Gioele Zardini, Joseph Lorenzetti,
Marco Pavone

Principles of Robot Autonomy

SEPTEMBER 24, 2024

Forward

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

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

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

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

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

Acknowledgments

These notes accompany (and are based largely on the content of) the courses *AA274A: Principles of Robot Autonomy I* and *AA274B: Principles of Robot Autonomy II*[2] at Stanford University. We would therefore like to acknowledge the students who have taken the course and provided useful feedback since its initial offering in 2017. Special acknowledgements are also reserved for the course assistants:

AA274A, Winter 2017: Andrew Bylard, Benoit Landry, Ed Schmerling,

AA274A, Winter 2018: Tommy Hu, Benoit Landry, Karen Leung, Ed Schmerling,

AA274A, Winter 2019: Christopher Covert, Amine Elhafsi, Karen Leung, Apoorva Sharma,

AA274A, Autumn 2019: Andrew Bylard, Boris Ivanovic, Jenna Lee, Toki Migimatsu, Apoorva Sharma,

AA274A, Autumn 2020: Somrita Banerjee, Abhishek Cauligi, Boris Ivanovic, Mengxi Li, Joseph Lorenzetti,

AA274B, Winter 2020: Ashar Alam, Erdem Bıyık, Jenna Lee, Toki Migimatsu,

AA274B, Winter 2021: Erdem Bıyık, Abhishek Cauligi,

who were instrumental in developing and refining the course material. In large part, additional material for the course such as homework and lectures are also publicly available[3].

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

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

Contents

PART ~ART.1I ROBOT MOTION PLANNING AND CONTROL

1 *Modeling Robot Dynamics* 7
- 1.1 *State Space Models* 7
- 1.2 *Kinematics and Dynamics* 11
- 1.3 *Wheeled Robot Motion Models* 20
- 1.4 *Simulating Robot Dynamics* 23

2 *Open-Loop Motion Planning & Control* 27
- 2.1 *Open-loop Optimal Control* 28
- 2.2 *Trajectory Generation with Differentially Flat Dynamics Models* 31
- 2.3 *Exercises* 38

3 *Closed-Loop Motion Planning & Control* 39
- 3.1 *Linear Closed-loop Control* 40
- 3.2 *Nonlinear Closed-loop Control* 43
- 3.3 *Trajectory Tracking Control* 45
- 3.4 *Code Exercises* 47

4 *Optimal Control and Trajectory Optimization* 49
- 4.1 *Indirect Methods* 50
- 4.2 *Direct Methods* 55

5 *Search-Based Motion Planning* 59
- 5.1 *Grid-based Motion Planners* 61
- 5.2 *Combinatorial Motion Planning* 67
- 5.3 *Exercises* 69

6 *Sampling-Based Motion Planning* 71
- 6.1 *Probabilistic Roadmap (PRM)* 72
- 6.2 *Rapidly-exploring Random Trees (RRT)* 73
- 6.3 *Theoretical Results for PRM and RRT* 74

6.4 *Fast Marching Tree Algorithm (FMT*)* 75
6.5 *Kinodynamic Planning* 76
6.6 *Deterministic Sampling-Based Motion Planning* 77
6.7 *Exercises* 78

PART ~ART.2II ROBOT PERCEPTION

7 *Introduction to Robot Sensors* 83
7.1 *Sensor Classifications* 83
7.2 *Sensor Performance* 84
7.3 *Common Sensors on Mobile Robots* 87
7.4 *Computer Vision* 92

8 *Camera Models and Calibration* 97
8.1 *Perspective Projection* 97
8.2 *Camera Calibration: Direct Linear Method* 102
8.3 *Camera Auto-Calibration* 107
8.4 *Limitations* 110
8.5 *Exercises* 111

9 *Stereo Vision and Structure From Motion* 113
9.1 *Stereo Vision* 114
9.2 *Structure From Motion (SFM)* 118

10 *Image Processing* 121
10.1 *Image Filtering* 121
10.2 *Image Feature Detection* 127
10.3 *Image Descriptors* 131
10.4 *Exercises* 131

11 *Information Extraction* 133
11.1 *Geometric Feature Extraction* 134
11.2 *Object Recognition* 140
11.3 *Exercises* 141

12 *Modern Computer Vision Techniques* 143
12.1 *Convolutional Neural Networks* 144
12.2 *Transformers* 147
12.3 *Modern Object Detection and Localization* 148

PART ~ART.3III ROBOT LOCALIZATION

13 *Introduction to Localization and Filtering* 153
13.1 *Preliminary Concepts in Probability* 154
13.2 *Markov Models* 160
13.3 *Bayes Filter* 161

14 *Parametric Filters* 165
14.1 *The Gaussian Distribution* 166
14.2 *Kalman Filter* 167
14.3 *Extended Kalman Filter (EKF)* 171
14.4 *Exercises* 173
14.5 *Exercises* 173

15 *Nonparametric Filters* 175
15.1 *Histogram Filter* 176
15.2 *Particle Filter* 177
15.3 *Exercises* 178

16 *Robot Localization* 181
16.1 *A Taxonomy of Robot Localization Problems* 181
16.2 *Robot Localization via Bayesian Filtering* 183
16.3 *Markov Localization* 185
16.4 *Extended Kalman Filter (EKF) Localization* 185
16.5 *Monte Carlo Localization (MCL)* 190

17 *Simultaneous Localization and Mapping (SLAM)* 193
17.1 *EKF SLAM Algorithm* 193
17.2 *EKF SLAM with Unknown Correspondences* 196
17.3 *Particle SLAM* 198
17.4 *Exercises* 202

18 *Sensor Fusion and Object Tracking* 203
18.1 *A Taxonomy of Sensor Fusion* 204
18.2 *Bayesian Approach to Sensor Fusion* 205
18.3 *Practical Challenges in Sensor Fusion* 207
18.4 *Object Tracking* 208

PART ~ART.4IV ROBOT DECISION MAKING

19 *Finite State Machines* 213
19.1 *Finite State Machines* 213

19.2 *Finite State Machine Architectures* 216
19.3 *Implementation Details* 218

20 *Sequential Decision Making* 221
20.1 *Deterministic Decision Making Problem* 222
20.2 *Stochastic Decision Making Problem* 227
20.3 *Challenges and Extensions of Dynamic Programming* 231

21 *Reinforcement Learning* 233
21.1 *Problem Formulation* 234
21.2 *Model-based Reinforcement Learning* 236
21.3 *Model-free Reinforcement Learning* 240
21.4 *Deep Reinforcement Learning* 244
21.5 *Exploration vs Exploitation* 244

22 *Imitation Learning* 245
22.1 *Problem Formulation* 245
22.2 *Behavior Cloning* 246
22.3 *DAgger: Dataset Aggregation* 247
22.4 *Inverse Reinforcement Learning* 247
22.5 *Learning From Comparisons and Physical Feedback* 253
22.6 *Interaction-aware Control and Intent Inference* 254

PART ~ART.5V ROBOT SOFTWARE

23 *Robot System Architectures* 261
23.1 *Architecture Structures* 262
23.2 *Architecture Styles* 264

24 *The Robot Operating System* 267
24.1 *Challenges in Robot Programming* 267
24.2 *Brief History of ROS* 268
24.3 *Robot Programming with ROS* 270
24.4 *Writing a Simple Publisher Node and Subscriber Node* 273
24.5 *Other Features in ROS Development Environment* 276

PART ~ART.6VI ADVANCED TOPICS IN ROBOTICS

25 *Formal Methods* 283
25.1 *Linear Temporal Logic* 284
25.2 *Verification* 286
25.3 *Reactive Synthesis* 287

26 *Robotic Manipulation* 291
26.1 *Grasp Modeling* 292
26.2 *Grasp Evaluation* 296
26.3 *Grasp Force Optimization* 300
26.4 *Learning-Based Approaches to Grasping* 302
26.5 *Learning-Based Approaches to Manipulation* 303

PART ~ART.7VII APPENDICES

A *Machine Learning* 309
A.1 *Loss Functions* 310
A.2 *Model Training* 311
A.3 *Neural Networks* 313
A.4 *Backpropagation and Computational Graphs* 315

Part I

Robot Motion Planning and Control

1
Modeling Robot Dynamics

A robot can take on many different forms: it can consist of rigid or flexible structures, it can be controlled through various forms of actuation, it can perceive the world through a diverse set of sensors, or it may even exist as a purely non-physical agent. One thing that almost all robotic systems have in common is that they are *dynamic* agents whose states change over time. Physical motion is one of the most obvious and common examples of robot dynamics, including changes in position, velocity, joint angles, and sensor orientations. It is important to understand the dynamical nature of a robotic system to improve the design and functional performance of the robot. For example, designing a bipedal robot that can walk and run requires a strong understanding of the dynamic motions involved to ensure the structure and actuators are adequate for the required forces and torques, as well as to design a control system that can achieve the desired motion.

In this chapter we first introduce a mathematical framework for modeling the dynamics of robotic systems. This framework, referred to as a *state space model*, consists of two key components: a *state* and a *model* that describes how the state changes in time and relates to the *inputs* and *outputs* of the system. We will then introduce some techniques for developing models for the physical motion of mobile robots, one of the most common applications in robotics. We will also be introducing and using other dynamics models throughout the other chapters of this book.

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)**.** A *state* is a set of variables that are sufficient to characterize the future behavior (i.e., future states) of the robotic system from an initial state, given the external inputs to the system.

The state of a robotic system can be finite or infinite-dimensional. For exam-

ple a simple mobile robot with a rigid body may be represented by the position and velocity of its center of mass (a finite and low-dimensional state), but a flexible robot may require an infinite-dimensional state to describe the body's continuous deformation.

It is important to note, however, that when modeling the dynamics of a robot we can *choose the state definition to best fit the application and purpose of the model*. In practice we can generally reduce the complexity of the state definition by ignoring parts of the robot's dynamics that aren't relevant to the problem being solved. Consider an autonomous car: a roboticist working on the motion planning problem may choose to consider only the position, orientation, and velocities in the state, ignoring variables related to the dynamics of the engine, tires, and suspension system[1]. In this book we will focus on applications and methods where the state is finite-dimensional, and will represent the state as a vector, $\boldsymbol{x} \in \mathbb{R}^n$, which we will refer to as the *state vector*.

[1] Alternatively, a roboticist working on an autonomous vehicle *racing* application may find these variables are very important to include in the state definition.

The second key component is a *model* that relates the state to a set of *inputs* and *outputs*. The *inputs*[2] to a model are the external factors that induce a change to the system's state. We generally choose the input definition in conjunction with the state to best fit the purpose of the model. Referring again to an autonomous car, a high-fidelity model used for low-level trajectory tracking control might use steering rates and throttle as inputs, while a lower-fidelity model used for path planning might abstract away the low-level tracking dynamics and use a simpler longitudinal acceleration and yaw-rate input definition. Similar to the state, the dimension of the input can vary, but we will assume in this book that the input is represented by a finite-dimensional vector, $\boldsymbol{u} \in \mathbb{R}^m$. The *output* of a model represents a measurable or observable variable that is a function of the state. The choice of a model's measurable variables is constrained by the types of sensors available to the robot. For example, a robot with a global navigation satellite system (GNSS) sensor can get direct measurements of the position state but not direct measurements of the heading. The relationship between the state and the output can also become quite complex, such as the relationship between the robot's inertial pose and a red-green-blue-depth (RGB-D) camera image. The reconstruction of the state from outputs is referred to as *state estimation*, which we discuss in more detail in later chapters. We again assume the output is finite-dimensional and represent it with the vector $\boldsymbol{y} \in \mathbb{R}^q$.

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

The change in the state induced by the inputs is typically modeled by a differential equation describing the change of the state with respect to an independent variable, which is usually time[3]:

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

[3] Ordinary differential equations are most common, but partial differential equations can also be used for more complex robotic systems, such as robots with flexible structures.

where $\dot{\boldsymbol{x}} = \frac{d\boldsymbol{x}(t)}{dt}$ and:

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

The function $f\colon \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}^n$ defines the *dynamics* of the state and $h\colon \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}^q$ defines the relationship between the state and output. The definitions

of the state $\boldsymbol{x}$, input $\boldsymbol{u}$, output $\boldsymbol{y}$, and the models defined in Equation (1.1) and Equation (1.2) together form the *state space model*. There are also many instances where it is more appropriate to represent the dynamics using a discrete-time difference equation. Discrete-time models are useful when modeling certain types of systems that don't evolve continuously in time, or to *approximate* a continuously varying system in a discrete computational setting.

In this book we will continue to assume the independent variable is time, and will simplify our notation by writing θ for $\theta(t)$. We will also continue to use the $\dot{\theta}$ notation to represent a derivative with respect to time, with $\ddot{\theta} = \frac{d^2\theta(t)}{dt^2}$ for second derivatives and $\theta^{(m)} = \frac{d^m\theta(t)}{dt^m}$ for higher-order derivatives.

1.1.1 Types of State Space Models

The models in Equation (1.1) and Equation (1.2) can be time-invariant or time-varying and linear or nonlinear. A model is *time-invariant* if the functions f and h do not explicitly depend on time t, and a model is *linear*[4] if the functions f and h are linear in both the state, x, and control, u.

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

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

$$\begin{aligned} \dot{x} &= x + u \\ y &= x \end{aligned}$$

is linear and time-invariant.

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

$$\begin{aligned} \dot{x} &= tx + u \\ y &= x^2 \end{aligned}$$

is nonlinear and time-varying.

We commonly write linear state space systems in 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 $A(t), B(t), C(t), D(t)$[5] are matrices of appropriate dimensions. We refer to the matrix A as the *dynamics* matrix, B as the *control* matrix, C as the *output* or *sensor* matrix, and D as the *direct* or *feed-forward* matrix. Linear models are widely used in robotics algorithms due to their simplicity. Even if the nominal dynamics model is nonlinear, we will often *approximate* the system with a linear model through a process called *linearization*.

[5] When the matrices A, B, C, D are constant we refer to the system as a linear time-invariant (LTI) system.

1.1.2 Converting Higher-Order Models Into State Space Form

The state space model described by Equation (1.1) is a first order differential equation, but we will also encounter higher-order differential equation models

in many robotics applications. For example, Newton's second law is a second order differential equation in position! However, we can convert higher-order differential equation models into the 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. We can convert this differential equation into state space form (i.e., into a first-order differential equation) by defining an n-dimensional state space:

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

Since $\dot{x}_2 = x_1$, $\dot{x}_3 = x_2$ and so on, we can now write a first order state space model as:

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

In this particular case the model is also linear, so we can also write the state space model in matrix form:

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

Example 1.1.3 (Newton's Second Law)**.** Newton's second law describes the acceleration response of a mass m from a force F:

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

where s is the one-dimensional position (i.e., displacement) of the mass. This second order system[6] can be converted into state space form by choosing the state to be:

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

We then write the state space model as:

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

where u represents the force F.

[6] Models of the form $\ddot{\theta} = u$ are a classic structure referred to in control theory as the *double integrator*.

1.2 Kinematics and Dynamics

Motion planning and control are fundamental tasks for robot autonomy[7]. For example, autonomous vehicles must plan trajectories and determine appropriate control inputs (e.g., throttle and steering) to navigate to new locations, and robot manipulators must create motion and grasping plans for object manipulation[8]. A key component for solving these tasks is to understand the robot's physical motion, which is governed by its *kinematics* and *dynamics*.

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

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

Definition 1.2.1 (Kinematics)**.** A robot's *kinematics* describe restrictions (constraints) on the robot's motion that reduce its degrees of freedom.

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

Definition 1.2.2 (Dynamics)**.** A robot's *dynamics* govern how forces or inputs acting on the robot change its physical state.

In the context of physical robot motion, dynamics are typically the result of Newton's Second Law. For example, the dynamics of an autonomous vehicle are characterized by the relationship between its acceleration and external forces such as tire friction, gravity, and aerodynamics.

A robot's kinematics and dynamics describe limitations on its motion in different ways[9], and are defined by the robot's design, geometry, mass, and other physical characteristics. In this section we introduce the concept of *generalized coordinates* for defining the physical configuration of a robot and discuss how we can define the robot's kinematics and dynamics in terms of these coordinates. In the next section we will show 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 see how these models can be used to develop robust and high-performing motion planning and control algorithms.

[9] A good heuristic for determining how a particular constraint/relationship should be classified is that dynamics are affected by changing the robot's mass, while kinematics are not.

1.2.1 Generalized Coordinates

A robot's physical state or *configuration* can be represented in different ways, and we refer to a specific selection of parameters used to define the robot's configuration as *generalized coordinates*[10], denoted by the vector $\boldsymbol{\xi}(t) \in \mathbb{R}^{n_g}$. Note that the generalized coordinates for a robotic system are typically going to represent only a subset of the overall state $\boldsymbol{x}$. We refer to the time derivatives of the generalized coordinates, $\dot{\boldsymbol{\xi}}$, as *generalized velocities*.

[10] We often use "configuration" and "generalized coordinates" interchangeably, even though the specific choice of generalized coordinates are not necessarily the only possible representation of the robot's configuration.

Example 1.2.1 (Rolling Wheel)**.** The configuration of the wheel rolling on a plane in Figure 1.1 can be represented by three parameters: the contact point

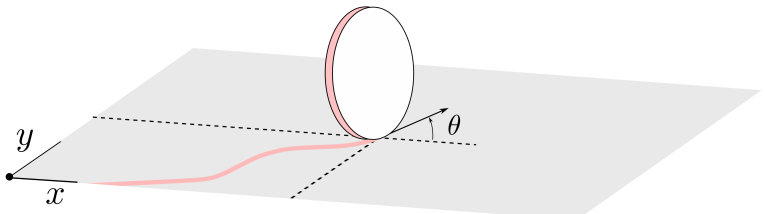


Figure 1.1: Generalized coordinates for a wheel rolling without slipping on a plane.

position coordinates (x, y), and the heading angle θ relative to a fixed reference frame. This set of parameters, $\boldsymbol{\xi} = [x, y, \theta]^\top$, are just one possible set of generalized coordinates to define the wheel's configuration. Another possible choice of generalized coordinates for this system would be the heading, θ, and a polar coordinate representation of the contact point position.

1.2.2 *Kinematic Constraints*

Once we choose a set of generalized coordinates $\boldsymbol{\xi}$ to represent a robot's configuration we can determine what *kinematic constraints* are relevant for the robot. Kinematic constraints define a relationship between the generalized coordinates and the generalized velocities and therefore describe limitations on the robot's motion.

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

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

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

In many robotics applications the kinematic constraints are *linear* with respect to the generalized velocities. We call constraints of this kind *Pfaffian constraints* and we write them mathematically as:

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

where $a_i(\boldsymbol{\xi}) \in \mathbb{R}^{n_g}$. We also express Pfaffian constraints compactly in matrix form as:

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

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

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.

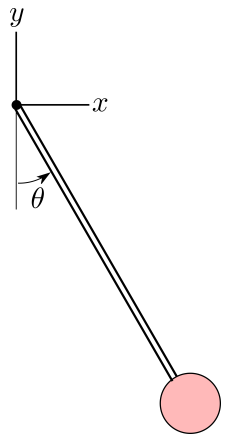


Figure 1.2: Generalized coordinates for a simple pendulum.

The generalized coordinate vector for this choice is $\boldsymbol{\xi} = [x, y]^\top$, and the generalized velocity vector is $\dot{\boldsymbol{\xi}} = [\dot{x}, \dot{y}]^\top$. The fact that the rod connecting the pivot point to the mass is rigid introduces a restriction on the motion of this system, which is represented by the kinematic constraint:

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

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:

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

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

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

and therefore the constraint can be written in Pfaffian form (1.6) with:

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

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

An alternative choice of generalized coordinates to represent the pendulum's configuration is to simply consider the angle θ between the vertical and the pendulum's rod orientation (i.e. $\boldsymbol{\xi} = [\theta]$). 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 that since $x = L\sin\theta$ and $y = -L\cos\theta$ that the kinematic constraint (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 $\boldsymbol{\xi} = [x, y, \theta]^\top$. 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 $e_v = [\cos\theta, \sin\theta]^\top$ describes the heading of the wheel, the lateral direction is given by the perpendicular unit vector $e_{v,\perp} = [\sin\theta, -\cos\theta]^\top$. We can compute the lateral velocity from the dot product of the lateral direction unit vector and the velocity vector $v = [\dot{x}, \dot{y}]^\top$, which gives the no-slip kinematic constraint:

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

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 under the cateogries of *holonomic* or *nonholonomic*, depending on how they restrict the motion of the system. Holonomic

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

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

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

are called *holonomic*.

Holonomic constraints are a unique subclass of kinematic constraints that *restrict the accessible configurations of the system*[11]. In fact, the space of accessible configurations for a system with n generalized coordinates under k holonomic constraints will have dimension $n - k$. Holonomic constraints can also *always* be equivalently expressed as Pfaffian constraints (1.6) since:

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

and by differentiating the expression

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

as we demonstrated in Example 1.2.2. However, it is important to note that *not all Pfaffian constraints are holonomic*. A Pfaffian constraint is only holonomic if it is integrable to the form in Equation (1.11).

[11] A system that is only subject to holonomic constraints is referred to as a *holonomic system*.

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 it can be expressed 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 (i.e. position and heading). Mathematically, the constraint in Equation (1.10) *cannot* be integrated to yield a constraint of the form $\tilde{a}_i(\boldsymbol{\xi}) = 0$, and thus this constraint is not holonomic.

While holonomic constraints restrict the system's accessible configurations, it is possible to have kinematic constraints that alternatively restrict the *motion between configurations*. These constraints are referred to as *nonholonomic* constraints[12].

[12] 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, $a_i(\boldsymbol{\xi})^\top \dot{\boldsymbol{\xi}} = 0$, but cannot be integrated to the form $\tilde{a}_i(\boldsymbol{\xi}) = 0$ form are called *nonholonomic*.

Another way to interpret the influence of a set of nonholonomic constraints on a system is to note that they restrict the instantaneous generalized velocities to lie in the null space of $A(\boldsymbol{\xi})^\top$.

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

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

The null space of $a_1(\boldsymbol{\xi})^\top$ 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 $[0,\ 0,\ 1]$ axis the wheel rotates about.

1.2.4 Kinematic Models

Once we have chosen an appropriate set of generalized coordinates $\boldsymbol{\xi}$ 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 $\boldsymbol{\xi} \in \mathbb{R}^{n_g}$ and k Pfaffian constraints[13], $A^\top(\boldsymbol{\xi})\dot{\boldsymbol{\xi}} = \mathbf{0}$, a *kinematic model* is a state space model of the form:

$$\dot{\boldsymbol{\xi}} = G(\boldsymbol{\xi})\boldsymbol{u}, \tag{1.13}$$

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

[13] Note these Pfaffian constraints can come from a combination of holonomic and non-holonomic constraints.

Each input in $\boldsymbol{u}$ corresponds to one degree of freedom of the system, and for any initial condition $\boldsymbol{\xi}(0)$ 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:

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

where $g_i \in \mathbb{R}^{n_g}$ 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 $\dot{\boldsymbol{\xi}}$ through a particular mode that is defined by the vector g_i. Since we have specified in Definition 1.2.6 that the column space of G spans the null space of $A^\top(\boldsymbol{\xi})$ we know that by definition:

$$A^\top(\boldsymbol{\xi})g_i(\boldsymbol{\xi})u_i = 0,$$

for any input $u_i \in \mathbb{R}$ and for all coordinates $\boldsymbol{\xi}$. 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:

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

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

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

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

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}\cos\theta & 0\\ \sin\theta & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}u_1\\ u_2\end{bmatrix}. \tag{1.14}$$

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 constraints and kinematic models are fundamental for describing restrictions on a robot's physical motion, but a robot's *dynamics* describe how forces and torques actually create or inhibit motion[14]. For example, a robot arm's motion is kinematically constrained to rotations about its joints, but the arm's dynamics govern how gravity and actuation torques can change the arm's momentum.

[14] Note that kinematic constraints can be the result of forces, but these forces do not do work on the system. For example the no-slip wheel kinematic constraint is the result of friction forces, but these forces will not change the momentum of the wheel.

Newton's laws of motion are the foundation of robot dynamics. Applied to a single particle mass, Newton's second law describes how an external force will change the position and velocity in time:

$$\boldsymbol{F}(\boldsymbol{\xi}, \dot{\boldsymbol{\xi}}, t) = m\ddot{\boldsymbol{\xi}}, \tag{1.15}$$

where m is the particle's mass, $\boldsymbol{\xi} = [x, y, z]^\top$ is the position of the particle, and $\boldsymbol{F}$ is a vector of forces acting on the particle.

Example 1.2.8 (Mass-Spring-Damper System)**.** The one-dimensional mass-spring-damper system is one of the most classic examples of in the study of dynamics. In this system the forces acting on the mass include external forces

that can be a function of time, a spring force that is proportional to the displacement of the spring, and a damping term that is proportional to the displacement rate:

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

where x is the displacement of the spring from its equilibrium position and $k, c > 0$ are positive coefficients. The mass-spring-damper dynamics are used as a simple model to approximate second order oscillatory motion in a number of robotics applications.

The dynamics of a single particle, governed by Newton's second law, can also be extended to general systems of particles. In robotics the "systems of particles" are often assumed to be *rigid bodies*[15], which allows us to greatly simplify the dynamics models. The dynamics of a rigid body can be described in two parts: translational dynamics and rotational dynamics. Translational dynamics can be described by Equation (1.15) where the position is the position of the center of mass. For general motion in three dimensions there are three translational degrees of freedom for the rigid body system. Rotational dynamics can be described by[16]:

$$\boldsymbol{M} = \dot{\boldsymbol{H}}, \tag{1.16}$$

where $\boldsymbol{M}$ is an external moment applied to the body and $\boldsymbol{H}$ is the angular momentum about a fixed point, usually the center of mass. There are also three degrees of freedom associated with the rotational dynamics of a system, corresponding to the orientation in a three dimensional space. While position is a very common generalized coordinate definition for translational motion, there are several useful generalized coordinate choices for rotational motion including *Euler angles* and *quaternions*.

An alternative to the Newton-Euler method (i.e., Equation (1.15) and Equation (1.16)) for deriving the dynamics equations for a rigid body is known as the *Lagrange Method*[17]. 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 we begin by defining a scalar quantity known as the *Lagrangian*:

$$L(\boldsymbol{\xi}, \dot{\boldsymbol{\xi}}, t) = T(\boldsymbol{\xi}, \dot{\boldsymbol{\xi}}, t) - V(\boldsymbol{\xi}, t), \tag{1.17}$$

where $T(\boldsymbol{\xi}, \dot{\boldsymbol{\xi}}, t)$ is the kinetic energy of the system and $V(\boldsymbol{\xi}, t)$ is the potential energy. *Lagrange's equations* then relate the Lagrangian to Pfaffian constraints[18] and external influences that act on the system:

$$\begin{aligned} \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\xi}_j}\right) - \frac{\partial L}{\partial \xi_j} &= Q_j + \sum_{i=1}^{k} \lambda_i a_{ij}(\boldsymbol{\xi}), \quad j = 1, \ldots, n_g, \\ a_i^\top(\boldsymbol{\xi})\dot{\boldsymbol{\xi}} &= 0, \qquad i = 1, \ldots, k, \end{aligned} \tag{1.18}$$

where $Q_j \in \mathbb{R}$ is a non-conservative[19] *generalized force* corresponding to the generalized coordinate ξ_j, a_{ij} is the j-th term of the Pfaffian constraint vector $a_i(\boldsymbol{\xi})$,

[15] Interesting examples in robotics where we may not be able to assume rigid bodies include soft robots, robots with soft end-effectors, or robots with flexible light-weight structures.

[16] This equation for rotational dynamics is generally referred to as *Euler's equation*.

[17] The Lagrange method can be simpler than the Newton-Euler method for deriving the dynamics for more complex multi-body systems common in robotics.

[18] In the general case Lagrange's method handles arbitrary kinematic constraints, but for simplicity we restrict our attention to constraints in Pfaffian form.

[19] Non-conservative generalized forces are generalized forces not derivable from a potential function. For example gravity would be classified as conservative and friction would be classified as non-conservative.

and $\lambda_i \in \mathbb{R}$ is a *Lagrange multiplier*. The first n_g equations of (1.18) describe the dynamics of the generalized coordinates due to external generalized forces and Pfaffian kinematic constraints, and the remaining k equations are the Pfaffian constraints from (1.6). This system of equations has $n_g + k$ equations and $n_g + k$ unknowns (the generalized coordinates and the Lagrange multipliers), and is generally referred to as the *standard non-holonomic form*. If the system is holonomic and the generalized coordinates are chosen to be independent we can instead leverage a simpler version of Lagrange's equations:

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

Note here that the number of generalized coordinates, n_g, is equal to the number of degrees of freedom of the system.

Example 1.2.9 (Pendulum)**.** Consider again the pendulum shown in Figure 1.2. We can model the dynamics of the pendulum by considering how gravity influences the motion of the mass. In this example we will show four different methods for deriving the equations of motion, using both Cartesian and polar coordinates and the Newton-Euler method and Lagrange method. This will show how the choice of generalized coordinates can impact the complexity of the approach.

First we use Newton's second law to derive the dynamics by analyzing the two forces that are acting on the mass: gravity and the force from the rod. The rod's force is assumed to act only axially along the rod and from the fixed-length constraint we know the rod's force must balance the gravity force along the axial direction and produce centripetal acceleration. The rod's force, acting axially along the rod, is:

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

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 gravity force is simply:

$$F_g = mg,$$

acting along the negative y-direction. The net force acting on the mass in the Cartesian coordinate system is therefore:

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

From Newton's second law (1.15) we can write the equations of motion for Cartesian coordinates x and y as:

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

To fully have the equations expressed in Cartesian coordinates we can use $x = L\sin\theta$ and $y = -L\cos\theta$ to get:

$$\begin{aligned} \ddot{x} &= \frac{1}{L^2}(gxy - xv^2), \\ \ddot{y} &= -\frac{1}{L^2}(gx^2 + yv^2). \end{aligned} \tag{1.20}$$

with $v^2 = \dot{x}^2 + \dot{y}^2$. This approach can be a little tricky, since it requires being able to do a force analysis which implicitly includes the constraints. Again using Cartesian coordinates, we can see how the Lagrange method can be a little simpler. First we define the kinetic and potential energies:

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

and in this case we have no external non-conservative generalized forces (since gravity is a conservative force). From Example 1.2.2 we also know we have a single Pfaffian constraint:

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

From Equation (1.18) we have the system of equations:

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

We now can solve for the Lagrange multiplier by leveraging the Pfaffian constraint by noting that:

$$x\dot{x} + y\dot{y} = 0 \implies \dot{x}^2 + \dot{y}^2 + x\ddot{x} + y\ddot{y} = 0,$$

and substituting the first two of Lagrange's equations for $\ddot{x}$ and $\ddot{y}$ gives:

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

which we can solve for λ:

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

where we have used that $L^2 = x^2 + y^2$ and again used $v^2 = \dot{x}^2 + \dot{y}^2$. Substituting the Lagrange multiplier back into Lagrange's equations and doing some simplifying algebra gives:

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

which is identical to the result (1.20) from Newton's method!

Now that we have used both the Newton-Euler and Lagrange methods in Cartesian coordinates, we can see that for this system things become much

simpler by choosing to use polar coordinates! To use Euler's equation (1.16) for rotational dynamics we will consider the coordinate system to be fixed at the pivot point. The force of gravity acting on the mass induces a moment on the system about the pivot point that is equal to:

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

and the angular momentum of the system about the pivot point is:

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

where mL^2 is the *moment of inertia*[20] of the system about the pivot point. Therefore from Euler's equation we have the dynamics:

$$\ddot{\theta} = -\frac{g}{L}\sin\theta, \tag{1.22}$$

which is quite a bit simpler than in Cartesian coordinates. The Lagrange method also becomes simpler when using polar coordinate, θ, because we do not have to worry about Pfaffian constraints. In polar coordinates the kinetic and potential energies are:

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

Using (1.18) we don't have any non-conservative generalized forces and no constraints, so we simply have the single equation:

$$\ddot{\theta} = -\frac{g}{L}\sin\theta, \tag{1.23}$$

which as expected is identical to the result (1.22) from using Euler's equation.

[20] The moment of inertia in Euler's equation is the analogue to mass in Newton's second law.

1.3 Wheeled Robot Motion Models

Robots exhibit diverse shapes, sizes, and configurations and feature a range of mobility options, but wheeled robots are particularly common due to their excellent mobility and simple design. In this section we show 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*.

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 from Example 1.2.3, essentially assuming the robot is constrained only by a no-slip constraint from a single wheel. Figure 1.3 shows a simplified diagram of the generalized coordinates for the unicycle model and the kinematic model is the same as the model in Equation (1.14):

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}\cos\theta & 0\\ \sin\theta & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}v\\ \omega\end{bmatrix}, \tag{1.24}$$

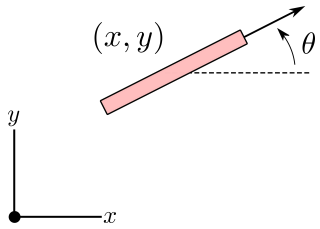


Figure 1.3: Generalized coordinates for a unicycle.

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

While this model may be a drastic simplification of the robot's true kinematics, it can still be useful to solve common robotics problems where high-fidelity motion models aren't required, such as high-level motion planning. The advantage of such a low-fidelity model is its simplicity, which often leads to much more computationally efficient algorithms. In practice, simple kinematic models like this are complemented with higher fidelity models as needed, for example for low-level trajectory tracking or for locally optimizing motion plans created by the unicycle model.

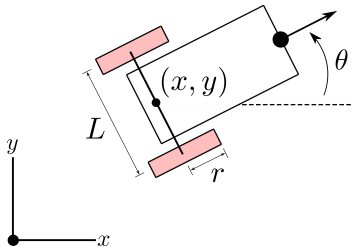


Figure 1.4: Generalized coordinates for a differential drive robot.

1.3.2 *Differential Drive Model*

The differential drive model is a variation on the unicycle model from Section 1.3.1 that assumes that two wheels are fixed on a shared rear axle, with a passive wheel that induces no kinematic constraints in the front. This model uses same generalized coordinates as the unicycle model, $\boldsymbol{\xi} = [x, y, \theta]^\top$, but we also have to define the geometry of the robot by the width of the rear axle, denoted by L, and the radius of the wheel, denoted by r (see Figure 1.4).

The differential drive model assumes the wheels roll without slipping and therefore the derivation of the kinematic constraints is similar to the single rolling wheel from Example 1.2.3. The heading vector of each wheel is given by $e_v = [\cos\theta, \sin\theta]^\top$ and therefore the lateral direction is $e_{v,\perp} = [\sin\theta, -\cos\theta]^\top$. From the lateral direction vector we define wheel's no-slip kinematic constraints by:

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

where $\dot{p}_l$ and $\dot{p}_r$ are the left and right wheel velocity vectors, respectively. Next we express the wheel velocity vectors, $\dot{p}_l$ and $\dot{p}_r$, as functions of the generalized coordinates and generalized velocities by leveraging the robot's geometry. First we can see that the position of the left and right wheel centers are computed from the generalized coordinates by:

$$p_l = \begin{bmatrix} x - \frac{L}{2}\sin\theta \\ y + \frac{L}{2}\cos\theta \end{bmatrix}, \quad p_r = \begin{bmatrix} x + \frac{L}{2}\sin\theta \\ y - \frac{L}{2}\cos\theta \end{bmatrix},$$

where p_l and p_r are the positions of the left and right wheels. The derivative of the positions with respect to time gives the velocity vectors:

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

After some algebraic manipulation we can see that the no-slip kinematic constraints on each wheel are equal:

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

which means having the no-slip constraint on both wheels is redundant and the constraint is the same as for the single wheel in Example 1.2.3. This makes

intuitive sense because the wheels are rigidly connected together, so if one wheel cannot move laterally then neither can the other. The kinematic model for the differential drive model is also identical to the single wheel model Equation (1.24), but the *inputs* can now be expressed in a more realistic form with respect to the actual geometry of the robot.

In particular, instead of the inputs being the forward speed v and body rotation rate ω as in Equation (1.24), the inputs for the differential drive model are the left and right wheel rotation rates, ω_l and ω_r. We can derive a relationship between these sets of inputs by exploiting the geometry of the robot and the no-slip wheel assumption. First we express the position $p = [x, y]^\top$ in terms of the wheel center positions by $p = \frac{1}{2}(p_l + p_r)$ and therefore the velocity vector is $\dot{p} = \frac{1}{2}(\dot{p}_l + \dot{p}_r)$. Due to the no-slip wheel assumption we can express the speed by $v = e_v^\top \dot{p}$ and then simplify:

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

where r is the radius of the wheel and v_l and v_r are the speeds of the left and right wheels. Additionally, the no-slip condition on each individual wheel is $v_l = e_v^\top \dot{p}_l$ and $v_r = e_v^\top \dot{p}_r$ which we expand to:

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

Noting that $\dot{x}\cos\theta + \dot{y}\sin\theta = v$ we simplify these expressions as $\frac{L}{2}\dot{\theta} = v_r - v$ and $\frac{L}{2}\dot{\theta} = v - v_l$. Combining these gives:

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

which relates the generalized velocity $\dot{\theta}$ to the wheel rotational speeds. In summary, we can define a mapping between the inputs:

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

which we can use to define the differential drive model:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \frac{r}{2}\cos\theta & \frac{r}{2}\cos\theta \\ \frac{r}{2}\sin\theta & \frac{r}{2}\sin\theta \\ \frac{r}{L} & -\frac{r}{L} \end{bmatrix} \begin{bmatrix} \omega_r \\ \omega_l \end{bmatrix}. \tag{1.25}$$

The overall complexity of this model has not increased significantly over the unicycle model, but by leveraging the geometry of the robot the inputs to this

model are more intuitive. This improvement on the unicycle model can make it more suitable for some motion planning and control tasks since the robot's actuation is generally from motors attached to the wheel's axles.

1.3.3 Bicycle/Simple Car Model

The bicycle model is a kinematic model that includes front and rear axles with no-slip wheels on each. Unlike the unicycle and differential drive models, the bicycle model has more complex kinematic constraints on how the vehicle can turn, and can better approximate motion for robots such as four-wheeled vehicles.

Consider the car model corresponding to Figure 1.6:

$$\begin{aligned} \dot{x} &= v\cos\theta, \\ \dot{y} &= v\sin\theta, \\ \dot{\theta} &= \frac{v}{L}\tan\phi, \end{aligned} \tag{1.26}$$

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

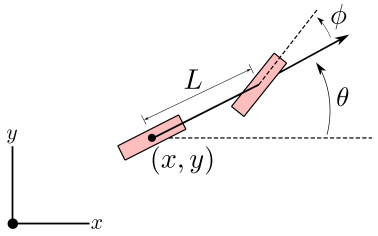


Figure 1.5: Simple model for an automobile. The state consists of the (x, y) position of the center of the rear axle and the heading angle θ. The control inputs are the steering angle ϕ and the forward velocity v.

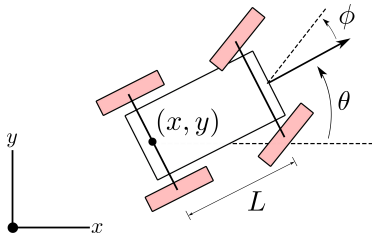


Figure 1.6: Simple model for an automobile. The state consists of the (x, y) position of the center of the rear axle and the heading angle θ. The control inputs are the steering angle ϕ and the forward velocity v.

1.4 Simulating Robot Dynamics

In Section 1.1 we introduced the concept of a *state space model* to mathematically describe how the state of a robot changes in time, and in Section 1.2 we showed how a robot's *kinematics and dynamics* are used to derive a state space model for physical motion. We will now introduce several computational techniques for *simulating* how the robot's state changes in time[21].

The state space model (1.1) is a general system of ordinary differential equations and in most cases it will be difficult or impossible to find an analytical solution. Numerical simulation is an approach to produce approximate solutions that can be extremely useful in practice for understanding the robot's dynamics more intuitively and for experimenting with and validating algorithms for robot autonomy. Typically when we say that we will *simulate* a system we mean that we will approximately solve an *initial value problem* for the system of differential equations from Equation (1.1):

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

where $h(\boldsymbol{x}(t), t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t))$ with the input $\boldsymbol{u}(t)$ being an fixed explicit function of time or with a closed-loop policy $\boldsymbol{u}(t) = \pi(\boldsymbol{x}(t), t)$. 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[22]. From the Fundamental Theorem of Calculus, we can write the solution to the initial value problem at some time t

[21] Note that the term *dynamics* is overloaded to mean both the physical motion models described by Newton's second law and more generally the temporal changes of a generic robot state.

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

as:

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

Evaluating this integral for an arbitrary time t is very challenging, so typically we use numerical integration approaches that involve a *discretization in time*:

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

where $t_0 < t_1 < \ldots < t_N$, and $t_N = t$. This discretization simplifies the problem into a sequence of integrals over small time steps $\Delta t_k = t_{k+1} - t_k$. There are many different numerical integration methods for ordinary differential equations, and they typically tradeoff between efficiency and accuracy. One of the simplest methods, the *Euler method*, integrates across a small time step using a first order Taylor series approximation at the start of the interval. More complex methods such as the *Runge-Kutta* methods evaluate the function at multiple points.

1.4.1 Euler method

The Euler method[23] for numerically integrating across a small time step Δt can be derived using a first-order Taylor series expansion:

$$\begin{aligned} \boldsymbol{x}(t + \Delta t) &\approx \boldsymbol{x}(t) + \Delta t \dot{\boldsymbol{x}}(t), \\ &= \boldsymbol{x}(t) + \Delta t h(\boldsymbol{x}(t), t). \end{aligned} \tag{1.27}$$

[23] Named after Leonard Euler (Swiss mathematician), and often referred to as *forward Euler* method.

Given a point $\boldsymbol{x}(t)$ along a trajectory, this method simply approximates the next step as if the current rate of change is constant across the time step. An alternative view of this method is that it is simply approximating the derivative $\dot{x}(t)$ using a finite difference approximation:

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

From the Taylor series analysis we can see that the error caused by one iteration of this method[24] is $\mathcal{O}(\Delta t^2)$.

[24] Referred to as the *local truncation error*.

1.4.2 Midpoint method

The Midpoint method can be derived by slightly modifying the Euler method. In the Euler method the derivative $\dot{\boldsymbol{x}}$ is evaluated at the start of the interval, and in the midpoint method we evaluate the derivative at the midpoint of the interval, $t + \frac{\Delta t}{2}$:

$$\dot{\boldsymbol{x}}(t + \frac{\Delta t}{2}) \approx \frac{\boldsymbol{x}(t + \Delta t) - \boldsymbol{x}(t)}{\Delta t}.$$

Therefore we compute the next state approximation as:

$$\boldsymbol{x}(t + \Delta t) \approx \boldsymbol{x}(t) + \Delta t h \left(\boldsymbol{x}(t + \frac{\Delta t}{2}), t + \frac{\Delta t}{2} \right). \tag{1.28}$$

Clearly this is not an *explicit* expression because it requires knowledge of the midpoint state $\boldsymbol{x}(t + \frac{\Delta t}{2})$. We can convert this into an explicit expression by again leveraging a first-order Taylor series expansion in an equivalent manner to the Euler method:

$$\boldsymbol{x}(t + \frac{\Delta t}{2}) \approx \boldsymbol{x}(t) + \frac{\Delta t}{2} h(\boldsymbol{x}(t), t). \tag{1.29}$$

Therefore we can compute the next state $\boldsymbol{x}(t + \Delta t)$ by first computing the midpoint using Euler's method in (1.29) and then using (1.28). The Midpoint method improves the local truncation error with respect to the Euler method from $\mathcal{O}(\Delta t^2)$ to $\mathcal{O}(\Delta t^3)$, at the cost of requiring a second evaluation of the derivative.

1.4.3 Runge-Kutta-4 method

Both the Euler and Midpoint methods operate by evaluating the derivative $\dot{\boldsymbol{x}}$ in the time step interval to approximate the change of $\boldsymbol{x}$ over the interval. The family of Runge-Kutta methods work in a similar fashion, but leverage even more evaluations of the derivative $\dot{x}$ for additional accuracy. One of the more common of the Runge-Kutta methods is the *fourth-order Runge-Kutta method*[25], which evaluates the derivative at four points in the Δt interval:

[25] The fourth-order Runge-Kutta method is 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:

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

The local truncation error of the fourth-order Runge-Kutta method is $\mathcal{O}(\Delta t^5)$, but again at the cost of requiring more evaluations of $h(\boldsymbol{x}(\tau), \tau)$ at each step.

1.4.4 Code Exercises

Code Exercise 1.4.1 (Closed-loop Dynamics of an Inverted Pendulum)**.**
In this exercise, you can try evaluating the closed-loop dynamics of an inverted pendulum. To do so, play around with notebook `inverted-pendulum`.

Code Exercise 1.4.2 (ODE integration "by hand")**.**
In this exercise you can try implementing an integration scheme "by hand". To do so, play around with notebook `hand-integration`.

Code Exercise 1.4.3 (Simulating a unicycle model)**.**

You can now try to set the control input of a unicycle robot to guide the robot towards a target point (x_d, y_d), and simulate the "closed-loop" system. You can do so using the notebook `unicycle-simu`. Here, the ODE integration is performed by `odeint` from `scipy.integrate`, which uses the RK45 scheme (i.e., Runge-Kutta with an adaptive size). One can plot different trajectories using `matplotlib`.

2
Open-Loop Motion Planning & Control

The previous chapter introduced the state space model as a mathematical formulation for modeling a robot's dynamics. State space models, typically expressed as differential equations, relate the temporal change of a state $\boldsymbol{x}(t) \in \mathbb{R}^n$ to the control inputs $\boldsymbol{u}(t) \in \mathbb{R}^p$ applied to the robot. We also saw how studying a robot's kinematics and dynamics can lead to the derivation of a state space model for a robot's physical motion. In this chapter we begin exploring how these dynamics models are leveraged for robot motion planning and control, where our goal is to determine a *control law* that specifies the inputs to apply to the robot to achieve a desired motion.

The ecosystem of techniques for robot control is vast and there are several ways to categorize different methods. The most fundamental categorization for a control law is if it is *open-loop* or *closed-loop*. Open-loop control laws do not rely on observations of the system to influence the choice of control input, while closed-loop control laws do. Mathematically, open-loop control laws are defined as functions that depend only on time and the initial condition of the system.

Definition 2.0.1 (Open-loop Control Law)**.** An open-loop control law is a function that maps time and the initial state of the system to a control input:

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

As a practical example, suppose you are standing in a room and wanted to walk to the other side and sit in a chair. If you were to plan a path to walk to the chair and then close your eyes and execute that plan, this would be considered an open-loop control method. Alternatively, a closed-loop control method would have you keep your eyes open the whole time while you move across the room.

Closed-loop control methods clearly have the capability to provide more robust performance since corrections can be made based on real-time observations. However, open-loop control methods can also have certain advantages, such as greater computational tractability, and are an important topic within the context of robotics. One popular open-loop problem found in robotics applications is open-loop *optimal control*, also commonly referred to as *trajectory*

optimization[1], which seeks to compute an *optimal* trajectory from one state to another along with the corresponding optimal control input sequence. Solving a single open-loop trajectory optimization problem is often significantly easier than solving for an optimal closed-loop control law that is defined for all states. Another common practice in robotics applications is to define a closed-loop controller by combining an open-loop trajectory with a closed-loop tracking controller, or by iteratively solving open-loop trajectory optimization problems[2]. These combinations of open and closed-loop techniques can balance computational tractability with robust and optimal performance.

In this chapter we introduce several common techniques for open-loop trajectory generation. First we will formally introduce a standard formulation of the optimal control problem and present some approaches for solving it. Then we will also introduce an alternative approach for generating open-loop trajectories for a class of robots whose dynamics have the special property of *differential flatness*. In the next chapter we will turn our attention to closed-loop control laws, including approaches that leverage open-loop techniques discussed in this chapter.

[1] D. E. Kirk. *Optimal Control Theory: An Introduction*. Dover Publications, 2004, R. M. Murray. *Optimization-Based Control*. California Institute of Technology, 2009

[2] These methods are typically referred to as *receding horizon control* or *model predictive control*.

2.1 Open-loop Optimal Control

Open-loop optimal control is a common technique for motion planning and control in robotics. The core objective in open-loop optimal control is to find an open-loop control policy from the current state of the robot, $\boldsymbol{x}(t_0)$, that will make the robot follow an optimal future trajectory. Formulating an optimal control problem requires two key components: a metric[3] to optimize and a state space model of the robot's dynamics constrains the set of feasible trajectories. Once the optimal control problem is *formulated*, we will also require an algorithm or analytical method for *solving* it.

[3] Generally referred to as a *cost function* in optimal control literature or a *reward function* in reinforcement learning literature.

2.1.1 Problem Formulation

The first step in formulating the optimal control problem is to define the performance metric that we want to optimize. This metric defines the quality of a sequence of states and control inputs, and we will refer to it as a *cost function*[4]. The standard form for defining the cost function in a *finite-horizon* optimal control problem is:

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

where $h(\boldsymbol{x}(t_f), t_f)$ is referred to as the *terminal cost* and where $g(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$ is referred to as a stage cost or running cost. In robotics, this cost function might quantify objectives such as "travel to a goal position as quickly as possible" or "move from one pose to another while using as little energy as possible".

An optimal control problem with the cost function (2.2) is referred to as a *finite-horizon* problem because the final time t_f is finite, and we do not con-

[4] If referring to the metric as a *reward function* the only difference in the formulation is that it will be a maximization problem rather than a minimization.

sider what happens after that time. In other contexts we might also choose to consider the *infinite-horizon* optimal control problem which considers the cost function:

$$J_\infty(\boldsymbol{x}(t), \boldsymbol{u}(t), t) = \int_{t_0}^{\infty} g(\boldsymbol{x}(t), \boldsymbol{u}(t), t)\mathrm{d}t, \tag{2.3}$$

which does not define a final time t_f and does not require defining a terminal cost[5]. In this chapter we will focus our attention on the finite-horizon problem since it is more commonly used for open-loop trajectory optimization due to better computational tractability under more general settings.

[5] The linear quadratic regulator (LQR) is a very well known infinite-horizon optimal closed-loop control method. The LQR problem requires $g(\boldsymbol{x}(t), \boldsymbol{u}(t), t$ to be quadratic in state and control.

The second step in formulating the optimal control problem is to define the state space model of the robot's dynamics that will constrain the optimization to only consider dynamically feasible trajectories. We have already discussed in **??** how to use a robot's kinematics and dynamics to derive state space models for robot motion, however the optimal control problem can be defined for other types of dynamics as well. Using the same notation from **??**, we consider a general state space model of the form:

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

A third, and optional, feature of optimal control problems are *state constraints* and *control constraints*. Often these constraints are expressed in the form:

$$\boldsymbol{x}(t) \in \mathcal{X}, \quad \boldsymbol{u}(t) \in \mathcal{U}, \quad \boldsymbol{x}(t_f) \in \mathcal{X}_f, \tag{2.4}$$

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

where $\mathcal{X}$ is the set of all admissible states, $\mathcal{U}$ is the set of all admissible control inputs, and $\mathcal{X}_f$ is a set of admissible terminal states. A common way to mathematically define the sets $\mathcal{X}$, $\mathcal{U}$, and $\mathcal{X}_f$ is by a set of inequalities on $\boldsymbol{x}$ and $\boldsymbol{u}$, respectively.

Finally, we can assemble the cost function, dynamics model, and constraints into the optimal control problem.

Definition 2.1.1 (Optimal Control Problem)**.** An *optimal control problem* seeks to find an admissible sequence of control inputs $\boldsymbol{u}(t)$ which drive the system to follow an admissible trajectory $\boldsymbol{x}(t)$ that minimizes the cost function $J(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$. This problem is formulated as the optimization problem:

$$\begin{aligned} \underset{\boldsymbol{u}(t)}{\text{minimize}} \; & J(\boldsymbol{x}(t), \boldsymbol{u}(t), t), \\ \text{s.t. } & \dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t), t), \\ & \boldsymbol{x}(t) \in \mathcal{X}, \quad \boldsymbol{u}(t) \in \mathcal{U}, \\ & \boldsymbol{x}(t_0) = \boldsymbol{x}_0, \\ & \boldsymbol{x}(t_f) \in \mathcal{X}_f, \end{aligned} \tag{2.5}$$

where t_0 is the initial time, t_f is either a fixed final time or an optimization variable, and $\boldsymbol{x}_0$ is the initial state.

The solution to the optimal control problem in Equation (2.5) is an admissible and optimal trajectory defined over the interval $t \in [t_0, t_f]$, and we denote the optimal trajectory using the notation $\boldsymbol{u}^*(t)$ and $\boldsymbol{x}^*(t)$.

Example 2.1.1 (Autonomous Racing Optimal Control)**.** Consider an autonomous robot race where the objective is to complete the fastest lap time of a known course. We can formulate this as a finite-horizon optimal control problem where we minimize the final time it takes to get to the goal position $(x_{\text{goal}}, y_{\text{goal}})$, subject to constraints that the robot must remain on the course at all times, where $\mathcal{X}_{\text{course}}$ is the set of all states that are on the course. If we model our vehicle using the simple kinematic car model from (**??**) our state is the position and heading $\boldsymbol{x} = [x, y, \theta]^\top$ and the control is defined as the speed and steering angle $\boldsymbol{u} = [v, \phi]^\top$. The optimal control problem can be then formulated as:

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

Since our cost function is not a function of the state or control, we can expect the optimal solution $\boldsymbol{x}^*(t)$, $\boldsymbol{u}^*(t)$ to ride the boundaries of the constraint sets, for example applying maximum throttle and steering commands, to minimize the time. This would likely not be a comfortable ride!

2.1.2 *Solving the Optimal Control Problem*

Once we have formulated the optimal control problem in Equation (2.5) the next step is to solve it. This can be challenging since Equation (2.5) describes an infinite-dimensional optimization problem[6]. In some special cases an analytical solution to the problem can be found, but more often we must transform the problem into a finite-dimensional problem that can be solved numerically. Algorithms for numerically solving optimal control problems are generally classified as either *direct* or *indirect* methods.

[6] The optimization is over an infinite-dimensional function (control trajectory $\boldsymbol{u}(t)$) and not a finite set of parameters.

Direct methods follow a "first discretize, then optimize" pattern. In the first step the optimal control problem (2.5) is converted into a finite-dimensional problem by discretizing the continuous functions $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$. For example this can be accomplished by sampling a finite number of time points t_i and optimizing over $\boldsymbol{x}(t_i)$ and $\boldsymbol{u}(t_i)$. The resulting finite-dimensional optimization problem is generally referred to as a *nonlinear program* (NLP), which can be solved with existing numerical algorithms[7].

[7] Example solvers for general nonlinear programs include IPOPT and SNOPT, and example software packages for formulating and solving optimal control problems using the direct method include DIDO, PROPT, and GPOPS.

Indirect methods flip the pattern of direct methods and follow a "first optimize, then discretize" approach. These methods first derive the necessary conditions of optimality for the infinite-dimensional problem (2.5), which are expressed

as a two-point boundary value problem. A two-point boundary value problem consists of a set of first-order ODEs with boundary conditions at two points[8]. The second step of an indirect method is to discretize and numerically solve the two-point boundary value problem.

[8] This is in contrast to an initial value problem, which has a single boundary condition and can easily be numerical integrated to find a solution.

Indirect methods are less commonly used in the field of robotics because the derivation of the necessary conditions of optimality can be quite challenging and must be done on a case by case basis for every new optimal control problem formulation. Determining the necessary conditions of optimality can also be particularly challenging when the optimal control problem features non-trivial state or control constraints, which are common in real-world applications. In contrast, direct methods offer much more flexibility and have been quite successful in practice. Both direct and indirect methods will be discussed in further detail in **??**.

2.2 *Trajectory Generation with Differentially Flat Dynamics Models*

Creating open-loop control laws by solving optimal control problems can be computationally challenging. Sometimes we may want to make a tradeoff between performance (i.e. optimality) and computational tractability by searching for "good" trajectories that may be sub-optimal. For the special class of *differentially flat* dynamics models, computing "good" trajectories can be done with relative simplicity compared to solving optimal control problems. Differentially flat models can be applied in several common robotics applications, and include simple car models and quadrotor models.

Example 2.2.1 (Differentially Flat Autonomous Vehicle Control)**.** Recall the motion planning task from Example 2.1.1 where the objective was to find an open-loop control sequence to drive a vehicle to through a course to a goal position in minimum time. If we drop the optimality requirement and instead just try to find *any trajectory* that follows the course and reaches a goal position we can leverage the differential flatness of the kinematic car model. Specifically, with the model (**??**):

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

it is sufficient to specify a differentiable trajectory for $x(t)$ and $y(t)$, and the heading and control inputs can be *analytically* determined! Therefore to solve the motion planning problem for the autonomous vehicle we just need to generate any differentiable trajectory that follows the course. Once we have a differentiable trajectory for $x(t)$ and $y(t)$ we can leverage the first two equations of the kinematic model to solve for the heading $\theta(t)$:

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

Next, we can use the heading trajectory to compute the speed control by leveraging either of the first two equations:

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

Finally, given θ and v we can directly solve for the steering angle from the last equation:

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

Mathematically we define the property of differential flatness by the existence of a special mapping between the state and control space and a space of *flat outputs*.

Definition 2.2.1 (Differential Flatness)**.** A non-linear system with state $\boldsymbol{x} \in \mathbb{R}^n$ and control $\boldsymbol{u} \in \mathbb{R}^p$:

$$\dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t)), \tag{2.6}$$

is *differentially flat* with flat output[9] $\boldsymbol{z} \in \mathbb{R}^p$ if there exists a function α such that:

$$\boldsymbol{z} = \alpha(\boldsymbol{x}, \boldsymbol{u}, \dot{\boldsymbol{u}}, \ldots, \boldsymbol{u}^{(p)}), \tag{2.7}$$

where $\boldsymbol{u}^{(i)}$ is the i-th time derivative of $\boldsymbol{u}$, and such that the trajectories $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ can be written as functions of the flat output $\boldsymbol{z}$ and a finite number of its derivatives:

$$\begin{aligned} \boldsymbol{x} &= \beta(\boldsymbol{z}, \dot{\boldsymbol{z}}, \ldots, \boldsymbol{z}^{(q)}) \\ \boldsymbol{u} &= \gamma(\boldsymbol{z}, \dot{\boldsymbol{z}}, \ldots, \boldsymbol{z}^{(q)}). \end{aligned} \tag{2.8}$$

[9] Note that the dimension of the flat output $\boldsymbol{z}$ is equal to the dimension of the control $\boldsymbol{u}$.

As we can see from this definition, for a differentially flat system we can write any feasible trajectory of the system as a function of the flat output $\boldsymbol{z}(t)$ and its time derivatives! In the context of motion planning and control this is extremely useful for trajectory design because the flat outputs can be specified and then *directly mapped* to the corresponding control inputs after.

Given a differentially flat dynamics model of the form (2.6), we can now explore different techniques for exploiting the differential flatness property for open-loop trajectory design. In the following sections we will explore various aspects to trajectory design in the flat output space, such as how to parameterize the trajectory, how to handle initial or terminal state constraints, and how to handle control constraints.

2.2.1 *Trajectory Parameterization*

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

$$z_j(t) = \sum_{i=1}^{N} \alpha_i^{[j]} \psi_i(t), \tag{2.9}$$

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

2.2.2 *Equality Constraints*

One of the most fundamental components of an open-loop motion planning problem is the initial state constraint, $\boldsymbol{x}(0) = \boldsymbol{x}_0$, and often we also wish to specify a terminal state, $\boldsymbol{x}(T) = \boldsymbol{x}_f$ at some final time T. We handle these initial and terminal conditions when planning in the flat output space by first mapping these conditions into constraints on the flat output $\boldsymbol{z}(0)$ and $\boldsymbol{z}(T)$ and their derivatives by noting that:

$$\begin{aligned} \boldsymbol{x}_0 &= \beta(\boldsymbol{z}(0), \dot{\boldsymbol{z}}(0), \ldots, \boldsymbol{z}^{(q)}(0)), \\ \boldsymbol{x}_f &= \beta(\boldsymbol{z}(T), \dot{\boldsymbol{z}}(T), \ldots, \boldsymbol{z}^{(q)}(T)). \end{aligned} \tag{2.10}$$

The resulting boundary conditions, for example requirements on $z_j(0)$, $\dot{z}_j(0)$, $\ldots$, $z_j^{(q)}(0)$ and $z_j(T)$, $\dot{z}_j(T)$, $\ldots$, $z_j^{(q)}(T)$, can be mapped quite easily to constraints on the coefficients $\alpha_i^{[j]}$ when using a series of smooth basis functions to parameterize the trajectory. Differentiating Equation (2.9) q times gives:

$$\begin{aligned} \dot{z}_j(t) &= \sum_{i=1}^{N} \alpha_i^{[j]} \dot{\psi}_i(t), \\ &\vdots \\ z_j^{(q)}(t) &= \sum_{i=1}^{N} \alpha_i^{[j]} \psi_i^{(q)}(t). \end{aligned} \tag{2.11}$$

Applying the flat output boundary conditions will then result in a set of linear equality constraints:

$$\begin{bmatrix} \psi_1(0) & \psi_2(0) & \ldots & \psi_N(0) \\ \dot{\psi}_1(0) & \dot{\psi}_2(0) & \ldots & \dot{\psi}_N(0) \\ \vdots & \vdots & & \vdots \\ \psi_1^{(q)}(0) & \psi_2^{(q)}(0) & \ldots & \psi_N^{(q)}(0) \\ \psi_1(T) & \psi_2(T) & \ldots & \psi_N(T) \\ \dot{\psi}_1(T) & \dot{\psi}_2(T) & \ldots & \dot{\psi}_N(T) \\ \vdots & \vdots & & \vdots \\ \psi_1^{(q)}(T) & \psi_2^{(q)}(T) & \ldots & \psi_N^{(q)}(T) \end{bmatrix} \begin{bmatrix} \alpha_1^{[j]} \\ \alpha_2^{[j]} \\ \vdots \\ \alpha_N^{[j]} \end{bmatrix} = \begin{bmatrix} z_j(0) \\ \dot{z}_j(0) \\ \vdots \\ z_j^{(q)}(0) \\ z_j(T) \\ \dot{z}_j(T) \\ \vdots \\ z_j^{(q)}(T) \end{bmatrix}. \tag{2.12}$$

We can see that this imposes some implicit requirements on the choices for the basis functions, since we would like to make sure that there is at least one

solution to this system of equations. In fact, if the matrix formed by the basis functions is full rank, then the initial and terminal conditions are sufficient for completely specifying the entire trajectory!

When using the smooth basis function parameterization in Equation (2.9) we can actually represent any equality constraint on the flat outputs or their derivatives in the linear equality form of Equation (2.12), not just constraints on the initial and terminal states. For example we may want to add a set of waypoints to a trajectory. If the number of constraints we add to Equation (2.12) is too large, such that the system is over-determined, there is likely not going to be a solution. To counter this issue we can simply choose a new set of basis functions that is sufficiently rich, although this will add complexity to the required computations.

2.2.3 *Inequality Constraints via Time Scaling*

Now that we have considered equality constraints on the flat output trajectory in Section 2.2.2 we can turn our attention to inequality constraints, which are also common in open-loop motion planning and control problems to address control limits or desired bounds on the state. For example, the simple car robot from Example 2.2.1 could have an upper bound on its speed:

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

One technique for handling these types of constraints when planning in the flat output space is to use *time scaling*. Time scaling works by first planning a trajectory to satisfy any equality constraints, such as by formulating and solving Equation (2.12), and then speeding up or slowing down parts of the resulting trajectory to satisfy the inequality constraints.

Specifically, we can start with a trajectory $\boldsymbol{x}(t)$ that satisfies our desired equality constraints and consider the *geometric path* of the trajectory, which is the sequence of states $\boldsymbol{x}$ of the trajectory but not associated with a particular time. Mathematically we can reason about the geometric path by reparameterizing the trajectory using a *path parameter*[10], $s(t)$:

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

where $s(0) = s_0$, $s(T) = s_f$, and $\dot{s}(t) > 0$[11]. Given a geometric path we can now create new temporal trajectories by simply varying the definition of the function $s(t)$, which is known as *time scaling* since we are essentially just redefining how we temporally move along the geometric path.

[10] An intuitive choice of the path parameter in motion planning problems is the arc-length traveled along a path.

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

Example 2.2.2 (Time Scaling for a Simple System)**.** Consider a scalar system with state $x \in \mathbb{R}$ and a geometric path that connects and initial and terminal state, x_0 and x_f. Let the geometric path be parameterized by the path parameter s as:

$$x(s) = x_0 + s(x_f - x_0),$$

for $s \in [0, 1]$. We can choose the function $s(t)$ to speed up or slow down the temporal motion along the path, for example we could simply choose to parameterize $s(t)$ with a cubic polynomial:

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

defined for the interval $t \in [0, T]$. Substituting this function into $x(s)$ yields one possible temporal trajectory $x(t)$:

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

We can then modulate the time that we reach the terminal state x_f by simply choosing smaller or larger values for T. This could be relevant for handling inequality constraints if we had a constraint on the maximum rate of change for x:

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

since from the equation above we can analytically compute:

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

and therefore the max velocity will occur at $t = \frac{T}{2}$. This can be converted into a constraint on T to ensure the inequality constraint is satisfied:

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

The simple case in Example 2.2.2 is not a state space model, and for state space models time scaling is more complex. Given a state space trajectory defined by $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ that satisfies the differential equation in (2.6) over the interval $t \in [0, T]$, we can convert this into a geometric path $\boldsymbol{x}(s)$ with control $\boldsymbol{u}(s)$ using a path parameter $s(t)$ defined over the interval $s \in [s_0, s_f]$ where $s(t)$ satisfies:

$$\frac{d\boldsymbol{x}(s)}{ds}\frac{ds(t)}{dt} = f(\boldsymbol{x}(s), \boldsymbol{u}(s)). \tag{2.13}$$

Time scaling for this system involves defining a new path parameter $\tilde{s}(t)$ over a new time interval $t \in [0, \tilde{T}]$, where $\tilde{s}(0) = s_0$ and $\tilde{s}(\tilde{T}) = s_f$[12]. It is important to note however that the new scaling must still satisfy the differential equation:

$$\frac{d\boldsymbol{x}(\tilde{s})}{d\tilde{s}}\frac{d\tilde{s}(t)}{dt} = f(\boldsymbol{x}(\tilde{s}), \boldsymbol{u}(\tilde{s})). \tag{2.14}$$

However, since the geometric path is already defined, the terms $\frac{d\boldsymbol{x}(\tilde{s})}{d\tilde{s}}$ and $\boldsymbol{x}(\tilde{s})$ are fixed. Therefore time scaling with a new path parameter $\tilde{s}(t)$ is only admissible if a new control $\tilde{\boldsymbol{u}}(\tilde{s})$ can also be found that satisfies the differential equation above. Luckily for some systems satisfying this requirement is easy with the appropriate choice of the path parameter.

[12] 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 2.2.3 (Time Scaling for the Simple Car Model)**.** Consider again the simple car model from **??**:

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

and suppose a candidate trajectory $\boldsymbol{x}_c(t)$ with control $\boldsymbol{u}_c(t)$ has been defined by leveraging the differential flatness of the model by setting up and solving Equation (2.12) and then mapping the flat outputs $\boldsymbol{z}_c(t)$ into the state and control space. For this model a good choice for the path parameter s is the arc-length, defined as:

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

such that $\dot{s}(t) = v(t)$. With this choice the geometric path function $\boldsymbol{x}_c(s)$ is defined over the interval $s \in [0, L_{\text{path}}]$ where L_{path} is the total length of the path. Rewriting the simple car dynamics model we have:

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

which any choice of the time scaling function $\tilde{s}(t)$ must satisfy[13]. By choosing the arc-length as the path parameter we have $\dot{\tilde{s}} = v(\tilde{s})$ and therefore these equations can be further simplified:

[13] The trivial choice of $\tilde{s}(t) = s(t)$ will automatically satisfy these equations with the candidate control inputs $\boldsymbol{u}_c(t)$

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

The first two equations are guaranteed to already be satisfied for any choice of $\tilde{s} \in [s_0, s_f]$ because the original candidate trajectory satisfies the dynamics. Additionally, the third equation is guaranteed to be satisfied by choosing $\phi(\tilde{s}) = \phi_c(\tilde{s})$ (i.e. using the same steering input as with the candidate trajectory). Therefore the dynamics equations are all satisfied *independently* of the choice of $\dot{\tilde{s}}$ and since $\dot{\tilde{s}} = v(\tilde{s})$ the speed input can be chosen arbitrarily while maintaining the same geometric path! This is extremely useful because it means that inequality constraints on the speed $|v(t)| \leq v_{\max}$ can be easily enforced.

Example 2.2.3 shows a relatively straightforward application of time scaling to a model derived from kinematic constraints. It turns out that there is a methodical approach for time scaling trajectories subject to any kinematic model of

the form:

$$\dot{\boldsymbol{x}}(t) = G(\boldsymbol{x}(t))\boldsymbol{u}(t). \tag{2.15}$$

From the chain rule we again have:

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

which we can then rewrite as:

$$\frac{d\boldsymbol{x}(s)}{ds} = G(\boldsymbol{x}(s))\boldsymbol{u}_g(s), \tag{2.16}$$

where $\boldsymbol{u}_g(s) = \frac{\boldsymbol{u}(t)}{\dot{s}}$[14]. The new control $\boldsymbol{u}_g(s)$ is referred to as the *geometric control* since it is defined only with respect to the path parameter s. From Equation (2.16) we can see that the geometric path $\boldsymbol{x}(s)$ for the kinematic model is fixed by the geometric control $\boldsymbol{u}_g(s)$. 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. Furthermore speeding up the trajectory is accomplished by simply scaling the geometric control by $\dot{s}(t)$ since $\boldsymbol{u}(t) = \dot{s}\boldsymbol{u}_g(s)$.

[14] Recall that the definition of the path parameter $s(t)$ requires $\dot{s}(t) > 0$.

To summarize, for kinematic models of the form (2.15) a trajectory $\boldsymbol{x}(t)$ with control $\boldsymbol{u}(t)$ defined over the interval $t \in [0, T]$ can be temporally scaled through the following procedure:

1. Choose the path parameter s (e.g. the arc-length) and compute $s(t)$ for the original trajectory $\boldsymbol{x}(t)$ as well as the interval $[s_0, s_f]$.
2. Re-parameterize the control $\boldsymbol{u}(t)$ as a function of s.
3. Compute the geometric control $\boldsymbol{u}_g(s) = \frac{\boldsymbol{u}(s(t))}{\dot{s}(t)}$ for each point $s \in [s_0, s_f]$.
4. Define a new path parameter function $\tilde{s}(t)$ over the interval $[0, \tilde{T}]$ with $\dot{\tilde{s}} > 0$ and where $\tilde{s}(0) = s_0$ and $\tilde{s}(\tilde{T}) = s_f$.
5. Compute the new control $\tilde{\boldsymbol{u}}(t) = \boldsymbol{u}_g(\tilde{s}(t))\dot{\tilde{s}}(t)$ for all $t \in [0, \tilde{T}]$.

Example 2.2.4 (Time Scaling for the Unicycle Model)**.** Consider the kinematic unicycle model:

$$\begin{aligned} \dot{x} &= v\cos\theta, \\ \dot{y} &= v\sin\theta, \\ \dot{\theta} &= \omega, \end{aligned} \tag{2.17}$$

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

Similar to the simple car model in Example 2.2.3 a convenient choice of path parameter for this system is the arc-length:

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

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

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

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

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

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

2.3 *Exercises*

here we need to make sure that the exercise is interactive. This was an assignment.

Code Exercise 2.3.1 (Trajectory Generation via Differential Flatness)**.**
In this exercise, you will use an extended unicycle model to practice generating dynamically feasible trajectories by levering the system's differential flatness property. You will also have the chance to use time scaling techniques to design trajectories that satisfy control constraints. Please refer to the files available at the repository: `https://github.com/PrinciplesofRobotAutonomy/AA274A_HW1`

3
Closed-Loop Motion Planning & Control

In the previous chapter 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. Mathematically, we 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 will introduce a few common approaches for designing closed-loop control laws. First, we will introduce approaches for closed-loop feedback control for *linear dynamical systems*, including the *linear quadratic regulator (LQR)* which is 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 will 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 will 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 (3.1) that will force the state $\boldsymbol{x}$ to converge to the origin from some arbitrary initial state. This can be accomplished[3] 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].

The linear feedback controller can also be used 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:

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

3.1.1 The Linear Quadratic Regulator (LQR)

Previously we discussed how a linear feedback controller of the form (3.3) can be used 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].

[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 $0 = A\boldsymbol{x}_d + B\boldsymbol{u}_d$.

[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 $K^*(t)$ is computed 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]:

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

Both the finite and infinite horizon LQR problems can also be formulated in discrete time and solved 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 has been developed over centuries and 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 these LQR results can also be applied 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.

We can also see the structure of the controller in the 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.

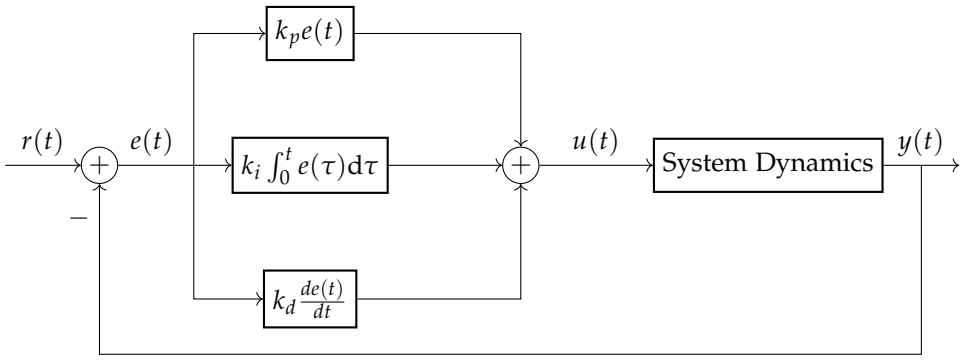


Figure 3.1: Block diagram for a PID controller in a feedback loop.

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:

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

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 simply choose the gains k_p and k_d to make the eigenvalues of the closed-loop dynamics matrix stable. The eigenvalues for this 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 would be to external disturbances. For example, in theory you 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.

[13] Note that a proportional controller alone ($k_d = 0$) 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.

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.

[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

Linear control techniques such as the linear quadratic regulator can be applied 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:

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

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

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 can be used 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

[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".

a set of states and the specific controller gains are modified depending on the region the system is currently operating in.

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 $\boldsymbol{x} \coloneqq \left[\theta, \dot{\theta}\right]^\top$ as:

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

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 now 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!

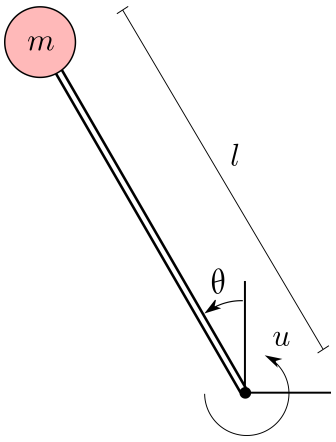


Figure 3.2: An inverted pendulum with a point mass m on a rigid rod of length l. The motion is described by the angle θ and u is the control torque about the axis of rotation.

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 significant 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, geometric control, and more.

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 functions that can be thought of as "energy" functions[19]. If a Lyapunov function can be shown to be decreasing along the "flow" of the dynamical system for a region around an equilibrium point, then this 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, a closed-loop controller can be derived from the CLF by choosing the control input that minimizes the gradient of the CLF.

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

[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, an open-loop method is used 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)**.** The general *trajectory tracking* controller[20] is expressed 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 (3.1), this control law can also be defined 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 **??**, and the feedback component, $\pi(\boldsymbol{x}(t), \boldsymbol{x}_d(t), t)$ can be designed using various approaches from Sections 3.1 and 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 $\delta\boldsymbol{x}(t) = \boldsymbol{x}(t) - \boldsymbol{x}_d(t)$ and $\delta\boldsymbol{u}(t) = \boldsymbol{u}(t) - \boldsymbol{u}_d(t)$. 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 (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 (i.e. the highest order of derivatives of the flat output that are needed to describe system dynamics), and $\boldsymbol{w}$ is a modified "virtual" control input term [22].

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

Since the system in (3.10) is linear we can leverage techniques from linear control theory in Section 3.1 to design a closed-loop feedback controller for computing the virtual control $\boldsymbol{w}$ which can be used to track an open-loop trajectory for the flat outputs. In particular, suppose a reference flat output trajectory $\boldsymbol{z}_d(t)$ and corresponding virtual input $\boldsymbol{w}_d(t)$ are computed by 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 (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. We can then 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 *Code Exercises*

Code Exercise 3.4.1 (Linearization and Automatic Differentiation)**.**
As we have previously discussed, we can control nonlinear systems by linearizing the dynamics and then applying tools from linear systems control

theory. The linearization procedure requires us to compute the Jacobian of the dynamics, which can be done by hand or by leveraging computational *auto-differentiation* libraries (e.g., JAX), which can automatically compute derivatives given an input function. For example, given $f(x) = \frac{1}{2}\|x\|_2^2$, we can use the JAX functionality `jax.grad` to get the function $\nabla f(x) = x$. Explore the notebook `autodiff` to compute the Jacobian for the unicycle model:

$$\frac{\partial f}{\partial \boldsymbol{\xi}} = \begin{bmatrix} 0 & 0 & -\bar{v}\sin\bar{\theta} \\ 0 & 0 & \bar{v}\cos\bar{\theta} \\ 0 & 0 & 0 \end{bmatrix},$$

and

$$\frac{\partial f}{\partial \boldsymbol{u}} = \begin{bmatrix} \cos\bar{\theta} & 0 \\ \sin\bar{\theta} & 0 \\ 0 & 1 \end{bmatrix},$$

Pratice calculating the Jacobian by hand and by using the JAX functionality `jax.jacobian`.

Code Exercise 3.4.2. Continuing previous exercises, you will implement the differential flatness-based trajectory tracking controller for the extended unicycle robot described in Example 3.3.2. `https://github.com/PrinciplesofRobotAutonomy/AA274A_HW1`.

4
Optimal Control and Trajectory Optimization

We previously presented the *optimal control*[1] problem formulation as a useful tool for open-loop robot motion planning and control in Chapter 2. In Definition 2.1.1 we presented the general form of the optimal control problem:

$$\begin{aligned} \underset{\boldsymbol{u}(t)}{\text{minimize}} \; & J(\boldsymbol{x}(t), \boldsymbol{u}(t), t), \\ \text{s.t. } & \dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t), t), \\ & \boldsymbol{x}(t) \in \mathcal{X}, \quad \boldsymbol{u}(t) \in \mathcal{U}, \\ & \boldsymbol{x}(t_0) = \boldsymbol{x}_0, \end{aligned} \tag{4.1}$$

where $\boldsymbol{x} \in \mathbb{R}^n$ is the robot state, $\boldsymbol{u} \in \mathbb{R}^m$ is the control input, $\boldsymbol{x}_0$ is the known initial condition, $f(\boldsymbol{x}, \boldsymbol{u}, t)$ is the robot's dynamics model, $\mathcal{X}$ is the set of all admissible states, $\mathcal{U}$ is the set of all admissible control inputs, t_0 is the initial time, and $J(\boldsymbol{x}, \boldsymbol{u}, t)$ is the cost functional[2]. The cost function can either represent a finite or infinite horizon cost, but for the purposes of this chapter we will focus on the finite horizon cost:

$$J(\boldsymbol{x}(t), \boldsymbol{u}(t), t) = h(\boldsymbol{x}(t_f), t_f) + \int_{t_0}^{t_f} g(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \mathrm{d}t, \tag{4.2}$$

where $h(\boldsymbol{x}(t_f), t_f)$ is the terminal cost, $g(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$ is the stage cost or running cost, and t_f is the final time. For simplicity, in this chapter we will consider the trivial choice for the admissible state and control sets: $\mathcal{X} = \mathbb{R}^n$ and $\mathcal{U} = \mathbb{R}^m$.

By solving the optimal control problem (4.1) we get the optimal trajectory $\boldsymbol{x}^*(t)$ and the corresponding optimal open-loop control $\boldsymbol{u}^*(t)$ which can be applied to the robot. Unfortunately, the optimal control problem (4.1) is particularly challenging to solve since it is *infinite-dimensional*[3]. Methods for solving (4.1) are categorized as either *direct* or *indirect*. Both types of methods generally require some form of discretization so that the problem can be solved numerically, but they differ in the manner in which the problem is discretized. *Indirect methods* follow a "first optimize, then discretize" approach. These methods first derive necessary conditions for optimality for the original infinite-dimensional problem and then recover a solution by discretizing these conditions. *Direct methods* follow a "first discretize, then optimize" approach. These methods first

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

[2] Called a cost *functional* because it is a function of functions, but often colloquially referred to as just a cost *function*.

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

discretize the original problem into a finite-dimensional problem called a *nonlinear program* which we then solve numerically. This chapter will provide an overview of both direct and indirect methods.

4.1 Indirect Methods

Indirect methods solve the optimal control problem in Equation (4.1) by first deriving *necessary optimality conditions*[4] (NOCs) and then by using a numerical procedure to find solutions that satisfy them, thereby "indirectly" solving the problem. As a brief example, for unconstrained finite-dimensional optimization problems the classic first-order necessary optimality condition is that the gradient of the function must be zero.

[4] We explicitly refer to the conditions as *necessary* conditions because they must be satisifed by the optimal solution, but they may not be *sufficient* to guarantee the solution is optimal. In other words, there may exist solutions that satisfy the *necessary* conditions but do not solve the original optimization problem.

Example 4.1.1 (First-order Necessary Optimality Condition)**.** For the finite-dimensional unconstrained optimization problem:

$$\underset{x \in \mathbb{R}}{\text{minimize}} \; x\,(x-1)\,(x+1)$$

the first-order necessary optimality condition is that $\nabla_x f = 0$. This function has two solutions that satisfy the necessary condition but only one is a local minimum, demonstrating that the necessary condition is not *sufficient*.

4.1.1 Constrained Finite-Dimensional Optimization

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

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

$$\begin{aligned} \underset{\boldsymbol{x} \in \mathbb{R}^n}{\text{minimize}} \; & f(\boldsymbol{x}), \\ \text{s.t. } & h_i(\boldsymbol{x}) = 0, \quad i = 1, \ldots, m, \end{aligned} \tag{4.3}$$

where the optimization variable is $\boldsymbol{x}$, $f(\boldsymbol{x})$ is the objective function, and the m functions $h_i(\boldsymbol{x})$ define the equality constraints.

We derive the necessary optimality conditions for the problem in Equation (4.3) by first defining the *Lagrangian* function $L(\boldsymbol{x}, \boldsymbol{\lambda})$, which augments the objective function with a weighted sum of the constraint functions:

$$L(\boldsymbol{x}, \boldsymbol{\lambda}) \coloneqq f(\boldsymbol{x}) + \sum_{i=1}^{m} \lambda_i h_i(\boldsymbol{x}), \tag{4.4}$$

where $\boldsymbol{\lambda} \in \mathbb{R}^m$ is a vector of *Lagrange multipliers*. The necessary optimality conditions for Equation (4.3) are defined from the Lagrangian as:

$$\begin{aligned} \nabla_{\boldsymbol{x}} L(\boldsymbol{x}^*, \boldsymbol{\lambda}^*) &= 0, \\ \nabla_{\boldsymbol{\lambda}} L(\boldsymbol{x}^*, \boldsymbol{\lambda}^*) &= 0, \end{aligned} \tag{4.5}$$

which are the gradients of the Lagrangian with respect to the variables $\boldsymbol{x}$ and the multipliers $\boldsymbol{\lambda}$. Note that the NOCs in (4.5) are a set of $n + m$ *algebraic* equations with $n + m$ unknowns. In contrast, we will see next that the NOCs for infinite-dimensional problems are not algebraic, but rather differential.

4.1.2 *Necessary Optimality Conditions*

Analogously to finite-dimensional optimization's Lagrangian, the first step to defining the necessary optimality conditions for the infinite-dimensional problem in Equation (4.1) is to define a function called the *Hamiltonian*:

$$H(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}(t), t) \coloneqq g(\boldsymbol{x}(t), \boldsymbol{u}(t), t) + \boldsymbol{p}^\top(t) f(\boldsymbol{x}(t), \boldsymbol{u}(t), t), \tag{4.6}$$

where $\boldsymbol{p}(t) \in \mathbb{R}^n$ is a multiplier referred to as a *costate*, $g(\boldsymbol{x}, \boldsymbol{u}, t)$ is the running cost, and $f(\boldsymbol{x}, \boldsymbol{u}, t)$ is the dynamics model. The NOCs are then given by the set of differential and algebraic equations:

$$\begin{aligned} \dot{\boldsymbol{x}}^*(t) &= \frac{\partial H}{\partial \boldsymbol{p}}(\boldsymbol{x}^*(t), \boldsymbol{u}^*(t), \boldsymbol{p}^*(t), t), \\ \dot{\boldsymbol{p}}^*(t) &= -\frac{\partial H}{\partial \boldsymbol{x}}(\boldsymbol{x}^*(t), \boldsymbol{u}^*(t), \boldsymbol{p}^*(t), t), \\ 0 &= \frac{\partial H}{\partial \boldsymbol{u}}(\boldsymbol{x}^*(t), \boldsymbol{u}^*(t), \boldsymbol{p}^*(t), t), \end{aligned} \tag{4.7}$$

which must be satisfied over the time interval $[t_0, t_f]$. These NOCs consist of $2n$ first order differential equations and m algebraic equations. Identifying unique solutions to the $2n$ differential equations requires $2n$ boundary conditions[6]. The initial condition $\boldsymbol{x}^*(t_0) = \boldsymbol{x}_0$ specifies n of these conditions, and the remaining conditions are given by:

$$\begin{aligned} &\left(\frac{\partial h}{\partial \boldsymbol{x}}(\boldsymbol{x}^*(t_f),\, t_f) - \boldsymbol{p}^*(t_f))\right)^\top \delta \boldsymbol{x}_f \\ &+ \left(H(\boldsymbol{x}^*(t_f),\, \boldsymbol{u}^*(t_f),\, \boldsymbol{p}^*(t_f),\, t_f) + \frac{\partial h}{\partial t}(\boldsymbol{x}^*(t_f),\, t_f)\right) \delta t_f = 0, \end{aligned} \tag{4.8}$$

where $h(\boldsymbol{x}(t_f), t_f)$ is the terminal cost and $\delta \boldsymbol{x}_f$ and δt_f are referred to as *variations*. If either the final time or final state is fixed in the optimal control problem the corresponding variation is set to be zero, which changes the boundary conditions in Equation (4.8). The resulting boundary conditions for the four possible scenarios are:

Fixed Final Time and Fixed Final State: If both t_f and $\boldsymbol{x}(t_f)$ are fixed, both variations δt_f and $\delta \boldsymbol{x}_f$ are set to zero. In this case the boundary conditions in Equation (4.8) are trivially satisfied, and the remaining boundary conditions on the NOCs in Equation (4.7) are given by:

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

[6] In problems where the final time is not fixed we require $2n + 1$ boundary conditions.

Fixed Final Time and Free Final State: If only t_f is fixed, then only the variation $\delta t_f = 0$. In this case the conditions in Equation (4.8) simplify and the boundary conditions for the NOCs in Equation (4.7) are given by:

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

Free Final Time and Fixed Final State: If only $\boldsymbol{x}_f$ is fixed, then only the variation $\delta \boldsymbol{x}_f = 0$. In this case the conditions in Equation (4.8) simplify and the boundary conditions for the NOCs in Equation (4.7) are given by:

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

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

Free Final Time and Free Final State: If neither t_f nor $\boldsymbol{x}(t_f)$ is fixed, then the boundary conditions for the NOCs in Equation (4.7) are given by:

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

Again, since the final time is free an additional boundary condition is added such that there are $2n + 1$ in total. Note that last two conditions are both extracted from Equation (4.8) because the variations $\delta \boldsymbol{x}_f$ and δt_f are independent.

4.1.3 Two-Point Boundary Value Problems

Finding solutions that satisfy the necessary optimality conditions in Equation (4.7) for the optimal control problem is challenging. Any solution must satisfy a set of $2n$ differential equations with boundary conditions specified at both t_0 and t_f. The problem of finding solutions to differential equations with boundary conditions specified at two points is called a *two-point boundary value problem*. Luckily, numerical procedures have been developed for solving these types of problems. For example the `scikits.bvp_solver` package in Python or the function `bvp4c` in Matlab implement schemes for solving these problems.

Most solvers for two-point boundary value problems typically assume the NOCs in Equation (4.7) and their boundary conditions are expressed in the standard form:

$$\dot{\boldsymbol{z}} = g(\boldsymbol{z},\, t), \quad l(\boldsymbol{z}(t_0),\, \boldsymbol{z}(t_f)) = 0. \tag{4.9}$$

However, some types of problems may not directly fit into this standard form. For these cases it is sometimes possible to convert a non-standard form problem into the standard form in Equation (4.9)[7].

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

In optimal control settings one common case where the two-point boundary value problem cannot be directly expressed in the standard form (4.9) is free final time problems, where t_f needs to be determined but does not have any associated dynamics. A useful trick in this context is to define a new variable $\tau = \frac{t}{t_f} \in [0, 1]$ to replace the time variable t. With this new variable the final time is now fixed to be $\tau_f = 1$. This variable substitution requires some additional changes to the NOCs:

1. Replace all time derivatives with derivatives with respect to τ, where from the chain rule we have $\frac{d(\cdot)}{d\tau} = t_f \frac{d(\cdot)}{dt}$.

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

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

We then add the "dummy" state r to the vector z and the NOCs are expressed in the standard form in Equation (4.9). In summary, this approach can be thought of as "tricking" the standard-form solver to think that the final time is 1 and that t_f is a state with trivial dynamics.

Example 4.1.2 (Free Final Time Optimal Control Problem)**.** Consider a double integrator system:

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

where $x \in \mathbb{R}$ is the state and $u \in \mathbb{R}$ is the control input. The control task 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 problem is a free final time problem with a fixed initial and final state, and the cost is formulated to find a trajectory that minimizes a combination of the time to reach the final state and the amount of control effort required to get there. The trade-off between minimizing final time and minimizing control effort is made by adjusting the weighting parameters α and β[8]. The cost functional's terminal and running cost components are:

[8] What would the optimal behavior be for $\alpha = 0$ or for $\beta = 0$?

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

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

$$\begin{aligned}\dot{x}_1 &= x_2,\\ \dot{x}_2 &= u,\end{aligned}$$

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

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

We can now construct the Hamiltonian:

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

where p_1 and p_2 are the costate variables. Next we construct the NOCs in Equation (4.7) by taking the partial derivatives of H with respect to p, x, and u:

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

Since this problem is a free final time and fixed final state problem, the boundary conditions for the NOCs are given by:

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

From the last NOC we can see that the optimal control u^* can be solved for in terms of the costate p_2^*:

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

which can then be substituted into the second NOC and the boundary conditions. At this point the resulting two-point boundary value problem can be expressed in the standard form in Equation (4.9) and solved numerically. However, it also turns out that this problem is simple enough to solve analytically as well. Integrating the differential equations for the costates p_1 and p_2 gives:

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

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

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

where C_3 and C_4 are additional constants. There are now five unknown quantities, C_1, C_2, C_3, C_4, and t_f, which can be determined by leveraging the five boundary conditions. From the condition $x_1^*(0) = 10$ and $x_2^*(0) = 0$ we can see that $C_3 = 0$ and $C_4 = 10$. We then use the remaining boundary conditions to analytically solve for the remaining constants, and in particular the final time is:

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

Interestingly we can see that as β approaches zero the cost functional penalizes the final time more and as expected the expression for t_f shows that t_f also approaches zero. Additionally, as α approaches zero the cost functional penalizes control inputs more heavily, and correspondingly we can see that in this case t_f approaches infinity. We can also see that the optimal control expression takes the form:

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

which shows that the control input is linear in time and its magnitude is inversely proportional to β.

4.2 *Direct Methods*

Unlike indirect methods, direct methods do not require a derivation of the necessary optimality conditions. Instead these methods directly discretize the original optimal control problem in Equation (4.1) to turn it into a finite-dimensional constrained optimization problem called a *nonlinear programming problem (NLP)*[9].

There are several approaches for discretizing the OCP. Perhaps the simplest approach is to just use a forward Euler time discretization, which we presented in Section 1.4 for integrating differential equations. Using the Euler approach with time step h_i the differential equations $\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u}, t)$ are discretized[10] as:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + h_i f(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i), \tag{4.10}$$

where $\boldsymbol{x}_i = \boldsymbol{x}(t_i)$, $\boldsymbol{u}_i = \boldsymbol{u}(t_i)$, and $h_i = t_{i+1} - t_i$. We can apply this time discretization to the optimal control problem in Equation (4.1) to partition the time interval $[t_0, t_f]$ into a finite set of N times $\{t_0, t_1, \ldots, t_N\}$ where $t_N = t_f$ and the time step between each is $h_i = t_{i+1} - t_i$. With this discretization the infinite-dimensional optimization problem becomes a finite-dimensional optimization over the state and control values $(\boldsymbol{x}_i, \boldsymbol{u}_i)$ for $i = 0, \ldots, N$.

[9] Converting the infinite-dimensional OCP into an NLP is often called *transcription*.

[10] The dynamics model $\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u}, t)$ is referred to as the *continuous time* model and the recursive formula $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + hf(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i)$ is referred to as the *discrete time* model.

Rewriting the original OCP in Equation (4.1) as a function of the discrete set of parameters t_i, $\boldsymbol{x}_i$, and $\boldsymbol{u}_i$ will also require modifications to both the constraints and the cost functional. First, the discrete time dynamics in Equation (4.10) are used to replace the dynamics constraint $\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u}, t)$ in the OCP. We then update the cost function by using a numerical approximation of the integral, such as by using one of the Newton-Cotes formulas. One simple approximation of the integral is:

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

We can now express the OCP in Equation (4.1) as the finite-dimensional nonlinear program:

$$\begin{aligned} \underset{\boldsymbol{u}_i, \boldsymbol{x}_i}{\text{minimize}}\ & h(\boldsymbol{x}_N,\ t_N) + \sum_{i=0}^{N-1} h_i g(\boldsymbol{x}_i,\ \boldsymbol{u}_i,\ t_i), \\ \text{s.t. } & \boldsymbol{x}_{i+1} = \boldsymbol{x}_i + h_i f(\boldsymbol{x}_i,\ \boldsymbol{u}_i,\ t_i), \quad i = 0, \ldots, N-1, \\ & \boldsymbol{x}_0 = \boldsymbol{x}(t_0). \end{aligned} \tag{4.11}$$

Other methods for transcribing the infinite-dimensional OCP into an NLP include shooting methods that only use the control values as optimization variables, Hermite-Simpson collocation, pseudospectral methods, and higher-order integration schemes like Runge-Kutta methods.

4.2.1 *Consistency of Time Discretization*

The finite-dimensional optimal control problem in Equation (4.11) is only an approximation of the original problem in Equation (4.1). However, we can justify that this approximation method is *consistent* with the original problem by analyzing the necessary optimality conditions for the NLP in Equation (4.11) and comparing them to the necessary optimality conditions for the original OCP in Equation (4.1).

In Section 4.1.1 we showed that the necessary optimality conditions for equality-constrained finite-dimensional optimization problems are derived from the Lagrangian. For the NLP in Equation (4.11) the Lagrangian is[11]:

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

and therefore the necessary optimality conditions are:

$$\begin{aligned} \nabla_{\boldsymbol{x}_i} L = h_i \frac{\partial g}{\partial \boldsymbol{x}}(\boldsymbol{x}_i, \boldsymbol{u}_i) + h_i \big(\frac{\partial f}{\partial \boldsymbol{x}}(\boldsymbol{x}_i, \boldsymbol{u}_i)\big)^\top \boldsymbol{\lambda}_i + (\boldsymbol{\lambda}_i - \boldsymbol{\lambda}_{i-1}) = 0, \quad & i = 1, \ldots, N-1 \\ \nabla_{\boldsymbol{x}_N} L = \frac{\partial h}{\partial \boldsymbol{x}}(\boldsymbol{x}_N) - \boldsymbol{\lambda}_{N-1} = 0, & \\ \nabla_{\boldsymbol{u}_i} L = h_i \frac{\partial g}{\partial \boldsymbol{u}}(\boldsymbol{x}_i, \boldsymbol{u}_i) + h_i \big(\frac{\partial f}{\partial \boldsymbol{u}}(\boldsymbol{x}_i, \boldsymbol{u}_i)\big)^\top \boldsymbol{\lambda}_i = 0, \quad & i = 0, \ldots, N-1 \\ \boldsymbol{x}_i + h_i f(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i) - \boldsymbol{x}_{i+1} = 0, \quad & i = 0, \ldots, N-1. \end{aligned} \tag{4.12}$$

[11] The initial condition constraint in Equation (4.11) can be ignored in the Lagrangian by not considering $\boldsymbol{x}_0$ as an optimization variable in the optimization problem, since it is a fixed value.

Now we can compare these conditions against the NOCs derived from the indirect method for a problem with fixed final time and free final state. From the the Hamiltonian defined in Equation (4.6), the corresponding optimality conditions in Equation (4.7), and boundary conditions in Equation (4.8), the NOCs for the infinite-dimensional OCP are:

$$\begin{aligned}
\frac{\partial g}{\partial \boldsymbol{x}}(\boldsymbol{x}(t),\boldsymbol{u}(t)) + \left(\frac{\partial f}{\partial \boldsymbol{x}}(\boldsymbol{x}(t),\boldsymbol{u}(t))\right)^\top \boldsymbol{p}(t) + \dot{\boldsymbol{p}}(t) &= 0, \quad t \in [t_0, t_f] \\
\frac{\partial h}{\partial \boldsymbol{x}}(\boldsymbol{x}(t_f)) - \boldsymbol{p}(t_f) &= 0, \\
\frac{\partial g}{\partial \boldsymbol{u}}(\boldsymbol{x}(t),\boldsymbol{u}(t)) + \left(\frac{\partial f}{\partial \boldsymbol{u}}(\boldsymbol{x}(t),\boldsymbol{u}(t))\right)^\top \boldsymbol{p}(t) &= 0, \quad t \in [t_0, t_f] \\
\dot{\boldsymbol{x}}(t) - f(\boldsymbol{x}(t),\boldsymbol{u}(t),t) &= 0, \quad t \in [t_0, t_f] \\
\boldsymbol{x}_0 - \boldsymbol{x}(t_0) &= 0.
\end{aligned} \tag{4.13}$$

The NOCs in Equation (4.12) for the discretized NLP and the NOCs for the original infinite-dimensional OCP in Equation (4.13) are remarkably similar. In fact, the NOCs in Equation (4.12) for the discretized NLP can be seen as simply the time-discretized versions of the infinite-dimensional NOCs in Equation (4.13). For example if we also use a forward Euler discretization of the NOCs in Equation (4.13) with:

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

we can see their equivalence. Therefore, as the time step h_i approaches zero the NOCs for the discretized NLP from the direct method converge to the NOCs derived for the original infinite-dimensional OCP using the indirect method!

Code Exercise 4.2.1 (Gradient descent)**.** In this exercise, you will implement for the first time gradient descent. To do so, please have a look at the notebook `gradient-descent`.

Code Exercise 4.2.2 (Direct and Indirect Methods for Unicycles)**.** In this exercise, you will continue previous exercises to compute a dynamically feasible and optimal trajectory for a unicycle robot leveraging both a direct and an indirect method. `https://github.com/PrinciplesofRobotAutonomy/AA274A_HW1`

Code Exercise 4.2.3 (Direct and Indirect Methods for Quadrotors)**.** In this exercise, you will compute a dynamically feasible and optimal trajectory for a quadrotor leveraging both a direct and an indirect method. To do so, please have a look at the notebook `quadrotor`.

5
Search-Based Motion Planning

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

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

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

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

Search-Based Motion Planning

Recall the general definition of the motion planning problem:

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

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

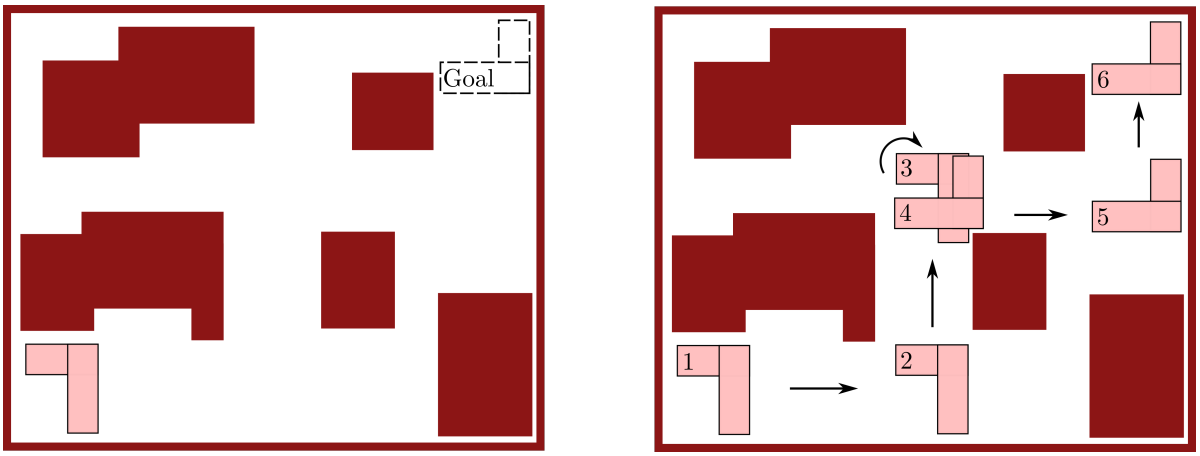


Figure 5.1: Motivating example: motion planning in a 2D workspace with obstacles.

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

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

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

Let's look at a motivating example.

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

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

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

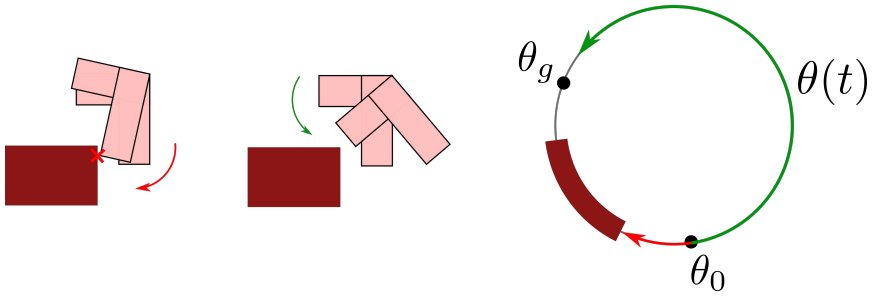


Figure 5.2: Example trajectory planning where the description of the configuration space using the manifold $\mathcal{S}^1$ is crucial to path planning. In particular, rotating clockwise leads to collision but rotating counterclockwise is a feasible path.

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

Properties of search algorithms It will be important to keep in mind the following properties:

- *Soundness*: An algorithm is *sound* if it returns valid solutions (or returns no solution);
- *Completeness*: An algorithm is *complete* if it terminates in finite time, returning a valid solution if one exists, and returning failure otherwise;
- *Time complexity*: This is the number of steps before termination, as a function of the input size (number of vertices, edges, branching factor, and other properties of the decision tree);
- *Space complexity*: This is the maximum number of memory locations, as a function of the input size.

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

5.1 Grid-based Motion Planners

Suppose the robot's configuration $\boldsymbol{q}$ is a d dimensional vector, then the C-space is a subset of $\mathbb{R}^d$. Critically, this is a continuous space and therefore there are an infinite number of potential configurations the robot could be in. To simplify

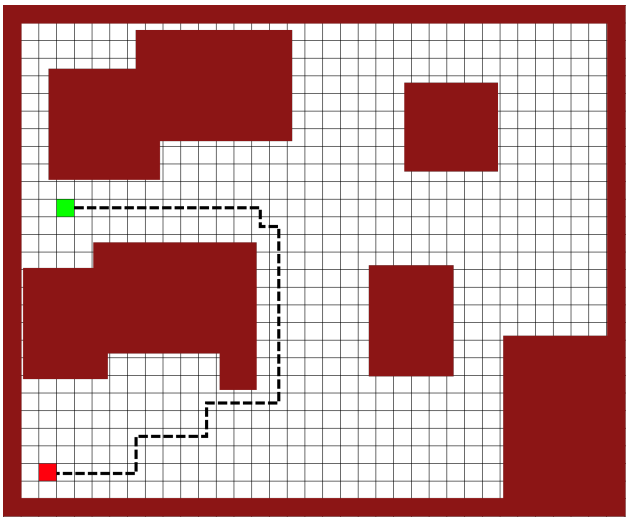


Figure 5.3: Discretizing the configuration space using a grid.

this problem, grid-based motion planners use a grid to discretize the C-space into a finite number of allowable configurations. For example, in a simple C-space in two dimensions the grid might look like that shown in Figure 5.3. In grid-based planners, undesirable configurations are simply represented by identifying some cells of the grid to be forbidden (e.g., for obstacle avoidance). The dynamics/kinematics of the robot are also abstracted away and it is assumed that the robot has the ability to move freely between adjacent cells (configurations). After such discretization, the resulting motion planning problem is sometimes referred to as a *discrete planning* problem because only a finite number of options are available at each step, and only a finite number of configurations are possible. The planning problem then reduces to finding a way to traverse through the cells from the initial configuration to a desired final configuration.

Mathematically, problems of this type are commonly expressed using discrete *graphs*. A graph $G = (V, E)$, is simply defined by a set of vertices V and a set of edges E. In the context of grid-based motion planners, each vertex $v \in V$ represents a *free* cell of the grid, and each edge $(v, u) \in E$ corresponds to a connection between adjacent cells. With the graph representation, the planning problem is to find a way to traverse through the graph to reach the desired vertex. Algorithms for solving such problems are referred to as *graph search methods*.

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

5.1.1 Label Correcting Algorithms

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

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

Definition 5.1.1 (Cost-of-Arrival)**.** The *cost-of-arrival* associated with a vertex q with respect to a starting vertex q_I is the cost associated with taking the best known route from q_I to q along edges of the graph, and is denoted $C(q)$.

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

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

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

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

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

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

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

Depth-First Search Depth-first search in a directed graph expands each node up to the deepest level of the graph, until a chosen node has no more successors. Another way to think about this in terms of the set Q is "last in/first out", where whenever a new vertex q is selected from Q it chooses those vertices that were most recently added. This algorithm is *sound* (i.e., if it returns a path, it is a valid path from the start to the goal). It is *not complete* on infinite graphs (i.e., if it starts in the wrong direction, it will not converge).

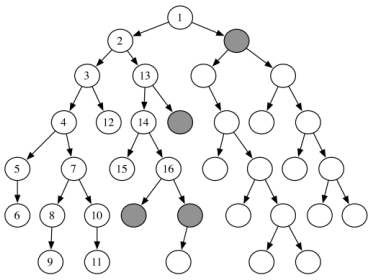


Figure 5.4: Depth-First Search.

Example 5.1.1. Let's see depth-first search at work. The general principle is that nodes to explore are added at the *front* of the queue. In particular, we report in a table 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 algorithm will have converged to the path (s,a,d,g).

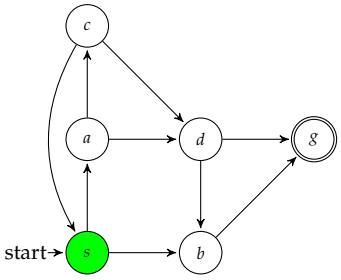


Figure 5.5: Graph for depth-first and breadth-first examples, including the start node s and the goal node g.

Breadth-First Search Breadth-first search begins with the start node and explores all of its neighboring nodes. Then for each of these nodes, it explores all their unexplored neighbors and so on. In terms of Q, this is like storing the frontier nodes as a queue with the first node added is the first node selected. This algorithm is *sound* (i.e., if it returns a path, it is a valid path from the start to the goal). It is *complete* on finite or countable infinite transition systems.

Example 5.1.2. Let's see breadth-first at work. The general principle is that nodes to explore are added at the *back* of the queue. Similarly to Example 5.1.1, referring to Figure 5.5, one has:

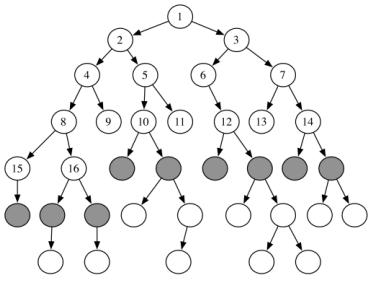


Figure 5.6: Breadth-First Search.

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)	$\{s, a, b, c, d, g\}$

The algorithm will have converged to the path (s, b, g).

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

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

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

5.1.2 A* Algorithm

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

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

In practice, the cost-to-go is not usually known, and therefore *heuristics* are used to provide approximate cost-to-go values $h(q)$. In order for the heuristic to be useful, it must be a positive *underestimate* of the true cost-to-go. An example of a heuristic h is to simply use 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 cost-of-arrival $C(q)$ plus an approximate cost-to-go $h(q)$. This provides a better estimate of the total quality of a path than just using the cost-of-arrival alone. The A* algorithm is defined in Algorithm 5.1. Note that in the case that the heuristic is chosen to be $h(q) = 0$ for all q then A* is the same as Djikstra's algorithm.

Algorithm 5.1: A* Algorithm

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

Example 5.1.3. Let's see A* at work on a practical example. Consider the graph in Figure 5.7, listing the heuristics and the actual costs on the edges.

We will fill a table (Table 5.1) considering the stage cost, the cost to go, and the total cost, by monitoring the paths being considered.

We start from the starting point (i.e., considering path (A)), which has $C = 0$, $h = 10$, and therefore total cost $f = 10$. Further, we expand to the next nodes and consider paths (A, F) and (A, B). The former has $C = 3$ (from the edge connecting A and F), $h = 6$, and therefore $f = 9$. Similarly, the latter has $C = 6$, $h = 8$, and $f = 14$. Given the lower cost, we proceed by expanding (A, F) into (A, F, G) and (A, F, H), which end up having $f = 9$ and $f = 13$, respectively. We therefore expand (A, F, G) into (A, F, G, I), with $f = 8$, and continue with the next expansion, which leads to considering (A, F, G, I, J), (A, F, G, I, H), and (A, F, G, I, E), with total costs 10, 12, and 15. At this point, the algorithm has converged. A good exercise for the reader is to now substitute the cost estimate for nodes B and D, with 2 and 1, respectively. By running the same algorithm, you will realize that it takes more steps to converge. This is due to the "worse" heuristic, and highlights the importance of choosing it right from the beginning.

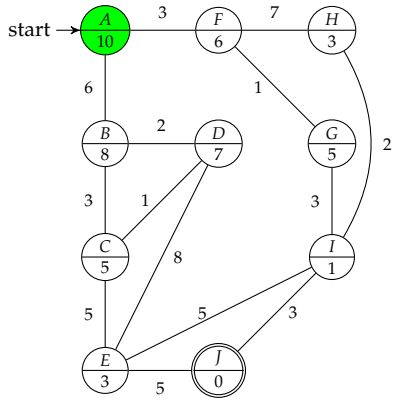


Figure 5.7: Graph for the A* example. The number at the nodes represent the heuristic costs to reach the goal.

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 5.1: Table summarizing the costs for the A* example.

5.2 *Combinatorial Motion Planning*

Combinatorial approaches to motion planning find paths through the continuous configuration space without resorting to discretizations, as it was the case in grid-based planners. Recall that in grid-based planners, cells in the discretized configuration space that were undesirable were blocked out and simply not considered in the resulting path search. However, in the case of combinatorial planners the structure of the free portion of the configuration space is considered in a different way.

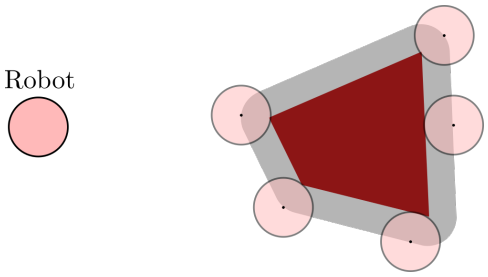


Figure 5.8: Free (white) and forbidden spaces (grey and red) of the configuration space for a simple circular robot in a 2D world. Note that the forbidden space accounts for the physical dimensions of the robot.

First, the subset of the configuration space C that is free (i.e., results in no collisions) is denoted as C_{free} and is called the *free space* (see Figure 5.8 and Figure 5.9). Combinatorial motion planning approaches operate by computing *roadmaps* through the free space C_{free}. A roadmap is a graph G where each vertex represents a point in C_{free} and each edge represents a path through C_{free} that connects a pair of vertices. The set S is then defined for a particular roadmap graph G as the set of all points in C_{free} that are either vertices of G or lie on any edge of G. This graph structure is similar to that used in grid-based planners, with the important distinction that the vertices can potentially be *any* configuration $q \in C_{\text{free}}$ while in grid-based planners the vertices are defined ahead of time by discretization. This distinction is very important because the flexibility of choosing the vertices does not result in any loss of information! Once the roadmap has been defined, a path can be defined by first connecting the initial configuration q_I and goal configuration q_G to the roadmap and then solving a

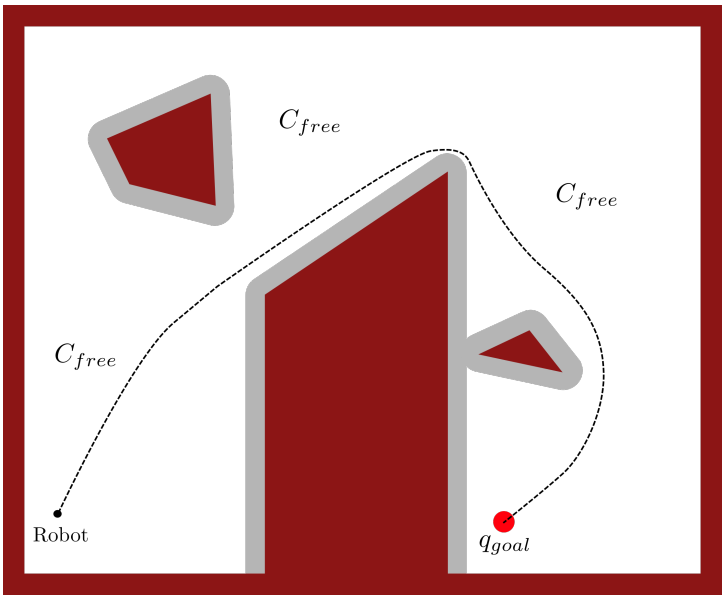


Figure 5.9: Once the free (white) and forbidden (grey and red) configurations have been identified, the physical dimensions of the robot can be ignored. This figure shows an example of a path planning problem in C-space with obstacles.

discrete graph search over the roadmap graph G.

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

5.2.1 Cell Decomposition

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

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

Example 5.2.1 (2D Cell Decomposition)**.** Consider a two-dimensional configuration space as shown in Figure 5.10. This space is decomposed into cells that are either lines or trapezoids by a process called vertical cell decomposition. Once the cells have been defined, the roadmap is generated by placing a vertex in each cell (e.g., at the centroid) as well as a vertex on each shared edge between cells.

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

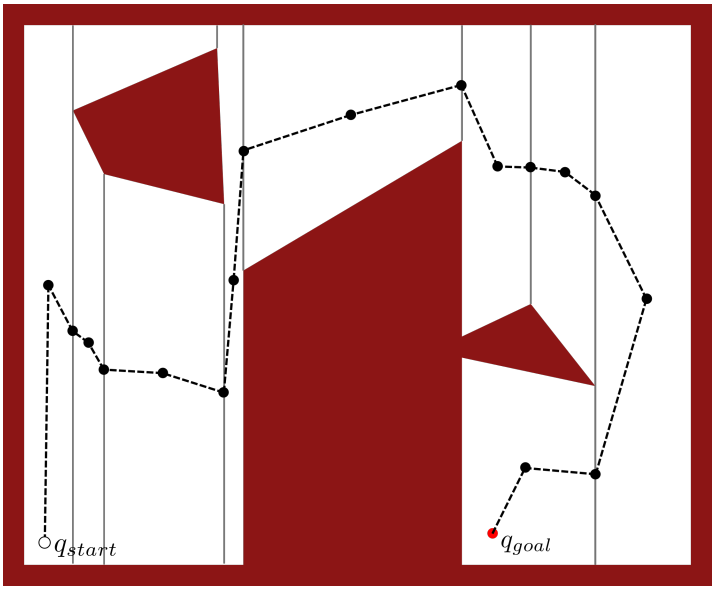


Figure 5.10: Example of 2D Cell Decomposition with C_{free} colored white. A roadmap is defined as the graph G with vertices shown as black dots and edges connecting them. To solve a planning problem with q_{start} and q_{goal} these points are first connected to the roadmap, and then the path is easily defined.

5.2.2 *Other Roadmaps*

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

5.3 *Exercises*

repository is unaccessible

Code Exercise 5.3.1 (A* Motion Planning)**.** In this exercise, you will you will implement the A* grid-based motion planning algorithm for some simple 2D environments. `https://github.com/PrinciplesofRobotAutonomy/AA274A_HW2`

6
Sampling-Based Motion Planning

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

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

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

Sampling-Based Motion Planning

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

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

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

6.1 Probabilistic Roadmap (PRM)

It is easiest to start the discussion with the probabilistic roadmap (PRM) algoruthm, as it closely relates to the combinatorial planners discussed previously. In particular, the PRM algorithm constructs a topological graph G, known as a *roadmap*. This graph includes vertices, which represent 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, a motion plan for a specific initial configuration q_I and goal configuration q_G can be determined by first linking them to the roadmap and subsequently employing a graph-search method (e.g., A*) to navigate a path through the roadmap graph. The primary distinction between PRM and combinatorial planners is found in the method of generating the roadmap.

The fundamental insight behind the PRM algorithm is that it sidesteps the need for a complete analysis of the free configuration space – a process that is typically resource-intensive. Instead, it randomly samples configurations q and utilizes a collision checker to determine if q belongs to C_{free}. The general outline of the algorithm is as follows:

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

An example of the roadmap resulting from applying this algorithm is shown in Figure 6.1. Note that using the *connectivity radius r* is a simple and efficient way of connecting the sampled vertices without having a burdensome number of edges. This is desirable because having too many edges is unnecessary, will

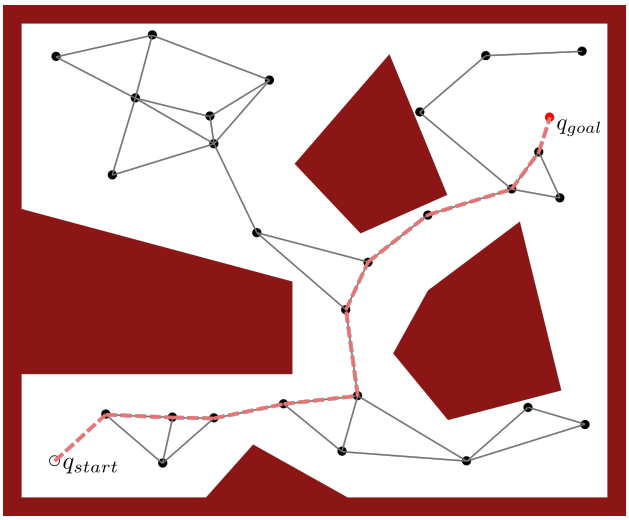


Figure 6.1: Example solution found via the PRM algorithm. The black dots represent the randomly sampled vertices of the graph, and the grey lines represent the edges created between vertices within a predefined radius r of each other. The initial configuration q_{start} and goal configuration q_{goal},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

make the graph-search more challenging, and will require more collision checks to be made[2]. On the flip side, making the radius r too small could mean not enough connections are made.

One of the limitations of PRM is that finding satisfactory solutions often requires a large number of samples n, to adequately span the configuration space. Similarly, an excess of samples leads to frequent use of the collision checker, which can be an expensive operation. However, in certain cases, such as in *multi-query* planning problems, thoroughly covering the space C_{free} with a roadmap proves advantageous. Multi-query problems involve the repeated use of the motion planner for various pairs of initial q_I and goal q_G configurations. In these scenarios, the PRM graph is constructed once to encompass C_{free} and can then be utilized repeatedly as required. Essentially, the initial resource-intensive sampling and collision checking are compensated by the long-term usability of the roadmap. This approach is efficient, provided that the environmental conditions do not change between queries. If the environment changes, the entire PRM roadmap would have to be rebuilt from scratch!

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

6.2 *Rapidly-exploring Random Trees (RRT)*

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

uration q_I, until the goal configuration q_G is reached. Additionally, the graph is built as a *tree*, which is a special type of graph that has only one path between any two vertices in the graph.

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

[3] The tree is a graph, however since it has special structure it is denoted 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.

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

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

6.3 *Theoretical Results for PRM and RRT*

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

[4] With a constant connectivity radius r.

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

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

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

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

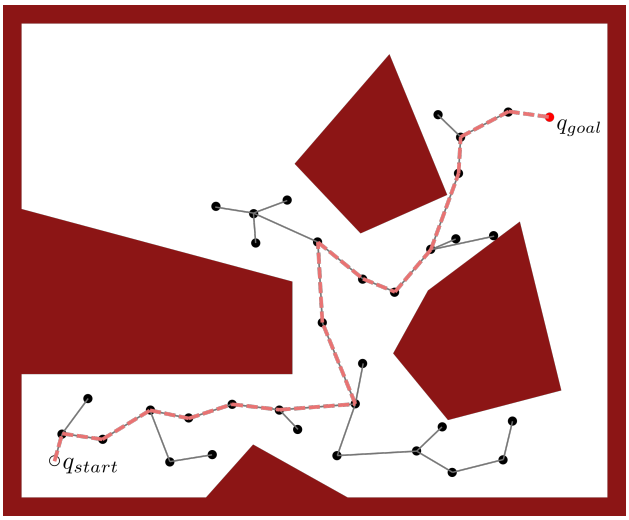


Figure 6.2: Example exploration tree by the RRT algorithm. The black dots represent points sampled at each iteration of the algorithm, which are connected to the nearest vertex that is currently part of the tree.

6.4 *Fast Marching Tree Algorithm (FMT*)*

As previously mentioned, PRM is asymptotically optimal, implying that it can find high-quality paths given sufficient samples. However, using a high number of samples in PRM leads to an increased number of collision checks, making it computationally expensive. In contrast, RRT is quick but typically fails to produce optimal paths.

The Fast Marching Tree (FMT*) algorithm, an advanced sampling-based motion planning technique, combines the benefits of both: it is both rapid and asymptotically optimal.[9]. FMT* builds a tree structured graph in the same way RRT does (which maintains the efficiency of RRT), but makes connections in a way that allows for asymptotic optimality. In particular, the technique used for making new connections is referred to as *dynamic programming*. Dynamic programming can be used to find the best paths with respect to a *cost-of-arrival*, denoted $c(q)$, which represents the cost to move from the initial configuration q_I to the configuration q. An example of a common metric is simply the Euclidean distance, which would result in a "shortest" path. In the context of motion planning, dynamic programming leverages Bellman's *principle of optimality*, which states that the optimal paths satisfy:

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

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

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

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

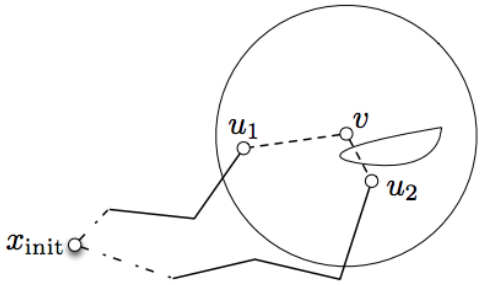


Figure 6.3: Example of a step in FMT*. Suppose the sample v has been selected to be the next point to be added to the tree. The candidate costs $\text{Cost}(u_1, v) + c(u_1)$ and $\text{Cost}(u_2, v) + c(u_2)$ are evaluated to see which connection would minimize c_v. Suppose u_2 was selected by this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

6.5 *Kinodynamic Planning*

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

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

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

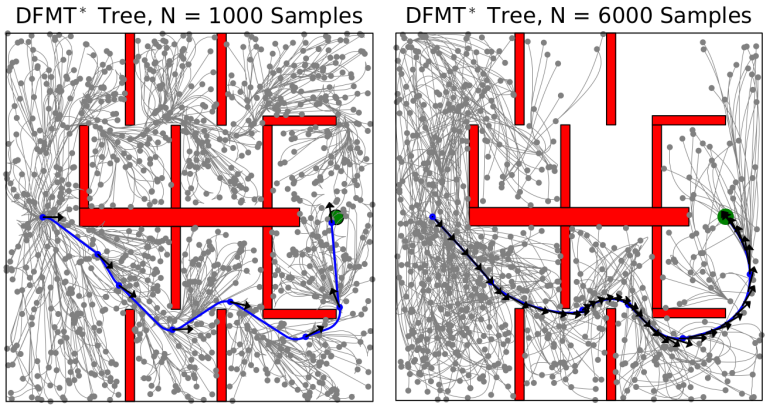
where $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{u} \in \mathbb{R}^m$. Note that the state space X is not necessarily the same as the configuration space C, but the configuration q is derivable from the state x. As previously mentioned, the configuration space is something that can be chosen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

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

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

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

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



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

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

6.6 *Deterministic Sampling-Based Motion Planning*

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

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

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

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

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

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

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

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

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

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

Code Exercise 6.6.2 (Motion planning in 2D configuration space)**.** In this code exercise, you will be able to implement and visualize several motion planning algorithms in a 2D configuration space, including RRT, RRT*, PRM*, and FMT*. For each algorithm, the goal is to find a feasible path from a starting point to a goal region, while avoiding obstacles. The configuration space is represented as a 2D grid, where the starting point, goal region, and obstacles are defined. You can play around with the notebooks at `motion-planning-2D`.

6.7 Exercises

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

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

6.7.1 Rapidly-Exploring Random Trees

it's unclear where the source of these problems resides?

Complete *Problem 2: Rapidly-Exploring Random Trees (RRT)*, where you will implement the RRT sample-based motion planning algorithm to plan paths in simple 2D environments. Additionally, in this problem you will start with a

simple geometric planner that does not consider robot dynamics, but will then extend the RRT algorithm to consider a wheeled robot modeled with Dubins car dynamics.

6.7.2 Motion Planning & Control

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

6.7.3 Bi-Directional Sampling-based Motion Planning

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

Part II

Robot Perception

7
Introduction to Robot Sensors

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

Introduction to Robot Sensors

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

[1] Global Navigation Satellite System

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

7.1 Sensor Classifications

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

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

Definition 7.1.2 (Exteroceptive)**.** Exteroceptive sensors acquire information from the robot's environment. For instance, they could measure distance, light intensity, and sound amplitude.

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

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

Definition 7.1.3 (Passive Sensor)**.** Passive sensors measure ambient environmental energy entering the sensor. Among these, we find thermometers and cameras.

Definition 7.1.4 (Active Sensor)**.** Active sensors emit energy into the environment and measure the reaction. Among these, we find ultrasonic sensors and laser rangefinders.

Classifying a sensor as active or passive is crucial, as it presents unique challenges. Passive sensors, for instance, are heavily influenced by environmental conditions. For example, a camera's ability to capture quality images relies on the ambient lighting.

7.2 Sensor Performance

Different types sensors also exhibit varying performance attributes. While some maintain exceptional accuragy in controlled laboratory settings, they may overcome with errors under natural real-world environmental conditions. Conversely, other sensors offer narrow, high-precision data across a variety of settings. In order to quantify and compare such performance characteristics it is necessary to define relevant metrics. Such metrics are generally either related to *design specifications* or *in situ* performance metrics (i.e., how well a sensor performs in the real environment).

7.2.1 Design Specification Metrics

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

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

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

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

2. *Resolution* is the minimum difference between two values that can be detected by a sensor. Usually, the lower limit of the dynamic range of a sensor

is equal to its resolution. However, this is not necessarily the case for digital sensors.

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

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

7.2.2 *In Situ Performance Metrics*

Metrics related to the design specifications can be reasonably quantified in a laboratory environment and then extrapolated to predict performance during real-world deployment. However, 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 error is defined as $e := m - v$.

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

4. *Precision* defines the reproducibility of the sensor results. For example, a sensor has high precision if multiple measurements of the same environmental

quantity are similar. It is important to note that precision is not the same as accuracy, a highly precise sensor can still be highly inaccurate.

7.2.3 Sensor Errors

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

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

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

To reliably employ a sensor in practice it is beneficial to have a characterization of the systematic and random errors, which could allow for corrections to make the sensor more accurate and provide information about its precision. The process of quantifying sensor errors and identifying their origins is referred to as *error analysis*. This analysis often entails identifying all sources of systematic errors, modeling random errors (e.g., using Gaussian distributions), and assessing the cumulative effect of such errors on the sensor's output.

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! A more concrete example involve active ranging sensors, which may fail in particular scenarios dictated by the sensor's relative positioning to environmental targets. For instance, a sonar sensor may return erroneus range readings duo to specular reflections when aimed at certain angles against a smooth wall. Such errors could seem random as they occur unpredictably with the robot's movement, yet become systematic if the robot remains stationary at an angle that consistently induces specular reflections. Thus, while sensor errors can be distinctly classified as systematic or random in controlled environments, accurately characterizing these errors becomes substantially more complex in real-world settings due to the vast array and variability of potential error sources.

7.2.4 Modeling Uncertainty

If we could perfectly model and understand all systematic errors in sensor measurements, in theory, these errors could be corrected. However, in practice, this is often not feasible, leading to the necessity for another way to represent sensor errors. In particular, characterizing uncertainty due to random errors is typically accomplished by using *probability distributions*.

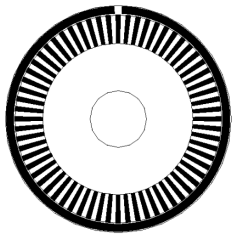
Given the practical impossibility of identifying all sources of random error, assumptions are commonly made regarding the error distribution of a sensor. It is commonly assumed that random errors have a zero-mean and are symmetric, or, more specifically, that they follow a Gaussian distribution. Generally, the assumption is that the error distribution is *unimodal*, simplifying the mathematical analysis involved.

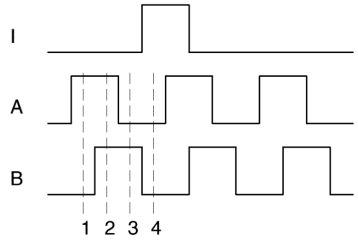
However, recognizing the limitations of these assumptions is critical. Broad assumptions, such as the unimodality of the distribution, may not hold true in real-world applications. For instance, consider a sonar sensor. The sonar's accuracy is high, with random errors mainly stemming from noise (e.g., from timing circuits), making a unimodal and possibly Gaussian oise distribution a reasonable assumption. However, in scenarios where the sonar encounters materials causing coherent reflections, distance overestimations become likely, suggesting a bias towards positive errors, and requiring a bimodal distribution for a comprehensive error model. Additionally, since overestimation is more common than underestimation, the distribution should also be asymmetric. Similarly stereo vision systems demonstrate variability in error profiles depending on operational conditions. Correct image correlations result in random errors primarily from camera noise, affecting measurement precision. Conversely, incorrect correlations lead to significant measurement errors, challenging both the assumptions of unimodality and symmetry of the error distribution.

7.3 Common Sensors on Mobile Robots

7.3.1 Encoders

Encoders are electro-mechanical instruments that convert mechanical motion into a series of digital signals. Such signals can be interpreted to measure relative or absolute position measurements. Encoders are commonly employed in applications such as sensing the rotation angle and speed of wheels or motors. Thanks to their extensive use beyond mobile robotics, significant advancements have been made in developing affordable encoders that provide high resolution. In the field of mobile robotics, encoders stand out as a preferred method to control the position or speed of wheels and other motor-driven joints. These sensors are proprioceptive and therefore their estimates are expressed in the reference frame of the robot. Optical encores 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 (see Figure 7.1). Through signal pro-





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

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

cessing, it is possible to 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. Encoders used in mobile robotics typically have a resolution of about 2,000 CPR, whereas the optical encoder industry is capable of producing encoders exhibiting resolutions as high as 10,000 CPR. In terms of bandwidth, it is critical that the encoder is sufficiently fast to handle the expected shaft rotation rates. Luckily, industrial optical encoders present no bandwidth limitation to mobile robot applications.

In mobile robotics, quadrature encores are often employed. This design features an additional set of illumination and detection components shifted by 90 degrees relative to the first, on the rotor disk. The outcome is two square waves that provide enhanced information (see Figure 7.1). The ordering of which square wave produces a rising edge first identifies the direction of rotation. Furthermore, the resolution is improved by a factor of four with no change to the rotor disc. Thus, a 2,000 CPR encoder in quadrature yields 8,000 counts.

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

7.3.2 Heading Sensors

Heading sensors can be proprioceptive (e.g., gyroscopes, inclinometers) or exteroceptive (e.g., compasses). They are used to determine the robot's orientation in space. Additionally, they can also be used to obtain position estimates by fusing the orientation and velocity information and integrating, a process known as *dead reckoning*.

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

field, and are therefore less well suited for indoor applications.

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

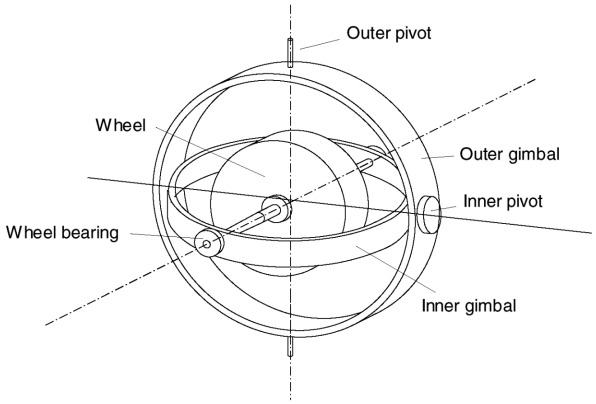


Figure 7.2: Two-axis mechanical gyroscope. (Figure from Siegwart et al.)

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

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

Optical gyroscopes are a relatively new invention. They use angular speed sensors with two monochromatic light beams, or lasers, emitted from the same source. Two beams are sent, one clockwise and the other counterclockwise, through an optical fiber. Since the laser traveling in the direction of rotation has a slightly shorter path, it will have a higher frequency. This frequency difference δf is proportional to the angular velocity, which can therefore be estimated. In modern optical gyroscopes, bandwidth can easily exceed 100 kHz, while resolution can be smaller than 0.0001 degrees/hr.

7.3.3 Accelerometer

An accelerometer is a device used to measure net accelerations (i.e. the net external forces acting on the sensor, including gravity). Mechanical accelerometers

are essentially spring-mass-damper systems that can be represented by the second order differential equation [3]:

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

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

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

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

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

7.3.4 *Inertial Measurement Unit (IMU)*

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

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

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

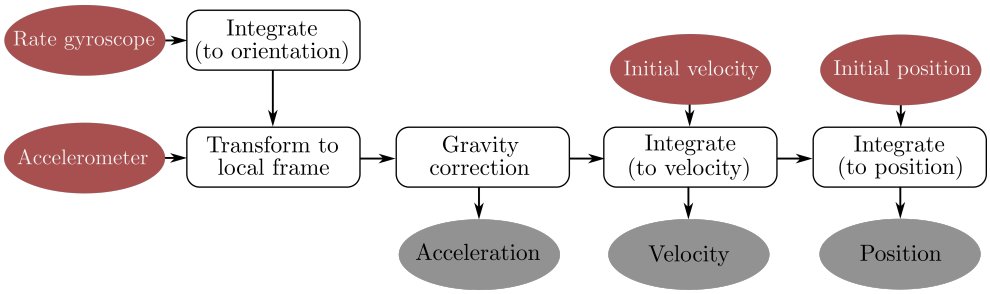


Figure 7.3: Inertial measurement unit (IMU) block diagram.

7.3.5 *Beacons*

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

7.3.6 *Active Ranging*

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

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

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

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

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

7.3.7 Other Sensors

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

7.4 Computer Vision

Vision sensors have become crucial sensors for perception in the context of robotics. This is generally due to the fact that vision provides an enormous amount of information about the environment and enables rich, intelligent interaction in dynamic environments[4]. The main challenges associated with vision-based sensing are related to processing digital images to extract salient information like object depth, motion and object detection, color tracking, feature detection, scene recognition, and more. The analysis and processing of images are generally referred to as *computer vision* and *image processing*. Tremendous advances and new theoretical findings in these fields over the last several decades have led to sophisticated computer vision and image processing techniques to be utilized in industrial and consumer applications such as photography, defect inspection, monitoring and surveillance, video games, movies, and more. This

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

section introduces some fundamental concepts related to these fields, and in particular will focus on cameras and camera models.

7.4.1 Digital Cameras

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

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

7.4.2 Image Formation

Before reaching the camera's sensor, light rays first originate from a light source. In general the rays of light reflected by an object tend to be scattered in many directions and may consist of different wavelengths. Averaged over time, the emitted wavelengths and directions for a specific object can be precisely described using object-specific probability distribution functions. In particular, the light reflection properties of a given object are the result of how light is reflected, scattered, or absorbed based on the object's surface properties and the wavelength of the light. For example, an object might look blue because blue wavelengths of light are primarily scattered off the surface while other wavelengths are absorbed. Similarly, a black object looks black because it absorbs most wavelengths of light, and a perfect mirror reflects all visible wavelengths.

Cameras capture images by sensing these light rays on a photoreceptive surface (e.g., a CCD or a CMOS sensor). However, since light reflecting off an object is generally scattered in many directions, simply exposing a planar photoreceptive surface to these reflected rays would result in many rays being cap-

tured at each pixel. This would lead to blurry images! A solution to this issue is to add a barrier in front of the photoreceptive surface that blocks most of these rays, and only lets some of them pass through an aperture (see Figure 7.5). The earliest approach to filtering light rays in this way was to have a small hole in the barrier surface. Cameras with this type of filter were referred to as *pinhole* cameras.

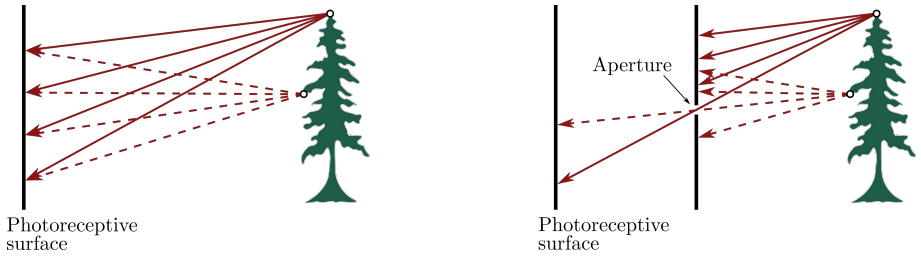


Figure 7.5: Light rays on a photoreceptive surface referred to as the image plane. On the left, numerous rays being reflected and scattered by the object leads to blurry images whereas (on the right), a barrier has been added so that the scattered light rays can be distinguished.

7.4.3 *Pinhole Camera Model*

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

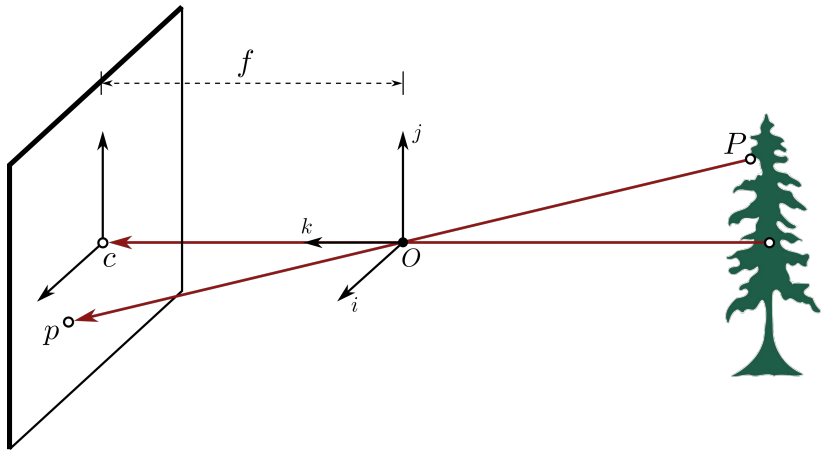


Figure 7.6: Pinhole camera model. Due to the geometry of the pinhole camera system, the object's image is inverted on the image plane. In this figure, O is the camera center, c is the image center, and p the principal point.

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

collinear it is possible to deduce the following relationships between the coordinates $P = (X, Y, Z)$ and $p = (x, y, z)$:

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

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

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

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

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

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

7.4.4 *Thin Lens Model*

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

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

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

where again the point P has coordinates (X, Y, Z), its corresponding point p on the image plane has coordinates (x, y, z), and f is the focal length. Combining these two equations yields the *thin lens equation*:

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

Note that in this model for a particular focal length f, a point P is only in sharp focus if the image plane is located a distance z from the lens. However, in practice an acceptable focus is possible withing some range of distances (called depth of field or depth of focus). Additionally, if Z approaches infinity light

would focus a distance of f away from the lens. Therefore, this model is essentially the same as a pinhole model if the lens is focused at a distance of infinity. As can be seen, this formula can also be used to estimate the distance to an object by knowing the focal length f and the current distance of the image plane to the lens z. This technique is called *depth from focus*.

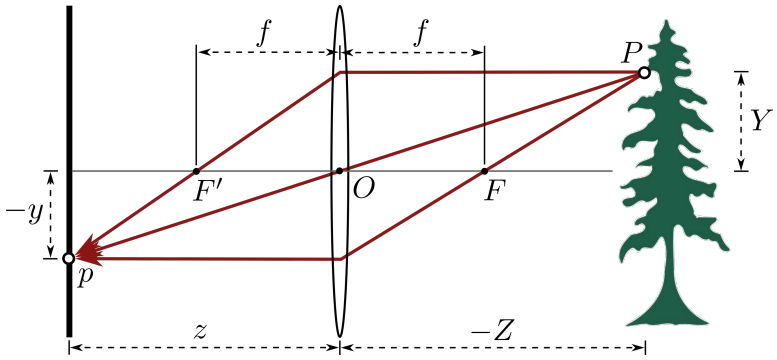


Figure 7.7: The thin lens model.

8
Camera Models and Calibration

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

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

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

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

Camera Models and Calibration

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

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

8.1 Perspective Projection

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

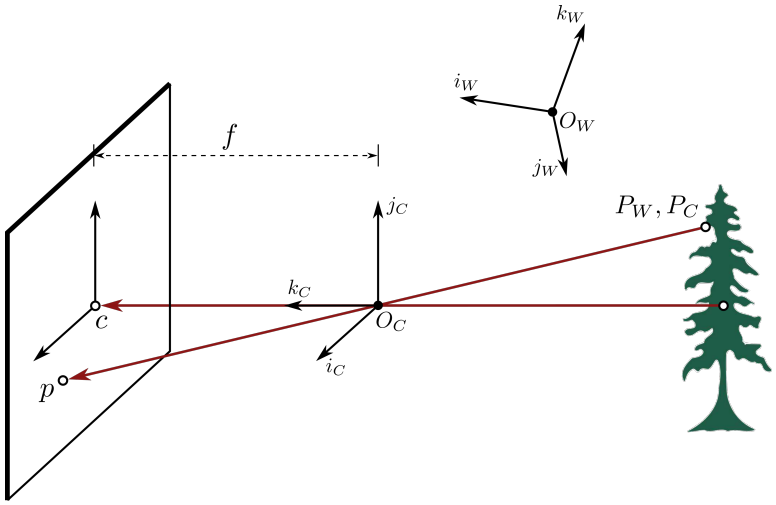


Figure 8.1: Graphical representation of the pinhole camera model. In this model the point O_C is the camera center, c is the image center, and f is the focal length of the camera. It is assumed that all light rays from point P in the scene pass through point O_C and are captured on the image plane at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

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

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

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

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

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

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

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

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

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

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

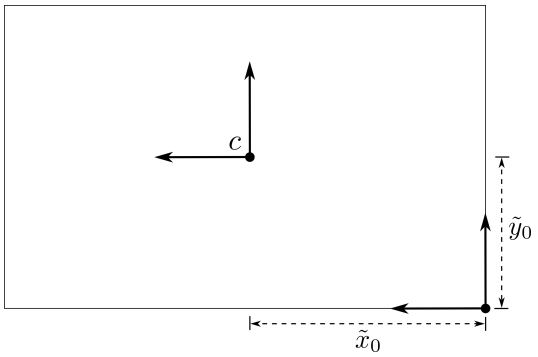


Figure 8.2: A new reference frame with coordinates denoted by $\tilde{(\cdot)}$ is defined with its origin in the lower corner of the image plane. The image center coordinates in this new frame are denoted $(\tilde{x}_0, \tilde{y}_0)$.

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

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

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

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

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

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

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

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

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

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

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

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

linear relationship:

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

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

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

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

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

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

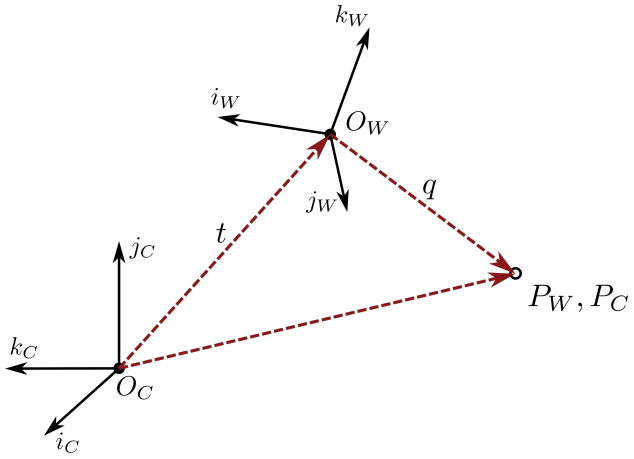


Figure 8.3: A depiction of the point P expressed either in camera coordinates, P_C, or in world frame coordinates, P_W. The world frame origin is denoted by O_W and the camera frame origin is denoted by O_C.

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

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

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

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

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

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

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

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

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

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

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

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

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

which can then be simplified to:

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

In Equation (8.13), P_W^h is the homogeneous coordinate representation of P_W and p^h is the homogeneous coordinate representation of p. Additionally, recall that the matrix $K \in \mathbb{R}^{3\times 3}$ is the matrix of intrinsic camera parameters, and the matrix $[R \quad t] \in \mathbb{R}^{3\times 4}$ contains *extrinsic* parameters (i.e., that describe the

camera's position and orientation relative the points in the scene). Note that the total number of degrees of freedom is 11, where 5 are from the intrinsic parameters that define K, 3 are from the rotation matrix R, and 3 are from the position vector t.

8.2 Camera Calibration: Direct Linear Method

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

8.2.1 Direct Linear Calibration: Step 1

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

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

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

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

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

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

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

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

or equivalently (via algebraic manipulation) the expressions:

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

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

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

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

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

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

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

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

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

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

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

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

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

and the necessary optimality conditions:

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

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

8.2.2 *Direct Linear Calibration: Step 2*

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

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

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

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

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

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

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

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

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

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

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

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

with

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

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

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

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

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

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

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

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

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

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

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

where:

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

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

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

or by combining them:

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

which is linear in the unknowns b. Importantly, while the homographies M are different for each image, the intrinsic camera parameters (i.e., the vector b) are the same! Therefore for N images from the same camera (but with potentially

different perspectives) these constraints in Equation (8.27) can be stacked to give:

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

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

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

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

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

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

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

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

As one final step, it is noted that the matrix R defined with columns r_1, r_2, and r_3 will not in generally satisfy the properties of a rotation matrix (i.e., orthonormality). One final step to this overall procedure is to correct this issue by

finding the rotation matrix that best corresponds to these column vectors. This is accomplished again by optimization, and in particular by formulating the problem:

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

where

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

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

8.3 Camera Auto-Calibration

The direct linear transformation or Zhang's flexible calibration method 8.2.3 requires the 2D and 3D point correspondences to calculate the intrinsic and extrinsic parameters. Auto-calibration offers an alternative approach that does not require a specific calibration pattern. Instead, it utilizes 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 SIFT [9], we can estimate the intrinsic matrix K and extrinsic matrices R and t via bundle adjustment [10] based on the static scene geometry.

The camera auto-calibration process consists of five key steps. First, we employ SIFT to extract and match key points across multiple views. Using these matched points, we then estimate the fundamental matrix through Singular Value Decomposition (SVD). Next, we initialize camera's intrinsic parameters and combine them with the fundamental matrix to calculate the essential matrix and camera's extrinsic parameters. When we get all the camera parameters, we can use triangulation to calculate the 3D points for the matched 2D key points. Finally, both the intrinsic and extrinsic parameters are refined using bundle adjustment, which minimizes the projection error for optimal calibration.

[9] 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

[10] 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

8.3.1 Step 1: Extract and match key points

First, we use SIFT to identify key points and their correspondences across multiple views. Let $\{p_0, p_1, \dots p_n\}$ denote the matching points across the views, where p_i indicates the corresponding point in the i-th image. The SIFT algorithm is robust in detecting and describing local features in images, making it effective for finding correspondences under varying conditions.

8.3.2 Step 2: Fundamental matrix

Consider N images of a static scene, where each image provides a different viewpoint. The fundamental matrix F relates corresponding points between pairs of images. For a pair of images (I_i, I_j), the fundamental matrix F_{ij} is defined as:

$$p_i^\top F_{ij} p_j = 0, \tag{8.33}$$

where p_i and p_j are corresponding points in images I_i and I_j, respectively. The fundamental matrix describes the epipolar geometry between the two views.

We construct a point correspondence matrix A, where each row represents a correspondence between points in the two images. Let (x_i, y_i) and (x_i', y_i') be the coordinates of the matched points p_i in the first and second image, respectively, obtained using SIFT. Each row of matrix A is formed using these coordinates as follows:

$$A = \begin{bmatrix} x_1'x_1 & x_1'y_1 & x_1' & y_1'x_1 & y_1'y_1 & y_1' & x_1 & y_1 & 1 \\ x_2'x_2 & x_2'y_2 & x_2' & y_2'x_2 & y_2'y_2 & y_2' & x_2 & y_2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n'x_n & x_n'y_n & x_n' & y_n'x_n & y_n'y_n & y_n' & x_n & y_n & 1 \end{bmatrix} \tag{8.34}$$

Then we use Singular value decomposition (SVD) to solve the linear system $Af = 0$ to get f, which is the vectorized form of the fundamental matrix F. A detailed derivation of this process can be found in the next section 9.1.1.

8.3.3 Step 3: Essential matrices and camera extrinsic parameters.

Assuming the intrinsic matrix K remains the same for all images, the essential matrix E_{ij} for the images I_i and I_j is given by:

$$E_{ij} = K^\top F_{ij} K. = [t]_\times R \tag{8.35}$$

where $[t]_\times$ is the matrix representation of the cross product with the translation t and R is the rotation matrix. We can get the camera extrinsic rotation and translation parameters through solve the SVD:

$$E = U\Sigma V^\top, \tag{8.36}$$

where $\Sigma = \text{diag}(1, 1, 0)$. There are two possible solutions for the camera extrinsics, and the correct solution can be identified by using them to compute the 3D points of the 2D matched key points in the next step. The correct extrinsics will ensure that most of the 3D points lie in front of both cameras, meaning they will have positive depth values.

$$\begin{aligned} R_1 &= UWV^\top, & R_2 &= UW^\top V^\top, \\ t_1 &= U[:,2], & t_2 &= -U[:,2], \end{aligned} \tag{8.37}$$

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{8.38}$$

8.3.4 Step 4: Calculate the 3D points using triangulation

The purpose of this step is to calculate the 3D points using triangulation. Since the depth information of corresponding points in 2D images is not available, we must rely on triangulation to determine the 3D positions. Given multiple views of a scene, we can use the correspondences between points in different images to estimate their 3D coordinates. The projection matrix P for an image is defined as:

$$P = K[R|t], \tag{8.39}$$

where K is the intrinsic matrix, R is the rotation matrix, and t is the translation vector. Let's denote the coordinates of the points in the images I_i and I_j as (x_i, y_i) and (x_j, y_j). The projection equation for a point (X, Y, Z) in homogeneous coordinates can be written as:

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = P_i \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}, \tag{8.40}$$

and similarly for the second image:

$$\begin{bmatrix} x_j \\ y_j \\ 1 \end{bmatrix} = P_j \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}. \tag{8.41}$$

To solve for (X, Y, Z), we rearrange these equations into a homogeneous linear system:

$$\begin{bmatrix} x_i P_{i,3}^\top - P_{i,1}^\top \\ y_i P_{i,3}^\top - P_{i,2}^\top \\ x_j P_{j,3}^\top - P_{j,1}^\top \\ y_j P_{j,3}^\top - P_{j,2}^\top \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = 0, \tag{8.42}$$

where, $P_{i,1}$, $P_{i,2}$, and $P_{i,3}$ are the first, second, and third rows of the projection matrix P_i, respectively, and similarly for $P_{j,1}$, $P_{j,2}$, and $P_{j,3}$.

We can apply the least squares method to solve this equation and get an estimate of the 3D coordinates.

8.3.5 Step 5: Refine the camera parameters.

The cost function to minimize is:

$$\sum_{i=1}^{N} \sum_{j=1}^{M} \|p_{ij} - P_i X_j\|^2, \tag{8.43}$$

where p_{ij} is the observed 2D point, P_i is the projection matrix, and X_j is the estimated 3D point.

To refine the intrinsic and extrinsic parameters, we begin by initializing the camera's intrinsic parameters, either by referencing the camera brand's configuration or using a standard setup. Next, we proceed through steps 1 to 4 to estimate the extrinsics and establish 2D-3D point correspondences. With these in place, we apply non-linear optimization techniques, such as Levenberg-Marquardt, to minimize the projection error. This process is repeated iteratively until convergence is reached.

8.4 Limitations

8.4.1 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 such effect that is not captured by the pinhole model is *radial distortion*, which is an effect seen in real lenses where either barrel distortion or pincushion distortion will affect the real pixel coordinates. Images showing both barrel and pincushion distortion are provided in Figure 8.4.

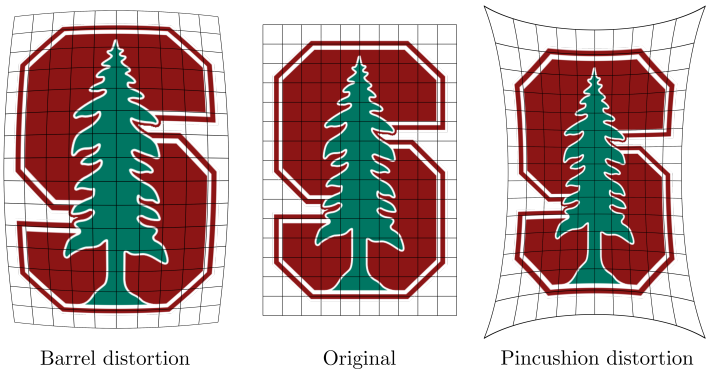


Figure 8.4: Different kinds of radial distortions that are seen in real lenses, which may affect the accuracy of the pinhole camera model.

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

$$\begin{bmatrix} u_d \\ v_d \end{bmatrix} = \begin{bmatrix} u_d \\ v_d \end{bmatrix} (1 + kr^2) \begin{bmatrix} u - u_{cd} \\ v - v_{cd} \end{bmatrix} + \begin{bmatrix} u_{cd} \\ v_{cd} \end{bmatrix}, \tag{8.44}$$

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.

8.4.2 *Measuring Depth*

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

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

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

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

8.5 *Exercises*

check with existing exercises

Code Exercise 8.5.1 (Manipulating Pointclouds from KITTI)**.** In this exercise you will learn how to manipulate sensor pointclouds from the KITTI dataset. Have a look at notebook `pointclouds`

Code Exercise 8.5.2 (Manipulating Pointclouds from KITTI)**.** This notebook implements a method for camera calibration based on the Direct Linear Transform (DLT) formulation. The technique corresponds to the one presented in the paper by Zhang et al. Have a look at notebook `pointclouds`.

add reference

9
Stereo Vision and Structure From Motion

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

[1] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, 2011, D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2011

Stereo Vision and Structure From Motion

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

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

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

9.1 Stereo Vision

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

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

9.1.1 Epipolar Constraints

As previously mentioned, the first step in the stereo vision process is to fuse the two (or more) images and generate point correspondences[3]. This task can be quite challenging, and erroneously matching features can lead to large errors in the reconstruction step. Therefore, several techniques are leveraged to make this task simpler. The most important simplifying technique is to impose an *epipolar constraint*.

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

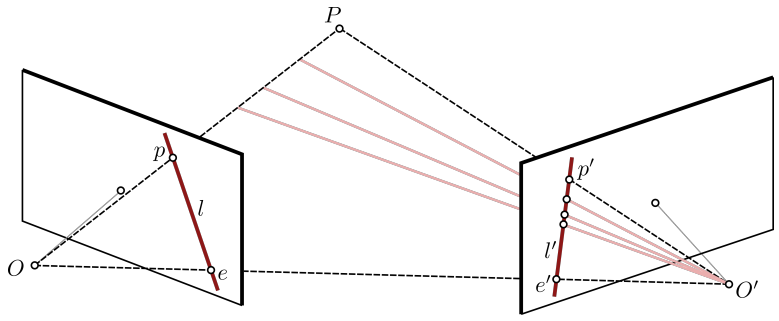


Figure 9.1: The point P in the scene, the optical centers O and O' of the two cameras, and the two images p and p' of P all lie in the same plane, referred to as the epipolar plane. The lines l and l' are the epipolar lines of the points p and p', respectively. Note that if the point p is observed in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

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

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

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

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

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

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

where F, referred to as the *fundamental matrix*, has seven degrees of freedom and is singular. For a derivation of the epipolar constraint see Section 7.1 from Forsyth et al.[4]. Additionally, the matrix F is only dependent on the intrinsic camera parameters for each camera and the geometry that defines their relative positioning, and can be assumed to be constant. The expression for the fundamental matrix in terms of the camera intrinsic parameters is:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.1.2 *Image Rectification*

Given a pair of stereo images, epipolar rectification is a transformation of each image plane such that all corresponding epipolar lines become colinear and parallel to one of the image axes, for convenience usually the horizontal axis. The resulting rectified images can be thought of as acquired by a new stereo camera obtained by rotating the original cameras about their optical centers. The great advantage of the epipolar rectification is the correspondence search becomes simpler and computationally less expensive because the search is done along the horizontal lines of the rectified images. The steps of the epipolar rectification algorithm are illustrated in Figure 9.2. Observe that after the rectification, all the epipolar lines in the left and right image are colinear and horizontal. For an in-depth discussion on algorithms for image rectification see [5].

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

9.1.3 *Correspondence Problem*

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

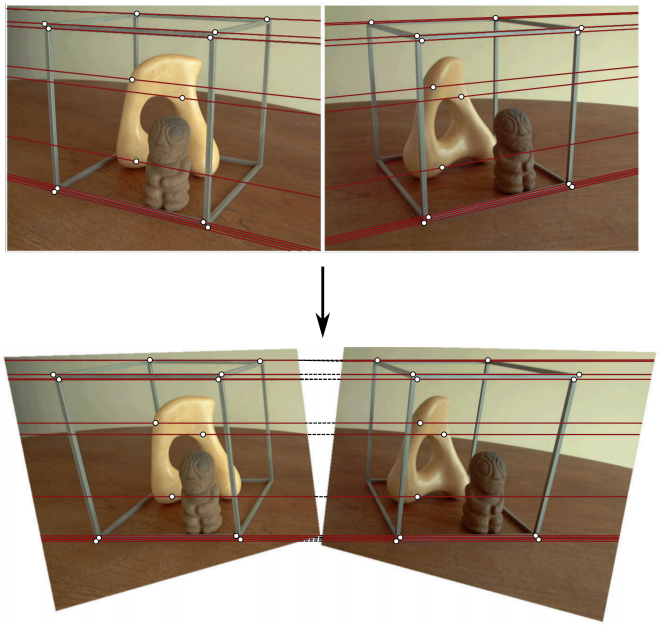


Figure 9.2: Epipolar rectification example from Loop et al. (1999).

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

9.1.4 *Reconstruction Problem*

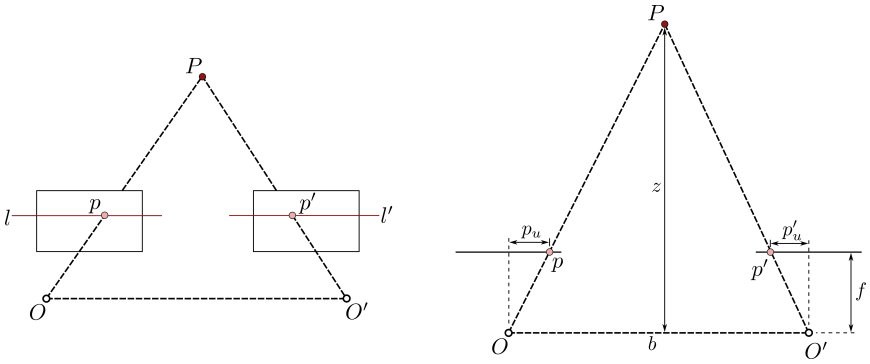


Figure 9.3: Triangulation with rectified images (horizontal view on the left, top-down view on the right).

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

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

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

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

which can be algebraically simplified to:

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

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

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

9.2 *Structure From Motion (SFM)*

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

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

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

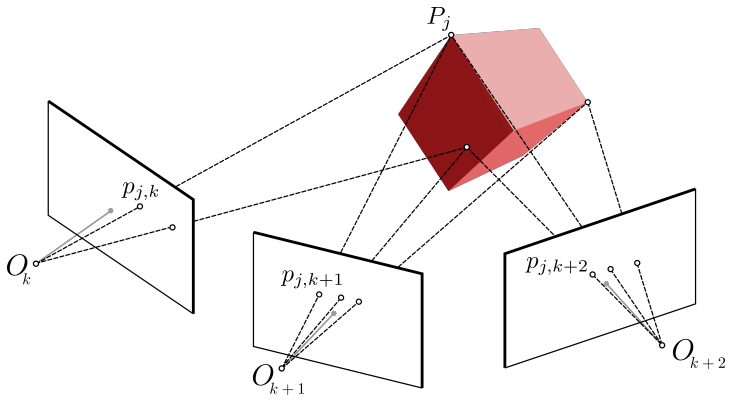


Figure 9.5: A depiction of the structure from motion (SFM) method. A single camera is used to take multiple images from different perspectives, which provides enough information to reconstruct the 3D scene.

object at a longer distance and a smaller object at a closer distance may yield the same projections.

One application of the SFM concept is known as *visual odometry*. Visual odometry estimates the motion of a robot by using visual inputs (and possible additional information). This approach is commonly used in practice, for example by rovers on Mars, and is useful because it not only allows for 3D scene reconstruction but also to recover the motion of the camera.

10
Image Processing

The previous chapters focused on using camera models to identify the relationship between points in a 3D scene and their projections onto the camera image, as well as how to leverage those models to reconstruct 3D scene structure from 2D images. Alternatively, this chapter begins to look at methods for extracting other types of information through *image processing*, for example to answer the question "what object am I seeing?" rather than "how far away is this object?". Extracting this type of visual content from raw images is important for mobile robots to be able to intelligently interpret their surroundings. In fact, it can have a major impact on the ability of the robot to perform several tasks including localization and mapping or decision making. This chapter focuses on some of the more commonly used tools in image processing including image filtering, feature detection, and feature description[1,2].

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

[2] H. P. Moravec. "Towards automatic visual obstacle avoidance". In: *5th International Joint Conference on Artificial Intelligence*. 1977

Image Processing

Image processing is a form of signal processing where the input signal is an image (such as a photo or a video) and the output is either an image or a set of parameters associated with the image. While a large number of image processing techniques exist, this chapter focuses on some of the more fundamental methods that are relevant for robotics. In particular, these methods will be related to image filtering, feature detection, and feature description[3].

[3] The software library OpenCV implements a number of useful image filtering algorithms: `https://docs.opencv.org`.

In the following methods, grayscale images are treated as functions $I\colon [a,b] \times [c,d] \to [0,L]$, where $I(x,y)$ represents the grayscale pixel intensity at (x,y). For a color image, I is a vector valued function with three components, one each for the red, green, and blue color channels of the image.

10.1 Image Filtering

Image filtering is one of the principal tasks in image processing. The terminology "filter" comes from frequency domain signal processing and refers to the process of accepting or rejecting certain frequency components of a signal (e.g. eliminating high-frequency noise).

Perhaps the most common type of image filtering is *spatial filtering*. The basic principle of spatial filtering is that a particular pixel is modified in the filtered image based only on the pixels in the immediate spatial neighborhood (see Figure 10.1). To be more specific, a spatial filter for an image $I(x, y)$ consists of:

1. A neighborhood S_{xy} of pixels around a particular point (x, y) under examination, typically rectangular.
2. A predefined operation F that is performed on the image pixels encompassed by the neighborhood S_{xy}.

Once the operation F has been applied to all pixels (x, y) in the image I a new image $I'(x, y)$ is defined.

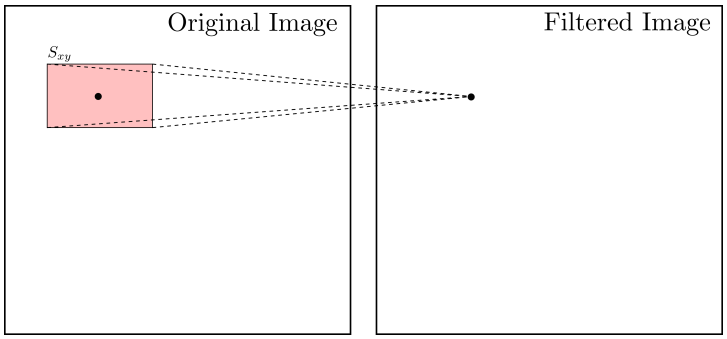


Figure 10.1: Illustration of the concept of spatial filtering. The spatial filter operates on a neighborhood S_{xy} of each point in the original image to produce a new pixel in the filtered image.

In general filters can be linear or nonlinear, but many of the most fundamental filters are linear and can be expressed mathematically as:

$$I'(x, y) = F \circ I = \sum_{i=-N}^{N} \sum_{j=-M}^{M} F(i, j) I(x + i, y + j), \tag{10.1}$$

where N and M are integers that define the width and height of a rectangular neighborhood S_{xy}. Based on the size of this neighborhood, it is said that this filter is of size $(2N + 1) \times (2M + 1)$. Additionally, the filter operation F is usually called a *mask* or *kernel*. Broadly speaking, filters expressed by (10.1) are referred to as *correlation filters*.

Another type of linear filters that are commonly used are referred to as *convolution* filters. Convolution filters are similar to correlation filters but use reverse image indices (in fact correlation and convolution filters are identical when the filter mask is symmetric in both the horizontal and vertical directions). In particular, these filters are expressed mathematically as:

$$I'(x, y) = F * I = \sum_{i=-N}^{N} \sum_{j=-M}^{M} F(i, j) I(x - i, y - j). \tag{10.2}$$

Convolution filters are associative, meaning that for two different filter masks F and G it is true that $F * (G * I) = (F * G) * I$. One example of how the associative property is useful is for smoothing an image *before* taking applying a

differentiation filter. Suppose the mask F implemented a derivative filter and G implemented a smoothing filter, then sequentially applying these filters would result in $F * (G * I)$. However, because of the associative property the masks can be convolved together *first* such that only one filter needs to be applied to the image (i.e. $(F * G) * I$).

Note that in both the correlation and convolution filters the boundaries of the image need some special care because of the width and height of the mask. For example, Figure 10.2 shows how the filtered image is smaller than the original due to the width and height of the mask. Some possible options to handle this include padding the image, cropping it, extending it, or wrapping it. However, as images are generally quite large the exact approach likely won't vary the final result significantly.

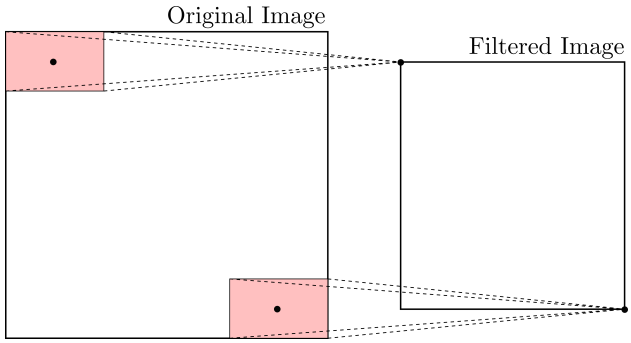


Figure 10.2: Due to the width and height of the mask, the filtered image may be smaller than the original. However this can be fixed with several techniques, such as padding.

Example 10.1.1 (Practical Considerations for Image Filtering)**.** Implementation of correlation and convolution filters typically leverages some additional "tricks" to make things easier to implement. In this example two such tricks will be introduced: zero-padding and a change in indexing.

First, to more simply accommodate varying sizes of filters (including even and odd sized filters) the indexing is often changed such that the coordinate of interest is associated with the top-left element in the window rather than the center. In particular, for a correlation filter this would correspond to:

$$I'(x,y) = F \circ I = \sum_{i=1}^{K} \sum_{j=1}^{L} F(x,y) I(x+i-1, y+j-1), \tag{10.3}$$

where K and L are integers that define the width and height of the filter and the pixel (x,y) is at row x and column y. However, note that with this formulation the output image I' will be shifted up and to the left. To see this consider the pixel at $x = 1$ and $y = 1$ in the new image I', which would correspond to the top-left pixel I'. This new pixel value is generated by applying the filter F over the pixels in the original image I at rows $\{1, \ldots, K\}$ and columns $\{1, \ldots, L\}$ (which is not centered at $(1,1)$ in the original image I). Therefore it will appear as if the image has been shifted! But in practice this isn't an issue as long as you always index with respect to the top-left corner. An example of top-left indexing is shown in Figure 10.3

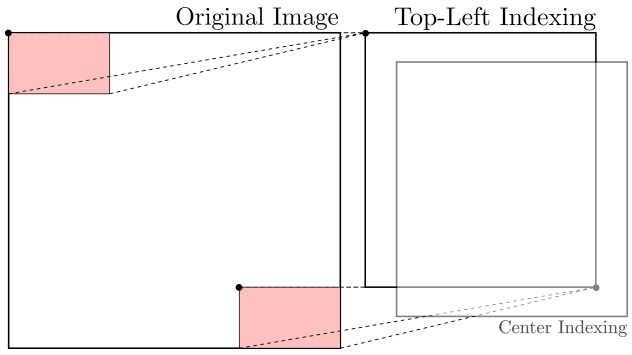


Figure 10.3: Top-left indexing is typically easier to implement than center indexing. Notice that when top-left indexing, it appears as if the filtered image has shifted with respect to when center indexing is used.

Zero-padding (also commonly referred to as *same padding*) is another simple trick that can be used to ensure that the output filtered image I' has the same dimension as the input image I. In this approach the left and right boundaries of the image are *each* padded by $\lfloor K/2 \rfloor$ columns of zeros, and the top and bottom boundaries are padded by $\lfloor L/2 \rfloor$ rows of zeros ($\lfloor \cdot \rfloor$ denotes the "floor" operation). For example the image:

$$I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix},$$

would become

$$I_{\text{padded}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ 0 & 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

for filters $F \in \mathbb{R}^{3\times3}$, $F \in \mathbb{R}^{2\times2}$, $F \in \mathbb{R}^{2\times3}$ and $F \in \mathbb{R}^{3\times2}$. When using this padding rule with the correlation filter (10.3) and a filter F with $K = 2,3$ and $L = 2,3$, the new image I' can be defined for values $x \in \{1,2,3\}$ and $y \in \{1,2,3\}$, resulting in I' being the same dimension as the original image I. The use of padding (along with top-left indexing) is also shown graphically in Figure 10.4

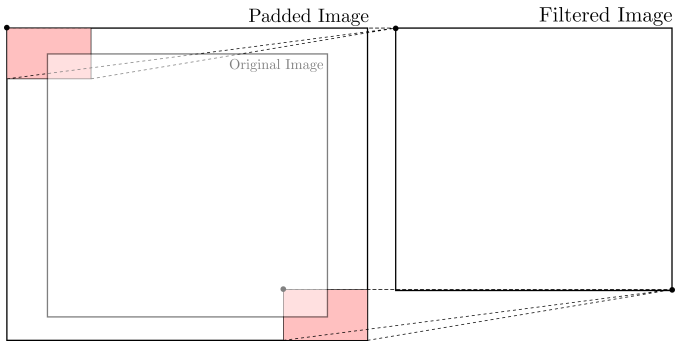


Figure 10.4: Image padding is a commonly used technique to ensure that the size of the filtered image is the same size as the original.

10.1.1 Moving Average Filter

The moving average filter returns the average of pixels in the mask, which achieves a smoothing effect (i.e. removes sharp features in the image). For example, a moving average filter with a normalized 3×3 mask is defined with the operation F in (10.1) chosen as:

$$F = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Note that due to symmetry of the mask, the correlation (10.1) and convolution (10.2) filters will be identical. Additionally, the normalization is used to maintain the overall brightness of the image.

10.1.2 Gaussian Smoothing Filter

Gaussian smoothing filters are similar to the moving average filer, but instead of weighting all of the pixels evenly they are weighted by the Gaussian function:

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(- \frac{x^2 + y^2}{2\sigma^2} \right).$$

This function is used to obtain the mask operation F by sampling the function about the center pixel (i.e. for the center pixel with $i = j = 0$ in (10.1), sample $G_\sigma(0, 0)$). For example, for a normalized 3×3 mask with $\sigma = 0.85$ this filter is approximately defined by:

$$F = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

Like the moving average filter, this filter mask is symmetric and therefore yields identical results with respect to the correlation (10.1) or convolution (10.2) filters. The advantage of the Gaussian filter is that it provides more weight to the neighboring pixels that are closer. An example of this filter is shown in Figure 10.5.

10.1.3 Separable Masks

A mask F is called *separable* if it can be broken down into the convolution of two kernels $F = F_1 * F_2$. If a mask is separable into "smaller" masks, then it is often cheaper to apply F_1 followed by F_2, rather than by F directly. One special case of this is when the mask can be represented as an outer product of two vectors (meaning it is equivalent to the 2D convolution of those two vectors). If the mask is of shape $M \times M$, and the input image has size $w \times h$, then the computational complexity of directly performing the convolution is $O(M^2wh)$. However, by separating the masks the computational cost is $O(2Mwh)$, which is

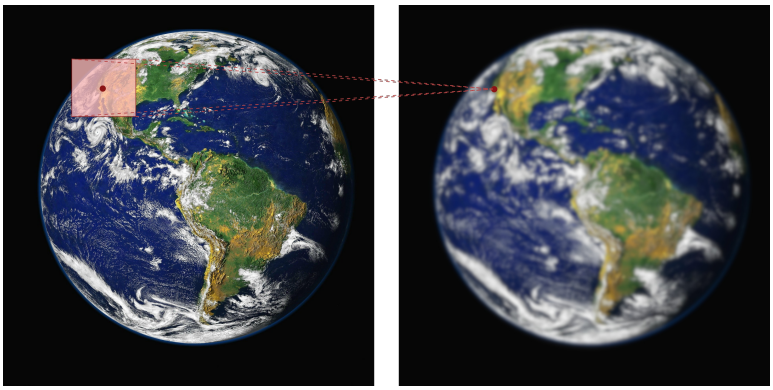


Figure 10.5: Example of a Gaussian smoothing filter, which produces a smoothing (blurring) effect on the filtered image.

linear in M rather than quadratic. As an example, consider the moving average filter mask from before:

$$F = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \frac{1}{9}\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}.$$

As another example, note that the Gaussian smoothing filter mask is also separable. To see why this is, note that the Gaussian weighting function can be decomposed as:

$$\begin{aligned} G_\sigma(x,y) &= \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right), \\ &= \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{x^2}{2\sigma^2}\right)\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{y^2}{2\sigma^2}\right), \\ &= g_\sigma(x)\cdot g_\sigma(y). \end{aligned}$$

10.1.4 Image Differentiation Filters

Taking the derivative of an image can be used to identify certain features, such as edges. On a basic level, the derivative of an image quantifies changes in pixel intensity in both the vertical and horizontal direction. However, since images are represented as functions defined over a discrete domain the traditional method for differentiating continuous functions can not be used. Instead it is more common to just compute differences between pixels, such as using a central difference method:

$$\begin{aligned} \frac{\partial I}{\partial x} &= \frac{I(x+1,y)-I(x-1,y)}{2}, \\ \frac{\partial I}{\partial y} &= \frac{I(x,y+1)-I(x,y-1)}{2}. \end{aligned} \tag{10.4}$$

where $\partial I/\partial x$ is the derivative in the horizontal direction and $\partial I/\partial y$ is the derivative in the vertical direction. It is of course also possible to define the derivatives using just one side, for example $\frac{\partial I}{\partial x} = I(x+1,y) - I(x,y)$.

It is also possible to differentiate an image using convolution filters. In particular, one common approach is to use a convolution filter (10.2) defined with a mask F called a *Sobel mask* (also referred to as simply a Sobel operator). For the x direction this mask is denoted as S_x and for the y direction as S_y:

$$S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \quad S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{10.5}$$

Sobel masks are similar to the central difference method but use more neighboring pixels when calculating the derivative (i.e. they also consider the rows above and below to compute the difference). Note that Sobel masks are also separable.

10.1.5 Similarity Measures

Filtering can also be used to find similar features in different images, which can be useful for solving the correspondence problem in stereo vision or structure-from-motion techniques. In particular, the similarity between the pixel (x, y) in image I_1 and pixel (x', y') in image I_2 can be computed by:

$$\begin{aligned} SAD &= \sum_{i=-N}^{N} \sum_{j=-M}^{M} |I_1(x+i, y+j) - I_2(x'+i, y'+j)|, \\ SSD &= \sum_{i=-N}^{N} \sum_{j=-M}^{M} [I_1(x+i, y+j) - I_2(x'+i, y'+j)]^2, \end{aligned} \tag{10.6}$$

where SAD is an acronym for "sum of absolute differences", SSD is an acronym for "sum of squared differences", and N and M define the size of the window around the pixels that is considered.

10.2 Image Feature Detection

A local feature (also sometimes referred to as interest points, interest regions, or keypoints) in an image is a pattern that differs from its immediate neighborhood in terms of intensity, color, or texture. Local features can generally be categorized in several ways, for example whether they provide semantic content or not. For example, features that may provide semantic content include edges or other geometric shapes (e.g. lanes of a road or blobs corresponding to blood cells in medical images). These types of features were some of the first for which feature detectors were proposed in the image processing literature. Features that do not provide semantic content may also be useful, for example in feature tracking, camera calibration, 3D reconstruction, image mosaicing, and panorama stitching. In these cases it may be more important that the feature be able to be located accurately and robustly over time. A third category of features are those that may not have semantic interpretations individually, but may have meaning as a collection. For instance, a scene could be recognized

by counting the number of feature matches between the observed scene and a query image. In this case only the number of matches is relevant and not the location or type of feature. Applications where these types of features are important include texture analysis, scene classification, video mining, and image retrieval.

In this section several feature detection strategies will be discussed. While many strategies exist for different types of features, the focus here will be on two common features that are often useful in robotics: edges and corners.

10.2.1 Edge Detection

An *edge* in an image is a region where there is a significant change in intensity values along one direction, and negligible change along the orthogonal direction. In one dimension an edge corresponds to a point where there is a sharp change in intensity, which mathematically can be thought of as a point of a function having a large first derivative and a small second derivative. Many edge detectors rely on this concept by differentiating images and looking for spikes in the derivative. An edge detector can be evaluated based on several criteria for robustness and performance, including accuracy, localization, and single response. Good accuracy implies few false positives or negatives (missed edges), good localization implies that the detected edge should be exactly where the true edge is in the image, and a single response implies *only* one edge is detected for each real edge. In practice, noise and discretization can make edge detection challenging.

Most edge detection methods rely on two key steps: smoothing and differentiation. Differentiation is performed in both the vertical and horizontal directions to find locations in the image with high intensity gradients. However, differentiation alone is vulnerable to false positives due to image noise, which is why many algorithms will first smooth the image.

Edge Detection in 1D: An example of how noise can corrupt image differentiation is given in Figure 10.6. Notice that in this case it is impossible to identify the jump in the signal due to the noise levels. Smoothing filters, such as the Gaussian smoothing filter discussed earlier, can help remedy this problem. In particular, suppose the original signal in Figure 10.6 is defined by $I(x)$. Then a smoothed version can be defined by applying a smoothing convolution filter:

$$s(x) = g_\sigma(x) * I(x),$$

where $g_\sigma(x)$ represents a Gaussian smoothing filter, and then by applying the differentiation filter:

$$s'(x) = \frac{d}{dx} * s(x).$$

This process is shown in Figure 10.7. Note however that since these filters are convolutions, the associativity property can be leveraged to actually combine

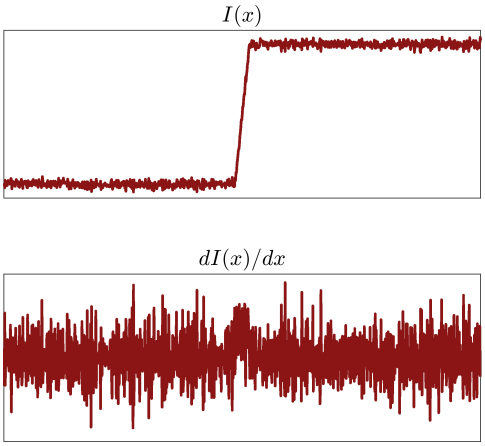


Figure 10.6: Differentiation of signal (e.g. for edge detection) with noise can be particularly challenging, which can be addressed by first smoothing the signal.

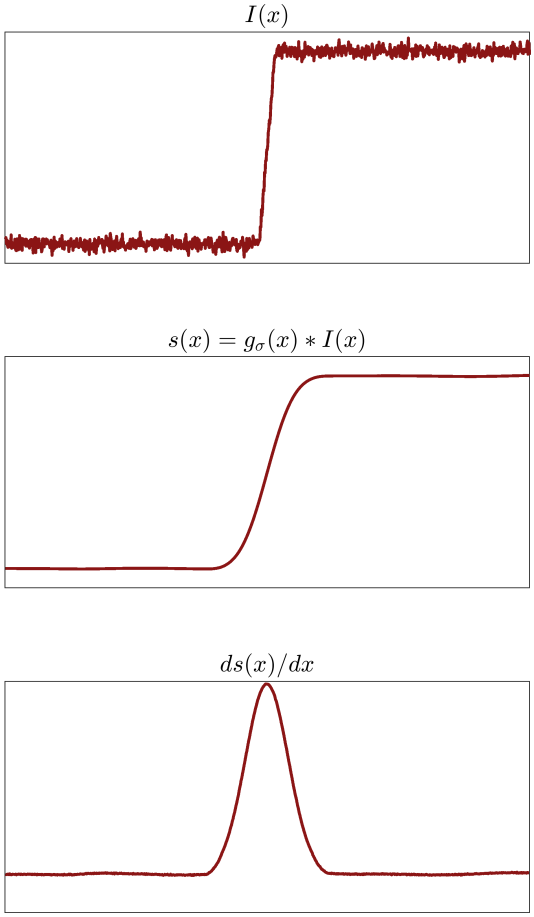


Figure 10.7: Edge detection through convolution with a Gaussian smoothing filter, followed by a differentiation filter.

them into a single filter:

$$s' = (\frac{d}{dx} * g_\sigma) * I.$$

Edge Detection in 2D: Edge detection in a two-dimensional image is quite similar to the example previously discussed in 1D. Let the smoothing filter be the Gaussian smoothing filter from before, and a differentiation filter such as the Sobel filter. The gradient of the smoothed image in both the x and y directions can be written as:

$$\nabla S = \begin{bmatrix} \frac{\partial}{\partial x} * G_\sigma * I \\ \frac{\partial}{\partial y} * G_\sigma * I \end{bmatrix} = \begin{bmatrix} G_{\sigma,x} * I \\ G_{\sigma,y} * I \end{bmatrix} = \begin{bmatrix} S_x \\ S_y \end{bmatrix},$$

where I is the original image and the associativity properties of the smoothing and differentiation convolution filters is used to define the combined filters $G_{\sigma,x}$ and $G_{\sigma,y}$. The magnitude of the gradient can then be computed by:

$$|\nabla S| = \sqrt{S_x^2 + S_y^2},$$

which can be used to check against a predefined threshold value for edge detection. To guarantee thin edges it is also possible to filter out points whose gradient magnitude are above the threshold but are not local maxima. Examples of this process are shown in Figures 10.8 and 10.9.

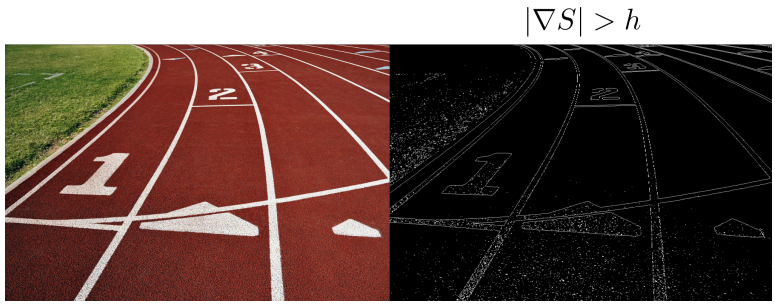


Figure 10.8: Edge detection using the "Sobel" edge detector.

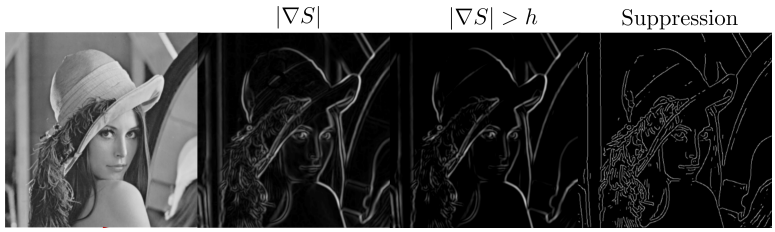


Figure 10.9: Edge detection using the "Canny" edge detector.

10.2.2 *Corner Detection*

A *corner* in an image is defined as an intersection of two or more edges, and also sometimes as a point where there is a large intensity variation in every direction. Important properties of corner detectors include repeatability and distinctiveness. Repeatability quantifies how well the same features can be found in multiple images even under geometric and photometric transformations. Distinctiveness refers to whether the information carried by the patch surrounding

the feature is distinctive, which can be used to reliably produce correspondences. Both of these properties are particularly important in applications such as panorama stitching and 3D reconstruction.

Generally corner detection can be thought of in a similar way to edge detection, except that instead of looking for change along one direction there should be changes in all directions. One well known corner detector is known as the Harris detector [4], which has the useful property that the detection is invariant to rotations and linear intensity changes (i.e. geometric and photometric invariance). However the Harris detector is not invariant to scale changes or geometric affine changes, which has led to the development of scale-invariant detectors such as the Harris-Laplacian detector or the scale-invariant feature transform (SIFT) detector.

[4] C. Harris and M. Stephens. "A combined corner and edge detector". In: *4th Alvey Vision Conference*. 1988

10.3 Image Descriptors

Image *descriptors* describe features so that they can be compared across images, or used for object detection and matching. Similar to image detectors, it is also desirable for image descriptors to be repeatable (i.e. invariant with respect to pose, scale, illumination, etc.) and distinct. Perhaps the simplest example of a descriptor is an $n \times m$ window of pixel intensities centered at the feature, which can be normalized to be illumination invariant. However, such a descriptor is not invariant to pose or scale and is not distinctive, and therefore is generally not useful in practice. Alternative detectors/descriptors that have become popular include SIFT, SURF, FAST, BRIEF, ORB, and BRISK.

10.4 Exercises

10.4.1 Linear Filtering

Complete *Problem 3: Linear Filtering* located in the online repository:

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

where you will explore the use of linear filters for image processing.

11
Information Extraction

The last chapter introduced some fundamental topics related to image processing, namely filtering, feature detection, and feature description. While these techniques are quite useful for a large number of computer vision applications, they may not be sufficient to extract higher-level information from images. For example, the features that were discussed are *local features* that describe important keypoints of the image, but these may be too localized to discuss higher-level features or semantic content. In some cases it may be possible to correlate local features to extract higher-level information (e.g. image matching), but in other cases higher-level algorithms may be useful (e.g. identifying a particular object in a scene, such as a person). In particular, object recognition is a very important task in robotics and therefore some common methods for object recognition will be discussed in this chapter.

This chapter will additionally focus on geometric feature extraction[1], which is used to extract structure from data in the form of geometric primitives (e.g. lines, circles, planes). This is very useful in robotics for localization and mapping, and these algorithms can generally be applied to different types of data, such as data extracted from images or even data collected via laser rangefinders or radar.

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

Information Extraction

This chapter will focus on common methods for extracting higher-level environmental information from sensor data that is useful for robotics. In particular, common algorithms for *geometric feature extraction* will be presented, as well as methods for *object recognition* in images. Such information is crucial for robots operating in real environments to enable intelligent decision making and task planning, as well as to execute plans safely in unknown environments with obstacles.

11.1 Geometric Feature Extraction

It is very common in robotic localization and mapping to represent the environment using simple geometric primitives (e.g. lines, circles, corners, planes) that can be efficiently extracted from sensor data. In particular, in this section techniques for line extraction from range data[2] will be presented. Lines in particular are one of the most fundamental geometric primitives to be extracted, and generally the techniques for extracting other primitives are conceptually similar.

[2] Range data can generally come from a variety of sources, including laser rangefinders, radar, or even computer vision.

There are two main challenges with extracting lines from range data. The first is called *segmentation*, which is the task of identifying which data points belong to which line (and inherently also identifying how many lines there are). The second is *fitting*, which is the task of estimating the parameters that define a line given a set of points. For simplicity this chapter will consider line extraction problems based on two-dimensional range data.

11.1.1 Line Segmentation

The line segmentation problem is to determine how many lines exist in a given set of data and also which data points correspond to each line. Three popular algorithms for line segmentation will be discussed, the *split-and-merge* algorithm, the *random sample consensus (RANSAC)* algorithm, and the *Hough-transform* algorithm.

Split-and-Merge: The split-and-merge algorithm is perhaps the most popular line extraction algorithm and is arguably the fastest (but not as robust to outliers). The concept of this algorithm is quite simple: repeatedly fit lines to sets of points and then split the set of points into two sets if any point lies more than distance d from the line. By repeating this process until no more splits occur, it is guaranteed that all points will lie less than distance d to a line. After this "split" process is completed, a second step merges any of the newly formed lines that are colinear. This algorithm is presented in more detail in Algorithm 11.1. A popular variant of the split-and-merge algorithm is known as the iterative-end-point-fit algorithm. This algorithm is simply the split-and-merge algorithm given in Algorithm 11.1 where the line is simply constructed by connecting the first and the last points of the set. This approach is shown graphically in Figure 11.1.

Random Sample Consensus (RANSAC): Random Sample Consensus (RANSAC) is an algorithm to estimate the parameters of a model from a set of data that may contain outliers (i.e. *robust* model parameter estimation). Outliers are data points that do not fit the model and may be the result of high noise in the data, incorrect measurements, or simply points which come from objects that are unrelated to the current model. For example, a typical laser scan of an indoor en-

Algorithm 11.1: 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
$L \leftarrow [S]$
$i \leftarrow 1$
while $i \leq len(L)$ **do**
  fit a line (α, r) to the set $L[i]$
  detect the point $P \in L[i]$ with maximum distance D to the line (α, r)
  if $D < d$ **then**
    $i \leftarrow i + 1$
  else
    split $L[i]$ at P into new sets S_1 and S_2
    $L[i] \leftarrow S_1$
    $L[i+1] \leftarrow S_2$
Merge colinear sets in L

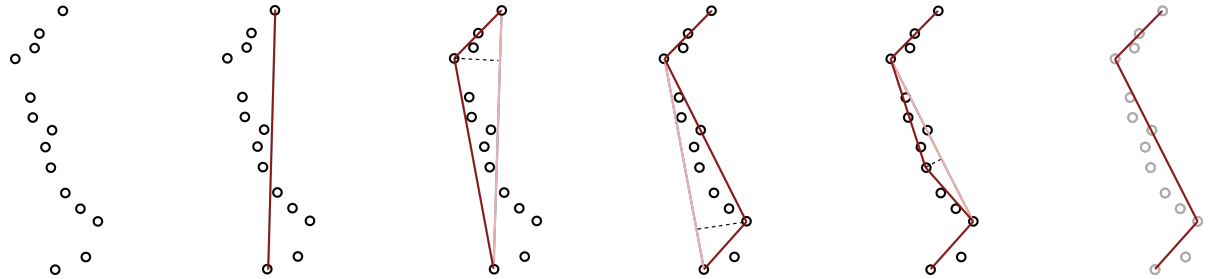


Figure 11.1: Iterative-end-point-fit variation of the split-and-merge algorithm for extracting lines from data.

vironment may contain distinct lines from the surrounding walls but also points from other static and dynamic objects (e.g. chairs or humans). In this case, if the goal was to extract lines to represent the walls then any data point corresponding to other objects would be an outlier. In general, RANSAC can be applied to many parameter estimation problems, and typical applications in robotics include line extraction from 2D range data, plane extraction from 3D point clouds, and structure-from-motion (where the goal is to identify image correspondences which satisfy a rigid body transformation). However for simplicity this section focuses on using RANSAC for line extraction from 2D data.

RANSAC is an iterative method and is non-deterministic (i.e. stochastic or random). Given a dataset S of N points, the algorithm starts by randomly selecting a sample of two points from S. Then a line is constructed from these two points and the distance of all other points to this line is computed. A set of *inliers* comprised of all the points whose distance to the line is within a predefined threshold d is then defined. By repeating this process k times, k inlier sets (and their associated lines) are generated and the inlier set with the most points is returned. This procedure is detailed in Algorithm 11.2 and is also illustrated in Figure 11.2.

Algorithm 11.2: 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 2 points from S
  fit line l_i through the 2 points
  compute distance of all other points to l_i
  construct set of points $\tilde{S}_i$ with distance less than d to l_i
  store line l_i and set of points $\tilde{S}_i$ $i \leftarrow i + 1$
Choose set $\tilde{S}_i$ with maximum number of points

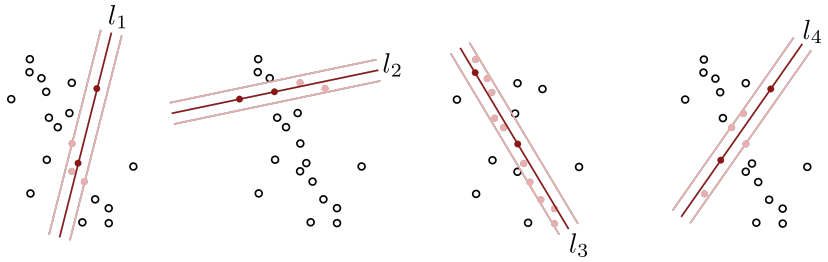


Figure 11.2: Example of the RANSAC algorithm, showing four iterations of the algorithm. If the algorithm was terminated after these four iterations, line l_3 would be returned since it contains the maximum number of points.

Due to the probabilistic nature of the algorithm, as the number of iterations k increases the probability of finding a good solution increases. This approach is used over a brute force search of all possible combinations of two points since the total number of combinations is $N(N-1)/2$, which can be extremely large. In fact, a simple statistical analysis of RANSAC can be performed.

Let p be the *desired* probability of finding a set of points free of outliers and let w be the probability of selecting an inlier from the dataset S of N points, which can be expressed as:

$$w = \frac{\text{\# inliers}}{N}.$$

Assuming point samples are drawn independently from S, the probability of drawing two inliers is w^2 (and $1 - w^2$ is the probability that at least one is an outlier). Therefore, with k iterations the probability that RANSAC *never* selects two points that are both inliers is $(1 - w^2)^k$. Therefore the minimum number of iterations $\bar{k}$ needed to find an outlier-free set with probability p can be found by solving:

$$1 - p = (1 - w^2)^k,$$

for k. In other words, $\bar{k}$ can be computed as:

$$\bar{k} = \frac{\log(1-p)}{\log(1-w^2)}.$$

While the value of w may not be known exactly[3], this expression can still be used to get a good estimate of the number of iterations k that are needed for

[3] There also exist advanced versions of RANSAC that can estimate w in an adaptive online fashion.

good results. It is important to note that this probabilistic approach often leads to a much smaller number of iterations than for brute force searching through all combinations. This can be attributed to the fact that $\bar{k}$ is only a function of w and not the total number of samples N in the dataset.

Overall, the main advantage of RANSAC is that it is a generic extraction method and can be used with many types of features given a feature *model*. It is also simple to implement and is robust with respect to outliers in the data. The main disadvantages are that the algorithm needs to be run multiple times if multiple features are to be extracted, and there are not guarantees that the solutions will be optimal.

Hough Transform: In the Hough transform algorithm, each point (x_i, y_i) of the set S "votes" for a *set* of possible line parameters (m, b) (i.e. slope and intercept). For any given point (x_i, y_i) the candidate set of line parameters (m, b) that could pass through this point must satisfy $y_i = mx_i + b$, which can also be written as:

$$b = -mx_i + y_i.$$

Therefore it can be noted that each point in the original space space (x, y) maps to a *line* in the Hough space (m, b) (see Figure 11.3). The Hough transform algorithm exploits this fact by noting that two points on the same line in the original space will yield two *intersecting* lines in Hough space. In particular, the point where they intersect in the Hough space corresponds to the parameters m^* and b^* that defines the line passing between the points in the original space (see Figure 11.4).

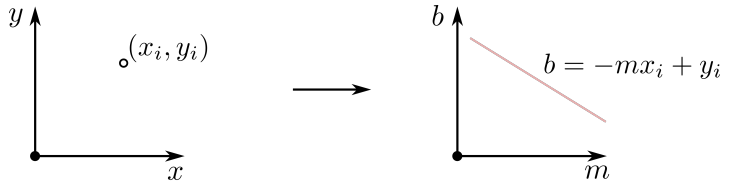


Figure 11.3: Each point (x_i, y_i) in the original space maps to a *line* in the Hough space which describes all possible parameters m and b that would generate a line passing through the point (x_i, y_i).

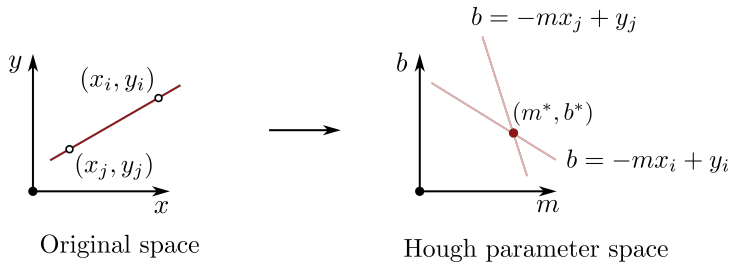


Figure 11.4: All points on a line in the original space yield lines in the Hough space that intersect at a common point.

This concept can be applied to line segmentation by searching in the Hough space for intersections among the lines that correspond to each point (x, y) in the set S. In practice, this can be done by discretizing the Hough space with a

grid and simply counting for each grid cell the number of lines corresponding to (x_i, y_i) points from S that pass through it. Local maxima among the cells then can be chosen as lines that "fit" the data set S.

However, performing a discretization of the Hough space requires a trade-off between range and resolution (in particular because m can range from $-\infty$ to ∞. Alternatively, it is possible to use a polar coordinate representation of the Hough space which defines a line as:

$$x \cos \alpha + y \sin \alpha = r,$$

where (α, r) are the new line parameters. With this representation, a point (x_i, y_i) from the original space gets mapped to the polar Hough space (α, r) as a sinusoidal curve (see Figure 11.5). An example of the Hough transform using the polar representation is given in Figure 11.6.

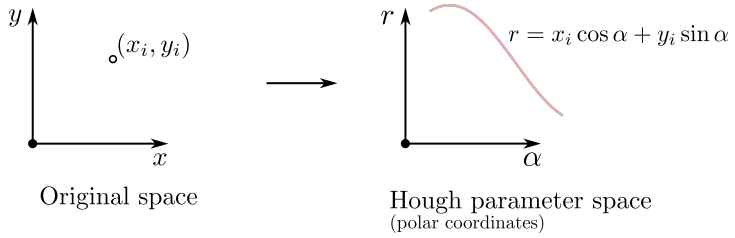


Figure 11.5: Representation of a point (x_i, y_i) in the Hough space when using a polar coordinate representation of a line with parameters α and r.

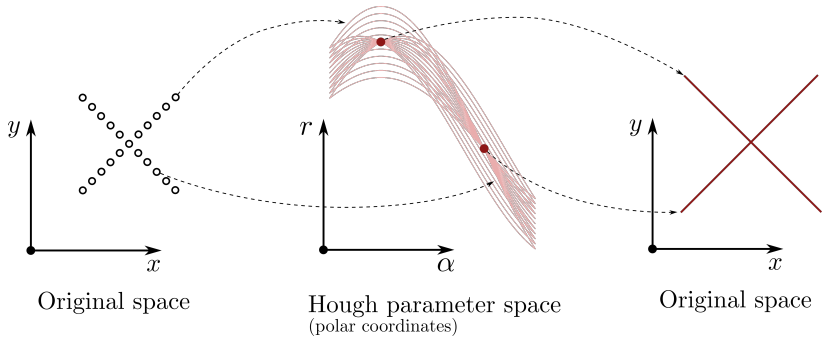


Figure 11.6: Example of the Hough transformation using a polar coordinate representation of lines.

11.1.2 *Line Fitting*

Line segmentation is the process of identifying which data points belong to a line, and line fitting is the process of estimating parameters of a line for those corresponding data points. For the line segmentation algorithms previously discussed (i.e. split-and-merge, RANSAC, and Hough-transform), a line associated with the segmented data points was also implicitly defined. However, the lines implicitly defined from the segmentation algorithms may not always be ideal and so other techniques have been developed to specifically address the line fitting task.

Line fitting algorithms search for lines that best fit a set of data points. In almost all cases the problem is over-determined (i.e. there are more data points

than parameters to choose) and noise in the data means that there is not perfect solution. Therefore one of the most common approaches to line fitting is based on *least-squares estimation*, which tries to find a line that minimizes the overall error in the fit. For this approach it is useful to work in polar coordinates defined by:

$$x = \rho \cos\theta, \quad y = \rho \sin\theta,$$

where (x, y) is the 2D Cartesian coordinate of a data point and (ρ, θ) is the 2D polar coordinate. In polar coordinates the equation of a line is given by

$$\rho \cos(\theta - \alpha) = r, \quad \text{or} \quad x \cos\alpha + y \sin\alpha = r, \tag{11.1}$$

where α and r are the parameters that define the line. For a visual representation of these definitions see Figure 11.7.

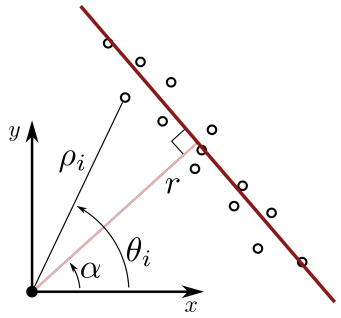


Figure 11.7: Representation of a line in polar coordinates, defined by the parameters r and α which are the distance and angle to the closest point on the line to the origin.

For a collection S of N points (ρ_i, θ_i), the error d_i corresponding to the perpendicular distance from a point to a line defined by parameters α and r can be computed by:

$$d_i = \rho_i \cos(\theta_i - \alpha) - r. \tag{11.2}$$

The line fitting task can then be formulated as an optimization problem over the parameters α and r to minimize the combined errors d_i for $i = 1, \ldots, N$. In particular, the combined errors are aggregated using a sum of the squared errors:

$$S(r, \alpha) = \sum_{i=1}^{N} d_i^2 = \sum_{i=1}^{N} (\rho_i \cos(\theta_i - \alpha) - r)^2. \tag{11.3}$$

This is a classic least squares optimization problem that can be efficiently solved. However, this cost function generally assumes that each of the data points is equally affected by noise (i.e. the uncertainty of each measurement is the same). In some cases it might be beneficial to account for differences in data quality for each point i, which could give preference to well known points.

Accounting for unique uncertainties in each data point leads to a *weighted least squares estimation* problem. In particular, it is assumed that the variance of each range measurement ρ_i is given by σ_i. The cost function (11.3) is then modified to be:

$$S_w(r, \alpha) = \sum_{i=1}^{N} w_i d_i^2 = \sum_{i=1}^{N} w_i (\rho_i \cos(\theta_i - \alpha) - r)^2, \tag{11.4}$$

where the weights w_i are given by:

$$w_i = \frac{1}{\sigma_i^2}.$$

It can be shown that the solution to the optimization problem defined by the

weighted cost function (11.4) is given by:

$$r = \frac{\sum_{i=1}^{N} w_i \rho_i \cos(\theta_i - \alpha)}{\sum_{i=1}^{N} w_i},$$
$$\alpha = \frac{1}{2}\text{atan2}\left(\frac{\sum_{i=1}^{N} w_i \rho_i^2 \sin(2\theta_i) - \frac{2}{\sum_{i=1}^{N} w_i}\sum_{i=1}^{N}\sum_{j=1}^{N} w_i w_j \rho_i \rho_j \cos\theta_i \sin\theta_j}{\sum_{i=1}^{N} w_i \rho_i^2 \cos(2\theta_i) - \frac{1}{\sum_{i=1}^{N} w_i}\sum_{i=1}^{N}\sum_{j=1}^{N} w_i w_j \rho_i \rho_j \cos(\theta_i + \theta_j)}\right) + \frac{\pi}{2}. \tag{11.5}$$

11.2 Object Recognition

Another high-level information extraction task that is common in robotics is *object recognition*. Object recognition is the task of classifying or naming discrete objects in the world (usually based on images or video). This can be a particularly challenging task because real world scenes are commonly made up of many varying types of objects which can appear at different poses and can occlude each other. Additionally, objects within a specific class can have a large amount of variability (e.g. breeds of dogs or car models). In this section three common methods for object recognition will be introduced, namely template matching, bag of visual words, and neural network methods.

11.2.1 Template Matching

Template matching[4] is a machine vision technique for identifying parts of an image that match a given image pattern[5]. This approach has seen success in a variety of applications, including manufacturing quality control, mobile robotics, and more. The two primary components needed for template matching are the source image I and a template image T

Given a source and template image, one approach to template matching is to leverage the linear spatial correlation filters discussed in the previous chapter. In particular, a naive approach would be to use the normalized template image as a filter mask in a correlation filter. By applying this filter mask to every pixel in the source image the resulting output would quantify the similarity of that region of the source image to the template. This type of approach is sometimes referred to as a *cross-correlation*. Another approach based on linear spatial filters from the previous chapter would be to leverage the similarity filters that compute the sum of absolute differences (SAD) metric for each pixel in the source image. Regions of the source image similar to the template would correspond to low SAD scores. The disadvantages of these approaches is that do not handle rotations or scale changes, which are quite common in real world applications.

One solution to the scaling issue in correlation filter based template matching is to simply re-scale the source image multiple times and perform template matching on each. This concept, referred to as using *image pyramids*[6], can also be used to accelerate object search by using a coarser resolution image first to localize the object and then using finer resolution images for actual detection.

[4] N. Perveen, D. Kumar, and I. Bhardwaj. "An overview on template matching methodologies and its applications". In: *International Journal of Research in Computer and Communication Technology* 2.10 (2013), pp. 988–995

[5] Advanced template matching algorithms allow finding pattern occurrences regardless of their orientation and local brightness.

[6] R. Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010

Building image pyramids can be accomplished in several ways. One naive approach is to simply eliminate some rows and columns of the image. Another approach is to first use a Gaussian smoothing filter to remove high frequency content form the image and *then* subsample the image. The sequence of images resulting from this approach is referred to as a *Gaussian pyramid*.

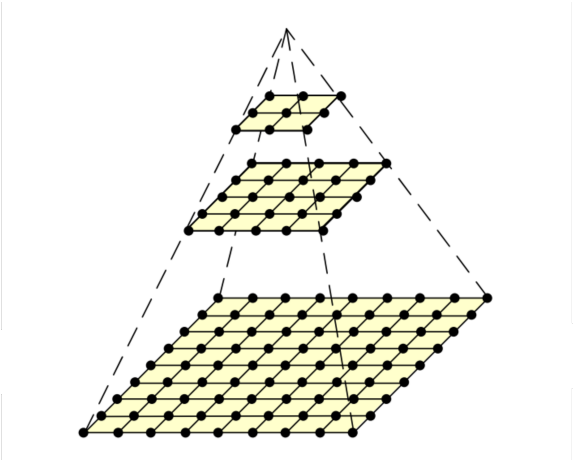


Figure 11.8: A traditional image pyramid: each level has half the resolution (width and height), and hence a quarter of the pixels, of its parent level. Figure from Szeliski (2010) .

11.2.2 Bag of Visual Words

The key idea behind the bag of visual words[7] approach is that object representations can be simplified by considering them as a collection of their subparts (e.g. a bike is an object with wheels, a frame, and handlebars), and the subparts are referred to as *visual words*. In this approach a source image is searched for *visual words*, and a distribution of visual words that are found in the image is created (in the form of a histogram). Object detection can then be performed by comparing this distribution to a set of training images. For example, suppose the source image contains a human face and the recognized features included eyes and a nose. Then by comparing the distribution to training images, it would likely be determined that training images that also have eyes and a nose are also images of faces.

[7] The model originated in natural language processing, where we consider texts such as documents, paragraphs, and sentences as collections of words - effectively "bags" of words.

11.2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) represent a relatively new and very powerful paradigm in object recognition. These approaches were first introduced in the field of computer vision for image recognition in 1989, and since then have significantly boosted performance in image recognition and classification tasks. Research in this field is also still very active.

11.3 Exercises

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

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

11.3.1 Line Extraction

Complete *Problem 2: Line Extraction*, where you will implement a line extraction algorithm (Split-and-Merge) to fit lines to simulated Lidar range data.

11.3.2 Template Matching

Complete *Problem 4: Template Matching*, where you will explore the use of the classic template matching algorithm, implemented in the open-source OpenCV library.

11.3.3 Image Pyramids

Complete *Extra Problem: Image Pyramids*, where you will learn about how template matching algorithms can be enhanced through the use of image pyramids (and image filtering).

12
Modern Computer Vision Techniques

Machine learning has become an extremely powerful tool in the realm of computer vision, particularly through the use of convolutional neural networks (CNNs), Transformers, and the increased availability of data and computational resources. This chapter introduces the fundamentals of CNNs, Transformers, as well as their application to computer vision and robotics.

Modern Computer Vision

Modern computer vision techniques[1] rely heavily on deep learning. The most common architectures are convolutional neural network [2] and Transformers [3]. A convolutional neural network is a type of neural network with additional structure that is beneficial for image processing tasks. In fact, CNNs can be said to be "regularized" neural networks since the additional structure reduces the ability of the network to overfit to data. Transformers, which include less constraints in their architectures, scale up better with more data and compute resources. This chapter will first introduce each component[4] in the architecture of a CNN, then discuss building blocks[5] of a Transformer, and finally discuss how CNNs/Transformers can be applied to problems in robotics.

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

[2] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016

[3]

[4] Convolutional layers, nonlinear activations, pooling layers, and fully-connected layers.

[5] self-attention layers, and multi-layer perceptrons

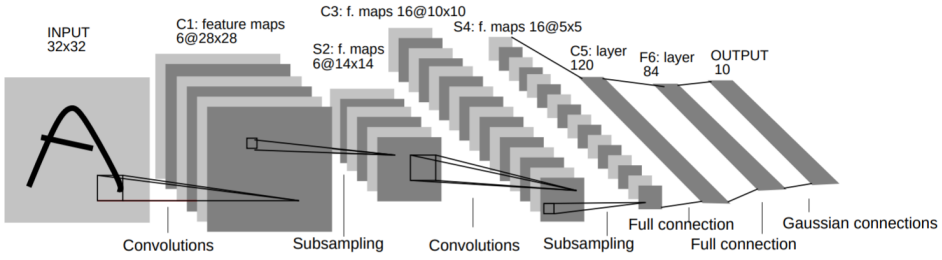


Figure 12.1: Example convolutional neural network architecture from LeCun et al. (1998).

12.1 Convolutional Neural Networks

12.1.1 Convolution Layers

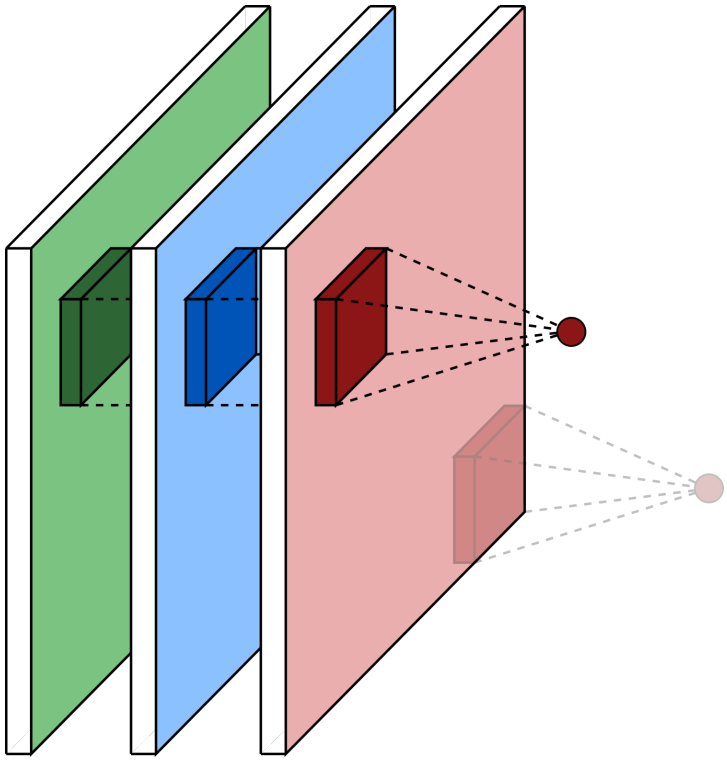


Figure 12.2: A convolution filter being applied to a 3-channel RGB image.

One of the main structural concepts that is unique to the architecture of a CNN is the use of convolution layers. These layers exploit the underlying *spatial locality* structure in images by using sliding "learned" filters, which are often much smaller than the image itself. Mathematically these filters perform operations in a similar way as other linear filters that have been used in image processing, such as Gaussian smoothing filters, and can be expressed as affine functions:

$$f(x) = w^\top x + b,$$

where w is a vectorized representation of the weight parameters that define the filter, x is a vectorized version of the image pixels covered by the filter, and b is a scalar bias term. For example in Figure 12.2 a filter is applied over an image with three color channels (red, green, blue). In this case the filter may have dimension $m \times n \times 3$, which could be vectorized to a weight vector w with $3mn$ elements. Additionally, the *stride* of the filter describes how many positions it shifts by when sliding over the input. The output of the filter is also passed through a nonlinear activation, typically a ReLU function.

Once the filter has been applied to the entire image, the collection of outputs

from the activation function will create a new "filtered image" typically referred to as an *activation map.* In practice a number of different filters are usually

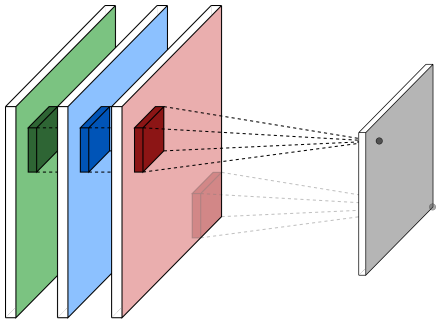


Figure 12.3: The outputs of a convolution filter and activation function applied across an image make up a new image, called an *activation* map.

learned in each convolution layer, which would simply produce a corresponding number of activation maps as the output[6]. This is crucial such that each filter can focus on learning one specific relevant feature. Examples of different filters that might be learned in different convolution layers of a CNN are shown in Figure 12.4[7]. Notice that the low-level features which are learned in earlier convolution layers look a lot like edge detectors (i.e. are more basic/fundamental) while later convolution layers have filters that look more like actual objects.

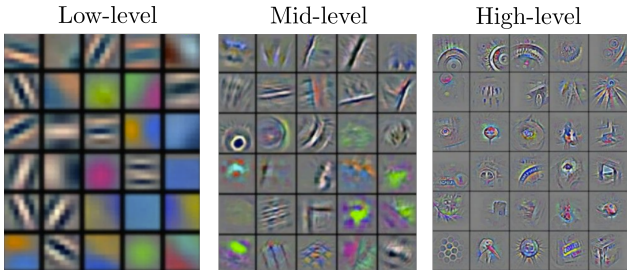


Figure 12.4: Low-level, mid-level, and high-level feature visualizations in a convolutional neural network from Zeiler and Fergus (2014).

In general, the use of convolution layers to exploit the spatial locality of images provides several benefits including:

1. *Parameter sharing*: the (small) filter's parameters are applied at all points on the image. Therefore the total number of learned parameters in the model is much smaller than if a fully-connected layer was used.

2. *Sparse interactions*: having the filter be smaller than the image allows for better detection of smaller, more meaningful features and improves computation time by requiring fewer mathematical operations to evaluate.

3. *Equivariant representation:* the convolution layer is equivariant to translation, meaning that the convolution of a shifted image is equivalent to the shifted convolution of the original image[8].

4. The ability to work with images of *varying size* if needed.

[6] 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.

[7] M. D. Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks". In: *European Conference on Computer Vision (ECCV)*. Springer, 2014, pp. 818–833

[8] However, convolution is not equivariant to changes in scale or rotation.

12.1.2 *Pooling Layers*

Pooling is the second major structural component in CNNs. Pooling layers typically come after convolution layers (and their nonlinear activation functions). Their primary function is to replace the output of the convolution layer's activation map at particular locations with a "summary statistic" from other spatially local outputs. This helps make the network more robust against small translations in the input, helps improve computational efficiency by reducing the size of the input (i.e. it lowers the resolution), and is useful in enabling the input images to vary in size[9]. The most common type of pooling is *max* pooling, but other types also exist (such as *mean* pooling).

Computationally, both max and mean pooling layers operate with the same filtering idea as in the convolution layers. Specifically, a filter of width m and height n slides around the layer's input with a particular stride. The difference between the two comes from the mathematical operation performed by the filter, which as their names suggest are either a maximum element or the mean over the filter. If the output of the convolution layer has N activation maps, the output of the pooling layer will also have N "images", since the pooling filter is only applied across the spatial dimensions.

[9] The size of the pooling can be modified to keep the size of the pooling layer output constant.

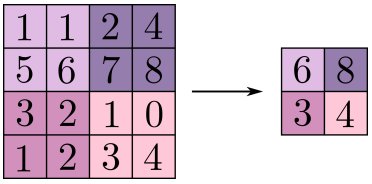


Figure 12.5: Max pooling example with 2 $\times$ 2 filter and stride of 2.

12.1.3 *Fully Connected Layers*

Downstream of the convolution and pooling layers are fully connected layers. These layers make up what is essentially just a standard neural network, which is appended to the end of the network. The function of these layers is to take the output of the convolution and pooling layers (which can be thought of as a highly condensed representation of the image) and actually perform a classification or regression. Generally the total number of fully connected layers at the end of the CNN will only make up a fraction of the total number of layers.

12.1.4 *CNN Performance*

A CNN can be said to learn how to process images *end-to-end* because it essentially learns how to perform two steps simultaneously: feature extraction and classification or regression (i.e. it learns the entire process from image input to the desired output). In contrast, classical approaches to image processing use hand-engineered feature extractors. Since 2012, the performance of end-to-end learning approaches to image processing have dominated and continue to improve[10]. This continuous improvement has generally been realized with the use of deeper networks.

[10] Of course in some specific applications hand-engineered features may still be better! For example if engineering insight can identify a structure to the problem that a CNN could not.

12.2 Transformers

12.2.1 Data Representation

The data that Transformers operate on is a set or sequence of N tokens of dimension D. These tokens can be collected into a matrix X^0 of dimension $N \times D$. According to the data modality, different tokenization approaches are needed:

1. *Text:* organized into a sequence of words where each word is represented by a token. The word-to-token mapping is performed through a learnable directionary lookup.

2. *Image:* organized into a sequence of patches where each patch is represented by a token. The patch-to-token mapping is achieved by a fully connected layer.

Compared to CNNs, Transformers enforce less structure constraints—as long as data can be organized into a set or sequence, it can be processed by Transformers. Indeed, the internal structure is purely learned from data by Transformers. In other words, we incorporate less domain knowledge into the deep learning algorithm, which makes it easier to generalize to different data modality.

12.2.2 Transformer Block

Transformers iteratively apply multiple identical Transformer block T^i into the input sequence X^i:

$$X^{i+1} = T^i(X^i),$$

where X^{i+1} is the output sequence. The length and feature dimensions of the input sequence are typically unchanged after each Transformer block. Here, X^i and X^{i+1} have the same dimensions $N \times D$. Each Transformer block consists of two consequent operations: The first operation (self-attention) is applied across the sequence to refine features based on the relationships between tokens across the sequence. The second operation (multi-layer perceptron) is applied independently on each token for further feature learning.

Self-Attention Layers Self-Attention Layers (SA) are the most important layers in Transformers. The key operation in these layers is *attention* which performs feature correlation and aggregation:

$$\text{Attention}(Q_i, K_i, V_i) = \text{softmax}(\frac{Q_i K_i^T}{\sqrt{d_k}})V_i,$$

where Q_i, K_i, V_i are either identical copies of X_i or embeded from X_i by additional linear layers. Here, Q_i, K_i, V_i are organized as a set of tokens, similarly to the input token lists.

Multi-Layer Perceptrons Multi-Layer Perceptions (MLP) are a sequence of layers, alternating fully connected layers and non-linearity, that perform non-linear feature transformation of each individual token:

$$\mathbf{x}_i^j = \text{MLP}_\theta(\mathbf{x}_{i-1}^j),$$

where $\mathbf{x}_i^j$ is each individual token of X_i.

12.3 Modern Object Detection and Localization

Modern computer vision techniques such as Convolutional Neural Networks and Vision Transformers have a large variety of applications in robotic perception, including object localization and detection.

12.3.1 CNN-based Object Detection

In object localization problems, the goal is to identify the position of an object in the image. This is usually accomplished by specifying four numbers that define a *bounding box* for the object in the image[11] (see Figure 12.6). To solve object localization problems with a CNN, the standard approach is to have the output of the network be both the bounding box coordinates and an object class. This can be accomplished by reusing the convolution and pooling layers of the CNN but then have two separate branches of the fully connected layers: one trained for classification and the other for localization[12]. To train a network to simultaneously perform classification and localization, a multi-task loss function can be defined by adding together the loss functions of each branch.

[11] Box coordinates are usually the (x, y) position of the top-left corner and the width w and height h of the box.

[12] Since the output of the localization branch is four real numbers (x, y, w, h), this would be considered a regression problem and it is common to use and l_2 loss function.

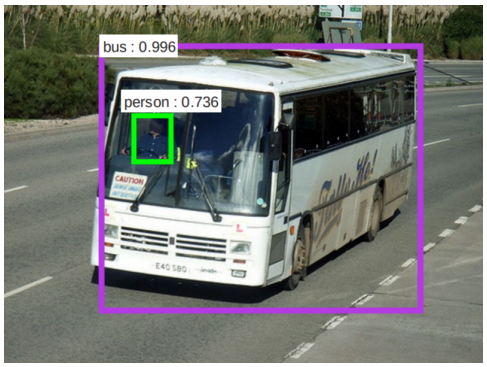


Figure 12.6: Bounding box prediction for several objects in an image from Ren, He, et al. (2017)

If multiple objects exist within a single image the object localization and classification problem becomes more difficult. First, the number of outputs of the network may change! For example, outputting a bounding box for n objects would require $4n$ outputs. A practical solution to handling the problem of varying outputs is to simply apply the CNN to a series of cropped images produced from the original image, where the network can also classify the image as "background" (i.e. not an object)[13]. However, a naive implementation of this

[13] This could be thought of as applying the entire CNN as a filter that slides across the image.

idea would likely result in an excessive number of CNN evaluations. Instead, different approaches have been developed for making this idea efficient by reducing the number of areas in the image that need to be evaluated. For example this has been accomplished by identifying "regions of interest" in the image through some other approach, or even partitioning the image into a grid.

12.3.2 *Transformer-based Object Detection*

Very recently, Transformers have been adapted to tackle object detection and shows several benefits over CNN-based models, pinoeered by DETR from Carion, Massa, et al. (2020) . DETR presents object detection as a direct set prediction problem, largely streamlining the detection pipeline. DETR removes many hand-designed components including "region proposal" and "non-maximum suppression" that are commonly used in CNN-based models. DETR, in its most basic incarnation, is a end-to-end Transformer model that takes in images as inputs and predicts a fixed set of potential bounding boxes.

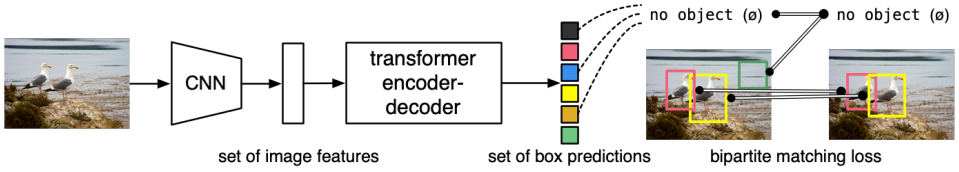


Figure 12.7: Detection Transformers (DETR) from Carion, Massa, et al. (2020)

In details, DETR uses a conventional CNN backbone to learn 2D feature maps from an input image, as shown in the left of 12.8. Then, the model 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 (a stack of SAs and MLPs) 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, whose dimension is $M \times D$, encode the object information (e.g. size,location, and category) distribution over an image. They are jointly optimized with the neural network parameters to learn dataset priors. Each object represents a potential object bounding box in the image of being processed.

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. Notably, a "no object" label is assigned to those queries without true objects detected, allowing the model to handle 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 proposed to produce an optimal bipartitle matching between predicted and ground-truth objects, followed by

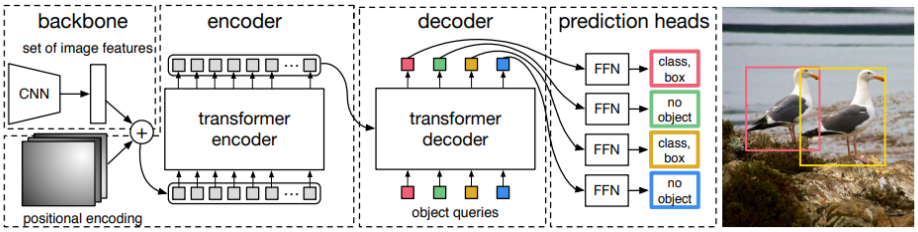


Figure 12.8: Detection Transformers (DETR) details from Carion, Massa, et al. (2020)

optimizing object-centric (bounding box) losses. To find a bipartite matching between these two sets, the algorithms first obtains a permutation of N elements with the lowest cost:

$$\hat{\sigma} = \underset{\sigma \in (\mathcal{S}\rangle\}\Updownarrow\dashv_N)}{\arg\min} \sum_{i}^{N} \mathcal{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)}), \tag{12.1}$$

where $\mathcal{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)})$ denotes a pair-wise matching cost between a ground-truth y_i and a prediction $\hat{y}_{\sigma(i)}$ and $\sigma(i)$ is the index of a matched prediction box.

check hy above

Having obtained the optimal object matching, a final pairwise loss is defined simiarly to the losses of common object detectors, including a negative log-likelihood for category prediction and a regression loss for size and center:

$$\mathcal{L}_{\text{detection}}(y, \hat{y}) = \sum_{i=1}^{N} \left[-\log \hat{p}_{\hat{\sigma}(i)}(c_i) + \mathbb{1}_{c_i \neq \varnothing} \mathcal{L}_{\text{box}}(b_i, \hat{b}_{\hat{\sigma}}(i)) \right], \tag{12.2}$$

where $\hat{\sigma}$ denotes the optimal assignment from the previous step and $\varnothing$ is the "no object" label.

Part III

Robot Localization

13
Introduction to Localization and Filtering

We have already discussed the robot motion planning problem as well as several common algorithms for motion planning, including optimal control and sampling-based methods. In these algorithms we generally assume that information about the current robot state is available, for example to be used in closed-loop control laws for feedback or in trajectory optimization problems to specify the initial conditions. In practice however this information has to be *estimated* from sensor measurements.

Robot perception, which we also discussed in previous chapters, is a foundational problem for sensing the environment and extracting useful semantic information that can be helpful for estimating the robot's state and the state of the world it is operating in. While methods from robot perception can provide instantaneous *local* information that is crucial for robots to navigate autonomously, they often need to be supplemented with additional *global* information for comprehensive autonomy. For example, distance measurements from a laser rangefinder might be useful for detecting objects in an environment, but they only provide information *relative* to the robot's current position. This is very useful for collision avoidance but is likely not sufficient for solving a full motion planning problem. Computer vision algorithms can also be used for object detection, but the information is limited to what is in the robot's current view and may not provide enough information to fully understand the current state of the world.

Robot localization and mapping is another fundamental problem in robot autonomy that bridges the gap between robot perception and motion planning and control[1]. Specifically, the primary goal of localization and mapping is to synthesize instantaneous *local* sensor measurements into a more *global* estimate and understanding of the robot's state and the world it is operating in. In this chapter we introduce the problem of *localization*, which specifically aims to give the robot the ability to understand its current state with respect to its environment in a global sense[2,3], such as the robot's position with respect to a global coordinate system or a map of the environment. For example consider the environment map given by the floor plan in Figure 13.1. Before a robot can navigate to a particular room it must know where in the building it is currently located.

[1] The localization and mapping problem is one of the core components of the "think" part of the "see, think, act" cycle.

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

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

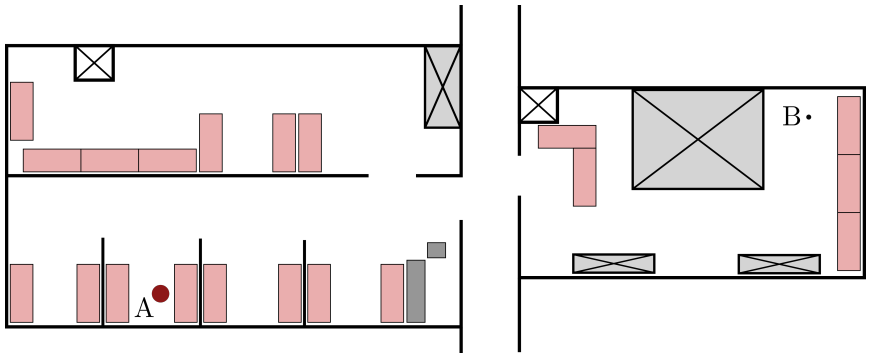


Figure 13.1: An example environment where localization is crucial for robotic autonomy. For a robot to move from location A to location B it must first understand which room it is in, and that the only path to B is through the hallway. Extracting such global information about the environment from local measurements (e.g. from a range sensor) requires specialized algorithms.

One fundamental practical challenge for robot localization is that there is inherent *uncertainty* due to limited or noisy sensor information. We therefore reason about the problem in a *probabilistic* fashion and try to compute a *belief representation* of the robot's state. In practice the belief representation can be as simple as a point estimate, but it is more powerful to use a probability distribution over states. The advantage of representing the belief as a probability distribution is that it can be used to extract point estimates as well as measures of uncertainty, which can be important for downstream motion planning and control tasks. For example if the robot has a very uncertain belief of its current position we can leverage the distribution for motion planning to reduce the probability of collisions or even to plan information gathering actions that could decrease the uncertainty.

In the following chapters we will introduce different probabilistic filtering[4] algorithms that are used in many contexts in robotics such as localization, as well as in numerous other fields of engineering. But first, in this chapter we will introduce some preliminary concepts that form the basis of these algorithms, including random variables, probability distributions, conditional probabilities and Bayes' rule, and Markov models. We will also introduce a canonical probabilistic filter known as the Bayes filter[5].

[4] The general process of extracting useful information from a noisy signal is often referred to as *filtering*.

[5] The Bayes filter is also referred to as *recursive Bayesian estimation*.

13.1 Preliminary Concepts in Probability

13.1.1 Random Variables

We model uncertain quantities such as sensor measurements, robot states, and environment variables as discrete or continuous *random variables*.

Definition 13.1.1 (Discrete Random Variable)**.** A *discrete random variable*, X, is a variable that can only take on values from a countable set. The probability that the random variable takes on a specific value, x, is denoted by $p(X = x)$ or the shortened notation $p(x)$. We also refer to $p(x)$ as the *probability mass function*, which must satisfy:

$$\sum_x p(x) = 1,$$

where the summation is over all possible values of X.

Definition 13.1.2 (Continuous Random Variable)**.** A *continuous random variable*, X, is a variable that can take on values in a continuous range. For continuous random variables $p(x)$ represents the *probability density function*[6] that satisfies:

$$\int_{-\infty}^{\infty} p(x)dx = 1.$$

[6] By convention the probability that a continuous random variable is a specific value is zero, and we only have non-zero probability mass for an interval of the probability density function.

The probability of the continuous random variable taking on a value in the interval $[a, b]$ is:

$$P(a \leq X \leq b) = \int_{a}^{b} p(x)dx.$$

Example 13.1.1 (Discrete and Continuous Random Variables)**.** A common example of a discrete random variable is the result of a coin flip, which can only take on two values: heads or tails. For a fair coin, the probability mass function would be:

$$p(\text{heads}) = \frac{1}{2}, \quad p(\text{tails}) = \frac{1}{2}.$$

A common example of a continuous random variable in robotics is the pose of the robot, which could take on an infinite number of 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[7], which is parameterized by a mean and variance. We illustrate several examples of probability distributions in Figure 13.2.

[7] The Gaussian distribution is also known as the Normal distribution.

The GMM figure should be updated.

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 applications of probability theory involve more than one random variable. In these instances it is useful to quantify probabilities associated with multiple random variables at the same time. The distribution over pairs of random variable outcomes is referred to as the *joint distribution*.

Definition 13.1.3 (Joint Distribution)**.** The *joint distribution* of two random variables X and Y defines the probability associated with both taking on specific values at the same time. Mathematically the joint distribution is denoted by $p(X = x \text{ and } Y = y)$ or shortened to simply $p(x, y)$.

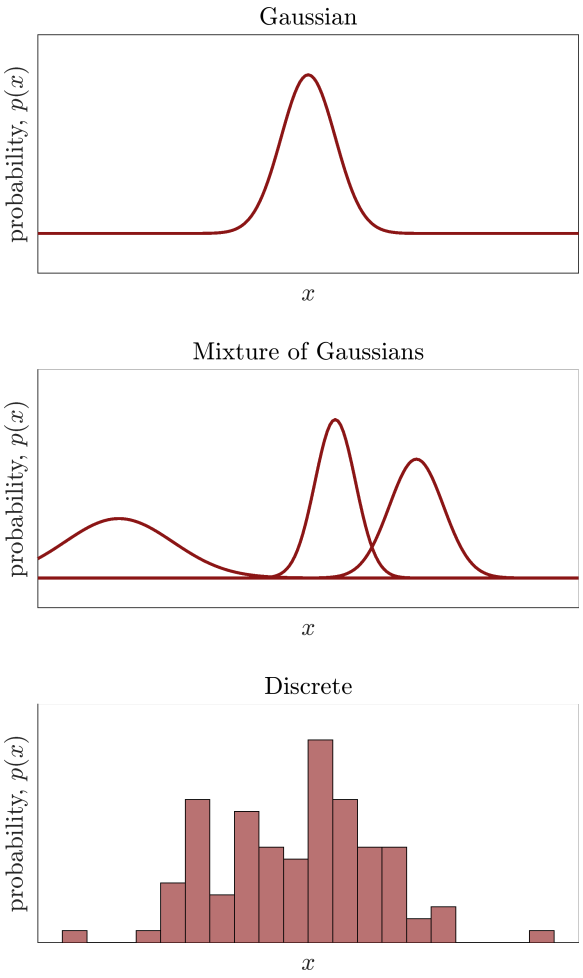


Figure 13.2: A graphical representation of different types of probabilistic representations: (a) a continuous unimodal Gaussian distribution, (b) a continuous multimodal distribution from a mixture of Gaussians, (c) a discrete representation with a finite number of possible values.

Random variables can also be related to each other. For example the random variables X = "today is cloudy" and Y = "today it is raining" are related to each other since it can't be raining if there are no clouds. Two random variables that do not have any influence on each other in a probabilistic sense are considered to be probabilistically independent.

Definition 13.1.4 (Probabilistic Independence)**.** Two random variables X and Y are probabilistically *independent* if and only if:

$$p(x, y) = p(x)p(y). \tag{13.1}$$

Independence holds when the occurrence of one value of a random variable does not affect the probability of another random variable taking on a specific value.

Example 13.1.2 (Probabilistically Independent Sensor Measurements)**.** Consider a proximity sensor and a temperature sensor with measurements represented by the random variables X and Y, respectively. X can take values

in $\{\text{close}, \text{medium}, \text{far}\}$, and Y in $\{\text{low}, \text{med}, \text{high}\}$. Suppose $p(X = \text{close}) = \frac{1}{3}$ and $p(Y = \text{med}) = \frac{1}{5}$. Assuming that the two sensors are independent, we can compute the joint probability:

$$p(X = \text{close and } Y = \text{med}) = \frac{1}{15}.$$

The *conditional probability* is another useful tool that relates two random variables when the value of one of the variables is known or fixed.

Definition 13.1.5 (Conditional Probability)**.** The *conditional probability* of a random variable X having a certain outcome x given that a second random variable Y had the outcome y is defined as:

$$p(x \mid y) := \frac{p(x, y)}{p(y)}. \tag{13.2}$$

We typically say this is the probability of X having outcome x *conditioned* on the fact that Y had the outcome y.

Notice that if the random variables X and Y are independent that the conditional probability is simply $p(x \mid y) = p(x)$. This reflects that the knowledge of Y taking having the outcome y provides no new information about the random variable X.

Example 13.1.3 (Sensor Conditional Probabilities)**.** Consider the sensor scenario from Example 13.1.2, but additionally let's now consider an obstacle detection random variable Z, which can have an outcome from the set $\{\text{detected}, \text{not detected}\}$. Let's assume $p(x) = 1/3$ and $p(y) = 1/3$ for any outcome of X and Y and also let's assume that the probability of detecting an obstacle depends on the sensor's proximity to it. Specifically, let's assume the conditional probabilities of detection given the proximity sensor measurements are:

$$\begin{aligned} p(Z = \text{detected} \mid X = \text{close}) &= 5/6, \\ p(Z = \text{detected} \mid X = \text{medium}) &= 1/3, \\ p(Z = \text{detected} \mid X = \text{far}) &= 1/5. \end{aligned}$$

Using the definition of conditional probability we can now compute the joint probability that we detect an obstacle and the proximity sensor provides a reading saying "close":

$$p(Z = \text{detected and } X = \text{close}) = p(Z = \text{detected} \mid X = \text{close}) p(X = \text{close}) = 5/18.$$

We also define another important notion of independence for two random variables when they are *conditioned* on a third random variable.

Definition 13.1.6 (Conditional Independence)**.** Two random variables X and Y are *conditionally independent*[8] given the outcome of a third random variable Z if and only if:

$$p(x, y \mid z) = p(x \mid z) p(y \mid z). \tag{13.3}$$

[8] Conditional independence does not imply independence, and vice versa.

13.1.4 Law of Total Probability

The law of total probability defines a relationship between probabilities, joint probabilities, and conditional probabilities.

Definition 13.1.7 (Law of Total Probability)**.** For discrete random variables X and Y the *law of total probability* states that:

$$p(x) = \sum_y p(x, y) = \sum_y p(x \mid y)p(y),$$

and for continuous random variables:

$$p(x) = \int p(x, y)dy = \int p(x \mid y)p(y)dy.$$

In words, this law says that the probability of a random variable X having outcome x can be found by looking at the joint probabilities between X and Y and accounting for *all* possible values of Y. The second part of the law is a direct result of applying the definition of conditional probabilities. We commonly refer to the process of extracting $p(x)$ from the joint probabilities $p(x, y)$ as *marginalizing*, and refer to $p(x)$ as a *marginal probability*.

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 the joint probability can be expressed 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*[9]:

[9] 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[10] $p(x)$ 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 $p(x \mid y)$[11]. Bayes' rule also extends to cases with additional random variables. For example with three random variables X, Y, and Z, Bayes' rule is:

$$p(x \mid y, z) = \frac{p(y \mid x, z)p(x \mid z)}{p(y \mid z)}.$$

[10] 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.

[11] This new distribution, which is obtained by updating the prior distribution $p(x)$ with the new information about Y, is commonly referred to as the *posterior* probability.

Example 13.1.4 (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 are very descriptive in the sense that they define a probability associated with every outcome of a random variable. However sometimes it is also useful to aggregate this information into a more compact summary or statistic. Two of the most commonly used values are the *expected value* and the *covariance*.

Definition 13.1.9 (Expected Value)**.** The expected value[12] for a random variable X is denoted as $E[X]$. For discrete random variables the expected value is computed by:

$$E[X] = \sum_x xp(x),$$

where the sum is over the possible outcomes of X. Similarly, the expected value for a continuous random variable is computed by:

$$E[X] = \int xp(x)dx.$$

[12] The expected value is also referred to as the *mean* or *first moment* of a distribution.

We can think of the expected value as the average outcome over an infinite number of samples. One nice property of expectation is that it is a linear operator such that:

$$E[aX + b] = aE[X] + b,$$

for any values $a, b \in \mathbb{R}$. For vector-valued random variables the expectation of the random vector is simply the vector of expectations of each element.

Definition 13.1.10 (Covariance)**.** The *covariance* between two random variables X and Y is denoted $\text{cov}(x, y)$ and is computed by:

$$\begin{aligned}\text{cov}(X, Y) &= E[(X - E[X])(Y - E[Y])^\top], \\ &= E[XY^\top] - E[X]E[Y]^\top.\end{aligned}$$

Covariance is a quantity used to describe the relationship between random variables and is positive if greater values of one variable generally correspond to greater values of the other similarly if lesser values of one correspond to lesser value of the other. A negative covariance means the variables tend to show opposite behavior relative to each other. If there is no general relationship between the two, such as for independent variables, then their covariance is zero.

13.2 Markov Models

In Chapter 1 we showed how a robot's physical motion is modeled by analyzing its kinematics and dynamics, and how the resulting model in Equation (1.1) is a set of first order differential equations that describe how the state $\boldsymbol{x}$ changes in time given the current state and the control input $\boldsymbol{u}$. In this section we introduce *Markov models*, which we will also use to describe changes in a state $\boldsymbol{x}$ over time, but in a probabilistic fashion. Markov models are commonly used in a variety of robotics applications including localization as well as in higher level planning tasks.

Similar to Chapter 1 we will define the state $\boldsymbol{x} \in \mathbb{R}^n$ of the Markov model as a collection of variables that are relevant to the task at hand. In motion planning and control tasks the state usually defines the physical state of the robot, such as its pose and velocity, but in localization tasks or higher-level planning tasks the state may also contain other variables, such as information about the positions and features of objects in the robot's environment. Markov models are also generally expressed in discrete time, so we will use the notation $\boldsymbol{x}_t$ for the state at time t rather than $\boldsymbol{x}(t)$. We will also define the notation $\boldsymbol{x}_{t_1:t_n} \coloneqq \boldsymbol{x}_{t_1}, \boldsymbol{x}_{t_2}, \ldots, \boldsymbol{x}_{t_n}$ for describing a sequence of states between times t_1 and t_n. For control inputs and measurements[13] at time t we will use the notation $\boldsymbol{u}_t$ and $\boldsymbol{z}_t$, respectively, and for control and measurement sequences we will use $\boldsymbol{u}_{t_1:t_n} \coloneqq \boldsymbol{u}_{t_1}, \boldsymbol{u}_{t_2}, \ldots, \boldsymbol{u}_{t_n}$ and $\boldsymbol{z}_{t_1:t_n} \coloneqq \boldsymbol{z}_{t_1}, \boldsymbol{z}_{t_2}, \ldots, \boldsymbol{z}_{t_n}$.

[13] Measurements can come from any number of sensors, including those discussed in the previous chapters on robotic perception such as cameras and laser rangefinders.

One of the key differences between Markov models and the dynamics models from Chapter 1 is that Markov models are probabilistic and not deterministic. This makes Markov models well suited for tasks such as localization where modeling uncertainty is crucial. We start by defining a general probabilistic model by:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{0:t-1}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}), \tag{13.5}$$

which defines a posterior probability distribution over the possible current state $\boldsymbol{x}_t$ given the state, measurement, and control *histories*. Note that the convention we use here is that the robot executes the control $\boldsymbol{u}_t$ first, and then the measurement $\boldsymbol{z}_t$ is made based on the resulting state $\boldsymbol{x}_t$. For example for the very first time step we have $p(\boldsymbol{x}_1 \mid \boldsymbol{x}_0, \boldsymbol{z}_1, \boldsymbol{u}_1)$ which describes the probability distribution for the next state given the previous state, the control input, and the measurement after taking the control from the previous state. We also define a general

probabilistic measurement model as:

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_{0:t}, \boldsymbol{z}_{1:t-1}, \boldsymbol{u}_{1:t}). \tag{13.6}$$

In many applications the state $\boldsymbol{x}$ is defined such that it is *complete*, which means that the states prior to $\boldsymbol{x}_t$ do not influence the states after time t. In other words, $\boldsymbol{x}_t$ contains a sufficient amount of information that the history of the states, measurements, and controls before time t is not required to define the transition model. This is known as the *Markov property*, and if the Markov property holds the probabilistic model in Equation (13.5) simplifies to:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t), \tag{13.7}$$

and the measurement model in Equation (13.6) simplifies to:

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_t). \tag{13.8}$$

It is common in robotics applications to leverage models with the Markov property, and we generally refer a combination of a state transition probability model from Equation (13.7) and a measurement model from Equation (13.8) as a *Markov model*[14]. Figure 13.3 depicts a graphical representation of a Markov model where we can more clearly see how the control comes first and then a measurement is taken.

[14] Markov models are also sometimes referred to as *partially observable Markov models* since the state is not necessarily fully observable from the measurements. When the system is not controlled by any inputs $\boldsymbol{u}$ this model is often referred to as a hidden Markov model (HMM) where the "hidden" also refers to the lack of full observability of the state.

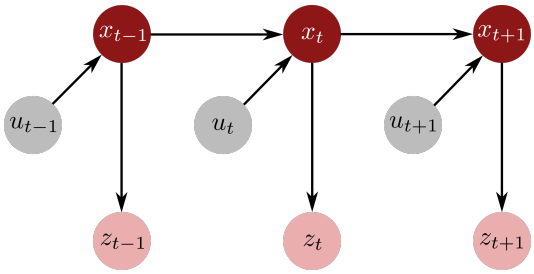


Figure 13.3: Graphical representation of the Markov model. Note that the sequencing assumes that the control is applied, and then a measurement is taken.

13.3 Bayes Filter

Robot localization is a classic filtering problem[15] 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 distribution*, which is a probability distribution over $\boldsymbol{x}_t$. Mathematically the belief distribution is denoted as $\text{bel}(\boldsymbol{x}_t)$ and is defined as:

$$\text{bel}(\boldsymbol{x}_t) := p(\boldsymbol{x}_t \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}). \tag{13.9}$$

In other words, the belief $\text{bel}(\boldsymbol{x}_t)$ is a posterior probability distribution over the state conditioned on the available history information. We also define a

[15] Localization is one instance of the more general filtering problem referred to as *state estimation*.

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

We outline the Bayes filter algorithm in Algorithm 13.1. The inputs for each step are the belief from the previous state[16] 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. The term η in the correction step is simply a normalization constant that ensures the resulting posterior $\text{bel}(\boldsymbol{x}_t)$ satisfies the requirements of a probability density function[17].

[16] In practice we typically initialize the prior distribution $\text{bel}(\boldsymbol{x}_0)$ using a best guess or simply a uniform distribution.

[17] The normalization constant comes from the denominator in Bayes' rule.

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**
  $\overline{\text{bel}}(\boldsymbol{x}_t) = \int p(\boldsymbol{x}_t \mid \boldsymbol{u}_t, \boldsymbol{x}_{t-1}) \text{bel}(\boldsymbol{x}_{t-1}) d\boldsymbol{x}_{t-1}$
  $\text{bel}(\boldsymbol{x}_t) = \eta p(\boldsymbol{z}_t \mid \boldsymbol{x}_t) \overline{\text{bel}}(\boldsymbol{x}_t)$
return $\text{bel}(\boldsymbol{x}_t)$

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 13.3 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*

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 fundamental concept of the Bayes filter is a great starting point to derive many useful algorithms, but the continuous Bayes filter is itself often not practical to implement. In particular the integrals in Algorithm 13.1 are not often easily computed, and approximating the integrals with a numerical scheme would be computationally challenging. The discrete Bayes filter is more tractable because we are working with a finite set of parameters for the probability mass

function and the summations can be efficiently computed. However even for the discrete case we might be practically limited to problems with small state spaces. Additionally, many problems in robotics, such as robot localization, inherently have continuous state spaces and discretizing the state space would be challenging. In the next chapter we will leverage the same underlying concepts used to derive the Bayes filter to explore more practical filtering algorithms.

Add exercises!

14
Parametric Filters

In the previous chapter we introduced fundamental tools from probability theory which are the basis of a probabilistic algorithmic framework for robot localization and state estimation. We also introduced the Bayes filter, which is a foundational algorithm for recursively updating a probability distribution over possible states, referred to as a *belief distribution*, given new noisy measurements. While the Bayes filter is generally intractable to implement in practice, it provides a mathematical foundation for developing more tractable algorithms based on approximations or by exploiting structure that limits the complexity of the problem.

In this chapter we will focus on tractable algorithms for probabilistic localization and state estimation that leverage the structure of *parametric* belief distributions. Parametric distributions are probability distributions that are fully specified by a fixed number of parameters, for example Gaussian distributions are defined by the mean and covariance. An example of a *non-parametric* distribution[1] would be a probability mass function that is defined by a probability for each outcome. Because of their structure parametric distributions are often represented by a much smaller set of values than non-parametric distributions.

Parametric filters[2] are a family of algorithms useful for robot localization and state estimation that model the robot's belief with a parametric distribution. Since parametric distributions are usually defined by a relatively small number of parameters they can generally be updated more efficiently that non-parametric distributions like the Bayes filter. For example, a filter that models the belief as a Gaussian distribution in one dimension only has to update two parameters, the mean and standard deviation, rather than an infinite-dimensional probability density function. In this chapter we introduce the *Kalman filter* family of parametric filters, which use a multivariate Gaussian distribution to represent the belief distribution. We will first introduce the original *Kalman filter*, which assumes a linear state transition model, and then we will show how this can be extended to more general nonlinear state transition models in the *extended Kalman filter* (EKF).

[1] The discrete Bayes filter, which we introduced in Chapter 13, is an example of a *non-parametric* filter since the belief distribution is defined by a probability for each state.

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

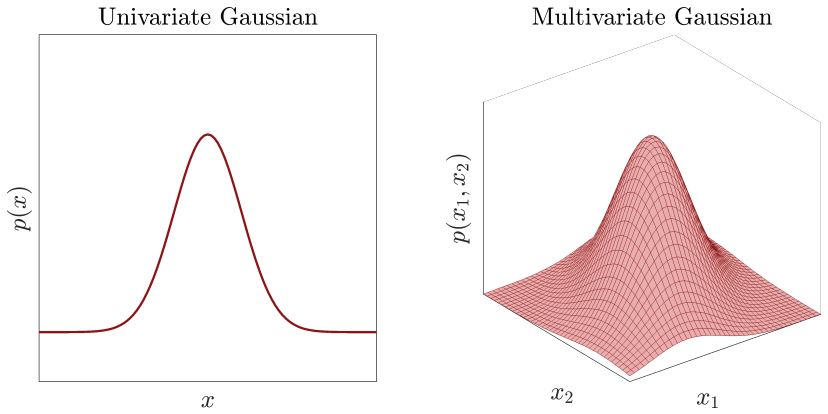


Figure 14.1: Univariate and multivariate Gaussian distributions.

14.1 The Gaussian Distribution

The Gaussian distribution, or *normal* distribution, is one of the most commonly used parametric distribution in various fields of science and engineering, including robotics. The probability density function for a one-dimensional[3] Gaussian distribution is:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}}, \tag{14.1}$$

where the parameter μ is the mean and the parameter σ is the standard deviation[4]. The notation we use to write that a random variable X is distributed according to a Gaussian distribution is $X \sim \mathcal{N}(\mu, \sigma^2)$. The probability density function for the multivariate Gaussian distribution is:

$$p(\boldsymbol{x}) = \frac{1}{\sqrt{\det(2\pi\boldsymbol{\Sigma})}} \exp\Big(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\Big), \tag{14.2}$$

where $\boldsymbol{x} \in \mathbb{R}^n$ and the parameters are the mean $\boldsymbol{\mu} \in \mathbb{R}^n$ and the covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{n \times n}$. The short form notation to write that a random vector X is distributed according to the multivariate Gaussian distribution is $X \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. These distributions are represented graphically for the univariate and bivariate case in Figure 14.1.

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 $Y \sim \mathcal{N}(A\boldsymbol{\mu} + b, A\boldsymbol{\Sigma}A^\top)$.

[3] We refer to a one-dimensional Gaussian as *univariate* and a Gaussian over multiple dimensions as *multivariate*.

[4] The quantity σ^2 is referred to as the *variance*.

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. To see this, 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 $\boldsymbol{\Sigma}$:

$$\boldsymbol{\Sigma} = (\boldsymbol{\Sigma}_1^{-1} + \boldsymbol{\Sigma}_2^{-1})^{-1},$$
$$\boldsymbol{\mu} = \boldsymbol{\Sigma}(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_2^{-1}\boldsymbol{\mu}_2).$$

14.2 Kalman Filter

The Kalman filter is an extremely well known parametric filtering algorithm for state estimation that leverages Gaussian distributions. Unlike the discrete Bayes filter from Chapter 13 this filter can be efficiently applied 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 with $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{R}_t)$, 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. The measurement model is assumed to take 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 only the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ need to be updated. 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 and Derivation

The Kalman filter algorithm is a recursive Bayes filter with prediction and measurement correction steps that take on a special form due to the structure of the Gaussian belief distribution and the linearity assumptions for the state transition and measurement models. We outline the Kalman filter algorithm in Algorithm 14.1 where the inputs are the mean and covariance of the belief at the previous time step and the current control and measurement, and the outputs are the updated mean and covariance. The prediction step of the filter computes

Algorithm 14.1: Kalman Filter

Data: $\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t$
Result: $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$
$\bar{\boldsymbol{\mu}}_t = A_t \boldsymbol{\mu}_{t-1} + B_t \boldsymbol{u}_t$
$\bar{\boldsymbol{\Sigma}}_t = A_t \boldsymbol{\Sigma}_{t-1} A_t^T + \boldsymbol{R}_t$
$K_t = \bar{\boldsymbol{\Sigma}}_t C_t^T (C_t \bar{\boldsymbol{\Sigma}}_t C_t^T + \boldsymbol{Q}_t)^{-1}$
$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t + K_t (\boldsymbol{z}_t - C_t \bar{\boldsymbol{\mu}}_t)$
$\boldsymbol{\Sigma}_t = (I - K_t C_t) \bar{\boldsymbol{\Sigma}}_t$
return $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$

the predicted mean $\bar{\boldsymbol{\mu}}_t$ and predicted covariance $\bar{\boldsymbol{\Sigma}}_t$ from the linear transition model and the remaining steps perform the measurement correction[5].

[5] The matrix K_t that is computed for the measurement correction is typically referred to as the *Kalman gain*.

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 will consider a more intuitive approach that directly leverages the properties of Gaussians presented in Section 14.1. First, from the prior belief distribution $\text{bel}(\boldsymbol{x}_{t-1}) \sim \mathcal{N}(\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1})$ the predicted belief $\overline{\text{bel}}(\boldsymbol{x}_{t-1})$ is computed 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 can 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 can 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 can 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 can now transform these equations in the form of the Kalman filter equations in Algorithm 14.1. From the matrix inversion lemma we can 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 then given by:

$$\boldsymbol{\Sigma}_t = \bar{\boldsymbol{\Sigma}}_t - K_t C_t \bar{\boldsymbol{\Sigma}}_t,$$

Through some additional algebra we can 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.[6].

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

14.2.2 *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 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 will take a look at a variant of the Kalman filter that can be applied 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) is a modified version of the Kalman filter that does not assume linearity for the state transition and measurement models. The EKF still uses the Gaussian distribution to represent the belief in a computationally efficient parametric way, but by generalizing to more complex state transition and measurement models the EKF can be applied to a wider variety of robotics state estimation and localization problems. Instead of the linear models (14.3) and (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 $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{R}_t)$ and $\delta_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}_t)$ 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. The first order Taylor series expansion of the state transition model $f(\boldsymbol{x}_{t-1}, \boldsymbol{u}_t)$ is performed 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 $G_t = \nabla_{\boldsymbol{x}} f(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t)$ 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.

A similar procedure is used for the measurement corrections. The measurement model is approximated 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.

The EKF prediction and measurement update steps are combined together in the overall EKF algorithm definition in Algorithm 14.2. Compare this to Algorithm 14.1 and you will notice only the small differences in that the EKF uses a combination of the nonlinear models and linear approximations from their Jacobians.

Algorithm 14.2: Extended Kalman Filter

Data: $\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t$
Result: $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$
$\bar{\boldsymbol{\mu}}_t = f(\boldsymbol{u}_t, \boldsymbol{\mu}_{t-1})$
$\bar{\boldsymbol{\Sigma}}_t = G_t \boldsymbol{\Sigma}_{t-1} G_t^\top + \boldsymbol{R}_t$
$K_t = \bar{\boldsymbol{\Sigma}}_t H_t^\top (H_t \bar{\boldsymbol{\Sigma}}_t H_t^\top + \boldsymbol{Q}_t)^{-1}$
$\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t + K_t(\boldsymbol{z}_t - h(\bar{\boldsymbol{\mu}}_t))$
$\boldsymbol{\Sigma}_t = (I - K_t H_t)\bar{\boldsymbol{\Sigma}}_t$
return $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$

14.3.1 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 series expansion can lead to issues where the filter does not perform well or even diverges if the approximation is not accurate enough[7]. 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.

[7] 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*.

14.4 Exercises

Code Exercise 14.4.1 (Free Flyer State Estimation)**.** The Free Flyer robot is a testbed that Stanford researchers use to develop algorithms for space robotics applications. The robot uses cold gas thrusters to accelerate and floats on a flat table using an air bearing. The robot uses a camera to measure its relative position with respect to a docking target, but cannot directly measure its velocity! In this exercise, we will use a Kalman Filter to estimate the full state of the Free Flyer robot. Please refer to `free-flyer`

Code Exercise 14.4.2 (Dubin Car State Estimation)**.** In this exercise, you will estimate the state of a Dubin's car dynamics model using an EKF and JAX. Please refer to `dubin`

14.5 Exercises

14.5.1 EKF Localization

Complete *Problem 1: EKF Localization* located in the online repository:

`https://github.com/PrinciplesofRobotAutonomy/AA274A_HW4,`

where you will implement an EKF for robot localization based on linear feature extraction and a map of known features.

15
Nonparametric Filters

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, in Chapter 14 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 these practical issues can be mitigated 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[1]. 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 the robot's position uncertainty cannot be effectively modeled 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 essentially 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 can be more computationally tractable for practical applications.

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

15.1 Histogram Filter

The *histogram filter* is a non-parametric filter that can be viewed as an extension of the discrete Bayes filter presented in Chapter 13 to continuous state spaces. This is accomplished by discretizing the continuous state space into a *finite* number of regions, and then representing the belief distribution over the discretized state space by a finite set of probability masses. Mathematically, for the random state vector X the continuous state space $\mathcal{X}$ is discreteized into a finite set of regions[2]:

$$\mathcal{X} = b_1 \cup b_2 \cup \ldots \cup b_K,$$

[2] In the context of the histogram filter we often refer to the the discretized regions as *bins*.

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.

We then modify the prediction and measurement update steps of the Bayes filter by discretizing the probabilistic state transition and measurement models with a representative "mean" state, $\hat{\boldsymbol{x}}_k$, for each bin:

$$\hat{\boldsymbol{x}}_k \coloneqq \frac{1}{|b_k|} \int_{b_k} \boldsymbol{x}\, d\boldsymbol{x}. \tag{15.1}$$

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:

$$p(b_{k,t} \mid b_{i,t-1}, \boldsymbol{u}_t) \approx \eta |b_k| p(\hat{\boldsymbol{x}}_{k,t} \mid \hat{\boldsymbol{x}}_{i,t-1}, \boldsymbol{u}_t), \tag{15.2}$$

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[3]. We discretize the probabilistic measurement model in a similar manner:

[3] If the bin areas $|b_k|$ are equal this term can be absorbed into the normalization constant η.

$$p(\boldsymbol{z}_t \mid b_{k,t}) \approx p(\boldsymbol{z}_t \mid \hat{\boldsymbol{x}}_{k,t}), \tag{15.3}$$

such that the measurement probability associated with a bin b_k is approximated by the measurement probability associated with the mean bin state $\hat{\boldsymbol{x}}_k$. Once we have discretized the state space with the bins b_k and the state transition and measurement models using the bin mean states $\hat{\boldsymbol{x}}_k$ the histogram filter mimics the discrete Bayes filter in Algorithm 13.2 and is detailed in Algorithm 15.1.

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

Algorithm 15.1: Histogram Filter

Data: $\{p_{k,t-1}\}, \boldsymbol{u}_t, \boldsymbol{z}_t$
Result: $\{p_{k,t}\}$
foreach k **do**
  $\overline{p}_{k,t} = \sum_i p(b_{k,t} \mid b_{i,t-1}, \boldsymbol{u}_t) p_{i,t-1}$
  $p_{k,t} = \eta p(\boldsymbol{z}_t \mid b_{k,t}) \overline{p}_{k,t}$
return $p_{k,t}$

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"!

15.2 Particle Filter

The *particle filter* is another non-parametric filter that can be applied 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*[4]. We define the set of particles at time t mathematically as $\mathcal{P}_t$:

$$\mathcal{P}_t := \{\boldsymbol{x}_t^{[1]}, \boldsymbol{x}_t^{[2]}, \ldots, \boldsymbol{x}_t^{[K]}\}, \tag{15.4}$$

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[5].

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 measurement update step. The prediction step considers each particle $\boldsymbol{x}_{t-1}^{[k]}$ in the prior set $\mathcal{P}_{t-1}$ and samples from the state transition model a new predicted sample:

$$\bar{\boldsymbol{x}}_t^{[k]} \sim p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}^{[m]}, \boldsymbol{u}_t).$$

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:

$$w_t^{[k]} = p(\boldsymbol{z}_t \mid \bar{\boldsymbol{x}}_t^{[k]}).$$

[4] The particle filter is sometimes referred to as a *Monte Carlo* algorithm due to its sampling approach.

[5] 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

The predicted particles $\bar{\boldsymbol{x}}_t^{[k]}$ and their associated weights $w_t^{[k]}$ are then collected 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 is defined by *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, the resampled points are collected to define the posterior belief distribution particle set, $\mathcal{P}_t$. The particle filter algorithm is outlined in Algorithm 15.2 and a few iterations of the algorithm for a simple robot localization problem are shown in Figure 15.1.

Algorithm 15.2: 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 $\bar{\boldsymbol{x}}_t^{[k]} \sim p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}^{[k]}, \boldsymbol{u}_t)$
  $w_t^{[k]} = p(\boldsymbol{z}_t \mid \bar{\boldsymbol{x}}_t^{[k]})$
  $\bar{\mathcal{P}}_t = \bar{\mathcal{P}}_t \cup (\bar{\boldsymbol{x}}_t^{[k]}, w_t^{[k]})$
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 computational 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.

15.3 Exercises

15.3.1 Monte Carlo Localization

Complete *Extra Credit: Monte Carlo Localization* located in the online repository:

`https://github.com/PrinciplesofRobotAutonomy/AA274A_HW4,`

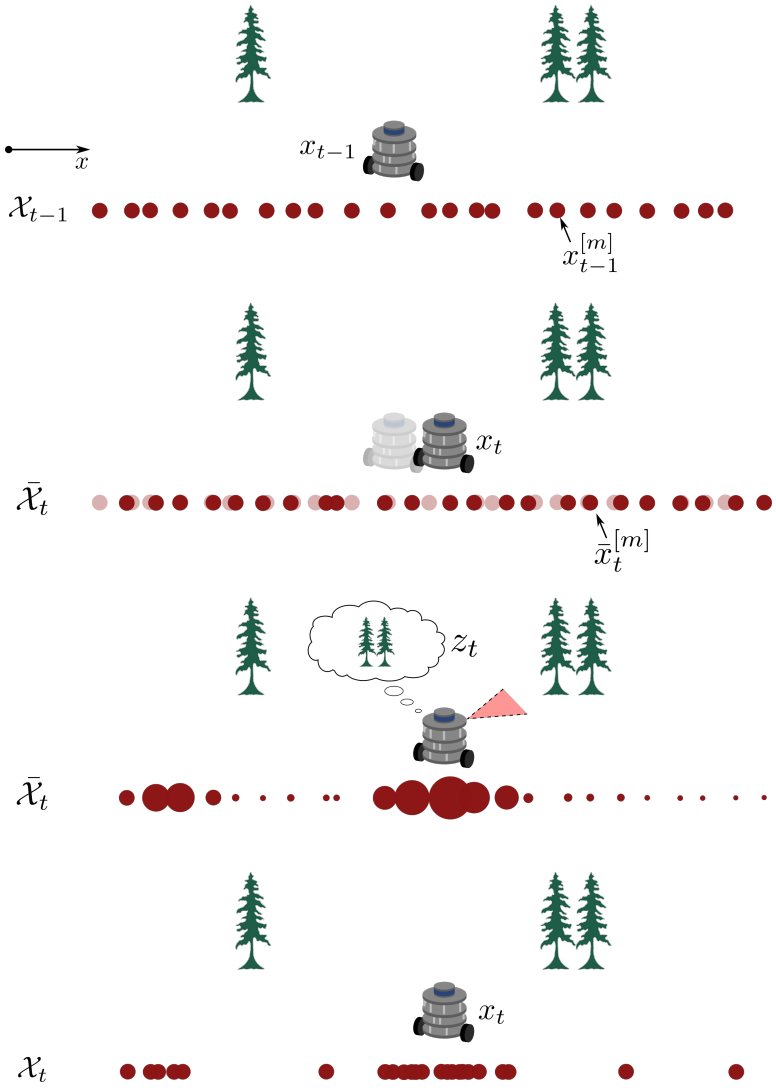


Figure 15.1: A particle filter used for robot localization. The initial set of particles are first updated according to the transition model, and then weighted according to the observation. Finally, a new set of particles is generated through weighted resampling.

where you will implement a particle filter for localizing a robot with line feature extraction, similar to the exercise on EKF localization from the previous chapter.

16
Robot Localization

Chapter 13 introduced a probabilistic framework for the localization and state estimation problem and presented the canonical Bayes filter algorithm. The previous chapters also introduced some widely used filtering algorithms based on 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 the notion of a local *map* of the environment, which 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 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 our goal can be viewed 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

16.1 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 may have, whether the environment is static or dynamic, if information is gathered passively or actively, and whether information gathering is performed by one robot or collaboratively with several robots.

Pose Tracking and Global Pose Localization: The first way we categorize robot localization problems is by how much knowledge is initially available, 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 further. 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 capture the global uncertainty well. For global localization problems we might prefer non-parametric filters that are better suited to model multiple hypotheses, such as the particle filter from Chapter 15. 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 elements in the filter algorithm, in addition to the robot's.

Active and Passive Localization: Information gathering 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 robots 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 robot's aren't limited to sharing sensor information, they can also share their belief distributions.

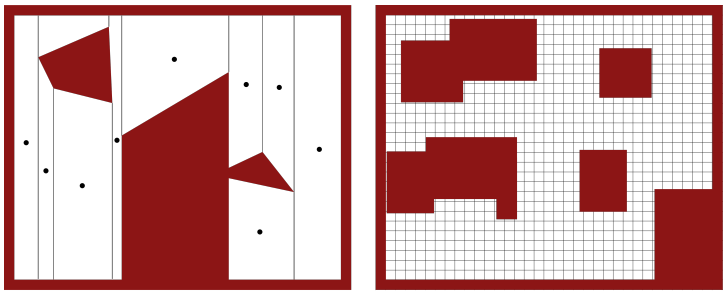
16.2 Robot Localization via Bayesian Filtering

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 15. 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 will 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 and is defined as:

$$\boldsymbol{m} = \{m_1, m_2, \ldots, m_N\}, \tag{16.1}$$

where m_i represents the set of properties of a specific object and N is the number of objects being tracked. We will generally consider two types of maps location-based maps and feature-based maps, which can lead to differences in both computational efficiency and expressiveness. 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 16.1. One



Cell Decomposition Map Grid-based Map

Figure 16.1: Two examples of location-based maps. Both represent the map as a set of volumetric objects, which in these examples are cells.

[3] In three dimensions the grid representation is usually referred to as a *voxel* representation.

potential disadvantage of cell-based maps is that their resolution is dependent on the size of the cells and therefore there is a clear tradeoff between computational complexity and accuracy. However, their advantage is that they can explicitly encode information about 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 16.2 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 the line-based map might make sense to use in highly structured environments such as buildings. However, their main disadvantage is that they may have lower resolution and might not capture spatial information about all potential obstacles.

[4] Feature-based maps can also be thought of as a collection of landmarks.

[5] Generally referred to as a *topological map*.

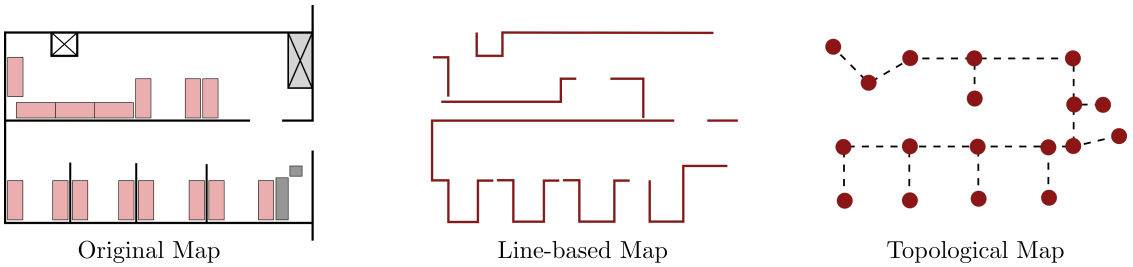


Figure 16.2: Two examples of feature-based maps. One uses a collection of lines and the other uses a graph representation of the empty spaces.

16.2.1 *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 will make an assumption that:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t, \boldsymbol{m}) \approx \eta \frac{p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t) p(\boldsymbol{x}_t \mid \boldsymbol{m})}{p(\boldsymbol{x}_t)}, \tag{16.2}$$

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 (16.2) 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.

16.2.2 *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. The new map-aware measurement model is expressed 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:

$$p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m}) = \prod_{i=1}^{p} p(z_t^i \mid \boldsymbol{x}_t, \boldsymbol{m}). \tag{16.3}$$

16.3 Markov Localization

The first map-based localization algorithm we introduce is referred to as the *Markov localization* algorithm and is conceptually the same as the Bayes filter except for the inclusion of the map in the probabilistic state transition $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_t, \boldsymbol{m})$ and measurement model $p(\boldsymbol{z}_t \mid \boldsymbol{x}_t, \boldsymbol{m})$. The algorithm, outlined in Algorithm 16.1, provides a general framework that can be used to address many of the localization problems discussed in our taxonomy in Section 16.1, including pose tracking and global pose localization problems, but it does assume the map is already known. Similarly to the Bayes filter from Chapter 13,

Algorithm 16.1: 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 $\text{bel}(\boldsymbol{x}_t)$

the Markov localization algorithm in Algorithm 16.1 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 16.4 we will again use a Gaussian belief distribution, like in the EKF, to design an *extended Kalman filter localization* algorithm. Then in Section 16.5 we will again use a sampling-based approach, like in the particle filter, to design the *Monte Carlo localization* algorithm.

16.4 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 which 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:

$$G_t = \nabla_{\boldsymbol{x}} f(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t), \tag{16.4}$$

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 each landmark m_j is defined 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 measurement i is associated with.

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 then compute the Jacobian H_t^i that is used in the EKF measurement update step. Specifically, for the i-th measurement the Jacobian of the new measurement model is computed by:

$$H_t^{c_t^i} = \nabla_{\boldsymbol{x}} h(\bar{\boldsymbol{\mu}}_t, c_t^i, \boldsymbol{m}), \tag{16.5}$$

where $\bar{\boldsymbol{\mu}}_t$ is the predicted mean from the EKF prediction step.

16.4.1 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 the correspondences are assumed to be known. For known correspondences the EKF localization algorithm is given in Algorithm 16.2. Compared to the general EKF algorithm in Algorithm 14.2 we can see that the main difference is that multiple measurements are processed at the same time. This is possible because of our assumption in in Equation (16.3) that the measurements are conditionally independent. Specifically, with this conditional independence assumption and some special properties of Gaussian distributions the multi-measurement update can be performed by sequentially looping over each measurement and applying the standard EKF measurement update steps.

Algorithm 16.2: 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 $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$

16.4.2 EKF Localization with Unknown Correspondences

Algorithm 16.2 shows what the EKF localization algorithms 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 the most likely value of the correspondences $\boldsymbol{c}_t$ is calculated 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 (16.3) 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 choose to 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:

$$\hat{c}_t^i = \arg\min_{c_t^i}\ d_t^{i, c_t^i}, \tag{16.6}$$

where:

$$d_t^{ij} = (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^j)^\top [S_t^j]^{-1} (\boldsymbol{z}_t^i - \hat{\boldsymbol{z}}_t^j), \tag{16.7}$$

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 16.2 gives the new EKF location algorithm in Algorithm 16.3.

Algorithm 16.3: 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 Mahalanobis smallest 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 16.4.1 (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 will assume that multiple measurements corresponding to different features are collected 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:

$$h(\boldsymbol{x}_t, j, \boldsymbol{m}) = \begin{bmatrix} \sqrt{(m_{j,x} - x)^2 + (m_{j,y} - y)^2} \\ \text{atan2}(m_{j,y} - y, m_{j,x} - x) - \theta \end{bmatrix}. \tag{16.8}$$

The measurement Jacobian H_t^j corresponding to a measurement from landmark

j is therefore:

$$H_t^j = \begin{bmatrix} -\frac{m_{j,x}-\bar{\mu}_{t,x}}{\sqrt{(m_{j,x}-\bar{\mu}_{t,x})^2+(m_{j,y}-\bar{\mu}_{t,y})^2}} & -\frac{m_{j,y}-\bar{\mu}_{t,y}}{\sqrt{(m_{j,x}-\bar{\mu}_{t,x})^2+(m_{j,y}-\bar{\mu}_{t,y})^2}} & 0 \\ \frac{m_{j,y}-\bar{\mu}_{t,y}}{(m_{j,x}-\bar{\mu}_{t,x})^2+(m_{j,y}-\bar{\mu}_{t,y})^2} & -\frac{m_{j,x}-\bar{\mu}_{t,x}}{(m_{j,x}-\bar{\mu}_{t,x})^2+(m_{j,y}-\bar{\mu}_{t,y})^2} & -1 \end{bmatrix}. \tag{16.9}$$

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. We typically use a diagonal covariance matrix in this case since we can assume that these two measurements are uncorrelated.

16.5 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 15, 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 16.4, which we can see is nearly identical to the particle filter algorithm in Algorithm 15.2 except that the map $\boldsymbol{m}$ is used in the probabilistic state transition and measurement models.

Algorithm 16.4: 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 $\bar{\boldsymbol{x}}_t^{[k]} \sim p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}^{[k]}, \boldsymbol{u}_t, \boldsymbol{m})$
  $w_t^{[k]} = p(\boldsymbol{z}_t \mid \bar{\boldsymbol{x}}_t^{[k]}, \boldsymbol{m})$
  $\bar{\mathcal{P}}_t = \bar{\mathcal{P}}_t \cup \left(\bar{\boldsymbol{x}}_t^{[k]}, w_t^{[k]}\right)$
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 $particleset_t$

17
Simultaneous Localization and Mapping (SLAM)

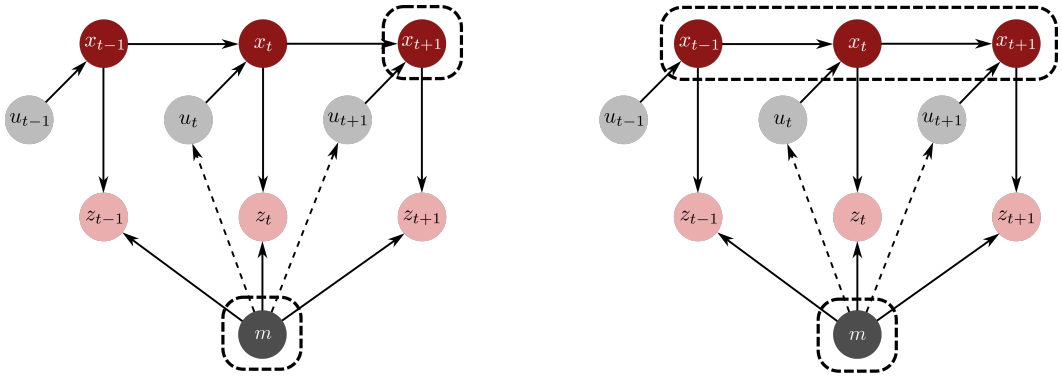
In Chapter 16 we introduced the robot localization problem, but assumed that a map of the environment, $\boldsymbol{m}$, was given. This assumption is not practical for some real-world applications of robot autonomy, and we need to now consider localization problems where the map also needs to be built in real-time. For example in autonomous search-and-rescue operations a robot needs to explore an unknown environment while maintaining an understanding of where it currently is and where it has already searched. We refer to this problem as *simultaneous localization and mapping (SLAM)* [1] and it involves using information about the measurements $\boldsymbol{z}$ and controls $\boldsymbol{u}$ to simultaneously build a map and localize the robot with respect to it.

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

We will categorize SLAM problems into two types: online SLAM and full SLAM. The *online* SLAM problem is to estimate the posterior $p(\boldsymbol{x}_t, \boldsymbol{m} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t})$ over the robot's current state $\boldsymbol{x}_t$ and the map $\boldsymbol{m}$. The *full* SLAM problem is to estimate the map and the *entire state trajectory* of the robot, $p(\boldsymbol{x}_{1:t}, \boldsymbol{m} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t})$, instead of just the current state. We highlight the difference between these two SLAM problems graphically in Figure 17.1. One of the main technical challenges for both SLAM problems is that error in the state estimate can cause error in map estimation and error in map estimation can cause error in the state estimate. In this chapter we introduce the *EKF SLAM* algorithm for solving the online SLAM problem and a particle filter-based algorithm called *FastSLAM* for solving the full SLAM problem.

17.1 EKF SLAM Algorithm

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 16. In this section we introduce the *EKF SLAM* algorithm, which is another extension of the EKF framework which can solve the online SLAM problem. Like in the previous chapters we model the robot's belief distribution parametrically using a Gaussian distribution. This gives the EKF SLAM algorithm computational benefits that make it practical to run in real-time, but again limits us to a unimodal belief hypothesis which can limit



Online SLAM

Full SLAM

Figure 17.1: Online SLAM problems only estimate the current robot state while full SLAM problems also estimate the robot's history.

the algorithm's performance. As in Chapter 16 we will also 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 $(m_{i,x}, m_{i,y})$. As in the EKF localization problem we consider both cases where the measurement correspondences are known or unknown[2].

Since the map feature data is unknown we consider it as a part of an augmented state vector that we seek to generate a belief distribution over. Specifically, we define a new augmented state vector, y, that contains both the robot's state and the map feature coordinates:

$$\boldsymbol{y}_t := \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{m}_1 \\ \vdots \\ \boldsymbol{m}_N \end{bmatrix}, \tag{17.1}$$

where $\boldsymbol{m}_i = [m_{i,x}, m_{i,y}]^\top$. The goal of the online SLAM problem is now to compute the posterior belief distribtution:

$$\text{bel}(\boldsymbol{y}_t) := p(\boldsymbol{y}_t \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}).$$

We consider a state transition model for the augmented state vector $\boldsymbol{y}$ of the form:

$$\boldsymbol{y}_t = g(\boldsymbol{y}_{t-1}, \boldsymbol{u}_t) + \boldsymbol{\epsilon}_t, \quad \boldsymbol{\epsilon}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{R}_t),$$

with the nonlinear vector function g defined by:

$$g(\boldsymbol{y}_{t-1}, \boldsymbol{u}_t) = \begin{bmatrix} f(\boldsymbol{x}_{t-1}, \boldsymbol{u}_t) \\ \boldsymbol{m}_{1,t-1} \\ \vdots \\ \boldsymbol{m}_{N,t-1} \end{bmatrix},$$

[2] The case of unknown measurement correspondences is much more common in practice.

where f is the robot motion model and where we are assuming that the features $m_i \in \boldsymbol{m}$ are constant in time. The noise covariance for the state transition model is defined as:

$$\boldsymbol{R}_t = \begin{bmatrix} \tilde{\boldsymbol{R}}_t & 0 \\ 0 & 0 \end{bmatrix},$$

where $\tilde{\boldsymbol{R}}_t$ is the noise covariance associated with the robot motion model and the rest of the matrix are zeros since we assume no noise related to motion of the map features. The Jacobian of the augmented motion model is $G_t = \nabla_{\boldsymbol{y}} g(\boldsymbol{\mu}_{t-1}, \boldsymbol{u}_t)$ where $\boldsymbol{\mu}_{t-1}$ is the mean of the belief distribution $\text{bel}(\boldsymbol{y}_{t-1})$ at the previous time. The measurement model is defined as in Chapter 16:

$$\boldsymbol{z}_t^i = h(\boldsymbol{y}_t, j) + \boldsymbol{\delta}_t,$$

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 $m_j \in \boldsymbol{m}$ that measurement i is associated with. The Jacobian is of this measurement model is $H_t^j = \nabla_{\boldsymbol{y}} h(\bar{\boldsymbol{\mu}}_t, j)$, where $\bar{\boldsymbol{\mu}}_t$ is the mean from the predicted belief distribution $\overline{\text{bel}}(\boldsymbol{y}_t)$ from the EKF prediction step.

17.1.1 EKF SLAM with Known Correspondences

As was the case for the EKF localization algorithm, we will start by considering the form of the EKF SLAM algorithm when the correspondences $\boldsymbol{c}_t = [c_t^1, \dots]^\top$ c_t^i between the measurements $\boldsymbol{z}_t$ and their associated features in the map are known. In fact, with known correspondences the EKF SLAM algorithm, detailed in Algorithm 17.1, is almost identical to the EKF localization algorithm in Algorithm 16.2 except that we use the augmented state vector $\boldsymbol{y}$. For this algorithm a general initialization of the belief distribution $\text{bel}(\boldsymbol{y}_0)$ is with the mean and covariance:

$$\boldsymbol{\mu}_0 = \begin{bmatrix} \boldsymbol{x}_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \boldsymbol{\Sigma}_0 = \begin{bmatrix} \tilde{\boldsymbol{\Sigma}}_0 & 0 & \cdots & 0 \\ 0 & \infty & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \infty \end{bmatrix},$$

where:

$$\boldsymbol{x}_0 = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \tilde{\boldsymbol{\Sigma}}_0 = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix},$$

and $\boldsymbol{x}_0$ and $\tilde{\boldsymbol{\Sigma}}$ are the initial robot state and associated covariance. Since the reference frame for the map is defined arbitrarily, this initialization puts the initial robot pose at the origin with certainty and then the uncertain map is estimated with respect to that origin. The covariance of the map feature positions is therefore initially set to infinity to reflect that there is initially no knowledge of their position. Algorithm 17.1 then re-initializes the mean for each feature based on the measurement when the feature is first observed[3]. Note however that this re-initialization step assumes that a single measurement is sufficient to compute

[3] This re-initialization is important to make sure that the linearization in the EKF prediction and measurement update steps is not arbitrarily performed at the origin.

Algorithm 17.1: EKF Online SLAM, Known Correspondences

Data: $\mu_{t-1}, \Sigma_{t-1}, u_t, z_t, c_t$
Result: μ_t, Σ_t
$\bar{\mu}_t = g(\mu_{t-1}, u_t)$
$\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t$
foreach z_t^i **do**
  $j = c_t^i$
  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 z_t^i
  $S_t^i = H_t^j \bar{\Sigma}_t [H_t^j]^T + Q_t$
  $K_t^i = \bar{\Sigma}_t [H_t^j]^T [S_t^i]^{-1}$
  $\bar{\mu}_t = \bar{\mu}_t + K_t^i (z_t^i - h(\bar{\mu}_t, j))$
  $\bar{\Sigma}_t = (I - K_t^i H_t^j) \bar{\Sigma}_t$
$\mu_t = \bar{\mu}_t$
$\Sigma_t = \bar{\Sigma}_t$
return μ_t, Σ_t

an initial guess of the position, which may not always be true in practice, such as if the robot only has bearing measurements[4].

[4] The *bearing* is the relative angle to an object, which does not include its range. Cameras are one type of sensor that may not necessarily give depth information but can give bearing information.

17.2 EKF SLAM with Unknown Correspondences

EKF SLAM is more challenging when the correspondences between measurements and features are unknown. In the EKF localization problem in Chapter 16, which assumes the map is known, we use a maximum likelihood method to compute the correspondences before the EKF updates are applied to the belief. We take a similar approach in EKF SLAM, but use a maximum likelihood method based on *estimated* feature positions. We now also need a mechanism for hypothesizing that a new feature has been found and updating the features in the map. To accomplish this we first use each measurement z_k^i to estimate the position of a potential new feature, which would increase the number of features from N_{t-1} from the previous time step to $N_t = N_{t-1} + 1$. Then we compute the Mahalanobis distance d_t^{ik} for measurement i and feature k for all existing map features and add the hypothesized feature to the map if the Mahalanobis distance is larger than a threshold α for all existing features[5]. We outline the full EKF SLAM algorithm with unknown correspondences in Algorithm 17.2.

[5] Choosing α to be too small could result in a lot of features being added to the map, which could increase computational complexity, but choosing it to be too large could result in missing features.

The EKF online SLAM algorithm is one of the earliest and simplest SLAM algorithms for unknown correspondences, but in practice it is not necessarily vary robust. For example, extraneous measurements from sensor noise can result in the creation of false map features that will continue to propagate forward to

Algorithm 17.2: 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 $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$

future steps and are never corrected. Issues such as this can be mitigated with some additional extensions, such as outlier rejection schemes or strategies to enhance the distinctiveness of features. Another important practical disadvantage of EKF SLAM is that its computational complexity is quadratic with the number of map features N, but generally a large number of features is required for good localization accuracy. Therefore it can be difficult to find a good practical tradeoff between computational runtime and performance.

Example 17.2.1 (Differential Drive Robot with Range and Bearing Measurements)**.** Consider a differential drive robot with state consisting of the two-dimensional position and heading such that $\boldsymbol{x} = [x, y, \theta]^\top$. Suppose a sensor is available which measures the range r and bearing ϕ of features $m_j \in \boldsymbol{m}$ relative to the robot's local coordinate frame. We also assume that multiple measurements corresponding to different features are collected 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. For the SLAM

problem we define the augmented state $\boldsymbol{y}_t$:

$$\boldsymbol{y}_t \coloneqq \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{m}_1 \\ \vdots \\ \boldsymbol{m}_N \end{bmatrix} = \begin{bmatrix} x & y & \theta & m_{1,x} & m_{1,y} & \dots & m_{N,x} & m_{N,y} \end{bmatrix}^\top.$$

Assuming the correspondences are known, the measurement model for the range and bearing is:

$$h(\boldsymbol{y}_t, j) = \begin{bmatrix} \sqrt{(m_{j,x} - x)^2 + (m_{j,y} - y)^2} \\ \text{atan2}(m_{j,y} - y, m_{j,x} - x) - \theta \end{bmatrix},$$

and the measurement 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:

$$q_{t,j} \coloneqq (\bar{\mu}_{j,x} - \bar{\mu}_{t,x})^2 + (\bar{\mu}_{j,y} - \bar{\mu}_{t,y})^2,$$

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 17.1 to initialize the feature position and in the unknown-correspondence case in Algorithm 17.2 to hypothesize the position of new features.

17.3 Particle SLAM

We can also solve the robot SLAM problem using the framework of the nonparametric particle filter. In fact, we can use the particle filter approach to SLAM to solve the *full* SLAM problem[6] which estimates the posterior distribution $p(\boldsymbol{x}_{1:t}, \boldsymbol{m} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t})$ that includes the map $\boldsymbol{m}$ and the full robot state history $\boldsymbol{x}_{1:t}$. The simplest implementation of a particle filter-based SLAM algorithm might look very similar to the Monte Carlo localization algorithm from Chapter 16, but using the augmented state vector $\boldsymbol{y}$ defined in Equation (17.1) that includes the map. However, in practice such a naive implementation would likely be computationally intractable since the number of particles required to accurately represent the belief distribution scales poorly with the dimension of the state space, and the number of features in the map can be very large. We

[6] This is in contrast to EKF SLAM, which only solves the *online* SLAM problem.

can avoid this curse of dimensionality with a key insight that the positions of the map features are *conditionally independent* given the state history of the robot and known correspondences. This insight allows us to factor the SLAM problem into a localization component handled by a particle belief and mapping components handled by a parametric belief for each map feature, consequently reducing the computational complexity of the algorithm.

As in our past discussions on the tradeoffs between EKF-based and particle filter-based approaches, particle SLAM algorithms have the advantage that they can handle more arbitrary models for state transition and measurement noise, and can represent multimodal belief distributions for the robot state. Sampling-based approaches like the particle filter can also end up being easier to implement in practice than EKF-based approaches since they do not require explicit derivation of the model Jacobians, and they can also be more robust to data association errors. Their relative disadvantage compared to EKF-based approaches is again that they may not scale as well to larger problems since the number of particles required to effectively represent the belief would be too large.

17.3.1 Factoring the Posterior

We start our discussion on particle SLAM methods by describing the key insight that will allow us to factor the SLAM problem into a more computationally tractable form. Using the augmented state vector $\boldsymbol{y}$ from Equation (17.1) the posterior belief distribution that the *full* SLAM problem is trying to estimate is $p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t})$[7]. The key insight is that this posterior distribution can be factored as:

$$p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) = p(\boldsymbol{x}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t}) \prod_{i=1}^{N} p(\boldsymbol{m}_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}), \tag{17.2}$$

[7] Note here we are assuming known correspondences $c_{1:t}$ and that there is only one measurement at each time step.

where $\boldsymbol{m}_i$ 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 $p(\boldsymbol{m}_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t})$ 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 $\boldsymbol{m}_i$ 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. Under

these two cases we can write the feature posterior for $\boldsymbol{m}_i$ as:

$$p(\boldsymbol{m}_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}) = \begin{cases} p(\boldsymbol{m}_i \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}), & i \neq c_t, \\ \frac{p(\boldsymbol{z}_t \mid \boldsymbol{m}_i, \boldsymbol{x}_t, c_t) p(\boldsymbol{m}_i \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1})}{p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t-1}, c_{1:t})}, & i = c_t, \end{cases}$$

where the first case is the result of the recent measurement having no impact on the posterior and in the second case we applied Bayes' rule and some other simplifications. For the case of $i = c_t$ we can therefore write the probability of the observed feature as:

$$p(\boldsymbol{m}_{c_t} \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}) = \frac{p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t-1}, c_{1:t}) p(\boldsymbol{m}_{c_t} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t})}{p(\boldsymbol{z}_t \mid \boldsymbol{m}_{c_t}, \boldsymbol{x}_t, c_t)}.$$

We can now show that the factorization in Equation (17.2) holds by induction. First, suppose that the feature posterior at the previous time $t-1$ is factored as[8]:

$$p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}) = \prod_{i=1}^{N} p(\boldsymbol{m}_i \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}).$$

[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 some simplifications on the feature posterior at time t:

$$\begin{aligned} p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}) &= \frac{p(\boldsymbol{z}_t \mid \boldsymbol{m}, \boldsymbol{x}_t, c_t) p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1})}{p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t-1}, c_{1:t})}, \\ &= \frac{p(\boldsymbol{z}_t \mid \boldsymbol{m}_{c_t}, \boldsymbol{x}_t, c_t)}{p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t-1}, c_{1:t})} \prod_{i=1}^{N} p(\boldsymbol{m}_i \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}). \end{aligned}$$

Next we apply the analysis of $p(\boldsymbol{m}_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{c}_{1:t})$ above for the cases where $i \neq c_t$ and $i = c_t$ to get:

$$\begin{aligned} p(\boldsymbol{m} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t}) &= \frac{p(\boldsymbol{z}_t \mid \boldsymbol{m}_{c_t}, \boldsymbol{x}_t, c_t)}{p(\boldsymbol{z}_t \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t-1}, c_{1:t})} p(\boldsymbol{m}_{c_t} \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}) \prod_{i \neq c_t} p(\boldsymbol{m}_i \mid \boldsymbol{x}_{1:t-1}, \boldsymbol{z}_{1:t-1}, c_{1:t-1}), \\ &= p(\boldsymbol{m}_{c_t} \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{c}_{1:t}) \prod_{i \neq c_t} p(\boldsymbol{m}_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{c}_{1:t}), \\ &= \prod_{n=1}^{N} p(\boldsymbol{m}_n \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, \boldsymbol{c}_{1:t}), \end{aligned}$$

which shows the correctness of the factorization.

17.3.2 *FastSLAM with Known Correspondences*

Now that we have shown that the posterior distribution $p(\boldsymbol{y}_{1:t} \mid \boldsymbol{z}_{1:t}, \boldsymbol{u}_{1:t}, c_{1:t})$ can be factored according to Equation (17.2) we introduce the particle SLAM algorithm referred to as *FastSLAM* which exploits this factorization for computational efficiency. Specifically, this algorithm 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 posteriors $p(\boldsymbol{m}_i \mid \boldsymbol{x}_{1:t}, \boldsymbol{z}_{1:t}, c_{1:t})$ 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 particle filter approach curse of dimensionality.

For this factorization the set of particles that FastSlam uses is defined as:

$$\mathcal{P}_t := \{P_t^{[1]}, P_t^{[2]}, ..., P_t^{[K]}\},$$

where the k-th particle is defined by:

$$P_t^{[k]} := \{\boldsymbol{x}_t^{[k]}, \boldsymbol{\mu}_{1,t}^{[k]}, \boldsymbol{\Sigma}_{1,t}^{[k]}, \ldots, \boldsymbol{\mu}_{N,t}^{[k]}, \boldsymbol{\Sigma}_{N,t}^{[k]}\},$$

where $\boldsymbol{x}_t^{[k]}$ is a hypothesis of the robot state at time t, $(\boldsymbol{\mu}_{i,t}^{[k]}, \boldsymbol{\Sigma}_{i,t}^{[k]})$ are the mean and covariance of the EKF associated with feature $\boldsymbol{m}_i$, and where we assume that there are N total features in the map $\boldsymbol{m}$. 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, therefore with a total of K particles there are a total of NK EKFs. The FastSLAM algorithm is outlined in

Algorithm 17.3: FastSLAM

Data: $\mathcal{P}_{t-1}, \boldsymbol{u}_t, \boldsymbol{z}_t, c_t$
Result: $\mathcal{P}_t$
for $k = 1$ **to** K **do**
  Sample $\boldsymbol{x}_t^{[k]} \sim p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}^{[k]}, \boldsymbol{u}_t)$
  $j = c_t$
  if *feature j never seen before* **then**
    Initialize feature: $(\boldsymbol{\mu}_{j,t-1}^{[k]}, \boldsymbol{\Sigma}_{j,t-1}^{[k]})$
  else
    $\hat{\boldsymbol{z}}^{[k]} = h(\boldsymbol{\mu}_{j,t-1}^{[k]}, \boldsymbol{x}_t^{[k]})$
    $S = H^j \boldsymbol{\Sigma}_{j,t-1}^{[k]} [H^j]^T + \boldsymbol{Q}_t$
    $K = \boldsymbol{\Sigma}_{j,t-1}^{[k]} [H^j]^T [S]^{-1}$
    $\boldsymbol{\mu}_{j,t}^{[k]} = \boldsymbol{\mu}_{j,t-1}^{[k]} + K(\boldsymbol{z}_t - \hat{\boldsymbol{z}}^{[k]})$
    $\boldsymbol{\Sigma}_{j,t}^{[k]} = (I - KH^j)\boldsymbol{\Sigma}_{j,t-1}^{[k]}$
    $w^{[k]} = \det(2\pi S)^{-1/2} \exp\big(-\frac{1}{2}(\boldsymbol{z}_t - \hat{\boldsymbol{z}}^{[k]}) \boldsymbol{Q}^{-1} (\boldsymbol{z}_t - \hat{\boldsymbol{z}}^{[k]})\big)$
  for $n \in \{1, \ldots, N\}$, $n \neq c_t$ **do**
    $\boldsymbol{\mu}_{n,t}^{[k]} = \boldsymbol{\mu}_{n,t-1}^{[k]}$
    $\boldsymbol{\Sigma}_{n,t}^{[k]} = \boldsymbol{\Sigma}_{n,t-1}^{[k]}$
$\mathcal{P}_t = \emptyset$
for $i = 1$ **to** K **do**
  Draw k with probability $\propto w_t^{[i]}$
  $\mathcal{P}_t = \mathcal{P}_t \cup (\bar{\boldsymbol{x}}_t^{[k]}, \boldsymbol{\mu}_{1,t}^{[k]}, \boldsymbol{\Sigma}_{1,t}^{[k]}, \ldots, \boldsymbol{\mu}_{N,t}^{[k]}, \boldsymbol{\Sigma}_{N,t}^{[k]})$
return $\mathcal{P}_t$

Algorithm 17.3. Note how this algorithm blends together elements of the classical particle filter algorithm with the EKF localization algorithm. In particular, we can see the particle filter steps in the sampling of the new pose x_t from the state transition model and the use of the weights w for resampling a new set of particles for the measurement update step. The EKF portions of the algorithm correspond to how the features are tracked, and in particular how the mean and covariance of the Gaussian distribution corresponding to each map feature are updated based on new measurements.

17.4 Exercises

17.4.1 EKF SLAM

Complete *Problem 2: EKF SLAM* located in the online repository:

```
https://github.com/PrinciplesofRobotAutonomy/AA274A_HW4,
```

where you will implement an EKF SLAM algorithm. Note that the EKF localization exercise from the chapter on parametric filters should be completed first.

18
Sensor Fusion 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 not uncommon for individual sensors to malfunction, either through a degradation of measurement accuracy over time or sometimes 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.

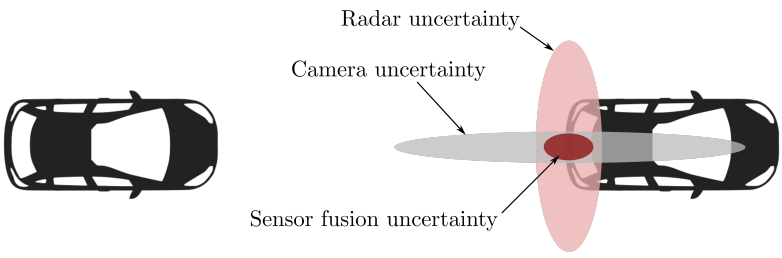


Figure 18.1: Sensor fusion can reduce uncertainty by providing more well-rounded data. For example a radar sensor may provide good longitudinal distance accuracy but slightly less lateral accuracy, and a camera may provide poor range estimation but good lateral position estimation. By fusing these two 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 stays 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

18.1 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.

18.2 Bayesian Approach to Sensor Fusion

The previous chapters present several algorithms for robot state estimation and localization based on Bayes filter that we can also view as algorithms that 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 a probabilistic approach that 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 it is also 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 18.2.1 (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 we have:

$$p(y_1, y_2 \mid x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}},$$

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.

18.2.1 Kalman Filter Sensor Fusion

The Kalman filter from Chapter 14 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 $\boldsymbol{R}_t$ which includes the covariance of each individual sensor. In fact, the Kalman filter will implicitly favor measurements with lower covariance when performing the correction step[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 18.2.2 (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 will 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 18.2 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.

18.3 Practical Challenges in Sensor Fusion

Sensor fusion problems are generally quite challenging and can vary significantly from application to application. Some practical problems in sensor fusion

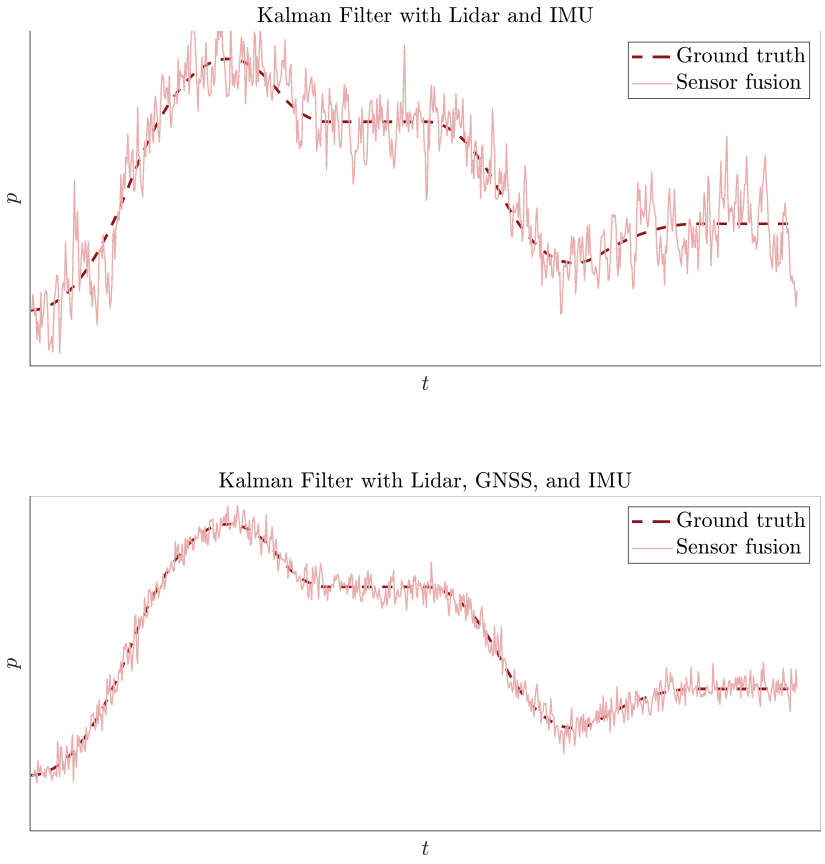


Figure 18.2: Kalman filter sensor fusion for Example 18.2.2. The position of a vehicle is estimated using noisy lidar, GNSS, and IMU data, and the resulting estimate tracks the ground truth. We can see that the addition of the lower-variance GNSS sensor results improves the estimate through sensor fusion.

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 their signals appropriately combined. Biases can also arise due to transformations of the data into the unified set of coordinates. Correlation between sensors can also occur, even if they are independently collecting data, and the knowledge of correlation between sensors can have an impact on the best way to fuse the information. In some robotics applications, such as is in multi-target tracking problems, data association can also be a challenge and is similar to the correspondence problem in the SLAM problems discussed in Chapter 17. Finally, out-of-sequence measurements[8] 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] One example cause of out-of-sequence measurements is due to communication limitations among agents in multi-agent settings.

18.4 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 measurements, occlusions, false signals, and inherent prediction uncertainty[9] is an important task. The *single-object tracking*[10] problem is well-understood and typically easy to solve, for example we can leverage algorithms 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.

[9] In other words, uncertainty in the dynamics of the object.

[10] Also sometimes referred to as *single hypothesis tracking*.

18.4.1 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.

18.4.2 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 will focus two solutions 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 tracks 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[11].

[11] Unlike the GNN, which makes a hard assignment, the JPDA approach performs a soft assignment of observations to tracks.

18.4.3 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.

18.4.4 Extended Object Tracking

One potential failure mode of standard multi-object tracking algorithms is when a single target generates multiple observations[12], 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.

[12] Standard multi-object tracking algorithms typically assume a single object detection per sensor.

Part IV

Robot Decision Making

19
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*.

19.1 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. Finite state machines can be used 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 19.1 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.

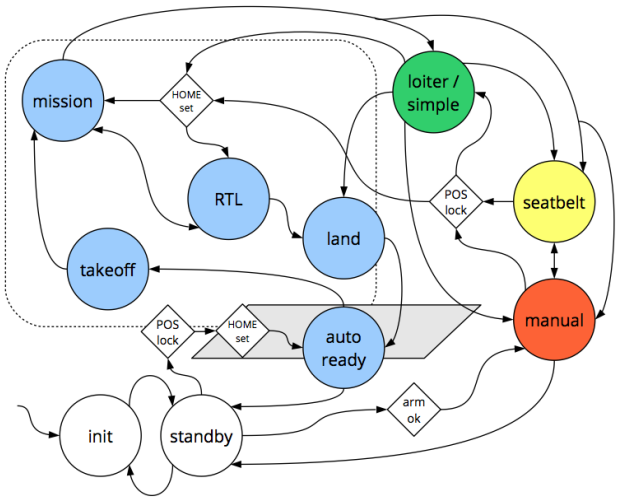


Figure 19.1: A graphical representation of the finite state machine for the open source flight software PX4, `https://px4.io/`. We can see that even for a relatively small number of states the FSM is become quite complex in order to model the full behavior of the system. Image retrieved from diydrones.com.

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 present FSMs graphically, which often gives a more intuitive understanding of how the system will behave. The graph representations are defined with nodes of the graph representing each state in the set S. Each directed edge of the graph corresponds to a possible transition between states that is defined by a particular input. We also typically include the outputs for a particular pair (s, i) along each directed edge. Figure 19.2 shows an example graphical representation of a simple finite state machine with three states.

Example 19.1.1 (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-

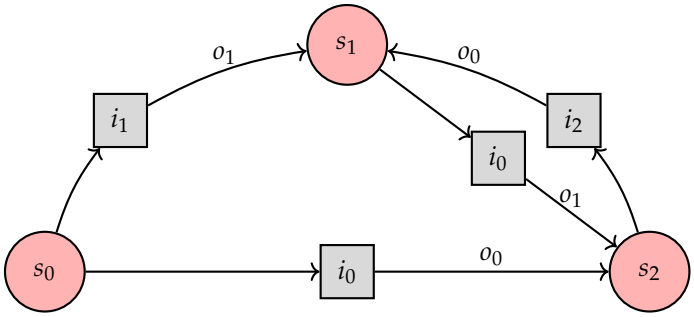


Figure 19.2: A graphical representation of a finite state machine with states $S = \{s_0, s_1, s_2\}$, inputs $I = \{i_0, i_1, i_2\}$ and outputs $O = \{o_0, o_1\}$. The directed edges correspond to the next-state functions and the output associated with each edge is defined by the output function. For example, in this FSM the transition $n(i_1, s_0) \rightarrow s_1$ is shown 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 will 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 will choose the states to be:

$$S := \{\text{down, raising, up, lowering}\}.$$

We will choose the set of inputs, which correspond to the available sensors, to be:

$$I := \{\text{car waiting, no car waiting, car past, car not past,}\\ \text{gate up, gate down, gate moving}\}.$$

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 19.3 shows a graphical representation of the full car parking gate finite state machine.

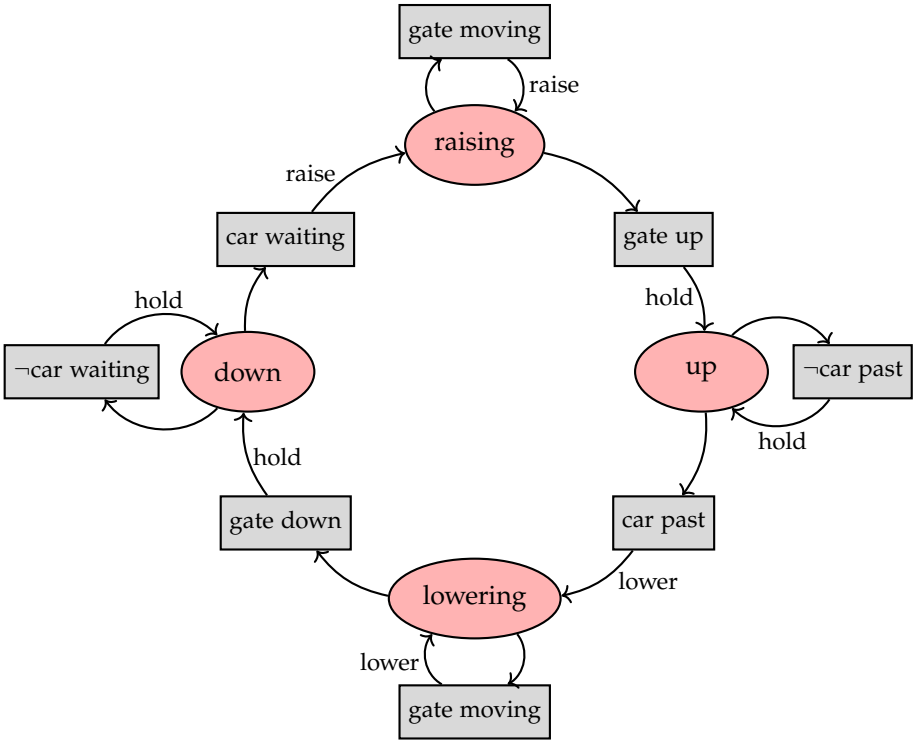


Figure 19.3: A graphical representation of the finite state machine for the parking gate controller discussed in Example 19.1.1.

19.2 *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.

19.2.1 *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 19.2.1.

Example 19.2.1 (Finite State Machine State Reduction)**.** Consider a finite state machine that detects the input sequences 010 or 110. The states, next-state function values, and output function values for this FSM are shown in Table 19.1. 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 19.1: 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 19.2.

[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 19.2: Reduced finite state machine for a sequence detector that accepts digits 0 and 1 and outputs True if the sequences 010 or 110 is generated.

19.2.2 *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.

19.2.3 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 19.4. 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.

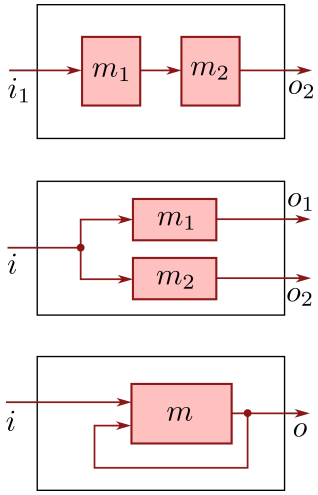


Figure 19.4: Cascade, parallel, and feedback compositions of finite state machines.

19.3 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 19.1.1 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()
```

20
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] which allows us to decompose longer horizon planning tasks into incremental actions and incorporate new information about the world, 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 19 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[2] that is commonly applied to a large variety of decision making problems. We also extend the problem formulation to consider problems with *uncertainty*[3], which is a fundamental aspect of practical robot autonomy[4]. In addition to the problem formulation we will 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] Like some of the methods discussed in Chapter 3 this new approach is considered a *closed-loop optimal control* method.

[3] These problems are referred to as *stochastic* decision making problems, and can include uncertainty in the robot's state and environment.

[4] The stochastic decision making problem formulation discussed in this chapter is also the foundation of the Markov decision process (MDP) formulation that we will discuss in the following chapter.

20.1 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. Specifically, 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 discussed in Chapter 2 and Chapter 3, except we will 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 4.

In the deterministic decision making problem we model robot's state transition model[7] in *discrete-time* as:

$$\boldsymbol{x}_{k+1} = f_k(\boldsymbol{x}_k, \boldsymbol{u}_k), \quad k = 0, \ldots, N-1, \tag{20.1}$$

[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_k defines how the robot's state changes at time step k, and N is an integer that defines a finite planning horizon for the decision making problem. We also assume that only some control actions are admissible at a given state, which we denote by the set $\mathcal{U}(\boldsymbol{x}_k)$. 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 k by:

$$\boldsymbol{u}_k \in \mathcal{U}(\boldsymbol{x}_k). \tag{20.2}$$

Note that there are generally no restrictions on how the set of admissible control is defined. For example $\mathcal{U}(\boldsymbol{x}_k)$ could be a finite set of actions 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:

$$J(\boldsymbol{x}_0, \boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}) = g_N(\boldsymbol{x}_N) + \sum_{k=0}^{N-1} g_k(\boldsymbol{x}_k, \boldsymbol{u}_k), \tag{20.3}$$

where g_N is a terminal state cost function and g_k for $k = 0, \ldots, N-1$ are stage cost functions[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 20.1.1 (Deterministic Decision Making Problem)**.** The deterministic decision making problem for the state transition model in Equation (20.1), control constraints in Equation (20.2), and cost function in Equation (20.3) is to compute the finite horizon control sequence that is the solution to the optimiza-

tion problem:

$$\begin{aligned} J^*(\boldsymbol{x}_0) = \underset{\boldsymbol{u}_k,\ k=0,\ldots,N-1}{\text{minimize}}\ & J(\boldsymbol{x}_0, \boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}), \\ \text{s.t. } & \boldsymbol{x}_{k+1} = f_k(\boldsymbol{x}_k, \boldsymbol{u}_k), \quad k = 0, \ldots, N-1, \\ & \boldsymbol{u}_k \in \mathcal{U}(\boldsymbol{x}_k),\ k = 0, \ldots, N-1 \end{aligned} \tag{20.4}$$

The solution to Equation (20.4) is the optimal *open-loop* control sequence $\{\boldsymbol{u}_0^*, \ldots, \boldsymbol{u}_{N-1}^*\}$ given the initial condition $\boldsymbol{x}_0$, which is similar to the trajectory optimization problem in Definition 2.1.1. This problem is generally challenging to solve in practice since there is no guarantee that the state transition model in Equation (20.1) and cost function in Equation (20.3) 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 $\{\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1}\}$, but this leads to a combinatorial explosion of options and is therefore not practical except for very small problems.

20.1.1 *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. This underlying problem structure is referred to as the *principle of optimality*[9].

[9] Also referred to as *Bellman's principle of optimality*.

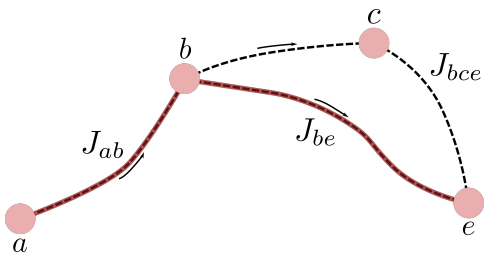


Figure 20.1: Starting from point a, let the red path $a \rightarrow b \rightarrow e$ be the optimal path from a to e, with a total cost of $J_{ae}^* = J_{ab} + J_{be}$. The principle of optimality says that the path $b \rightarrow e$ must therefore be the optimal path when starting from point b. We can prove this by contradiction, since if the path $b \rightarrow c \rightarrow e$ had a lower cost than path $b \rightarrow e$ such that $J_{bce} < J_{be}$, then the original path $a \rightarrow b \rightarrow e$ 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 20.1. 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 20.1.2.

Theorem 20.1.2 (Principle of Optimality (Deterministic Case)). *Let* $\{\boldsymbol{u}_0^*, \boldsymbol{u}_1^* \ldots, \boldsymbol{u}_{N-1}^*\}$ *be an optimal control sequence to the deterministic decision making problem in Equation* (20.4) *with a given initial condition* $\boldsymbol{x}_0^*$*, and let the resulting optimal state sequence be* $\{\boldsymbol{x}_0^*, \boldsymbol{x}_1^* \ldots, \boldsymbol{x}_N^*\}$*. The tail sequence* $\{\boldsymbol{u}_k^*, \ldots, \boldsymbol{u}_{N-1}^*\}$ *is then an optimal control sequence when starting from* $\boldsymbol{x}_k^*$ *and minimizing the cost:*

$$J_{tail}(\boldsymbol{x}_k, \boldsymbol{u}_k, \ldots, \boldsymbol{u}_{N-1}) = g_N(\boldsymbol{x}_N) + \sum_{i=k}^{N-1} g_i(\boldsymbol{x}_i, \boldsymbol{u}_i),$$

from time k to time N

We now demonstrate how the principle of optimality can be applied to simplify the decision making problem for the scenario in Figure 20.2. In this case our goal is to find an optimal path from point b to point f, and we assume that optimal paths from c, d, and e to f are already known. A brute force search over all possible paths in this problem would require 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 20.2 the goal point f is evaluated first, then the points c, d, and e, and then finally the point b.

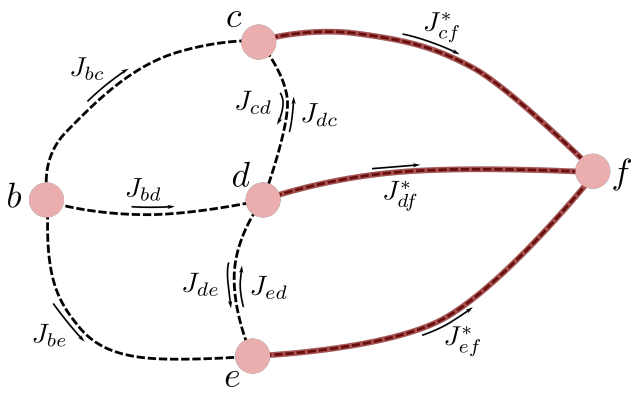


Figure 20.2: Suppose the optimal paths from points c, d and e to f are known (shown in red). Using the principle of optimality we can find the optimal path from point b to f by only searching over paths from b to c, d, and e, and determining the lowest cost from the candidates $\{J_{bc} + J^*_{cf}, J_{bd} + J^*_{df}, J_{be} + J^*_{ef}\}$. In other words, we can leverage the *optimal tails* to reduce the total number of paths that we need to be considere when finding the optimal path from b to f!

20.1.2 *Dynamic Programming (Deterministic Case)*

Dynamic programming (DP) is a foundational algorithm that leverages the principle of optimality[10] to *globally* solve the deterministic decision making problem in Equation (20.4). The dynamic programming algorithm, detailed in Algorithm 20.1, 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 $J^*_k(\boldsymbol{x}_k)$ for each time step $k = 0, \ldots, N$ and states $\boldsymbol{x}_k$, 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 $\{\boldsymbol{u}^*_0, \ldots, \boldsymbol{u}^*_{N-1}\}$ 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 20.1: Dynamic Programming (Deterministic)

$J_N^*(\boldsymbol{x}_N) = g_n(\boldsymbol{x}_N)$, for all $\boldsymbol{x}_N$
for $k = N-1$ **to** 0 **do**
  $J_k^*(\boldsymbol{x}_k) = \underset{\boldsymbol{u}_k \in \mathcal{U}(\boldsymbol{x}_k)}{\text{minimize}}\ g_k(\boldsymbol{x}_k, \boldsymbol{u}_k) + J_{k+1}^*(f_k(\boldsymbol{x}_k, \boldsymbol{u}_k))$, for all $\boldsymbol{x}_k$
return $J_0^*(\cdot), \ldots, J_N^*(\cdot)$

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 DP algorithm in Algorithm 20.1 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 20.1 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 20.1.1 (Deterministic Dynamic Programming)**.** Consider the environment shown in Figure 20.3 where the goal is to start at point a and reach point h while incurring the smallest cost. In this problem we represent the state as the current location and the 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 20.3.

For this problem we start the dynamic programming recursion at the goal point h with:

$$J_N^*(h) = 0,$$

since we assume there is no cost to stay at point h. Moving backward in time, we can see that the possible states $\boldsymbol{x}_{N-1}$ that can transition to $\boldsymbol{x}_N = h$ are the points h, e, and g, again assuming it is possible to stay at h with no cost. There-

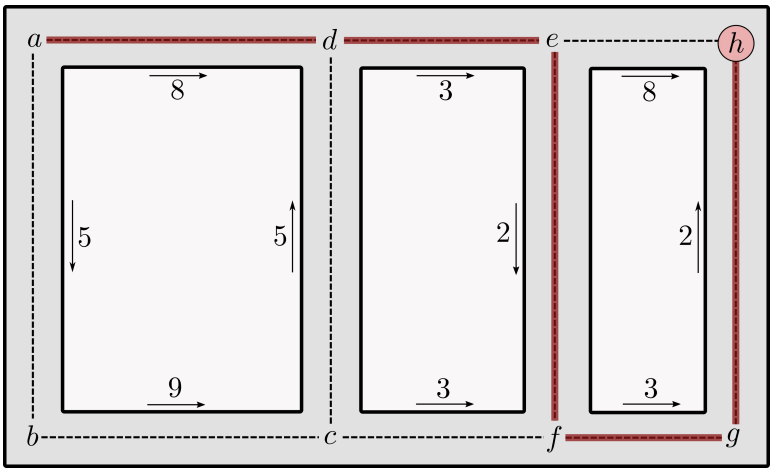


Figure 20.3: A deterministic decision making problem where the goal is to move from point a to point h while incurring the minimal amount of cost. The red path indicates the optimal path. We solve this problem by dynamic programming in Example 20.1.1.

fore in the first step of the DP recursion we compute:

$$\begin{aligned} J^*_{N-1}(h) &= 0 + J^*_N(h) = 0, \quad u^*_{N-1}(h) = \text{stay}. \\ J^*_{N-1}(e) &= 8 + J^*_N(h) = 8, \quad u^*_{N-1}(e) = \text{right}, \\ J^*_{N-1}(g) &= 2 + J^*_N(h) = 2, \quad u^*_{N-1}(g) = \text{up}, \end{aligned}$$

Note that $J^*_k(h) = 0$ for all $k \leq N$ and therefore we will not explicitly include it in the following steps. In the next step:

$$\begin{aligned} J^*_{N-2}(e) &= 8 + J^*_{N-1}(h) = 8, \quad u^*_{N-2}(e) = \text{right}, \\ J^*_{N-2}(g) &= 2, \quad u^*_{N-2}(g) = \text{up}, \\ J^*_{N-2}(d) &= 3 + J^*_{N-1}(e) = 11, \quad u^*_{N-2}(d) = \text{right}, \\ J^*_{N-2}(f) &= 3 + J^*_{N-1}(g) = 5, \quad u^*_{N-2}(f) = \text{right}, \end{aligned}$$

At this point these optimal tail costs 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:

$$\begin{aligned} J^*_{N-3}(e) &= \min\{8 + J^*_{N-2}(h),\, 2 + J^*_{N-2}(f)\} = 7, \quad u^*_{N-3}(e) = \text{down}, \\ J^*_{N-3}(g) &= 2, \quad u^*_{N-3}(g) = \text{up}, \\ J^*_{N-3}(d) &= 3 + J^*_{N-2}(e) = 11, \quad u^*_{N-3}(d) = \text{right}, \\ J^*_{N-3}(f) &= 5, \quad u^*_{N-3}(f) = \text{right}, \\ J^*_{N-3}(a) &= 8 + J^*_{N-2}(d) = 19, \quad u^*_{N-3}(a) = \text{right}, \\ J^*_{N-3}(c) &= \min\{5 + J^*_{N-2}(d),\, 3 + J^*_{N-2}(f)\} = 8, \quad u^*_{N-3}(c) = \text{right}. \end{aligned}$$

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 $a \rightarrow drightarrowerightarrow f$), 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 DP recursion:

$$\begin{aligned}
J^*_{N-4}(e) &= 7, \quad u^*_{N-4}(e) = \text{down}, \\
J^*_{N-4}(g) &= 2, \quad u^*_{N-4}(g) = \text{up}, \\
J^*_{N-4}(d) &= 3 + J^*_{N-3}(e) = 10, \quad u^*_{N-4}(d) = \text{right}, \\
J^*_{N-4}(f) &= 5, \quad u^*_{N-4}(f) = \text{right}, \\
J^*_{N-4}(a) &= 8 + J^*_{N-3}(d) = 19, \quad u^*_{N-4}(a) = \text{right} \\
J^*_{N-4}(c) &= \min\{5 + J^*_{N-3}(d),\, 3 + J^*_{N-3}(f)\} = 8, \quad u^*_{N-4}(c) = \text{right}, \\
J^*_{N-4}(b) &= 9 + J^*_{N-3}(c) = 17, \quad u^*_{N-4}(b) = \text{right},
\end{aligned}$$

and finally with one more iteration:

$$\begin{aligned}
J^*_{N-5}(e) &= 7, \quad u^*_{N-5}(e) = \text{down}, \\
J^*_{N-5}(g) &= 2, \quad u^*_{N-5}(g) = \text{up}, \\
J^*_{N-5}(d) &= 10, \quad u^*_{N-5}(d) = \text{right}, \\
J^*_{N-5}(f) &= 5, \quad u^*_{N-5}(f) = \text{right}, \\
J^*_{N-5}(a) &= \min\{8 + J^*_{N-4}(d),\, 5 + J^*_{N-4}(b)\} = 18, \quad u^*_{N-5}(a) = \text{right} \\
J^*_{N-5}(c) &= \min\{5 + J^*_{N-4}(d),\, 3 + J^*_{N-4}(f)\} = 8, \quad u^*_{N-5}(c) = \text{right}, \\
J^*_{N-5}(b) &= 9 + J^*_{N-4}(c) = 17, \quad u^*_{N-5}(b) = \text{right}.
\end{aligned}$$

Further iterations would no longer change the costs and optimal decisions so the algorithm has converged. We can finally see with a sufficiently long horizon, in this case $N \geq 5$, 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 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 $N = 3$ steps we can immediately see that the optimal path is $c \to f \to g \to h$ and the optimal cost is 8.

20.2 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 (20.1) may not be sufficient to achieve robust autonomy. Instead of leveraging the deterministic problem in Definition 20.1.1 we can instead consider a *stochastic* decision making problem that leverages a stochastic discrete-time state transition model:

$$\boldsymbol{x}_{k+1} = f_k(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w}_k), \quad k = 0, \ldots, N-1, \tag{20.5}$$

where $\boldsymbol{w}_k$ represents a stochastic disturbance. We also assume that the disturbance $\boldsymbol{w}_k$ has a known conditional probability distribution $p_k(\boldsymbol{w}_k \mid \boldsymbol{x}_k, \boldsymbol{u}_k)$ which

can change over time[12]. Note that we assume the disturbance is only dependent on the current state $\boldsymbol{x}_k$ and control $\boldsymbol{u}_k$, 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[13]:

$$J_\pi(\boldsymbol{x}_0) = E_{\boldsymbol{w}}\left[g_N(\boldsymbol{x}_N) + \sum_{k=0}^{N-1} g_k(\boldsymbol{x}_k, \pi_k(\boldsymbol{x}_k), \boldsymbol{w}_k)\right], \tag{20.6}$$

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, usually denoted $\boldsymbol{u} = \pi(\boldsymbol{x})$, 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 can now define the stochastic decision making problem in Definition 20.2.1.

Definition 20.2.1 (Stochastic Decision Making Problem)**.** The stochastic decision making problem for the stochastic state transition model in Equation (20.5), control constraints in Equation (20.2), and cost function in Equation (20.6) is to compute the finite horizon sequence of policies $\pi := \{\pi_0, \ldots, \pi_{N-1}\}$ that solves:

$$\begin{aligned} J^*(\boldsymbol{x}_0) = \underset{\pi}{\text{minimize}}\ & J_\pi(\boldsymbol{x}_0), \\ \text{s.t. } & \boldsymbol{x}_{k+1} = f_k(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w}_k), \quad k = 0, \ldots, N-1, \\ & \pi_k(\boldsymbol{x}_k) \in \mathcal{U}(\boldsymbol{x}_k),\ k = 0, \ldots, N-1 \end{aligned} \tag{20.7}$$

20.2.1 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 20.2.2 (Principle of Optimality (Stochastic Case))**.** *Let $\pi^* = \{\pi_0^*, \pi_1^* \ldots, \pi_{N-1}^*\}$ be an optimal policy for the stochastic decision making problem in 20.7 and assume the state $\boldsymbol{x}_k$ is reachable. Then the tail policy sequence $\{\pi_k^*, \ldots, \pi_{N-1}^*\}$ is an optimal policy sequence when starting from $\boldsymbol{x}_k$ to minimize the cost from time k to time N.*

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.

[12] Note that in the next chapter we introduce the *Markov decision process*, which is a stochastic decision making problem where the problem formulation represents the stochastic state transition model directly as the distribution $p(\boldsymbol{x}_{k+1} \mid \boldsymbol{x}_k, \boldsymbol{u}_k)$ rather than Equation (20.5).

[13] Using the expected value for the cost, which minimizes the cost *on average*, is often referred to as a *risk-neutral* formulation.

20.2.2 *Dynamic Programming (Stochastic Case)*

The dynamic programming algorithm for the stochastic setting, defined in Algorithm 20.2, is similar to Algorithm 20.1 for the deterministic case. This algorithm

Algorithm 20.2: Dynamic Programming (Stochastic Case)

$J_N^*(\boldsymbol{x}_N) = g_n(\boldsymbol{x}_N)$, for all $\boldsymbol{x}_N$
for $k = N-1$ **to** 0 **do**
  $J_k^*(\boldsymbol{x}_k) = \min_{\boldsymbol{u}_k \in \mathcal{U}(\boldsymbol{x}_k)} E_{\boldsymbol{w}_k}\big[g_k(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w}_k) + J_{k+1}^*(f_k(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w}_k))\big]$, for all $\boldsymbol{x}_k$
return $J_0^*(\cdot), \ldots, J_N^*(\cdot)$

computes the optimal costs $J_k^*(\boldsymbol{x})$ for each time step k and for all states $\boldsymbol{x}$. Once these values are known we can extract the optimal policy as:

$$\pi_k^*(\boldsymbol{x}_k) = \operatorname*{arg\,min}_{\boldsymbol{u}_k \in \mathcal{U}(\boldsymbol{x}_k)} E_{\boldsymbol{w}_k}\big[g_k(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w}_k) + J_{k+1}^*(f_k(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w}_k))\big]. \tag{20.8}$$

Example 20.2.1 (Stochastic Dynamic Programming)**.** Consider an inventory control problem where the available stock of a particular item is the state $x_k \in \mathbb{N}$, the ability to add to the inventory is the control $u_k \in \mathbb{N}$, and the demand for the item is a stochastic variable $w_k \in \mathbb{N}$. We model the dynamics of the available stock as:

$$x_{k+1} = \max\{0, x_k + u_k - w_k\},$$

which models the fact that demand reduces available stock but the stock can also never be negative. We also consider the control constraints:

$$x_k + u_k \leq 2,$$

which limits the amount of additional inventory that we can add based on the current available stock to ensure that $x_k \leq 2$. We model the stochastic demand w_k by the probability distribution:

$$p(w_k = 0) = 0.1, \quad p(w_k = 1) = 0.7, \quad p(w_k = 2) = 0.2.$$

Finally we define the cost for a horizon of $N = 3$ as:

$$E\big[\sum_{k=0}^{2} u_k + (x_k + u_k - w_k)^2\big],$$

which penalizes ordering new stock at each time step and also having available stock at the next time step 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:

$$J_2(0) = \underset{u_2 \in \{0,1,2\}}{\text{minimize}}\ E\left[u_2 + (u_2 - w_2)^2\right] = \underset{u_2 \in \{0,1,2\}}{\text{minimize}}\ u_2 + 0.1u_2^2 + 0.7(u_2 - 1)^2 + 0.2(u_2 - 2)^2 = 1.3,$$
$$J_2(1) = \underset{u_2 \in \{0,1\}}{\text{minimize}}\ E\left[u_2 + (1 + u_2 - w_2)^2\right] = 0.3,$$
$$J_2(2) = E\left[(2 - w_2)^2\right] = 1.1,$$

where the last cost is easily evaluated since the constraint makes 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:

$$J_1(0) = \underset{u_1 \in \{0,1,2\}}{\text{minimize}}\ E\left[u_1 + (u_1 - w_1)^2 + J_2(\max\{0, u_1 - w_1\})\right] = 2.5,$$
$$J_1(1) = \underset{u_1 \in \{0,1,\}}{\text{minimize}}\ E\left[u_1 + (1 + u_1 - w_1)^2 + J_2(\max\{0, 1 + u_1 - w_1\})\right] = 1.5,$$
$$J_1(2) = E\left[(2 - w_1)^2 + J_2(\max\{0, 2 - w_1\})\right] = 1.68,$$

with optimal stage policies:

$$\pi_1^*(0) = 1,$$
$$\pi_1^*(1) = 0,$$
$$\pi_1^*(2) = 0.$$

Finally, in the last step:

$$J_0(0) = \underset{u_0 \in \{0,1,2\}}{\text{minimize}}\ E\left[u_0 + (u_0 - w_0)^2 + J_1(\max\{0, u_0 - w_0\})\right] = 3.7,$$
$$J_0(1) = \underset{u_0 \in \{0,1,\}}{\text{minimize}}\ E\left[u_0 + (1 + u_0 - w_0)^2 + J_1(\max\{0, 1 + u_0 - w_0\})\right] = 2.7,$$
$$J_0(2) = E\left[(2 - w_0)^2 + J_1(\max\{0, 2 - w_0\})\right] = 2.818,$$

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.

20.3 Challenges and Extensions of Dynamic Programming

Dynamic programming is a powerful algorithm but suffers from several practical considerations referred to as the *curse of dimensionality*, the *curse of modeling*, and the *curse of time*. The curse of dimensionality is that there is an exponential growth of the computational and storage requirements based on the dimension of the state. For example if the state is n-dimensional and each state variable can take on M different discrete values then at each step of the DP 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. The curse of modeling is simplify the fact that it can be very practically challenging to model state transition probabilities of real-world stochastic systems. Lastly, the curse of time is that the data of the problem may not be known ahead of time and therefore the dynamic programming algorithm cannot be run offline. Therefore we may have to run the dynamic programming algorithm online as the data becomes available, or when the data changes and the problem needs to be resolved. In addition to these challenges it is also important to mention that some practical applications of decision making problems require an extension to account for *uncertain state information* in addition to uncertain state transitions[14]. For example this may be required if the entire state is not *observable* by direct measurements. In these settings 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 at each time step.

These challenges related to dynamic programming motivated the development of suboptimal dynamic programming approaches that make *approximations* to the original problem to make it more practical for specific settings, such as with high-dimensional states, when the model is not known, and more. Broadly speaking, there are two main categories of approximations. The first category includes approximations in the "value function" space where the optimal cost function is approximated. The second category includes approximations in the policy space where the policy is approximated by a parameterized function.

[14] In contexts where stochastic decision making problems are expressed as Markov decision processes (MDP) this extension is referred to as a *partially observable Markov decision process (POMDP).*

21
Reinforcement Learning

The previous chapter introduced the deterministic and stochastic sequential decision making problems, and demonstrated how these problems can be solved by dynamic programming. While dynamic programming is a powerful algorithm, it also suffers from several practical challenges. This chapter briefly introduces some of the key ideas in *reinforcement learning*[1,2], a set of ideas which aims to solve a more general problem of behaving in an optimal way within a given *unknown* environment. That is, the reinforcement learning setting assumes only the ability to (1) interact with an unknown environment and (2) receive a reward signal from it. How the actions affect the future state evolution or the future reward is not known *a priori*. Reinforcement learning includes a class of approximation algorithms which can be much more practical than dynamic programming in real world applications.

[1] D. Bertsekas. *Reinforcement learning and optimal control*. Athena Scientific, 2019

[2] R. Sutton and A. Barto. *Reinforcement learning: An introduction*. MIT Press, 2018

Reinforcement Learning

Reinforcement learning (RL) is a broad field that studies autonomous sequential decision making, but extends to more general and challenging problems than have been considered in previous chapters. The standard RL problem is to determine closed-loop control policies that drive an agent to maximize an accumulated reward[3]. However, in the general case it is not required that a *system model* be known! This paradigm can be represented by Figure 21.1, where it can be seen that given a control input the environment specifies the next state and reward, and the environment can be considered to be a black box (it is not necessarily known how the state is generated, nor the reward computed).

[3] Note that the maximization of "reward" in the context of reinforcement learning is essentially equivalent to minimization of "cost" in optimal control formulations.

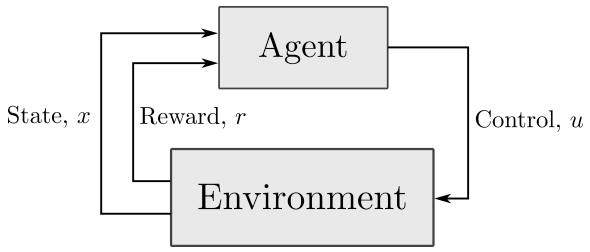


Figure 21.1: In reinforcement learning problems, the robot (agent) learns how to make decisions by interacting with the environment.

To account for this model uncertainty (which is notably distinct from the state transition uncertainty inherent in a stochastic but *known* system model, as considered in the previous chapter), an agent must instead learn from its experience to produce good policies. Concisely, RL deals with the problem of how to learn to act optimally in the long term, from interactions with an unknown environment which provides only a momentary reward signal.

RL is a highly active field of research, and has seen successes in several applications including acrobatic control of helicopters, games, finance, and robotics. In this chapter the fundamentals of reinforcement learning are introduced, including the formulation of the RL problem, RL algorithms that leverage system models ("model-based" methods; value iteration/dynamic programming and policy iteration), and a few RL algorithms that do not require system models ("model-free" methods; Q-learning, policy gradient, actor-critic).

21.1 Problem Formulation

The problem setting of reinforcement learning is similar to that of stochastic sequential decision making from the previous chapter, but here we will adopt slightly different notation more consistent with how Markov Decision Processes (MDPs) are typically framed in this community.[4] The state and control input for the system is denoted as $\boldsymbol{x}$ and $\boldsymbol{u}$, and the set of admissible states and controls are denoted as $\mathcal{X}$ and $\mathcal{U}$. However, the stochastic state transition model will now be written explicitly as a probability distribution (where before this was implicit in the influence of the stochastic variables $\boldsymbol{w}$ on the system dynamics f):

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}), \tag{21.1}$$

which is the conditional probability distribution over $\boldsymbol{x}_t$, given the previous state and control. The environment also has a reward function which defines the reward associated with every state and control

$$r_t = R(\boldsymbol{x}_t, \boldsymbol{u}_t). \tag{21.2}$$

The goal of the RL problem is to interact with the environment over a (possibly infinite) time horizon and *accumulate the highest possible reward in expectation*. To accommodate infinite horizon problems and to account for the fact that an agent is typically more confident about the ramifications of its actions in the short term than the long term, the accumulated reward is typically defined as the *discounted total expected reward* over time

$$E\left[\sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \boldsymbol{u}_t)\right], \tag{21.3}$$

where $\gamma \in (0, 1]$ is referred to as a *discount factor*. The tuple

$$\mathcal{M} = \left(\mathcal{X}, \mathcal{U}, p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}), R(\boldsymbol{x}_t, \boldsymbol{u}_t), \gamma\right)$$

[4] The fields of optimal control and reinforcement learning have significant overlap, but each community has developed its own standard notation. Most often, the state in the optimal control community is represented by $\boldsymbol{x}$ and in the RL community as $\boldsymbol{s}$. Similarly, in control theory the control input is $\boldsymbol{u}$ while in the RL community it is referred to as an action $\boldsymbol{a}$.

defines the Markov Decision Process (MDP), the environment in which the reinforcement learning problem is set.

In this chapter we will consider infinite horizon MDPs for which the notion of a stationary policy π applied at all time steps, i.e.,

$$\boldsymbol{u}_t = \pi(\boldsymbol{x}_t), \tag{21.4}$$

is appropriate. The goal of the RL problem is to choose a policy that maximizes the cumulative discounted reward

$$\pi^* = \arg\max_{\pi} \; E\left[\sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t)))\right] \tag{21.5}$$

where the expectation is notionally computed with respect to the stochastic dynamics p, but in practice is estimated empirically by drawing samples from the environment encoding $\mathcal{M}$ (i.e., in constructing π we may not assume exact knowledge of $\mathcal{M}$).

21.1.1 *Value function*

A policy π defines a value function which corresponds to the expected reward accumulated starting from a state $\boldsymbol{x}$

$$V^{\pi}(\boldsymbol{x}) = E\big[\sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \pi_t(\boldsymbol{x}_t)) \mid \boldsymbol{x}_0 = \boldsymbol{x}\big], \tag{21.6}$$

which can also be expressed in the *tail* formulation

$$V^{\pi}(\boldsymbol{x}) = R(\boldsymbol{x}, \pi(\boldsymbol{x})) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \pi(\boldsymbol{x})) V^{\pi}(\boldsymbol{x}'). \tag{21.7}$$

The optimal policy π^* satisfies *Bellman's equation*

$$\begin{aligned} V^{\pi^*}(\boldsymbol{x}) = V^*(\boldsymbol{x}) &= \max_{\boldsymbol{u} \in \mathcal{U}} \left(R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^*(\boldsymbol{x}') \right) \\ \pi^*(\boldsymbol{x}) &= \arg\max_{\boldsymbol{u} \in \mathcal{U}} \left(R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^*(\boldsymbol{x}') \right) \end{aligned} \tag{21.8}$$

and also satisfies $V^*(\boldsymbol{x}) = V^{\pi^*}(\boldsymbol{x}) \geq V^{\pi}(\boldsymbol{x})$ for all $\boldsymbol{x} \in \mathcal{X}$ for any alternative policy π. That is, the optimal policy induces the maximum value function and solves the RL problem of maximizing the accumulated discounted reward.

21.1.2 *Q-function*

Motivated by Bellman's equation above, in addition to the (state) value function $V^{\pi}(\boldsymbol{x})$ it makes sense to define the state-action value function $Q^{\pi}(\boldsymbol{x}, \boldsymbol{u})$ which corresponds to the expected reward accumulated starting from a state $\boldsymbol{x}$ and taking a first action $\boldsymbol{u}$ before following the policy π for all subsequent time steps. That is,

$$Q^{\pi}(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^{\pi}(\boldsymbol{x}'). \tag{21.9}$$

Similarly, the *optimal* Q-function is:

$$Q^*(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^*(\boldsymbol{x}'),$$

where the shorthand notation $Q^*(\boldsymbol{x}, \boldsymbol{u}) = Q^{\pi^*}(\boldsymbol{x}, \boldsymbol{u})$ is used. Note that from the Bellman equation (21.8) the optimal value function can be written as an optimization over the optimal Q-function:

$$V^*(\boldsymbol{x}) = \max_{\boldsymbol{u} \in \mathcal{U}} Q^*(\boldsymbol{x}, \boldsymbol{u}),$$

so,

$$Q^*(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}' \in \mathcal{X}} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) \max_{\boldsymbol{u}' \in \mathcal{U}} \Big(Q^*(\boldsymbol{x}', \boldsymbol{u}') \Big).$$

Therefore, instead of computing the optimal value function using value iteration it is possible to deal directly with the Q-function!

21.2 *Model-based Reinforcement Learning*

Model-based reinforcement learning methods rely on the use of an explicit parameterization of the transition model (21.1), which is either fit to observed transition data (i.e., learned) or, in special cases, known *a priori*. For example, for discrete state/control spaces it is possible to empirically approximate the transition probabilities $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1})$ for every pair $(\boldsymbol{x}_t, \boldsymbol{u}_t)$ by counting the number of times each transition occurs in the dataset! More sophisticated models include linear models generated through least squares, or Gaussian process or neural network models trained through an appropriate loss function. Given a learned model, the problem of optimal policy synthesis reduces to the sequential decision making problem of the previous chapter.

21.2.1 *Value Iteration (Dynamic Programming)*

While the dynamic programming algorithm was covered in the previous chapter, it will also be included here in the context of the RL problem formulation. In this case, the "principle of optimality" again says that the optimal *tail* policy is optimal for *tail* subproblems, which leads to the recursion:

$$V^*_{k+1}(\boldsymbol{x}) = \max_{\boldsymbol{u} \in \mathcal{U}} \Big(R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V^*_k(\boldsymbol{x}') \Big), \tag{21.10}$$

which is commonly referred to as the *Bellman recursion*. In words, the optimal reward associated with starting at the state $\boldsymbol{x}$ and having $k+1$ steps to go can be found as an optimization over the immediate control by accounting for the (expected) optimal tail rewards. The full dynamic programming algorithm for solving the RL problem (21.5) is given in Algorithm 21.1.

In the context of RL, this procedure is commonly referred to as *value iteration* and in many cases it is assumed that the horizon N is infinite. For infinite-horizon problems the "value iteration" in Algorithm 21.1 is performed either

Algorithm 21.1: Dynamic Programming/Value Iteration (RL)

$V_0^*(\boldsymbol{x}) = 0$, for all $\boldsymbol{x} \in \mathcal{X}$
for $k = 0$ **to** $N-1$ **do**
  $V_{k+1}^*(\boldsymbol{x}) = \max_{\boldsymbol{u}\in\mathcal{U}} R(\boldsymbol{x},\boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x},\boldsymbol{u}) V_k^*(\boldsymbol{x}')$, for all $\boldsymbol{x} \in \mathcal{X}$
  $\pi_{N-1-k}^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}\in\mathcal{U}} R(\boldsymbol{x},\boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x},\boldsymbol{u}) V_k^*(\boldsymbol{x}')$, for all $\boldsymbol{x} \in \mathcal{X}$
return $V_0^*(\cdot), \ldots, V_N^*(\cdot), \pi_0^*(\cdot), \ldots, \pi_{N-1}^*(\cdot)$

over a finite-horizon (which yields an approximate solution), or until convergence to a stationary (i.e. time-invariant) optimal value function/policy[5].

[5] In the infinite horizon case, the optimal value function is unique and the optimal policy is stationary and deterministic, but not necessarily unique.

To solidify the relationship between value iteration in the context of RL and dynamic programming in the context of stochastic decision making from the previous chapter, the inventory control example from the previous chapter is revisited:

Example 21.2.1 (Inventory Control)**.** Consider again the inventory control problem from the previous chapter, where the available stock of a particular item is the state $x_t \in \mathbb{N}$, the control $u_t \in \mathbb{N}$ adds items to the inventory, the demand w_t is uncertain, and the dynamics and constraints are:

$$x_t = \max\{0, x_{t-1} + u_{t-1} - w_{t-1}\},$$
$$p(w=0) = 0.1, \quad p(w=1) = 0.7, \quad p(w=2) = 0.2.$$

and

$$x_t + u_t \leq 2.$$

Based on the dynamics, the probabilistic model (21.1) is given by:

$$p(x_t = \{0,1,2\} \mid x_{t-1} = 0, u_{t-1} = 0) = \{1, 0, 0\},$$
$$p(x_t = \{0,1,2\} \mid x_{t-1} = 0, u_{t-1} = 1) = \{0.9, 0.1, 0\},$$
$$p(x_t = \{0,1,2\} \mid x_{t-1} = 0, u_{t-1} = 2) = \{0.2, 0.7, 0.1\},$$
$$p(x_t = \{0,1,2\} \mid x_{t-1} = 1, u_{t-1} = 0) = \{0.9, 0.1, 0\},$$
$$p(x_t = \{0,1,2\} \mid x_{t-1} = 1, u_{t-1} = 1) = \{0.2, 0.7, 0.1\},$$
$$p(x_t = \{0,1,2\} \mid x_{t-1} = 2, u_{t-1} = 0) = \{0.2, 0.7, 0.1\},$$

where some transition values are not explicitly written due to the control constraints. Next, the reward function is defined as:

$$\begin{aligned} R(x_t, u_t) &= -E\left[u_t + (x_t + u_t - w_t)^2\right], \\ &= -\left(u_t + (x_t + u_t - E[w_t])^2 + Var(w_t)\right), \end{aligned}$$

and a discount factor of $\gamma = 1$ is used. As in the previous chapter, this reward penalizes (a negative reward is a penalty) ordering new stock and having available stock at the next time step (i.e. having to store stock).

Algorithm 21.1 can now be applied, starting with the value function with no steps to go:

$$V_0^*(x) = 0,$$

and then recursively computing:

$$V_1^*(0) = \max_{u \in \{0,1,2\}} -\left(u + (u - 1.1)^2 + 0.29\right) = -1.3,$$
$$V_1^*(1) = \max_{u_2 \in \{0,1\}} -\left(u + (1 + u - 1.1)^2 + 0.29\right) = -0.3,$$
$$V_1^*(2) = -\left((2 - 1.1)^2 + 0.29\right) = -1.1,$$

where $E[w] = 1.1$ and $Var(w) = 0.29$. The optimal stage policies associated with this step are:

$$\pi_{N-1}^*(0) = 1, \quad \pi_{N-1}^*(1) = 0, \quad \pi_{N-1}^*(2) = 0.$$

In the next step:

$$V_2^*(0) = \max_{u \in \{0,1,2\}} -\left(u + (u - 1.1)^2 + 0.29\right) + \sum_{x'} p(x' \mid x = 0, u) V_1^*(x') = -2.5,$$
$$V_2^*(1) = \max_{u \in \{0,1,\}} -\left(u + (1 + u - 1.1)^2 + 0.29\right) + \sum_{x'} p(x' \mid x = 1, u) V_1^*(x') = -1.5,$$
$$V_2^*(2) = -\left((2 - 1.1)^2 + 0.29\right) + \sum_{x'} p(x' \mid x = 2, u = 0) V_1^*(x') = -1.68,$$

with optimal stage policies:

$$\pi_{N-2}^*(0) = 1, \quad \pi_{N-2}^*(1) = 0, \quad \pi_{N-2}^*(2) = 0.$$

Finally, in the last step:

$$V_3^*(0) = \max_{u \in \{0,1,2\}} -\left(u + (u - 1.1)^2 + 0.29\right) + \sum_{x'} p(x' \mid x = 0, u) V_2^*(x') = -3.7,$$
$$V_3^*(1) = \max_{u \in \{0,1,\}} -\left(u + (1 + u - 1.1)^2 + 0.29\right) + \sum_{x'} p(x' \mid x = 1, u) V_2^*(x') = -2.7,$$
$$V_3^*(2) = -\left((2 - 1.1)^2 + 0.29\right) + \sum_{x'} p(x' \mid x = 2, u = 0) V_2^*(x') = -2.818,$$

with optimal stage policies:

$$\pi_{N-3}^*(0) = 1, \quad \pi_{N-3}^*(1) = 0, \quad \pi_{N-3}^*(2) = 0.$$

These results are, in fact, identical to the results from the example in the previous chapter! The only difference is the formulation of the problem in the RL framework instead of the stochastic decision making problem framework.

21.2.2 *Policy Iteration*

Another common algorithm that can be used to solve the reinforcement learning problem (21.5) is *policy iteration*. The main idea of policy iteration is that if the value function can be computed for any arbitrary finite horizon policy $\pi = \{\pi_0, \pi_1, \ldots, \pi_{N-1}\}$, then the policy can be incrementally improved to yield a better policy $\pi' = \{\pi'_0, \pi'_1, \ldots, \pi'_{N-1}\}$.

Policy Evaluation: The first key element of the policy iteration algorithm is *policy evaluation*, which is used to compute the value function $V_k^\pi(\boldsymbol{x})$ for a given policy π. Policy evaluation is based on the recursion:

$$V_{k+1}^\pi(\boldsymbol{x}) = R(\boldsymbol{x}, \pi(\boldsymbol{x})) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \pi(\boldsymbol{x})) V_k^*(\boldsymbol{x}'), \tag{21.11}$$

which is very similar to the Bellman equation (21.8) except that there is no optimization over the control (since it is fixed). The policy evaluation algorithm is given in Algorithm 21.2.

Algorithm 21.2: Policy Evaluation

Data: π
Result: $V_0^\pi(\cdot), \ldots, V_N^\pi(\cdot)$
$V_0^\pi(\boldsymbol{x}) = 0$, for all $\boldsymbol{x} \in \mathcal{X}$
for $k = 0$ **to** $N - 1$ **do**
  $V_{k+1}^\pi(\boldsymbol{x}) = R(\boldsymbol{x}, \pi_{N-1-k}(\boldsymbol{x})) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \pi_{N-1-k}(\boldsymbol{x})) V_k^\pi(\boldsymbol{x}')$, for all $\boldsymbol{x} \in \mathcal{X}$
return $V_0^\pi(\cdot), \ldots, V_N^\pi(\cdot)$

In the infinite-horizon case where a stationary policy is used, the iteration in Algorithm 21.2 stops when the value function has converged to its stationary value. Indeed, since the infinite horizon value function is the stationary point of this recursion, it is possible to directly solve for it by setting both $V_{k+1}^\pi = V_k^\pi = V_\infty^\pi$ in (21.11). In the case of a discrete state space with N possible states, this creates a linear system of N equations which can be used to solve for V_∞^π directly.

Policy Iteration Algorithm: The policy iteration algorithm incrementally updates the policy by performing local optimizations of the Q-function. In particular, a single iteration of the policy update is shown in Algorithm 21.3. It can be

Algorithm 21.3: Policy Iteration Step

Data: π
Result: π'
$V_0^\pi(\cdot), \ldots, V_N^\pi(\cdot) \leftarrow$ PolicyEvaluation(π)
for $k = 0$ **to** $N - 1$ **do**
  $Q_{k+1}^\pi(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) V_k^\pi(\boldsymbol{x}')$ for all $\boldsymbol{x} \in \mathcal{X}$
  $\pi'_{N-1-k}(\boldsymbol{x}) = \arg\max_{\boldsymbol{u} \in \mathcal{U}} Q_{k+1}^\pi(\boldsymbol{x}, \boldsymbol{u})$, for all $\boldsymbol{x} \in \mathcal{X}$
return $\pi' = \{\pi'_0, \ldots, \pi'_{N-1}\}$

proven theoretically that under the policy iteration algorithm the value function is monotonically increasing with each new policy, and the procedure is run until convergence. While policy iteration and value iteration are quite similar, policy iteration can end up converging faster in some cases.

21.3 Model-free Reinforcement Learning

The value and policy iteration algorithms are applicable only to problems where the model $\mathcal{M}$ is *known*, i.e., they rely on direct access to the probabilistic system dynamics $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1})$ and reward function $R(\boldsymbol{x}_t, \boldsymbol{u}_t)$, or at least learned approximations of these functions fit to observed data. Model-free RL algorithms, on the other hand, sidestep the explicit consideration of p and R entirely.

21.3.1 Q-Learning

The canonical model-free reinforcement learning algorithm is *Q-learning*. The core idea behind Q-learning is that it is possible to collect data samples $(\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1})$ from interaction with the environment, and over time learn the long-term value of taking certain actions in certain states, i.e., directly learning the optimal Q-function $Q^*(\boldsymbol{x}, \boldsymbol{u})$. For simplicity an infinite-horizon ($N = \infty$) problem will be considered, such that the optimal value and Q-functions will be stationary, and in particular the optimal Q-function will satisfy:

$$Q^*(\boldsymbol{x}, \boldsymbol{u}) = R(\boldsymbol{x}, \boldsymbol{u}) + \gamma \sum_{\boldsymbol{x}'} p(\boldsymbol{x}' \mid \boldsymbol{x}, \boldsymbol{u}) \max_{\boldsymbol{u}' \in \mathcal{U}} Q^*(\boldsymbol{x}', \boldsymbol{u}').$$

In a model-free context, the dynamics p above are notional (i.e., the problem is described by some MDP $\mathcal{M}$, we just don't know exactly what it is).[6] We may instead rewrite the above equation in terms of an expectation over trajectory samples drawn from p (i.e., drawn from the environment as a "black box") while implementing the policy $\boldsymbol{u}_t = \pi^*(\boldsymbol{x}_t)$:

[6] Or even if we have a environment simulator, in which case it could be argued that the dynamics are exactly described by the simulation code, the dynamics are too complex/opaque to be considered in this form.

$$Q^*(\boldsymbol{x}_t, \boldsymbol{u}_t) = E\big[r_t + \gamma \max_{\boldsymbol{u}' \in \mathcal{U}} Q^*(\boldsymbol{x}_{t+1}, \boldsymbol{u}')\big],$$

or equivalently,

$$E\left[\left(r_t + \gamma \max_{\boldsymbol{u}' \in \mathcal{U}} Q^*(\boldsymbol{x}_{t+1}, \boldsymbol{u}')\right) - Q^*(\boldsymbol{x}, \boldsymbol{u})\right] = 0,$$

where $(r_t + \gamma \max_{\boldsymbol{u}' \in \mathcal{U}} Q^*(\boldsymbol{x}_{t+1}, \boldsymbol{u}')) - Q^*(\boldsymbol{x}, \boldsymbol{u})$ is known as the *temporal difference error*. The idea of Q-learning is that an approximation of the optimal Q-function can be improved over time by collecting data and trying to ensure that the above conditions holds. This leads to the Q-learning algorithm described in Algorithm 21.4. The iterations of Q-learning, each a local deterministic correction to the Q-function, in aggregate aim to ensure that the expected temporal difference error is 0.

Q-learning is referred to as a *model-free* method because it forgoes explicitly estimating the true (unknown) system dynamics, and directly estimates the Q-function. It is also called a *value-based* model-free method since it does not directly build the policy, but rather estimates the optimal Q-function to implicitly define the policy. Q-learning is also called an *off-policy* algorithm because the Q-function can be learned from stored experiences and does not require interacting with the environment directly.

Algorithm 21.4: Q-learning

Data: Set $\mathcal{S}$ of trajectory samples $\{\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1}\}$, learning rate α
Result: $Q(\boldsymbol{x}, \boldsymbol{u})$
Initialize $Q(\boldsymbol{x}, \boldsymbol{u})$ for all $\boldsymbol{x} \in \mathcal{X}$ and $\boldsymbol{u} \in \mathcal{U}$
for $\{\boldsymbol{x}_t, \boldsymbol{u}_t, r_t, \boldsymbol{x}_{t+1}\} \in \mathcal{S}$ **do**
  $Q(\boldsymbol{x}_t, \boldsymbol{u}_t) \leftarrow Q(\boldsymbol{x}_t, \boldsymbol{u}_t) + \alpha\Big(r_t + \gamma \max_{\boldsymbol{u} \in \mathcal{U}} Q(\boldsymbol{x}_{t+1}, \boldsymbol{u}) - Q(\boldsymbol{x}_t, \boldsymbol{u}_t)\Big)$
return $Q(\boldsymbol{x}, \boldsymbol{u})$

Q-learning can be guaranteed to converge to the optimal Q-function under certain conditions, but has some practical disadvantages. In particular, unless the number of possible states and controls are finite and relatively small, it can be intractable to store the Q-value associated with each state-control pair. Another disadvantage of Q-learning is that sometimes the Q-function can be complex and therefore potentially hard to learn.

Fitted Q-learning: One variation of the Q-learning algorithm to handle large or continuous state and control spaces is to parameterize the Q-function as $Q_\theta(\boldsymbol{x}, \boldsymbol{u})$ and to simply update the parameters $\boldsymbol{\theta}$. This approach is also known as *fitted Q-learning*. While this method often works well in practice, convergence is not guaranteed.

A principled way of performing fitted Q-learning involves minimizing the expected squared temporal difference error for the Q-function

$$E\left[\left(\left(r_t + \gamma \max_{u' \in \mathcal{U}} Q_\theta(\boldsymbol{x}_{t+1}, \boldsymbol{u}')\right) - Q_\theta(\boldsymbol{x}_t, \boldsymbol{u}_t)\right)^2\right].$$

For a given parameterization θ fitted Q-learning minimizes the total temporal difference error over all collected transition samples

$$\theta^* = \arg\min_\theta \frac{1}{|S_{\text{exp}}|} \sum_{(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{x}_{t+1}, r_t) \in S_{\text{exp}}} \left(r_t + \gamma \max_{u' \in \mathcal{U}} Q_\theta(\boldsymbol{x}_{t+1}, \boldsymbol{u}') - Q_\theta(\boldsymbol{x}_t, \boldsymbol{u}_t)\right)^2$$

where S_{exp} denotes the experience set of all transition tuples with a reward signal. This minimization is typically performed using stochastic gradient descent, yielding the parameter update

$$\theta \leftarrow \theta + \alpha \left(r_t + \gamma \max_{u' \in \mathcal{U}} Q_\theta(\boldsymbol{x}_{t+1}, \boldsymbol{u}') - Q_\theta(\boldsymbol{x}_t, \boldsymbol{u}_t)\right) \nabla_\theta Q_\theta(\boldsymbol{x}_t, \boldsymbol{u}_t)$$

applied iteratively for each $(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{x}_{t+1}, r_t) \in S_{\text{exp}}$.

21.3.2 Policy Gradient

The *policy gradient* method is another algorithm for model-free reinforcement learning. This approach, which directly optimizes the policy, can be particu-

larly useful for scenarios where the optimal policy may be relatively simple compared to the Q-function, in which case Q-learning may be challenging.

In the policy gradient approach, a class of *stochastic*[7] candidate policies $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t)$ is defined based on a set of parameters $\boldsymbol{\theta}$, and the goal is to *directly* modify the parameters $\boldsymbol{\theta}$ to improve performance. This is accomplished by using trajectory data to estimate a gradient of the performance with respect to the policy parameters $\boldsymbol{\theta}$, and then using the gradient to update $\boldsymbol{\theta}$. Because this method works directly on a policy (and does not learn a model or value function), it is referred to as a *model-free policy-based* approach.

[7] A stochastic policy defines a distribution over actions at a given state, is useful for exploration, and sometimes is even required for optimality.

Considering the original problem (21.5), the objective function can be written as:

$$J(\boldsymbol{\theta}) = E\big[\sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t))\big],$$

where the $J(\boldsymbol{\theta})$ notation is used to explicitly show the dependence on the parameters. Implementing a policy gradient approach therefore requires the computation of $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. One of the most common approaches is to *estimate* this quantity using data, using what is known as a *likelihood ratio method*.

Let τ represent a *trajectory* of the system (consisting of sequential states and actions) under the current policy $\pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t)$. As a shorthand notation, consider the total discounted reward over a trajectory τ to be defined written as:

$$r(\tau) = \sum_{t=0}^{\infty} \gamma^t R(\boldsymbol{x}_t, \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t), \tag{21.12}$$

such that $J(\boldsymbol{\theta})$ can be expressed equivalently as $J(\boldsymbol{\theta}) = E\big[r(\tau)\big]$. Additionally, let the probability that the trajectory τ occurs be expressed by the distribution $p_{\boldsymbol{\theta}}(\tau)$. Then the expectation from the objective function can be expanded as:

$$J(\boldsymbol{\theta}) = \int_{\tau} r(\tau) p_{\boldsymbol{\theta}}(\tau) d\tau,$$

and its gradient given by:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \int_{\tau} r(\tau) p_{\boldsymbol{\theta}}(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) d\tau.$$

From standard calculus $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) = \frac{1}{p_{\boldsymbol{\theta}}(\tau)} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau)$, which replaces the use of the gradient $\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau)$ with $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)$. This is a very useful "trick" when it comes to approximately computing the integral, as will be seen shortly.

Rather than explicitly computing this integral it is much easier to approximate using sampled data (i.e. sampled trajectories). This is possible since the integral can be written as the expectation $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = E\big[r(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)\big]$, which can be estimated using a Monte Carlo method. While in general a number of sampled trajectories could be used to estimate the gradient, for data efficiency it is also possible to just use a single sampled trajectory τ and approximate:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx r(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau). \tag{21.13}$$

In particular the sampled quantities $r(\tau)$ can be directly computed from (21.12), and it turns out that the term $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)$ can be evaluated quite easily as[8]:

$$\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau) = \sum_{t=0}^{N-1} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t). \tag{21.14}$$

[8] Using Bayes' rule: $p_{\boldsymbol{\theta}}(\tau) = p(\boldsymbol{x}_0) \Pi_{t=1}^{N-1} p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}) \pi_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{u}_{t-1})$. Then the log converts the product into a sum.

Importantly, notice that only the gradient of the policy is needed, and knowledge of the transition model $p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1})$ is not! This occurs because only the policy is dependent on the parameters $\boldsymbol{\theta}$.

In summary, the gradient of $J(\theta)$ can be approximated given a trajectories τ under the current policy $\pi_{\boldsymbol{\theta}}$ by:

1. Compute $r(\tau)$ for the sampled trajectory using (21.12).

2. Compute $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)$ for the sampled trajectory using (21.14), which only requires computing gradients related to the current policy $\pi_{\boldsymbol{\theta}}$.

3. Approximate $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ using (21.13).

The process of sampling trajectories from the current policy, approximating the gradient, and performing a gradient descent step on the parameters $\boldsymbol{\theta}$ is referred to as the *REINFORCE* algorithm[9].

In general, policy-based RL methods such as policy gradient can converge more easily than value-based methods, can be effective in high-dimensional or continuous action spaces, and can learn stochastic policies. However, one challenge with directly learning policies is that they can get trapped in undesirable local optima. Policy gradient methods can also be data inefficient since they require data from the *current* policy for each gradient step and cannot easily reuse old data. This is in contrast to Q-learning, which is agnostic to the policy used and therefore doesn't waste data collected from past interactions.

[9] There are some other modified versions of this algorithm, for example some contain a baseline term $b(\boldsymbol{x}_0)$ in the gradient by replacing $r(\tau)$ with $r(\tau) - b(\boldsymbol{x}_0)$ to reduce the variance of the Monte Carlo estimate.

21.3.3 Actor-Critic

Another popular reinforcement learning algorithm is the *actor-critic* algorithm, which blends the concepts of value-based and policy-based model-free RL. In particular, a parameterized policy $\pi_{\boldsymbol{\theta}}$ (actor) is learned through a policy gradient method along side an estimated value function for the policy (critic). The addition of the critic helps to reduce the variance in the gradient estimates for the actor policy, which makes the overall learning process more data-efficient[10].

In particular, the policy $\pi_{\boldsymbol{\theta}}$ is again learned through policy gradient like in the REINFORCE algorithm, but with the addition of a learned approximation of the value function $V_{\boldsymbol{\phi}}(\boldsymbol{x})$ as a baseline:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \sum_{t=0}^{N-1} \big(r(\tau) - V_{\boldsymbol{\phi}}(\boldsymbol{x}_0)\big) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\boldsymbol{u}_t \mid \boldsymbol{x}_t).$$

[10] This is a similar variance reduction approach to adding a baseline $b(\boldsymbol{x}_\tau)$ to the REINFORCE. In fact the baseline is chosen as the value function!

Recall that the value function $V(\boldsymbol{x})$ quantifies the expected total return starting from state $\boldsymbol{x}$ (i.e. the average performance). Therefore the quantity $r(\tau) - V_{\boldsymbol{\phi}}(\boldsymbol{x}_0)$ now represents a performance increase over average. Of course in this method the learned value function approximation $V_{\boldsymbol{\phi}}(\boldsymbol{x})$ is also updated along with the policy by performing a similar gradient descent on the parameters $\boldsymbol{\phi}$.

21.4 Deep Reinforcement Learning

Neural networks are a powerful function approximator that can be utilized in reinforcement learning algorithms.

Q-learning: In Q-learning the Q-function can be approximated by a neural network to extend the approach to nonlinear, continuous state space domains.

Policy Gradient: In policy gradient methods, the policy π_θ can be parameterized as a neural network, enabling the policy to operate on high-dimensional states including images (i.e. visual feedback)!

Actor-Critic: In actor-critic methods, both the policy π_θ and the value function V_ϕ can be parameterized as a neural network which often leads to a space efficient nonlinear representations of the policy and the value function.

21.5 Exploration vs Exploitation

When learning from experience (e.g. using Q-learning, policy gradient, actor-critic, deep RL, etc.) it is important to ensure that the experienced trajectories (i.e. the collected data points) are meaningful! For example, an abundance of data related to a particular set of actions/states will not necessarily be sufficient to learn good policies for *all* possible situations. Therefore an important part of reinforcement learning is *exploring* different combinations of states and actions. One simple approach to exploration is referred to as ϵ-greedy exploration, where a random control is applied instead of the current (best) policy with probability ϵ.

However, exploration can lead to suboptimal performance since any knowledge accumulated about the optimal policy is ignored[11]. This leads to the *exploration vs exploitation* trade-off: a fundamental challenge in reinforcement learning.

[11] In other words, actions with known rewards may be foregone in the hope that exploring leads to an even better reward.

22
Imitation Learning

As discussed in the previous chapter, the goal of reinforcement learning is to determine closed-loop control policies that result in the maximization of an accumulated reward, and RL algorithms are generally classified as either model-based or model-free. In both cases it is generally assumed that the reward function is known, and both typically rely on collecting system data to either update a learned model (model-based), or directly update a learned value function or policy (model-free).

While successful in many settings, these approaches to RL also suffer from several drawbacks. First, determining an appropriate reward function that can accurately represent the true performance objectives can be challenging[1]. Second, rewards may be *sparse*, which makes the learning process expensive in terms of both the required amount of data and in the number of failures that may be experienced when exploring with a suboptimal policy[2]. This chapter introduces the *imitation learning* approach to RL, where a reward function is not assumed to be known *a priori* but rather it is assumed the reward function is described implicitly through expert demonstrations.

[1] RL agents can sometimes learn how to exploit a reward function without actually producing the desired behavior. This is commonly referred to as *reward hacking*. Consider training an RL agent with a reward for each piece of trash collected. Rather than searching the area to find more trash (the desired behavior), the agent may decide to throw the trash back onto the ground and pick it up again!

[2] This issue of sparse rewards is less relevant if data is cheap, for example when training in simulation.

Imitation Learning

The formulation of the imitation learning problem is quite similar to the RL problem formulation from the previous chapter. The main difference is that instead of leveraging an explicit reward function $r_t = R(\boldsymbol{x}_t, \boldsymbol{u}_t)$ it will be assumed that a set of demonstrations from an expert are provided.

22.1 *Problem Formulation*

It will be assumed that the system is a Markov Decision Process (MDP) with a state $\boldsymbol{x}$ and control input $\boldsymbol{u}$, and the set of admissible states and controls are denoted as $\mathcal{X}$ and $\mathcal{U}$. The system dynamics are expressed by the probabilistic transition model:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{t-1}), \tag{22.1}$$

The field of RL often uses $\boldsymbol{s}$ to express the state and $\boldsymbol{a}$ to represent an action, but $\boldsymbol{x}$ and $\boldsymbol{u}$ will be used here for consistency with previous chapters.

which is the conditional probability distribution over $\boldsymbol{x}_t$, given the previous state and control. As in the previous chapter, the goal is to define a *policy* π that defines the closed-loop control law[3]:

[3] This chapter will consider a stationary policy for simplicity.

$$\boldsymbol{u}_t = \pi(\boldsymbol{x}_t). \tag{22.2}$$

The primary difference in formulation from the previous RL problem is that we do not have access to the reward function, and instead we have access to a set of expert demonstrations where each demonstration ξ consists of a sequence of state-control pairs:

$$\xi = \{(\boldsymbol{x}_0, \boldsymbol{u}_0), (\boldsymbol{x}_1, \boldsymbol{u}_1), \dots\}, \tag{22.3}$$

which are drawn from the expert policy π^*. The imitation learning problem is therefore to determine a policy π that imitates the expert policy π^*:

Definition 22.1.1 (Imitation Learning Problem)**.** For a system with transition model (22.1) with states $\boldsymbol{x} \in \mathcal{X}$ and controls $\boldsymbol{u} \in \mathcal{U}$, the imitation learning problem is to leverage a set of demonstrations $\Xi = \{\xi_1, \dots, \xi_D\}$ from an expert policy π^* to find a policy $\hat{\pi}^*$ that imitates the expert policy.

There are generally two approaches to imitation learning: the first is to directly learn how to imitate the expert's policy and the second is to indirectly imitate the policy by instead learning the expert's reward function. This chapter will first introduce two classical approaches to imitation learning (*behavior cloning* and the *DAgger* algorithm) that focus on directly imitating the policy. Then a set of approaches for learning the expert's reward function will be discussed, which is commonly referred to as *inverse reinforcement learning*. The chapter will then conclude with a couple of short discussions into related topics on learning from experts (e.g. through comparisons or physical feedback) as well as on interaction-aware control.

22.2 *Behavior Cloning*

Behavior cloning approaches use a set of expert demonstrations $\xi \in \Xi$ to determine a policy π that imitates the expert. This can be accomplished through supervised learning techniques, where the difference between the learned policy and expert demonstrations are minimized with respect to some metric. Concretely, the goal is to solve the optimization problem:

$$\hat{\pi}^* = \arg\min_{\pi} \sum_{\xi \in \Xi} \sum_{\boldsymbol{x} \in \xi} L(\pi(\boldsymbol{x}), \pi^*(\boldsymbol{x})),$$

where L is the cost function[4], $\pi^*(\boldsymbol{x})$ is the expert's action for at the state $\boldsymbol{x}$, and $\hat{\pi}^*$ is the approximated policy.

[4] Different loss functions could include p-norms (e.g. Euclidean norm) or f-divergences (e.g. KL divergence) depending on the form of the policy.

However this approach may not yield very good performance since the learning process is only based on a set of samples provided by the expert. In many cases these expert demonstrations will not be uniformly sampled across the

entire state space and therefore it is likely that the learned policy will perform poorly when not close to states found in ξ. This is particularly true when the expert demonstrations come from a *trajectory* of sequential states and actions, such that the *distribution* of the sampled states $\boldsymbol{x}$ in the dataset is defined by the expert policy. Then, when an estimated policy $\hat{\pi}^*$ is used in practice it produces its own distribution of states that will be visited, which will likely not be the same as in the expert demonstrations! This distributional mismatch leads to compounding errors, which is a major challenge in imitation learning.

22.3 DAgger: Dataset Aggregation

One straightforward idea for addressing the issue of distributional mismatch in states seen under the expert policy and the learned policy is to simply collect new expert data as needed[5]. In other words, when the learned policy $\hat{\pi}^*$ leads to states that aren't in the expert dataset just query the expert for more information! The behavioral cloning algorithm that leverages this idea is known as DAgger[6] (Dataset Aggregation).

[5] Assuming the expert can be queried on demand.

[6] S. Ross, G. Gordon, and D. Bagnell. "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011, pp. 627–635

Algorithm 22.1: DAgger: Dataset Aggregation

Data: π^*
Result: $\hat{\pi}^*$
$\mathcal{D} \leftarrow 0$
Initialize $\hat{\pi}$
for $i = 1$ **to** N **do**
  $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}$
  Rollout policy π_i to sample trajectory $\tau = \{\boldsymbol{x}_0, \boldsymbol{x}_1, \dots\}$
  Query expert to generate dataset $\mathcal{D}_i = \{(\boldsymbol{x}_0, \pi^*(\boldsymbol{x}_0)), (\boldsymbol{x}_1, \pi^*(\boldsymbol{x}_1)), \dots\}$
  Aggregate datsets, $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$
  Retrain policy $\hat{\pi}$ using aggregated dataset $\mathcal{D}$
return $\hat{\pi}$

As can be seen in Algorithm 22.1, this approach iteratively improves the learned policy by collecting additional data from the expert. This is accomplished by rolling out the current learned policy for some number of time steps and then asking the expert what actions they would have taken at each step along that trajectory. Over time this process drives the learned policy to better approximate the true policy and reduce the incidence of distributional mismatch. One disadvantage to the approach is that at each step the policy needs to be retrained, which may be computationally inefficient.

22.4 Inverse Reinforcement Learning

Approaches that learn policies to imitate expert actions can be limited by several factors:

1. Behavior cloning provides no way to understand the underlying reasons for the expert behavior (no reasoning about outcomes or intentions).
2. The "expert" may actually be suboptimal[7].
3. A policy that is optimal for the expert may not be optimal for the agent if they have different dynamics, morphologies, or capabilities.

[7] Although the discussion of inverse RL in this section will also assume the expert is optimal, there exist approaches to remove this assumption.

An alternative approach to behavioral cloning is to reason about and try to learn a representation of the underlying reward function R that the expert was using to generate its actions. By learning the expert's intent, the agent can potentially outperform the expert or adjust for differences in capabilities[8]. This approach (learning reward functions) is known as *inverse reinforcement learning*.

[8] Learned reward representations can potentially generalize across different robot platforms that tackle similar problems!

Inverse RL approaches assume a specific parameterization of the reward function, and in this section the fundamental concepts will be presented by parameterizing the reward as a linear combination of (nonlinear) features:

$$R(\boldsymbol{x}, \boldsymbol{u}) = \boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}, \boldsymbol{u}),$$

where $\boldsymbol{w} \in \mathbb{R}^n$ is a weight vector and $\boldsymbol{\phi}(\boldsymbol{x}, \boldsymbol{u}) : \mathcal{X} \times \mathcal{U} \to \mathbb{R}^n$ is a feature map. For a given feature map $\boldsymbol{\phi}$, the goal of inverse RL can be simplified to determining the weights $\boldsymbol{w}$. Recall from the previous chapter on RL that the total (discounted) reward under a policy π is defined for a time horizon T as:

$$V_T^\pi(\boldsymbol{x}) = E\big[\sum_{t=0}^{T-1} \gamma^t R(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t)) \mid \boldsymbol{x}_0 = \boldsymbol{x}\big].$$

Using the reward function $R(\boldsymbol{x}, \boldsymbol{u}) = \boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}, \boldsymbol{u})$ this value function can be expressed as:

$$V_T^\pi(\boldsymbol{x}) = \boldsymbol{w}^\top \mu(\pi, \boldsymbol{x}), \quad \mu(\pi, \boldsymbol{x}) = E_\pi\big[\sum_{t=0}^{T-1} \gamma^t \boldsymbol{\phi}(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t)) \mid \boldsymbol{x}_0 = \boldsymbol{x}\big],$$

where $\mu(\pi, \boldsymbol{x})$ is defined by an expectation over the trajectories of the system under policy π (starting from state $\boldsymbol{x}$) and is referred to as the *feature expectation*[9]. One insight that can now be leveraged is that by definition the optimal expert policy π^* will always produce a greater value function:

$$V_T^{\pi^*}(\boldsymbol{x}) \geq V_T^\pi(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \mathcal{X}, \quad \forall \pi,$$

[9] Feature expectations are often computed using a Monte Carlo technique (e.g. using the set of demonstrations for the expert policy).

which can be expressed in terms of the feature expectation as:

$$\boldsymbol{w}^{*\top} \mu(\pi^*, \boldsymbol{x}) \geq \boldsymbol{w}^{*\top} \mu(\pi, \boldsymbol{x}), \quad \forall \boldsymbol{x} \in \mathcal{X}, \quad \forall \pi. \tag{22.4}$$

Theoretically, identifying the vector $\boldsymbol{w}^*$ associated with the expert policy can be accomplished by finding a vector $\boldsymbol{w}$ that satisfies this condition. However this can potentially lead to ambiguities! For example, the choice $\boldsymbol{w} = 0$ satisfies this condition trivially! In fact, reward ambiguity is one of the main challenges associated with inverse reinforcement learning[10]. The algorithms discussed in the following chapters will propose techniques for alleviating this issue.

[10] A. Ng and S. Russell. "Algorithms for Inverse Reinforcement Learning". In: *Proceedings of the Seventeenth International Conference on Machine Learning*. 2000, pp. 663–670

22.4.1 *Apprenticeship Learning*

The apprenticeship learning[11] algorithm attempts to avoid some of the problems with reward ambiguity by leveraging an additional insight from condition (22.4). Specifically, the insight is that it doesn't matter how well $\boldsymbol{w}^*$ is estimated as long as a policy π can be found that *matches the feature expectations*. Mathematically, this conclusion is derived by noting that:

$$\|\mu(\pi, \boldsymbol{x}) - \mu(\pi^*, \boldsymbol{x})\|_2 \leq \epsilon \implies |\boldsymbol{w}^\top \mu(\pi, \boldsymbol{x}) - \boldsymbol{w}^\top \mu(\pi^*, \boldsymbol{x})| \leq \epsilon$$

for any $\boldsymbol{w}$ as long as $\|\boldsymbol{w}\|_2 \leq 1$. In other words, as long as the feature expectations can be matched then the performance will be as good as the expert *even if the vector $\boldsymbol{w}$ does not match $\boldsymbol{w}^*$*. Another practical aspect to the approach is that it will be assumed that the initial state $\boldsymbol{x}_0$ is drawn from a distribution D such that the value function is also considered in expectation as:

$$E_{\boldsymbol{x}_0 \sim D}\big[V_T^\pi(\boldsymbol{x}_0)\big] = \boldsymbol{w}^\top \mu(\pi), \quad \mu(\pi) = E_\pi\big[\sum_{t=0}^{T-1} \gamma^t \boldsymbol{\phi}(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t))\big].$$

This is useful to avoid having to consider all $\boldsymbol{x} \in \mathcal{X}$ when matching features[12].

To summarize, the goal of the apprenticeship learning approach is to find a policy π that matches the feature expectations with respect to the expert policy (i.e. makes $\mu(\pi)$ as similar as possible to $\mu(\pi^*)$)[13]. This is accomplished through Algorithm 22.2, which uses an iterative approach to finding better policies.

Algorithm 22.2: Apprenticeship Learning

Data: $\mu(\pi^*)$, ϵ
Result: $\hat{\pi}^*$
Initialize policy π_0
for $i = 1$ **to** $\ldots$ **do**
  Compute $\mu(\pi_{i-1})$ (or approximate via Monte Carlo)
  Solve problem (22.5) with policies $\{\pi_0, \ldots, \pi_{i-1}\}$ to compute $\boldsymbol{w}_i$ and t_i

$$(\boldsymbol{w}_i, t_i) = \arg\max_{\boldsymbol{w}, t} \; t, \quad \text{s.t. } \boldsymbol{w}^\top \mu(\pi^*) \geq \boldsymbol{w}^\top \mu(\pi) + t, \quad \forall \pi \in \{\pi_0, \ldots, \pi_{i-1}\}, \quad \|\boldsymbol{w}\|_2 \leq 1. \tag{22.5}$$

  if $t_i \leq \epsilon$ **then**
    $\hat{\pi}^* \leftarrow$ best feature matching policy from $\{\pi_0, \ldots, \pi_{i-1}\}$
    return $\hat{\pi}^*$
  Use RL to find an optimal policy π_i for reward function defined by $\boldsymbol{w}_i$

To better understand this algorithm it is useful to further examine the optimization problem (22.5)[14]. Suppose that instead of making $\boldsymbol{w}$ a decision variable

[11] P. Abbeel and A. Ng. "Apprenticeship Learning via Inverse Reinforcement Learning". In: *Proceedings of the Twenty-First International Conference on Machine Learning*. 2004

[12] Trying to find a policy that matches features for every possible starting state $\boldsymbol{x}$ is likely intractable or even infeasible.

[13] See Example 22.4.1 for an example of why matching features is intuitively useful.

[14] This problem can be thought of as an inverse RL problem that is seeking to find the reward function vector $\boldsymbol{w}$ such that the expert *maximally outperforms* the other policies.

it was actually fixed, then the resulting optimization would be:

$$t^*(\boldsymbol{w}) = \max_t \; t,$$
$$\text{s.t. } \boldsymbol{w}^\top \mu(\pi^*) \geq \boldsymbol{w}^\top \mu(\pi) + t, \quad \forall \pi \in \{\pi_0, \pi_1, \dots\},$$

which is essentially computing the smallest performance loss among the candidate policies $\{\pi_0, \pi_1, \dots\}$ with respect to the expert policy, *assuming the reward function weights are* $\boldsymbol{w}$. If $\boldsymbol{w}$ was known, then if $t^*(\boldsymbol{w}) \leq \epsilon$ it would guarantee that one of the candidate policies would effectively perform as well as the expert.

Since $\boldsymbol{w}$ is not known, the actual optimization problem (22.5) maximizes the smallest performance loss across *all vectors* $\boldsymbol{w}$ *with* $\|\boldsymbol{w}\|_2 \leq 1$. Therefore, if $t_i \leq \epsilon$ (i.e. the termination condition in Algorithm 22.2), then there must be a candidate policy whose performance loss is small *for all possible choices of* $\boldsymbol{w}$! In other words, there is a candidate policy that matches feature expectations well enough that good performance can be guaranteed without assuming the reward function is known, and without attempting to estimate the reward accurately.

Example 22.4.1 (Apprenticeship Learning vs. Behavioral Cloning)**.** Consider a problem where the goal is to drive a car across a city in as short of time as possible. In the imitation learning formulation it is assumed that the reward function is not known, but that there is an expert who shows how to drive across the city (i.e. what routes to take). A behavioral cloning approach would simply try to mimic the actions taken by the expert, such as memorizing that whenever the agent is at a particular intersection it should turn right. Of course this approach is not robust when at intersections that the expert never visited!

The apprenticeship learning approach tries to avoid the inefficiency of behavioral cloning by instead identifying features of the expert's trajectories that are more generalizable, and developing a policy that experiences the same feature expectations as the expert. For example it could be more efficient to notice that the expert takes routes without stop signs, or routes with higher speed limits, and then try to find policies that also seek out those features!

22.4.2 *Maximum Margin Planning*

The maximum margin planning approach[15] uses an optimization-based approach to computing the reward function weights $\boldsymbol{w}$ that is very similar to (22.5) but with some additional flexibility. In its most standard form the MMP optimization is:

$$\hat{w}^* = \arg\min_{\boldsymbol{w}} \; \|\boldsymbol{w}\|_2^2,$$
$$\text{s.t. } w^\top \mu(\pi^*) \geq w^\top \mu(\pi) + 1, \quad \forall \pi \in \{\pi_0, \pi_1, \dots\}.$$

Again this problem computes the reward function vector $\boldsymbol{w}$ such that the expert policy *maximally outperforms* the policies in the set $\{\pi_0, \pi_1, \dots\}$.

[15] N. Ratliff, J. A. Bagnell, and M. Zinkevich. "Maximum Margin Planning". In: *Proceedings of the 23rd International Conference on Machine Learning*. 2006, pp. 729–736

However the formulation is also improved in two ways: it adds a slack term to account for potential expert suboptimality and it adds a similarity function that gives more "margin" to policies that are dissimilar to the expert policy. This new formulation is:

$$\begin{aligned} \hat{w}^* = \underset{\boldsymbol{w},v}{\arg\min}\ & \|\boldsymbol{w}\|_2^2 + Cv, \\ \text{s.t. } & w^\top \mu(\pi^*) \geq w^\top \mu(\pi) + m(\pi^*, \pi) - v, \quad \forall \pi \in \{\pi_0, \pi_1, \dots\}, \end{aligned} \tag{22.6}$$

where v is a slack variable that can account for expert suboptimality, $C > 0$ is a hyperparameter that is used to penalize the amount of assumed suboptimality, and $m(\pi^*, \pi)$ is a function that quantifies how dissimilar two policies are.

One example of where this formulation is advantageous over the apprenticeship learning formulation (22.5) is when the expert is suboptimal. In this case it is possible that there is no $\boldsymbol{w}$ that makes the expert policy outperform all other policies, such that the optimization (22.5) returns $\boldsymbol{w}_i = 0$ and $t_i = 0$ (which is obviously not the appropriate solution). Alternatively the slack variables in the MMP formulation allow for a reasonable $\boldsymbol{w}$ to be computed.

22.4.3 *Maximum Entropy Inverse Reinforcement Learning*

While the apprenticeship learning approach shows that matching feature counts is a necessary and sufficient condition to ensure a policy performs as well as an expert, it also has some ambiguity (similar to the reward weight ambiguity problem discussed before). This ambiguity is associated with the fact that there could be different policies that lead to the same feature expectations!

This issue can also be thought of in a slightly more intuitive way in terms of distributions over trajectories. Specifically, a policy π induces a distribution over trajectories[16] $\tau = \{(\boldsymbol{x}_0, \pi(\boldsymbol{x}_0)), (\boldsymbol{x}_1, \pi(\boldsymbol{x}_1)), \dots\}$ that is denoted as $p_\pi(\tau)$. The feature expectations can be rewritten in terms of this distribution as:

$$\mu(\pi) = E_\pi\big[f(\tau)\big] = \int p_\pi(\tau) f(\tau) d\tau,$$

where $f(\tau) = \sum_{t=0}^{T-1} \gamma^t \phi(\boldsymbol{x}_t, \pi(\boldsymbol{x}_t))$. Now suppose a policy π was found that matched feature expectations[17] with an expert policy π^* such that:

$$\int p_\pi(\tau) f(\tau) d\tau = \int p_{\pi^*}(\tau) f(\tau) d\tau.$$

Crucially this condition is not sufficient to guarantee that $p_\pi(\tau) = p_{\pi^*}(\tau)$ (which would be ideal). In fact, the distribution $p_\pi(\tau)$ could also have an *arbitrary preference* for some paths that is *unrelated* to the feature matching objective.

The main idea in the maximum entropy inverse RL approach[18] is to not only match the feature expectations, but also remove ambiguity in the path distribution $p_\pi(\tau)$ by trying to make $p_\pi(\tau)$ *as broadly uncommitted as possible*. In other words, find a policy that matches feature expectations but otherwise has no additional path preferences. This concept is known as the maximum entropy principle[19].

[16] This distribution can be visualized as a set of paths generated by simulating the system many times with policy π (i.e. using a Monte Carlo method).

[17] For example by using apprenticeship learning.

[18] B. D. Ziebart et al. "Maximum Entropy Inverse Reinforcement Learning". In: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*. 2008, pp. 1433–1438

[19] A maximum entropy distribution can be thought of as the *least informative distribution* of a class of distribution. This is useful in situations where it is undesirable to encode unintended prior information.

The maximum entropy IRL approach finds a minimally preferential, feature expectation matching distribution by solving the optimization problem:

$$\begin{aligned} p^*(\tau) = \arg\max_p \; & \int -p(\tau)\log p(\tau)d\tau, \\ \text{s.t.} \; & \int p(\tau)f(\tau)d\tau = \int p_{\pi^*}(\tau)f(\tau)d\tau, \\ & \int p(\tau)d\tau = 1, \\ & p(\tau) \geq 0, \quad \forall \tau, \end{aligned} \tag{22.7}$$

where the objective is the mathematical definition of a distribution's entropy, the first constraint requires feature expectation matching, and the remaining constraints ensure that $p(\tau)$ is a valid probability distribution. It turns out that the solution to this problem has the exponential form:

$$p^*(\tau, \boldsymbol{\lambda}) = \frac{1}{Z(\boldsymbol{\lambda})} e^{\boldsymbol{\lambda}^\top f(\tau)}, \quad Z(\boldsymbol{\lambda}) = \int e^{\boldsymbol{\lambda}^\top f(\tau)} d\tau,$$

where $Z(\boldsymbol{\lambda})$ normalizes the distribution, and where $\boldsymbol{\lambda}$ must be chosen such that the feature expectations match:

$$\int p^*(\tau, \boldsymbol{\lambda}) f(\tau) = \int p_{\pi^*}(\tau) f(\tau) d\tau.$$

In other words the maximum entropy IRL approach tries to find a distribution parameterized by $\boldsymbol{\lambda}$ that match features, but also requires that the distribution $p^*(\tau, \boldsymbol{\lambda})$ belong to the exponential family.

To determine the value of $\boldsymbol{\lambda}$ that matches features, it is assumed that the expert also selects trajectories with high reward with exponentially higher probability:

$$p_{\pi^*}(\tau) \propto e^{\boldsymbol{w}^{*\top} f(\tau)},$$

and therefore ideally $\boldsymbol{\lambda} = \boldsymbol{w}^*$. Of course $\boldsymbol{w}^*$ (and more generally $p_{\pi^*}(\tau)$) are not known, and therefore a maximum likelihood estimation approach is used to compute $\boldsymbol{\lambda}$ to best approximate $\boldsymbol{w}^*$ based on the sampled expert demonstrations[20].

In particular, an estimate $\hat{\boldsymbol{w}}^*$ of the reward weights is computed from the expert demonstrations $\Xi = \{\xi_0, \xi_1, \dots\}$ (which each demonstration ξ_i is a trajectory) by solving the maximum likelihood problem:

$$\begin{aligned} \hat{\boldsymbol{w}}^* &= \arg\max_{\boldsymbol{\lambda}} \prod_{\xi_i \in \Xi} p^*(\xi_i, \boldsymbol{\lambda}), \\ &= \arg\max_{\boldsymbol{\lambda}} \sum_{\xi_i \in \Xi} \boldsymbol{\lambda}^\top f(\xi_i) - \log Z(\boldsymbol{\lambda}), \end{aligned}$$

which can be solved using a gradient descent algorithm where the gradient is computed by:

$$\nabla_{\boldsymbol{\lambda}} J(\boldsymbol{\lambda}) = \sum_{\xi_i \in \Xi} f(\xi_i) - E_{\tau \sim p^*(\tau, \boldsymbol{\lambda})}[f(\tau)].$$

[20] By assuming the expert policy is also exponential, the maximum likelihood estimate is theoretically *consistent* (i.e. $\boldsymbol{\lambda} \to \boldsymbol{w}^*$ as the number of demonstrations approaches infinity).

The first term of this gradient is easily computable since the expert demonstrations are known, and the second term can be approximated through Monte Carlo sampling. However, this Monte Carlo sampling estimate is based on sampling trajectories from the distribution $p^*(\tau, \boldsymbol{\lambda})$. This leads to a iterative algorithm:

1. Initialize $\boldsymbol{\lambda}$ and collect the set of expert demonstrations $\Xi = \{\xi_0, \xi_1, \dots\}$.
2. Compute the optimal policy[21] $\pi_{\boldsymbol{\lambda}}$ with respect to the reward function with $\boldsymbol{w} = \boldsymbol{\lambda}$.
3. Using the policy $\pi_{\boldsymbol{\lambda}}$, sample trajectories of the system and compute an approximation of $E_{\tau \sim p^*(\tau, \boldsymbol{\lambda})}[f(\tau)]$.
4. Perform a gradient step on $\boldsymbol{\lambda}$ to improve the maximum likelihood cost.
5. Repeat until convergence.

[21] For example through traditional RL methods.

To summarize, the maximum entropy inverse reinforcement learning approach identifies a distribution over trajectories that matches feature expectations with the expert, but by restricting the distribution to belong to the exponential family ensures that spurious preferences (path preferences not motivated by feature matching) are not introduced. Additionally, this distribution over trajectories is parameterized by a value that is an estimate of the reward function weights.

22.5 Learning From Comparisons and Physical Feedback

Both behavioral cloning and inverse reinforcement learning approaches rely on expert demonstrations of behavior. However in some practical scenarios it may actually be difficult for the expert to provide complete/quality demonstrations. For example it has been shown[22] that when humans are asked to demonstrate good driving behavior in simulation they retroactively think their behavior was too aggressive! As another example, if a robot has a high-dimensional control or state space it could be difficult for the expert to specify the full high-dimensional behavior. Therefore another interesting question in imitation learning is to find a way to learn from alternative data sources besides complete demonstrations.

[22] C. Basu et al. "Do You Want Your Autonomous Car to Drive Like You?" In: *12th ACM/IEEE International Conference on Human-Robot Interaction*. 2017, pp. 417–425

22.5.1 Learning from Comparisons

One alternative approach is to use *pairwise comparisons*[23], where an expert is shown two different behaviors and then asked to rank which behavior is better. Through repeated queries it is possible to converge to an understanding of the underlying reward function. For example, suppose two trajectories τ_A and τ_B are shown to an expert and that trajectory τ_A is preferred. Then assuming that the reward function is:

$$R(\tau) = \boldsymbol{w}^\top f(\tau),$$

[23] D. Sadigh et al. "Active Preference-Based Learning of Reward Functions". In: *Robotics: Science and System*. 2017

where $f(\tau)$ are the collective feature counts (same as in Section 22.4), this comparison can be used to conclude that:

$$\boldsymbol{w}^\top f(\tau_A) > \boldsymbol{w}^\top f(\tau_B).$$

In other words, this comparison has split the space of possible reward weights $\boldsymbol{w}$ in half through the hyperplane:

$$(f(\tau_A) - f(\tau_B))^\top \boldsymbol{w} = 0.$$

By continuously querying the expert with new comparisons[24], the space of possible reward weights $\boldsymbol{w}$ will continue to shrink until a good estimate of $\boldsymbol{w}^*$ can be made. In practice the expert decision may be a little noisy and therefore the hyperplanes don't define hard cutoffs, but rather can be used to "weight" the possible reward vectors $\boldsymbol{w}$.

[24] The types of comparisons shown can be selectively chosen to maximally split the remaining space of potential $\boldsymbol{w}$ in order to minimize the total number of expert queries that are required.

22.5.2 *Learning from Physical Feedback*

Another alternative to learning from complete expert demonstrations is to simply allow the expert to physically interact with the robot to correct for undesirable behavior[25]. In this approach, a physical interaction (i.e. a correction) is assumed to occur when the robot takes actions that result in a lower reward than the expert's action.

[25] A. Bajcsy et al. "Learning Robot Objectives from Physical Human Interaction". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 217–226

For a reward function of the form $R(\boldsymbol{x}, \boldsymbol{u}) = \boldsymbol{w}^\top \phi(\boldsymbol{x}, \boldsymbol{u})$ the robot maintains an estimate of the reward weights $\hat{\boldsymbol{w}}^*$ and the expert is assumed to have act according to a true set of optimal weights $\boldsymbol{w}^*$. Suppose the robot's policy, which is based on the estimated reward function with weights $\hat{\boldsymbol{w}}^*$, yields a trajectory τ_R. Then, if the expert physically interacts with the robot to make a correction the resulting actual trajectory τ_H is assumed to satisfy:

$$\boldsymbol{w}^{*\top} f(\tau_H) \geq \boldsymbol{w}^{*\top} f(\tau_R),$$

which simply states that the reward of the new trajectory is higher. This insight is then leveraged in a maximum a posteriori approach for updating the estimate $\hat{\boldsymbol{w}}^*$ after each interaction. Specifically, this update takes the form:

$$\hat{\boldsymbol{w}}^* \leftarrow \hat{\boldsymbol{w}}^* + \beta(\boldsymbol{f}(\tau_H) - \boldsymbol{f}(\tau_R)),$$

where $\beta > 0$ is a scalar step size. The robot then uses the new estimate to change its policy, and the process iterates. Note that this idea yields an approach that is similar to the concept of matching feature expectations from inverse reinforcement learning, except that the approach is iterative rather than requiring a batch of complete expert demonstrations.

22.6 *Interaction-aware Control and Intent Inference*

Yet another interesting problem in robot autonomy arises when robots and humans are interacting to accomplish shared or individual goals. Many classical

examples of this problem arise in autonomous driving settings, when human-driven vehicles interact with autonomous vehicles in settings such as highway merging or at intersections. While the imitation learning problems from the previous sections are focused on understanding the expert's behavior for the purpose of *imitating* the behavior, in this setting the human's behavior needs to be understood in order to ensure safe interactions. However there is an additional component to understanding interactions: *the robot's behavior can influence the human's behavior*[26].

[26] It is particularly important in interaction-aware robot control to understand the effects of the robot's actions on the human's behavior. Otherwise the human's could simply be modeled as dynamic obstacles!

22.6.1 Interaction-aware Control with Known Human Model

One common approach is to model the interaction between humans and robots as a dynamical system that has a combined state $\boldsymbol{x}$, where the robot controls are denoted $\boldsymbol{u}_R$ and the human decisions or inputs are denoted as $\boldsymbol{u}_H$. The transition model is therefore defined as:

$$p(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{u}_{R,t-1}, \boldsymbol{u}_{H,t-1}).$$

In other words the interaction dynamics evolve according to the actions taken by both the robot and the human. In this interaction the robot's reward function is denoted as $R_R(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H)$ and the human's reward function is denoted as $R_H(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H)$, which are both functions of the combined state and both agent's actions[27].

[27] While R_R and R_H do not have to be the same, choosing $R_R = R_H$ may be desirable for the robot to achieve human-like behavior.

Under the assumption that both the robot and the human act optimally[28] with respect to their cost functions:

$$\boldsymbol{u}_R^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}_R} \; R_R(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H^*(\boldsymbol{x})),$$
$$\boldsymbol{u}_H^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}_H} \; R_H(\boldsymbol{x}, \boldsymbol{u}_R^*(\boldsymbol{x}), \boldsymbol{u}_H).$$

[28] While not necessarily true, this assumption is important to make the resulting problem formulation tractable to solve in practice.

Additionally, assuming both reward functions R_R and R_H are known[29], computing $\boldsymbol{u}_R^*$ is still extremely challenging due to the two-player game dynamics of the decision making process. However this problem can be made more tractable by modeling it as a *Stackelberg game*, which restricts the two-player game dynamics to a leader-follower structure. Under this assumption it is assumed that the robot is the "leader" and that as the follower the human acts according to:

[29] The reward function R_H could be approximated using inverse reinforcement learning techniques.

$$\boldsymbol{u}_H^*(\boldsymbol{x}, \boldsymbol{u}_R) = \arg\max_{\boldsymbol{u}_H} \; R_H(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H). \tag{22.8}$$

In other words the human is assumed to see the action taken by the robot *before* deciding on their own action. The robot policy can therefore be computed by solving:

$$\boldsymbol{u}_R^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}_R} \; R_R(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H^*(\boldsymbol{x}, \boldsymbol{u}_R)), \tag{22.9}$$

which can be solved using a gradient descent approach. For the gradient descent approach the gradient of:

$$J(\boldsymbol{x}, \boldsymbol{u}_R) = R_R(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H^*(\boldsymbol{x}, \boldsymbol{u}_R)),$$

can be computed using the chain rule as:

$$\frac{\partial J}{\partial \boldsymbol{u}_R} = \frac{\partial R_R}{\partial \boldsymbol{u}_R} + \frac{\partial R_R}{\partial \boldsymbol{u}_H^*} \frac{\partial \boldsymbol{u}_H^*}{\partial \boldsymbol{u}_R}.$$

Since the reward function R_R is known the terms $\partial R_R / \partial \boldsymbol{u}_R$ and $\partial R_R / \partial \boldsymbol{u}_H^*$ can be easily determined. In order to compute the term $\partial \boldsymbol{u}_H^* / \partial \boldsymbol{u}_R$, which represents how much the robot's actions impact the human's actions, an additional step is required. First, assuming the human acts optimally according to (22.8) the necessary optimality condition is:

$$g(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H^*) = 0, \quad g = \frac{\partial R_H}{\partial \boldsymbol{u}_H},$$

which for the fixed values of $\boldsymbol{x}$ and $\boldsymbol{u}_R$ specifies $\boldsymbol{u}_H^*$. Then, by implicitly differentiating this condition with respect to the robot action $\boldsymbol{u}_R$:

$$\frac{\partial g}{\partial \boldsymbol{u}_R} + \frac{\partial g}{\partial \boldsymbol{u}_H^*} \frac{\partial \boldsymbol{u}_H^*}{\partial \boldsymbol{u}_R} = 0,$$

which can be used to solve for:

$$\frac{\partial \boldsymbol{u}_H^*}{\partial \boldsymbol{u}_R}(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H^*) = -\left(\frac{\partial g}{\partial \boldsymbol{u}_H^*}\right)^{-1} \frac{\partial g}{\partial \boldsymbol{u}_R}.$$

Notice that every term in this expression can be computed[30] and therefore it can be substituted into the gradient calculation:

$$\frac{\partial J}{\partial \boldsymbol{u}_R} = \frac{\partial R_R}{\partial \boldsymbol{u}_R} - \frac{\partial R_R}{\partial \boldsymbol{u}_H^*} \left(\frac{\partial g}{\partial \boldsymbol{u}_H^*}\right)^{-1} \frac{\partial g}{\partial \boldsymbol{u}_R},$$

which can then be computed as long as it is possible to compute $\boldsymbol{u}_H^*(\boldsymbol{x}, \boldsymbol{u}_R)$.

To summarize, one approach to interaction-aware control is to model the interaction as a Stackelberg game, where it is assumed that both the human and the robot act optimally with respect to some reward functions. This formulation of the problem enables the robot to choose actions based on an implicit understanding of how the human will react.

[30] Assuming the human's reward function is known.

22.6.2 *Intent Inference*

One disadvantage to the approach for interaction-aware control from the previous section is that it assumes the human acts optimally with respect to a *known* reward function. While a reward function could be learned through inverse reinforcement learning, this is not practical for real-world settings where different humans behave differently. Returning to the example of interaction between human drivers and autonomous vehicles, the human could exhibit drastically different behavior depending on whether they have an aggressive or passive driving style. In these settings the problem of *intent inference* focuses on identifying underlying behavioral characteristics that can lead to more accurate behavioral models[31].

[31] This problem can be formulated as a partially observable Markov decision process (POMDP) since the underlying behavioral characteristic is not directly observable, yet influences the system's behavior.

One approach to intent inference[32] is to model the underlying behavioral differences through a set of unknown parameters $\boldsymbol{\theta}$ which need to be inferred by observing the human's behavior. Mathematically this is expressed by defining the human's reward function $R_H(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H, \boldsymbol{\theta})$ to be a function of $\boldsymbol{\theta}$, and assuming the human chooses actions according to:

$$p(\boldsymbol{u}_H \mid \boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{\theta}) \propto e^{R_H(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H, \boldsymbol{\theta})}.$$

In other words this model assumes the human is exponentially more likely to pick optimal actions[33], but that they may pick suboptimal actions as well.

The objective of intent inference is therefore to estimate the parameters $\boldsymbol{\theta}$, which can be accomplished through Bayesian inference methods. In the Bayesian approach a *probability distribution* over parameters $\boldsymbol{\theta}$ is updated based on observations. Specifically the belief distribution is denoted as $b(\boldsymbol{\theta})$, and given an observation of the human's actions $\boldsymbol{u}_H$ the belief distribution is updated as:

$$b_{t+1}(\boldsymbol{\theta}) = \frac{1}{\eta} p(\boldsymbol{u}_{H,t} \mid \boldsymbol{x}_t, \boldsymbol{u}_{R,t}, \boldsymbol{\theta}) b_t(\boldsymbol{\theta}),$$

where η is a normalizing constant. This Bayesian update is simply taking the prior belief over $\boldsymbol{\theta}$ and updating the distribution based on the likelihood of observing human action $\boldsymbol{u}_H$ under that prior. Note that this concept is quite similar to the concepts of inverse reinforcement learning: a set of parameters that describe the human's (experts) behavior are continually updated when new observations of their actions are gathered.

While the robot could sit around and *passively* observe the human act to collect samples for the Bayesian updates, it is often more efficient for the robot to *probe* the human to take interesting actions that are more useful for revealing the intent parameters $\boldsymbol{\theta}$. This can be accomplished by choosing the robot's reward function to be:

$$R_R(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H, \boldsymbol{\theta}) = I(b(\boldsymbol{\theta}), \boldsymbol{u}_R) + \lambda R_{\text{goal}}(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H, \boldsymbol{\theta})$$

where $\lambda > 0$ is a tuning parameter and $I(b(\boldsymbol{\theta}), \boldsymbol{u}_R)$ denotes a function that quantifies the amount of information gained with respect to the belief distribution from taking action $\boldsymbol{u}_R$. In other words the robot's reward is a tradeoff between exploiting the current knowledge of $\boldsymbol{\theta}$ to accomplish the objective and taking exploratory actions to improve the intent inference. With this robot reward function the robot's actions are chosen to maximize the expected reward:

$$\boldsymbol{u}_R^*(\boldsymbol{x}) = \arg\max_{\boldsymbol{u}_R} \; E_{\boldsymbol{\theta}}[R_R(\boldsymbol{x}, \boldsymbol{u}_R, \boldsymbol{u}_H, \boldsymbol{\theta})].$$

To summarize, this robot policy will try to simultaneously accomplish the robot's objective and gather more information to improve the inference of the human's intent (modeled through the parameters $\boldsymbol{\theta}$). In a highway lane changing scenario this type of policy might lead the robot to nudge into the other lane to see if the other car will slow down (passive driving behavior) or try to

[32] D. Sadigh et al. "Planning for cars that coordinate with people: leveraging effects on human actions for planning and active information gathering over human internal state". In: *Autonomous Robots* 42.7 (2018), pp. 1405–1426

[33] This assumption was also used in the Maximum Entropy IRL approach.

block the lane change (aggressive driving behavior). Once the robot has a strong enough belief about the human's behavior it may choose to either complete the lane change or slow down to merge behind the human driver.

Part V

Robot Software

23
Robot System Architectures

A robotic system is fundamentally just a collection of sensors and actuators that can interact with the environment to accomplish a set of tasks. While this definition may seem simple, the systems required to implement this definition tend to be extremely complex due to the infinite variability and uncertainty of real-world environments and the diversity among sensors and actuators. Therefore, careful and practical design of robotic systems is crucial for managing complexity, and as a byproduct enabling robust and successful robotic operations. This chapter will introduce some of the fundamental concepts, paradigms, and tools in the design of robot system architectures to enable full robot autonomy while also managing system complexity[1].

[1] D. Kortenkamp, R. Simmons, and D. Brugali. "Robotic Systems Architectures and Programming". In: *Springer Handbook of Robotics*. Springer, 2008, pp. 283–302

Robot System Architectures

The primary objective of a robotic system is to accomplish a specific set of tasks, but there are often many peripheral tasks that must also be handled to ensure the robot operates in a safe and robust way. For example a robot's goal may be to pick up objects and place them in certain locations, but in order to accomplish this task the robot should also be aware of obstacles (static or dynamic) in its environment, should be robust to sensor failures or sensor noise, and more.

Definition 23.0.1 (Robot Goal)**.** Complete desired tasks while monitoring and reacting to unexpected situations. Handle inputs and outputs (control/perception) from actuators and sensors in real-time[2] and under uncertainty.

[2] Real-time requirements are crucial, some situations require near instantaneous reactions (e.g. less than 1 ms reaction time).

The design of the robot's system architecture is important for enabling the robot to achieve its goal without requiring extremely complex software systems for implementation. In general, the *system architecture* is defined by two major parts: the *structure* and the *style*. The structure defines the way in which the system is broken down into components, as well as how the components interact with each other[3]. Alternatively the style of the architecture refers to the computational concepts that define the implementation of the design.

[3] The structure could be represented visually as a diagram of boxes (components) that are connected by arrows (interactions).

Generally speaking there is no specific architecture that is optimal for every robotic system, but there are some paradigms that have been proven to be use-

ful, which will be introduced in more detail in the following sections. In fact, any given system architecture may consist of multiple types of structures or styles! For a given robot, the specific choice of architecture should aim to reduce complexity[4] while not being overly restrictive and thus limit performance.

[4] For example subsystem segmentation can be useful for reusability as well as validation and unit-testing.

23.1 Architecture Structures

The architecture's *structure* defines how the system is subdivided into subsystems and how the subsystems interact. Some form of hierarchical structure is commonly used for this decomposition, which reduces complexity through abstraction (e.g. tasks at one level are of the hierarchy are composed of a group of tasks from lower-levels of the hierarchy).

23.1.1 Sense-Plan-Act Architecture

This architecture is one of the first developed, and consists of three main subsystems: sensing, planning, and execution. These components were organized in a sequential fashion, with sensor data being passed to the planner, who then passes information to the controller, who sends actuator commands. However this approach has significant drawbacks. First, the planning component was a computational bottleneck that held up the controller subsystem. Second, since the controller did not have direct access to sensor data the overall system was not very *reactive*.

23.1.2 Subsumption Architecture

An alternative to the sense-plan-act architecture that emerged not long after is the *subsumption architecture*[5]. This architecture decomposes the overall desired robot behavior into sub-behaviors in a bottom-up fashion. In this hierarchical structure the higher-level behaviors *subsume* the lower-level behaviors. In other words, the high-level behaviors can outsource smaller scale tasks to be handled by the low-level behaviors. From an implementation standpoint this architecture can be thought of as layers of finite state machines[6] that all connect sensors to actuators, and where multiple behaviors are evaluated in parallel. An arbitration mechanism is also included to choose which of the behaviors is currently activated. For example an explore behavior may sit on top of (subsume) a collision avoidance behavior, and the arbitration mechanism would decide when the exploration behavior should be overridden by the collision avoidance behavior.

[5] R. Brooks. "A robust layered control system for a mobile robot". In: *IEEE Journal on Robotics and Automation* 2.1 (1986), pp. 14–23

[6] Each finite state machine was often referred to as a *behavior*.

While this architecture is much more reactive than the sense-plan-act architecture, there are also disadvantages. The primary disadvantage of this approach is that there is no good way to do long-term planning or behavior optimization. This can make it challenging to design the system to accomplish long-term objectives.

23.1.3 Three-tiered Architecture

The *three-tiered architecture* is one of the most commonly used architectural designs. This architecture contains a planning, an executive, and a behavioral control level that are hierarchically linked.

1. *Planning*: this layer is at the highest-level, and focuses on task-planning for long-term goals.

2. *Executive*: the executive layer is the middle layer connecting the planner and the behavioral control layers. The executive specifies priorities for the behavioral layer to accomplish a specific task. While the task may come directly from the planning layer, the executive can also split higher-level tasks into sub-tasks.

3. *Behavioral control*: at the lowest-level. the behavioral control layer handles the implementation of low-level behaviors and is the interface to the robot's actuators and sensors.

The primary advantage of this architecture is that it combines benefits of the behavioral-based subsumption architecture (i.e. reactive planning) with better long-term planning capabilities (i.e. resulting from the planning level). Each of these levels will now be discussed in slightly further detail, however in practice the division among these levels is often quite blurred!

Behavioral Control Level: The components at the behavioral control level typically focus on small, localized behaviors or skills and directly interface with the robot's sensors and actuators[7]. These behaviors are typically *situated*, meaning that they only make sense with respect to a specific situation that the robot may be in. Importantly, the behavioral control components should have an awareness of the current situation (i.e. they should be able to identify if the current situation is appropriate for a specific behavior), but they are not responsible for knowing how to *change* the situation (this is left to the executive level).

[7] This layer includes algorithms from classical control theory: PID control, Kalman filtering, etc.

The tight interaction between the sensors and actuators in the behavioral control level enables a high level of reactivity in this architecture. However, high reactivity also requires that the behavioral control level not incorporate algorithms with high computational complexity. In general, the algorithms at this level should be able to operate *at least* several times per second.

Executive Level: The components of the executive level are responsible for translating high-level plans into low-level behaviors, orchestrating when low-level behaviors are executed, as well as monitoring for and handling exceptions. This component is typically implemented as a hierarchical finite state machine, but might also incorporate motion planning and decision making algorithms to break a high-level task into a sequence of smaller tasks. To orchestrate the sequence and timing for behaviors to be implemented, the executive considers

temporal constraints on behaviors (e.g. whether two actions can be executed concurrently).

Planning Level: Finally, the planning level focuses on high-level decision making and planning for long-term behavior. This forward-thinking component is crucial to optimize the long-term behavior of the robot. However, the implementation of the decisions from the planner are deferred to the executive layer. In practice it might also be useful to have multiple planning levels, for example to split up mission level planning (very abstract planning) with shorter horizon planning[8].

[8] This split might be useful for computational performance reasons.

Example 23.1.1 (Office Mail Delivery Robot)**.** To further explore the components of the three-tiered robot system architecture, consider a robot whose primary task is to deliver mail within an office setting. In general, tasks that might be required of this robot include: the ability to move through hallways and rooms, avoid humans and other obstacles, open and close doors, announce a delivery, find a particular room, recharge its batteries, etc.

If a three-tiered architecture is used, the planner level would be in charge of high-level decision making tasks. For example the planner might specify the delivery order for each piece of mail to optimize the overall efficiency (i.e. by considering the relative locations of each delivery). The planner would also choose when to schedule time for recharging.

Given a task from the planner such as "Deliver package to Rm 009", the executive level would then coordinate how to accomplish the task. This might include sub-tasks such as move to the end of the hallway, open the door, enter Rm 009, announce delivery, and then wait and monitor to see if the package is retrieved. If the package is never retrieved within a specified amount of time the executive level could also choose to then carry on with the next set of tasks and send a message to the planner that the task was not completed.

Finally, the behavioral control layer would execute the tasks as specified by the executive level. This might include controlling the robot's wheels to move across the hallway, avoiding obstacles along the way. Or it could involve using a manipulator to open a door. If the current task specified by the executive was to open a door and the door was locked, the behavioral control level should eventually recognize failure and report back to the executive level.

23.2 *Architecture Styles*

In addition to choosing the robot system architecture, another very important task is to choose the architecture's *style*. An architecture's style refers to the computational structure that defines communication between components within the architecture. For example in the three-tiered architecture the style would define the method for communicating among the planning, executive, and behavioral control levels, or even between components of each individual level. The implementation of the connection style is typically referred to as

middleware, and two of the most common architecture styles are referred to as *client-server* and *publish-subscribe*.

23.2.1 Client-Server

Middleware based on the client-server style consists of message requests from clients that the server responds to (i.e. there is a request-response message pairing). This type of connection style can also be thought of as being *on-demand* messaging. One of the disadvantages of such a messaging style is that the client typically waits for the response from the server before continuing, leading to potential deadlocks (e.g. if the server crashes).

23.2.2 Publish-Subscribe

Middleware based on the publish-subscribe style uses asynchronous message broadcasting from publishers, which can then be subscribed to by other components of the system as needed. One disadvantage of this approach is that the interfaces are less well-defined (interactions are only one-way), but the main advantage is in reliability since deadlocks cannot occur (e.g. the system is robust to missing messages or messages arriving out of order). The middleware ROS (Robot Operating System) is a very popular publish-subscribe middleware used within the robotics community today.

24
The Robot Operating System

Introduction to the Robot Operating System (ROS)

This chapter introduces the fundamentals of the Robot Operating System (ROS)[1,2], a popular framework for creating robot software. Unlike what its name appears to suggest, ROS is not an operating system (OS). Rather, ROS is a "middleware" that encompasses tools, libraries and conventions to operate robots in a simplified and consistent manner across a wide variety of robotic platforms. ROS is a critical tool in the field of robotics today, and is widely used in both academia and industry.

This chapter begins by introducing specific challenges in robot programming that motivates the need for a middleware such as ROS. Afterwards, a brief history of ROS will be presented to shed some light on its development and motivations for its important features. Next, the fundamental operating structure of ROS will be discussed in further detail to provide insights into how ROS is operated on real robotic platforms. Lastly, specific features and tools of the ROS environment that greatly simplify robot software development will be presented.

[1] L. Joseph. *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*. Apress, 2018

[2] M. Quigley, B. Gerkey, and W. D. Smart. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. O'Reilly Media, 2015

24.1 Challenges in Robot Programming

Robot programming is a subset of computer programming, but it differs greatly from more classical software programming applications. One of the defining characteristics of robot programming is the need to manage many different individual hardware components that must operate in harmony (e.g. sensors and actuators on board the robot). In other words, robot software needs to not only run the "brain" of the robot to make decisions, but also to handle multiple input and output devices at the same time. Therefore, the following features are needed for robot programming:

- *Multitasking:* Given a number of sensors and actuators on a robot, robot software needs to multitask and work with different input/output devices in different threads at the same time. Each thread needs to be able to communi-

cate with other threads to exchange data.

- *Low level device control:* Robot software needs to be compatible with a wide variety of input and output devices: GPIO (general purpose input/output) pins, USB, SPI among others. C, C++ and Python all work well with low-level devices, so robot software needs to support either of these languages, if not all.
- *High level Object Oriented Programming (OOP):* In OOP, codes are encapsulated, inherited, and reused as multiple instances. Ability to reuse codes and develop programs in independent modules makes it easy to maintain code for complex robots.
- *Availability of 3rd party libraries and community support:* Ample third-party libraries and community support not only expedite software development, but also facilitate efficient software implementation.

24.2 *Brief History of ROS*

Until the advent of ROS, it was impossible for various robotics developers to collaborate or share work among different teams, projects or platforms. In 2007, early versions of ROS started to be conceived with the Stanford AI Robot (STAIR) project, which had the following vision:

- The new robot development environment should be free and open-source for everyone, and need to remain so to encourage collaboration of community members.
- The new platform should make core components of robotics – from its hardware to library packages – readily available for anyone who intends to launch a robotics project.
- The new software development platform should integrate seamlessly with existing frameworks (OpenCV for computer vision, SLAM for localization and mapping, Gazebo for simulation, etc).

Development of ROS started to gain traction when Scott Hassan, a software architect and entrepreneur, and his startup Willow Garage took over the project later that year to develop standardized robotics development platform. While mostly self-funded by Scott Hassan himself, ROS really satiated the dire needs for a standardized robot software development environment at the time. In 2009, ROS 0.4 was released, and a working ROS robot with a mobile manipulation platform called PR2 was developed. Eleven PR2 platforms were awarded to eleven universities across the country for further collaboration on ROS development, and in 2010 ROS 1.0 was released. Many of the original features from ROS 1.0 are still in use. In 2012, the Open Source Robotics Foundation (OSRF) started to supervise the future of ROS by supporting development, distribution,

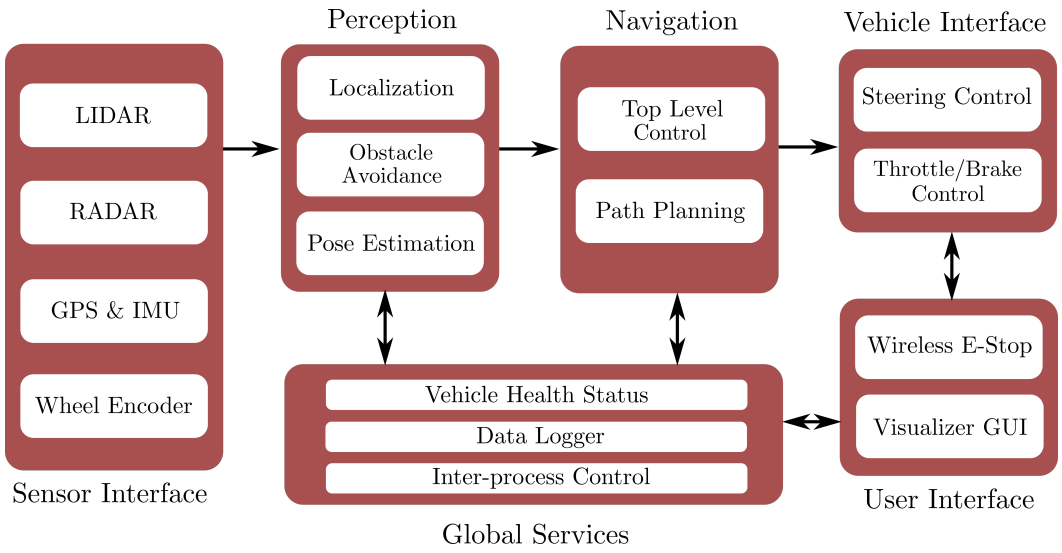


Figure 24.1: Modular software architecture designed to handle complexity of robot programming

and adoption of open software and hardware for use in robotics research, education, and product development. In 2014 the first long-term support (LTS) release, ROS Indigo Igloo, became available. Today, ROS has been around for 12 years, and the platform has become what is closest to the "industry standard" in robotics.

24.2.1 *Characteristics of ROS*

Building off of the initial needs first conceived by the STAIR project and the unique challenges persistent in robot programming, the ROS framework provides the following important capabilities:

- *Modularity:* ROS handles complexity of a robot through modularity: Each robot component that performs separate functions can be developed independently in units called *nodes* (Figure 24.1). Each node can share data with other nodes, and acts as the basic building blocks of ROS. Different functional capabilities on a robot can be developed in units called packages. Each package may contain a number of nodes that are defined from source code, configuration files, and data files. These packages can be distributed and installed on other computers.

- *Message passing:* ROS provides a message passing interface that allows nodes (i.e. programs) to communicate with each other. For example, one node might detect edges in a camera image, then send this information to an object recognition node, which in turn can send information about detected obstacles to a navigation module.

- *Built-in algorithms:* A lot of popular robotics algorithms are already built-in and available as off-the-shelf packages: PID[3], SLAM[4], and path planners such

[3] http://wiki.ros.org/pid
[4] http://wiki.ros.org/gmapping

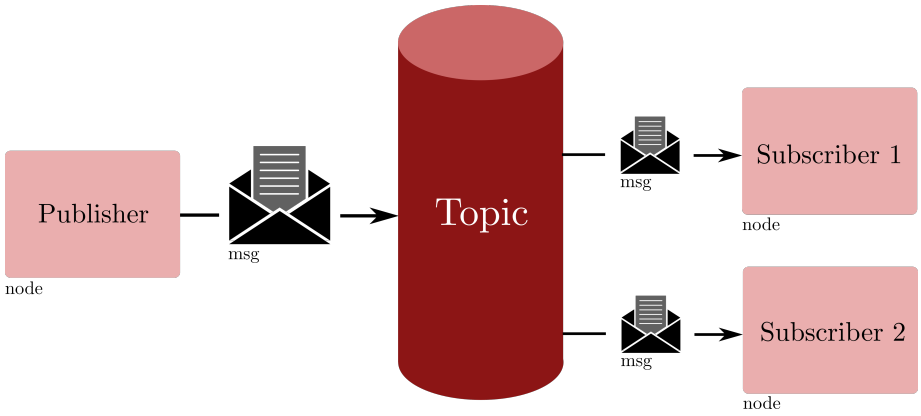


Figure 24.2: The ROS publish/-subscribe (pub/sub) model.

as A* and Dijkstra[5] are just a few examples. These built-in algorithms can significantly reduce time needed to prototype a robot.

- *Third-party libraries and community support:* The ROS framework is developed with pre-existing third-party libraries in mind, and most popular libraries such as OpenCV for computer vision[6] and PCL[7] integrate simply with a couple lines of code. In addition, ROS is supported by active developers all over the world to answer questions (ROS Answers[8] or to discuss various topics and public ROS-related news[9].

[5] http://wiki.ros.org/global_planner

[6] https://opencv.org

[7] http://pointclouds.org

[8] https://answers.ros.org/questions/

[9] https://discourse.ros.org

24.3 Robot Programming with ROS

Before jumping into using the functions and tools that ROS provides it is critical to understand a little more about how ROS operates. In particular, it is important to know that ROS uses a publish/subscribe (pub/sub) structure for communicating between different components or modules. This pub/sub structure (graphically shown in Figure 24.2) allows messages to be passed in between components or modules through a shared virtual "chat room". To support this structure there are four primary components of ROS:

1. Nodes: the universal name for the individual components or modules that need to send or receive information,
2. Messages: the object for holding information that needs communicated between nodes,
3. Topics: the virtual "chat rooms" where messages are shared,
4. Master: the "conductor" that organizes the nodes and topics.

24.3.1 Nodes

Definition 24.3.1 (Node)**.** A node[10] is a process that performs computation.

[10] http://wiki.ros.org/Nodes

Nodes are combined together to communicate with one another using streaming topics, RPC services, and the Parameter Server.

Nodes are the basic building block of ROS that enables object-oriented robot software development. Each robot component is developed as an individual encapsulated unit of *nodes*, which are later reused and inherited, and a typical robot control system will be comprised of many nodes. The use of independent nodes, and their ability to be reused and inherited, greatly simplifies the complexity of the overall software stack.

For example, suppose a robot is equipped with a camera and you want to find an object in the environment and drive to it. Examples of nodes that might be developed for this task are: a `camera` node that takes the image and pre-processes it, an `edge_detection` node that takes the pre-processed image data and runs an edge detection algorithm, a `path_planning` node that plans a path between two points, and so on.

At the individual level, nodes are responsible for **pub**lishing or **sub**scribing to certain pieces of information that are shared among all other nodes. In ROS, the pieces of information are called “messages" and they are shared in virtual chat rooms called “topics".

24.3.2 *Messages*

Definition 24.3.2 (Messages)**.** Nodes communicate with each other by publishing simple data structures to topics. These data structures are called *messages*[11].

[11] `http://wiki.ros.org/Messages`

A message is defined by field types and field names. The field type defines the type of information the message stores and the name is how the nodes access the information. For example, suppose a node wants to publish two integers x and y, a message definition might look like:

```
int32 x
int32 y
```

where `int32` is the field type and `x/y` is the field name. While `int32` is a primitive field type, more complex field types can also be defined for specific applications. For example, suppose a sensor packet node publishes sensor data as an array of a user-defined `SensorData` object. This message, called `SensorPacket`, could have the following fields:

```
time            stamp
SensorData[]    sensors
uint32          length
```

In this case `SensorData` is a user-defined field type and the empty bracket `[]` is appended to indicate that field is an *array* of `SensorType` objects.

More generally, field types can be either the standard primitive types (integer, floating point, boolean, etc.), arrays of primitive types, or other user-defined types. Messages can also include arbitrarily nested structures and arrays as well.

Primitive message types available in ROS are listed below in Table 24.1. The first column contains the message type, the second column contains the serialization type of the data in the message and the third column contains the numeric type of the message in Python.

Primitive Type	Serialization	Python
bool (1)	unsigned 8-bit int	bool
int8	signed 8-bit int	int
uint8	unsigned 8-bit int	int (3)
int16	signed 16-bit int	int
uint16	unsigned 16-bit int	int
int32	signed 32-bit int	int
uint32	unsigned 32-bit int	int
int64	signed 64-bit int	long
uint64	unsigned 64-bit int	long
float32	32-bit IEEE float	float
float64	64-bit IEEE float	float
string	ascii string (4)	str
time	secs/nsecs unsigned 32-bit ints	rospy.Time

Table 24.1: Built-in ROS Messages

24.3.3 *Topics*

Definition 24.3.3 (Topics)**.** Topics[12] are named units over which nodes exchange messages.

[12] http://wiki.ros.org/Topics

A given topic will have a specific message type associated with it, and any node that either publishes or subscribes to the topic must be equipped to handle that type of message. The command `rostopic type <topic>` can be used to see what kind of message is associated with a particular topic. Any number of nodes can publish or subscribe to a given topic.

Fundamentally, topics are for unidirectional, streaming communication. This is perhaps not well suited for all types of communication, such as communication that demands a response (i.e. a service routine).

The `rostopic` command line tool can be used in several ways to monitor active topics/messages. Three of the most common rostopic commands are:

- `rostopic list`: lists all active topics
- `rostopic echo < topic >`: prints messages received on topic
- `rostopic hz < topic >`: measures topic publishing rate

The last command is particularly useful in debugging responsiveness of an application.

24.3.4 Master

Definition 24.3.4 (Master)**.** The master is a process that can run on any piece of hardware to track publishers and subscribers to topics as well as services in the ROS system.

Master is responsible for assigning network addresses and enabling individual ROS nodes to locate one another, even if they are running on different computers. Once these nodes have located each other, the communication will be peer-to-peer, i.e., the master will not send nor receive the messages.

In any given ROS system, there is exactly one master running at any time. A unique feature of the master is that master does not need to exist within the robot's hardware as long as a network connection exists. The master can be facilitated remotely, on a much larger and more powerful computer.

24.4 Writing a Simple Publisher Node and Subscriber Node

24.4.1 Publisher Node

A simple publisher node that publishes String messages ten times per second can be implemented in Python via the following code[13]:

[13] http://wiki.ros.org/rospy_tutorials/Tutorials/WritingPublisherSubscriber

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def talker():
    rospy.init_node('talker', anonymous=True)
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rate = rospy.get_param('~rate',1)
    ros_rate = rospy.Rate(rate)

    rospy.loginfo("Starting ROS node talker...")

    while not rospy.is_shutdown():
        msg= "Greetings humans!"

        pub.publish(msg)
        ros_rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

The first line:

```
#!/usr/bin/env python
```

will be included in every Python ROS Node at the top of the file. This line makes sure your script is executed as a Python script.

Next are the statements for importing specific Python libraries:

```
import rospy
from std_msgs.msg import String
```

Note that the library `rospy` must be imported when writing a ROS Node. The `std_msgs.msg` import enables the use of the `std_msgs/String` message type (a simple string container) for publishing string messages.

Next is the definition of the ROS publisher node:

```
rospy.init_node('talker', anonymous=True)
pub = rospy.Publisher('chatter', String, queue_size=10)
```

which creates a node called "talker" and defines the talker's interface to the rest of ROS. In particular:

- `pub = rospy.Publisher("chatter", String, queue_size=10)` declares that the node is publishing to the "chatter" topic using the String message type. String here is actually the ROS datatype (`std_msgs.msg.String`), and not Python's String datatype. The `queue_size` argument limits the amount of queued messages that are allowed, for situations where a subscriber is not receiving the published messages fast enough.

- `rospy.init_node(NAME, ...)` tells `rospy` the name of the node. Until `rospy` has this information, it cannot start communicating with the ROS Master. In this case, your node will take on the name talker. NOTE: the name must be a base name (i.e. it cannot contain any slashes "/").

- `anonymous=True` is a flag that tells `rospy` to generate a unique name for the node, since ROS requires that each node have a unique name. If a node with the same name comes up, it bumps the previous one so that malfunctioning nodes can easily be kicked off the network. This makes it easy to run multiple talker.py nodes.

- `anonymous = True` is another flag that ensures that the node has a unique name by adding random numbers to the end of NAME.

The next lines of code:

```
rate = rospy.get_param('~rate',1)
ros_rate = rospy.Rate(rate)
```

defines a ROS rate that can be used to time how often the node publishes. In particular, `rospy.get_param(param_name, default_value)` reads a private ROS parameter (indicated by '~') called `rate`. This rate value is then used to create a `Rate` object `ros_rate` in the second line. The `Rate` object's `sleep()` method offers a convenient way for looping at the desired frequency. For example, if `rate` is 10, ROS should go through the loop 10 times per second (as long as the processing time does not exceed 1/10th of a second!).

The line:

```
rospy.loginfo("Starting ROS node talker...")
```

performs three functions: it causes messages to get printed to screen, to be written to the Node's log file, and to be written to `rosout`. `rosout` is a handy tool for debugging that makes it possible to pull up messages using `rqt_console` instead of having to find the console window with your Node's output.

The loop:

```
while not rospy.is_shutdown():
    msg = "Greetings humans!"
    pub.publish(msg)
    ros_rate.sleep()
```

is a fairly standard rospy construct for first checking the `rospy.is_shutdown()` flag and then doing work. The `is_shutdown()` check is used to see if the program should exit (e.g. if there is a Ctrl-C interrupt). In this particular example, the "work" that is then performed inside of the loop is a call to `pub.publish(msg)`, which publishes a string to the "chatter" topic. The loop also calls `ros_rate.sleep()` so that it sleeps just long enough to maintain the desired rate through the loop.

24.4.2 Subscriber Node

A subscriber node called `listener` can now be created to subscribe to the published "chatter" topic:

```
!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(msg):
    rospy.loginfo("Received: %s", msg.data)

def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
```

```
    rospy.spin()

if __name__ == '__main__':
    listener()
```

The code for `listener.py` is similar to `talker.py`, except that a new callback-based mechanism for subscribing to messages is introduced.

First, the lines:

```
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
```

declare that the node subscribes to the "chatter" topic, which is of type `std_msgs.msgs.String`. When new messages are received, the function `callback` is invoked with the message as the first argument.

The line:

```
    rospy.spin()
```

then simply keeps the node from exiting until the node has been shutdown.

24.4.3 *Compiling and Running*

Once both the `talker.py` and `listener.py` nodes are ready, the `catkin` build system can be used to compile the code, and then both nodes can be run. Specifically, this is accomplished by running the following commands:

```
$ cd ~/catkin_ws
$ catkin_make
$ python talker.py
$ python listener.py
```

24.5 *Other Features in ROS Development Environment*

24.5.1 *Launch files*

As a robot project grows in scale, the number of nodes and configuration files grow very quickly. In practice, it could be very cumbersome to manually start up each individual node. A *launch file* provides a convenient way to start up multiple nodes and a master, as well as set up other configurations, all at the same time.

Definition 24.5.1. Launch files are .launch files with a specific XML format that can be placed anywhere within a package directory to initialize multiple nodes, configuration files, and a master.

While `.launch` files can be placed anywhere within a package directory, it is standard practice to create a `launch` folder inside the workspace directory to organize launch files. Launch files must start and end with a pair of launch tags: `<launch> ... </launch>`. To start a node using a launch file the following syntax should be used within the launch file:

```
<node name="..." pkg="..." type="..."/>
```

In this line, **pkg** points to the package associated with the node that is to be launched, **type** refers to the name of the node executable file, and the name of the node can be overwritten with the **name** argument (this will take priority over the name that is given to the node in the code). For example,

```
<node name="bar1" pkg="foo_pkg" type="bar" />
```

launches the `bar` node from the `foo_pkg` package with a new name, `bar1`. Alternatively,

```
<node name="listener1" pkg="rospy_tutorials" type="listener.py"
     args="--test" respawn="true" />
```

launches the `listener1` node using the `listener.py` executable from the `rospy_tutorials` package with the command-line argument `-test`. Additionally, if the node dies it will automatically be respawned.

There are other attributes that can be set when starting a node. While only `args` and `respawn` were introduced in this section, `http://wiki.ros.org/roslaunch/XML/node` is a great resource for additional parameters that can be used in the `<node>` tag.

24.5.2 Catkin Workspace

`catkin`[14] is a build system that compiles ROS packages. While `catkin`'s workflow[15] is very similar to CMake's, `catkin` adds support for automatic 'find package' infrastructure, for building multiple, dependent projects at the same time, and also supports both C and Python.

[14] `catkin` refers to the tail-shaped flower cluster on willow trees – a reference to Willow Garage where, `catkin` was created.

[15] `http://wiki.ros.org/catkin/conceptual_overview`

When developing with ROS, `catkin` should be run whenever a new project is started, or if there are any additions to packages. This is accomplished by creating a directory called `catkin_ws` and then running the compile command `catkin_make` in that directory:

```
mkdir -p ~/catkin_ws/src  # builds the catkin_ws in the home dir
cd ~/catkin_ws            # change current directory to catkin_ws
catkin_make               # run catkin
```

Once the `catkin` workspace is compiled, it will automatically contain the files `CMakeLists.txt` and `package.xml`. There are other sub-folders in `catkin_ws` as well, as shown in Figure 24.3, which can be changed as needed.

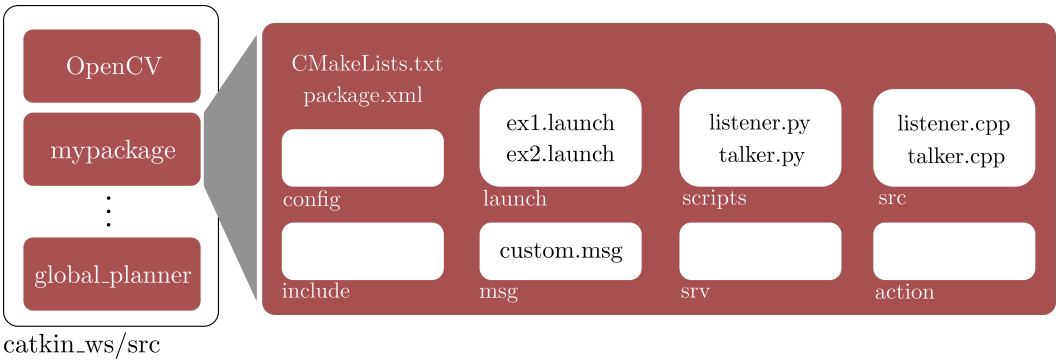


Figure 24.3: Components of an example ROS package named `mypackage` in a `catkin` workspace.

24.5.3 *Debugging*

Robot programming requires a lot of debugging. There are a few ways to debug your robot software, including (but not limited to):

- `rostopic`: a tool that monitors ROS topics in the command line,
- `rospy.loginfo()`: starts a background process that writes ROS messages to a ROS logger, viewable through a program such as `rqt_console`,
- `rosbag`: provides a convenient way to record a number of ROS topics for playback,
- `pdb`: provides a useful tool for debugging python scripts.

24.5.4 *Gazebo*

Gazebo[16] is a popular 3D dynamic simulator with the ability to accurately and efficiently simulate robots in complex environments (see Figure 24.4). While similar to game engines, Gazebo offers a higher fidelity physics simulation, a suite of sensors models, and interfaces for both users and programs. Gazebo can be used in any stage of robot development, from testing algorithms to running regression testing in realistic scenarios. Integration of Gazebo with ROS is possible via `gazebo_ros_pkgs` package.

[16] `http://gazebosim.org`

Figure 24.4: Screenshot of a scene in Gazebo.

Part VI

Advanced Topics in Robotics

25
Formal Methods

The safety and explainability of robotic systems has become increasingly important as applications for robotic systems transition to more unstructured and interactive environments. While one component to developing safe robots is to design robust and high-performing autonomy algorithms, another critical component is the analysis of the system's design. System analysis could come in several forms, including unit, component, or system-level testing, but one challenging aspect of testing is the determination of appropriate success criteria. It is also highly desirable to develop systems that are *provably correct* or *correct-by-construction* with respect to the stated success criteria.

This chapter introduces a set of rigorous mathematical tools and concepts for specifying desired behavior (i.e. requirements or specifications), proving that the system achieves the desired behavior, and synthesizing robot systems to be correct-by-construction. These mathematical tools are known as *formal methods*[1].

[1] E. M. Clarke et al. *Model Checking*. 2nd ed. MIT Press, 2018

Formal Methods

Formal methods provide a mathematical framework for reasoning about a system's specifications as well as analyzing whether the system's behavior guarantees their satisfaction. Approaches for synthesizing provably correct behavior can also be built on top of these tools and are commonly incorporated within the umbrella of formal methods. Historically these techniques have been developed within the computer science community, and have been used to study problems related to logic, automatically proving properties of algorithms, checking the correctness of properties of circuits, and more. However in this chapter formal methods will be explored within the context of autonomous systems.

Definition 25.0.1 (Formal Methods)**.** Formal methods are mathematically based techniques for the specification, development, and verification of software and hardware systems.

It is important to note that formal methods are not just particular solutions or algorithms but rather are a class of tools and formalisms. Accordingly, this chapter will not focus on a particular algorithm (solution) for applying formal

methods to problems in robotics, but will rather introduce several broadly used tools and techniques. Specifically, Section 25.1 will introduce *linear temporal logic*, which is a specialized language for writing specifications. Then the concept of the robotic system as a reactive model that can be *verified* to satisfy the stated specifications will be introduced in Section 25.2. Finally, the ability to *synthesize* robot systems to provably be able to satisfy a specification will be explored in Section 25.3.

25.1 Linear Temporal Logic

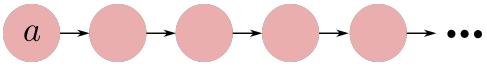
Linear temporal logic (LTL) is a mathematical *language* for formally expressing specifications or requirements on the system's behavior[2]. LTL is useful in robotics applications because it extends propositional logic[3] to handle *temporal* components (assuming discrete time steps), which are common in sequential decision making problems and other robotics tasks. For example propositional logic can be used to write a specification that "proposition a and proposition b must both be true", while LTL extends the possible specifications to include temporal constraints such as "proposition a must be true *until* proposition b is true".

[2] Similar to how ordinary differential equations provide a mathematical "language" that is useful for modeling the kinematics and dynamics of physical systems.

[3] A formalism for expressing logical operations including conjunction (and), disjunction (or), and negation (not).

The language of LTL can be expressed in terms of several atomic operators, the first few of which are inherited from propositional logic:

1. *true* or *false* (Boolean values)[4] represent Boolean constants.
2. $a, b, \ldots$ (propositional symbols) denote single variables that can either be true or false at the current time step.



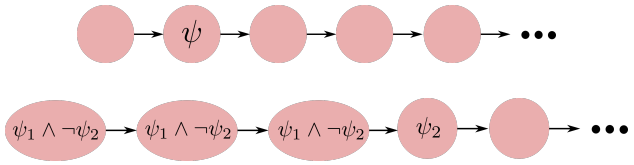
3. $\neg\psi$ (negation operator[5]) denotes the negation of ψ.
4. $\psi_1 \wedge \psi_2$ (conjunction "and" operator) which can be read as "ψ_1 and ψ_2".
5. $\psi_1 \vee \psi_2$ (disjunction "or" operator) which can be read as "ψ_1 or ψ_2". This operator can be expressed in terms of "and" and "not" as $\psi_1 \vee \psi_2 = \neg(\neg\psi_1 \wedge \neg\psi_2)$.
6. $\psi_1 \rightarrow \psi_2$ (implication operator) denotes that ψ_1 implies ψ_2. This operator can be expressed in terms of "not' and "or" as $\psi_1 \rightarrow \psi_2 = \neg\psi_1 \vee \psi_2$.

[4] Technically false can also be written as $\neg$true, where $\neg$ is the "not" operator.

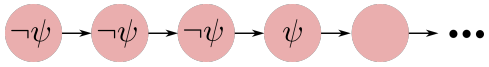
[5] Technically *false* can be written as $\neg$*true*.

Additional operators that are fundamental to LTL provide the capability for expressing temporal constraints:

7. $X\psi$ ("next" operator) denotes that ψ happens next (at the next time step).
8. $\psi_1 U \psi_2$ ("until" operator) denotes that ψ_1 should happen until ψ_2 happens.



9. $F\psi$ ("eventually" operator) denotes that ψ happens at some point in the *future*. This operator can be expressed in terms of the eventually operation as $F\psi = \text{true}\, U\, \psi$.



10. $G\psi$ ("always" operator) denotes that ψ happens globally (at all times). This operator can be expressed in terms of the negation and future operations as $G\psi = \neg F \neg \psi$.

From these atomic operators it is possible to define many new specifications through *composition*, and they can become arbitrarily complex as needed. A couple of common and useful compositions include:

11. $GF\psi$ ("infinitely often" composition) denotes that ψ will eventually happen an infinite number of times (i.e. globally, ψ will happen eventually). In other words there is always a ψ in the future.

12. $FG\psi$ ("stability"[6] composition) denotes that at some point in time ψ will be true for all time thereafter (i.e. eventually ψ will happen globally).

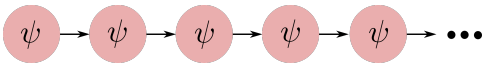
13. $G(\psi_1 \rightarrow F\psi_2)$ ("response" composition) denotes that for all time, whenever ψ_1 occurs then ψ_2 will occur sometime in the future. There are other useful variations on this composition, such as by replacing the F operator with X operator.

[6] This notion of stability is similar but not directly the same as the notions of stability from control theory.

Linear temporal logic provides a very powerful tool[7] for abstractly talking about time, and in general the specifications written using LTL can in a way be more "vague". For example the eventually operator F does not explicitly state *when* something must occur, just that at *some point* it will. It is also important to keep in mind that LTL is not an algorithm or technique for solving problems, but rather a language for *formulating* problems (i.e. for expressing properties of interest such as system specifications).

[7] There are also alternatives to LTL that provide even more powerful features, for example by not requiring a "linear" temporal structure but rather allowing for temporal "branching".

Example 25.1.1 (Coffee Machine Specification)**.** Consider a simple robot that makes coffee. This robot has a button that a user can press, and has two functions: grinding coffee beans and brewing coffee. The desired behavior of this robot could be expressed by the designer as: *if the start button is pressed, the robot will immediately start grinding beans for the next two cycles, and then brew the coffee for the next two cycles after that*. This specification, denoted as ϕ, could be



expressed via linear temporal logic as:

$$\phi : G\big(\text{button} \rightarrow \text{grind} \wedge X\text{grind} \wedge XX(\neg\text{grind} \wedge \text{brew}) \wedge XXX(\neg\text{grind} \wedge \text{brew})\big).$$

Note that the entire statement needs to be wrapped in the "always" operator to ensure this behavior can occur at any arbitrary time that a user presses the button.

25.2 *Verification*

System verification is the process of *proving* that the system's behavior will satisfy the stated requirements and specifications (often expressed using linear temporal logic). In the terminology of formal methods, this system is typically referred to as the *model*[8] and the output of the verification[9] procedure is a simple yes/no stating whether the model satisfies the specification. In this chapter the model will be denoted by P and the specification by ϕ, and the notation for stating that model P satisfies specification ϕ is $P \models \phi$ (which can be read as "P models ϕ").

[8] The model could refer to a piece of software, a hardware component, an individual algorithm, or even an entire robot.

[9] Also commonly referred to as *model checking*.

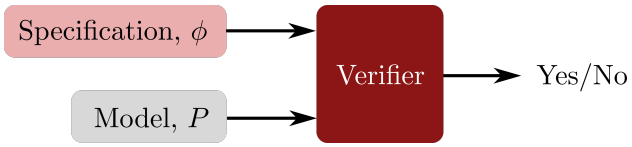


Figure 25.1: Given a system (model) and a specification, the process of verification proves whether the system satisfies the specification.

In this chapter it is assumed that the model P is a *reactive system*, meaning that its behavior is defined based on inputs i which effect the system's outputs o[10]. In contrast to robotic control problems where the "inputs" are generally the control inputs determined by the control algorithm, the inputs i within this context refer to signals coming from the environment. The outputs o of the model can then be thought of as the result of the system's decision making or underlying algorithm/process. As was mentioned previously, the model P can take on many forms depending on whether the system is a hardware component, software component, algorithm, finite state machine, or even simply a mathematical function such as a machine learning model or control law.

[10] The inputs and outputs occur at each time step, and the behavior is assumed to be non-terminating.

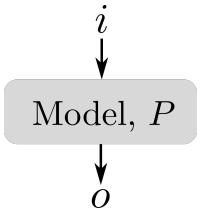


Figure 25.2: The model P is a system with inputs i and outputs o. These inputs and outputs are used to express the model's specification ϕ.

For a given model P the specification ϕ is assumed to be written in terms of the input and output sequences (i.e. the behavior is defined by the inputs and outputs of the system). Specifically, these sequences will be denoted as $\hat{\imath} = (i_0, i_1, \dots)$ and $\hat{o} = (o_0, o_1, \dots)$. With these definitions, the expression that P satisfies ϕ can equivalently be written as $P \models \phi$ or $\hat{\imath} \cup \hat{o} \models \phi$.

To summarize, the problem of model verification is to simply determine whether the input-output behavior of the model P guarantees that ϕ is satisfied

for all possible input sequences. There are several existing techniques that can perform system verification, and they may be tailored to the specific model form[11,12].

[11] M. Kwiatkowska, G. Normal, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. 2011, pp. 585–591

[12] G. Katz et al. "The Marabou Framework for Verification and Analysis of Deep Neural Networks". In: *Computer Aided Verification*. 2019, pp. 443–452

25.3 Reactive Synthesis

Given a reactive model P and a LTL specification ϕ, the problem of verification is to determine whether the behavior of P satisfies ϕ for all possible input sequences $\hat{\imath}$. But several important questions remain: how should the system be designed, and what should be changed if the verification step shows that $P \not\models \phi$? *Reactive synthesis* addresses these problems by *synthesizing* the system model P to be correct-by-construction. In other words, in reactive synthesis the specification ϕ is first defined and then a model is constructed from scratch to satisfy the specification.

Figure 25.3: Given a specification ϕ, the process of reactive synthesis generates a model P that realizes the specification under all possible environmental inputs.

25.3.1 Specification Satisfiability and Realizability

The first step in reactive synthesis is to determine whether a model[13] even exists which can satisfy the LTL specification ϕ for all possible input sequences. If no such model exists, then the system designer should reevaluate the specification itself.

[13] Technically speaking, a *finite state model*.

In the nomenclature of formal methods, the specification ϕ is said to be *realizable* if it can be satisfied for all possible input sequences, and it is *satisfiable* if there exists at least one input sequence leading to satisfaction. These properties can be more rigorously defined in terms of the input and output sequences that describe the system's behavior:

Definition 25.3.1 (Satisfiability)**.** A specification ϕ is *satisfiable* if for some input sequence there exists an output sequence that satisfies the specification. Mathematically:

$$\exists\, \hat{\imath} = (i_0, i_1, \dots), \quad \exists\, \hat{o} = (o_0, o_1, \dots), \quad s.t.\ \hat{\imath} \cup \hat{o} \models \phi.$$

Definition 25.3.2 (Realizability)**.** A specification ϕ is *realizable* if for all possible input sequences there exists an output sequence that satisfies the specification. Mathematically:

$$\forall\, \hat{\imath} = (i_0, i_1, \dots), \quad \exists\, \hat{o} = (o_0, o_1, \dots), \quad s.t.\ \hat{\imath} \cup \hat{o} \models \phi.$$

Obviously the property of satisfiability is weaker than realizability, and realizability is much more important in practice. For example in order to guarantee

safety in a rigorous way it is not sufficient to show that the system will be safe under a single scenario, but rather it should be shown for all scenarios. However, designing specifications that are realizable can be quite challenging, even in seemingly simple problems. As an example consider again the coffee machine robot from Example 25.1.1.

Example 25.3.1 (Coffee Machine Realizability)**.** The coffee machine robot from Example 25.1.1 had inputs $I = \{\text{button}\}$ and outputs $O = \{\text{grind}, \text{brew}\}$. For simplicity let $i_{\text{button}} = 1$ and $i_{\text{button}} = 0$ denote the button is pressed and not pressed, respectively. Additionally let $o_{\text{grind}} = 1$ and $o_{\text{brew}} = 1$ denote that the actions are occurring and let them be zero otherwise.

Recall that the linear temporal logic specification for the robot's behavior was defined as:

$$\phi : G\big(\text{button} \rightarrow \text{grind} \wedge X\text{grind} \wedge XX(\neg\text{grind} \wedge \text{brew}) \wedge XXX(\neg\text{grind} \wedge \text{brew})\big).$$

This specification can now be analyzed to determine whether it is realizable. First, notice that one possible sequence of inputs and outputs that satisfies this specification is:

$$\begin{aligned}(i_{\text{button}}, o_{\text{grind}}, o_{\text{brew}})_k = (0,\, 0,\, 0)_0,\\ (1,\, 1,\, 0)_1,\\ (0,\, 1,\, 0)_2,\\ (0,\, 0,\, 1)_3,\\ (0,\, 0,\, 1)_4,\\ (0,\, 0,\, 0)_5,\\ \vdots\end{aligned}$$

Because there exists a sequence that satisfies the specification ϕ it is by definition *satisfiable*. However, consider a second input sequence $\hat{i} = (0, 1, 1, 0, 0, \dots)$ where the coffee machine's button is pressed twice in a row:

$$\begin{aligned}(i_{\text{button}}, o_{\text{grind}}, o_{\text{brew}})_k = (0,\, 0,\, 0)_0,\\ (1,\, 1,\, 0)_1,\\ (1,\, 1,\, 0)_2,\\ (0,\, ?,\, 1)_3,\end{aligned}$$

At time step $k = 3$ there is no combination of outputs that will satisfy the specification, since the first button press requires that $o_{\text{grind},3} = 0$ but the second button press requires that $o_{\text{grind},3} = 1$! Therefore by definition this specification is not *realizable*[14].

[14] Does this mean it is impossible to automate a coffee maker? No! It just demonstrates that writing *specifications* can be challenging.

25.3.2 *Synthesis for Realizable LTL Specifications*

If a LTL specification ϕ is realizable, then the synthesis problem seeks to find a finite state system that satisfies ϕ under all possible inputs. This can be accomplished by formulating the problem as a two-player game where the objective

is for the system to generate "winning" outputs while the environment generates adversarial inputs. The two-player game formulation can be expressed mathematically by defining the following components:

1. With the inputs I and outputs O, at each time step the environment gets to choose from a set of $2^{|I|}$ actions and the system gets to choose from a set of $2^{|O|}$ actions.

2. The strategy of the system is expressed as a function $f : (2^{|I|})^* \to 2^{|O|}$, where f is a function from a finite sequence of environmental inputs to a specific output.

3. The linear temporal logic specification ϕ is defined by the input and output sequences.

4. The game is played for an infinitely long horizon, generating sequences $\hat{i} = (i_0, i_1, \dots)$ and $\hat{o} = (o_0, o_1, \dots)$.

5. The game is won if $\hat{i} \cup \hat{o} \models \phi$.

The process of converting a problem specification into this two-player game follows two main steps. First, the specification is converted into a *non-deterministic Büchi automaton* and then the automaton is determinized to yield the game. Once the game is appropriately formulated, it can be solved using existing algorithms to generate the policy f that defines the system's behavior. Unfortunately, converting the specification into the automaton is computationally very challenging! In fact the computational complexity is *doubly-exponential* in the size of the specification[15], which significantly limits the complexity of the problems that can be considered[16].

While the precise details for converting a specification into a two-player game and solving the game are beyond the scope of this chapter, the process can be explored visually through the following example.

[15] A doubly exponential function has the form $f(x) = a^{b^x}$.

[16] This is one of the most significant limitations of formal methods in practical robotic settings, and approaches to overcome this complexity are still a topic of research.

Example 25.3.2 (Simple Reactive Synthesis Problem)**.** Consider the LTL specification $\phi : G(r \to Xg)$ which states that whenever a request r is received the system should provide a grant g in the next time step. In this problem $I = \{r\}$ where r denotes a request or no request and $O = \{g\}$ where g specifies if a grant was made or not. The first step in transforming this specification into the two-player synthesis game is to generate the following Büchi automaton representation, as shown in Figure 25.4. In Figure 25.4 the variables q_0 and q_1 represent states of the automaton, and the transitions between the states are dependent on the environmental inputs and the system's behavior.

The two-player game is generated from this Büchi automaton[17] by introducing intermediate states as well as the unsafe "contradiction" state (denoted in Figure 25.5 as $\perp$). This game is represented graphically in Figure 25.5 where the $*$ denotes that any action could be taken by the model and the small grey circles represent the intermediate states. The system can "win" this two-player game

[17] This automaton is already deterministic, so no determinizing step is needed.

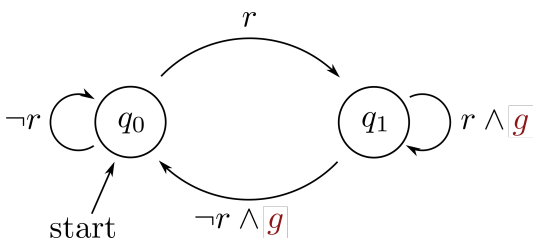


Figure 25.4: The Büchi automaton representation of the LTL specification $\phi : G(r \rightarrow Xg)$.

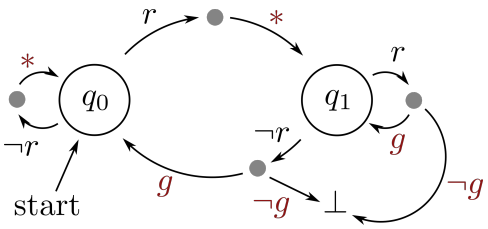


Figure 25.5: The two-player game representation derived from the Büchi automaton in Figure 25.4.

by ensuring that the contradiction state is never reached, which then defines the system's behavior! By analyzing Figure 25.5, it turns out that one "winning" strategy strategy (behavior) is for the system to *always* provide a grant! This strategy is shown graphically in Figure 25.6.

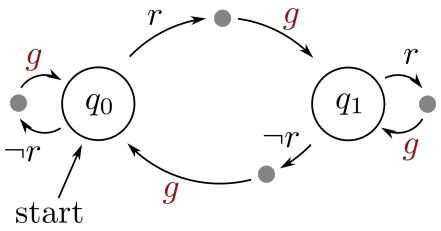


Figure 25.6: A strategy for the system in Example 25.3.2 to ensure the specification is met is to *always* provide grants, which is guaranteed to avoid the contradiction state in the two-player game shown in Figure 25.5.

26
Robotic Manipulation

Robotic manipulation, where a robot *physically* interacts and changes the environment, is one of the most challenging tasks in robot autonomy from the perspectives of perception, planning, and control. Consider a simple pick-and-place problem: the robot needs to identify the object, find a good place to grasp, stably pick up the object, and move it to a new location, all while ensuring no part of the robot collides with the environment. In practice even this simple task can become much harder, for example if other objects are in the way and must be moved first, if the object does not have particularly good grasping features, if the weight, size, and surface texture of the object is unknown, or if the lighting is poor[1]. Manipulation tasks are also commonly composed of sequences of interactions, such as making a sandwich or opening a locked door. This chapter focuses on *grasping*[2], which is a fundamental component to all manipulation tasks.

[1] Generally speaking the infinite variability of the real world makes *robust* manipulation extremely difficult.

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

Grasping

Grasping is a fundamental component of robotic manipulation that focuses on obtaining complete control of an object's motion (in contrast to other interactions such as pushing).

Definition 26.0.1 (Grasp)**.** A *grasp* is an act of restraining an object's motion through application of forces and torques at a set of contact points.

Grasping is challenging for several reasons:

1. The configuration of the gripper may be high-dimensional. For example the Allegro Hand (Figure 26.1) has 4 fingers with 3 joints each for a total of 12 dimensions. Plus there are an additional 6 degrees of freedom in the wrist posture (position and orientation), and all of these degrees of freedom vary continuously.

2. Choosing contact points can be difficult. An ideal choice of contact points would lead to a robust grasp, but the space of feasible contacts is restricted

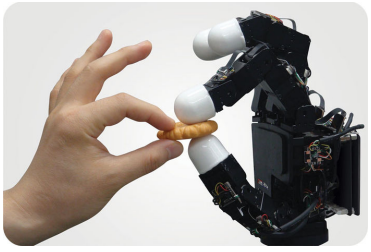


Figure 26.1: The Allegro Hand. Image retrieved from `wiki.wonikrobotics.com`.

by the gripper's geometry. A rigid body object also has 6 degrees freedom, which affects where the contact points are located in the robot's workspace.

3. While the robot is attempting the grasp it must be sure that its entire body does not come into collision with the environment.

4. Once a grasp has been performed it is important to evaluate how robust the grasp is. While the grasp quality would ideally be optimized during the planning step, it may be important to also check retroactively in case uncertainty led to a different grasp than planned.

To address each of these challenges, the grasp can be subdivided into parts: planning, acquisition/maintenance, and evaluation. This chapter will focus on the fundamentals of how a grasp can be *modeled* and *evaluated* from a mathematical perspective, as well as how grasps can be *planned*[3] using grasp force optimization. Learning-based approaches to grasping and manipulations will also be discussed at a high level in Section 26.4 and 26.5.

[3] Part of grasp planning also includes the motion planning of the entire robot, but the focus of this chapter is on the grasp itself.

26.1 Grasp Modeling

A grasp plan may be parameterized in several ways, including by the approach vector or wrist orientation of the gripper, by the initial finger configuration, or directly by points of contact with the object. However, regardless of the planning parameterization the resulting contacts between the gripper and the object will define the quality of the grasp. Therefore it is useful and convenient for grasp modeling to consider the contact points as the interaction interface between the gripper and object.

26.1.1 Contact Types

There are generally three types of contact that can occur in grasping scenarios:

1. *Point*: a point contact occurs when a single point comes in contact with either another point, a line, or a plane. A point contact is only stable if it is a point-on-plane contact[4], point-on-point or point-on-line contacts are unstable.

2. *Line*: line contacts occur when a line comes in contact with another line or a plane. Line-on-plane and line-on-nonparallel line contacts are stable, but line-on-parallel line contacts are unstable. Line contacts can also be represented as two point contacts.

3. *Plane*: plane-on-plane contacts are always stable. Plane contacts can also be represented as point contacts by converting a distribution of normal forces across a region into a weighted sum of point forces at the vertices of the region's convex hull.

[4] Point-on-plane contacts are by far the most commonly modeled contact types and will almost always be used in grasp analysis.

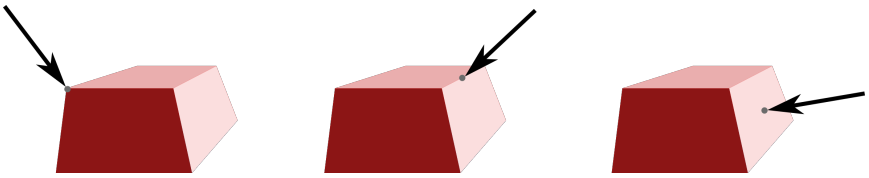


Figure 26.2: Grasping contacts are generally either point-on-point (left), point-on-line (middle), point-on-plane (right).

26.1.2 *Point-on-Plane Contact Models*

Point-on-plane contact models are by far the most commonly used for grasping since the possible contact points for most objects are almost always surface points (and not sharp edges or points). The purpose of the contact model is to specify the admissible forces and torques that can be transmitted through a particular contact. Considering a local reference frame defined at the contact point with the z direction pointing along the object's surface normal (with the positive direction defined as into the object), the force $\boldsymbol{f}$ can be written as:

$$\boldsymbol{f} = \boldsymbol{f}_{\text{normal}} + \boldsymbol{f}_{\text{tangent}},$$

where $\boldsymbol{f}_{\text{normal}} = [0,\ 0,\ f_z]^\top$ is the vector component along the normal direction (with magnitude f_z) and $\boldsymbol{f}_{\text{tangent}} = [f_x,\ f_y,\ 0]^\top$ is the vector component tangent to the surface. For all types of contact only an inward force can be applied, therefore $f_z \geq 0$. Three types of contact models are commonly used, and each defines a set $\mathcal{F}$ of admissible forces that can be applied through the contact:

1. Frictionless Point Contact: forces can only be applied along the surface normal, no torques or forces tangential with the surface are possible ($\boldsymbol{f}_{\text{tangent}} = 0$):

   $$\mathcal{F} = \{\boldsymbol{f}_{\text{normal}} \mid f_z \geq 0\}.$$

   These types of contact models are more common in form closure grasps.

2. Point Contact with Friction[5]: it is possible to apply forces in directions other than just the surface normal. The admissible forces (i.e. forces that don't lead to slipping) are typically defined by a *friction cone*:

   $$\mathcal{F} = \{\boldsymbol{f} \mid \|\boldsymbol{f}_{\text{tangent}}\| \leq \mu_s \|\boldsymbol{f}_{\text{normal}}\|, \quad f_z \geq 0\}.$$

   where μ_s is the static friction coefficient associated with the surface (see Figure 26.3).

   A pyramidal inner-approximation of the friction cone is often more useful from a computational standpoint, since its definition only requires a *finite* set of vectors (see Figure 26.4). The point contact with friction model is more common in force closure grasps.

3. Soft-finger Contact Model: allows for a torque τ_{normal} around the surface normal axis and also includes a friction cone for the forces as in the point

[5] Also referred to as the *hard finger* model.

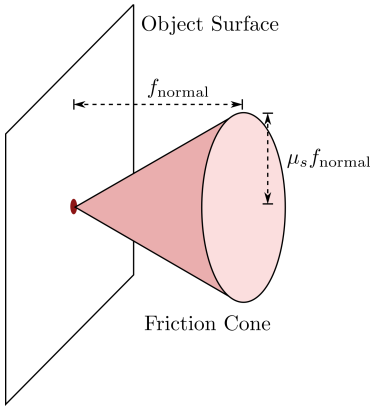


Figure 26.3: Friction cone defined by a static coefficient of friction μ_s.

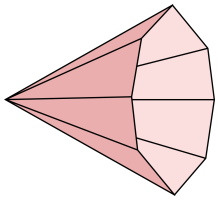


Figure 26.4: Linearized friction cone to *inner* approximate the true cone.

contact with friction model. The admissible torques are also constrained by friction:

$$\mathcal{F} = \{(\boldsymbol{f}, \tau_{\text{normal}}) \mid \|\boldsymbol{f}_{\text{tangent}}\| \leq \mu_s \|\boldsymbol{f}_{\text{normal}}\|, \quad f_z \geq 0, \quad |\tau_{\text{normal}}| \leq \gamma f_z\}.$$

where $\gamma > 0$ is the torsional friction coefficient.

26.1.3 Wrenches and Grasp Wrench Space

Under the assumption of a specific contact model, a grasp (defined by a set of contact points) can be quantified and evaluated by determining the *grasp wrench space*[6], which defines how the grasp can influence the object through an applied wrench.

[6] The grasp wrench space is a *subset* of the *wrench space* $\mathbb{R}^n$, where $n = 6$ in 3D settings and $n = 3$ in 2D settings.

Definition 26.1.1 (Wrench)**.** A *wrench* is a vector valued quantity that describes the forces and torques being applied to an object. For a force $\boldsymbol{f} \in \mathbb{R}^3$ and torque $\boldsymbol{\tau} \in \mathbb{R}^3$ applied at the object's center of mass, the wrench is the stacked vector:

$$\boldsymbol{w} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{\tau} \end{bmatrix} \in \mathbb{R}^6,$$

and is typically written with respect to a frame fixed in the body.

Each contact point i in a grasp applies a wrench to the object. Additionally the torque $\boldsymbol{\tau}_i$ can be computed by $\boldsymbol{\tau}_i = \boldsymbol{d}_i \times \boldsymbol{f}_i$ where $\boldsymbol{d}_i$ is the vector defining the position of the i-th contact point with respect to the object's center of mass. The wrench can then be written as:

$$\boldsymbol{w}_i = \begin{bmatrix} \boldsymbol{f}_i \\ \lambda(\boldsymbol{d}_i \times \boldsymbol{f}_i) \end{bmatrix}, \tag{26.1}$$

where the constant $\lambda \in \mathbb{R}$ is arbitrary but can be used to scale the torque magnitude if desired[7].

[7] If the forces $\boldsymbol{f}_{i,j}$ are dimensional a value of $\lambda = 1$ is common. When the forces are unit-dimension (i.e. scaled by their maximum magnitude), λ could be chosen to non-dimensionalize the entire wrench $\boldsymbol{w}_i$ by non-dimensionalizing the distance vector $\boldsymbol{d}_i$ (i.e. scale by an object size metric).

Using this definition of a wrench[8], a grasp can be defined as the set of all possible wrenches that can be achieved by the grasp's contact points. Mathematically, an admissible force $\boldsymbol{f}_i$ applied at the i-th contact point can be linearly mapped into the corresponding wrench on the object as $G_i \boldsymbol{f}_i$, where G_i is a wrench basis matrix that also includes a transformation from the local contact reference frame to an object-defined global reference frame. Therefore the total wrench on the object from all contacts is:

[8] For a soft-finger contact model the additional torque term must also be included.

$$\boldsymbol{w} = \sum_{i=1}^{k} G_i \boldsymbol{f}_i = G \begin{bmatrix} \boldsymbol{f}_1 \\ \vdots \\ \boldsymbol{f}_k \end{bmatrix}, \quad G = \begin{bmatrix} G_1 & \dots & G_k \end{bmatrix}, \tag{26.2}$$

where the combined matrix G is referred to as the *grasp map* (which varies depending on the type of contact model used).

The *grasp wrench space* can then be defined as:

Definition 26.1.2 (Grasp Wrench Space)**.** The grasp wrench space $\mathcal{W}$ for a grasp with k contact points is the set of all possible wrenches $\boldsymbol{w}$ that can be applied to the object through admissible forces:

$$\mathcal{W} := \{\boldsymbol{w} \mid \boldsymbol{w} = \sum_{i=1}^{k} G_i \boldsymbol{f}_i, \quad \boldsymbol{f}_i \in \mathcal{F}_i, \quad i = 1, \ldots, k\}. \tag{26.3}$$

In other words, the grasp wrench space is defined by the output of (26.2) over all possible applied force combinations $\{\boldsymbol{f}_i\}_{i=1}^{k}$. If the grasp wrench space is large the grasp can compensate for a bigger set of external wrenches that might be applied to the object, leading to a more robust grasp.

Example 26.1.1 (Computing a Grasp Wrench Space from Friction Cones)**.** Consider a grasping problem with k contact points with friction, and let contact point i be associated with a linearized friction cone $\mathcal{F}_i$ whose edges are defined by the set of m forces:

$$\{\boldsymbol{f}_{i,1}, \boldsymbol{f}_{i,2}, \ldots \boldsymbol{f}_{i,m}\},$$

such that any force $\boldsymbol{f}_i \in \mathcal{F}_i$ can be written as a positive combination of these vectors:

$$\boldsymbol{f}_i = \sum_{j=1}^{m} \alpha_{i,j} \boldsymbol{f}_{i,j}, \quad \alpha_{i,j} \geq 0.$$

The condition $\sum_{j=1}^{m} \alpha_{i,j} \leq 1$ will also be imposed to constrain the overall magnitude[9]. Geometrically, this means that the friction cone $\mathcal{F}_i$ is the convex hull of the points $\boldsymbol{f}_{i,j}$ and the origin of the local contact reference frame (see Figure 26.5).

[9] In practice the physical hardware has limitations on the magnitude of the normal forces that can be applied.

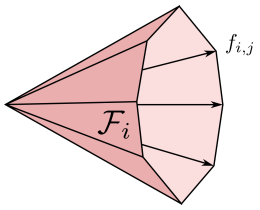


Figure 26.5: Any force $\boldsymbol{f}_i \in \mathcal{F}_i$ can be written as a convex combination of the forces along the edge vectors $\boldsymbol{f}_{i,j}$.

This friction cone can then be mapped into the wrench space using (26.1). Assuming the forces $\boldsymbol{f}_{i,j}$ and position vector $\boldsymbol{d}_i$ are already expressed in a reference frame fixed in the object that is common to all i contact points, the grasp wrench space $\mathcal{W}$ can be written as:

$$\mathcal{W} = \{\boldsymbol{w} \mid \boldsymbol{w} = \sum_{i=1}^{k} \boldsymbol{w}_i, \quad \boldsymbol{w}_i = \sum_{j=1}^{m} \alpha_{i,j} \boldsymbol{w}_{i,j}, \quad \boldsymbol{w}_{i,j} = \begin{bmatrix} \boldsymbol{f}_{i,j} \\ \lambda(\boldsymbol{d}_i \times \boldsymbol{f}_{i,j}) \end{bmatrix}, \\ \sum_{j=1}^{m} \alpha_{i,j} \leq 1, \quad \alpha_{i,j} \geq 0\}.$$

In other words, the grasp wrench space is defined by taking the Minkowski sum over the sets of wrenches that can be generated from each individual contact!

For example, consider the 2D problem shown in Figure 26.6 where there are $k = 2$ contact points with friction. The friction cones are defined by the convex hull of the vectors $\{\boldsymbol{f}_{1,1}, \boldsymbol{f}_{1,2}\}$ and $\{\boldsymbol{f}_{2,1}, \boldsymbol{f}_{2,2}\}$ (and their origins) and the distance vectors from the center of mass to the contact points are $\boldsymbol{d}_1$ and $\boldsymbol{d}_2$. The force vectors $\boldsymbol{f}_{i,j}$ are then mapped into the wrenches $\boldsymbol{w}_{i,j}$ (shown on a 2D plot of vertical force f_y and torque τ in Figure 26.6, ignoring the horizontal force components f_x). The grasp wrench space $\mathcal{W}$ is then shown in the grey region of the wrench space, where the solid grey line is the boundary of $\mathcal{W}$.

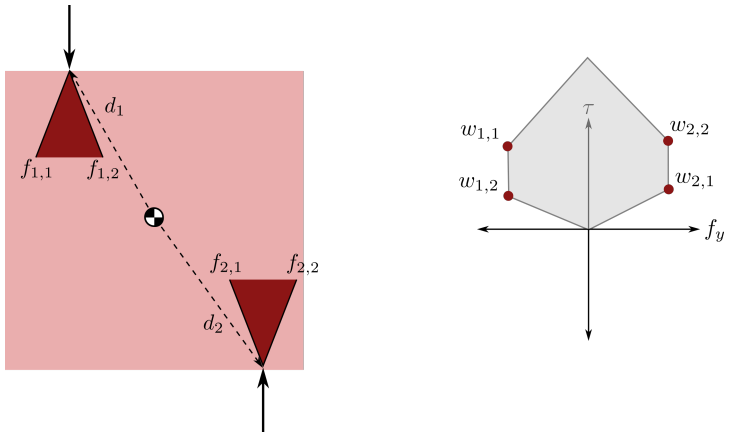


Figure 26.6: An example 2D grasp consisting of two point contacts with friction. The friction cones shown in the figure on the left yield the grasp wrench space in the figure on the right (showing only the vertical force and torque dimensions). Note that the grasp wrench space is bounded because it is assumed the magnitude of the contact forces are bounded. The solid grey line represents the boundary of $\mathcal{W}$.

26.2 Grasp Evaluation

Now that the basics of grasp modeling have been introduced[10] it is possible to explore techniques for evaluating whether a grasp is "good". In particular, an ideal grasp is one that has *closure*.

[10] contact types, contact models, grasp wrench spaces

Definition 26.2.1 (Grasp Closure)**.** Grasp closure occurs when the grasp can be maintained for every possible disturbance load.

For example having grasp closure on a book would enable the gripper to maintain its grasp even if the book was hit by another object or if another book was suddenly stacked on top of it. In practice it may not be reasonable to assume that every *magnitude* disturbance load could be accounted for, but the concept of closure is useful nonetheless.

[11] Also called power grasps or enveloping grasps. A grasp must have at least seven contacts to provide form closure for a 3D object.

It can also be helpful to distinguish between two types of grasp closure. A *form closure*[11] grasp typically has the gripper joint angles locked and there is no "wiggle" room for the object (i.e. the object is kinematically constrained). Alternatively, a *force closure*[12] grasp uses forces applied at contact points to be able to *resist* any external wrench. Force closure grasps typically rely on *friction* and generally require fewer contact points than are required for form closure, but may not be able to actually cancel all disturbance wrenches if the friction forces are too weak. This chapter will primarily focus on evaluating force closure grasps since these are most common in robotics.

[12] Also called a precision grasp. Under a point contact with friction model, a grasp must have at least three contacts to provide force closure for a 3D object.

26.2.1 Force Closure Grasps

The concept of force closure can be related to the grasp modeling concepts from Section 26.1:

Definition 26.2.2 (Force Closure Grasp)**.** A grasp is a *force closure grasp* if for any

Figure 26.7: Examples of grasps with form closure (left) and force closure under the soft-finger contact model (right).

external wrench $\boldsymbol{w}$ there exist contact forces $\{\boldsymbol{f}_i\}_{i=1}^k$ such that:

$$-\boldsymbol{w} = \sum_{i=1}^{k} G_i \boldsymbol{f}_i. \quad \boldsymbol{f}_i \in \mathcal{F}_i, \quad i = 1, \ldots, k,$$

or equivalently such that:

$$-\boldsymbol{w} \in \mathcal{W}.$$

This definition implies that the grasp wrench space must satisfy $\mathcal{W} = \mathbb{R}^n$ for a force closure grasp, which implicitly assumes that the contact forces can be infinite in magnitude.[13] Since real hardware has limitations on the magnitude of the applied contact forces, a more practical definition of force closure is to be able to *resist* any external wrenches. The conditions for force closure can be summarized by the following theorem:

[13] For 2D objects $n = 3$ and for 3D objects $n = 6$.

Theorem 26.2.3. *In an n-dimensional vector space with:*

$$\mathcal{W} := \{\boldsymbol{w} \mid \boldsymbol{w} = \sum_{k=1}^{N} \beta_k \boldsymbol{w}_k, \quad \beta_k \geq 0\},$$

$\mathcal{W} = \mathbb{R}^n$ *if and only if the set* $\{\boldsymbol{w}_k\}_{k=1}^N$ *contains at least* $n+1$ *vectors, n of the vectors are linearly independent, and there exists scalars* $\beta_k > 0$ *such that:*

$$\sum_{k=1}^{N} \beta_k \boldsymbol{w}_k = 0.$$

From a practical perspective this theorem specifies a minimum number of different wrenches that must be used as a basis for the grasp wrench space, and also states that it must be possible for the grasp to apply zero wrench *even when some of the contact forces are non-zero*. These conditions are equivalent to saying that grasp wrench space $\mathcal{W}$ must contain the origin in its *interior*[14].

[14] The grasp is not in force closure if the origin is on the *boundary* of $\mathcal{W}$.

Note that in the practical case where the applied contact forces are assumed to be bounded, the conditions of Theorem 26.2.3 must still hold to guarantee the origin is in the interior of $\mathcal{W}$, which is required to resist any external wrench. The implications of this theorem are explored further in the following examples.

Example 26.2.1 (2D Object (Forces Only))**.** Consider a simplified 2D problem where instead of complete force closure (i.e. ability to withstand any *wrench*) it is sufficient to only require the cancellation of external *forces*. In this case $n = 2$ and Theorem 26.2.3 states that it must be possible for the grasp to generate 3 force vectors where 2 are linearly independent and where:

$$\beta_1 \boldsymbol{f}_1 + \beta_2 \boldsymbol{f}_2 + \beta_3 \boldsymbol{f}_3 = 0, \quad \beta_1, \beta_2, \beta_3 > 0.$$

Two examples grasps are shown in Figures 26.8 and 26.9. In Figure 26.8 the three contacts are frictionless, but even though there are 3 possible force vectors with 2 linearly independent, there is no way to generate zero force with non-zero forces at each contact! Therefore this grasp does not have force closure.

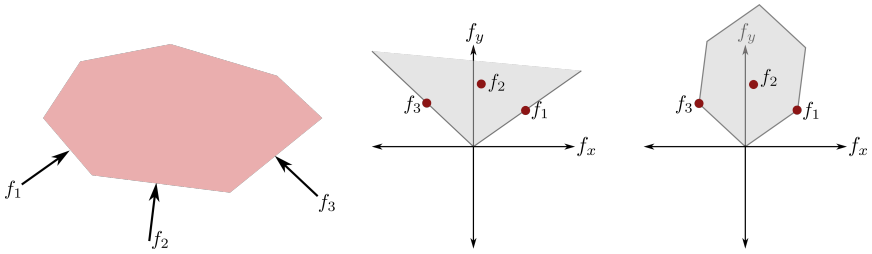


Figure 26.8: A 2D grasp with frictionless contacts that cannot compensate for all possible external forces on the object. The middle and right-side figures show the space of all possible applied forces for the cases of unbounded and bounded contact force magnitude, respectively.

Alternatively, Figure 26.9 shows a case where it is possible to have force closure using a point contact without friction and a point contact with friction (a hypothetical example). In this case all of the conditions in Theorem 26.2.3 are satisfied, and it can be seen that the origin is contained in the interior of the space of possible applied forces.

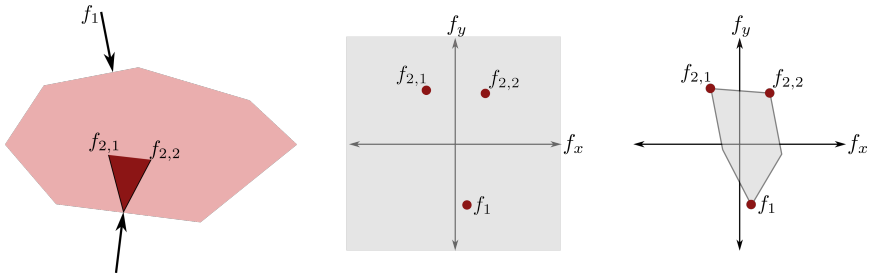


Figure 26.9: A 2D grasp consisting of a contact with friction and a contact without friction. The middle and right-side figures show the space of possible forces for the cases of unbounded and bounded magnitude, respectively. In the unbounded case it is possible to compensate for any external force, and in the bounded case it is possible to *resist* an arbitrary force.

Example 26.2.2 (2D Object)**.** In the more general case with 2D objects where the torque is also considered, the grasp wrench space is in 3D (i.e. $\mathcal{W} \subseteq \mathbb{R}^3$). Therefore, Theorem 26.2.3 states the grasp wrench space satisfies $\mathcal{W} = \mathbb{R}^3$ if and only if it is possible for the grasp to generate at least 4 different wrenches, with 3 being linearly independent, and where:

$$\beta_1 \boldsymbol{w}_1 + \beta_2 \boldsymbol{w}_2 + \beta_3 \boldsymbol{w}_3 + \beta_4 \boldsymbol{w}_4 = 0, \quad \beta_1, \beta_2, \beta_3, \beta_4 > 0.$$

If frictionless contacts are assumed these conditions require *at least* 4 contact points and in the friction case *at least* 2 contacts are required.

Consider again the grasp shown in Example 26.1.1 (Figure 26.6). The 4 edges of the friction cones create a set of 3 linearly independent wrenches, but there is no way to generate zero wrench with non-zero contact forces. This is evident in the fact that it is not possible to generate a negative torque, which means the grasp is not in force closure[15]. An alternative grasp that is in force closure is shown in Figure 26.10, which leverages a third contact point to ensure the grasp achieves stability.

[15] Notice again that the origin is not contained in the interior of the grasp wrench space!

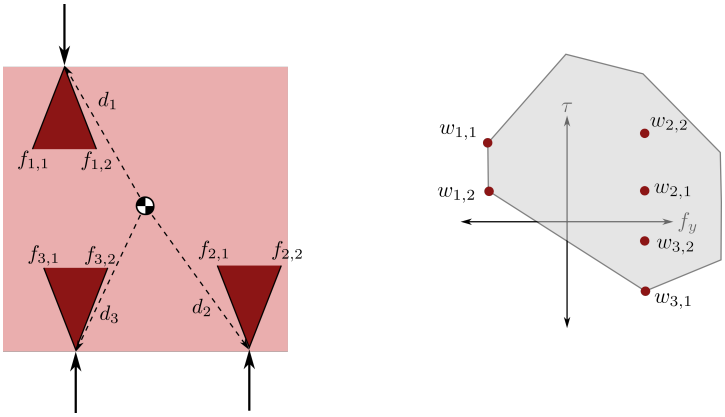


Figure 26.10: A 2D grasp consisting of three point contacts with friction. The friction cones shown in the figure on the left yield the grasp wrench space in the figure on the right (showing only the vertical force and torque dimensions and assuming bounded contact force magnitudes). This grasp is in force closure because it can resist any external wrench (the origin is contained in the interior of $\mathcal{W}$).

Example 26.2.3 (3D Object)**.** For 3D objects the grasp wrench space is in 6D (i.e. $\mathcal{W} \subseteq \mathbb{R}^6$). Theorem 26.2.3's conditions therefore require that the grasp to be able to generate at least 7 different wrenches, with 6 being linearly independent, and where:

$$\sum_{k=1}^{7} \beta_k \boldsymbol{w}_k = 0, \quad \beta_k > 0.$$

If frictionless contacts are assumed these conditions require *at least* 7 contact points and in the friction case *at least* 3 contacts are required.

26.2.2 *Grasp Wrench Hull*

The grasp wrench space $\mathcal{W}$ defines the set of all possible wrenches that can be applied to an object by a grasp, but unfortunately computing this set can be quite cumbersome in practice. One alternative approach for characterizing a grasp is through the definition of the *grasp wrench hull*, which can be efficiently computed. Given a set of linearized friction cones $\mathcal{F}_i$ defined by the set of bounded forces $\{\boldsymbol{f}_{i,1}, \boldsymbol{f}_{i,2}, \ldots, \boldsymbol{f}_{i,m}\}$ for each contact in the grasp, the wrench hull $\tilde{\mathcal{W}}$ is mathematically defined as:

$$\tilde{\mathcal{W}} = \{\boldsymbol{w} \mid \boldsymbol{w} = \sum_{i=1}^{k}\sum_{j=1}^{m} \alpha_{i,j}\boldsymbol{w}_{i,j}, \quad \boldsymbol{w}_{i,j} = \begin{bmatrix} \boldsymbol{f}_{i,j} \\ \lambda(\boldsymbol{d}_i \times \boldsymbol{f}_{i,j}) \end{bmatrix}, \quad \sum_{i=1}^{k}\sum_{j=1}^{m} \alpha_{i,j} = 1, \quad \alpha_{i,j} \geq 0\},$$

where $\boldsymbol{d}_i$ is again the vector from the object center of mass to contact point i. Note that this is almost identical to the grasp wrench space definition ex-

cept that the constraint $\sum_{j=1}^{m} \alpha_{i,j} \leq 1$ for all i has been replaced by the constraint $\sum_{i=1}^{k} \sum_{j=1}^{m} \alpha_{i,j} = 1$. Put simply, the wrench hull is the convex hull of the wrenches $\boldsymbol{w}_{i,j}$! The difference between the grasp wrench space and the wrench hull is shown in Figure 26.11 for the grasp from Example 26.2.2.

Importantly the property $\tilde{\mathcal{W}} \subseteq \mathcal{W}$ holds by definition. Therefore grasp force closure is also guaranteed when the origin is in the interior of the wrench hull space. This fact, coupled with the fact that $\tilde{\mathcal{W}}$ is easier to compute than $\mathcal{W}$, makes it a useful characterization of grasps for evaluating grasp quality.

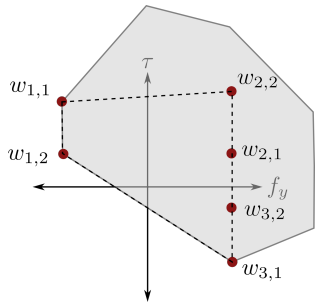


Figure 26.11: Difference between grasp wrench space $\mathcal{W}$ (grey area) and wrench hull $\tilde{\mathcal{W}}$ (area enclosed by black dashed line) for the grasp in Example 26.2.2.

26.2.3 *Grasp Quality*

If the gripper could apply contact forces with infinite magnitude then a "good" grasp could simply be defined as one that is in force closure. However a more practical definition of grasp quality should be based on the assumption that the magnitude of the contact forces is bounded. In other words, a metric for grasp quality should quantify *how well the grasp can resist external wrenches for a given bound on the contact force*.

To accomplish this, grasp quality metrics can be defined based on the definition of the grasp wrench hull $\tilde{\mathcal{W}}$. In particular, a useful metric is the radius of the largest ball centered at the origin that is completely contained in the grasp wrench hull (Figure 26.12). This metric is useful for the following reasons:

1. If the radius is zero, the origin is not contained in the interior of the wrench hull and therefore the grasp is not in force closure.

2. For a radius greater than zero, the metric represents the magnitude of the *smallest* external wrench that pushes the grasp to the limits. The direction from the origin to where the ball touches the boundary of $\mathcal{W}$ identifies the (opposite) direction in which the grasp is least able to resist external wrenches.

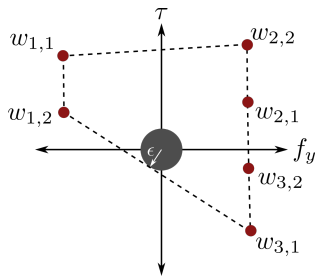


Figure 26.12: Grasp quality can be measured as the radius ϵ of the largest ball contained in the grasp wrench hull centered at the origin.

Another method for quantifying the grasp quality is to compute the volume of the grasp wrench hull $\tilde{\mathcal{W}}$. This approach provides more of an average-case metric rather than a worst-case metric, and can help differentiate between different grasp spaces that have the same worst-case metric. For example Figure 26.13 shows a grasp with the same worst-case metric as the grasp in Figure 26.12, but which would be considered worse with respect to the volumetric (average-case) metric.

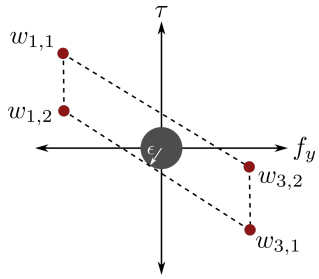


Figure 26.13: Using the volume of $\tilde{\mathcal{W}}$ as a metric for grasp quality can help differentiate between two grasps with the same worst-case performance. For example this grasp would be considered less robust than the grasp in Figure 26.12 since it has smaller volume.

26.3 *Grasp Force Optimization*

Recall from Section 26.1 that for a particular contact model a grasp map matrix G can be defined such that:

$$\boldsymbol{w} = G \begin{bmatrix} f_1 \\ \vdots \\ f_k \end{bmatrix}, \tag{26.4}$$

where $\boldsymbol{f}_i$ is the force vector associated with contact point i and $\boldsymbol{w}$ is the total wrench applied to the object. Additionally, recall that the matrix G also includes a rotational transformation such that the force vectors $\boldsymbol{f}_i$ are in local contact reference frames but $\boldsymbol{w}$ is represented in a common frame fixed in the object body (at the center of mass).

The next logical question to ask is how to compute force vectors $\{\boldsymbol{f}_i\}_{i=1}^k$ to achieve a desired wrench $\boldsymbol{w}$[16]. While one naive approach would be to just solve (26.4) using a least-squares method, this would fail to account for any constraints on the force vectors. In particular, this section will focus on the point contact with friction model where each force vector $\boldsymbol{f}_i = [f_x^{(i)}, f_y^{(i)}, f_z^{(i)}]^\top$ must satisfy the friction cone constraint:

$$\sqrt{{f_x^{(i)}}^2 + {f_y^{(i)}}^2} \leq \mu_{s,i} f_z^{(i)}, \quad f_z^{(i)} \geq 0,$$

where $\mu_{s,i}$ is the static friction coefficient for contact i. In this section the compact notation:

$$\boldsymbol{f}_i \in \mathcal{F}_i, \quad \mathcal{F}_i \coloneqq \{\boldsymbol{f} \in \mathbb{R}^3 \mid \sqrt{f_x^2 + f_y^2} \leq \mu_{s,i} f_z, \quad f_z \geq 0\},$$

for the friction cone constraint will again be used. It might also be desirable to include additional constraints on the force vectors, for example to account for hardware limitations (e.g. torque limits) or kinematic constraints. These additional constraints will be generally referred to by a convex constraint set C, such that $\boldsymbol{f}_i \in C$ is required for all $i = 1, \ldots, k$.

To summarize, the problem is to find a set of force vectors $\{\boldsymbol{f}_i\}_{i=1}^k$ such that (26.4) is satisfied[17] and such that $\boldsymbol{f}_i \in \mathcal{F}_i$ and $\boldsymbol{f}_i \in C$ for $i = 1, \ldots, k$. This problem can then be solved by formulating it as a convex optimization problem[18,19]:

$$\begin{aligned} \underset{\boldsymbol{f}_i,\ i \in \{1,\ldots,k\}}{\text{minimize}} \ & J(\boldsymbol{f}_1, \ldots, \boldsymbol{f}_k), \\ \text{s.t.} \ & \boldsymbol{f}_i \in \mathcal{F}_i, \quad i = 1, \ldots, k, \\ & \boldsymbol{f}_i \in C, \quad i = 1, \ldots, k, \\ & \boldsymbol{w} - G \begin{bmatrix} \boldsymbol{f}_1 & \ldots & \boldsymbol{f}_k \end{bmatrix}^\top = 0, \end{aligned} \tag{26.5}$$

where $J(\cdot)$ is the objective function that is convex in each $\boldsymbol{f}_i$. Simply choosing $J = 0$ would result in a convex feasibility problem, but a more common choice is:

$$J(\boldsymbol{f}_1, \ldots, \boldsymbol{f}_k) = \max\{\|\boldsymbol{f}_1\|, \ldots, \|\boldsymbol{f}_k\|\},$$

which is the maximum applied force magnitude among all contact points.

Note that the fundamental disadvantage of this optimization-based approach is that the positions of all contacts with respect to the object's center of mass and the object's friction coefficients are assumed to be known. It is also assumed that the desired wrench $\boldsymbol{w}$ is known!

[16] The desired wrench may be used to counter an external disturbance (to maintain equilibrium) or to *manipulate* the object.

[17] In the case that the desired wrench is used to counter an external disturbance, the condition (26.4) is referred to as the equilibrium constraint.

[18] S. Boyd and B. Wegbreit. "Fast Computation of Optimal Contact Forces". In: *IEEE Transactions on Robotics* 23.6 (2007), pp. 1117–1132

[19] The problem is technically a second-order cone program because of the friction cone constraints.

26.4 Learning-Based Approaches to Grasping

Model-based methods for grasp evaluation and optimization require several assumptions that may be either difficult to validate in practice, or may not even be valid in all scenarios. These assumptions include:

1. A coulomb (static) friction model defines the friction cone, and the coefficient μ_s is known.

2. The object's geometry and mass distribution is known[20], such that given a contact point the vector $\boldsymbol{d}_i$ from the object's center of mass to the contact is known.

3. The object is a rigid body.

4. The desired forces $\boldsymbol{f}_i$ can be applied perfectly.

[20] One option would be to build a database of known objects, but this may not be scalable to real world problems.

Learning-based methods for grasp analysis[21] can leverage data to decrease reliance on these assumptions, for example by not requiring explicit knowledge of the object's physical parameters. Learning-based methods can also combine the task of grasping with other parts of the manipulation pipeline, such as perception and motion planning.

This section will introduce some recent learning-based approaches to robotic grasping, which is still a very active area of research. Specifically, these examples will demonstrate several learning-based strategies including approaches that create synthetic training data from model-based simulators and approaches that use real hardware to generate data.

[21] J. Bohg et al. "Data-Driven Grasp Synthesis—A Survey". In: *IEEE Transactions on Robotics* 30.2 (2014), pp. 289–309

26.4.1 Choosing a Grasp Point from an RGB Image[22]

The objective of this supervised learning approach was to learn how to find a good grasp point in an RGB image of an object, and then generate a prediction of the point's 3D position. Since supervised learning techniques can require a lot of training data, this approach auto-generated training images *synthetically* using realistic rendering techniques (see Figure 26.14). The use of synthetic data also made it easier to collect a diverse training set including images with different lighting, object color, and object orientation and size.

Once a model was trained to produce good grasp point classifications, 3D predictions of the target grasp position were generated by *structure-from-motion*, where two images were used to triangulate the point in space. While there are certainly limitations to this approach, this work produced promising results, including good grasp success rates on novel objects (that weren't included in the training dataset). This work also had substantial influence on future learning-based grasping and manipulation approaches.

[22] A. Saxena, J. Driemeyer, and A. Ng. "Robotic Grasping of Novel Objects using Vision". In: *The International Journal of Robotics Research* 27.2 (2008), pp. 157–173

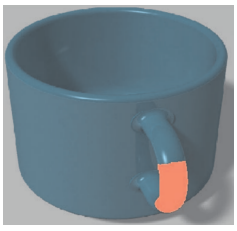


Figure 26.14: Synthetic image of a cup and its labeled grasp point from Saxena et al. (2008).

26.4.2 *Exploiting Simulation and a Database of Objects to Predict Grasp Success*[23]

The Dex-Net approach for learning to grasp is another supervised learning approach that relies on simulations to generate training data. Specifically this approach assumes a parallel jaw gripper and contains a large ($> 10,000$) database of 3D object models. For each model in the Dex-Net database, the simulator uses analytical (model-based) techniques to evaluate a large number of potential grasps for probability of success[24]. This is accomplished by empirically sampling a grasp some number of times and determining (through simulation) the percentage that result in force closure[25]. The database of objects and potential grasps can then be used to train a model to predict the probability of force closure for a new grasp/object.

In practice a number of candidate grasps are generated for a given (potentially novel) object, are evaluated by the learned model to predict probability of success, and then a *multi-armed bandit*[26] approach is used to select which grasp to take. The learned models are then updated based on the outcome of the action for continuous improvement. This work showed that leveraging the prior information from the object database can significantly improve grasping for new objects (even if they are not in the database), and later improvements have enhanced the approach even further.

26.4.3 *Learning to Grasp Through Real-World Trial-and-Error*[27]

Instead of leveraging simulators to generate synthetic data this work uses hardware experiments to generate real-world data. The resulting experiences are then used in a self-supervised approach to learn an *end-to-end* framework to grasp objects in cluttered environments. One of the reasons this work is significant is the lack of assumptions that are made: 3D object models are not needed, only RGB images are required, it does not use contact models or simulated data, no physical object information is used, and no hand-engineered path/grasp planning algorithms are used. Instead the system just learns through trial-and-error, exploring approaches to actuate the robot arm and gripper that eventually lead to robust grasps.

This approach showed impressive results over hand-designed or open-loop approaches, but at the cost that it took six months and a large number of robots to generate enough training data.

26.5 *Learning-Based Approaches to Manipulation*

The previous sections of this chapter have focused on the problem of grasping, but many robotic manipulation tasks involve more than simply grasping an object. For example it is possible to manipulate objects *without* force closure grasps, such as by pushing the object[28]. Many manipulation tasks that do involve grasping also involve other complex steps, such as using the grasped object to manipulate other objects (e.g. hitting a hammer with a nail) or placing

[23] J. Mahler et al. "Dex-Net 1.0: A cloud-based network of 3D objects for robust grasp planning using a Multi-Armed Bandit model with correlated rewards". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1957–1964

[24] The Dex-Net database consists of 2.5 million tested grasps.

[25] There is simulated uncertainty in object and gripper pose, as well as the surface friction.

[26] A fundamental reinforcement learning problem focused on uncertain decision making.

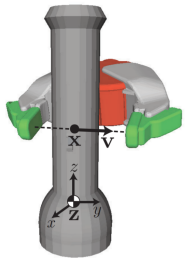


Figure 26.15: Dex-Net grasps are parameterized by the centroidal position of the gripper x and the approach direction v, Mahler et al. (2016).

[27] S. Levine et al. "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection". In: *The International Journal of Robotics Research* 37.4-5 (2018), pp. 421–436

[28] Not only is it possible, but may be necessary if the object is too large or heavy to grasp.

the object in a certain position (e.g. inserting a key into a lock). This section will introduce at a high-level some interesting and foundational problems in manipulation, and present the high-level ideas found in some recent research on learning-based approaches to solving them.

26.5.1 Planar Pushing

Planar pushing is a fundamental manipulation task where the goal is to control the pose of an object in a 2D setting by only using "pushing" contacts. While the contact point models used for grasping can also be applied in this problem, the interaction of the object and the surface must also be accounted for.

Similar to the physics-based contact models for grasping, physics-based models can also be developed to predict the sliding interactions between the object and the surface. In particular, the concept of a *friction limit surface*[29] can be used to model the interaction between the object and the surface. The friction limit surface is a boundary in wrench space that separates wrenches that the surface can apply to the object through friction and those it can't. The part of the wrench space enclosed by this surface will contain the origin (i.e. the zero wrench), and most importantly whenever the object is *slipping* the wrench applied on the object lies *on the friction limit surface*. This surface can be determined numerically if the coefficient of friction, the contact area, and the pressure distribution are known. For simplicity, it is common to approximate this surface as an ellipsoid. To summarize:

[29] I. Kao, K. Lynch, and J. Burdick. "Contact Modeling and Manipulation". In: *Springer Handbook of Robotics*. Springer, 2016, pp. 931–951

1. If an external wrench applied to the object is within the region of the wrench space enclosed by the friction limit surface, friction between the object and the surface will cause the object to remain motionless.

2. If the part slides quasistatically[30], the pushing wrench must lie on the friction limit surface and the motion (velocity) of the object can be determined.

[30] Assumption that the part moves slowly enough that inertial effects are negligible.

The friction limit surface provides the foundation for a physics-based model that predicts how an object will slide across a surface under external contact forces. Such a model could be used to design a controller (e.g. with model predictive control) for planar pushing tasks. However, these physics-based models are based on approximations and assumptions that may impact their accuracy or applicability to real problems. In fact some studies have been performed to evaluate the accuracy of physics-based pushing models[31].

Assumptions in physics-based pushing model: ellipsoidal friction limit surface, coulomb friction, perfectly planar object/surface, rigid body object, physical properties of object are known.

While physics-based controllers such as model predictive control can handle some uncertainty via feedback control mechanisms, it is still desirable to improve the modeling accuracy and eliminate assumptions requiring knowledge of the parameters that define the models[32]. Learning-based approaches are one possible solution to some of these challenges, where real-world data can be used to either completely replace or augment the physics-based models.

In fact, recent work[33] has compared the use of physics-based, hybrid (physics + learning), and learning-based models for planar pushing tasks. In this work

[31] K. Yu et al. "More than a million ways to be pushed. A high-fidelity experimental dataset of planar pushing". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 30–37

[32] For example the physical properties of the object and surface.

[33] A. Kloss, S. Schaal, and J. Bohg. "Combining learned and analytical models for predicting action effects from sensory data". In: *The International Journal of Robotics Research* (2020)

the hybrid model learned a mapping from sensor measurements (RGB images) into a set of parameters that were required for the physics-based motion model and the learning-based model just directly learned a single neural network for mapping sensor measurements to motion predictions directly. As might be expected the hybrid approach achieved better generalization by leveraging the physics-based model's structure, while the learning-based approach overfit to the training data[34].

[34] This is a classic example of bias-variance tradeoff in modeling.

26.5.2 Contact-Rich Manipulation Tasks

Many 6D manipulation problems involve grasping an object and then using it to physically interact with the environment. Classic everyday examples include hitting a nail with a hammer, inserting a key into a lock, and plugging a cord into an electrical outlet. These types of *contact-rich* tasks tasks typically rely on multiple different sensing modalities including haptic and visual feedback. Consider the task of inserting a key into a lock: without sight it would be challenging to correctly position the key and without tactile sensing (e.g. force/torque sensing) it would be challenging to know when the key is perfectly aligned and can be inserted.

However, it can be quite challenging to *integrate* multiple sensing modalities toward a common task, especially when the sensing modalities are so different and since manipulation tasks can be quite complex. One approach may be to individually develop systems for different subtasks and manually find a common interface to stitch them together, however this could be challenging from a system engineering perspective. An alternative is to use machine learning techniques to automatically integrate the sensing modalities.

One learning-based approach to this problem is to design an end-to-end system that takes as input all sensor data streams and outputs actions for the robot to execute the task. However, when implemented in a naive way (e.g. a single massive neural network architecture) end-to-end approaches can be data inefficient. An alternative is to add additional structure to the learning-based approach by leveraging some insights into the problem, similar to how the physics-based motion model was used in the learning-based planar pushing example discussed in the previous section.

A structured approach for manipulation tasks relying on multiple sensing modalities is introduced by Lee et al.[35]. In this work an end-to-end system that takes sensor data streams as input and outputs robot actions is split into two parts: first transforming the multi-modal sensor data streams into a low-dimensional feature representation that contains task relevant information[36], and then using these features as the input to a learned policy that generates robot actions. In other words, the insight is that the learning process can be made more efficient by first learning a way to compress and summarize all of the sensor data, and then learning how to use the summarized information to generate a good policy. Another benefit to this approach is that the sensor

[35] M. Lee et al. "Making Sense of Vision and Touch: Self-Supervised Learning of Multimodal Representations for Contact-Rich Tasks". In: *International Conference on Robotics and Automation (ICRA)*. 2019, pp. 8943–8950

[36] This is accomplished by training an autoencoder network.

data *encoder* can generalize more effectively to new tasks, meaning that only the policy portion needs to be retrained!

Part VII

Appendices

A
Machine Learning

Many algorithms and tools in robotic autonomy leverage models of the world that are often based on first-principles: physics-based kinematic models are used to design controllers, sensor models are used in localization algorithms, and geometric principles are used in understanding stereo vision. However, there are also many scenarios in robotics where these techniques may fail to capture the complexity of unstructured real-world environments. For example, how can a stop-sign be reliably detected in camera images when it could be rainy, foggy, or dark out, or when the stop-sign is partially occluded? Are there first-principles models that can accurately predict the behavior of a human driver, and distinguish between aggressive and defensive driving behavior? How can a robot be programmed to pick up objects with an infinite number of variations in size, shape, color, and texture? In the last few decades, advancements in *machine learning*[1] have led to start-of-the-art approaches for many of these challenging problems[2]. This chapter presents an introduction to machine learning to provide a knowledge of the fundamental tools that are used in learning-based algorithms for robotics, including computer vision, reinforcement learning, and more.

[1] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer, 2017

[2] Of course in many settings it is beneficial to use first-principles and machine learning techniques *in concert*.

Machine Learning

At their most fundamental level, machine learning techniques seek to extract useful patterns from *data*[3], and are typically classified as either *supervised* or *unsupervised*.

[3] In many cases the data will come from real world experiments, but in other cases may come from simulation.

Definition A.0.1 (Supervised Learning)**.** Given a collection of n data points:

$$\{(x_1, y_1), \ldots, (x_n, y_n)\},$$

where x_i is an input variable and y_i is an output, the *supervised learning* problem is to find a function $y = f(x)$ that fits the data and can be used to predict outputs y for new inputs x.

Definition A.0.2 (Unsupervised Learning)**.** Given a collection of n data points $\{x_1, \ldots, x_n\}$, the *unsupervised learning* problem is to find patterns in the data.

Supervised learning problems, such as regression and classification[4], are generally more common in robotics applications and will be the focus of this chapter. For example, robotic imitation learning-based controllers[5] can be expressed as a regression problem where the input x is the state of the robot and y is the action the robot should take. Classification problems also arise frequently in robotic computer vision, for example to identify whether the image x belongs to a particular class y (e.g. a dog or cat).

In both regression and classification problems, the learned function f is categorized as either *parametric* or *non-parametric*. Parametric functions are generally more structured and can be written down in an analytical form[6], while non-parametric functions are generally defined by the data points themselves[7]. The best choice between parametric or non-parametric functions is generally dependent on the particular problem and the type of data available. However, some of the most popular choices are parametric, such as polynomials and *neural networks*.

[4] In regression problems the output y is continuous and in classification problems the output y is discrete (categorical).

[5] Imitation learning refers to the process of learning to mimic a policy (e.g. from an expert) through example decisions.

[6] The most basic parametric function would be a linear function $f(x) = Wx$, parameterized by the "weight" matrix W.

[7] In the non-parametric k-nearest neighbors method, the value $f(x)$ is defined by the value of the data points y_i corresponding to the k closest points, x_i, to x.

A.1 Loss Functions

In supervised learning problems, a metric known as a *loss function* is used to evaluate and compare candidate models $f(x)$ that could be used to fit the data. Many loss functions for supervised learning problems exist, but some of the most common examples include the l_2 and l_1 loss (for regression) and the $0-1$ and cross entropy loss (for classification).

1. The l_2 loss function is defined by:

$$L = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2, \tag{A.1}$$

   where the summation is over some set of data points (x^i, y^i). From this loss function it can be seen that a penalty arises from the function f not perfectly matching the data at the sampled data points, but most importantly that the penalty is *quadratic* with respect to this residual. This loss function will therefore favor more small residuals over a few large residuals, which tends to make the model perform better "on average". However this also makes the l_2 loss sensitive to outliers in the data, making the training less robust.

2. The l_1 loss function is defined by:

$$L = \frac{1}{n}\sum_{i=1}^{n}|f(x_i) - y_i|. \tag{A.2}$$

   Unlike the l_2 loss, this loss function only penalizes the *absolute value* of the residual. Therefore this loss function will favor all residuals on a more equal footing and generally leads to a more robust training procedure that is less sensitive to outliers in the data.

3. The $0-1$ loss function is defined by:

$$L = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\{f(x_i) \neq y_i\}, \tag{A.3}$$

where $\mathbf{1}\{\cdot\}$ is the indicator function. This loss function can be used in classification problems and provides a loss of 1 whenever the classification is incorrect, and 0 otherwise. However, the use of this loss function introduces practical issues when training with gradient-based optimization, since this function is either flat or not differentiable at all points in the domain.

4. The cross entropy loss function[8] is defined by:

$$L = -\frac{1}{n}\sum_{i=1}^{n} y_i^\top \log f(x_i), \tag{A.4}$$

and is a common loss function in classification problems. To get an intuitive feeling for how the cross entropy loss works, consider a classification problem where the classes are $c \in \{1, 2, \ldots, C\}$ and where the function $f(x_i)$ outputs a *vector* of the probabilities of each class (which is normalized to sum to 1)[9]. Additionally, for each data point the vector y_i is a "one-hot" vector[10] specified by the class associated with x_i. Therefore the loss for a particular data point can be written as:

$$-y_i^\top \log f(x_i) = -\begin{bmatrix}0, \ldots, 1, \ldots, 0\end{bmatrix}\begin{bmatrix}\log f_1(x_i) \\ \vdots \\ \log f_C(x_i)\end{bmatrix} = -\log f_c(x_i),$$

where C is the number of classes, the 1 element in y_i is in the position corresponding to the correct class, and $f_c(x_i)$ is the probability of the correct class output by the model. Thus, to minimize the loss for this particular data point it is good to make $f_c(x_i) = 1$ (in fact as $f_c(x_i) \to 0$ the loss approaches infinity!). Cross entropy loss can also be derived from a statistical perspective, where it can be shown to be the same as maximizing the log-likelihood over all data points.

[8] Cross entropy loss is more practical than $0-1$ loss since it is a differentiable function.

[9] This can be accomplished by using the *softmax* function.

[10] A one-hot vector is a vector with all zeros and a single 1.

A.2 Model Training

In supervised learning problems with a predetermined parametric model (e.g. linear model or neural network), the *values* of the parameters can be optimized to best fit the data (i.e. minimize the specified loss function). This process of parameter optimization is referred to as *model training*. While in some special cases the optimal set of parameters can be computed analytically, it is more common to search for a good set of parameters in an iterative fashion using numerical optimization techniques.

Example A.2.1 (Linear Least Squares)**.** One of the most fundamental regression problems, linear least squares, can be solved analytically. In this problem, the parametric model is a linear model[11]:

$$f(x) = \theta^\top x,$$

where $x \in \mathbb{R}^p$ is the input and $\theta \in R^p$ is the set of model parameters, and the loss function is the l_2 loss (A.1). Given n data points (x_i, y_i), the loss function can be expressed in matrix form as:

$$L(\theta) = \frac{1}{n}\|Y - X\theta\|_2^2,$$

where the matrix $Y \in \mathbb{R}^n$ and $X \in \mathbb{R}^{n\times p}$ are defined by the data as:

$$Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}, \quad X = \begin{bmatrix} x_1^\top \\ \vdots \\ x_n^\top \end{bmatrix}.$$

The parameters θ are then chosen to minimize the loss function by taking the derivative:

$$\frac{dL}{d\theta} = \frac{2}{n}X^\top X\theta - \frac{2}{n}X^\top Y,$$

and setting it equal to zero, which gives $\theta^* = (X^\top X)^{-1}X^\top Y$.[12]

[11] This approach can also be extended to nonlinear settings through the use of basis functions. In particular the model becomes $f(x) = \theta^\top \phi(x)$, where $\phi(x)$ are nonlinear basis functions (sometimes referred to as *features*.

[12] Note that directly computing the inverse of $X^\top X$ may be challenging, but alternative numerical methods exist to compute the value of θ^* that satisfies the necessary condition of optimality.

A.2.1 Numerical Optimization

In many cases parameter optimization cannot be performed analytically and therefore numerical optimization algorithms are used. Two of the most fundamental algorithms for numerical optimization-based training of parametric models are *gradient descent* and *stochastic gradient descent*[13].

In gradient descent, the parameters $\theta \in \mathbb{R}^p$ of a model $f_\theta(x)$ are iteratively updated by:

$$\theta \leftarrow \theta - \eta\nabla_\theta L(\theta),$$

where $\nabla_\theta L(\theta)$ is the gradient of the loss function with respect to the parameters and the hyperparameter η is referred to as the *learning rate* or *step-size*. By leveraging the gradient, this update rule seeks to iteratively improve the parameters to incrementally decrease the loss.

Notice that the gradient of the loss can be written as:

$$\nabla_\theta L(\theta) = \frac{1}{n}\sum_{i=1}^{n}\nabla_\theta L_i(\theta),$$

where L_i is the term of the loss function associated with the i-th data point. Therefore computing the gradient of the loss function could be computationally intensive if the number of data points is very large. To address this issue, stochastic gradient descent uses an approximation of the gradient computed by randomly sampling the gradients over a smaller *batch* of data points S[14]:

[13] Gradient descent is referred to as a *first-order* method.

[14] The batch S is resampled at every iteration of the algorithm.

$$\nabla_\theta L(\theta) \approx \frac{1}{|S|} \sum_{i \in S \subset \{1,\dots,n\}} \nabla_\theta L_i(\theta),$$

where $|S|$ is the number of data points in the batch.

Beyond gradient descent approaches lie a broad set of additional numerical optimization algorithms that are commonly used in practice[15]. Often times these advanced methods may lead to faster learning rates or more robust learning, and some algorithms may also be more applicable to problems with larger amounts of data or larger numbers of model parameters.

[15] M. J. Kochenderfer and T. A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019

A.2.2 Training and Test Sets + Regularization

In supervised learning with parametric models, the goal is to train a model $f(x)$ that accurately predicts the output y for inputs x that are not seen in the data set. In other words, the goal is to find a model that *generalizes* to unseen data. It is important to note however that simply optimizing the loss function over a dataset *does not* guarantee that the model generalizes well, since it is possible to *overfit* the model to the data.

A model is overfit to a set of data if it predicts the set of data well (i.e. has a low loss) but fails to accurately predict new data. To counter this issue, one very common practice in machine learning is to split the full dataset into two parts: a training set and a test set[16]. As the names suggest, the model can be trained with the training data and then the test set can be used to verify whether overfitting has occured. To test for overfitting, the loss function can be evaluated over both sets of data. Overfitting has occured if the training loss is significantly lower than the test loss.

[16] There isn't an optimal way to split the data, but common splits range from 80/20 training/test to 50/50 training/test.

While splitting the data into training and test sets provides a good way to *verify* whether the learned model generalizes well, there are also techniques that can be employed in during the training process to avoid overfitting. In particular, the most common technique is known as *regularization*. One form of regularization is implemented by adding terms to the loss function to penalize "model complexity". For example, with a model $f_\theta(x)$ parameterized by the vector θ, two common forms of regularization include:

1. l_2 regularization, which consists of the addition of the term $\|\theta\|_2$ to the loss function,

2. l_1 regularization, which which consists of the addition of the term $\|\theta\|_1$ to the loss function.

A.3 Neural Networks

One very common parametric model used in machine learning is the *neural network*[17]. Neural networks are models with very specific structures, consisting of a hierarchical sequence of linear and nonlinear functions, which makes them

[17] Also known as the multi-layer perceptron.

very powerful function approximators. Mathematically, neural networks are typically described as a sequence of functions:

$$\begin{aligned} h_1 &= f_1(W_1 x + b_1), \\ h_2 &= f_2(W_2 h_1 + b_2), \\ &\vdots \\ \hat{y} &= f_K(W_K h_{K-1} + b_K), \end{aligned} \tag{A.5}$$

which is an easier notation than writing the equivalent composite function:

$$\hat{y} = f_K(W_K f_{K-1}(\dots) + b_K).$$

In this model, the parameters are the weights $W_1, \dots, W_K$ and biases $b_1, \dots, b_K$, and the structure of the model is predefined by the choice of the *activation functions* $f_1, \dots, f_K$ and the number of *layers* K. The intermediate variables $h_1, \dots, h_{K-1}$ are the outputs of the *hidden layers*, aptly named since they are not the input or the output of the model.

To fully specify the structure of the model, a practitioner needs to specify the number of hidden layers[18], the dimensionality of each of the intermediate variables h_i (usually chosen to be the same for all hidden layers), and the activation functions f_i.

[18] Neural networks with many layers are referred to as *deep neural networks*.

A.3.1 Activation Functions

Commonly used activation functions $f_1, \dots, f_K$ in neural networks include sigmoid functions, hyperbolic tangent functions, rectified linear units (ReLU), and leaky ReLU functions[19].

[19] It is typical for the same activation function to be used for all layers of the network.

1. Sigmoid function (also denoted as $\sigma(x)$):

$$f(x) = \frac{1}{(1 + e^{-x})},$$

2. Hyperbolic tangent function:

$$f(x) = \tanh(x),$$

3. ReLU function:

$$f(x) = \max\{0, x\},$$

4. Leaky ReLU function:

$$f(x) = \max\{0.1x, x\},$$

It is important to note that each of these activation functions share two important characteristics: they are *nonlinear* and they are easy to differentiate. It is critical that the activation function be nonlinear since a composition of linear functions will remain linear, and therefore no additional benefit is gained in modeling capability by adding more than a single layer to the network. Differentiability is also critical because the gradients must be easily computable during training[20].

[20] While ReLU and leaky ReLU are not strictly differentiable, this issue is easily mitigated in practice.

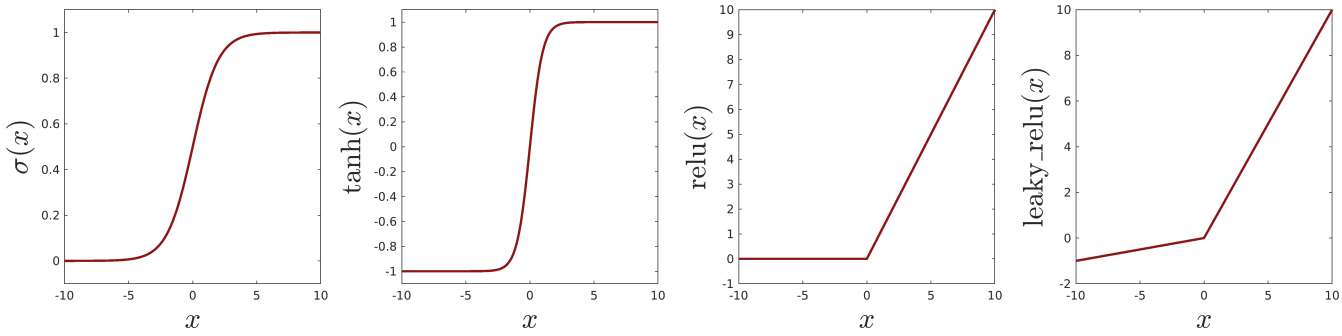


Figure A.1: Common activation functions used in neural networks.

A.3.2 *Training Neural Networks*

Neural networks are trained with gradient-based numerical optimization techniques, such as those mentioned in Section A.2.1 (e.g. stochastic gradient descent). Therefore once a particular loss function L has been chosen, the gradients $\frac{\partial L}{\partial \theta}$ must be computed for each parameter. Since neural networks can contain a large number of parameters, this gradient computation must be accomplished in a computationally efficient way. In particular, the gradients are computed using an algorithm referred to as *backpropagation*, which leverages the chain rule of differentiation and the layered structure of the network.

As with other parametric models, it is very important to avoid overfitting when training neural networks[21]. This can partially be accomplished using the division of the dataset into training and test sets, as well as by using regularization techniques as mentioned in Section A.2.2. Another technique for avoiding overfitting in neural networks is referred to as *dropout*, where some "connections" in the network are occasionally removed during the training process. This essentially forces the network to learn more redundant representations, which has been shown to improve generalization. Of course another useful technique to avoid overfitting is just to have an extremely large dataset, but in many cases this may not be very practical.

[21] It is quite easy to overfit when training neural networks since they have such a large number of parameters.

A.4 *Backpropagation and Computational Graphs*

From a theoretical standpoint, computing the gradients $\frac{dL}{d\theta}$ of the loss function with respect to the parameters is relatively straightforward. However, from a practical standpoint computing these gradients can be computationally expensive, especially for complex models such as neural networks. *Backpropagation*[22] is an algorithm that addresses this issue by computing all required gradients in an efficient way.

Backpropagation computes gradients by cleverly choosing the order in which operations required to compute the gradient are performed. By doing so it seeks to avoid redundant computations, and can in fact be viewed as an example of dynamic programming. While in some simple cases the backpropagation

[22] Sometimes also referred to as auto-differentiation.

Many software tools, such as PyTorch (`https://pytorch.org/`) and TensorFlow (`https://www.tensorflow.org/`) will automatically be able to perform backpropagation for a large class of functions.

algorithm may provide only a small advantage, in many cases (and in particular for neural network training) backprop can be orders of magnitude faster than naive approaches.

A *computational graph* is another practical tool that is useful when using the backpropagation algorithm to compute gradients. A computational graph provides a way to express a mathematical function using representations from graph theory. In particular the function is expressed as a directed graph where the nodes represent mathematical operations or function inputs and the edges represent intermediate quantities. Using a computational graph, a forward pass through the graph (starting at the root nodes, which are function inputs) is equivalent to evaluating the function.

This representation makes it very easy to see the structure of the mathematical operation that can be exploited by the backpropagation algorithm. As an example, consider the function $L(x,y) = g(f(x,y))$ and its associated computational graph shown in Figure A.2 (which includes the intermediate variable z). Using the chain rule, the gradient of L with respect to x is $\frac{\partial L}{\partial x} = \frac{dL}{dz}\frac{\partial z}{\partial x}$. The backpropagation algorithm uses this structure to convert the computation of the gradient $\frac{\partial L}{\partial x}$ into a sequence of *local* gradient computations $\frac{dL}{dz}$ and $\frac{\partial z}{\partial x}$, corresponding to each computation node in the graph. With this structure redundant computation can be be avoided. For example, when computing $\frac{\partial L}{\partial y}$ the partial gradient $\frac{dL}{dz}$ can be reused.

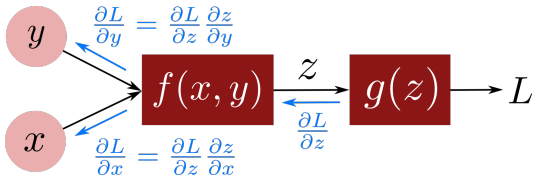


Figure A.2: Example computational graph for a function $L(x,y) = g(f(x,y))$.

To summarize, the backpropagation algorithm follows the following basic steps:

1. Perform a forward pass through the computational graph to compute any intermediate variables that may be needed for computing local gradients[23].
2. Starting from the graph output, perform a backwards pass over the graph where at each computation node the local gradient of the node with respect to its inputs and outputs is computed. Then, compute the gradient of the graph's output with respect to the inputs of the local computation node, leveraging the chain rule and previously calculated gradients. In Figure A.2, the first step of backprop would be to compute $\frac{\partial L}{\partial z} = \frac{dg}{dz}$, and the second step would use $\frac{\partial L}{\partial z}$ to compute the remaining gradients $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$.

[23] For example if $g(z) = z^2$ the gradient $\frac{dg}{dz} = 2z$ depends on the current value of z

Example A.4.1 (Training a Simple Model)**.** Consider a supervised learning problem with a parametric model defined as:

$$f(x) = (x + a)(x + b),$$

where a and b are parameters of the model, and a l_2 loss function (A.1) is used for training. A computational graph for computing the loss from a single data point with this model is shown in Figure A.3.

For this model and loss function the gradients required for training can be computed analytically as:

$$\frac{\partial L_i}{\partial a} = -2(y - (x+a)(x+b))(x+b),$$
$$\frac{\partial L_i}{\partial b} = -2(y - (x+a)(x+b))(x+a).$$

Computing the gradients in this way (the naive approach) would require 7 operations each (4 sums and 3 multiplications), for a total of 14 operations.

Alternatively the gradients can be computed in a more efficient way using backpropagation, which avoids redundant computations. This approach can be viewed as taking a *backward pass* over the computation graph. Starting at the output of the graph:

$$\frac{\partial L_i}{\partial z_1} = 2z_1.$$

Then moving on through the next operations and using the chain rule (and reusing the previous computations):

$$\frac{\partial L_i}{\partial \hat{y}} = \frac{\partial L_i}{\partial z_1}\frac{\partial z_1}{\partial \hat{y}} = -\frac{\partial L_i}{\partial z_1},$$

and:

$$\frac{\partial L_i}{\partial z_2} = \frac{\partial L_i}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z_2} = \frac{\partial L_i}{\partial \hat{y}}z_3,$$
$$\frac{\partial L_i}{\partial z_3} = \frac{\partial L_i}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z_3} = \frac{\partial L_i}{\partial \hat{y}}z_2.$$

Finally, the next step backward reaches the parameters a and b:

$$\frac{\partial L_i}{\partial a} = \frac{\partial L_i}{\partial z_2}\frac{\partial z_2}{\partial a} = \frac{\partial L_i}{\partial z_2},$$
$$\frac{\partial L_i}{\partial b} = \frac{\partial L_i}{\partial z_3}\frac{\partial z_3}{\partial b} = \frac{\partial L_i}{\partial z_3}.$$

To actually compute the numerical values of these gradients:

1. First perform a forward pass through the network to compute the values z_1, z_2, and z_3 (5 operations).
2. Then perform the backward pass computations to compute $\frac{\partial L_i}{\partial z_1}$, $\frac{\partial L_i}{\partial \hat{y}}$, $\frac{\partial L_i}{\partial z_2}$, $\frac{\partial L_i}{\partial z_3}$, $\frac{\partial L_i}{\partial a}$, and $\frac{\partial L_i}{\partial b}$ (4 operations).

Using backpropagation, only 9 operations are required to compute the gradients $\frac{\partial L_i}{\partial a}$, and $\frac{\partial L_i}{\partial b}$, which is a non-negligible reduction over the naive approach!

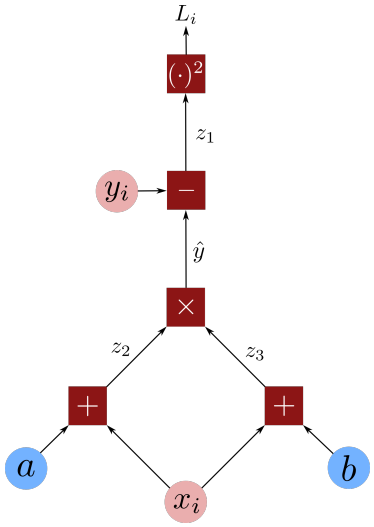


Figure A.3: Computational graph for computing the loss for a single data point for the model $f(x) = (x+a)(x+b)$ with l_2 loss (see Example A.4.1). The values (x_i, y_i) are the data point, the model output is $\hat{y}$, and z_1, z_2, z_3 are intermediate variables. The quantities a and b are parameters of the model.

Bibliography

[1] P. Abbeel and A. Ng. "Apprenticeship Learning via Inverse Reinforcement Learning". In: *Proceedings of the Twenty-First International Conference on Machine Learning*. 2004.

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

[3] A. Bajcsy et al. "Learning Robot Objectives from Physical Human Interaction". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 217–226.

[4] C. Basu et al. "Do You Want Your Autonomous Car to Drive Like You?" In: *12th ACM/IEEE International Conference on Human-Robot Interaction*. 2017, pp. 417–425.

[5] D. Bertsekas. *Reinforcement learning and optimal control*. Athena Scientific, 2019.

[6] J. Bohg et al. "Data-Driven Grasp Synthesis—A Survey". In: *IEEE Transactions on Robotics* 30.2 (2014), pp. 289–309.

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

[8] S. Boyd and B. Wegbreit. "Fast Computation of Optimal Contact Forces". In: *IEEE Transactions on Robotics* 23.6 (2007), pp. 1117–1132.

[9] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[10] R. Brooks. "A robust layered control system for a mobile robot". In: *IEEE Journal on Robotics and Automation* 2.1 (1986), pp. 14–23.

[11] E. M. Clarke et al. *Model Checking*. 2nd ed. MIT Press, 2018.

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

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

[14] 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.

[15] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[16] F. Gustafsson. *Statistical Sensor Fusion*. Studentlitteratur, 2013, p. 554.

[17] C. Harris and M. Stephens. "A combined corner and edge detector". In: *4th Alvey Vision Conference*. 1988.

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

[19] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer, 2017.

[20] 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.

[21] L. Joseph. *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*. Apress, 2018.

[22] L. Kaelbling et al. *6.01SC: Introduction to Electrical Engineering and Computer Science I*. MIT OpenCourseWare. 2011.

[23] I. Kao, K. Lynch, and J. Burdick. "Contact Modeling and Manipulation". In: *Springer Handbook of Robotics*. Springer, 2016, pp. 931–951.

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

[25] G. Katz et al. "The Marabou Framework for Verification and Analysis of Deep Neural Networks". In: *Computer Aided Verification*. 2019, pp. 443–452.

[26] 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.

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

[28] A. Kloss, S. Schaal, and J. Bohg. "Combining learned and analytical models for predicting action effects from sensory data". In: *The International Journal of Robotics Research* (2020).

[29] M. J. Kochenderfer and T. A. Wheeler. *Algorithms for Optimization*. MIT Press, 2019.

[30] D. Kortenkamp, R. Simmons, and D. Brugali. "Robotic Systems Architectures and Programming". In: *Springer Handbook of Robotics*. Springer, 2008, pp. 283–302.

[31] M. Kwiatkowska, G. Normal, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. 2011, pp. 585–591.

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

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

[34] Y. LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[35] M. Lee et al. "Making Sense of Vision and Touch: Self-Supervised Learning of Multimodal Representations for Contact-Rich Tasks". In: *International Conference on Robotics and Automation (ICRA)*. 2019, pp. 8943–8950.

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

[37] S. Levine et al. "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection". In: *The International Journal of Robotics Research* 37.4-5 (2018), pp. 421–436.

[38] 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.

[39] 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.

[40] J. Mahler et al. "Dex-Net 1.0: A cloud-based network of 3D objects for robust grasp planning using a Multi-Armed Bandit model with correlated rewards". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1957–1964.

[41] H. P. Moravec. "Towards automatic visual obstacle avoidance". In: *5th International Joint Conference on Artificial Intelligence*. 1977.

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

[43] A. Ng and S. Russell. "Algorithms for Inverse Reinforcement Learning". In: *Proceedings of the Seventeenth International Conference on Machine Learning*. 2000, pp. 663–670.

[44] N. Perveen, D. Kumar, and I. Bhardwaj. "An overview on template matching methodologies and its applications". In: *International Journal of Research in Computer and Communication Technology* 2.10 (2013), pp. 988–995.

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

[46] M. Quigley, B. Gerkey, and W. D. Smart. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. O'Reilly Media, 2015.

[47] N. Ratliff, J. A. Bagnell, and M. Zinkevich. "Maximum Margin Planning". In: *Proceedings of the 23rd International Conference on Machine Learning*. 2006, pp. 729–736.

[48] S. Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.6 (2017), pp. 1137–1149.

[49] S. Ross, G. Gordon, and D. Bagnell. "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011, pp. 627–635.

[50] D. Sadigh et al. "Active Preference-Based Learning of Reward Functions". In: *Robotics: Science and System*. 2017.

[51] D. Sadigh et al. "Planning for cars that coordinate with people: leveraging effects on human actions for planning and active information gathering over human internal state". In: *Autonomous Robots* 42.7 (2018), pp. 1405–1426.

[52] A. Saxena, J. Driemeyer, and A. Ng. "Robotic Grasping of Novel Objects using Vision". In: *The International Journal of Robotics Research* 27.2 (2008), pp. 157–173.

[53] D. Scharstein and R. Szeliski. "High-accuracy stereo depth maps using structured light". In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Vol. 1. 2003.

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

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

[56] D. Simon. *Optimal State Estimation: Kalman, H_∞, and Nonlinear Approaches*. John Wiley & Sons, 2006.

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

[58] R. Sutton and A. Barto. *Reinforcement learning: An introduction*. MIT Press, 2018.

[59] R. Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

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

[61] 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.

[62] 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.

[63] K. Yu et al. "More than a million ways to be pushed. A high-fidelity experimental dataset of planar pushing". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 30–37.

[64] M. D. Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks". In: *European Conference on Computer Vision (ECCV)*. Springer, 2014, pp. 818–833.

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

[66] 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.