more importance to future rewards. For example, in the limit case with $\gamma = 0$, the agent will only consider immediate rewards, while in the case with $\gamma = 1$, the agent will consider all future rewards equally. Mathematically, discounting is also crucial in ensuring that the sum of rewards in V^π is finite even in the infinite horizon case with $T = \infty$, such that if $\gamma < 1$ and the rewards, r_t, are bounded, the sum of the rewards will be finite. In practice, the choice of γ is often problem-dependent, and it is common to use a value close to 1 to ensure that the agent considers future rewards.

20.1.3 Value Functions and Bellman Equations

Almost all reinforcement learning algorithms involve estimating *value functions*. At its core, a value function is a function of state or state-action pairs that defines how good[11] it is for the agent to be in a given state or to take a given action in a given state. Since the reward an agent expects to receive in the future depends on the actions it will take, the value function is inherently defined with respect to a particular policy, π.

[11] As defined in the previous section, value functions define quality in terms of expected cumulative future rewards.

We define the *state-value function*, $V^\pi(\boldsymbol{x})$, as the expected sum of future rewards when starting in state $\boldsymbol{x}$ and following policy π thereafter[12]:

[12] Throughout this chapter, we primarily consider the infinite-horizon case when referring to V^π, although the same concepts extend to the finite-horizon case.

$$V^\pi(\boldsymbol{x}) \coloneqq \mathbb{E}_{\tau \sim p_\pi(\tau)} \left[\sum_{k=0}^{\infty} \gamma^{t+k} R(\boldsymbol{x}_{t+k}, \pi(\boldsymbol{x}_{t+k})) \mid \boldsymbol{x}_t = \boldsymbol{x} \right]. \tag{20.3}$$

Similarly, the *action-value function*, $Q^\pi(\boldsymbol{x}, \boldsymbol{u})$, is the expected return when starting in state $\boldsymbol{x}$, taking action $\boldsymbol{u}$, and then following policy π thereafter:

$$Q^\pi(\boldsymbol{x}, \boldsymbol{u}) \coloneqq \mathbb{E}_{\tau \sim p_\pi(\tau)} \left[\sum_{k=0}^{\infty} \gamma^{t+k} R(\boldsymbol{x}_{t+k}, \pi(\boldsymbol{x}_{t+k})) \mid \boldsymbol{x}_t = \boldsymbol{x}, \pi(\boldsymbol{x}_t) = \boldsymbol{u} \right]. \tag{20.4}$$

A key property of value functions used in the context of reinforcement learning and dynamic programming is that they satisfy the *Bellman equations*. The Bellman equations describe a recursive relationship that decomposes the value of a state or state-action pair into the immediate reward and the value of the next state or state-action pair. Formally, for any policy, π, and any state, $\boldsymbol{x}$, the Bellman equation defines the self-consistency condition:

$$V^\pi(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{u} \sim \pi(\cdot|\boldsymbol{x})} \left[R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \mathbb{E}_{\boldsymbol{x}' \sim p(\cdot|\boldsymbol{x}, \boldsymbol{u})} \left[V^\pi(\boldsymbol{x}') \right] \right], \tag{20.5}$$

where, to simplify notation, we have omitted the time index, t, and used $\boldsymbol{x}'$ to denote the next state.

Similarly, the Bellman equation for the action-value function is:

$$Q^\pi(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \mathbb{E}_{\boldsymbol{x}' \sim p(\cdot|\boldsymbol{x}, \boldsymbol{u}), \boldsymbol{u}' \sim \pi(\cdot|\boldsymbol{x}')} \left[Q^\pi(\boldsymbol{x}', \boldsymbol{u}') \right]. \tag{20.6}$$

The value functions V^π and Q^π are unique solutions to the Bellman equations. In the remainder of this chapter, we show how we can use the Bellman equations to derive algorithms for estimating and approximating value functions, and how we can use these value functions to derive optimal policies.

Optimal policies and *optimal value functions* are another important concept in the context of reinforcement learning. Value functions define a partial ordering over policies. We consider a policy, π, to be better than or equal to policy π' if its value function is greater than or equal to the value function of the other policy for all states. Formally, $\pi \geq \pi'$ if and only if $V^\pi(\boldsymbol{x}) \geq V^{\pi'}(\boldsymbol{x})$ for all states $\boldsymbol{x}$. An *optimal policy*, π^*, is a policy that is better than or equal to all other policies, such that $\pi^* \geq \pi$ for all policies π. While the optimal policy does not

need to be unique, all optimal policies share the same *optimal value function*, V^*, which satisfies:

$$V^*(\boldsymbol{x}) = \max_{\pi} V^{\pi}(\boldsymbol{x}), \quad \forall x \in \mathcal{X}.$$

Optimal policies also share the same optimal action-value function, $Q^*(\boldsymbol{x}, \boldsymbol{u})$, which satisfies:

$$Q^*(\boldsymbol{x}, \boldsymbol{u}) = \max_{\pi} Q^{\pi}(\boldsymbol{x}, \boldsymbol{u}), \quad \forall x \in \mathcal{X},\, u \in \mathcal{U}.$$

As discussed above, V^* and Q^* are value functions for the optimal policy, thus, they must satisfy the Bellman equations with respect to the optimal policy. However, because V^* and Q^* are the optimal value functions, we can write the Bellman equations in a special form that does not depend on a specific policy by leveraging the fact that the value of a state under the optimal policy is the expected return of the best action in that state.

We can derive the Bellman equations for the optimal state-value function and action-value function[13] by substituting the expectation over the policy from Equation (20.5) and Equation (20.6) with a maximization over actions:

[13] Referred to as the *Bellman optimality equations*.

$$V^*(\boldsymbol{x}) = \max_{\boldsymbol{u}} \left[R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \mathbb{E}_{\boldsymbol{x}' \sim p(\cdot|\boldsymbol{x},\boldsymbol{u})} \left[V^*(\boldsymbol{x}') \right] \right], \tag{20.7}$$

$$Q^*(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \mathbb{E}_{\boldsymbol{x}' \sim p(\cdot|\boldsymbol{x},\boldsymbol{u})} \left[\max_{\boldsymbol{u}'} Q^*(\boldsymbol{x}', \boldsymbol{u}') \right]. \tag{20.8}$$

The optimal value functions, V^* and Q^*, are important in reinforcement learning because we can use them to derive the optimal policy by acting greedily with respect to the optimal value function:

$$\pi^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}} \left[R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \mathbb{E}_{\boldsymbol{x}' \sim p(\cdot|\boldsymbol{x},\boldsymbol{u})} \left[V^*(\boldsymbol{x}') \right] \right]. \tag{20.9}$$

In other words, once we have access to V^*, we can determine the optimal policy through a one-step search and by selecting the actions that lead to states with the highest value.

Access to the optimal action-value function, Q^*, simplifies the process even further. For any state $\boldsymbol{x}$, we can obtain the optimal policy by selecting the action that maximizes Q^*:

$$\pi^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}} Q^*(\boldsymbol{x}, \boldsymbol{u}). \tag{20.10}$$

20.2 Dynamic Programming Methods

As we introduced in Chapter 19, the key idea of dynamic programming is to decompose a complex problem into simpler subproblems. This is achieved by using value functions to systematically organize and structure the search for optimal policies. In this section, we show how we can leverage dynamic programming algorithms in the context of reinforcement learning by turning the Bellman equations into iterative update rules for the estimation of value

functions. In particular, we explore how to use dynamic programming ideas to derive algorithms for two distinct but interconnected tasks: *prediction* and *control*.

Definition 20.2.1 (Prediction)**.** In the context of reinforcement learning, we often refer to the task of estimating the value function for a given policy as *prediction*.

Definition 20.2.2 (Control)**.** In the context of reinforcement learning, we often refer to the task of finding the optimal policy as *control*.

20.2.1 Prediction: Policy Evaluation

We first consider the prediction problem of estimating the value function, V^π, under a given policy, π. According to the Bellman equation in Equation (20.5), the value of a state, $\boldsymbol{x}$, under policy π is defined as an expectation with respect to the policy and state transition model. For simplicity, we assume that the state transition model and policy describe probability distributions over discrete states and actions, respectively, which allows us to express the expectations in the Bellman equation as sums rather than integrals[14]:

$$V^\pi(\boldsymbol{x}) = \sum_{\boldsymbol{u} \in \mathcal{U}} \pi(\boldsymbol{u} \mid \boldsymbol{x}) \left[R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^\pi(\boldsymbol{x}') \right]. \tag{20.11}$$

[14] The extension to continuous states and actions is fundamentally equivalent and just requires the replacement of summations with integrals.

Policy evaluation is an iterative algorithm to solve the prediction problem. Consider a sequence of approximations to the value function, denoted as $V_0, V_1, V_2, \ldots, V^\pi$, where V_0 is an arbitrarily chosen initial guess[15]. Policy evaluation uses the Bellman equation in Equation (20.11) as an update rule, such that at iteration k, the value function for all states $x \in \mathcal{X}$ is updated according to:

$$V_{k+1}(\boldsymbol{x}) = \sum_{\boldsymbol{u} \in \mathcal{U}} \pi(\boldsymbol{u} \mid \boldsymbol{x}) \left[R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_k(\boldsymbol{x}') \right]. \tag{20.12}$$

[15] Under the condition that any *terminal state*, occurring when $t = T$ in the finite-horizon setting or when the episode terminates in the infinite-horizon setting, must be assigned a value of zero.

It is important to note that $V_k = V^\pi$ is a fixed point of the update rule in Equation (20.12) since the Bellman equation for V^π ensures equality in this case. Under mild regularity conditions, we can show that the sequence of value functions, $\{V_k\}$, converges to V^π as $k \to \infty$. As we will see in the remainder of this chapter, the ideas described above are at the core of many reinforcement learning algorithms, including both model-based and model-free methods.

20.2.2 Policy Improvement

Now that we have introduced an approach to solve the prediction problem of estimating the value function under a given policy, we now turn our attention to the problem of *control*, which is to find the optimal policy. To solve the control problem, we leverage the *policy improvement theorem*, which provides a way to update a given policy that is guaranteed to be better than or equal to the

original policy. Consider a pair of policies, π and π', such that for all states $x \in \mathcal{X}$:

$$Q^{\pi}(\boldsymbol{x}, \pi'(\boldsymbol{x})) \geq V^{\pi}(\boldsymbol{x}). \tag{20.13}$$

Then, the policy π' is guaranteed to be better than or equal to π, such that:

$$V^{\pi'}(\boldsymbol{x}) \geq V^{\pi}(\boldsymbol{x}).$$

Consider the greedy policy, π', which selects the action that maximizes the action-value function, Q^{π}, from policy π[16]:

$$\begin{aligned} \pi'(\boldsymbol{x}) &\coloneqq \arg\max_{\boldsymbol{u}} Q^{\pi}(\boldsymbol{x}, \boldsymbol{u}) \\ &= \arg\max_{\boldsymbol{u}} \left[R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^{\pi}(\boldsymbol{x}') \right]. \end{aligned} \tag{20.14}$$

[16] In other words, the greedy policy selects the best action after a one-step lookahead based on the current value function V^{π}.

By construction, this greedy policy is guaranteed to satisfy the condition of the policy improvement theorem from Equation (20.13), and is therefore better than or equal to the original policy. We refer to the process of constructing a new policy by greedily selecting actions with respect to the current value function as *policy improvement*.

Suppose now that the greedy policy, π', is as good as the original policy, π, such that $V^{\pi'} = V^{\pi}$. From the definition of the greedy policy in Equation (20.14):

$$V^{\pi'}(\boldsymbol{x}) = \max_{\boldsymbol{u}} \left[R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^{\pi}(\boldsymbol{x}') \right].$$

However, since $V^{\pi'} = V^{\pi}$, this is the same as the Bellman optimality equation, and therefore the policy π' must be optimal.

20.2.3 *Control: Policy Iteration*

The policy improvement theorem provides a concrete strategy to improve a policy by greedily selecting actions with respect to the current value function. In this section, we discuss how we use this strategy, in tandem with policy evaluation, to construct an algorithm for finding the optimal policy. We refer to this algorithm for finding the optimal policy as *policy iteration*. At a high level, the key idea of policy iteration is that once we have improved a given policy, π, according to V^{π} to yield a new policy, π', we can evaluate this new policy to obtain $V^{\pi'}$. We can then use the new value function, $V^{\pi'}$ to further improve the policy into π'', and repeat this process until convergence to the optimal policy.

More formally, policy iteration defines a sequence of monotonically improving policies by alternating between policy evaluation and policy improvement:

$$V_0 \xrightarrow{\text{PE}} \pi_0 \xrightarrow{\text{PI}} V_1 \xrightarrow{\text{PE}} \pi_1 \xrightarrow{\text{PI}} V_2 \xrightarrow{\text{PE}} \pi_2 \xrightarrow{\text{PI}} \dots,$$

where PE denotes policy evaluation and PI denotes policy improvement. We outline the policy iteration algorithm in Algorithm 20.1. Policy iteration is guar-

Algorithm 20.1: Policy Iteration

Data: Initial policy, π, and value function, V_0, arbitrarily initialized for all $\boldsymbol{x} \in \mathcal{X}$.
Result: Policy, $\pi \approx \pi^*$, and value function, $V^\pi \approx V^*$.
Policy Evaluation:
for $k = 0, \ldots, \infty$ **do**
  for $x \in \mathcal{X}$ **do**
    $V_{k+1}(\boldsymbol{x}) = \sum_{\boldsymbol{u} \in \mathcal{U}} \pi(\boldsymbol{u} \mid \boldsymbol{x}) \left[R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_k(\boldsymbol{x}') \right]$
  if $\| V_{k+1} - V_k \| < \epsilon$ **then**
    $V^\pi = V_{k+1}$
    break
Policy Improvement:
for $x \in \mathcal{X}$ **do**
  $\pi'(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}} \left[R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^\pi(\boldsymbol{x}') \right]$
if *policy has converged* **then**
  $\pi = \pi'$
  break
return $\pi \approx \pi^*$ and $V^\pi \approx V^*$

anteed to converge to the optimal policy and value function, given enough iterations. In practice, since the policy evaluation step is an iterative algorithm, we typically initialize the value function to the value function from the previous step of policy iteration. This can increase the speed of convergence since the value function does not typically change substantially between iterations.

20.2.4 Control: Value Iteration

One drawback of policy iteration is that it requires a full policy evaluation step at each iteration. This makes it computationally expensive because the algorithm must wait for the value function to converge before proceeding to the policy improvement step, which only happens in the limit[17]. To address this issue, *value iteration* is an alternative algorithm that combines policy evaluation and policy improvement into a single step. Value iteration defines the following update rule:

[17] Several variants of policy iteration use truncated policy evaluation steps.

$$V_{k+1}(\boldsymbol{x}) = \max_{\boldsymbol{u}} \left[R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_k(\boldsymbol{x}') \right]. \tag{20.15}$$

For an arbitrary initial value function, V_0, the sequence of value functions, $\{V_k\}$, generated by the value iteration algorithm is guaranteed to converge to the optimal value function, V^*. We can interpret value iteration from the perspective of the Bellman optimality equation, where the update rule in Equation (20.15) is equivalent to the Bellman optimality operator from Equation (20.7) applied to the value function V_k. In other words, we can view policy iteration as the iterative application of the Bellman equation in the policy evaluation step followed

by one step of policy improvement, and we can view value iteration as the iterative application of the Bellman optimality equation. We outline the complete value iteration algorithm in Algorithm 20.2.

Algorithm 20.2: Value Iteration

Data: Initial value function, V_0, arbitrarily initialized for all $\boldsymbol{x} \in \mathcal{X}$.
Result: Policy, $\pi \approx \pi^*$, and value function, $V^\pi \approx V^*$.
for $k = 0$ **to** ∞ **do**
  for $x \in \mathcal{X}$ **do**
    $V_{k+1}(\boldsymbol{x}) = \max_{\boldsymbol{u}} \left[R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_k(\boldsymbol{x}') \right]$
  if $\|V_{k+1} - V_k\| < \epsilon$ **then**
    $V^\pi = V_{k+1}$
    $\pi(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}} \left[R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_k(\boldsymbol{x}') \right]$, for all $\boldsymbol{x} \in \mathcal{X}$
    return $V^\pi \approx V^*$ and $\pi \approx \pi^*$

Policy iteration and value iteration are two of the most foundational algorithms in reinforcement learning and are the basis for many modern reinforcement learning algorithms.

20.2.5 *Inheriting the Limitations of Dynamic Programming*

Dynamic programming methods, including policy iteration and value iteration, are powerful tools for solving MDPs. However, these methods also inherit the limitations of dynamic programming as they require a complete model of the environment, $p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u})$, to compute the expectations in the Bellman equations, and are computationally expensive for large state and action spaces. In the next sections, we discuss two key approaches to address these limitations. We first introduce *sampling methods*, which relax the requirement of having a complete model of the environment, and then we introduce the concept of *function approximation*, which helps address the computational complexity of dynamic programming methods.

20.3 *Reinforcement Learning Paradigms*

Monte Carlo (MC) methods and *temporal-difference (TD) learning* are two classes of learning methods that estimate value functions and compute optimal policies without requiring a complete model of the environment. These approaches are particularly effective in practice since they are applicable to a wide range of problems where modeling the environment dynamics is either impractical or infeasible. Similar to our discussion of dynamic programming methods, we first address the prediction problem for both Monte Carlo and temporal-difference methods before extending the analysis to the control problem.

20.3.1 Monte Carlo Methods

The term *Monte Carlo* broadly refers to a class of algorithms that rely on random sampling to estimate quantities of interest. In the context of reinforcement learning, Monte Carlo methods represent a class of approaches for solving the reinforcement learning problem based on averaging observed cumulative rewards from experience[18]. Monte Carlo methods are particularly well-suited for problems modeled as episodic MDPs, where the interaction between the agent and the environment is divided into *episodes*. Each episode consists of a finite sequence of states, actions, and rewards, starting from an initial state and progressing until the episode terminates. An episode ends when either a terminal state is reached or the horizon, T, is exceeded.

[18] We often use the terms *samples* or *experience* to refer to sequences of states, actions, and rewards collected by interacting with the environment.

Episodic MDPs naturally describe tasks with well-defined beginnings and endings, such as navigating a maze, playing a game, or completing a robotic assembly. At the end of each episode, the agent has access to a complete trajectory of experience, which we can use to estimate quantities of interest. These include updates to value functions, such as the state-value function, $V^\pi(\boldsymbol{x})$, or the action-value function, $Q^\pi(\boldsymbol{x}, \boldsymbol{u})$, or updates to the policy, π.

Monte Carlo methods for solving the prediction problem aim to learn the value function, V^π, or action value function, Q^π, given a policy, π. In this section, we first address the problem of using Monte Carlo methods for learning the state-value function, V^π, and then extend the discussion to learning the action-value function, Q^π. Since the value of a state is defined as the expected cumulative reward starting from the state, we compute the Monte Carlo estimate of the value by averaging the observed cumulative rewards from samples passing through the state. The estimate of the value function will then become more accurate with an increased number of visits to the state.

Specifically, suppose we wish to learn the value, $V^\pi(\boldsymbol{x})$, of the state $\boldsymbol{x}$ under the policy π, given a set of N episodes, $\{\tau_1, \tau_2, \ldots, \tau_N\}$, passing through $\boldsymbol{x}$. We can apply the Monte Carlo method defined in Algorithm 20.3 for estimating the desired state-value function.

Algorithm 20.3: Monte Carlo Prediction

Data: Initial value function, V, arbitrarily initialized for all $\boldsymbol{x} \in \mathcal{X}$.
Result: Value function estimate, V^π.
Initialize the state visit count, $N(\boldsymbol{x}) = 0$, for all $\boldsymbol{x} \in \mathcal{X}$
for *each episode* $\tau_i = \{\boldsymbol{x}_0, \boldsymbol{u}_0, r_0, \boldsymbol{x}_1, \boldsymbol{u}_1, r_1, \ldots, \boldsymbol{x}_T\}$ **do**
  for $t = T - 1$ **to** 0 **do**
    Compute the cumulative future reward from state $\boldsymbol{x}_t$:
    $G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k$
    Update the value estimate: $V(\boldsymbol{x}_t) \leftarrow V(\boldsymbol{x}_t) + \frac{1}{N(\boldsymbol{x}_t)}(G_t - V(\boldsymbol{x}_t))$
    Increment the visit count: $N(\boldsymbol{x}_t) \leftarrow N(\boldsymbol{x}_t) + 1$
return $V \approx V^\pi$

Monte Carlo prediction uses the incremental update rule $V(\boldsymbol{x}_t) \leftarrow V(\boldsymbol{x}_t) + \frac{1}{N(\boldsymbol{x}_t)}(G_t - V(\boldsymbol{x}_t))$ to update the value estimate, $V(\boldsymbol{x}_t)$, towards the average future reward, where $N(\boldsymbol{x}_t)$ is the number of times state $\boldsymbol{x}_t$ has been visited. This incremental update is an efficient way to compute the average of a sequence of values, equivalent to $V(\boldsymbol{x}_t) = \frac{1}{N(\boldsymbol{x}_t)} \sum_{i=1}^{N(\boldsymbol{x}_t)} G_t^i$, where G_t^i is the cumulative future reward observed in the i-th visit to state $\boldsymbol{x}_t$.

We can also use Monte Carlo methods for learning action values[19]. At a high level, Monte Carlo methods for estimating action value functions, $Q^\pi(\boldsymbol{x}, \boldsymbol{u})$, are essentially equivalent to the method presented above for estimating state values, with the only difference being that we now consider visits to state-action pairs instead of states. We consider a state-action pair to have been visited in an episode if the agent is in state $\boldsymbol{x}$ and takes action $\boldsymbol{u}$ at some point during the episode. As we discussed for estimating state values, Monte Carlo methods estimate the action value, $Q^\pi(\boldsymbol{x}, \boldsymbol{u})$, by averaging the observed cumulative rewards from the visits to the state-action pair, $(\boldsymbol{x}, \boldsymbol{u})$.

[19] Learning action values, Q^π, is useful in practice because we can use them to directly derive an optimal policy using Equation (20.10).

However, if we use a deterministic policy to collect samples from the environment, we will only observe rewards for one action in each state[20]. This means we will not be able to estimate the value of the other actions in that state, and therefore will not be able to derive the policy by maximizing over actions. We refer to this as the problem of *maintaining exploration*, which plays a crucial role in reinforcement learning.

[20] Since a deterministic policy will always select the same action from the same state.

As we will see in the remainder of this chapter, a common approach to address this issue is to consider policies that lead to all state-action pairs being encountered with non-zero probability. For example, we can achieve this by considering stochastic policies that assign non-zero probability to all actions in each state.

20.3.2 *Temporal-Difference Learning*

Temporal-difference learning is widely considered one of the most influential concepts in reinforcement learning. At a high level, temporal-difference learning combines elements of both Monte Carlo methods and dynamic programming. Like dynamic programming methods, temporal-difference learning can use incomplete sequences of experience[21] by updating estimates based on other learned estimates in process is known as *bootstrapping*. Additionally, like Monte Carlo methods, temporal-difference learning directly leverages raw experience without requiring a model of the environment in a process known as *sampling*. In this sense, temporal-difference learning serves as a bridge between the two approaches, inheriting the advantages of both.

[21] In other words, without waiting for the end of an episode.

Similar to Monte Carlo methods, temporal-difference methods address the prediction problem by collecting samples from the environment and using the experience to update value estimates. Recall that in Monte Carlo methods, we must wait until the end of the episode to compute the cumulative reward following time t, which we denote as G_t, and then use G_t to define the target for

the value function update:

$$V(\boldsymbol{x}_t) \leftarrow V(\boldsymbol{x}_t) + \alpha(G_t - V(\boldsymbol{x}_t)), \tag{20.16}$$

where $\leftarrow$ denotes the assignment operator and α is an externally-specified step-size parameter[22]. In contrast, temporal-difference methods update the value function estimate at each time step, t, based on the observed reward, r_t, and the estimate of the value function at the next state, $\boldsymbol{x}_{t+1}$, by the update:

[22] In Algorithm 20.3, we used $\alpha = 1/N(\boldsymbol{x}_t)$

$$V(\boldsymbol{x}_t) \leftarrow V(\boldsymbol{x}_t) + \alpha(r_t + \gamma V(\boldsymbol{x}_{t+1}) - V(\boldsymbol{x}_t)). \tag{20.17}$$

This update rule is known as the *TD(0)* update, where the subscript 0 denotes that the update is based on a single step of experience[23]. By comparing the two update rules, we can see that we require a full episode of experience to compute the Monte Carlo target, G_t, while we can compute the temporal-difference target, $r_t + \gamma V(\boldsymbol{x}_{t+1})$, at each time step, t. We provide a complete algorithm for TD(0) in Algorithm 20.4. It is worth noting that we can interpret the quantity

[23] TD(0) is a special case of the more general TD(λ) approach.

Algorithm 20.4: Temporal-Difference Learning (TD(0))

Data: Initial value function, V, arbitrarily initialized for all $\boldsymbol{x} \in \mathcal{X}$.
Result: Value function estimate, V^π.
for *each episode* **do**
  Initialize state $\boldsymbol{x}$
  while $\boldsymbol{x}$ *is not terminal* **do**
    Take action $\boldsymbol{u}$ according to π
    Observe r and x'
    $V(\boldsymbol{x}) \leftarrow V(\boldsymbol{x}) + \alpha(r + \gamma V(\boldsymbol{x}') - V(\boldsymbol{x}))$
    $x \leftarrow x'$
return $V \approx V^\pi$

$r + \gamma V(\boldsymbol{x}') - V(\boldsymbol{x})$ as an error measuring the difference between the current estimate of the value function, $V(\boldsymbol{x})$, and a better[24] estimate, $r + \gamma V(\boldsymbol{x}')$. This quantity, which we refer to as the *TD error*, plays a crucial role in the development of various reinforcement learning algorithms.

[24] This quantity represents a more accurate estimate of the value in state $\boldsymbol{x}$ because it can leverage one step of true reward realization, r, in the transition from $\boldsymbol{x}$ to $\boldsymbol{x}'$.

20.3.3 *Example: Monte Carlo Control*

We have already discussed how we can use the Monte Carlo and temporal-difference learning paradigms to address the problem of learning without a model of the environment. However, we solely discussed these methods in the context of the prediction problem, where we are trying to estimate the value function of a given policy. In this section, we introduce a generic Monte Carlo control method as our first complete example of a reinforcement learning algorithm for learning optimal policies. Later in this chapter, we introduce various reinforcement learning algorithms that, in one way or another, build upon the principles of model-free control that we discuss here.

The overall idea of using Monte Carlo estimation for control is to proceed according to the same principles of Policy Iteration that we introduced in Section 20.2, but where we leverage Monte Carlo prediction methods for the policy evaluation step. We refer to this Monte Carlo-based policy iteration approach as an example of *Generalized* Policy Iteration (GPI), which is the general framework that encompasses all methods that alternate between policy evaluation and policy improvement.

In this method, we learn action values, $Q(\boldsymbol{x}, \boldsymbol{u})$, since, as we discussed in Section 20.3.1, we can use them to directly derive a policy[25]. This GPI method alternates between the following steps:

[25] Unlike state value functions, which require a model to perform the one-step lookahead to derive the policy.

$$\pi_0 \xrightarrow{\text{E}} Q^{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} Q^{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \dots \xrightarrow{\text{I}} \pi^* \xrightarrow{\text{E}} Q^{\pi^*},$$

where $\xrightarrow{\text{E}}$ denotes policy evaluation, and $\xrightarrow{\text{I}}$ denotes policy improvement. In contrast to the Policy Iteration algorithm, we use the Monte Carlo prediction approach from Section 20.3.1 in the policy evaluation step rather than using the exact Bellman equation to update the value function, which would require a model of the environment. The policy improvement step remains the same, where we define the new policy by acting greedily with respect to the current action-value function by choosing $\pi_{i+1}(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}} Q^{\pi_i}(\boldsymbol{x}, \boldsymbol{u})$.

While this approach encompasses various core principles of many reinforcement learning algorithms, it is relatively simplistic. For example, this approach doesn't address the problem of *maintaining exploration*[26]. In the current form, this approach would only work under the *exploring starts* assumption, which ensures that when resetting the episode the agent will start in each state-action pair with non-zero probability. We have to remove this assumption to obtain a practical algorithm. Later in the chapter, we introduce various practical methods to ensure that the agent explores the environment sufficiently.

[26] All state-action pairs must be explored with non-zero probability to ensure that we can correctly estimate the action-value function.

20.3.4 *A Unifying View of Reinforcement Learning*

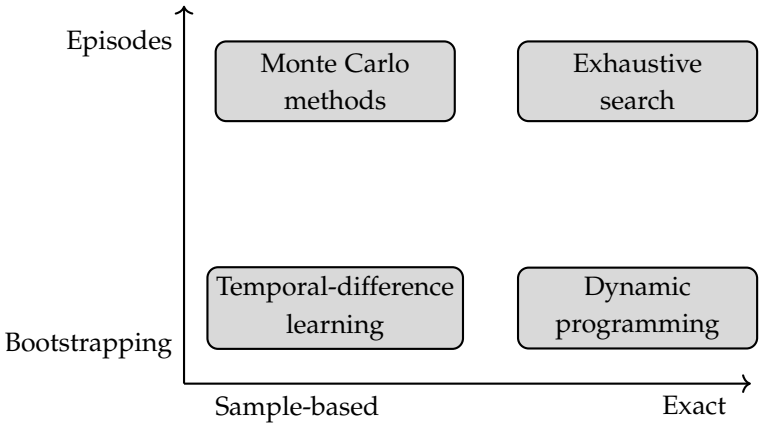


Figure 20.2: We can categorize reinforcement learning methods along two-axes based on whether they are sample-based and whether they bootstrap.

Monte Carlo, temporal-difference, and dynamic programming methods are often presented as distinct approaches to reinforcement learning. However, it is worth noting that these methods are extremes of a spectrum. To appreciate this, we can consider the advantages, disadvantages, and commonalities of each of these paradigms.

Monte Carlo and temporal-difference methods have an advantage over dynamic programming methods in that they do not require a model of the environment, and they can learn directly from sampled experiences with the environment. This capability greatly extends the applicability of these methods to real-world problems, where the environment could be unknown or too complex to be modeled.

An advantage of temporal-difference and dynamic programming methods over Monte Carlo methods is that they do not require waiting until the end of an episode to update value estimates and can learn from incomplete sequences of experience by *bootstrapping*. This can be an important advantage in practice since some applications have very long, or even non-terminating, episodes.

On the other hand, Monte Carlo methods have an advantage over temporal-difference methods in that they are unbiased estimators of the true value function. This is because Monte Carlo methods use the true return, G_t, to update the value function, whereas temporal-difference methods use a biased estimate of the return, $r_{t+1} + \gamma V(\boldsymbol{x}_{t+1})$. Relative to Monte Carlo methods, temporal-difference methods accept some amount of bias for a reduction in variance from updating the value estimates towards targets that depend on fewer steps of stochasticity.

We organize these paradigms based on how they are related in Figure 20.2. At the top right, we have exhaustive search, where we compute quantities of interest, such as value estimates, exactly by model-based simulation of the entire set of possible system evolutions. At the bottom right, we have dynamic programming, where we exploit the principle of optimality, together with a model of the environment to perform a one-step look-ahead and use future values to compute current value estimates by bootstrapping. As we move to the left, we relax the requirement of having a model of the environment or infinite computation, and instead learn from direct experience with the environment by sampling. At the top left, we have Monte Carlo methods, where we avoid exhaustive search by learning from complete samples from the environment. Finally, at the bottom left, we have temporal-difference methods, where we combine the sampling of Monte Carlo methods with the bootstrapping of dynamic programming, thus being able to learn from incomplete sequences of experience.

20.4 A Taxonomy of Reinforcement Learning

Over the last years, the field of reinforcement learning has seen a rapid growth in the number of algorithms and methods, each with its own strengths and

weaknesses. While an exhaustive treatment of all these methods is beyond the scope of this chapter, we aim to provide an overall picture of the different types of algorithms that exist, a deeper understanding of the core principles that underlie these algorithms, and a number of representative examples from each category. In this section, we provide a bird's-eye view of the field of reinforcement learning and classify the different algorithms into a taxonomy, shown graphically in Figure 20.3, that can serve as reference through the rest of the chapter.

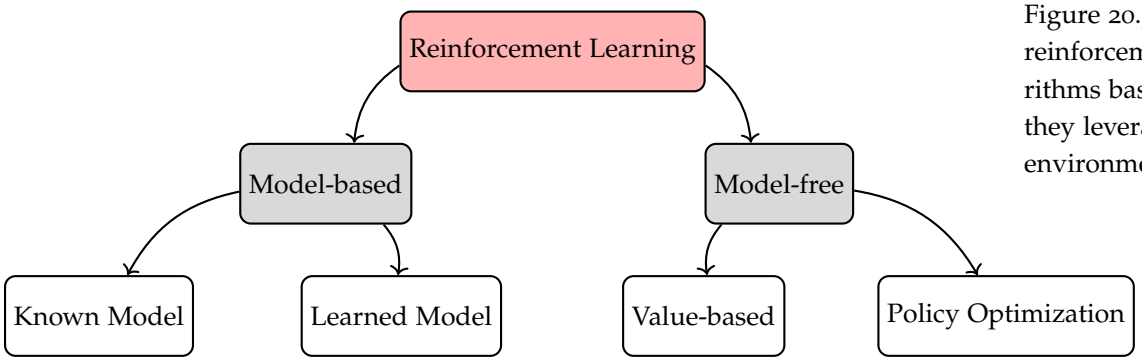


Figure 20.3: A taxonomy of reinforcement learning algorithms based on whether or not they leverage a model of the environment.

The first fundamental distinction among reinforcement learning algorithms is between *model-free* and *model-based* methods. Model-free methods, which are the focus of Section 20.5, attempt to learn the optimal policy directly from experience, without explicitly modeling the environment. Model-based methods, on the other hand, aim to learn a model of the environment and then use this model to derive optimal policies. We cover model-based methods in Section 20.6.

Looking at model-free methods, we can further distinguish between *value-based* and *policy optimization* methods. Value-based methods, similar to the ones described in previous sections, define the policy implicitly through a value function. By doing so, the main focus of value-based methods lies in accurately estimating the optimal value function, which we can then use to derive the optimal policy. Policy optimization methods, on the other hand, represent the policy explicitly via a parametric function and optimize the parameters of this function to maximize the reinforcement learning objective.

On the other side of the graph in Figure 20.3, we can see how we can further divide model-based methods into algorithms that either focus on learning a model of the environment from data, or algorithms that use a known model[27] to derive optimal policies. In this chapter, we focus on methods that learn a model of the environment from data. As we will see in Section 20.6, we can use learned models in various ways, such as within planning routines or for accelerating model-free algorithms.

[27] For example, we could derive the model from physics or other domain knowledge.

20.4.1 On-policy vs Off-policy Learning

Another crucial distinction among reinforcement learning algorithms is between *on-policy* and *off-policy* learning. On-policy methods aim to evaluate or improve the policy that is used to interact with the environment. Off-policy methods, on the other hand, evaluate or improve a policy that is different from the one used to interact with the environment. For example, the Monte Carlo control algorithm we presented above is an on-policy method as it attempts to improve the same policy used to make decisions in the environment. Off-policy methods define two distinct policies: one that must be updated and ideally becomes the optimal policy and one used to interact with the environment. We refer to the policy being learned as the *target policy*, while we refer to the policy used to generate samples from the environment as the *behavior policy*.

Generally, on-policy methods are simpler to implement and are often more stable, while off-policy methods are typically more complex and require additional considerations. Due to this additional complexity, we often consider off-policy methods less stable and characterized by slower convergence. However, off-policy methods are also more powerful and general, encompassing on-policy methods as a special case where the target and behavior policies are the same. We can use off-policy methods to learn from data collected by other policies, such as data collected by a human demonstrator or other conventional non-learning-based policies, to learn about multiple policies simultaneously, or to learn about the optimal policy while still exploring the environment.

Throughout the next sections, we introduce various on-policy and off-policy methods for learning and discuss the advantages and disadvantages of each approach.

20.5 Model-free Reinforcement Learning

Model-free reinforcement learning methods are commonly considered to be the most popular and widely used class of algorithms in the field. This popularity is largely due to their recent successes in a wide range of applications, including playing games[28], robot control[29], and the fine-tuning of large-scale AI chatbots[30]. In this section, we discuss representative reinforcement learning algorithms while maintaining our focus on key principles important for understanding and implementing new algorithms.

[28] D. Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489

[29] S. Levine et al. "End-to-End Training of Deep Visuomotor Policies". In: *Journal of Machine Learning Research* 17.39 (2016), pp. 1–40

[30] Y. Bai et al. "Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback". In: (2022). URL: https://arxiv.org/abs/2204.05862

20.5.1 Value-based Methods

We begin our discussion on model-free reinforcement learning with value-based methods, which, similarly to the Monte Carlo control algorithm presented in Section 20.3.3, estimate the value function of a policy and use this estimate to derive the optimal policy.

Q-learning: One of the first breakthroughs in reinforcement learning was the introduction of the *Q-learning* algorithm[31]. Q-learning is an algorithm that learns the optimal action-value function, $Q^*(\boldsymbol{x}, \boldsymbol{u})$, via temporal-difference learning. Q-learning is an off-policy algorithm, where regardless of the behavior policy used to interact with the environment, the learned action-value function, $Q(\boldsymbol{x}, \boldsymbol{u})$, directly estimates the optimal action value function, $Q^*(\boldsymbol{x}, \boldsymbol{u})$. However, the behavior policy is relevant in characterizing which state-action pairs are visited, and thus, which action values are updated. As long as the behavior policy is sufficiently exploratory[32], Q-learning will converge to the optimal action-value function, $Q^*(\boldsymbol{x}, \boldsymbol{u})$.

[31] C. J. C. H. Watkins and P. Dayan. "Q-learning". In: *Machine Learning* 8.3 (1992), pp. 279–292

[32] In the sense that we ensure all state-action pairs continue to be visited in the limit.

Given a transition from the environment, $(\boldsymbol{x}, \boldsymbol{u}, r, \boldsymbol{x}')$, Q-learning uses the following update rule for the action-value function, $Q(\boldsymbol{x}, \boldsymbol{u})$:

$$Q(\boldsymbol{x}, \boldsymbol{u}) \leftarrow Q(\boldsymbol{x}, \boldsymbol{u}) + \alpha \left(r + \gamma \max_{\boldsymbol{u}'} Q(\boldsymbol{x}', \boldsymbol{u}') - Q(\boldsymbol{x}, \boldsymbol{u}) \right), \tag{20.18}$$

where $\alpha > 0$ is a tunable learning rate parameter. Regardless of the behavior policy used to generate the transition, $(\boldsymbol{x}, \boldsymbol{u}, r, \boldsymbol{x}')$, Equation (20.18) updates the current action-value estimate, $Q(\boldsymbol{x}, \boldsymbol{u})$, towards the value of the greedy policy with respect to the current Q-function given by $r + \gamma \max_{\boldsymbol{u}'} Q(\boldsymbol{x}', \boldsymbol{u}')$. In other words, Q-learning recursively enforces the Bellman optimality equation for the action-value function until convergence[33]. We can interpret Q-learning as a sample-based approximation of Q-Value Iteration[34], where we update the Q-function using a single sample of the Bellman optimality equation.

[33] As we discussed in Section 20.2.4, the fixed-point of the Bellman optimality operator is unique and corresponds to the optimal action-value function, $Q^*(\boldsymbol{x}, \boldsymbol{u})$.

[34] In other words, Algorithm 20.2 with action-value functions.

To ensure that the behavior policy is sufficiently exploratory, Q-learning typically employs an *ϵ-greedy* policy to select actions in the environment. Despite its simplicity, the ϵ-greedy policy approach is a popular and effective exploration strategy where the policy selects the greedy action with respect to the current Q-function with probability $1 - \epsilon$, and selects an action at random with probability ϵ. This simple technique ensures that $\pi(\boldsymbol{u} \mid \boldsymbol{x}) > 0$ for all $\boldsymbol{u}$ and $\boldsymbol{x}$, which is a key requirement for the convergence of Q-learning. We provide a full description of the Q-learning algorithm in Algorithm 20.5.

Value Function Approximation: So far, we have assumed that the state and action spaces, $\mathcal{X}$ and $\mathcal{U}$, are finite and small enough to be easily stored in a look-up table and to allow for meaningful state-action space exploration within a reasonable computation budget. However, in robotics applications, the state and action spaces are often continuous or extremely high-dimensional, making it impractical to compactly store and efficiently update value functions. This challenge motivates us to consider methods that rely on *parametrized function approximation*. Within this class of methods, we represent value functions by a parametric function, $Q_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{u})$, or alternatively $V_{\boldsymbol{\theta}}(\boldsymbol{x})$, with parameters $\boldsymbol{\theta}$. The goal of learning is then to find the optimal parameters, $\boldsymbol{\theta}^*$, such that the value function estimator is close to the optimal value function.

Function approximation has two key advantages. First, it allows us to represent value functions compactly, as the number of parameters, $\boldsymbol{\theta}$, is typically

Algorithm 20.5: Q-Learning

Data: Initial action values, $Q(\boldsymbol{x}, \boldsymbol{u})$, learning rate, α, discount factor, γ, exploration rate, ϵ.
Result: Updated action values, $Q(\boldsymbol{x}, \boldsymbol{u})$.
for *each episode* **do**
  Initialize the state, $\boldsymbol{x}_0$.
  for *each step in the episode* **do**
    Select an action, $\boldsymbol{u}_t$, using an ϵ-greedy policy with respect to $Q(\boldsymbol{x}_t, \boldsymbol{u})$.
    Execute the action $\boldsymbol{u}_t$ and observe the reward, r_t, and the next state, $\boldsymbol{x}_{t+1}$.
    Update the action-value function using the transition $(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1})$.
    $Q(\boldsymbol{x}_t, \boldsymbol{u}_t) \leftarrow Q(\boldsymbol{x}_t, \boldsymbol{u}_t) + \alpha \left(r_t + \gamma \max_{\boldsymbol{u}'} Q(\boldsymbol{x}_{t+1}, \boldsymbol{u}') - Q(\boldsymbol{x}_t, \boldsymbol{u}_t)\right)$
return $Q(\boldsymbol{x}, \boldsymbol{u}) \approx Q^*(\boldsymbol{x}, \boldsymbol{u})$

much smaller than the number of states and actions. Second, it enables the value function estimator to generalize to unseen states and actions, potentially reducing the amount of exploration required to learn a good policy[35]. While there are many choices for the function approximator, including linear functions, neural networks, and decision trees, we focus our discussion on differentiable functions.

[35] In other words, if the function approximator is able to generalize successfully, the agent does not need to visit every state-action pair.

Given a dataset, $\mathcal{D}$, of transitions, $(\boldsymbol{x}, \boldsymbol{u}, r, \boldsymbol{x}')$, policy evaluation via function approximation entails learning the parameters, $\boldsymbol{\theta}$, that minimize the loss function $J(\boldsymbol{\theta})$:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{u},r,\boldsymbol{x}')\sim\mathcal{D}}\left[Q^\pi(\boldsymbol{x}, \boldsymbol{u}) - Q_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{u})\right]. \tag{20.19}$$

Intuitively, $J(\boldsymbol{\theta})$ measures the discrepancy between the estimated value function, $Q_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{u})$, and the target value function, $Q^\pi(\boldsymbol{x}, \boldsymbol{u})$, where $Q^\pi(\boldsymbol{x}, \boldsymbol{u})$ is the value function under the policy π that generated the transitions in $\mathcal{D}$. We can solve this optimization problem with stochastic gradient descent, which uses the update rule:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta_{\boldsymbol{\theta}}, \tag{20.20}$$

where:

$$\Delta_{\boldsymbol{\theta}} = \alpha \left(Q^\pi(\boldsymbol{x}, \boldsymbol{u}) - Q_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{u})\right) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{u}),$$

where $\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{u})$ is the gradient of the value function estimator with respect to its parameters, $\boldsymbol{\theta}$.

In practice, however, the update rule in Equation (20.20) is not directly applicable since the true target value function, $Q^\pi(\boldsymbol{x}, \boldsymbol{u})$, is unknown. Leveraging ideas from model-free control, we can replace the unknown value function with an update target based on Monte Carlo, temporal-difference, or dynamic programming methods. For example, we could use a Monte Carlo target to define

the update:

$$\Delta_{\boldsymbol{\theta}} = \alpha \left(G_t - Q_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{u}_t)\right) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{u}_t), \tag{20.21}$$

where $G_t = \sum_t^{T-1} \gamma^t r_t$, or we could use a temporal-difference target:

$$\Delta_{\boldsymbol{\theta}} = \alpha \left(r_t + \gamma Q_{\boldsymbol{\theta}}(\boldsymbol{x}_{t+1}, \boldsymbol{u}_{t+1}) - Q_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{u}_t)\right) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{u}_t), \tag{20.22}$$

where we would compute both the Monte Carlo and temporal-difference targets from sampled interactions from applying the behavior policy in the environment.

Fitted Q-learning: A particularly popular algorithm that combines function approximation with temporal-difference learning is the *Fitted Q-Learning* algorithm. Fitted Q-Learning updates the parameters, $\boldsymbol{\theta}$, of a Q-function estimator by applying the update rule in Equation (20.20) with:

$$\Delta_{\boldsymbol{\theta}} = \alpha \left(r_t + \gamma \max_{\boldsymbol{u}'} Q_{\boldsymbol{\theta}}(\boldsymbol{x}_{t+1}, \boldsymbol{u}') - Q_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{u}_t)\right) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{u}_t). \tag{20.23}$$

It is important to note that the target $r_t + \gamma \max_{\boldsymbol{u}'} Q_{\boldsymbol{\theta}}(\boldsymbol{x}_{t+1}, \boldsymbol{u}')$ is equivalent to the temporal-difference target used in Q-learning. Essentially, Fitted Q-Learning mimics the update rule of Q-learning, but rather than directly updating the action values to explicitly enforce the Bellman optimality equation, it updates the parameters, $\boldsymbol{\theta}$, of the Q-function estimator to approximately enforce it. In other words, rather than updating the entries of a look-up table representing the value function, Fitted Q-learning updates the parameters, $\boldsymbol{\theta}$, to minimize the error with respect to the fixed point of the Bellman optimality operator.

In this section, we discussed a few foundational examples of value-based reinforcement learning methods. While these are only a subset of the vast literature in value-based methods, they convey the key ideas and challenges of learning value functions for control. Specifically, these methods highlight the central idea of approximating value functions from experience and leveraging them to derive optimal policies. Key challenges include ensuring sufficient exploration, which we can address through strategies like ϵ-greedy policies, and addressing stability and convergence of the learning process when using value function approximators, particularly in high-dimensional or continuous spaces. Value-based methods in reinforcement learning are fundamentally built on a concise set of core principles, such as generalized policy iteration and (approximate) value iteration, and differ primarily in their usage of value update targets[36], function approximators, or behavior policies. Therefore, understanding these foundational concepts provides us a lens through which we can interpret the majority of value-based algorithms.

[36] For instance, Monte Carlo, temporal-difference, or dynamic programming approaches.

20.5.2 *Policy Optimization Methods*

In contrast with value-based methods, policy optimization methods take an alternative, model-free, approach to solving the reinforcement learning problem.

To better motivate policy optimization, we recall the reinforcement learning objective from Section 20.1:

$$V^{\pi} = \mathbb{E}_{\tau \sim p_{\pi}(\tau)} \left[\sum_{t=0}^{T-1} \gamma^t R(\boldsymbol{x}_t, \boldsymbol{u}_t) \right].$$

Rather than learning a value function and deriving a policy from it, as in value-based methods, policy optimization methods define a parameteric policy, $\pi_{\boldsymbol{\theta}}$, and directly optimize the parameters, $\boldsymbol{\theta}$, to maximize the reinforcement learning objective, V^{π}. In other words, the goal of policy optimization methods is to find the optimal policy parameters, $\boldsymbol{\theta}^*$:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \; V(\boldsymbol{\theta}), \tag{20.24}$$

where, for simplicity, we use $V(\boldsymbol{\theta})$ to refer to the reinforcement learning objective, $V^{\pi_{\boldsymbol{\theta}}}$, under policy $\pi_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$. Policy optimization methods solve this optimization problem through a two-step approach. First, they estimate the gradient of the reinforcement learning objective with respect to the policy parameters, $\nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta})$, and then they update the parameters by applying approximate gradient ascent on $V(\boldsymbol{\theta})$:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta}), \tag{20.25}$$

where α denotes the user-defined learning rate.

The first challenge that all policy optimization methods face is estimating the gradient of the reinforcement learning objective. To simplify the notation, we define the cumulative reward as $R(\tau) = \sum_{t=0}^{T-1} \gamma^t r_t$ and assume $\gamma = 1$[37]. By definition of expectation, we have:

$$V(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim p_{\boldsymbol{\theta}}(\tau)} \left[R(\tau) \right] = \int p_{\boldsymbol{\theta}}(\tau) R(\tau) d\tau, \tag{20.26}$$

where $p_{\boldsymbol{\theta}}(\tau)$ denotes the trajectory distribution induced by the policy $\pi_{\boldsymbol{\theta}}$. Equation (20.26) allows us to write the gradient of the reinforcement learning objective as:

$$\nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \int R(\tau) p_{\boldsymbol{\theta}}(\tau) d\tau = \int \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau) R(\tau) d\tau. \tag{20.27}$$

However, we cannot compute this gradient directly because it depends on unknown dynamics through the trajectory distribution, $p_{\boldsymbol{\theta}}(\tau)$[38].

To address this issue, we resort to the following useful identity:

$$p_{\boldsymbol{\theta}}(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) = p_{\boldsymbol{\theta}}(\tau) \frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau)}{p_{\boldsymbol{\theta}}(\tau)} = \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau), \tag{20.28}$$

and use it to rewrite the gradient of the reinforcement learning objective in Equation (20.27) as:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta}) &= \int \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau) R(\tau) d\tau \\ &= \int p_{\boldsymbol{\theta}}(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) R(\tau) d\tau \\ &= \mathbb{E}_{\tau \sim p_{\boldsymbol{\theta}}(\tau)} \left[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) R(\tau) \right]. \end{aligned}$$

[37] The extension to the discounted case is equivalent and relatively straightforward.

[38] Recall the definition of the trajectory distribution is $p_{\boldsymbol{\theta}}(\tau) = p(\boldsymbol{x}_0) \prod_{t=0}^{T} p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t) \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t)$.

We can then approximate the expectation using Monte Carlo methods by sampling from the trajectory distribution, $p_{\boldsymbol{\theta}}(\tau)$, through interaction with the environment. However, directly computing the gradient of the log-probability, $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)$, remains intractable. To address this, we recall the definition of the trajectory distribution, $p_{\boldsymbol{\theta}}(\tau)$:

$$p_{\boldsymbol{\theta}}(\tau) \coloneqq p(\boldsymbol{x}_0) \prod_{t=0}^{T-1} p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t) \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t), \tag{20.29}$$

and therefore the log-probability is:

$$\log p_{\boldsymbol{\theta}}(\tau) = \log p(\boldsymbol{x}_0) + \sum_{t=0}^{T-1} \log p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t) + \log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t). \tag{20.30}$$

By substituting Equation (20.30) into the gradient of the reinforcement learning objective, we obtain:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta}) &= \mathbb{E}_{\tau \sim p_{\boldsymbol{\theta}}(\tau)} \left[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) R(\tau) \right] \\ &= \mathbb{E}_{\tau \sim p_{\boldsymbol{\theta}}(\tau)} \left[\nabla_{\boldsymbol{\theta}} \left(\log p(\boldsymbol{x}_0) + \sum_{t=0}^{T-1} \log p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t) + \log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t) \right) R(\tau) \right].\end{aligned}$$

Notably, the terms $\log p(\boldsymbol{x}_0)$ and $\log p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t)$ do not depend on $\boldsymbol{\theta}$, and thus we can ignore them when computing the gradient of the reinforcement learning objective. Moreover, evaluating the gradient of the log-probability of the action, $\log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t)$, is tractable and we can easily compute it, for example by using automatic differentiation tools.

This leads to the following expression for the gradient of the reinforcement learning objective:

$$\nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim p_{\boldsymbol{\theta}}(\tau)} \left[\sum_{t=0}^{T-1} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t) R(\tau) \right], \tag{20.31}$$

which is tractable to compute and which we can estimate using samples from the environment. For example, given N episodes of interaction with the environment, we can estimate the gradient of the reinforcement learning objective as:

$$\nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t^i \mid \boldsymbol{x}_t^i) R(\tau^i). \tag{20.32}$$

This is a crucial result and lays the foundations for almost all policy optimization algorithms in reinforcement learning.

REINFORCE: The derivations above directly lead to one of earliest examples of policy optimization methods[39] known as the *REINFORCE algorithm*[40]. At a high level, the REINFORCE algorithm estimates the gradient of the reinforcement learning objective in Equation (20.32) using samples from the environment, and after each episode updates the policy parameters, $\boldsymbol{\theta}$, in the direction of the gradient. We outline the REINFORCE algorithm in Algorithm 20.6.

[39] In the reinforcement learning literature, these are often also referred to as *policy gradient methods*.

[40] R. J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8.3 (1992), pp. 229–256

Algorithm 20.6: REINFORCE Algorithm

Data: Initial policy parameters, $\boldsymbol{\theta}$, learning rate, α
Result: Update policy parameters, $\boldsymbol{\theta}$
for *each episode* **do**
  Generate a trajectory, $\tau = \{\boldsymbol{x}_0, \boldsymbol{u}_0, r_0, \ldots, \boldsymbol{x}_T\}$, using the policy $\pi_{\boldsymbol{\theta}}$ in the environment.
  Compute the return, $R(\tau) = \sum_{t=0}^{T-1} r_t$.
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta})$, where $\nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta})$ is computed by Equation (20.32).
return $\boldsymbol{\theta}$

From Equation (20.31), we can see that the gradient is computed as the sum of the gradients of the log-probabilities of the actions, weighted by the return of the trajectory. The intuitive understanding of this expression is that by taking a step in the direction of the policy gradient, we update the policy parameters to increase the log-probability of the actions that lead to high returns, and decrease the log-probability of the actions that lead to low returns. In other words, the policy gradient formalizes the idea of trial and error learning, making good behavior more likely and bad behavior less likely.

Policy optimization methods represent a popular and intuitive approach to reinforcement learning and have several advantages and disadvantages compared to value-based methods. A first key advantage of policy optimization methods is that they can naturally handle both discrete and continuous action spaces because we can compute the gradient of the policy using automatic differentiation tools[41]. For example, in the case of continuous action spaces, we can parameterize the policy as a Gaussian distribution. Additionally, policy optimization methods have the notable advantage of directly optimizing the reinforcement learning objective. This ensures that improvements to the policy are measured against a well-defined metric, since better values of the reinforcement learning objective imply a better policy. In contrast, value-based methods rely on fixed-point iterations of value functions to satisfy the Bellman equation with the goal to eventually converging to the optimal value function. While this approach is theoretically sound, it is unclear how suboptimal the policy is during intermediate iterations.

[41] Assuming the policy is parameterized by a differentiable function.

Policy gradient methods also have some disadvantages. First, the policy optimization methods we have presented so far are inherently *on-policy* methods, which can result in high sample inefficiency[42]. A second disadvantage is that the gradient defined in Equation (20.32) is a high-variance estimator of the true gradient from Equation (20.31). In practice, this can lead to extremely noisy updates and therefore slow convergence.

As we will see in the remainder of this section, a lot of research in the domain of policy optimization has focused on addressing these limitations to develop sample-efficient and lower-variance policy optimization methods.

[42] A number of off-policy policy optimization algorithms have been introduced to allow the policy to be updated using experiences collected from different policies. These methods aim to approximate the behavior of classical on-policy algorithms while improving sample efficiency. Despite this advantage, they often introduce additional complexities, such as the need for more sophisticated exploration strategies and managing the stability of the off-policy updates.

Actor-Critic Methods: Actor-critic methods represent an important extension of policy optimization that reduces the high variance associated with policy gradient estimates. Recall that we can express the policy gradient, with a slight rearrangement of the summation terms, as:

$$\nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t^i \mid \boldsymbol{x}_t^i) \left(\sum_{t'=t}^{T-1} r_{t'}^i \right).$$

We refer to the term $\sum_{t'=t}^{T-1} r_{t'}^i$ as the *reward-to-go* The reward-to-go is a one-sample estimate of the true return, which is defined as the expected cumulative reward under the trajectory distribution, $\mathbb{E}_{\tau \sim p_{\boldsymbol{\theta}}(\tau)} \left[\sum_{t'=t}^{T-1} r_{t'} \right]$. While conceptually straightforward, this reward-to-go estimate introduces significant variance, leading to noisy policy gradient updates.

Actor-critic methods address this challenge by introducing a *critic*, which is a parametric approximation of the value function. Since the value function estimates the expected reward-to-go, the critic enables us to replace the high-variance sample-based estimate with a lower-variance, learned approximation. Concretely, the policy gradient becomes:

$$\nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t^i \mid \boldsymbol{x}_t^i) Q_{\boldsymbol{\phi}}(\boldsymbol{x}_t^i, \boldsymbol{u}_t^i), \tag{20.33}$$

where $Q_{\boldsymbol{\phi}}(\boldsymbol{x}_t, \boldsymbol{u}_t)$ is the critic's estimate of the action-value function. We can update the critic using any value estimation method, such as the value-based methods with function approximation discussed in Section 20.5.1.

A particularly popular choice for the definition of the policy gradient in actor-critic methods is through the *advantage function*:

$$A^{\pi}(\boldsymbol{x}_t, \boldsymbol{u}_t) = Q^{\pi}(\boldsymbol{x}_t, \boldsymbol{u}_t) - V^{\pi}(\boldsymbol{x}_t), \tag{20.34}$$

which quantifies the relative merit of taking action $\boldsymbol{u}_t$ in state $\boldsymbol{x}_t$, compared to the average value of the state. Intuitively, the advantage function emphasizes actions that are better than average while downplaying less promising ones. In practice, we often approximate the advantage function to avoid estimating both Q^{π} and V^{π}, using:

$$A^{\pi}(\boldsymbol{x}_t, \boldsymbol{u}_t) \approx r_t + \gamma V^{\pi}(\boldsymbol{x}_{t+1}) - V^{\pi}(\boldsymbol{x}_t). \tag{20.35}$$

Incorporating the advantage function into the computation of the policy gradient results in the *Advantage Actor-Critic (A2C)*[43] algorithm, which significantly reduces variance and improves learning stability. Unlike REINFORCE, which adjusts the policy solely based on raw returns, A2C leverages the advantage function to normalize updates, focusing on actions that perform better than expected.

[43] V. Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *Proceedings of The 33rd International Conference on Machine Learning*. 2016, pp. 1928–1937

In practice, actor-critic methods involve iterative updates of both the policy parameters, $\boldsymbol{\theta}$, and the value function parameters, $\boldsymbol{\phi}$. At each step, we improve the policy based on the estimated advantage and we refine the value function to

Algorithm 20.7: Advantage Actor Critic (A2C)

Data: Initial policy parameters, $\boldsymbol{\theta}$, and value function parameters, ϕ, learning rates, $\alpha_{\boldsymbol{\theta}}$, α_{ϕ}, discount factor, γ
Result: Updated policy parameters, $\boldsymbol{\theta}$
for *each episode* **do**
  Sample trajectories, $\tau = \{\boldsymbol{x}_0, \boldsymbol{u}_0, r_0, \ldots, \boldsymbol{x}_T\}$, using $\pi_{\boldsymbol{\theta}}$.
  for $t = 0$ ***to*** $T - 1$ **do**
    Compute the value target, for example using temporal-difference: $y_t = r_t + \gamma V_{\phi}^{\pi}(\boldsymbol{x}_{t+1})$.
    $\phi \leftarrow \phi + \alpha_{\phi} \nabla_{\phi} (y_t - V_{\phi}^{\pi}(\boldsymbol{x}_t))^2$
    $A_{\phi}^{\pi}(\boldsymbol{x}_t, \boldsymbol{u}_t) = y_t - V_{\phi}^{\pi}(\boldsymbol{x}_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t) A_{\phi}^{\pi}(\boldsymbol{x}_t, \boldsymbol{u}_t)$
return $\boldsymbol{\theta}$

better approximate future rewards. The pseudocode for A2C in Algorithm 20.7 highlights these alternating updates.

Actor-critic methods blend the strengths of policy optimization and value-based approaches, achieving a balance between expressive policy representations and efficient variance reduction. However, the introduction of a critic also adds computational complexity and tuning challenges that we must consider in practical implementations.

20.5.3 *Limitations of Model-free Reinforcement Learning*

Despite their successes, model-free reinforcement learning methods suffer from several challenges. The first challenge is *sample efficiency*. Model-free methods typically require a large number of samples before being able to learn a good policy, which can be prohibitively expensive in many real-world robotics applications. Next, in their standard form, model-free methods are inherently designed to be *single-task learners*. Given the specification of a reward function for a given task, model-free methods learn a policy that maximizes the expected return solely for that task. This makes it difficult to transfer knowledge across tasks. Finally, in many real-world applications, the reward signal can be sparse, delayed, or noisy, making it difficult for model-free methods to learn a good policy.

20.6 *Model-based Reinforcement Learning*

Model-based reinforcement learning methods aim to address the limitations of model-free methods by learning a model of the environment. In this section, we introduce two broad classes of model-based reinforcement learning methods: *model-based planning* methods that learn a model and use it to plan and *model-based policy optimization* methods that learn a model and use it to accelerate

model-free policy learning.

20.6.1 Model-based Planning

If we had access to a model of the dynamics, $p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t)$, we could leverage tools from model-based optimal control to compute an optimal action plan. Based on this observation, the main idea behind model-based planning methods is to learn an approximate model of the true dynamics model from data, and then use this learned model to plan. For example, we could apply the following high-level strategy:

1. Run a base policy, π_0, in the environment and collect a dataset of transitions, $\mathcal{D} = \{(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1})\}$.
2. Fit a dynamics model, $p_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{u}_t)$, to the observed data to minimize the prediction error, for example by minimizing the mean squared error between the predicted and true next state: $\min_{\boldsymbol{\theta}} \sum_{(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1}) \in \mathcal{D}} \left\| p_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{u}_t) - \boldsymbol{x}_{t+1} \right\|^2$.
3. Use the learned dynamics model to plan a sequence of actions for the agent to execute.

Despite its simplicity, this scheme works for relatively well-behaved systems, where the dataset, $\mathcal{D}$, guarantees sufficient coverage of the state-action space, and where the learned model, $p_{\boldsymbol{\theta}}$, is accurate enough to enable effective planning[44]. However, in practice, learning an accurate model of the dynamics is often challenging, especially when dealing with high-dimensional, non-linear, and stochastic systems. Additionally, any inaccuracies in the learned model can be exploited by the optimization-based process.

[44] This scheme is essentially equivalent to a task known as *system identification*.

A popular approach to address this issue is to consider a measure of uncertainty in the model's predictions, and to use this uncertainty to inform the planning process. While there are many ways to quantify uncertainty, we consider methods that aim to learn a *posterior distribution* over the model parameters. In these methods, rather than learning a single estimate of the model parameters, $\boldsymbol{\theta}$, through standard maximum likelihood estimation:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\mathcal{D} \mid \boldsymbol{\theta}), \tag{20.36}$$

where $p_{\boldsymbol{\theta}}(\mathcal{D} \mid \boldsymbol{\theta})$ is the likelihood of the data given the model parameters, we instead aim to learn a posterior distribution, $p(\boldsymbol{\theta} \mid \mathcal{D})$, over the model parameters. In other words, we aim to learn a full distribution over the model parameters that is consistent with the observed data, potentially capturing multiple plausible models that explain the data, and ultimately enabling us to reason about the uncertainty in the model's predictions. We can achieve this by applying Bayes' rule to compute the posterior distribution[45]:

[45] K. P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022

$$p(\boldsymbol{\theta} \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(\mathcal{D})}, \tag{20.37}$$

where $p(\mathcal{D} \mid \boldsymbol{\theta})$ is the likelihood of the data given the model parameters, $p(\boldsymbol{\theta})$ is the prior distribution over the model parameters, and $p(\mathcal{D})$ is the marginal likelihood of the data.

Once we have an estimate of the posterior distribution over the model parameters, we can apply it in the following model-based planning scheme:

1. Run a base policy, π_0, in the environment and collect a dataset of transitions, $\mathcal{D} = \{(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1})\}$.
2. Use $\mathcal{D}$ to estimate a posterior distribution, $p(\boldsymbol{\theta} \mid \mathcal{D})$, over the model parameters.
3. Sample a set of K plausible models, $\{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_K\} \sim p(\boldsymbol{\theta} \mid \mathcal{D})$.
4. For each sampled model, $\boldsymbol{\theta}_k$, and given a candidate action plan, $(\boldsymbol{u}_1, \ldots, \boldsymbol{u}_{T-1})$, use the model to compute the expected return:

$$V(\boldsymbol{u}_1, \ldots, \boldsymbol{u}_{T-1}) = \frac{1}{K} \sum_{k=1}^{K} \sum_{t=1}^{T-1} R(\boldsymbol{x}_t, \boldsymbol{u}_t),$$

   where $\boldsymbol{x}_{t+1} \sim p_{\boldsymbol{\theta}_k}(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t)$.
5. Execute the first action from the best plan according to the expected return.

This scheme allows us to leverage the uncertainty in the model's predictions by considering multiple plausible models and to reason about the expected return under each model rather than optimizing under a single model. Despite its effectiveness, it is important to note that this scheme is just a high-level description of the model-based planning process, and there are many practical considerations that we would need to address to make this approach work in practice.

20.6.2 *Model-based Policy Optimization*

The second class of model-based reinforcement learning methods we consider is *model-based policy optimization*. In contrast to model-based planning, which uses the learned model to plan a sequence of actions, model-based policy optimization uses the learned model to improve model-free policy learning.

Specifically, having a learned model allows us to consider two sources of experience: real-world data collected by executing the policy in the environment and synthetic data generated by the model. Given an MDP, $\mathcal{M} = (\mathcal{S}, \mathcal{A}, p, r, \gamma)$, and a learned model, $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t+1}, r_t \mid \boldsymbol{x}_t, \boldsymbol{u}_t)$[46], we can consider two sources of experience:

$$(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1}) \sim \text{Environment data},$$
$$(\boldsymbol{x}_t, \boldsymbol{u}_t, \hat{r}_t, \hat{\boldsymbol{x}}_{t+1}) \sim p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t+1}, r_t \mid \boldsymbol{x}_t, \boldsymbol{u}_t).$$

The basic idea of model-based policy optimization is to use both sources of experience to improve model-free policy learning. One of the earliest and most popular methods in this category is the *Dyna-Q algorithm*[47].

[46] Here we consider the general case where we learn both the next state and the reward, but we can extend the discussion to the case where we only have to learn one of the two.

[47] R. S. Sutton. "Dyna, an integrated architecture for learning, planning, and reacting". In: *SIGART Bull.* 2.4 (1991), pp. 160–163

Dyna-Q: The Dyna-Q algorithm is a model-based reinforcement learning algorithm that improves the learning efficiency of Q-learning by using the learned model to generate synthetic data[48]. The algorithm is based on the idea that in addition to updating the Q-function using real-world data, we can also update the Q-function using synthetic data generated by the learned model. At a high-level, the algorithm alternates between three main steps. First, we perform the "standard" Q-learning steps, where we update the Q-function using real-world data. Next is a model learning step where we update the model using real-world data. Finally, we have a model-based acceleration step where we update the Q-function using synthetic data generated by the model. We provide a description of this algorithm in Algorithm 20.8.

[48] The term Dyna-Q derives from the fact that the algorithm combines Q-learning with model-based acceleration. The term *Dyna* more generally refers to the idea of using a learned model to generate synthetic data for model-free learning.

Algorithm 20.8: Dyna-Q

Data: Model, $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t+1}, r_t \mid \boldsymbol{x}_t, \boldsymbol{u}_t)$, number of model-based acceleration steps, n
Result: Updated model parameters, $\boldsymbol{\theta}$, action value function $Q(\boldsymbol{x}, \boldsymbol{u}) \approx Q^*(\boldsymbol{x}, \boldsymbol{u})$
for *each episode* **do**
  Initialize $\boldsymbol{x}_0$.
  for *each step* t **do**
    Select action $\boldsymbol{u}_t = \pi(\boldsymbol{x}_t)$.
    Observe reward r_t and next state $\boldsymbol{x}_{t+1}$.
    $Q(\boldsymbol{x}_t, \boldsymbol{u}_t) \leftarrow Q(\boldsymbol{x}_t, \boldsymbol{u}_t) + \alpha \left(r_t + \gamma \max_{\boldsymbol{u}'} Q(\boldsymbol{x}_{t+1}, \boldsymbol{u}') - Q(\boldsymbol{x}_t, \boldsymbol{u}_t)\right)$
    Update model parameters, $\boldsymbol{\theta}$, using sample $(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1})$.
    for $i = 1, \ldots, n$ **do**
      Sample $\boldsymbol{x}_t, \boldsymbol{u}_t$ from real-world data.
      Generate synthetic data: $(\hat{\boldsymbol{x}}_{t+1}, \hat{r}_t) \sim p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t+1}, r_t \mid \boldsymbol{x}_t, \boldsymbol{u}_t)$.
      $Q(\boldsymbol{x}_t, \boldsymbol{u}_t) \leftarrow Q(\boldsymbol{x}_t, \boldsymbol{u}_t) + \alpha \left(\hat{r}_t + \gamma \max_{\boldsymbol{u}'} Q(\hat{\boldsymbol{x}}_{t+1}, \boldsymbol{u}') - Q(\boldsymbol{x}_t, \boldsymbol{u}_t)\right)$
  return $\boldsymbol{\theta}$, $Q(\boldsymbol{x}, \boldsymbol{u}) \approx Q^*(\boldsymbol{x}, \boldsymbol{u})$

The Dyna-Q algorithm is a simple yet powerful method that demonstrates the potential of model-based reinforcement learning to improve the learning efficiency of model-free algorithms.

20.6.3 Limitations of Model-based Reinforcement Learning

Model-based methods are an extremely active and promising area of research in reinforcement learning, but they are also subject to several limitations. First, model learning entails optimizing the parameters of the model to minimize prediction error. However, this objective does not necessarily align with the objective of the agent, which is to maximize the expected cumulative reward, and this discrepancy can lead to suboptimal policies. Second, model-based methods are sensitive to model errors, which can cause the agent to learn suboptimal policies or exploit the model errors to achieve high rewards, poten-

tially leading to catastrophic failures. Finally, learning an accurate model of the environment is a challenging task, especially in complex environments with high-dimensional state and action spaces.

20.7 The Promise of Learning from Interaction

In this chapter, we provided a comprehensive overview of the field of reinforcement learning. Rather than presenting an exhaustive list of algorithms, we focused on a conceptual understanding of the key ideas and principles that underlie reinforcement learning (Section 20.3). In particular, we discussed how Monte Carlo methods (Section 20.3.1) and temporal-difference learning (Section 20.3.2) represent two foundational paradigms in reinforcement learning, and how we can use these methods to estimate value functions and learn optimal policies. Most importantly, we highlighted how these methods, together with dynamic programming, define a full spectrum of possible approaches to the problem of learning from interaction (Section 20.3.4). Finally, we also discussed concrete examples of the main algorithmic families in reinforcement learning, including model-free (Section 20.5) and model-based (Section 20.6) methods, and highlighted the key ideas behind some of the most popular algorithms within these categories.

References

[4] Y. Bai et al. "Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback". In: (2022). URL: https://arxiv.org/abs/2204.05862.

[6] D. Bertsekas. *Reinforcement learning and optimal control*. Athena Scientific, 2019.

[39] S. Levine et al. "End-to-End Training of Deep Visuomotor Policies". In: *Journal of Machine Learning Research* 17.39 (2016), pp. 1–40.

[45] V. Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *Proceedings of The 33rd International Conference on Machine Learning*. 2016, pp. 1928–1937.

[48] K. P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022.

[65] D. Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[69] R. Sutton and A. Barto. *Reinforcement learning: An introduction*. MIT Press, 2018.

[70] R. S. Sutton. "Dyna, an integrated architecture for learning, planning, and reacting". In: *SIGART Bull.* 2.4 (1991), pp. 160–163.

[75] C. J. C. H. Watkins and P. Dayan. "Q-learning". In: *Machine Learning* 8.3 (1992), pp. 279–292.

[76] R. J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8.3 (1992), pp. 229–256.

21
Imitation Learning

In Chapter 18, we introduced a strategy for autonomous robot decision making that requires a very manual and sometimes intractable process of specifying desired actions from every possible state. Then, in Chapter 19 and Chapter 20, we formulated the sequential decision making problem as an optimization problem where we must specify a cost or reward function that we want the robot to minimize or maximize. This optimization-based approach is more general and scalable, but it still requires us to figure out how to appropriately embed our preferences into the form of a mathematical function. Reward design can be very challenging in practice, and by the nature of optimization-based approaches, the cost or reward function can be inadvertently exploited in undesirable ways. Additionally, in the reinforcement learning context, we require continuous and exploratory interactions with the environment that could be costly or unsafe[1] as well as sophisticated learning algorithms that are able to learn from experience.

In practice, it can sometimes be easier, more efficient, or safer for human experts to *demonstrate* the desired task or behavior than it is to precisely program it, try to encode it in a cost function, or let the robot freely interact with the environment. The goal of *imitation learning* in the context of robotics is to leverage a limited set of expert demonstrations to accelerate or completely train a robot to autonomously perform a desired behavior. In this chapter, we begin in Section 21.1 by introducing the concept of imitation learning in the context of robotics, provide a formal problem formulation, and discuss key design considerations. We then present a canonical imitation learning approach known as *Behavioral Cloning* in Section 21.2, which aims to directly learn a policy from expert demonstrations. Lastly, in Section 21.3, we introduce *Inverse Reinforcement Learning (IRL)*, an alternative approach to imitation learning that learns a reward function from expert demonstrations.

[1] For example, some robots operate in close collaboration with humans or in other safety-critical environments where the risk of exploring sub-optimal actions is significant.

21.1 Imitation Learning in Robotics

Imitation learning is a class of methods that enable skills to be transferred from an expert to a learner. In the context of robotics, the expert is typically a human operator or a pre-existing control policy, and the learner is the robot that aims

to mimic the expert's behavior. While the literature on imitation learning is vast, in this section, we focus on core design decisions and concepts essential for understanding and applying imitation learning to robotic systems.

We should consider several key aspects when designing an imitation learning system. First, we should determine if imitation learning is the best approach for the problem, since it might not always be the most suitable method for learning a task. For example, reinforcement learning might be a more effective approach if it is inexpensive for us to obtain samples from the environment[2]. Imitation learning also may not yield the desired performance if the expert's behavior is suboptimal or inconsistent. Second, we should be able to identify the features of the demonstrations that we want the robot to actually learn to imitate. In many cases, sensor or control input data recorded from the expert demonstration may contain a large amount of information that is irrelevant for learning the task. Third, we should determine how to select the expert, since the choice of expert can significantly impact the quality of the learned behavior. In many cases, the expert is a human operator who demonstrates the task. However, the expert could also be a pre-existing control policy, a set of historical data, or a mixture of multiple experts. Fourth, we should carefully select how to represent the policy, since this can greatly influence the learning process. For example, we can equivalently represent expert behavior at different levels of abstraction, such as low-level motor commands, mid-level state trajectories, or high-level symbolic actions. Additionally, different functional forms of the policy[3] can impact the expressiveness and generalization capabilities of the learned policy. Finally, the choice of the learning algorithm can significantly impact the efficiency and performance of the imitation learning process. Many algorithms have been proposed for imitation learning, each with strengths and weaknesses. Understanding the characteristics of different algorithms and their suitability for the given task is essential for designing an effective imitation learning system.

[2] Such as if we have a good *simulator* for the task because it could be very safe and cheap to collect data.

[3] For example, whether the policy is defined as a linear function or a neural network.

21.1.1 *Differences Among Imitation Learning, Supervised Learning, and Reinforcement Learning*

Imitation learning is often compared with supervised learning and reinforcement learning, as all three paradigms involve learning from data. While these methods share similarities, they also differ in several key aspects. Supervised learning methods aim to learn a mapping from input data, such as camera images, to output labels, such as object categories, based on a dataset of input-output pairs. While the imitation learning task of deriving a policy from a dataset of expert demonstrations is closely related to supervised learning, there are several key differences. First, in imitation learning, the solution may have inherent structural properties, such as physical constraints or temporal dependencies[4], that are not present in standard supervised learning tasks. Second, in a traditional supervised learning setting, we assume that the source domain,

[4] For example, in robot planning and control we often have actuation limits and the structure from the temporally sequential nature of the decision making task.

which includes the dataset used for training, and the target domain, which includes the test data, are the same. In imitation learning, we may not be able to directly transfer the expert's behavior to the learner's environment. For example, the embodiment of the expert may differ from the learner, such as if the expert is a human and the learner is a robot, leading to expert demonstrations of actions that are not directly executable by the robot. Imitation learning is also typically exposed to the *covariate shift* problem, where the distribution of the expert's data may differ from the distribution of the learner's data. Specifically, the learner may encounter situations not represented in the expert's demonstrations, requiring it to generalize beyond the expert's behavior. Lastly, obtaining expert demonstrations can be costly or time-consuming, making data collection a significant concern.

Imitation learning is also closely related to reinforcement learning, as both paradigms involve learning a policy from data that maximizes a reward. However, reinforcement learning methods typically require a predefined reward function to guide the robot's behavior. In contrast, imitation learning assumes that the expert directly provides optimal, or at least good, behavior, bypassing the need for a reward function.

21.1.2 Problem Formulation

In imitation learning problems, we typically assume that we have access to a dataset, $\mathcal{D}$, of expert demonstrations. The dataset generally consists of a set of trajectories and contexts, and we denote it mathematically as $\mathcal{D} = \{(\tau_i, s_i)\}_{i=1}^N$ where N is the number of samples and $\tau_i = \{\boldsymbol{x}_{i,0}, \boldsymbol{u}_{i,0}, \ldots, \boldsymbol{x}_{i,T}\}$ is a trajectory executed by the expert in a given context, s_i. The context, s_i, may represent a task description, an environmental configuration, or any other relevant information characterizing the expert's behavior. Alternatively, the dataset may consist of state-action pairs, where we would write $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{u}_i)\}_{i=1}^N$.

Given this dataset, we can reproduce the expert's behavior by learning a mapping from contexts to trajectories or from states to actions:

$$\pi(s) = \tau \quad \text{or} \quad \pi(\boldsymbol{x}) = \boldsymbol{u}.$$

We can learn these policies through supervised learning techniques using an approach known as *Behavioral Cloning*[5].

Alternatively, we can use the expert demonstrations to learn a reward function, $R(\boldsymbol{x}, \boldsymbol{u})$, that implicitly defines the expert's behavior, and then infer a policy that maximizes this reward:

$$\pi^*(\boldsymbol{x}) = \arg\max_{\pi} V^{\pi} = \arg\max_{\hat{\pi}} \mathbb{E}_{\tau \sim p(\tau)} \left[\sum_{t=0}^{T-1} \gamma^t R(\boldsymbol{x}_t, \boldsymbol{u}_t) \right],$$

where V^{π} is the expected sum of future rewards for policy π where the expectation is over possible trajectories, τ, that are distributed according to $p(\tau)$. This approach is known as *Inverse Reinforcement Learning* (IRL)[6] or *Inverse Optimal*

[5] T. Osa et al. "An Algorithmic Perspective on Imitation Learning". In: (2018). URL: https://arxiv.org/abs/1811.06711

[6] S. Arora and P. Doshi. "A survey of inverse reinforcement learning: Challenges, methods and progress". In: *Artificial Intelligence* 297 (2021), p. 103500

Control (IOC).

Behavioral Cloning and IRL are the two primary approaches to imitation learning, and each has its own distinct strengths and limitations. In the following sections, we discuss these approaches in more detail and provide insights into when each method is most appropriate.

21.2 Behavioral Cloning

Behavioral Cloning is an approach to imitation learning that learns a mapping[7] from states to actions or contexts to trajectories that mimics the expert's behavior. We formulate the Behavioral Cloning task as a supervised learning problem, where we learn the policy, π, by solving a regression problem. We outline the general procedure for Behavioral Cloning in Algorithm 21.1. The first step entails collecting a dataset, $\mathcal{D}$, of expert demonstrations, for example from logged data from a human operator. Next, we define a model architecture for the policy, $\pi_{\boldsymbol{\theta}}$, which can be a neural network, a linear model, or some other function parameterized by $\boldsymbol{\theta}$. Our choice of model architecture depends on the task's complexity and the available data. For example, we should choose a model that is expressive enough to capture the expert's behavior but not so complex that it can overfit to the data. In the next step, we define a loss function, $\mathcal{L}$, that we optimize to determine the policy parameters, $\boldsymbol{\theta}$. The loss function should represent a well-defined measure of discrepancy between the predicted actions and the expert's actions. Common choices for the loss function include mean squared error, L-1 loss, Hinge loss, and Kullback-Leibler divergence. Finally, we optimize the model parameters, $\boldsymbol{\theta}$, to minimize the loss function, $\mathcal{L}$.

[7] Behavioral Cloning learns a policy directly, rather than trying to model a reward function, like IRL methods.

Algorithm 21.1: Behavioral Cloning

Collect a dataset, $\mathcal{D}$, of expert demonstrations.
Define a model architecture for the policy, $\pi_{\boldsymbol{\theta}}$.
Define a loss function, $\mathcal{L}$.
Optimize the loss function, $\mathcal{L}$, with respect to the model parameters, $\boldsymbol{\theta}$.
return Trained policy, $\pi_{\boldsymbol{\theta}}$.

Behavioral cloning methods are an attractive approach to learning-based decision making, primarily due to their simplicity, effectiveness, and broad applicability. However, ensuring the learned policy performs reliably in real-world settings presents significant challenges. One of the primary obstacles to trustworthy deployment of policies learned through Behavioral Cloning is the issue of *covariate shift*.

21.2.1 The Covariate Shift Problem

Covariate shift occurs when the distribution of the training data differs from the distribution observed during deployment. In the context of Behavioral Cloning,

this problem arises when we allow the learned policy to interact with its environment in a closed-loop fashion. Although the robot's policy is trained to minimize the discrepancy between its own actions and the expert's actions, small errors are inevitable and can accumulate over time. As a result, the robot may encounter regions of the state space not observed in the expert's demonstrations that we used to learn the robot's policy. This distribution shift in the observed states can lead to increasingly large errors, compounding over time and potentially resulting in task failure.

While it is impractical to gather data covering all possible states a robot might encounter, several strategies have been developed to mitigate the impact of covariate shift. These strategies typically follow an iterative process that alternates between updating the robot's policy and targeted data collection based on the robot's current state distribution. In this section, we outline two primary approaches to address covariate shift: *confidence-based methods* and *data aggregation methods*.

Confidence-Based Methods: In the class of confidence-based methods[8], we equip the robot with a mechanism to estimate the uncertainty in its predicted actions. This uncertainty estimate gives the robot the ability to identify situations where its policy is likely to make errors, allowing for it to take corrective measures, such as collecting more data for additional training. In some cases, data collection is also prompted by expert intervention, where the expert steps in to correct the robot's actions. At a high-level, methods based on this iterative process aim to mitigate the effects of covariate shift by empirically aligning the policy with the true state distribution. We provide a general framework of this approach in Algorithm 21.2, but the specific implementation details may vary depending on the task and the available data.

[8] S. Chernova and M. Veloso. "Interactive policy learning through confidence-based autonomy". In: *Journal of Artificial Intelligence Research* 34.1 (2009), pp. 1–25. ISSN: 1076-9757

Algorithm 21.2: Confidence-based Methods

Data: Dataset of expert demonstrations, $\mathcal{D}$, confidence estimatation function, $c(\boldsymbol{x})$, confidence threshold, c_0
Result: Trained policy, $\pi_{\boldsymbol{\theta}}$
Train a policy, $\pi_{\boldsymbol{\theta}}$, on the dataset, $\mathcal{D}$.
while *true* **do**
  Observe the state, $\boldsymbol{x}_t$.
  Compute the confidence estimate, $c(\boldsymbol{x}_t)$.
  if $c(\boldsymbol{x}_t) < c_0$ ***or*** *expert intervention is necessary* **then**
    Compute additional demonstration data, $(\boldsymbol{x}_t, \boldsymbol{u}_t^{\text{expert}})$.
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\boldsymbol{x}_t, \boldsymbol{u}_t^{\text{expert}})\}$.
  Train the policy, $\pi_{\boldsymbol{\theta}}$, on the updated dataset, $\mathcal{D}$.
return Trained policy, $\pi_{\boldsymbol{\theta}}$.

Data Aggregation Methods: Data aggregation methods are another primary class of approaches within the context of Behavioral Cloning that address the covariate shift problem. DAGGER[9] is a well-known method for addressing the covariate shift problem through data aggregation techniques. Specifically, DAGGER seeks to reduce covariate shift by explicitly gathering expert demonstrations under the state distribution induced by the robot. As we outline in Algorithm 21.3, DAGGER follows an iterative two-step process. First, we allow the robot to interact with the environment and observe states according to its own state distribution, and then we relabel these states using expert demonstrations.

[9] 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 21.3: DAGGER Algorithm

Data: Initial dataset of expert demonstrations, $\mathcal{D}$, initial policy, $\pi_{\boldsymbol{\theta}}^1$, number of iterations, N
Result: Trained policy, $\pi_{\boldsymbol{\theta}}^N$
for $i = 1, 2, \ldots, N$ **do**
  Collect trajectories, $\tau = \{(\boldsymbol{x}_t, \boldsymbol{u}_t^{\text{robot}})\}$, using the policy $\pi_{\boldsymbol{\theta}}^i$.
  Gather dataset of states visited by the robot and actions given by the expert, $\mathcal{D}^i = \{(\boldsymbol{x}_t, \boldsymbol{u}_t^{\text{expert}})\}$.
  Aggregate the dataset, $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}^i$.
  Train the policy, $\pi_{\boldsymbol{\theta}}^{i+1}$, on the updated dataset, $\mathcal{D}$.
return Trained policy, $\pi_{\boldsymbol{\theta}}^N$.

In its simplest form, DAGGER begins by initializing the policy, $\pi_{\boldsymbol{\theta}}^1$, using a set of previously collected expert demonstrations. The robot then interacts with the environment with policy $\pi_{\boldsymbol{\theta}}^1$, collecting trajectories, τ, that reflect the state distribution under the current policy. We then relabel the actions of the trajectories using the expert's actions for the visited states. Then, we use the relabeled trajectories to train an updated policy, $\pi_{\boldsymbol{\theta}}^2$, which we then employ to collect additional trajectories under the state distribution induced by the updated policy. We repeat this process for a fixed number of iterations, resulting in the final trained policy, $\pi_{\boldsymbol{\theta}}^N$.

By collecting expert demonstrations under the state distribution induced by the robot's current policy, DAGGER effectively reduces covariate shift and enhances the performance of the learned policy. This method operates as a form of interactive supervised learning, where the robot actively collects data to refine its performance. This iterative process minimizes the amount of expert data required, making it more effective across a range of tasks.

In summary, confidence-based methods and data aggregation techniques both provide solutions for addressing the covariate shift problem in Behavioral Cloning. While there are many variations of these methods, the core principles we outline in Algorithm 21.2 and Algorithm 21.3 provide a foundational understanding of how to mitigate covariate shift through targeted data collection. However, methods following these principles still suffer from other common

limitations of Behavioral Cloning, such as dependence on the quality of expert demonstrations. In Section 21.2.2, we discuss approaches that leverage ideas from Behavioral Cloning to learn from broader, and potentially suboptimal, sets of expert demonstrations.

21.2.2 *Reinforcement Learning via Supervised Learning (RvS)*

Recent work[10] has explored the idea of converting the reinforcement learning problem, which we discussed in Chapter 20, into a *conditional*, *filtered*, or *weighted* imitation learning problem. In this class of approaches, we use the insight that rather than relying on optimal demonstrations, we can leverage a broader set of demonstrations from suboptimal policies or from diverse, but related, tasks. We often refer to these approaches as *reinforcement learning via supervised learning* (RvS) methods. These approaches typically involve conditioning on goals or reward values, but can also entail reweighting or filtering demonstrations.

[10] S. Emmons et al. "RvS: What is Essential for Offline RL via Supervised Learning?" In: 2021

Filtering or Weighting Demonstrations: One common approach to RvS is to filter or weight the expert demonstrations based on their quality[11] or relevance to the task. Revisiting the outline of the Behavioral Cloning algorithm in Algorithm 21.1, we can consider a modification that involves filtering the expert demonstrations based on their quality, as defined by the reward information.

[11] For example, if they obtain higher rewards.

A primitive approach could entail the following process. First, we rank the expert demonstrations based on their return[12]:

$$r(\tau) = \sum_{t=0}^{T-1} \gamma^t R(\boldsymbol{x}_t, \boldsymbol{u}_t).$$

[12] The *return* of a trajectory is the sum of rewards obtained along the trajectory. We rank trajectories by their long-term performance, rather than just the immediate rewards, for better overall behavior.

Then, we filter the dataset, $\mathcal{D}$, to include only the top k% of trajectories based on their return:

$$\tilde{\mathcal{D}} = \{\tau \in \mathcal{D} \mid r(\tau) \geq \bar{r}\}.$$

where $\bar{r}$ is the return such that k percent of the set $\mathcal{D}$ has a return higher than that amount. Finally, we train the policy, $\pi_{\boldsymbol{\theta}}$, on the filtered dataset, $\tilde{\mathcal{D}}$.

This represents a simple form of RvS, where we filter expert demonstrations based on their return. A more sophisticated approach might involve weighting each individual transition of a trajectory based on its reward, rather than filtering entire trajectories. We can measure the quality of individual actions by using its *advantage*[13] or *Q-value* instead of its immediate reward. Once we have computed action weights, we incorporate them into the Behavioral Cloning process by defining the loss function in Algorithm 21.1 as:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{u})\sim\tilde{\mathcal{D}}} \left[-\log \pi_{\boldsymbol{\theta}}(\boldsymbol{u} \mid \boldsymbol{x}) A(\boldsymbol{x}, \boldsymbol{u}) \right].$$

This is a modified version of the standard Behavioral Cloning loss where we now weight the action probabilities, $\log \pi_{\boldsymbol{\theta}}(\boldsymbol{u} \mid \boldsymbol{x})$, by their advantage values, $A(\boldsymbol{x}, \boldsymbol{u})$. This approach is known as *advantage-weighted Behavioral Cloning* and has been shown to improve the performance of Behavioral Cloning in practice.

[13] Recall from Chapter 20 that the advantage of an action is the difference between the action-value function and the value function, $A(\boldsymbol{x}_t, \boldsymbol{u}_t) = Q(\boldsymbol{x}_t, \boldsymbol{u}_t) - V(\boldsymbol{x}_t)$. Intuitively, the advantage represents how much better an action is compared to the average action.

Goal or Reward Conditioning: Another common approach to RvS is to condition the policy on a goal or reward value. This approach is particularly useful in settings where the expert demonstrations are suboptimal or collected from a different task. Consider a dataset of previously collected trajectories, $\mathcal{D} = \{\tau_i\}$. Each trajectory, τ_i, might be described using different outcomes[14], such as the final state of the trajectory, the total reward obtained, or a specific state visited during the trajectory. We denote a specific outcome occurring in a trajectory, τ, as ω. The goal of conditioning-based RvS is to learn an outcome-conditioned policy, $\pi_{\boldsymbol{\theta}}(\boldsymbol{u} \mid \boldsymbol{x}, \omega)$, that optimizes:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{u},\omega)\sim\mathcal{D}}\left[-\log \pi_{\boldsymbol{\theta}}(\boldsymbol{u} \mid \boldsymbol{x}, \omega)\right].$$

[14] In other words, a condition that is verified during or at the end of the trajectory.

Goal and state-conditioned RvS are particularly relevant approaches where the policy is conditioned on a specific state or goal outcome, $\omega = \boldsymbol{x} \in \mathcal{X}$, that the robot should reach. For example, in a robotic manipulation task, the robot might be conditioned on reaching a specific configuration. Another common form of conditioning is reward-conditioned RvS, where we condition the policy on a specific reward value, $\omega = \sum_{t=0}^{T-1} R(\boldsymbol{x}_t, \boldsymbol{u}_t)$.

In both cases, conditioning on outcomes enables the robot to extract meaningful information from suboptimal or diverse expert demonstrations, often leading to improved performance in practice. For example, consider two policies, $\pi_{\boldsymbol{\theta}_1}(\boldsymbol{u} \mid \boldsymbol{x})$ and $\pi_{\boldsymbol{\theta}_2}(\boldsymbol{u} \mid \boldsymbol{x}, \omega)$, that are trained on the same dataset, $\mathcal{D}$. Suppose $\pi_{\boldsymbol{\theta}_1}$ is trained to imitate expert demonstrations that implicitly optimize a specific reward function. This reward-centric approach restricts $\pi_{\boldsymbol{\theta}_1}$ to behaviors that closely follow the expert's trajectory distribution. On the other hand, $\pi_{\boldsymbol{\theta}_2}$ is goal-conditioned and trained to achieve any specified goal state, ω, independent of the underlying reward function. By explicitly incorporating the goal into its policy, $\pi_{\boldsymbol{\theta}_2}$ decouples the process of achieving desired outcomes from the reward structure. As a result, $\pi_{\boldsymbol{\theta}_2}$ is likely to generalize better to novel tasks or unseen goal states, as it learns a flexible mapping from states and goals to actions. In contrast, $\pi_{\boldsymbol{\theta}_1}$ remains constrained by the expert's reward-aligned demonstrations, making it less adaptable to scenarios with divergent or ambiguous reward structures.

21.3 *Inverse Reinforcement Learning*

In the previous section, we discussed Behavioral Cloning as a form of imitation learning that directly learns a policy from expert demonstrations. Inverse Reinforcement Learning (IRL)[15] takes an orthogonal approach to imitation learning by attempting to recover a reward function from a policy, or from demonstrations of a policy. In certain cases, identifying the reward function can offer deeper insights into the task's underlying structure, making it potentially more informative than directly learning a policy. Additionally, a policy that is optimal for the expert may not be optimal for the agent if they have different dynamics, morphologies, or capabilities[16].

[15] A. Ng and S. Russell. "Algorithms for Inverse Reinforcement Learning". In: *Proceedings of the Seventeenth International Conference on Machine Learning*. 2000, pp. 663–670

[16] Learned reward representations can also potentially generalize across different robot platforms that tackle similar problems.

Example 21.3.1 (Inverse Reinforcement Learning vs Behavioral Cloning)**.** Consider a scenario where the robot's objective is to drive across a city as quickly as possible. In the context of imitation learning, we assume the reward function is unknown, but an expert provides example routes to navigate the city. Behavioral Cloning approaches attempt to replicate the expert's actions, such as by learning to turn right at a particular intersection. This strategy lacks robustness since it can fail when the robot encounters intersections that the expert never visited. IRL approaches offer a more generalizable alternative by focusing on identifying key features of the expert's trajectories, rather than just mimicking actions. For example, instead of merely copying the expert's turns, the robot could learn to recognize useful patterns, such as preferring roads with higher speed limits or fewer stop signs. The robot can then develop a policy that takes routes with similar advantageous characteristics, even if they differ from the exact paths the expert took.

Formally, the goal of IRL is to recover a reward function, $R : \mathcal{X} \times \mathcal{U} \to \mathbb{R}$, from a set of expert demonstrations, $\mathcal{D} = \{\tau_i\}$, where $\tau_i = \{(\boldsymbol{x}_0, \boldsymbol{u}_0, \ldots, \boldsymbol{x}_T)\}$ is an example trajectory. We parameterize the reward function by parameters $\boldsymbol{w}$, and therefore our goal is to recover the reward function by optimizing $\boldsymbol{w}$ to best explain the expert demonstrations. We can then define a policy by optimizing the learned reward function, for example using methods from the previous chapters on sequential decision making and reinforcement learning.

In practice, many IRL algorithms require an iterative learning process consisting of two primary steps that are repeated until convergence. First, we update the reward function parameters, $\boldsymbol{w}$. Then, we adjust the policy parameters, $\boldsymbol{\theta}$, to maximize the current estimate of the reward function. While each IRL algorithm has a different way of performing these steps, Algorithm 21.4 provides a high-level overview of the general process.

Algorithm 21.4: High-level IRL Algorithm

Data: Expert demonstrations, $\mathcal{D}$, initialized reward function parameters, $\boldsymbol{w}$, initialized policy parameters, $\boldsymbol{\theta}$
Result: Learned reward function parameters, $\boldsymbol{w}$, learned policy parameters, $\boldsymbol{\theta}$
while *not converged* **do**
  Update the reward function parameters, $\boldsymbol{w}$.
  Update the policy parameters, $\boldsymbol{\theta}$, to maximize the current estimate of the reward function.
return Optimized reward and policy parameters: $\boldsymbol{w}$, $\boldsymbol{\theta}$.

One challenge in IRL is that the expert policy may be optimal for multiple reward functions and therefore the problem of recovering the reward function is inherently ill-posed. Numerous alternative objectives have been proposed to address this issue, such as optimizing for the maximum margin between the optimal and suboptimal policies, or maximum entropy formulations.

In the following sections, we first introduce the concept of *feature expectation* and then discuss three popular IRL methods: apprenticeship learning, maximum margin planning, and maximum entropy IRL.

21.3.1 Feature Expectation

We start with an assumption that a true reward function, R^*, exists and that we can express it as a linear combination of features:

$$R^*(\boldsymbol{x}, \boldsymbol{u}) = \boldsymbol{w}^{*\top} \boldsymbol{\phi}(\boldsymbol{x}, \boldsymbol{u}),$$

where $\boldsymbol{\phi} : \mathcal{X} \times \mathcal{U} \to [0,1]^d$ is some vector of features[17]. In Chapter 20, we saw that the value function for a policy, π, which we define as the expected cumulative discounted reward, is:

$$V_T^\pi(\boldsymbol{x}) = \mathbb{E}_{\tau \sim p_\pi(\tau)} \left[\sum_{t=0}^{T-1} \gamma^t R(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t)) \mid \boldsymbol{x}_0 = \boldsymbol{x} \right]. \tag{21.1}$$

[17] To ensure that the rewards are bounded by 1, we also assume that $\|\boldsymbol{w}^*\|_2 \leq 1$.

Using the reward function $R(\boldsymbol{x}, \boldsymbol{u}) = \boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}, \boldsymbol{u})$, we can rewrite the value function as:

$$V_T^\pi(\boldsymbol{x}) = \boldsymbol{w}^\top \mu(\pi, \boldsymbol{x}), \tag{21.2}$$

where:

$$\mu(\pi, \boldsymbol{x}) = \mathbb{E}_{\tau \sim p_\pi(\tau)} \left[\sum_{t=0}^{T-1} \gamma^t \boldsymbol{\phi}(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t)) \mid \boldsymbol{x}_0 = \boldsymbol{x} \right],$$

where we refer to $\mu(\pi, \boldsymbol{x})$[18] as the *feature expectation*.

[18] For brevity, we may also denote $\mu(\pi, \boldsymbol{x})$ as simply $\mu(\pi)$.

An important insight is that, by definition, the optimal expert policy, π^*, will always yield a value function greater than or equal to that of any other policy and therefore:

$$V_T^{\pi^*}(\boldsymbol{x}) \geq V_T^\pi(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \mathcal{X}, \quad \forall \pi.$$

We can equivalently express this condition in terms of the feature expectation as:

$$\boldsymbol{w}^{*\top} \mu(\pi^*, \boldsymbol{x}) \geq \boldsymbol{w}^{*\top} \mu(\pi, \boldsymbol{x}), \quad \forall \boldsymbol{x} \in \mathcal{X}, \quad \forall \pi. \tag{21.3}$$

Theoretically, we can derive the expert's reward vector, $\boldsymbol{w}^*$, by finding a vector, $\boldsymbol{w}$, that satisfies this inequality. However, this can potentially lead to ambiguities, such as the fact that $\boldsymbol{w} = 0$ trivially satisfies this condition. Reward ambiguity is a key challenge in IRL, and the algorithms we discuss in the following sections provide techniques to address this issue.

21.3.2 Apprenticeship Learning

The apprenticeship learning algorithm[19] addresses the problem of reward ambiguity by finding a policy, π, such that the feature expectation induced by π is close to that of the expert policy, π^*. Mathematically, the goal of apprenticeship learning is to find a policy such that $\|\mu(\pi, \boldsymbol{x}) - \mu(\pi^*, \boldsymbol{x})\|_2 \leq \epsilon$ for all $\boldsymbol{x} \in \mathcal{X}$,

[19] P. Abbeel and A. Ng. "Apprenticeship Learning via Inverse Reinforcement Learning". In: *Proceedings of the Twenty-First International Conference on Machine Learning*. 2004

where ϵ is a small positive constant. For such a policy, π, we would have that for any $\boldsymbol{w}$ with $\|\boldsymbol{w}\|_2 \leq 1$:

$$\begin{aligned} |V_T^{\pi}(\boldsymbol{x}) - V_T^{\pi^*}(\boldsymbol{x})| &= |\boldsymbol{w}^\top \mu(\pi, \boldsymbol{x}) - \boldsymbol{w}^\top \mu(\pi^*, \boldsymbol{x})|, \\ &\leq \|\boldsymbol{w}\|_2 \|\mu(\pi, \boldsymbol{x}) - \mu(\pi^*, \boldsymbol{x})\|_2 \\ &\leq 1 \cdot \epsilon = \epsilon, \end{aligned} \tag{21.4}$$

where the first equality follows from the definition of the value function as a function of the feature expectation in Equation (21.2), the first inequality follows from the fact that $|x^\top y| \leq \|x\|_2 \|y\|_2$ for any vectors x and y, and the second inequality follows from the assumption that $\|\boldsymbol{w}\|_2 \leq 1$. In practice, this leads to a practical reformulation of the IRL problem where as long as we match the feature expectations, such that $\|\mu(\pi, \boldsymbol{x}) - \mu(\pi^*, \boldsymbol{x})\|_2 \leq \epsilon$, the performance of the learned policy will be comparable to the expert even if $\boldsymbol{w}$ does not match $\boldsymbol{w}^*$.

Within this context, the IRL problem is reduced to finding a policy, π, that induces feature expectations, $\mu(\pi)$, that are close to those of the expert, $\mu(\pi^*)$. We provide a schematic overview of apprenticeship learning in Algorithm 21.5. At iteration i in Algorithm 21.5, we have already found policies $\pi_0, \pi_1, \ldots, \pi_{i-1}$

Algorithm 21.5: Apprenticeship Learning

Data: Expert's feature expectations, $\mu^* = \mu(\pi^*)$, initial policy, π_0
Result: Learned parameters, $\boldsymbol{w}$, and policy, $\hat{\pi}^*$
$i \leftarrow 1$
while *true* **do**
  Compute $\mu^{(i-1)} = \mu(\pi_{i-1})$ (or approximate via Monte Carlo methods).
  Compute $t^{(i)} = \max_{w:|w|_2 \leq 1} \min_{j \in \{0,\ldots,(i-1)\}} \boldsymbol{w}^\top \left(\mu^* - \mu^{(j)}\right)$ by solving:

$$\begin{aligned} (\boldsymbol{w}_i, t_i) \leftarrow \max_{\boldsymbol{w}, t}\ & t, \\ \text{s.t. } & \boldsymbol{w}^\top \mu^* \geq \boldsymbol{w}^\top \mu^{(j)} + t, \quad \forall j \in \{0, \ldots, (i-1)\}, \\ & \|\boldsymbol{w}\|_2 \leq 1. \end{aligned} \tag{21.5}$$

  if $t_i \leq \epsilon$ **then**
    $\hat{\pi}^* \leftarrow$ best feature matching policy from $\{\pi_0, \ldots, \pi_{i-1}\}$
    $\hat{\boldsymbol{w}}^* \leftarrow \boldsymbol{w}_i$
    return $\hat{\pi}^*$, $\hat{\boldsymbol{w}}^*$
  Compute an optimal policy, π_i, for the reward function defined by $\hat{\boldsymbol{w}}$.
  $i \leftarrow i + 1$

and the corresponding feature expectations $\mu^{(0)}, \mu^{(1)}, \ldots, \mu^{(i-1)}$. Within the inner loop, we formulate the optimization problem defined by Equation (21.5) to find the reward function vector, $\boldsymbol{w}$, being optimized by the expert. Specifically, from Equation (21.2), we can view the constraint $\boldsymbol{w}^\top \mu^* \geq \boldsymbol{w}^\top \mu^{(j)} + t$ as $V_T^{\pi^*}(\boldsymbol{x}_0) \geq V_T^{\pi_j}(\boldsymbol{x}_0) + t$. In other words, we are seeking to find the reward function parameters, $\boldsymbol{w}$, for which the expert does better by a margin of t than

any of the policies we previously found. We then compute a new policy[20], π_i, that is optimal for the reward function $\boldsymbol{w}_i$. This process continues until the margin, t_i, is less than a predefined threshold, ϵ, meaning that we have sufficiently matched the expert's feature expectations.

[20] For example, using reinforcement learning methods from Chapter 20.

21.3.3 Maximum Margin Planning

Maximum margin planning (MMP)[21] is a generalization of apprenticeship learning that aims to find a reward function that maximally separates the expert policy from a set of policies. Specifically, MMP modifies Equation (21.5) from the apprenticeship learning algorithm as follows:

$$\begin{aligned} \hat{\boldsymbol{w}}^* = \arg\min_{\boldsymbol{w},\xi} \; & \|\boldsymbol{w}\|_2^2 + C\xi, \\ \text{s.t. } & \boldsymbol{w}^\top \mu^* \geq \boldsymbol{w}^\top \mu^{(j)} + m(\pi^*, \pi^{(j)}) - \xi, \quad \forall j \in \{0, \ldots, (i-1)\}, \end{aligned} \tag{21.6}$$

[21] 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

where $m(\pi, \pi')$ is a distance function[22] between any two policies π and π', ξ represents a slack variable allowing for constraint violations, and C is a hyperparameter used to penalize constraint violations. Intuitively, through this modified formulation, MMP enforces that the margin should be larger for policies that are very different from π^*.

[22] For example, $m(\pi, \pi')$ could represent the number of states in which π and π' disagree.

An advantage of the MMP formulation over the apprenticeship learning approach in Equation (21.5) arises when the expert is suboptimal. In such cases, it may be impossible to find a reward vector, $\boldsymbol{w}$, that makes the expert policy outperform all other policies This causes the apprenticeship learning optimization problem defined by Equation (21.5) to return trivial values of $\boldsymbol{w}_i = 0$ and $t_i = 0$. In contrast, the MMP formulation incorporates slack variables that allow us to compute a meaningful reward vector $\boldsymbol{w}$ when the expert is imperfect.

21.3.4 Maximum Entropy Inverse Reinforcement Learning

As we described in Section 21.3, the IRL problem in inherently ill-posed since there are infinitely many reward functions that could explain the expert's behavior. While maximum margin approaches are highly effective when there is a single reward function that is clearly better than alternatives, in some cases, optimizing for a distribution of reward functions is more appropriate. Maximum entropy inverse reinforcement learning (MaxEnt IRL)[23] aims to find a distribution over reward functions that explains the expert's behavior that ideally matches the feature expectations of the expert[24] and has maximum entropy[25].

[23] 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

[24] Similar to maximum margin methods.

[25] In other words, is as "random" as possible.

We denote the distribution over trajectories induced by a policy, π, as $p_\pi(\tau)$, and we can write the feature expectations in terms of this distribution as:

$$\mu(\pi) = \mathbb{E}_\pi\left[f(\tau)\right] = \int p_\pi(\tau) f(\tau) d\tau,$$

where $f(\tau) = \sum_{t=0}^{T-1} \gamma^t \phi(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t))$. In practice, MaxEnt IRL proposes to learn a

policy that maximizes the entropy:

$$\mathcal{H}(p(\tau)) = \int -p(\tau) \log p(\tau) d\tau, \tag{21.7}$$

subject to the constraints:

$$\begin{aligned} \int p(\tau) f(\tau) d\tau &= \int p_{\pi^*}(\tau) f(\tau) d\tau, \\ \int p(\tau) d\tau &= 1, \end{aligned} \tag{21.8}$$

where the first constraint enforces the feature expectations of the robot and expert policies to match, and the second constraint enforces $p(\tau)$ to represent a valid probability distribution.

Among the distributions that satisfy the constraint $\int p(\tau) f(\tau) d\tau = \int p_{\pi^*}(\tau) f(\tau) d\tau$, the maximum entropy distribution follows the exponential form:

$$p(\tau) \propto \exp(\boldsymbol{w}^\top f(\tau)).$$

Specifically, we can express the trajectory distribution as a function of $\boldsymbol{w}$ as:

$$p(\tau \mid \boldsymbol{w}) = \frac{1}{Z(\boldsymbol{w})} \exp\left(\boldsymbol{w}^\top f(\tau)\right), \tag{21.9}$$

where $Z(\boldsymbol{w})$ is the partition function given by $Z(\boldsymbol{w}) = \int \exp\left(\boldsymbol{w}^\top f(\tau)\right) d\tau$.

However, Equation (21.9) only holds for deterministic environments where the next state is fully determined by the current state and action. In stochastic environments, the trajectory distribution is also influenced by the random environment dynamics, and in this case we express the distribution over trajectories as:

$$p(\tau \mid \boldsymbol{w}) = \frac{1}{Z(\boldsymbol{w})} \exp\left(\boldsymbol{w}^\top f(\tau)\right) \prod_{\{\boldsymbol{x}_{t+1}, \boldsymbol{u}_t, \boldsymbol{x}_t\} \in \tau} p(\boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \boldsymbol{u}_t). \tag{21.10}$$

We can therefore obtain the parameter vector, $\boldsymbol{w}$, for the reward function by maximizing the likelihood of the observed data under the maximum entropy distribution defined in Equation (21.10) as:

$$\hat{\boldsymbol{w}}^* = \arg\max_{\boldsymbol{w}} \mathcal{L}_{\text{MLE}} = \arg\max \sum_{\tau \in \mathcal{D}} \ln p(\tau \mid \boldsymbol{w}). \tag{21.11}$$

References

[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] S. Arora and P. Doshi. "A survey of inverse reinforcement learning: Challenges, methods and progress". In: *Artificial Intelligence* 297 (2021), p. 103500.

[9] S. Chernova and M. Veloso. "Interactive policy learning through confidence-based autonomy". In: *Journal of Artificial Intelligence Research* 34.1 (2009), pp. 1–25. ISSN: 1076-9757.

[14] S. Emmons et al. "RvS: What is Essential for Offline RL via Supervised Learning?" In: 2021.

[50] A. Ng and S. Russell. "Algorithms for Inverse Reinforcement Learning". In: *Proceedings of the Seventeenth International Conference on Machine Learning*. 2000, pp. 663–670.

[51] T. Osa et al. "An Algorithmic Perspective on Imitation Learning". In: (2018). URL: https://arxiv.org/abs/1811.06711.

[55] 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.

[56] 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.

[79] 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.

Part V

Robot Software

22
Robot System Architectures

A robotic system is fundamentally a collection of sensors and actuators that can interact with the environment to accomplish a set of tasks. While this definition is 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. In this chapter, we 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

22.1 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 the robot must also be handle to ensure it 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 static and dynamic obstacles in its environment, should be robust to sensor failures or sensor noise, and more.

Definition 22.1.1 (Robot Goal)**.** Complete desired tasks while monitoring and reacting to unexpected situations. Handle inputs and outputs from actuators and sensors in real-time[2].

[2] Real-time requirements are crucial. Some functions require near instantaneous reactions, such as on the order of a millisecond.

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, we define the robot's *system architecture* by two major parts, the *structure* and the *style*. The structure defines how the system is broken down into components, as well as how the components interact with each other[3]. 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. However, there are paradigms that have been proven to be

useful, which we introduce 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 performance limiting.

[4] For example, subsystem segmentation can be useful for reusability as well as validation and unit-testing.

22.2 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 a common choice for this decomposition, where tasks at one level of the hierarchy are composed of a group of tasks from lower-levels of the hierarchy. This structure reduces complexity through abstraction.

22.2.1 Sense-Plan-Act Architecture

The sense-plan-act architecture is one of the first structures developed, and consists of three main subsystems: sensing, planning, and execution. These components are organized in a sequential fashion, with sensor data being passed to the planner, which then passes information to the controller, which generates actuator commands. This approach has significant drawbacks. First, the planning component is a computational bottleneck that holds up the controller subsystem, which might ideally operate at a higher frequency. Second, since the controller does not have direct access to sensor data, the overall system is not very reactive.

22.2.2 Subsumption Architecture

An alternative to the sense-plan-act architecture 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, we can view this architecture as layers of finite state machines[6] that all connect sensors to actuators, and where multiple behaviors are evaluated in parallel. We would include an arbitration mechanism to choose which of the behaviors is currently activated. For example, an "explore" behavior may sit on top of, or in other words, subsume, a collision avoidance behavior, and the arbitration mechanism would decide when the exploration behavior should be overridden by the collision avoidance 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.

[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 is referred to as a *behavior*.

22.2.3 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*: The planning layer is at the highest-level of the hierarchy, and focuses on task-planning for long-term goals.
2. *Executive*: The executive layer is the middle layer of the hierarchy, 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, which provides reactive planning, with better long-term planning capabilities from the planning level. We now discuss each of these levels in 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 to identify if the current situation is appropriate for a specific behavior, but they are not typically responsible for knowing how to change the situation, which is the responsibility of the executive level.

[7] This layer includes algorithms from classical control theory, such as PID control and Kalman filtering.

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 many 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 or by leveraging 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, the executive considers temporal constraints on behaviors, such as 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, which can be very abstract, with shorter horizon planning[8].

[8] This split might also be useful for computational performance reasons.

Example 22.2.1 (Office Mail Delivery Robot)**.** To further explore the components of the three-tiered robot system architecture, we can consider a robot whose primary task is to deliver mail within an office setting. Tasks that we might require this robot to perform 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, and recharge its batteries.

Using a three-tiered architecture, 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, such as by considering the relative locations of each delivery. The planner could also choose when to schedule time for recharging.

Given a task from the planner, such as "Deliver package to Rm 009", the executive level coordinates how to accomplish the task. This might include subtasks such as moving to the end of the hallway, opening the door, entering Rm 009, announcing delivery, and then waiting and monitoring 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 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 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 is to open a door and the door is locked, the behavioral control level should eventually recognize failure and report back to the executive level.

22.3 *Architecture Styles*

In addition to choosing the robot system architecture, another 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 defines the method for communicating among the planning, executive, and behavioral control levels, or even between components of each individual level. We typically refer to the implementation of the connection style as *middleware*, and we refer to two of the most common architecture styles as *client-server* and *publish-subscribe*.

22.3.1 Client-Server

Middleware based on the client-server style consists of message requests from clients that the server responds to[9]. We can think of this type of connection style as *on-demand* messaging. One of the disadvantages of this messaging style is that the client typically waits for the response from the server before continuing, leading to potential deadlocks, for example if the server crashes.

[9] In other words, there is a request-response message pairing.

22.3.2 Publish-Subscribe

Middleware based on the publish-subscribe style uses asynchronous message broadcasting from publishers, which other components of the system can subscribe to as needed. One disadvantage of this approach is that the interfaces are less well-defined[10], but the main advantage is in reliability, since deadlocks cannot occur and therefore 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.

[10] Interactions are only one-way.

References

[7] R. Brooks. "A robust layered control system for a mobile robot". In: *IEEE Journal on Robotics and Automation* 2.1 (1986), pp. 14–23.

[33] D. Kortenkamp, R. Simmons, and D. Brugali. "Robotic Systems Architectures and Programming". In: *Springer Handbook of Robotics*. Springer, 2008, pp. 283–302.

23
The Robot Operating System

This chapter introduces the fundamentals of the Robot Operating System (ROS)[1,2], a popular framework for creating robot software. Unlike what its name suggests, ROS is not an operating system. 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 used in both academia and industry.

In this chapter, we begin by introducing specific challenges in robot programming that motivate the need for a middleware such as ROS. Afterwards, we present a brief history of ROS to shed some light on its development and motivations for its important features. Next, we discuss the fundamental operating structure of ROS in further detail to provide insights into how ROS is operated on real robotic platforms. Lastly, we present specific features and tools of the ROS environment that greatly simplify robot software development. It is important to note that ROS is an active project and is constantly changing, and there are many free resources available for up-to-date documentation and examples.

[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

23.1 Challenges in Robot Programming

Robot programming is a subset of computer programming, but it differs 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[3]. 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, we look for the following features when developing software for robots:

[3] For example, the potentially many sensors and actuators onboard the robot.

1. *Multitasking:* A robot often consists of a number of sensors and actuators, and therefore robot software needs to enable multitasking to work with different input/output devices in different threads at the same time. Each thread also needs to be able to communicate with other threads to exchange data.
2. *Low Level Device Control:* Robot software needs to be compatible with a wide

variety of input and output devices, such as general purpose input/output (GPIO) pins, USB, SPI, among others. We should also be able to support multiple programming languages that are common among low-level devices, such as C, C++ and Python.

3. *High level Object Oriented Programming (OOP):* In OOP, code is encapsulated, inherited, and reused as multiple instances. Having the ability to reuse code and develop programs in independent modules makes it easy to maintain code for complex robots.

4. *Availability of 3rd Party Libraries and Community Support:* The ability to leverage third-party libraries and having community support can help expedite software development and facilitate efficient software implementation.

23.2 A Brief History of ROS

Until the advent of ROS, it was difficult for individual 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 vision that it should be free and open-source for everyone to encourage collaboration and community development, it should make core components[4] of robotics readily available for anyone, and it should integrate seamlessly with existing robotics frameworks[5].

Development of ROS started to gain traction when Scott Hassan, a software architect and entrepreneur, and his startup, Willow Garage, took over the project to develop a standardized robotics development platform. While mostly self-funded by Scott Hassan, ROS 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. In 2012, the Open Source Robotics Foundation (OSRF) started to supervise the future of ROS by supporting development, distribution, 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 many years, and the platform has become what is closest to the industry standard in robotics.

[4] Ranging from hardware to software packages.

[5] For example, OpenCV for computer vision, SLAM packages for localization and mapping, and the Gazebo simulation software.

23.3 Characteristics of ROS

The ROS framework provides the following important capabilities:

1. *Modularity:* Robots can be complex systems, and ROS handles this complexity through modularity. Specifically, we can independently develop each robot

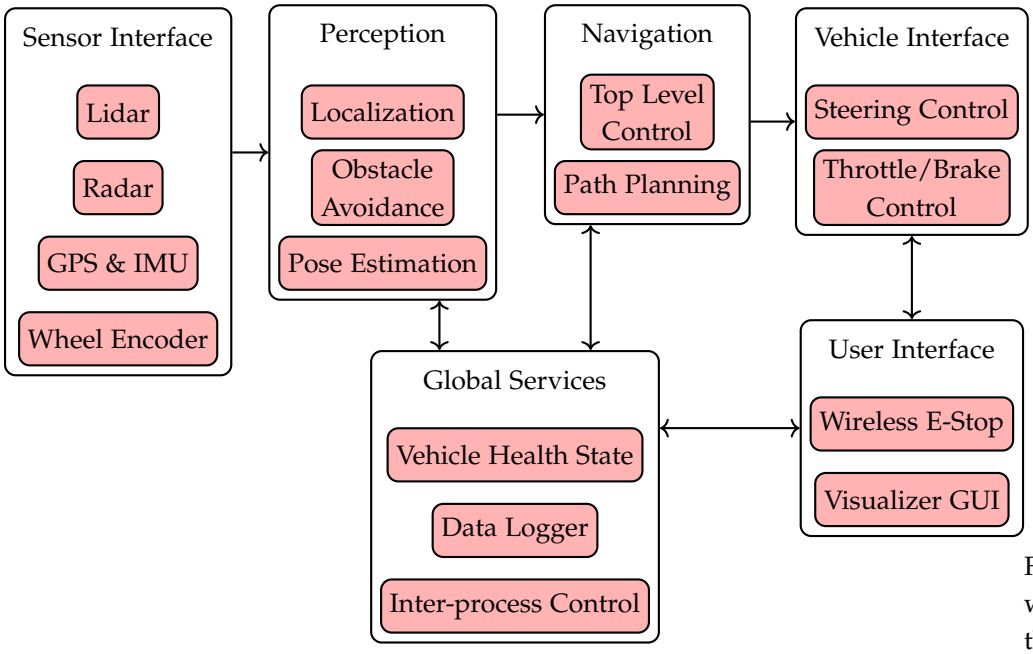


Figure 23.1: Modular software architectures help handle the complexity of robot programming. We can define the interior units of each module as ROS *nodes* that live within a common *package*. Arrows denote communication pathways.

software component that performs a separate function in units called *nodes*. Each node can share data with other nodes, and nodes act as the basic building blocks of ROS. We can develop different functional capabilities in units called *packages* that can contain a number of nodes defined from source code, configuration files, and data files, and we can distribute and install packages on other computers. This architecture of nodes and packages is visualized in Figure 23.1, where the interior blocks can represent nodes that are organized into packages.

2. *Message Passing:* ROS provides a message passing interface that allows nodes 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. ROS passes messages using a *publish/subscribe* structure, which we show graphically in Figure 23.2. This message passing scheme allows messages to be passed between ROS nodes through a shared virtual "chat room" called a ROS *topic*.

3. *Built-in Algorithms:* A lot of popular robotics algorithms are already built-in and available as off-the-shelf packages. For example, the ROS community has developed libraries for PID control, SLAM, and path planners such as A* and Dijkstra[6]. These built-in algorithms can significantly reduce time needed to prototype a robot.

[6] The documentation for these packages can be readily found online.

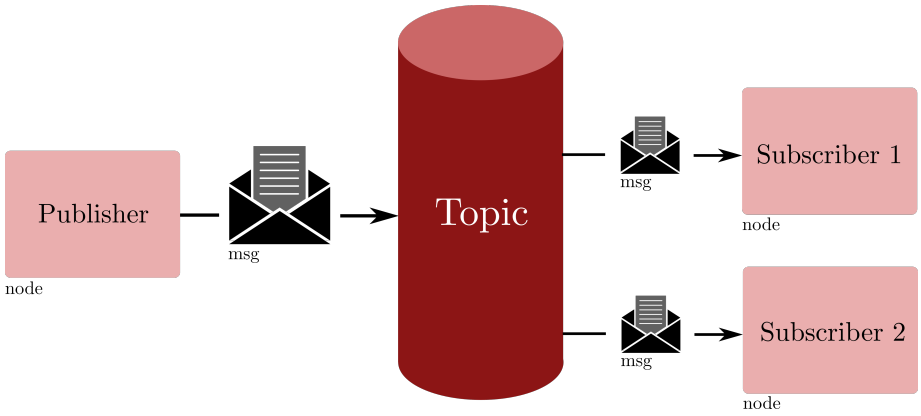


Figure 23.2: The ROS publish/-subscribe (pub/sub) model.

4. *Third-party Libraries and Community Support:* The ROS framework is developed with pre-existing third-party libraries in mind. We can integrate many popular libraries, such as OpenCV for computer vision and the Point Cloud Library (PCL), with a couple lines of code. In addition, ROS is supported by active developers all over the world and there are several forums for questions and discussion.

23.4 Robot Programming with ROS

As we mentioned before, ROS uses a publish/subscribe message passing scheme for communicating among nodes, as we show in Figure 23.2. There are several primary features of ROS that are crucial to support this communication structure: *nodes* that define individual components that send or receive information[7], *messages* that are objects for holding information that is communicated between nodes, and *topics* that are virtual "chat rooms" where messages are shared[8].

[7] In other words, a unit of computation.

[8] In ROS2, there are other options including *services* and *actions*, which have slightly more communication structure, such as defining request and response structures.

23.4.1 Nodes

Definition 23.4.1 (Node)**.** A node is a process that performs a unit of computation. 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 we might develop 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.

23.4.2 Messages

Definition 23.4.2 (Messages)**.** Nodes can communicate with each other by publishing simple data structures to topics. These data structures are called *messages*, and when a node sends a message there is no expectation of receiving a response[9].

[9] Unlike other ROS structures that we don't dive into in this chapter, such as *services* and *actions*.

We define a message object by a set of 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, we can also define more complex field types for specific applications. For example, suppose a sensor packet node publishes sensor data as an array of user-defined `SensorData` objects. This message, which we could call `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 one of the standard primitive types, such as an integer, floating point, or boolean, arrays of primitive types, or other user-defined types. Messages can also include arbitrarily nested structures and arrays. Primitive message types available in ROS are listed below in Table 23.1. 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.

23.4.3 Topics

Definition 23.4.3 (Topics)**.** Topics are named units over which nodes exchange messages using a publish/subscribe system.

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

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 23.1: Built-in ROS Messages

that type of message. Any number of nodes can publish or subscribe to a given topic. Fundamentally, topics are for unidirectional, streaming communication. This is not well suited for all types of communication, such as communication that demands a response like a service routine.

References

[25] L. Joseph. *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*. Apress, 2018.

[53] M. Quigley, B. Gerkey, and W. D. Smart. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. O'Reilly Media, 2015.

References

[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] S. Arora and P. Doshi. "A survey of inverse reinforcement learning: Challenges, methods and progress". In: *Artificial Intelligence* 297 (2021), p. 103500.

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

[4] Y. Bai et al. "Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback". In: (2022). URL: https://arxiv.org/abs/2204.05862.

[5] D. Bertsekas. *Nonlinear Programming*. Athena Scientific, 2016.

[6] D. Bertsekas. *Reinforcement learning and optimal control*. Athena Scientific, 2019.

[7] R. Brooks. "A robust layered control system for a mobile robot". In: *IEEE Journal on Robotics and Automation* 2.1 (1986), pp. 14–23.

[8] Nicolas Carion et al. "End-to-End Object Detection with Transformers". In: *Computer Vision – ECCV 2020*. Springer International Publishing, 2020, pp. 213–229.

[9] S. Chernova and M. Veloso. "Interactive policy learning through confidence-based autonomy". In: *Journal of Artificial Intelligence Research* 34.1 (2009), pp. 1–25. ISSN: 1076-9757.

[10] Roy De Maesschalck, Delphine Jouan-Rimbaud, and Désiré L Massart. "The mahalanobis distance". In: *Chemometrics and intelligent laboratory systems* 50.1 (2000), pp. 1–18.

[11] Frank Dellaert and Michael Kaess. "Square root SAM: Simultaneous localization and mapping via square root information smoothing". In: *The International Journal of Robotics Research* 25.12 (2006), pp. 1181–1203.

[12] A. Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *International Conference on Learning Representations*. 2021.

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

[14] S. Emmons et al. "RvS: What is Essential for Offline RL via Supervised Learning?" In: 2021.

[15] Martin A. Fischler and Robert C. Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". In: *Commun. ACM* 24.6 (1981), pp. 381–395.

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

[17] 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.

[18] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[19] F. Gustafsson. *Statistical Sensor Fusion*. Studentlitteratur, 2013, p. 554.

[20] C. Harris and M. Stephens. "A combined corner and edge detector". In: *4th Alvey Vision Conference*. 1988.

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

[22] J. Hertling. "Numerical Methods for Two-Point Boundary Value Problems (Herbert B. Keller)". In: *SIAM Review* 12.2 (1970), pp. 313–315.

[23] Jonathan P. How. *Lecture Notes for Principles of Optimal Control*. 2008.

[24] 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.

[25] L. Joseph. *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*. Apress, 2018.

[26] L. Kaelbling et al. *6.01SC: Introduction to Electrical Engineering and Computer Science I*. MIT OpenCourseWare. 2011.

[27] Michael Kaess, Ananth Ranganathan, and Frank Dellaert. "iSAM: Incremental smoothing and mapping". In: *IEEE Transactions on Robotics* 24.6 (2008), pp. 1365–1378.

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

[29] 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.

[30] M. Kelly. "An Introduction to Trajectory Optimization: How to Do Your Own Direct Collocation". In: *SIAM Review* 59.4 (2017), pp. 849–904.

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

[32] M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray. *Algorithms for Decision Making*. MIT Press, 2022.

[33] D. Kortenkamp, R. Simmons, and D. Brugali. "Robotic Systems Architectures and Programming". In: *Springer Handbook of Robotics*. Springer, 2008, pp. 283–302.

[34] Rainer Kümmerle et al. "g 2 o: A general framework for graph optimization". In: *2011 IEEE international conference on robotics and automation*. IEEE. 2011, pp. 3607–3613.

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

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

[37] John J Leonard and Hugh F Durrant-Whyte. "Simultaneous map building and localization for an autonomous mobile robot." In: *IROS*. Vol. 3. 1991, pp. 1442–1447.

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

[39] S. Levine et al. "End-to-End Training of Deep Visuomotor Policies". In: *Journal of Machine Learning Research* 17.39 (2016), pp. 1–40.

[40] 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.

[41] David G Lowe. "Object recognition from local scale-invariant features". In: *Proceedings of the seventh IEEE international conference on computer vision*. Vol. 2. Ieee. 1999, pp. 1150–1157.

[42] T. Lozano Perez. "Spatial planning: a configuration space approach". In: *Autonomous Robot Vehicles*. 1990.

[43] Feng Lu and Evangelos Milios. "Robot pose estimation in unknown environments by matching 2d range scans". In: *Journal of Intelligent and Robotic systems* 18.3 (1997), pp. 249–275.

[44] K. M. Lynch and K. C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017. Chap. 8.

[45] V. Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *Proceedings of The 33rd International Conference on Machine Learning*. 2016, pp. 1928–1937.

[46] Michael Montemerlo et al. "FastSLAM: A factored solution to the simultaneous localization and mapping problem". In: *Aaai/iaai* 593598.2 (2002), pp. 593–598.

[47] H. P. Moravec. "Towards automatic visual obstacle avoidance". In: *5th International Joint Conference on Artificial Intelligence*. 1977.

[48] K. P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022.

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

[50] A. Ng and S. Russell. "Algorithms for Inverse Reinforcement Learning". In: *Proceedings of the Seventeenth International Conference on Machine Learning*. 2000, pp. 663–670.

[51] T. Osa et al. "An Algorithmic Perspective on Imitation Learning". In: (2018). URL: https://arxiv.org/abs/1811.06711.

[52] 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.

[53] M. Quigley, B. Gerkey, and W. D. Smart. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. O'Reilly Media, 2015.

[54] A. Rao. "A Survey of Numerical Methods for Optimal Control". In: *Advances in the Astronautical Sciences* 135 (2010).

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] M. D. Shuster. "Survey of attitude representations". In: *Journal of the Astronautical Sciences* 41.4 (1993), pp. 439–517.

[60] B. Siciliano and O. Khatib. *Springer Handbook of Robotics*. Springer-Verlag, 2007.

[61] B. Siciliano et al. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 2008. Chap. 7.

[62] B. Siciliano et al. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 2008.

[63] Bruno Siciliano et al. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 2008. Chap. 2.

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

[65] D. Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[66] D. Simon. *Optimal State Estimation: Kalman, H_∞, and Nonlinear Approaches*. John Wiley & Sons, 2006.

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

[68] Randall Smith, Matthew Self, and Peter Cheeseman. "Estimating uncertain spatial relationships in robotics". In: *Autonomous robot vehicles*. Springer, 1990, pp. 167–193.

[69] R. Sutton and A. Barto. *Reinforcement learning: An introduction*. MIT Press, 2018.

[70] R. S. Sutton. "Dyna, an integrated architecture for learning, planning, and reacting". In: *SIGART Bull.* 2.4 (1991), pp. 160–163.

[71] R. Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

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

[73] Bill Triggs et al. "Bundle adjustment—a modern synthesis". In: *Vision Algorithms: Theory and Practice: International Workshop on Vision Algorithms Corfu, Greece, September 21–22, 1999 Proceedings*. Springer. 2000, pp. 298–372.

[74] 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.

[75] C. J. C. H. Watkins and P. Dayan. "Q-learning". In: *Machine Learning* 8.3 (1992), pp. 279–292.

[76] R. J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8.3 (1992), pp. 229–256.

[77] M. D. Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks". In: *European Conference on Computer Vision (ECCV)*. Springer, 2014, pp. 818–833.

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

[79] 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.