

Algorithms for Validation

Algorithms for Validation

Mykel J. Kochenderfer

Sydney M. Katz

Anthony L. Corso

Robert J. Moss

Stanford, California

© 2025 Kochenderfer, Katz, Corso, and Moss

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording or information storage and retrieval) without permission in writing from the publisher.

This book was set in \TeX Gyre Pagella by the authors in \LaTeX .

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data is available.

ISBN:

10 9 8 7 6 5 4 3 2 1

To our families.

Contents

<i>Acknowledgments</i>	xiii
<i>Preface</i>	xv
1 <i>Introduction</i>	1
1.1 Validation	1
1.2 History	3
1.3 Societal Consequences	6
1.4 Validation Algorithms	8
1.5 Challenges	14
1.6 Overview	16
2 <i>System Modeling</i>	19
2.1 Model Building	19
2.2 Probability	20
2.3 Parameter Learning	27
2.4 Agent Models	39
2.5 Model Validation	42
2.6 Summary	51
2.7 Exercises	52
3 <i>Property Specification</i>	61
3.1 Properties of Systems	61
3.2 Metrics for Stochastic Systems	62
3.3 Composite Metrics	64
3.4 Logical Specifications	70
3.5 Temporal Logic	73

3.6	Reachability Specifications	81
3.7	Summary	87
3.8	Exercises	87
4	<i>Falsification through Optimization</i>	93
4.1	Direct Sampling	93
4.2	Disturbances	94
4.3	Fuzzing	97
4.4	Falsification through Optimization	100
4.5	Objective Functions	101
4.6	Optimization Algorithms	104
4.7	Summary	107
4.8	Exercises	108
5	<i>Falsification through Planning</i>	111
5.1	Shooting Methods	111
5.2	Tree Search	113
5.3	Heuristic Search	114
5.4	Monte Carlo Tree Search	126
5.5	Reinforcement Learning	130
5.6	Simulator Requirements	131
5.7	Summary	134
5.8	Exercises	134
6	<i>Failure Distribution</i>	139
6.1	Distribution over Failures	139
6.2	Rejection Sampling	140
6.3	Markov Chain Monte Carlo	144
6.4	Probabilistic Programming	152
6.5	Summary	154
6.6	Exercises	155

7	<i>Failure Probability Estimation</i>	159
7.1	Direct Estimation	159
7.2	Importance Sampling	165
7.3	Adaptive Importance Sampling	171
7.4	Sequential Monte Carlo	177
7.5	Ratio of Normalizing Constants	181
7.6	Multilevel Splitting	191
7.7	Summary	194
7.8	Exercises	194
8	<i>Reachability for Linear Systems</i>	203
8.1	Forward Reachability	203
8.2	Set Propagation Techniques	205
8.3	Set Representations	212
8.4	Reducing Computational Cost	216
8.5	Linear Programming	221
8.6	Summary	225
8.7	Exercises	227
9	<i>Reachability for Nonlinear Systems</i>	231
9.1	Interval Arithmetic	231
9.2	Inclusion Functions	233
9.3	Taylor Models	240
9.4	Concrete Reachability	242
9.5	Optimization-Based Nonlinear Reachability	247
9.6	Partitioning	250
9.7	Neural Networks	254
9.8	Summary	257
9.9	Exercises	258
10	<i>Reachability for Discrete Systems</i>	263
10.1	Graph Formulation	263
10.2	Reachable Sets	265
10.3	Satisfiability	267
10.4	Probabilistic Reachability	274
10.5	Discrete State Abstractions	280
10.6	Summary	283
10.7	Exercises	285

11	<i>Explainability</i>	291
11.1	Explanations	291
11.2	Policy Visualization	292
11.3	Feature Importance	293
11.4	Policy Explanation through Surrogate Models	303
11.5	Counterfactual Explanations	309
11.6	Failure Mode Characterization	315
11.7	Summary	319
11.8	Exercises	319
12	<i>Runtime Monitoring</i>	323
12.1	Operational Design Domain Monitoring	323
12.2	Uncertainty Quantification	330
12.3	Failure Monitoring	347
12.4	Summary	351
12.5	Exercises	351

APPENDICES

A	<i>Systems</i>	357
A.1	Default Implementations	357
A.2	Simple Gaussian System	358
A.3	Multivariate Gaussian System	358
A.4	Mass-Spring-Damper System	359
A.5	Inverted Pendulum System	361
A.6	Grid World System	362
A.7	Continuum World System	362
A.8	Aircraft Collision Avoidance System	365
B	<i>Mathematical Concepts</i>	367
B.1	Measure Spaces	367
B.2	Probability Spaces	368
B.3	Metric Spaces	368
B.4	Normed Vector Spaces	368
B.5	Positive Definiteness	370
B.6	Information Content	370
B.7	Entropy	370
B.8	Cross Entropy	371
B.9	Relative Entropy	371
B.10	Taylor Expansion	371

<i>C Neural Representations</i>	375
<i>D Julia</i>	381
D.1 Types	381
D.2 Functions	394
D.3 Control Flow	397
D.4 Packages	399
D.5 Convenience Functions	403
<i>References</i>	405
<i>Index</i>	418

Acknowledgments

We wish to thank the many individuals who have provided valuable feedback on early drafts of our manuscript, including Matthias Althoff, Maxime Bouton, Stephen Boyd, Hugo Buurmeiher, Caroline Cahilly, Emmanuel Candés, François Chabard, Harrison Delecki, Aaron Feldman, Niveditha Iyer, Arec Jangochian, Grace Kim, Hanna Krasowski, Liam Kruse, Daniel Neamati, Shreya Parjan, Prashin Sharma, John Siddiqui, Anna Sulzer, Hazem Torfah, Alexandros Tzikas, Romeo Valentin, Joe Vincent, Jun Wang, and Asta Wu. Many of the algorithms discussed in this book were explored during the development of the ACAS X aircraft collision avoidance systems with the generous support and leadership of Neal Suchy of the Federal Aviation Administration. The participants of Dagstuhl Seminar 24361 provided valuable input to the topics included in the book. It has been a pleasure working with Elizabeth Swayze and the editing team from the MIT Press in preparing this manuscript for publication.

The style of this book was inspired by Edward Tufte. Among other stylistic elements, we adopted his wide margins and use of small multiples. The typesetting of this book is based on the Tufte-LaTeX package by Kevin Godby, Bil Kleb, and Bill Wood. The book’s color scheme was adapted from the Monokai theme by Jon Skinner of Sublime Text (sublimetext.com) and a palette that better accommodates individuals with color blindness.¹ For plots, we use the viridis color map defined by Stéfan van der Walt and Nathaniel Smith.

We have also benefited from the various open-source packages on which this textbook depends (see appendix D). The authors thank Tor Fjelde for his help with `Turing.jl`. The typesetting of the code was done with the help of `pythontex`, which is maintained by Geoffrey Poore. The typeface used for the algorithms is `JuliaMono` (github.com/cormullion/juliamono). The plotting was handled by `pgfplots`, which is maintained by Christian Feuersänger.

¹ B. Wong, “Points of View: Color Blindness,” *Nature Methods*, vol. 8, no. 6, pp. 441–442, 2011.

Preface

This book provides a broad introduction to algorithms for validating safety-critical systems. We cover a wide variety of topics related to validation, introducing the underlying mathematical problem formulations and the algorithms for solving them. Figures, examples, and exercises are provided to convey the intuition behind the various approaches.

This book is intended for advanced undergraduates and graduate students, as well as professionals. It requires some mathematical maturity and assumes prior exposure to multivariable calculus, linear algebra, and probability concepts. Some review material is provided in the appendices. Disciplines where the book would be especially useful include mathematics, statistics, computer science, aerospace, electrical engineering, and operations research.

Fundamental to this textbook are the algorithms, which are all implemented in the Julia programming language. We have found this language to be ideal for specifying algorithms in human-readable form. The priority in the design of the algorithmic implementations was interpretability rather than efficiency. Industrial applications, for example, may benefit from alternative implementations. Permission is granted, free of charge, to use the code snippets associated with this book, subject to the condition that the source of the code is acknowledged.

MYKEL J. KOCHENDERFER

SYDNEY M. KATZ

ANTHONY L. CORSO

ROBERT J. MOSS

Stanford, California

March 13, 2025

1 *Introduction*

Before deploying decision-making systems in high-stakes settings, it is important to ensure that they will operate as intended. We refer to the process of analyzing the behavior of these systems as *validation*. Validation is a critical component of the development process for decision-making systems in a variety of domains including autonomous vehicles, robotics, and healthcare. As these systems and their operating environments increase in complexity, understanding the full spectrum of possible behaviors becomes more challenging and requires a rigorous validation process. This book discusses these challenges and presents a variety of computational methods for validating autonomous systems. This chapter begins with a broad overview of validation. We motivate the need for validation from a historical perspective and outline the societal consequences of validation failures. We then introduce the validation framework that we will use throughout the book. We discuss the challenges associated with validation and conclude with an overview of the remaining chapters in the book.

1.1 *Validation*

The concept of validation is defined differently by different communities, and the word itself is often used in conjunction with other terms such as *verification* and *testing*.¹ In this book, we define validation as the broad process of establishing confidence that a system will behave as desired when deployed in the real world. We define verification as a special type of validation that provides guarantees about the correctness of a system with respect to a specification. We define testing as a technique used for validation that involves evaluating the system on a discrete set of test cases.

¹ For a discussion on these definitions, see section 1.2.3 of A. Engel, *Verification, Validation, and Testing of Engineered Systems*. John Wiley & Sons, 2010, vol. 73.

From a systems engineering perspective, validation is viewed as a phase of the development cycle for autonomous systems (figure 1.1). A typical development cycle begins by defining a set of operational requirements for the system. For example, the developers of an aircraft collision avoidance system may identify a requirement on the probability of collision when deployed in the airspace. Designers then use these requirements to produce an initial version of the system. In the aircraft collision avoidance example, the system may consist of a decision-making agent that selects actions to avoid collisions based on sensor information.

A common technique for designing a system to match a set of desired requirements is to optimize the system with respect to an objective function or reward model that captures the requirements. However, the models used to perform the optimization may be imperfect, the optimization objective may not perfectly capture the requirements, and the optimization process itself is often approximate. This misalignment can result in a mismatch between the desired behavior of the system and its actual behavior when deployed in the real world. We refer to this phenomenon as the *alignment problem*.²

The alignment problem motivates the need for the validation phase of the development cycle. Given the requirements and design, validation algorithms analyze whether the system will behave as intended when deployed in its operating environment. Based on the results of the validation process, developers may need to revise the design or requirements. This process is often repeated multiple times before the system is ready for deployment. It is important to perform validation early in the development cycle to detect bugs and misalignments before they become more costly to fix. For example, repairing a software bug during maintenance is often orders of magnitude more expensive than fixing the bug early in the development cycle.³

This book focuses entirely on the validation phase of the development cycle. We assume that we are given a system that has been designed to meet a set of established requirements, and we discuss methods to translate the system and its requirements to computational models and formal specifications that allow us to apply a variety of validation algorithms. In other words, this book is not about systems engineering or the development of systems.⁴ Instead, we focus on algorithms that validate the behavior of these systems in their operating environments.

Validation techniques have been developed for a wide variety of systems ranging from aircraft parts to medical devices to customer service chatbots. For

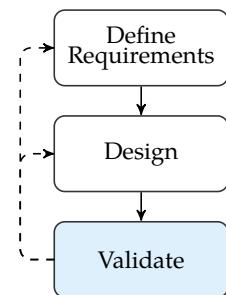


Figure 1.1. A typical development cycle for an autonomous system. This book focuses on the validation phase of development.

² A detailed discussion of the alignment problem is provided in B. Christian, *The Alignment Problem: Machine Learning and Human Values*. W. W. Norton & Company, 2020.

³ C. Baier and J.-P. Katoen, "Principles of Model Checking," in MIT Press, 2008, ch. 1.

⁴ More information about the systems engineering process can be found in A. Kossiakoff, S. M. Biemer, S. J. Seymour, and D. A. Flanigan, *Systems Engineering Principles and Practice*. John Wiley & Sons, 2020. A variety of algorithms for designing decision-making systems are provided in M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray, *Algorithms for Decision Making*. MIT Press, 2022.

example, aircraft designers validate the structural integrity of the wings through extensive stress testing, and medical device manufacturers validate the safety of their devices through clinical trials. In this book, we present an algorithmic perspective on validation and focus specifically on the validation of decision-making agents.

Decision-making agents interact with the environment and make decisions based on the information they receive. These agents range from fully automated systems that operate independently within their environment to decision-support systems that inform human decision-makers.⁵ Examples include aircraft collision avoidance systems, adaptive cruise control systems, hiring assistants, disaster response systems, and other cyberphysical systems.⁶ While the algorithms presented in this book can be applied to many different types of decision-making agents, we place a particular emphasis on sequential decision-making agents, which make a series of decisions over time. For example, an autonomous vehicle must make a sequence of decisions to navigate from one location to another.

1.2 History

The history of validation is deeply intertwined with the evolution of complex systems across many domains. Early forms of validation can be traced back to the ideas of ancient Greek philosophers such as Aristotle (384–322 BC).⁷ Aristotle advocated for a continuous cycle of observation and experimentation to validate hypotheses. The scientific method introduced during the scientific revolution of the 16th and 17th centuries formalized this notion. During this time, Francis Bacon (1561–1626) proposed a method for validating scientific hypotheses through empirical observation and experimentation.

The technological changes brought on by the industrial revolutions accelerated progress in validation. During the First Industrial Revolution of the late 18th and early 19th centuries, the complexity of systems increased dramatically, and the field of validation shifted from validating ideas and hypotheses to validating machines and production processes. The increase of mass production in factories during the Second Industrial Revolution (1870–1914) further motivated the need for validation. Supervisors began to perform quality control checks on products to ensure that they met the desired specifications.⁸ As production volume increased in the following years, supervisors could no longer inspect every product, and factories began to hire designated inspectors for quality control.

⁵ Autonomy and automation have different definitions in different communities. Autonomy is often defined as the automation of high-level tasks such as driving. The algorithms in this book can be applied to decision-making systems with any level of automation or autonomy.

⁶ Cyberphysical systems are computational systems that interact with the physical world.

⁷ W. M. Dickie, “A Comparison of the Scientific Method and Achievement of Aristotle and Bacon,” *The Philosophical Review*, vol. 31, no. 5, pp. 471–494, 1922.

⁸ K. Ishikawa and J. H. Loftus, “Introduction to Quality Control,” in Springer, 1990, vol. 98, ch. 1.

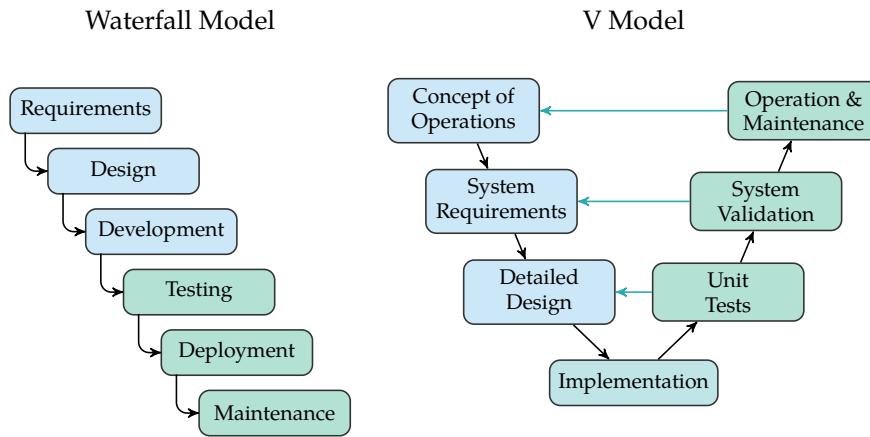


Figure 1.2. Comparison of the waterfall and V models of the software development lifecycle.

During World War II, production volume increased to the point where it was no longer possible to inspect every product. This increase in production output led to the adoption of statistical quality control methods, which relied on sampling to speed up inspection. These ideas were developed by W. Edwards Deming⁹ (1900–1993) and Joseph M. Juran¹⁰ (1904–2008) and marked the beginning of the field of statistical process control. Deming and Juran introduced these ideas to Japanese manufacturers after World War II, which played a key role in the post-war economic recovery of Japan.

The advancements in computing technology in the latter half of the 20th century increased our ability to use statistical methods to validate complex systems. In the late 1940s, scientists at Los Alamos National Laboratory developed the Monte Carlo method, which uses random sampling to solve complex mathematical problems.¹¹ These methods were later used to validate complex systems in a variety of domains such as aviation and finance. Progress in computing technology also led to new challenges in validation. The development of software systems required new validation techniques and best practices to ensure that the software operated correctly.

In the 1970s, software engineers began formalizing the software development life cycle into phases that supported rigorous testing and validation. The *waterfall model* of software development, introduced in 1970, divided the software development process into distinct phases including requirements, design, im-

⁹ W. M. Tsutsui, "W. Edwards Deming and the Origins of Quality Control in Japan," *Journal of Japanese Studies*, vol. 22, no. 2, pp. 295–325, 1996.

¹⁰ D. Phillips-Donaldson, "100 Years of Juran," *Quality Progress*, vol. 37, no. 5, pp. 25–31, 2004.

¹¹ A. F. Bielajew, "History of Monte Carlo," in *Monte Carlo Techniques in Radiation Therapy*, CRC Press, 2021, pp. 3–15.

plementation, testing, and maintenance.¹² In the 1990s, the waterfall model was refined into the *V model*, which emphasizes the importance of testing and validation throughout the software development process.¹³ The V model aligns testing and validation activities with the corresponding development activities, ensuring that the system is validated at each stage of development. Figure 1.2 compares the waterfall and V models of the software development life cycle.

The 20th century also saw the emergence of regulatory bodies to guide the safe development of new technologies. The Food and Drug Administration (FDA) was established in the United States in 1906 after a series of food and drug safety incidents.¹⁴ In 1947, the International Organization for Standardization (ISO) was founded to develop international standards for products and services.¹⁵ After a series of midair collisions between aircraft, the Federal Aviation Administration (FAA) was formed in 1958 to regulate civil aviation in the United States.¹⁶

As technology matured in the late 20th and early 21st centuries, these regulatory bodies introduced new standards and requirements. For example, the Radio Technical Commission for Aeronautics (RTCA) introduced the DO-178 standard in 1982 to provide guidelines for the development of safety-critical software in aviation. DO-178 has been updated multiple times in the following years to account for new technological advancements in the field and has been used frequently by the FAA to certify the safety of aircraft software.¹⁷ In 2011, ISO 26262 was introduced as an international standard relating to the functional safety of automotive systems. While ISO 26262 was developed specifically for electronic/electric systems in road vehicles, many researchers have used it as a guideline for the development of both hardware and software for autonomous vehicles.¹⁸

Starting in the 2010s, artificial intelligence (AI) and machine learning systems became increasingly prevalent in a variety of applications. For example, AI systems were introduced into autonomous vehicles, aircraft, medical diagnosis, and financial trading. The increased capabilities and applications of AI led to new validation challenges and techniques. Not only are the systems themselves complex, but they also operate in complex environments, making validation of these systems particularly challenging. In 2020, the European Union Aviation Safety Agency (EASA) published initial guidelines related to the design assurance of neural networks, which are a key component of many machine learning systems.¹⁹ In that document, they outline a modification of the traditional V

¹² W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," *IEEE WESCON*, 1970.

¹³ K. Forsberg and H. Mooz, "The Relationship of System Engineering to the Project Cycle," *Center for Systems Management*, vol. 5333, 1991.

¹⁴ A. T. Borchers, F. Hagie, C. L. Keen, and M. E. Gershwin, "The History and Contemporary Challenges of the US Food and Drug Administration," *Clinical Therapeutics*, vol. 29, no. 1, pp. 1–16, 2007.

¹⁵ C. N. Murphy and J. Yates, *The International Organization for Standardization (ISO): Global Governance Through Voluntary Consensus*. Routledge, 2009.

¹⁶ J. W. Gelder, "Air Law: The Federal Aviation Act of 1958," *Michigan Law Review*, vol. 57, no. 8, pp. 1214–1227, 1959.

¹⁷ More information on the history of software standards in aviation can be found in L. Rierson, *Developing Safety-Critical Software: a Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2017.

¹⁸ M. A. Gosavi, B. B. Rhoades, and J. M. Conrad, "Application of Functional Safety in Autonomous Vehicles Using ISO 26262 Standard: A Survey," in *SoutheastCon*, 2018.

¹⁹ EASA AI Task Force, "Concepts of Design Assurance for Neural Networks," *EASA*, 2020.

model to account for validation of the learning process. In general, the validation of AI systems is still an active area of research.

1.3 *Societal Consequences*

The validation of decision-making agents is critical in ensuring that these systems are properly integrated into society. Failures in validation can have severe societal consequences. This section discusses the impacts of validation on various aspects of society.

1.3.1 *Safety*

Validation is necessary for ensuring the safety of systems that interact with the physical world. Failures of safety-critical systems can result in catastrophic accidents that cause injury or loss of life. For example, unintended behavior of the safety-critical software used by the Therac-25 radiation therapy machine caused radiation overdoses that resulted in death or serious injury to six patients.²⁰ Safety is also important for transportation systems such as aircraft and cars. In 2002, a mid-air collision over Überlingen, Germany resulted in 71 fatalities when the traffic alert and collision avoidance system (TCAS) and air traffic control (ATC) systems issued conflicting instructions to the pilots.²¹ Furthermore, it is important to ensure that autonomous vehicles make safe decisions in a wide range of scenarios to prevent potential accidents. Since their introduction, autonomous vehicles have been involved in accidents that have resulted in injuries or fatalities.²²

²⁰ N. G. Leveson and C. S. Turner, "An Investigation of the Therac-25 Accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

²¹ J. Kuchar and A. C. Drumm, "The Traffic Alert and Collision Avoidance System," *Lincoln Laboratory Journal*, vol. 16, no. 2, p. 277, 2007.

²² R. L. McCarthy, "Autonomous Vehicle Accident Data Analysis: California OL 316 Reports: 2015–2020," *ASCE-ASME Journal of Risk and Uncertainty in Engineering Systems, Part B: Mechanical Engineering*, vol. 8, no. 3, p. 034502, 2022.

1.3.2 *Fairness*

When agents make decisions that affect the lives of large groups of people, we must ensure that their decisions are fair and unbiased. Validation helps researchers and organizations identify and correct biases in decision-making systems before deployment. If these biases are not addressed, they can have serious consequences for individuals and society as a whole. For example, an automated hiring system developed by Amazon was ultimately discontinued after it was found to be biased against women due to biases in the historical data it was trained on.²³ In another case, a software system designed to predict recidivism rates in criminal defendants called COMPAS was found to be biased toward certain demographics

²³ A. L. Hunkenschroer and A. Kriebitz, "Is AI Recruiting (Un)ethical? A Human Rights Perspective on the Use of AI for Hiring," *AI and Ethics*, vol. 3, no. 1, pp. 199–213, 2023.

based on empirical data.²⁴ Using the outputs of these systems to make decisions can result in the unfair treatment of individuals. Validating these systems before deployment can help prevent this type of failure.

1.3.3 Public Trust

Public trust in autonomous systems is critical for their widespread adoption, and validation plays a key role in developing this trust. For example, trust has been identified as a key factor in the eventual adoption of autonomous vehicles into society.²⁵ For this reason, autonomous vehicle designers and manufacturers have invested heavily in validation to ensure that their vehicles are safe and reliable. The aviation industry is another example of an industry that relies on public trust. The industry has maintained public trust by upholding a rigorous safety process that has resulted in a strong safety record. However, failures in validation can erode public trust. For instance, when the Boeing 737 MAX 8 aircraft was grounded worldwide after two fatal crashes, public trust in the aviation industry was significantly impacted.²⁶ Validation also allows us to anticipate possible ethical dilemmas before deployment.²⁷ Addressing these dilemmas is crucial to maintaining trust.

1.3.4 Economics

Systems that operate expensive equipment or control finances require validation to decrease the risk of significant economic loss. In 1996, the maiden voyage of the Ariane 5 rocket ended in an explosion that could ultimately be traced back to a software bug caused by overflow when converting from a 64-bit to 16-bit value.²⁸ The failure resulted in loss of the rocket and the research satellites it was carrying to space for a total of \$370 million in damages. Furthermore, failures in financial decision-making systems can affect entire economic systems. For example, the failure of the Long-Term Capital Management (LTCM) hedge fund in 1998 nearly caused a global financial crisis and required a \$3.6 billion bailout. The fund used a trading strategy that failed to account for extreme events.²⁹ When these events occurred, LTCM suffered massive losses.

²⁴ Other research has argued that the system is fair under a different definition of fairness. A detailed discussion is provided in J. Kleinberg, S. Mullaithan, and M. Raghavan, "Inherent Trade-Offs in the Fair Determination of Risk Scores," in *Innovations in Theoretical Computer Science (ITCS) Conference*, 2017.

²⁵ J. K. Choi and Y.G. Ji, "Investigating the Importance of Trust on Adopting an Autonomous Vehicle," *International Journal of Human-Computer Interaction*, vol. 31, no. 10, pp. 692–702, 2015.

²⁶ J. Herkert, J. Borenstein, and K. Miller, "The Boeing 737 MAX: Lessons for Engineering Ethics," *Science and Engineering Ethics*, vol. 26, pp. 2957–2974, 2020.

²⁷ An example of an ethical analysis for autonomous vehicles can be found in J. Siegel and G. Pappas, "Morals, Ethics, and the Technology Capabilities and Limitations of Automated and Self-Driving Vehicles," *AI & Society*, vol. 38, no. 1, pp. 213–226, 2023.

²⁸ M. Dowson, "The Ariane 5 Software Failure," *Software Engineering Notes*, vol. 22, no. 2, p. 84, 1997.

²⁹ P. Jorion, "Risk Management Lessons from Long-Term Capital Management," *European Financial Management*, vol. 6, no. 3, pp. 277–300, 2000.

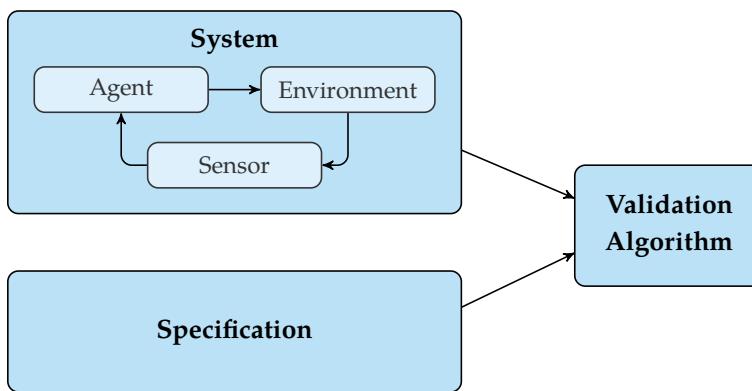


Figure 1.3. Validation algorithms check whether a given system satisfies a specification. The system consists of an agent operating in an environment, which it perceives using a sensor or set of sensors.

1.4 Validation Algorithms

Validation algorithms require two inputs, as shown in figure 1.3. The first input is the system under test, which we will refer to as the *system*. The system represents a decision-making agent operating in an environment. The agent makes decisions based on information from the environment that it receives from sensors.³⁰ The second input is a *specification*, which expresses an operating requirement for the system. Specifications often pertain to safety, but they may also address other key design objectives. Given these inputs, validation algorithms output metrics to help us understand the scenarios in which the system does or does not satisfy the specification. The rest of this section provides a high-level overview of these inputs and outputs.

1.4.1 System

A system (algorithm 1.1) consists of three main components: an environment, an agent, and a sensor. The *environment* represents the world in which the agent operates. We refer to an agent's configuration within its environment as its state s . The state space \mathcal{S} represents the set of all possible states. An environment consists of an initial state distribution and a transition model. When the agent takes an action, the state evolves probabilistically according to the transition model. The transition model $T(s' | s, a)$ denotes the probability of transitioning to state s' from state s when the agent takes action a .

³⁰ Up to this point, we have informally used the term *system* to refer to only the agent and its sensors. For the remainder of the book, we will also include the operating environment as part of the system.

```

abstract type Agent end
abstract type Environment end
abstract type Sensor end

struct System
    agent::Agent
    env::Environment
    sensor::Sensor
end

```

Algorithm 1.1. A system consists of an agent, its operating environment, and the sensor or set of sensors that it uses to perceive its environment.

For physical systems, the state often represents an agent's position and velocity in the environment, and the transition model is typically governed by the agent's equations of motion. Figure 1.4 shows an example of a state for an inverted pendulum system. The state and transition model may also contain information about other agents in the environment. For example, the environment for an aircraft collision avoidance system contains the other aircraft in the airspace that the agent must avoid. The other agents may also be human agents such as other drivers or pedestrians in the environment of an autonomous vehicle. The presence of other agents in the environment often increases our uncertainty in the outcome of a particular action.

In many real-world systems, agents do not have access to their true state within the environment and instead rely on observations from sensors. We define the *sensor* component of a system as a mechanism for sensing information about the environment. Many real-world systems rely on multiple sensors, so the sensor component may contain multiple sensing modalities. For example, an autonomous vehicle senses its position in the world using a combination of sensors such as global positioning systems (GPS), cameras, and LiDAR. We model the sensor component using an observation model $O(o | s)$, which represents the probability of producing observation o in state s . Observations come in multiple forms based on the sensing modality. For example, GPS sensors output coordinates, while camera sensors output image data. We call the set of all possible observations for a system its observation space \mathcal{O} .

An *agent* uses observations to select actions from a set of possible actions known as the action space \mathcal{A} . Agents may use a number of decision-making algorithms or frameworks to select actions. While some agents select actions based entirely on the observation, other agents use the observation to first estimate the state and then select an action based on this estimate. Furthermore, some

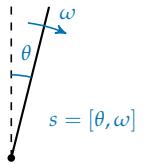


Figure 1.4. The state s of an inverted pendulum system can be compactly represented as its current angle from the vertical θ and its angular velocity ω .

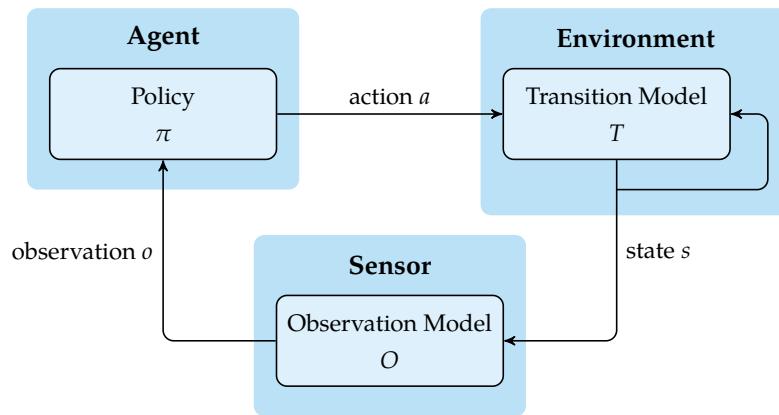


Figure 1.5. A system consists of an agent with policy π , an environment governed by transition model T , and a sensor with observation model O .

agents may keep track of previous actions and observations internally to improve their state estimate. For example, an aircraft that only observes its altitude may keep track of previous altitude measurements to estimate its climb or descent rate. We abstract these behaviors of the agent using the notion of a policy π , which is responsible for selecting an action given the current observation and information the agent has stored previously. An agent's policy can be stochastic or deterministic. A stochastic policy samples actions according to a probability distribution, while a deterministic policy will always produce the same action given the same information.

The transition model $T(s' | s, a)$ satisfies the *Markov assumption*, which requires that the next state depend only on the current state and action. The state space, action space, observation space, observation model, and transition model are all elements of a sequential decision-making framework known as a *partially observable Markov decision process* (POMDP).³¹ Figure 1.5 demonstrates how these elements fit into the components of a system. Appendix A provides implementations of these components for the example systems discussed in this book.

We analyze the behavior of a system over time by considering the sequence of states, observations, and actions that the agent experiences. This sequence is known as a *trajectory*. We generate trajectories by performing a *rollout* of the system (algorithm 1.2). A rollout begins by sampling an initial state from the initial state distribution associated with the environment. At each time step, the sensor produces an observation based on the current state, the agent selects an

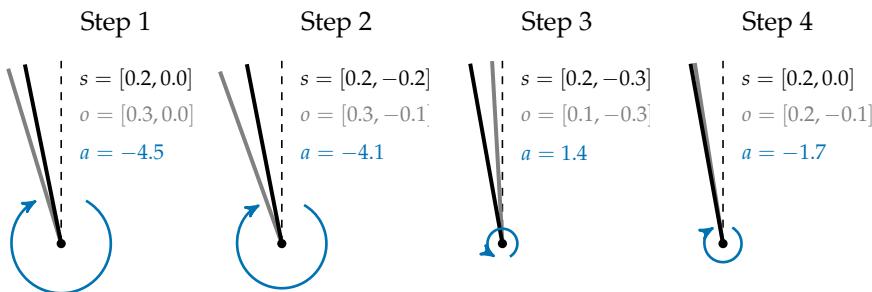
³¹ M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray, *Algorithms for Decision Making*. MIT Press, 2022.

```

function step(sys::System, s)
    o = sys.sensor(s)
    a = sys.agent(o)
    s' = sys.env(s, a)
    return (; o, a, s')
end

function rollout(sys::System; d)
    s = rand(Ps(sys.env))
    τ = []
    for t in 1:d
        o, a, s' = step(sys, s)
        push!(τ, (; s, o, a))
        s = s'
    end
    return τ
end

```



Algorithm 1.2. A function that performs a rollout of a system `sys` to a depth `d` and returns the resulting trajectory `τ`. It samples an initial state from the initial state distribution associated with the environment. It then repeatedly calls the `step` function, which steps the system forward in time. The `step` function takes in the current state `s`, produces an observation `o` from the sensor, gets the action `a` from the agent based on this observation, and determines the next state `s'` from the environment.

Figure 1.6. Example trajectory of depth $d = 4$ for the inverted pendulum system. At each time step, the sensor produces a noisy observation of the true state, and the agent tries to keep the pendulum upright by selecting a torque to apply at the base of the pendulum.

action based on the observation, and the environment transitions to a new state based on the action. We repeat this process to a desired depth d to generate a trajectory $τ = (s_1, o_1, a_1, \dots, s_d, o_d, a_d)$ where $s_{i+1} \sim T(\cdot | s_i, a_i)$, $o_i \sim O(\cdot | s_i)$, and $a_i \sim π(\cdot | o_i)$. Figure 1.6 shows an example trajectory for the inverted pendulum system.

1.4.2 Specification

A *specification* ψ is a formal expression of a requirement that the system must satisfy when deployed in the real world. These requirements may be derived from domain knowledge or other systems engineering principles. Some industries

have regulatory agencies that govern requirements. These agencies are especially common in safety-critical industries. For example, the FAA and the FDA in the United States provide regulations and requirements for aircraft and healthcare systems, respectively.

We express specifications by translating operating requirements to logical formulas that can be evaluated on trajectories.³² For example, the specification for an aircraft collision avoidance system is that the agent should not collide with other aircraft in the airspace. Given a trajectory, we want to check whether any of the states in the trajectory represent a collision.

Algorithm 1.3 defines a general framework for specifications that we will use throughout this book. Evaluating a specification on a trajectory results in a Boolean value that indicates whether the specification is satisfied. We consider a trajectory to be a failure if the specification is not satisfied. Example 1.1 demonstrates this idea on a simple grid world system. We can also derive higher-level metrics from specifications such as the probability of failure or the expected cost of failure.

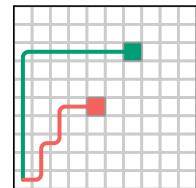
```
abstract type Specification end
function evaluate( $\psi$ ::Specification,  $\tau$ ) end
isfailure( $\psi$ ::Specification,  $\tau$ ) = !evaluate( $\psi$ ,  $\tau$ )
```

In the grid world example shown on the right, the agent's goal is to navigate to the green goal state while avoiding the red obstacle state. Therefore, given a trajectory, the specification ψ will be satisfied if the trajectory contains the goal state and does not contain the obstacle state. The green trajectory in the figure satisfies the specification, while the red trajectory represents a failure. Chapter 3 will discuss how to express this specification as a logical formula.

³² Chapter 3 discusses this process in detail.

Algorithm 1.3. Definition of a specification. We evaluate specifications on trajectories. We consider a trajectory to be a failure if the specification is not satisfied.

Example 1.1. Example trajectories evaluated against a specification for the grid world system.



1.4.3 Algorithm Outputs

Validation algorithms provide a variety of outputs that help us understand the behavior of a system. These outputs can be used to make decisions about the system's design, requirements, and deployment. Different validation algorithms are designed to output different metrics. The algorithms presented in this book support the following categories of analysis:

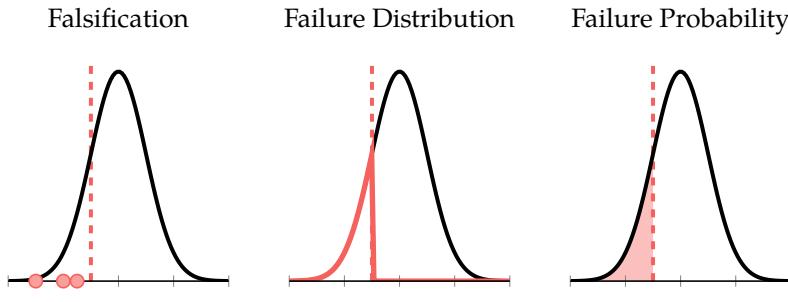


Figure 1.7. Failure analysis outputs for a simple system where failures occur to the left of the dashed red line with likelihood represented by the height of the black curve. The plot on the left shows a set of failure samples that could be identified through falsification. The plot in the middle highlights the shape of the failure distribution, and the shaded region in the plot on the right corresponds to the probability of failure.

- *Failure analysis:* Common types of failure analysis include falsification, failure distribution estimation, and failure probability estimation (figure 1.7). *Falsification* involves searching for possible scenarios that result in a failure. Some falsification algorithms also use a probabilistic model of the system to search for the most likely failure scenario. Other algorithms use this model to draw samples from the full distribution over failures or to estimate the probability of failure. We can use the results of failure analysis to inform future design decisions. Depending on the type and severity of the failure modes, system designers may enhance the system’s sensors, change the agent’s policy, revise the system’s requirements, adapt the training of human operators, or bring in other mitigations. Designers may also simply recognize the failure modes as limitations and move on or use them as grounds to abandon the project altogether. Furthermore, an estimate of the probability of failure can be used to make decisions about the system’s deployment. For example, the FAA places requirements on the probability of failure for aircraft systems before they can be deployed in the airspace.³³
- *Formal guarantees:* Some algorithms output formal guarantees, or proofs, that a system satisfies a specification. One common type of formal guarantee is a reachability guarantee, in which we determine the set of states that a system could reach over time. The result can be used to prove that a system will never enter a dangerous state. For example, we could prove that an aircraft collision avoidance system will never reach a collision state. Formal guarantees are always based on a set of assumptions such as the set of possible initial states. If the assumptions are violated, the guarantees may no longer hold.

³³ These requirements are based on the type and severity of the failure. More information can be found in T. L. Arel, *Safety Management System Manual*, Air Traffic Organization, Federal Aviation Administration, 2022.

- *Explanations:* The ability to explain the behavior of a system helps us build confidence that it is operating as intended. Explanations can take many forms. We may want to explain why an agent made a decision at a particular instance in time or identify the root cause of a failure trajectory we found through falsification. We can use explanations during design to debug the system, identify potential failure modes, and suggest possible improvements. Explanations can also be used to build trust with stakeholders and regulatory bodies.
- *Runtime assurances:* The validation metrics we compute before deploying a system are typically based on a set of assumptions about its operating environment. If the operating environment changes during deployment, these metrics may no longer be valid. Runtime monitoring algorithms check whether these assumptions are being violated during operation and provide assurances that the system is operating as intended. We can use runtime monitoring to detect when the system deviates from its intended behavior and provide alerts to operators.

In most real-world settings, we cannot guarantee that a system will behave as intended using a single validation algorithm or metric. Instead, we use a combination of these techniques to build a *safety case*. This idea is inspired by the *Swiss cheese model* of accident causation (figure 1.8).³⁴ This model views validation algorithms as slices of Swiss cheese³⁵ with holes, or limitations, that may cause us to miss potential failure modes. If we stack enough slices of Swiss cheese together, the holes in one slice will be covered by the cheese in another slice. By using a combination of validation algorithms, we increase our chances of catching potential failure modes before they could occur during operation.

³⁴ J. Reason, "Human Error: Models and Management," *British Medical Journal*, vol. 320, no. 7237, pp. 768–770, 2000.

³⁵ Swiss cheese is a type of cheese that is known for having holes in its slices.

1.5 Challenges

Validating that a decision-making agent will behave as intended when deployed in the real world is a challenging problem. Several factors contribute to this difficulty:

- *Complexity of the agent:* It can be difficult to predict how a decision-making agent will behave in all possible scenarios. For example, the autonomy stack of a self-driving car contains multiple components that interact with one another in complex ways. This complexity makes it challenging to understand how the system will react to different inputs such as sensor data, maps, and traffic laws.

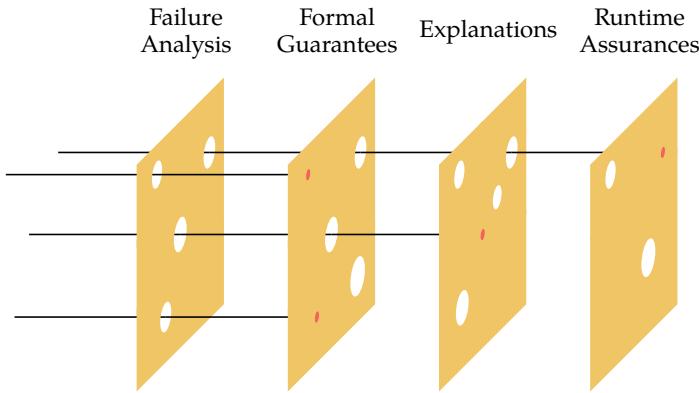


Figure 1.8. The swiss cheese model for safety validation. Each layer represents a different validation algorithm. The holes in each layer represent the limitations of the validation algorithm. By stacking the layers together, we prevent potential failures from getting through to deployment.

Furthermore, it is especially difficult to predict the behavior of decision-making agents that use machine learning models such as neural networks. These models are often difficult to interpret and can exhibit unexpected behaviors.

- *Complexity of the environment:* As the capabilities of autonomous agents increase, they are deployed in increasingly complex environments. For example, self-driving cars must navigate through environments with pedestrians, traffic signs, construction, and other vehicles. To validate these agents, we must be able to properly model this complexity. Another challenge arises when agents use complex sensors to perceive their environment. For example, for systems that use camera sensors, we need to understand the set of images the camera could produce from the environment.
- *Cost and safety:* Testing systems in the real world is expensive and can lead to safety issues. For example, testing an aircraft collision avoidance system involves operating aircraft in close proximity with one another for long periods of time. For this reason, we often rely on simulation to test systems before deploying them in the real world. We must be careful to ensure that the simulated system accurately models the real-world system. However, capturing the full complexity of the real world in simulation can result in simulators that are computationally expensive to run.
- *Edge cases:* Systems designed for safety-critical applications tend to behave safely in the vast majority of scenarios. However, rare edge cases can lead to

catastrophic failures. Because these edge cases occur infrequently, they are often difficult to identify.

1.6 Overview

This section outlines the remaining chapters of the book, which can be organized into several categories:

- *Problem formulation:* Chapters 2 and 3 discuss techniques to formulate validation problems. Specifically, chapter 2 relates to the system, which is the first input to validation algorithms. We discuss how to build computational models of each system component using data and domain knowledge. The accuracy of the validation process depends on the accuracy of these models. Therefore, we also discuss techniques to validate the accuracy of these models. Chapter 3 addresses the specification, which is the second input to validation algorithms. In this chapter, we discuss techniques to translate operating requirements for systems to formal specifications on their behavior.
- *Sampling-based methods:* Chapters 4 to 7 discuss methods that use trajectory samples from a system to analyze its behavior. Since it is often impossible to sample all possible behaviors of a system, these techniques typically focus on failure analysis rather than formal guarantees. Chapters 4 and 5 discuss efficient techniques to search for possible failures of a system using optimization and planning algorithms respectively. Chapter 6 outlines a set of techniques to draw samples from the full distribution over failures for a system, and chapter 7 discusses efficient techniques to estimate the probability of failure from samples.
- *Formal methods:* Chapters 8 to 10 discuss formal methods that provide guarantees on the behavior of a system. These methods can be used to systematically search for failures of a system or to prove the absence of failures if there are none. Chapter 8 discusses reachability techniques that compute the set of states that a system could reach over time. We can use the results of this analysis to determine whether a system reaches any states that violate the specification. Chapter 9 extends these techniques to systems with nonlinear models. In chapter 10, we perform reachability analysis on discrete systems.

- *Runtime monitoring and explainability:* Chapters 11 and 12 discuss techniques to explain a system's decisions and monitor its behavior. Chapter 11 outlines a set of methods that can be used to explain the behavior of a system to its operators and other stakeholders. In chapter 12, we discuss a form of online validation called runtime monitoring, which checks whether a system is operating as intended during deployment.

2 *System Modeling*

Applying validation algorithms directly to real-world systems is often cost prohibitive, unsafe, and impractical due to the constraints of the physical world. To address the limitations of real-world testing, we build models of the system components and perform validation on these models. This chapter begins by discussing probability distributions, which are useful in modeling system components that involve uncertain outcomes. We introduce a number of model classes and discuss how they can be used to represent system components. We then discuss methods for selecting the parameters of these model classes based on observed data or domain knowledge. We also discuss techniques to construct models of other agents in the environment that may be interacting with our system. Because the validity of any analysis that uses these models depends on their accuracy, it is important to assess whether the model adequately captures the behavior of the real world system. We conclude by discussing techniques for validating the performance of a model.

2.1 *Model Building*

As outlined in section 1.4.1, a system can be described by its environment model $T(s' | s, a)$, agent model $\pi(a | o)$, and observation model $O(o | s)$. Building these models requires the following three steps:

1. *Select a model class.* A *model class* is a set of mathematical models defined by a set of parameters.
2. *Select the parameters for the model class.* This process involves selecting the parameters that best represent the system based on available data or expert knowledge.

3. *Validate the model.* Once selected, the model should be validated to ensure that it accurately represents the system.

In this chapter, we will discuss the different model classes that can be used to represent the system components and the methods for selecting the parameters of these models.

There are a variety of challenges when building models. We want to select a model class that is expressive enough to capture the true system, which requires capturing all possible scenarios the system may encounter. For example, a model of an aircraft collision avoidance system must account for all possible pilot and intruder behaviors. However, complex models can be difficult to use for validation. Therefore, we want to ensure that we select the simplest model class that can accurately represent the behavior of the system.¹ Additionally, building models requires data and expert knowledge, which may require significant effort to produce. A final challenge is selecting the objective and optimization technique used to determine the best model parameters. Given these challenges, it is important that we carefully validate the performance of the final model.

2.2 Probability

Many systems have components with multiple possible outcomes and uncertainty over which outcome will occur. To build mathematical models that account for this uncertainty, we use the concept of *probability*.² The probability of a particular outcome is a number between 0 and 1 that quantifies the likelihood of that outcome occurring, relative to all possible outcomes. If one outcome is more likely to occur than another, it has a higher probability.

2.2.1 Probability Distributions

A *probability distribution* is a function that assigns probabilities to different outcomes. Probability distributions are represented differently depending on whether the outcomes are discrete or continuous. Distributions over discrete outcomes are represented by *probability mass functions*. The probability mass function $P(x)$ for a discrete variable X assigns a probability to each possible value of X . To be a valid probability mass function, the probabilities across all outcomes must sum

¹ This idea is captured in a quote from British statistician George E. P. Box (1919–2013), which states that “all models are wrong, but some are useful.” G. E. Box, “Science and Statistics,” *Journal of the American Statistical Association*, vol. 71, no. 356, pp. 791–799, 1976.

² A detailed overview of probability theory is provided by E. T. Jaynes, *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.

to 1 such that

$$\sum_x P(x) = 1 \quad (2.1)$$

where $0 \leq P(x) \leq 1$ for all x . Figure 2.1 shows an example of a probability mass function for a discrete distribution.

Many distributions over continuous outcomes are naturally represented using *probability density functions*. For many continuous distributions, the probability that a variable takes on a particular value is infinitesimally small. Therefore, unlike probability mass functions, probability density functions do not assign probabilities to individual outcomes. Instead, they assign probabilities to intervals of possible outcomes. The probability that the value of a continuous variable X falls between the values a and b is given by the integral of the probability density function $p(x)$ over that interval:

$$P(a \leq x \leq b) = \int_a^b p(x) dx \quad (2.2)$$

Figure 2.2 shows an example of this process. To be a valid probability density function, the integral of the probability density function over all possible outcomes must integrate to 1 such that

$$\int_{-\infty}^{\infty} p(x) dx = 1 \quad (2.3)$$

where $p(x) \geq 0$ for all x . The *support* of a continuous distribution is the set of all values x for which $p(x) > 0$.

Probability distributions are a common type of model class because they are often represented using probability mass or density functions that are determined by a set of parameters θ . For example, the probability density function of a common distribution called the *Gaussian distribution* (also known as the *normal distribution*) is parameterized by its mean μ and variance σ^2 such that $\theta = [\mu, \sigma^2]$ (example 2.1). For a discrete distribution, the parameters θ typically correspond to the probability mass associated with each possible outcome. In general, we will use $P_\theta(x)$ and $p_\theta(x)$ to denote the probability mass or probability density function of a distribution with parameters θ .

We can form more complex distributions by mixing together simpler distributions. Distributions formed in this way are known as *mixture models*. Many common distributions such as the Gaussian distribution are *unimodal*, meaning that they have a single peak. We can represent complex *multimodal* distributions

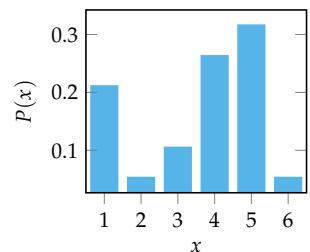


Figure 2.1. A probability mass function for a distribution over a variable X that can take on a value between 1 and 6.

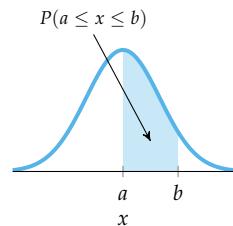
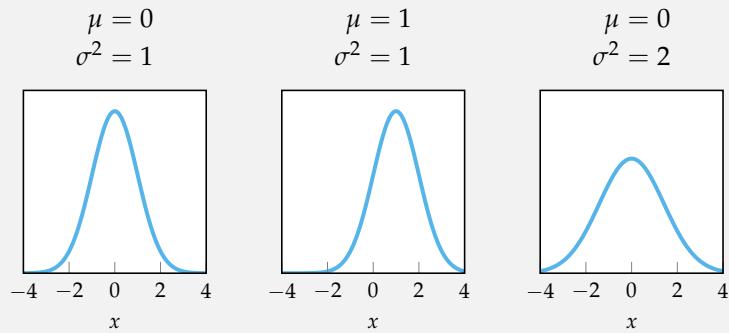


Figure 2.2. A probability density function for a continuous distribution over a variable x . We can find the probability that x falls between two values a and b by integrating the probability density function over that interval.

One common distribution used to describe continuous variables is the *Gaussian distribution* (also called the *normal distribution*) $\mathcal{N}(\mu, \sigma^2)$. A Gaussian distribution is parameterized by its mean μ and its variance σ^2 . The probability density function for a Gaussian distribution is given by

$$\mathcal{N}(x | \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right) \quad (2.4)$$

where $\mathcal{N}(x | \mu, \sigma^2)$ represents the probability density function evaluated at x given a mean μ and variance σ^2 . The mean controls the location of the center of the distribution, while the variance controls the spread of the distribution. The plots below show examples of Gaussian distributions with different means and variances.



Example 2.1. The Gaussian distribution for modeling continuous variables. The mean of a Gaussian distribution controls the location of the center of the distribution, while the variance controls the spread of the distribution.

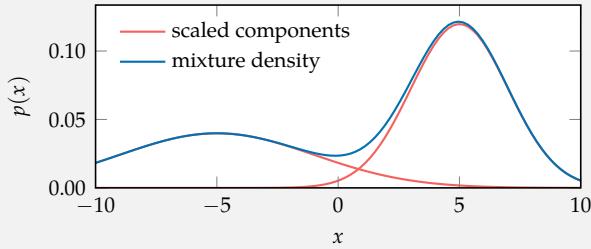
as mixtures of unimodal distributions. For example, a *Gaussian mixture model* is a mixture model that represents a distribution as a combination of multiple Gaussian distributions (example 2.2).

A Gaussian mixture model is a mixture model that is simply a weighted average of various Gaussian distributions. The parameters of a Gaussian mixture model include the parameters of the Gaussian distribution components $\mu_{1:n}, \sigma_{1:n}^2$, as well as their weights $\rho_{1:n}$. The density is given by

$$p(x | \mu_{1:n}, \sigma_{1:n}^2, \rho_{1:n}) = \sum_{i=1}^n \rho_i \mathcal{N}(x | \mu_i, \sigma_i^2) \quad (2.5)$$

where the weights must sum to 1.

We can create a Gaussian mixture model with components $\mu_1 = 5, \sigma_1 = 2$ and $\mu_2 = -5, \sigma_2 = 4$, weighted according to $\rho_1 = 0.6$ and $\rho_2 = 0.4$. The plot below shows the density of two components scaled by their weights.



Example 2.2. An example of a Gaussian mixture model.

We can also represent complex distributions as transformations of simpler distributions. Suppose we have a variable Z that is distributed according to a simple, unimodal distribution p_Z . We can transform Z into a more complex distribution X by applying a transformation f such that $X = f(Z)$. If f is invertible and differentiable, the distribution over X is

$$p_X(x) = p_Z(g(x))|g'(x)| \quad (2.6)$$

where g is the inverse of f . Multiplying the original density by the absolute value of the derivative of g corrects for the stretching or shrinking of the distribution that occurs when transforming the variable. Figure 2.3 transforms a Gaussian distribution into a multimodal distribution. *Normalizing flows* are a class of models

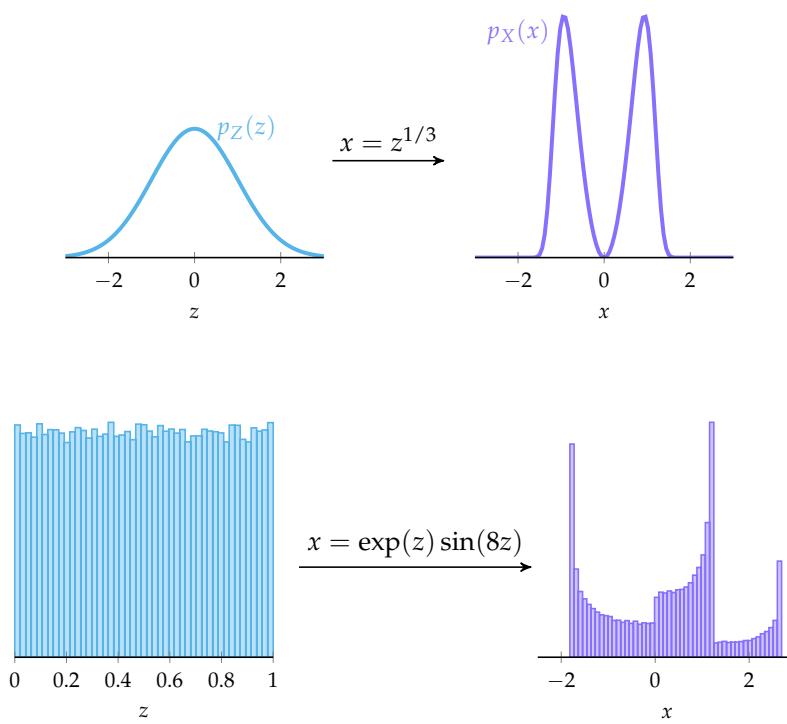


Figure 2.3. Transforming a Gaussian distribution $p_Z(z)$ into a multimodal distribution $p_X(x)$ by applying an invertible and differentiable transformation.

that use this idea to transform simple distributions into complex distributions by applying a series of invertible transformations.³

For some problems, we may not be able to produce an analytical form for the probability density function of the distribution. However, we can still generate samples from the distribution by applying transformations to samples from a *pseudorandom number generator*.⁴ We refer to models represented in this way as *generative models*. *Generative adversarial networks* (GANs) are an example of a generative model that learns to generate samples from complex distributions by transforming samples from a simple distribution into samples that resemble the complex distribution.⁵

Figure 2.4. Examples of a generative model that transforms samples from calls to a pseudorandom number generator that produces samples uniformly between 0 and 1 to samples from a complex distribution.

³ A comprehensive introduction to normalizing flows is provided in I. Kobyzev, S. J. Prince, and M. A. Brubaker, “Normalizing Flows: An Introduction and Review of Current Methods,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 11, pp. 3964–3979, 2020.

⁴ Pseudorandom number sequences, such as those produced by a sequence of calls to `rand`, are deterministic given a particular seed but appear random.

⁵ I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative Adversarial Nets,” *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 27, 2014.

2.2.2 Joint Distributions

A *joint distribution* is a probability distribution over multiple variables. A distribution over a single variable is called a *univariate distribution*, and a joint distribution over multiple variables is called a *multivariate distribution*. Joint distributions represent the likelihood of multiple outcomes occurring simultaneously. For example, the joint distribution over two discrete variables X and Y is represented by the probability mass function $P(x, y)$, which outputs the probability that both $X = x$ and $Y = y$.

We use different strategies to represent joint distributions depending on whether the variables are discrete or continuous. For discrete variables, we can represent the joint distribution as a table such as the one shown in table 2.1. The table assigns a probability to each possible combination of outcomes. These probabilities represent the parameters of the distribution.

We often want to represent joint distributions over many variables with many possible outcomes, which can require a large number of parameters. If we make additional assumptions about the structure of the joint distribution such as independence between variables, we can use other representations such as decision trees or Bayesian networks to reduce the number of parameters required to represent the distribution.⁶ We can represent continuous joint distributions using multivariable functions. For example, a common distribution used to model uncertainty in multiple continuous variables is the *multivariate Gaussian distribution* (example 2.3).

2.2.3 Conditional Distributions

A *conditional distribution* is a distribution over a variable given the value of one or more other variables. The definition of conditional probability states that

$$P(y | x) = \frac{P(y, x)}{P(x)} \quad (2.8)$$

where $P(y | x)$ is read as “probability of y given x ” and represents the probability that the variable Y takes on the value y given that the variable X takes on the value x . The agent, environment, and observation models introduced in section 1.4.1 are all conditional distributions. For example, the transition model $T(s' | s, a)$ is a conditional distribution over the next state s' given the current state s and action a .

X	Y	Z	$P(X, Y, Z)$
0	0	0	0.08
0	0	1	0.31
0	1	0	0.09
0	1	1	0.37
1	0	0	0.01
1	0	1	0.05
1	1	0	0.02
1	1	1	0.07

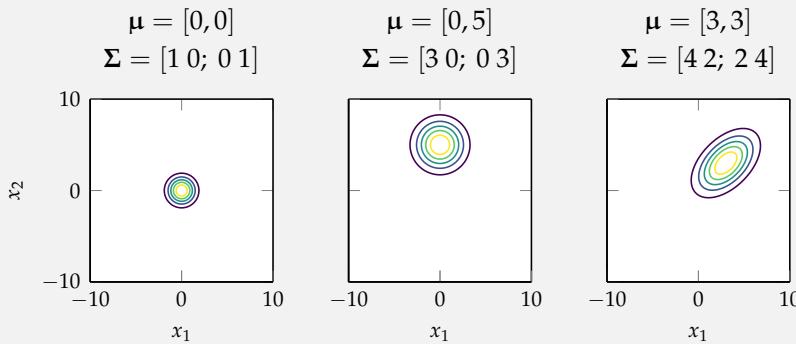
Table 2.1. Example of a joint distribution involving binary variables X , Y , and Z . This distribution has 8 parameters $\theta_1, \dots, \theta_8$ that represent the probabilities of each possible combination of outcomes.

⁶ For more details on representing complex probability distributions, see chapter 2 of M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray, *Algorithms for Decision Making*. MIT Press, 2022. A comprehensive overview is provided by D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

The multivariate Gaussian distribution extends the Gaussian distribution over n variables using the following probability density function:

$$\mathcal{N}(\mathbf{x} | \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{n/2} |\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right) \quad (2.7)$$

where \mathbf{x} is a vector in \mathbb{R}^n , $\boldsymbol{\mu}$ is the mean vector, and $\boldsymbol{\Sigma}$ is the covariance matrix. The mean vector $\boldsymbol{\mu}$ controls the location of the center of the distribution, while the covariance matrix $\boldsymbol{\Sigma}$ controls the spread of the distribution. The off-diagonal elements of the covariance matrix control the correlation between the values of each variable. The entries of the mean vector and covariance matrix are parameters that fully describe a multivariate Gaussian distribution (with some conditions on the parameters of the covariance matrix). The plots below show examples of the probability density functions of multivariate Gaussian distributions with different mean vectors and covariance matrices. Brighter contours indicate higher probability density.



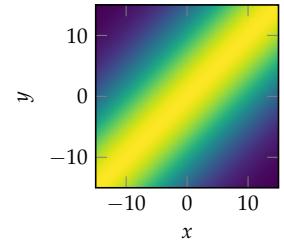
Example 2.3. The multivariate Gaussian distribution, a common multivariate distribution used to model uncertainty in multiple continuous variables.

A common model class used to model the uncertainty in one continuous variable conditioned on the value of another continuous variable is the *conditional Gaussian distribution*. Specifically, we represent the conditional distribution $p_{\theta}(y | x)$ as a Gaussian distribution with a mean that depends on the value of x :

$$p_{\theta}(y | x) = \mathcal{N}(y | f_{\theta'}(x), \sigma^2)$$

where $f_{\theta'}$ is a function of x with parameters θ' , and the full set of parameters for the model is $\theta = [\theta', \sigma^2]$. We often select $f_{\theta'}$ based on domain knowledge of the physical laws that govern the system. For example, if we know that a sensor produces noisy measurements of the true state, we may set $f_{\theta'}(x) = x$ so that the measurements will be centered around the true state. The figure in the caption shows an example of a conditional Gaussian distribution where the mean of the distribution is determined by the function $f_{\theta'}(x) = x$ and the variance is 10^2 . Brighter colors indicate higher probability density.

Example 2.4. The conditional Gaussian distribution. The plot below shows the probability density for the conditional Gaussian model $p(y | x) = \mathcal{N}(x, 10^2)$.



Conditional distributions can be represented using probability mass or density functions. For discrete variables, we can represent a conditional distribution as a table similar to a joint distribution. Table 2.2 provides an example. For continuous variables, we can represent a conditional distribution by defining a probability density function that depends on the conditioning variables. For example, we could represent the conditional distribution $p(y | x)$ as a Gaussian distribution with a mean that depends on the value of x (example 2.4). We can also represent conditional distributions in which some variables are discrete and others are continuous. *Sigmoid models*, for example, are a common class of models that represent the conditional probability of a binary variable given a continuous variable.⁷

2.3 Parameter Learning

Once we have selected a model class, we need to determine the parameters of the model that best represent the system. We refer to the process of selecting these parameters as *parameter learning*.⁸ We can learn parameters using data, expert knowledge, or a combination of both. This section will focus on two methods

X	Y	Z	$P(X Y, Z)$
0	0	0	0.08
0	0	1	0.15
0	1	0	0.05
0	1	1	0.10
1	0	0	0.92
1	0	1	0.85
1	1	0	0.95
1	1	1	0.90

Table 2.2. An example of a conditional distribution involving the binary variables X, Y, and Z.

⁷ For more details on representing conditional distributions, see section 2.4 of M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray, *Algorithms for Decision Making*. MIT Press, 2022.

⁸ In the field of machine learning, this process is often referred to as *training* the model.

to learn parameters from data.⁹ We will assume that we have a dataset D of m observations $o_{1:m}$. Each observation is a pair of input and output values such that $o_i = (\mathbf{x}_i, \mathbf{y}_i)$. Our goal is to learn the parameters θ given the dataset D .¹⁰

2.3.1 Maximum Likelihood Parameter Learning

In *maximum likelihood parameter learning*, we search for the parameters of a distribution that maximize the likelihood of observing the data. The *maximum likelihood estimate* is

$$\hat{\theta} = \arg \max_{\theta} P(D | \theta) \quad (2.9)$$

where $P(D | \theta)$ is the likelihood of the data given the parameters θ . There are two challenges associated with maximum likelihood parameter learning. One challenge is to choose the appropriate probability model for $P(D | \theta)$. We often assume that the samples in our data D are *independently and identically distributed*, which means that our samples $D = o_{1:m}$ are drawn from a distribution¹¹ $o_i \sim P(\cdot | \theta)$ with

$$P(D | \theta) = \prod_{i=1}^m P(o_i | \theta) \quad (2.10)$$

where $P(o_i | \theta) = P_{\theta}(y_i | x_i)$.

The other challenge is performing the maximization in equation (2.9). A common approach is to maximize the *log-likelihood*, often denoted as $\ell(\theta)$. Since the log-transformation is monotonically increasing, maximizing the log-likelihood produces an equivalent solution to maximizing the likelihood:¹²

$$\hat{\theta} = \arg \max_{\theta} \sum_i \log P(o_i | \theta) \quad (2.11)$$

Computing the sum of log-likelihoods tends to be more numerically stable compared to computing the product of many small probability masses or densities. Maximizing the log-likelihood forms the basis of many common objective functions used in machine learning. For example, maximizing the log-likelihood of the parameters of a conditional Gaussian distribution leads to the *least-squares* objective function, which is commonly used in regression problems (example 2.5).

Algorithm 2.1 provides a general algorithm for maximum likelihood parameter learning. We can apply several optimization algorithms to maximize the log-likelihood (see section 4.6). Example 2.6 uses algorithm 2.1 to learn the parameters of a conditional Gaussian observation model for the inverted pendulum

⁹ Techniques for learning parameters from expert knowledge are related to the preference elicitation techniques discussed in section 3.3.3. An example is provided by S. M. Katz, A.-C. LeBihan, and M. J. Kochenderfer, “Learning an Urban Air Mobility Encounter Model from Expert Preferences,” in *Digital Avionics Systems Conference (DASC)*, 2019.

¹⁰ This section focuses on learning model parameters from data, which is an important component of the field of *machine learning*. There are other methods not covered in this section such as adversarial training. A broad introduction to the field is provided by several textbooks. C. M. Bishop and H. Bishop, *Deep Learning: Foundations and Concepts*. Springer Nature, 2023. T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. Springer Series in Statistics, 2001. K. P. Murphy, *Probabilistic Machine Learning: An Introduction*. MIT Press, 2022.

¹¹ We use the notation $x \sim P(\cdot)$ to represent sampling from a distribution represented by the function P .

¹² Although it does not matter whether we maximize the natural logarithm (base e) or the common logarithm (base 10) in this equation, throughout this book we will use $\log(x)$ to mean the logarithm of x with base e .

Suppose we want to find the parameters θ' of the conditional Gaussian distribution introduced in example 2.4 by maximizing the log-likelihood of a dataset of m observations. The optimal parameters correspond to the solution of the following optimization problem:

$$\begin{aligned}
 \hat{\theta}' &= \arg \max_{\theta'} \sum_{i=1}^m \log p_{\theta'}(y_i | x_i) \\
 &= \arg \max_{\theta'} \sum_{i=1}^m \log \mathcal{N}(y_i | f_{\theta'}(x_i), \sigma^2) \\
 &= \arg \max_{\theta'} \sum_{i=1}^m \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - f_{\theta'}(x_i))^2}{2\sigma^2}\right) \\
 &= \arg \max_{\theta'} \sum_{i=1}^m \left[\log(1) - \log(\sqrt{2\pi\sigma^2}) - \frac{(y_i - f_{\theta'}(x_i))^2}{2\sigma^2} \right] \\
 &= \arg \max_{\theta'} \sum_{i=1}^m -(y_i - f_{\theta'}(x_i))^2 \\
 &= \arg \min_{\theta'} \sum_{i=1}^m (y_i - f_{\theta'}(x_i))^2
 \end{aligned}$$

The result minimizes the sum of the squared errors between the model outputs and the true outputs and is often referred to as the least-squares objective function. This result also extends to the multivariate case.

Example 2.5. Derivation of the least-squares objective function by maximizing the log-likelihood of a conditional Gaussian distribution.

```

struct MaximumLikelihoodParameterEstimation
    likelihood # p(y) = likelihood(x; θ)
    optimizer   # optimization algorithm: θ = optimizer(f)
end

function fit(alg::MaximumLikelihoodParameterEstimation, data)
    f(θ) = sum(-logpdf(alg.likelihood(x, θ), y) for (x,y) in data)
    return alg.optimizer(f)
end

```

Algorithm 2.1. Maximum likelihood parameter estimation algorithm. The algorithm takes a likelihood function, which returns a distribution over the output given the input and parameters. It also takes in an optimization algorithm, which takes in a function and returns a minimum. Given a dataset, the `fit` function returns the maximum likelihood estimate of the parameters.

Suppose we have a dataset of states and observations for the inverted pendulum system (shown in the caption). For simplicity, we will assume in this example that the pendulum state only consists of its current angle. We can model the observation model $O(o | s)$ as a conditional Gaussian distribution with a mean that depends on the state of the pendulum such that

$$O(o | s) = \mathcal{N}(o | f_{\theta'}(s), \sigma^2) \quad (2.12)$$

We will further assume that the observation is a linear function of the state such that $f_{\theta'}(s) = \theta_1 s + \theta_2$. We can learn the parameters $\theta = [\theta_1, \theta_2, \sigma^2]$ using algorithm 2.1 with the following code:

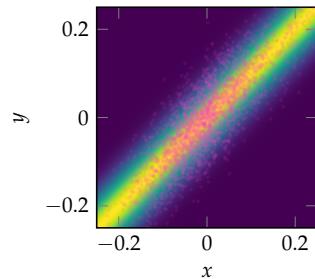
```
using Optim
likelihood(x, θ) = Normal(θ[1] * x + θ[2], exp(θ[3]))
optimizer(f) = minimizer(optimize(f, zeros(3), Optim.GradientDescent()))
alg = MaximumLikelihoodParameterEstimation(likelihood, optimizer)
θ = fit(alg, data)
```

The code uses the `Optim.jl` package to perform gradient descent optimization to learn the parameters θ starting with an initial guess of $\theta = [0, 0, 0]$. The optimized parameters result in the following model:

$$O(o | s) = \mathcal{N}(o | 1.02s + 0.00, 0.05^2)$$

These results indicate that the observation model is centered around the true state of the pendulum with a small amount of noise. As noted in example 2.5, determining θ_1 and θ_2 in this way is equivalent to optimizing the least squares objective. The figure in the caption shows the learned observation model behind the samples. Brighter colors indicate higher probability density.

Example 2.6. Learning a conditional Gaussian observation model for the inverted pendulum system. The plot shows the learned observation model with brighter colors indicating higher probability density. The data points are plotted on top of the observation model in pink.



system. Depending on the model class and optimization algorithm, algorithm 2.1 may not find the global minimum. However, for many common model classes, we can perform this optimization analytically instead. Example 2.7 derives an analytical solution for the maximum likelihood estimate of the parameters of a discrete distribution, while examples 2.8 and 2.9 derive analytical solutions for the parameters of Gaussian and conditional Gaussian distributions.

Suppose we have a binary variable X that takes on the value 1 with probability θ and the value 0 with probability $1 - \theta$. The probability of a sequence of m samples with n occurrences of 1 is

$$P(D | \theta) = \theta^n(1 - \theta)^{m-n}$$

The log-likelihood of the parameter θ is

$$\begin{aligned}\ell(\theta) &= \log(\theta^n(1 - \theta)^{m-n}) \\ &= n \log \theta + (m - n) \log(1 - \theta)\end{aligned}$$

To find the maximum likelihood estimate of θ , we set the derivative of the log-likelihood with respect to θ to zero:

$$\frac{\partial}{\partial \theta} \ell(\theta) = \frac{n}{\theta} - \frac{m - n}{1 - \theta} = 0$$

Solving for θ results in the maximum likelihood estimate $\hat{\theta} = \frac{n}{m}$. Computing the maximum likelihood estimate for a variable X that can assume k values results in a similar formula. The maximum likelihood estimate for $P(x_i | n_{1:k})$ is given by

$$\hat{\theta}_i = \frac{n_i}{\sum_{j=1}^k n_j}$$

where $n_{1:k}$ are the observed counts for the k different values.

Example 2.7. Maximum likelihood parameter learning for a binary variable and a variable with k possible values.

Algorithm 2.1 requires that we have all of the data required to learn the parameters. In practice, we may have missing data. For example, when we train a Gaussian mixture model, we may not know which component of the mixture generated each data point. Furthermore, when we learn the transition model and observation models, we may only have access to the observations and actions

In a Gaussian distribution, the log-likelihood of the mean μ and variance σ^2 with m samples is given by

$$\ell(\mu, \sigma^2) = -m \log(\sqrt{2\pi}) - m \log \sigma - \frac{\sum_i (o_i - \mu)^2}{2\sigma^2} \quad (2.13)$$

We can use the standard technique for finding the maximum of a function by setting the partial derivative of ℓ with respect to each parameter to 0 and solving for the parameter:

$$\frac{\partial}{\partial \mu} \ell(\mu, \sigma^2) = \frac{\sum_i (o_i - \hat{\mu})}{\hat{\sigma}^2} = 0 \quad (2.14)$$

$$\frac{\partial}{\partial \sigma} \ell(\mu, \sigma^2) = -\frac{m}{\hat{\sigma}} + \frac{\sum_i (o_i - \hat{\mu})^2}{\hat{\sigma}^3} = 0 \quad (2.15)$$

After some algebraic manipulation, we get

$$\hat{\mu} = \frac{1}{m} \sum_i o_i \quad \hat{\sigma}^2 = \frac{1}{m} \sum_i (o_i - \hat{\mu})^2 \quad (2.16)$$

where $\hat{\mu}$ is the population mean and $\hat{\sigma}^2$ is the population variance.

Example 2.8. Analytical solution for finding the parameters of a Gaussian distribution using maximum likelihood parameter learning.

Consider a linear Gaussian model $p(\mathbf{y} | \mathbf{x}) = \mathcal{N}(\mathbf{y} | \mathbf{Ax} + \mathbf{b}, \Sigma)$. Suppose we want to find the maximum likelihood estimate of \mathbf{A} and \mathbf{b} given a dataset of m observations. As shown in example 2.5, this process is equivalent to minimizing the sum of the squared errors between the model outputs and the true outputs:

$$\arg \min_{\mathbf{A}, \mathbf{b}} \sum_{i=1}^m \|\mathbf{Ax}_i + \mathbf{b} - \mathbf{y}_i\|^2$$

Let the matrix \mathbf{X} be defined such that each row is a data point \mathbf{x}_i augmented with a one in the final column, and let \mathbf{Y} be a matrix such that each row is a data point \mathbf{y}_i . If we let $\theta = [\mathbf{A} \ \mathbf{b}]^\top$, we can rewrite the optimization problem as

$$\arg \min_{\theta} \|\mathbf{X}\theta - \mathbf{Y}\|^2$$

Setting the gradient of the objective function with respect to θ to zero and solving for θ results in the following closed-form solution:

$$\hat{\theta} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{Y}$$

where $(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$ is often referred to as the *pseudoinverse* of \mathbf{X} .

We can use the following code to analytically solve for θ_1 and θ_2 in the linear Gaussian model in example 2.6:

```
X = hcat(s, ones(length(s)))
θ₁, θ₂ = pinv(X) * o
```

where the `pinv` function computes the pseudoinverse of the matrix `X`. The result is $\theta_1 = 1.02$ and $\theta_2 = 0.00$, which matches the result from example 2.6.

Example 2.9. Analytical solution for finding the parameters of a linear Gaussian model using maximum likelihood parameter learning.

taken by the agent and not the true state of the environment. In these cases, we can use the *expectation-maximization (EM)* algorithm to learn the parameters of the model, which involves iterative improvement of the parameter estimate.¹³

2.3.2 Bayesian Parameter Learning

In *Bayesian parameter learning*, we estimate a distribution over model parameters given the data. We write this distribution as $P(\theta | D)$.¹⁴ This distribution can help us quantify our uncertainty about the true value of θ . We can convert this distribution into a point estimate by computing the expectation:

$$\hat{\theta} = \mathbb{E}_{\theta \sim P(\cdot | D)}[\theta] = \sum_{\theta} \theta P(\theta | D) \quad (2.17)$$

In some cases, however, the expectation may not be an acceptable estimate, as illustrated in figure 2.5. An alternative is to use the *maximum a posteriori* estimate:

$$\hat{\theta} = \arg \max_{\theta} P(\theta | D) \quad (2.18)$$

This estimate corresponds to a value of θ that is assigned the greatest density. This is often referred to as the *mode* of the distribution. As shown in figure 2.5, the mode may not be unique.

We can derive an expression for $P(\theta | D)$ in terms of the likelihood model introduced in section 2.3.1 using Bayes' rule:¹⁵

$$P(\theta | D) = \frac{P(D | \theta)P(\theta)}{\sum_{\theta} P(D | \theta)P(\theta)} \quad (2.19)$$

In addition to the likelihood model $P(D | \theta)$, we need to specify a *prior distribution* $P(\theta)$ over the parameters. The prior distribution encodes our beliefs about the values of the parameters before observing the data. The output of equation (2.19) is often referred to as the *posterior distribution*.

In general, computing the posterior distribution using equation (2.19) is challenging because the denominator is often difficult or impossible to compute analytically.¹⁶ The number of terms in the summation scales exponentially with the number of parameters, and for continuous parameters, the integral is often intractable. For some model classes and priors, however, an analytical solution is possible. Example 2.10 shows an example of Bayesian parameter learning for a simple model class using a *conjugate prior*. A conjugate prior is a prior distribution

¹³ An overview of the EM algorithm is provided in section 4.4 of M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray, *Algorithms for Decision Making*. MIT Press, 2022.

¹⁴ If θ is continuous, the distribution is represented by a probability density $p(\theta | D)$ instead of a probability mass. In this case, the summations in equations (2.17) and (2.19) change to integrals.

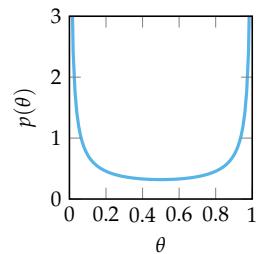


Figure 2.5. An example of a distribution where the expected value of θ is not a good estimate. The expected value of 0.5 has a lower density than occurs at the extreme values of 0 or 1.

¹⁵ Bayes' rule can be derived from the definition of conditional probability and is named for the English statistician and Presbyterian minister Thomas Bayes (c. 1701–1761) who provided a formulation of this theorem. A history is provided by S. B. MacGrayne, *The Theory That Would Not Die*. Yale University Press, 2011.

¹⁶ Note that it is possible to compute the maximum a posteriori estimate using equation (2.18) without computing the denominator of equation (2.19).

that, when combined with a likelihood model, results in a posterior distribution that is in the same class as the prior distribution.¹⁷

If we cannot compute the posterior distribution analytically, we can approximate it with a set of samples using *probabilistic programming*. Probabilistic programming languages allow us to specify the prior and likelihood models such that we can automatically generate samples from the posterior distribution.¹⁸ We will discuss probabilistic programming techniques in more detail in chapter 6. Algorithm 2.2 provides a probabilistic programming implementation of Bayesian parameter learning. It takes in the prior and likelihood models and uses a sampling algorithm from a probabilistic programming package to generate m samples from the posterior distribution.

Example 2.11 provides an implementation of Bayesian parameter learning for the inverted pendulum system, and figure 2.6 shows the results for different dataset sizes. In general, we decrease our uncertainty about the parameters as we observe more data, so the posterior distribution becomes more concentrated with more data. Bayesian parameter learning also provides a principled way to incorporate prior knowledge into the learning process. The prior distribution can encode expert knowledge about the parameters, which can be particularly useful when we have limited data.

2.3.3 Generalization

An important metric to consider when selecting model parameters is *generalization performance*. The generalization performance of a model is a measure of its performance over the distribution over its full input space, including points that were not used to train the model. We measure generalization performance with respect to a performance metric. A common performance metric is the average log-likelihood that the model assigns to points in a dataset sampled from the distribution over the input space. We want to select the parameters with the best generalization performance. This section discusses techniques for estimating generalization performance.

It may be tempting to estimate the generalization performance by computing the performance metric on the training data. However, performing well on the training data does not necessarily indicate good generalization performance. Complex models may perform well on the training set, but they may not provide good predictions at other points in the input space. This concept is often referred

¹⁷ Distributions in the natural exponential family have conjugate priors. The natural exponential family includes many common distributions such as the Gaussian, Bernoulli, and Poisson distributions.

¹⁸ The `Turing.jl` package, for example, provides a common probabilistic programming interface. H. Ge, K. Xu, and Z. Ghahramani, “Turing: a Language for Flexible Probabilistic Inference,” in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2018.

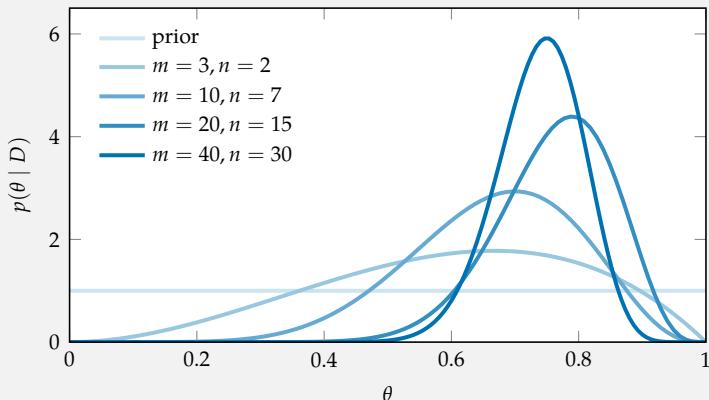
Suppose we have a binary variable X that takes on the value 1 with probability θ and the value 0 with probability $1 - \theta$. The likelihood of observing a sequence of m samples with n occurrences of 1 is given by

$$P(D | \theta) = \theta^n (1 - \theta)^{m-n}$$

which corresponds to a product of Bernoulli distributions. The Beta distribution is a conjugate prior for the Bernoulli distribution. Given a prior distribution of $p(\theta) = \text{Beta}(\theta | \alpha, \beta)$, the posterior distribution is

$$p(\theta | D) = \text{Beta}(\theta | \alpha + n, \beta + m - n)$$

The distribution $\text{Beta}(1, 1)$ assigns uniform probability to also possible values of θ between 0 and 1. The plot below shows examples of the Beta distribution with different datasets. As more data is observed, the posterior distribution becomes more concentrated indicating less uncertainty in the parameter value.



Example 2.10. Bayesian parameter learning for a binary variable using the Beta distribution as a prior. The plot shows the Beta distribution with different datasets.

```

struct BayesianParameterEstimation
    likelihood #  $p(y) = \text{likelihood}(x, \theta)$ 
    prior      # prior distribution
    sampler    # Turing.jl sampler
    m         # number of samples from posterior
end

function fit(alg::BayesianParameterEstimation, data)
    x, y = first.(data), last.(data)
    @model function posterior(x, y)
         $\theta \sim \text{alg.prior}$ 
        for i in eachindex(x)
             $y[i] \sim \text{alg.likelihood}(x[i], \theta)$ 
        end
    end
    return Turing.sample(posterior(x, y), alg.sampler, alg.m)
end

```

Algorithm 2.2. Bayesian parameter estimation algorithm using the `Turing.jl` probabilistic programming package. The algorithm takes a likelihood function similar to the one used in algorithm 2.1, a prior distribution over the model parameters, a probabilistic programming sampler from `Turing.jl`, and the number of samples to generate from the posterior distribution. The `fit` function creates a probabilistic program that specifies that the parameters are drawn from the prior and the likelihood model generates each data point. The function returns m samples from the posterior distribution.

Consider the linear Gaussian observation model for the inverted pendulum introduced in example 2.6, and suppose we want to use Bayesian parameter learning to sample a distribution over the parameters $\theta = [\theta_1, \theta_2, \sigma^2]$. We can use the following code to generate $m = 1000$ samples from the posterior distribution over θ using algorithm 2.2:

```

likelihood(x, θ) = Normal(θ[1] * x + θ[2], exp(θ[3]))
prior = MvNormal(zeros(3), 4I)
alg = BayesianParameterEstimation(likelihood, prior, NUTS(), 1000)
θ = fit(alg, data)

```

The code uses the `NUTS`, or No U-Turn Sampler, from the `Turing.jl` package to generate samples from the posterior distribution. Figure 2.6 shows the results for different dataset sizes.

Example 2.11. Implementation of Bayesian parameter learning for the inverted pendulum observation model. The results are shown in figure 2.6. A detailed overview of the NUTS algorithms is provided by M. D. Hoffman, A. Gelman, et al., “The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo,” *Journal of Machine Learning Research (JMLR)*, vol. 15, no. 1, pp. 1593–1623, 2014.

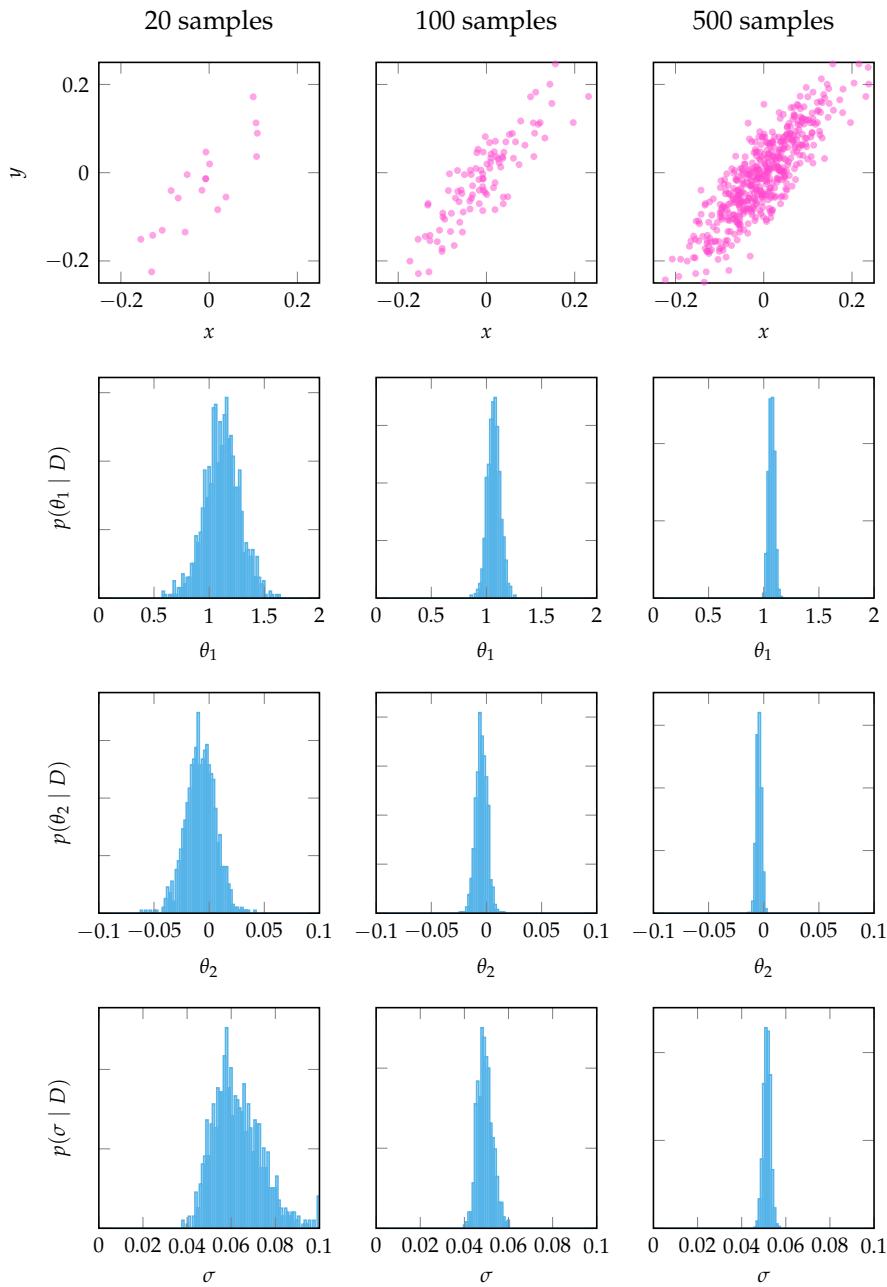


Figure 2.6. Learning the parameters of a linear Gaussian observation model for the inverted pendulum system given different amounts of data from the dataset in example 2.6. The top row shows the data points used for each column, and the remaining three rows show the posterior distribution over the parameters θ_1 , θ_2 , and σ for different amounts of data. As more data is observed, the posterior distribution becomes more concentrated around the true parameter values.

to as *overfitting* (figure 2.7). Therefore, instead of selecting the model parameters that best fit the training data, we should select the model parameters that result in the best performance on separate dataset that the model has not seen before.

A simple approach to estimating the generalization performance using an unseen dataset is the *holdout method*, which partitions the available data into a *test set* and a *training set*. We use the training set to learn the model parameters and the test set for evaluation. Depending on the size and nature of the dataset, we may use different ratios to split the training and test data ranging from 50% train and 50% test to 99% train and 1% test. Using too few samples for training can result in poor fits, whereas using too many will result in poor generalization estimates.

Using a train-test partition can be wasteful because our model tuning can take advantage only of a segment of our data. We can often obtain better results using *k-fold cross validation*.¹⁹ To perform this technique, we randomly partition the data into k segments of approximately equal size. We then train k models, one on each subset of $k - 1$ sets, and we use the withheld set to estimate the generalization performance. The cross-validation estimate of generalization performance is the mean generalization performance over all folds.²⁰

2.4 Agent Models

For some systems, the environment may contain other agents that we need to incorporate into our environment model. Depending on the available data and the assumptions we make about the other agents, we can use different techniques to model their behavior. This section discusses three categories of techniques for modeling other agents.

2.4.1 Imitation Learning

Imitation learning is a technique for learning a policy by observing the behavior of another agent. A common technique for imitation learning is *behavioral cloning*. Given a dataset of state-action pairs from the expert agent, we can use *behavioral cloning* to learn a policy that maps states to actions. Behavioral cloning methods learn a policy $\pi_\theta(a | s)$ by finding a θ that maximizes the likelihood of the actions taken in the dataset. Once we select a model class to represent the policy, we

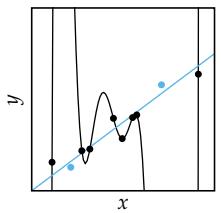


Figure 2.7. Example where a complex model (black line) fits the training data (black) perfectly but does not generalize well to other data points (blue). A simpler linear model (blue line) provides the best fit when considering all data-points.

¹⁹ This method is also known as *rotation estimation*.

²⁰ Another common approach related to cross-validation is the *bootstrap method*, which involves resampling the dataset with replacement to estimate the generalization performance. B. Efron, "Bootstrap Methods: Another Look at the Jackknife," in *Breakthroughs in Statistics: Methodology and Distribution*, Springer, 1992, pp. 569–593.

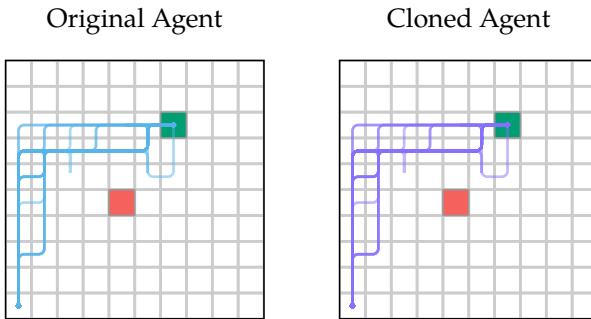


Figure 2.8. Behavioral cloning of a grid world agent. The behavioral clone is trained on the set of trajectories shown in the left plot. The right plot shows rollouts of the learned policy, which appear similar to the original trajectories.

can use the techniques from section 2.3.1 to learn the parameters of the policy. Figure 2.8 shows an example of behavioral cloning for a grid world agent.

If the model class used for behavioral cloning is not expressive enough or the dataset contains errors, the learned policy may not be optimal. This result can lead to *cascading errors*, which occur when small errors compound during a rollout and eventually lead to states that are poorly represented in the training data. The policy of a cloned agent may not generalize well to these states, causing inaccurate behavior. One way to address the problem of cascading errors is to correct the learned policy with additional data. *Sequential interactive demonstration* methods such as *data set aggregation (DAgger)*²¹ alternate between collecting new data in states reached by the trained policy and using this data to improve the policy.

Another common technique for imitation learning is *inverse reinforcement learning*. In inverse reinforcement learning, we assume that the expert agent is optimizing an unknown reward function when selecting its actions, and our goal is to determine this reward function given a dataset of trajectory rollouts from the expert agent. Common techniques for inverse reinforcement learning select a parametric function form for the reward function and learn the parameters of the reward function according to a particular objective.

One common objective for learning reward function parameters involves maximizing the margin between the reward of the expert agent and the reward of other agents. This technique is known as *maximum margin inverse reinforcement learning*.²² Another common objective is to maximize the entropy of the distribution over trajectories produced by the learned policy, which is known as *maximum*

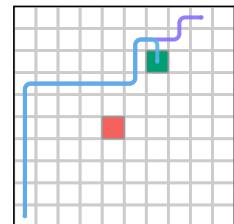


Figure 2.9. Cascading errors in behavioral cloning for a grid world agent. The clone is trained on a trajectory that does not reach any states above the goal state. While the original agent (blue) is able to correctly turn back toward the goal in this region, the clone (purple) continues to move away from the goal.

²¹ S. Ross, G. J. Gordon, and J. A. Bagnell, "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning," in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, vol. 15, 2011.

²² P. Abbeel and A. Y. Ng, "Apprenticeship Learning via Inverse Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, 2004.

*entropy inverse reinforcement learning.*²³ Once we have learned the reward function, we can use it to model the policy of the agent.²⁴

2.4.2 Behavior Models

If we have a model of the utility of the an agent, we can use a *behavior model* to predict its actions. In particular, suppose we have a utility function $U(s, a)$ that assigns a utility to each state-action pair. We can model the behavior as a policy that selects actions to maximize the utility of the agent. This policy is given by

$$\pi(a | s) = \arg \max_a U(s, a) \quad (2.20)$$

The policy in equation (2.20) is known as the *best response* policy.

Some agents are not perfectly rational optimizers of their utility functions. Humans, for example, do not always select actions that maximize their utility.²⁵ We can model this behavior using a *softmax response* policy.²⁶ The principle underlying the softmax response model is that agents are more likely to make errors in their optimization that are less costly. Given a *precision parameter* $\lambda \geq 0$, the softmax response policy is given by

$$\pi(a | s) = \frac{\exp(\lambda U(s, a))}{\sum_{a'} \exp(\lambda U(s, a'))} \quad (2.21)$$

As λ approaches 0, the policy selects actions uniformly at random, and as λ approaches infinity, the policy approaches the best response policy. We can treat λ as a parameter that can be learned from data using, for example, maximum likelihood estimation.

2.4.3 Interaction Models

Agents that operate in the presence of other agents may base their decisions on their belief over the behavior of the other agents.²⁷ For example, consider an aircraft collision avoidance scenario in which an intruder aircraft is approaching at the same altitude as our aircraft. In this scenario, we want our aircraft to take the opposite action of the intruder aircraft (figure 2.10). If the intruder chooses to climb, our best action is to descend, while if the intruder chooses to descend, our best action is to climb.

²³ B. D. Ziebart, A. Maas, J. A. Bagnell, and A. K. Dey, "Maximum Entropy Inverse Reinforcement Learning," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2008.

²⁴ An overview of methods for learning policies from reward functions is provided by M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray, *Algorithms for Decision Making*. MIT Press, 2022.

²⁵ Several recent books discuss apparent human irrationality. D. Ariely, *Predictably Irrational: The Hidden Forces That Shape Our Decisions*. Harper, 2008. J. Lehrer, *How We Decide*. Houghton Mifflin, 2009.

²⁶ This response is sometimes referred to as a *logit response* or *quantal response*.



Figure 2.10. If the intruder aircraft (gray) chooses to descend, the best action for our aircraft (purple) is to climb. In general, the best action for our aircraft is the opposite of the action chosen by the intruder.

²⁷ An overview of many behavioral models is provided in C. F. Camerer, *Behavioral Game Theory: Experiments in Strategic Interaction*. Princeton University Press, 2003.

We can model this interaction between agents using *interaction models*.²⁸ For example, a *hierarchical* interaction model specifies the *depth of rationality* of an agent by a level of $k \geq 0$. A level 0 agent selects its action without regard to the actions of other agents. A level 1 agent selects its action by assuming that all other agents are level 0 agents. In general, a level k agent selects its action by assuming that all other agents are level $k - 1$ agents. Figure 2.11 shows an example of this model for an aircraft collision avoidance scenario.

Another common behavior model is the *hierarchical softmax* model, which accounts for the fact that agents may have different levels of rationality.²⁹ A level 0 agent selects actions uniformly at random. A level 1 agent selects actions according to the softmax response policy with a precision parameter λ that assumes that all other agents are level 0 agents. A level k agent selects actions according to a softmax model of the other players playing level $k - 1$. We can learn the k and λ parameters from data using maximum likelihood estimation.

2.5 Model Validation

Since the validity of any downstream analysis of a system depends on the accuracy of the models we use, it is important to rigorously validate our models. We can use a variety of features to validate a model. Given a dataset, we can compare characteristics of the model distribution to the empirical distribution of the data. We can also compute features by comparing rollouts of the model to rollouts of the true system. For example, given a model of aircraft collision avoidance behavior, we can compare the average miss distance of the aircraft when using the model to average miss distance of the true system trajectories. This section discusses common model validation techniques that compare features of the model to the true system.

2.5.1 Visual Diagnostics

One important part of the model validation process involves ensuring that the distribution over model features matches that of the true system. For example, we may want to confirm that our analytical model of sensor observations matches a dataset of real sensor measurements. In this case, the distribution from our model is represented analytically, while the distribution from the true system is represented as a set of samples. In other cases, both the model distribution

²⁸ The topic of interaction models is closely related to the field of game theory. Several introductory books include D. Fudenberg and J. Tirole, *Game Theory*. MIT Press, 1991. Y. Shoham and K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game Theoretic, and Logical Foundations*. Cambridge University Press, 2009.

²⁹ This approach is sometimes called *quantal-level-k* or *logit-level-k*. D. O. Stahl and P. W. Wilson, "Experimental Evidence on Players' Models of Other Players," *Journal of Economic Behavior & Organization*, vol. 25, no. 3, pp. 309–327, 1994.

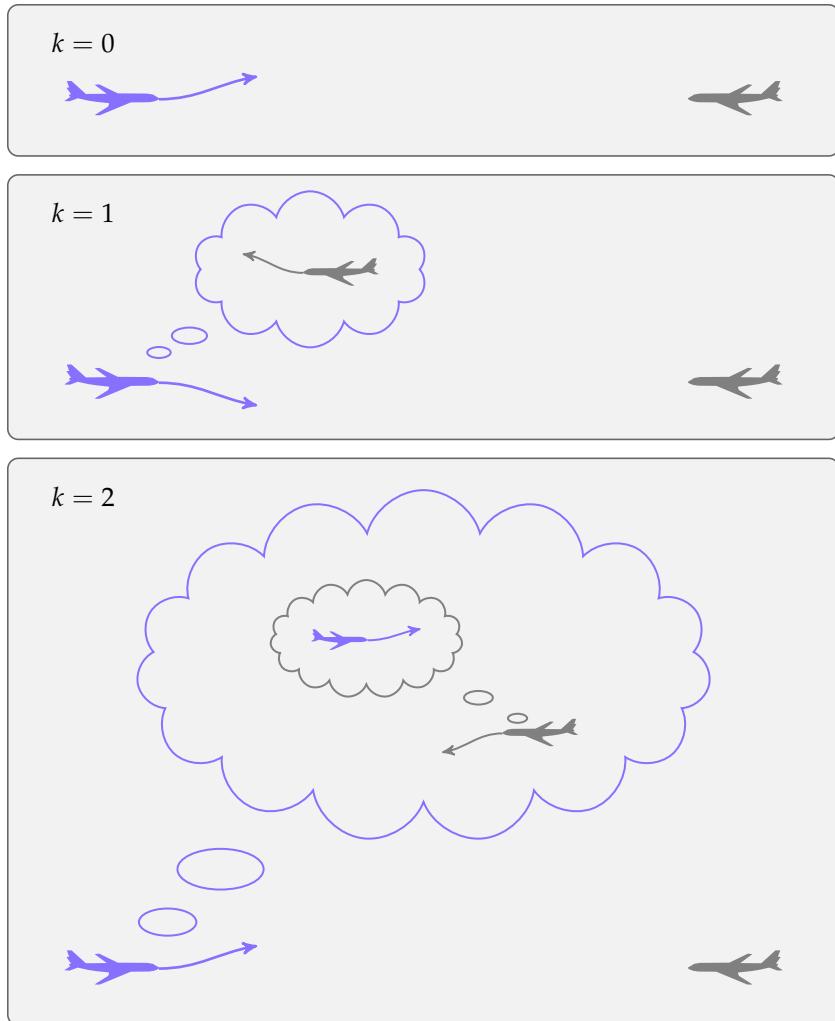


Figure 2.11. A hierarchical interaction model for an aircraft collision avoidance scenario in which the intruder aircraft (gray) is approaching at the same altitude as our aircraft (purple). A level 0 agent selects the best action according to its policy, which is to climb. A level 1 agent assumes that the intruder has the policy of a level 0 agent and selects the best action given this assumption, which is to descend. A level 2 agent assumes that the intruder has the policy of a level 1 agent and chooses to climb.

and true distribution are represented as a set of samples. For example, we may want to compare the distribution over airspeed from rollouts of an aircraft encounter model to the distribution over airspeed from trajectories of true aircraft encounters.³⁰

One way to compare two feature distributions is to compare their probability density functions. We can plot the probability density function of the model on the same plot as the probability density function of the data. For distributions that are represented as a set of samples, we can plot an approximate probability density by creating a histogram of the samples. We may also compare the cumulative distribution functions of a variable X ($P(X \leq x)$) for the model and data. If we do not have an analytical model of the cumulative distribution function, we can plot the empirical cumulative distribution function of the samples. Figure 2.12 compares the empirical cumulative distribution function of two sets of samples.

Another common visual diagnostic is the *quantile-quantile plot* (Q-Q plot). The α -quantile of a distribution is the value q for which

$$P(X \leq q) = \alpha \quad (2.22)$$

A Q-Q plot compares the quantiles of the model distribution to the quantiles of the data. The horizontal axis of a Q-Q plot represents the quantiles of the model distribution, while the vertical axis represents the quantiles of the data. If the model distribution matches the data, the points in the Q-Q plot will lie on the line that passes through the origin with a slope of 1.

We can also compare distributions using *calibration plots*. The horizontal axis of a calibration plot corresponds to values of α between 0 and 1, while the vertical axis corresponds to the fraction of data points that lie below the α -quantile of the model. Similar to the Q-Q plot, a well-calibrated model will produce a calibration plot that lies on the line that passes through the origin with a slope of 1. Figure 2.13 shows examples of probability density, cumulative distribution, Q-Q, and calibration plots for a set of samples and four different analytical models.

2.5.2 Summary Metrics

It is often desirable to summarize the differences between the modeled and true distribution using a single quantity. For example, we may want to compare the probability density function of the model to the data using a single number. One common quantity used to compare two densities is the *Kullback-Leibler divergence*

³⁰ M. J. Kochenderfer, M. W. M. Edwards, L. P. Espindle, J. K. Kuchar, and J. D. Griffith, "Airspace Encounter Models for Estimating Collision Risk," *AIAA Journal on Guidance, Control, and Dynamics*, vol. 33, no. 2, pp. 487–499, 2010.

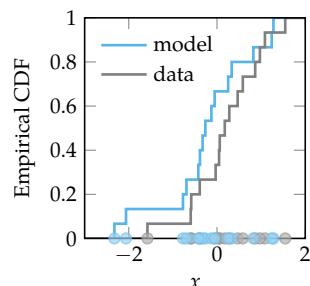


Figure 2.12. Comparison of the empirical cumulative distribution function for two sets of samples represented by the blue and purple dots. The function represents the fraction of samples below each value of x .

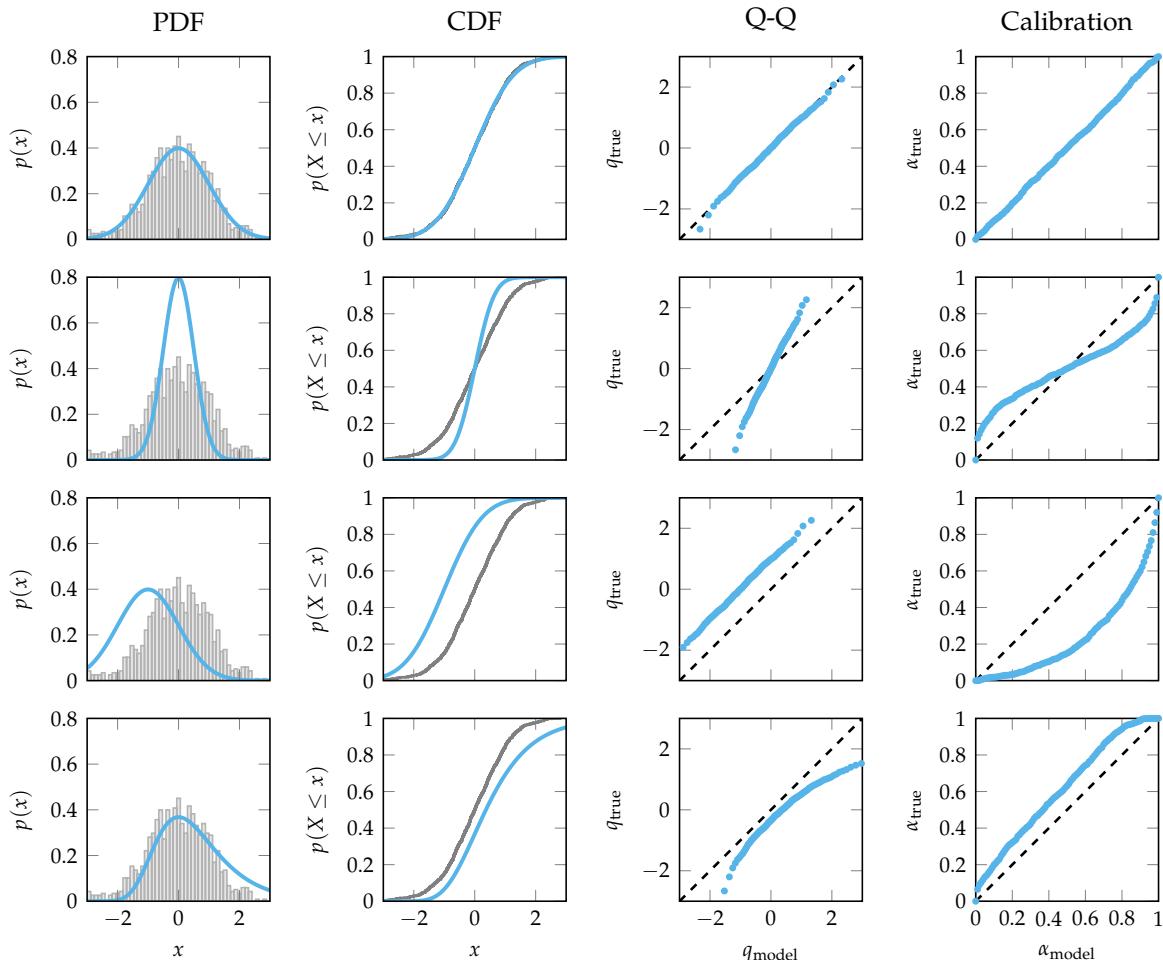


Figure 2.13. Visual diagnostics that compare a set of samples (gray) to four possible models (blue). Each row shows a different model. As shown in the plots, the model in the top row fits the data better than the models in the remaining rows.

(KL divergence).³¹ The KL divergence of density $p(x)$ from density $q(x)$ is defined as

$$D_{\text{KL}}(p \parallel q) = \int p(x) \log \left(\frac{p(x)}{q(x)} \right) dx \quad (2.23)$$

The KL divergence is 0 if $p(x) = q(x)$ for all x and greater than 0 otherwise. If the model and data distributions are represented as samples, we can estimate the KL divergence using densities estimated from their histograms.³²

The KL divergence is part of a broad class of divergences used in information theory and statistics called the *F-divergences*.³³ Another common divergence measure in this class is the Jensen-Shannon divergence,³⁴ which is a symmetric version of the KL divergence. The F-divergences do not necessarily have all of the properties of a distance metric. For example, the KL divergence is not symmetric, meaning that $D_{\text{KL}}(p \parallel q) \neq D_{\text{KL}}(q \parallel p)$. Furthermore, the KL divergence is not defined if the support of $p(x)$ is not a subset of the support of $q(x)$. We can use the *Wasserstein distance* to compare two densities with different supports.³⁵

We can summarize the cumulative distribution plot by calculating the maximum distance between the cumulative distribution functions of the two distributions. This distance is called the *Kolmogorov-Smirnov statistic (K-S statistic)* and is defined as

$$D_{\text{KS}} = \max_x |P(X \leq x) - Q(X \leq x)| \quad (2.24)$$

where $P(X \leq x)$ and $Q(X \leq x)$ are the cumulative distributions of the model and data.³⁶ We can compute a similar metric from the calibration plot by computing the maximum distance between the points on the calibration plot and the line passing through the origin with a slope of 1. This quantity is referred to as the *maximum calibration error (MCE)*. It is also common to compute the *expected calibration error (ECE)* by averaging the distances. Figure 2.14 illustrates these metrics for the two sets of samples shown in figure 2.12.

2.5.3 Comparing Multiple Features

We often want to compare multiple features of a model to the true system. One option is to compare one feature at a time using the techniques in sections 2.5.1 and 2.5.2. However, it is also important to ensure that the model accurately captures the relationships between features of the true system, and checking one feature at a time may cause us to miss these relationships. Figure 2.15 shows an example in which performing visual diagnostics on the individual features

³¹ The KL divergence is named after American mathematicians Solomon Kullback (1907–1994) and Richard Leibler (1914–2003), who introduced the concept. S. Kullback and R. A. Leibler, “On Information and Sufficiency,” *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.

³² Another approach is to fit a kernel density estimate to the samples and then compute the KL divergence between the two kernel density estimates.

³³ A detailed overview is provided by F. Liese and I. Vajda, “On Divergences and Informations in Statistics and Information Theory,” *IEEE Transactions on Information Theory*, vol. 52, no. 10, pp. 4394–4412, 2006.

³⁴ Named for Danish mathematician Johan Jensen (1859–1925) and American mathematician Claude Shannon (1916–2001).

³⁵ The Wasserstein distance is named after Russian-American mathematician Leonid Vaserstein (1944–pres.). It is also known as the *earth mover’s distance* because it can be interpreted as the amount of work required to transform a pile of earth representing one distribution to a pile of earth representing the other.

³⁶ This statistic is often used in a K-S test to check whether two sets of samples are drawn for the same distribution. It is named after Soviet mathematicians Andrey Kolmogorov (1903–1987) and Nikolai Smirnov (1900–1966).

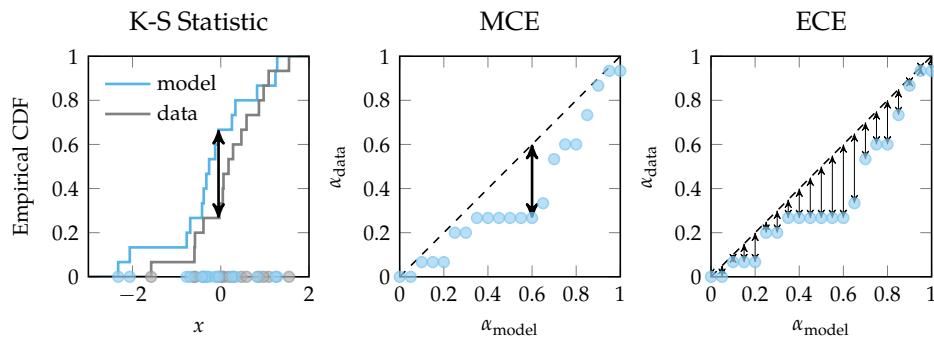


Figure 2.14. Illustration of the K-S statistic, MCE, and ECE for two sets of samples. The black arrows indicate the quantity of interest for each plot. The ECE is computed by averaging the lengths of the arrows on the plot.

produces misleading results. The model distribution does not match the data points in two-dimensional space, but the individual feature distributions match the data.

Several techniques allow us to check whether a model accurately captures the relationships between features of the true system. One technique involves creating a single feature that captures the relationships between the features of the true system. We can then use techniques discussed in sections 2.5.1 and 2.5.2 to compare the single feature of the model to the single feature of the true system. Figure 2.16 shows an example of creating a single feature that models the relationship between the features in figure 2.15. One drawback of this approach is that it requires domain knowledge to create the single feature.

Another way to compare multiple features is to extend the metrics discussed in section 2.5.2 to multivariate distributions. Many of the comparison metrics for probability density functions have straightforward extensions. The KL divergence, for example, can be extended to multivariate distributions by using the probability density of the joint distribution of the model and data for the variables of interest (figure 2.17). In contrast, the visual diagnostics and metrics that use the cumulative distribution or quantile functions are less straightforward to extend because the quantile function is not defined in multiple dimensions. Therefore, these metrics require extensions of the quantile function to higher dimensions.³⁷

We may also want to compare the distribution over a feature conditioned on the value of another feature. For example, we may want to compare the distribution over sensor observations conditioned on the true state. One way to make this

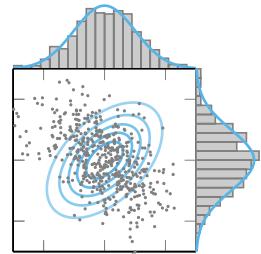


Figure 2.15. Example in which performing visual diagnostics on individual features can produce misleading results. The model distribution (blue contours indicating points of equal density) does not match the data points (gray) in two-dimensional space. However, the individual feature distributions of the model match the individual feature distributions of the data.

³⁷ Multiple definitions of the quantile function in higher dimensions have been proposed. P. Chaudhuri, "On a Geometric Notion of Quantiles for Multivariate Data," *Journal of the American Statistical Association*, vol. 91, no. 434, pp. 862–872, 1996.

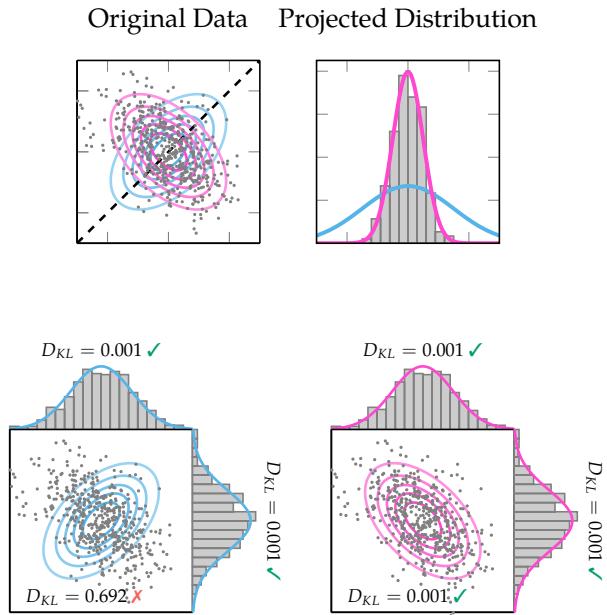


Figure 2.16. Creating a single feature that captures the relationships between the features in figure 2.15 by projecting the model and data onto the line $y = x$. This feature allows us to identify a mismatch between the model (blue) and the data (gray), as shown in the right-most plot. The pink contours show an alternative model that better matches the data.

Figure 2.17. Comparison of the features of two possible models to a set of data sampled from the true system using KL divergence. If we calculate the KL divergence of each feature separately, the models appear to match the data equally well. However, if we calculate the KL divergence of the joint distribution of the features, we see that the pink model matches the data better than the blue model.

comparison is to partition the conditioning variable into a set of bins and then compare the distribution over the feature in each bin using the metrics described in this section. Figure 2.18 shows an example of this technique. It is also possible to create a single calibration plot for the conditional distribution checking the α -quantile for each x -value checking how often the corresponding y -value is below the α -quantile of the model distribution.

2.5.4 Subjective Evaluation

We can also evaluate models based on expert knowledge. One evaluation metric is the ability of an expert to distinguish between samples produced by the model and samples produced by the true system.³⁸ A model represents the true system well if an expert cannot distinguish between the generated samples and the true samples. This idea is similar to the *Turing test*, which was proposed as a way to test whether a machine has human intelligence.³⁹

³⁸ R. Bhattacharyya, S. Jung, L. Kruse, R. Senanayake, and M.J. Kochenderfer, “A Hybrid Rule-Based and Data-Driven Approach to Driver Modeling Through Particle Filtering,” *IEEE Transactions on Intelligent Transportation Systems*, no. 2108.12820, 2021.

³⁹ This test was first proposed by English mathematician and computer scientist Alan Turing (1912–1954) in an 1950 essay. He originally called the test the *imitation game* in which a human judge interacts with a machine and a human and must determine which is which. A. M. Turing, “Computing Machinery and Intelligence,” *Mind*, vol. 59, pp. 433–460, 1950.

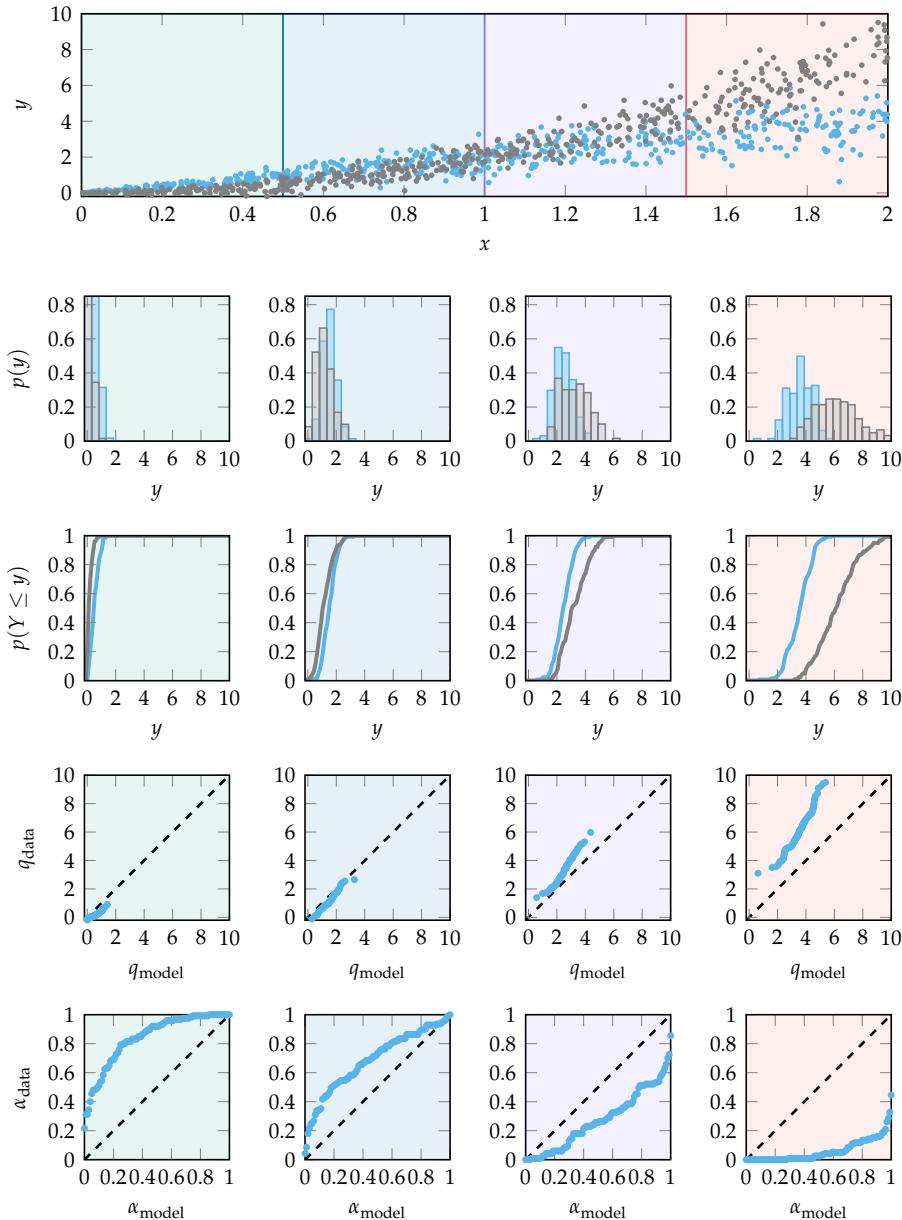


Figure 2.18. Analysis of samples from the conditional distribution $p(y | x)$ for the model (blue) and true system (gray). The top plot shows samples from each distribution for different values of x . The background colors each represent a bin containing a range of x values. We can analyze the performance of the conditional model in each bin by comparing the distributions of y values using the visual diagnostics in section 2.5.1. The plots in the remaining rows show the visual diagnostics for each bin. As we can see, the model mismatch increases for larger values of x .

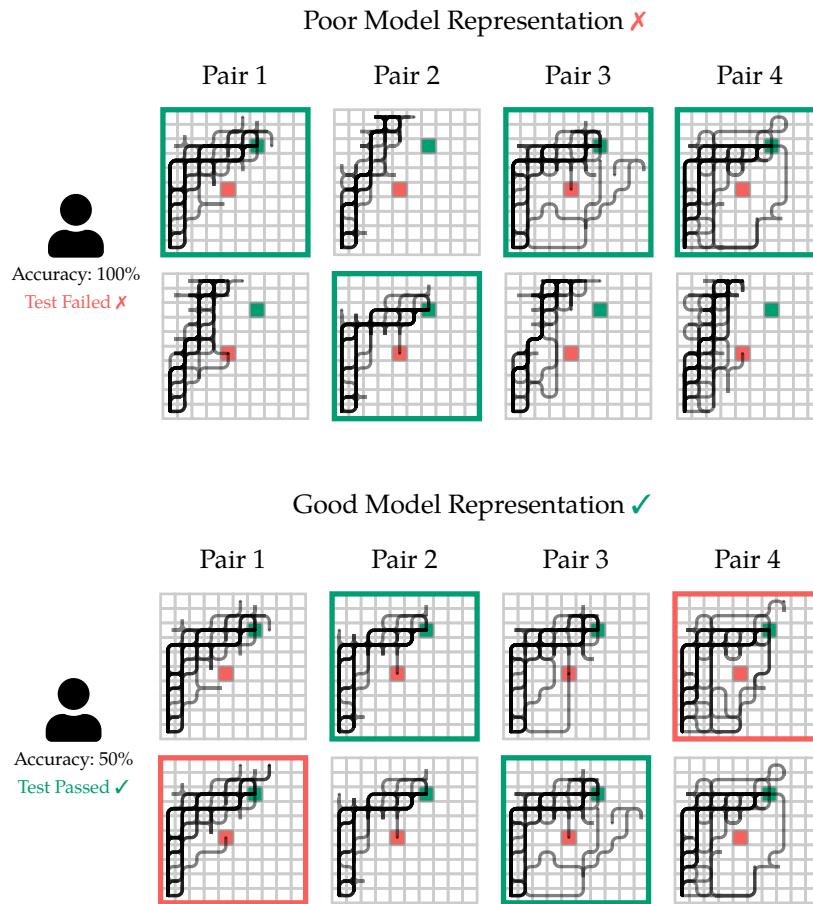


Figure 2.19. Validating a model of a grid world agent using expert knowledge. Each column represents a pair of rollouts from the true system and the model in a random order. The expert's selection is highlighted in green if they correctly identified the true system and in red if they incorrectly identified the model. The top row shows an example where the model does not represent the true system well, and the expert is able to determine the true model in each pair. The bottom row shows an example where the model represents the true system well, and the expert is unable to distinguish between the true system and the model.

We can evaluate this metric by showing an expert pairs consisting of one set of rollouts from the true system and one set of rollouts from the model. We can then ask the expert to identify which set of rollouts was produced by the true system. We can quantify the performance of the model by measuring the expert's accuracy in distinguishing between the two sets. If the expert's accuracy is around 50%, their performance is no better than random guessing, and we can conclude that the model is a good representation of the true system. Figure 2.19 shows an example of this test for a model of a grid world agent.

2.5.5 Sensitivity Analysis

No matter what parameter learning scheme we use, the resulting model is unlikely to be a perfect representation of the behavior of our system. Because there will always be inherent uncertainty in the parameters that we select, we often want to understand how sensitive our downstream analysis might be to the particular choice of model parameters. We can make small perturbations to the parameters and check how much the resulting analysis changes. For example, we might have a collision avoidance system whose safety is influenced by the pilot response time to its resolution advisories.⁴⁰ If the probability of a mid-air collision is highly sensitive to particular parameter settings for the pilot response model, then it would suggest that additional analysis and data collection may be merited. If the probability of a mid-air collision is not highly sensitive, we can be more justified in proceeding with the learned parameters. Sensitivity will be discussed in more detail in section 11.3.1. A notional example is illustrated in figure 2.20.

2.6 Summary

- To accurately model a system, we need to build models of the agent, environment, and sensor.
- The general process for creating a model involves selecting a model class, learning the parameters of the model, and validating the model.
- Probability distributions are a common type of model class that assigns probabilities to different outcomes.
- We can learn the parameters of from data using maximum likelihood estimation or Bayesian estimation.

⁴⁰J. P. Chryssanthacopoulos and M. J. Kochenderfer, "Collision Avoidance System Optimization with Probabilistic Pilot Response Models," in *American Control Conference (ACC)*, 2011.

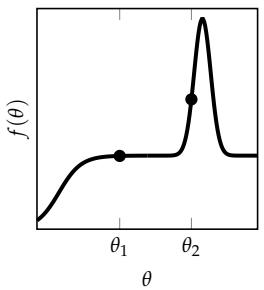


Figure 2.20. Sensitivity analysis of a model parameter θ with respect to downstream quantity of interest $f(\theta)$. If our learned parameter is θ_1 , we may be more confident in our downstream quantity compared to θ_2 where there is much greater sensitivity. A small perturbation to θ_1 is unlikely to change $f(\theta)$, but a small perturbation to θ_2 might.

- For systems that operate in environments with other agents, it is important to incorporate models of these agents into the environment model.
- We can validate models using test data or expert knowledge.

2.7 Exercises

Exercise 2.1. A common continuous distribution is the *uniform distribution* $\mathcal{U}(a, b)$, which has the following probability density:

$$p(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$$

where a and b are constants. Suppose we have a uniform distribution with $a = 1$ and $b = 5$, what is the probability that we sample a value between 2 and 4? What is the probability that we sample a value between 4 and 100?

Solution:

$$\begin{aligned} p(2 \leq x \leq 4) &= \int_2^4 \frac{1}{5-1} dx \\ &= \frac{1}{2} \end{aligned}$$

$$\begin{aligned} p(4 \leq x \leq 100) &= \int_4^5 \frac{1}{5-1} dx + \int_5^{100} 0 dx \\ &= \frac{1}{4} \end{aligned}$$

Exercise 2.2. Suppose we have a variable Z distributed according to a uniform distribution $\mathcal{U}(0, 2)$. We apply the following differentiable and invertible function to Z to obtain a new variable X :

$$x = z^{1/5}$$

What is the formula for the density $p_Z(z)$? What is the formula for the density $p_X(x)$?

Solution: The formula for $p_Z(z)$ is

$$p_Z(z) = \begin{cases} 0.5 & \text{if } 0 \leq z \leq 2 \\ 0 & \text{otherwise} \end{cases} \quad (2.25)$$

We can use equation (2.6) to calculate $p_X(x)$ with $g(x) = x^5$ and $g'(x) = 5x^4$ as follows:

$$p_X(x) = p_Z(x^5) |5x^4|$$

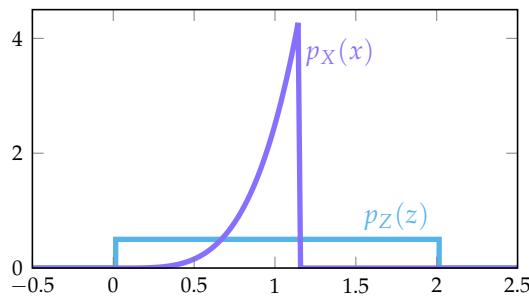
Since $p_Z(z)$ is only nonzero when its input is between 0 and 2, $p_X(x)$ will only be nonzero when x^5 is between 0 and 2. In other words, it will be nonzero when x is between 0 and $2^{1/5} \approx 1.15$. Specifically, it will be equal to

$$0.5|5x^4| = 2.5x^4$$

Therefore, the formula for $p_X(x)$ is

$$p_X(x) = \begin{cases} 2.5x^4 & \text{if } 0 \leq x \leq 2^{1/5} \\ 0 & \text{otherwise} \end{cases}$$

The following figure shows the results.



Exercise 2.3. We can use samples from a uniform distribution to generate samples from any distribution for which we know the cumulative distribution function $F(x)$ using a process called *inverse transform sampling*. We first draw a set of samples of a random variable Z where $Z \sim U(0, 1)$. We then transform these samples to obtain a new variable X using the following transformation:

$$x = F^{-1}(z)$$

where F^{-1} is the inverse of the cumulative distribution function for the distribution we would like to draw samples from. Show that new variable X has cumulative distribution function F . Note that the derivative of the cumulative distribution function $F(x)$ for a random variable is equal to the probability density function $f(x)$ of the random variable.

Solution: We can apply equation (2.6) with $g(x) = F(x)$ as follows:

$$\begin{aligned} p_X(x) &= p_Z(F(x))|F'(x)| \\ &= (1)|F'(x)| \\ &= f(x) \end{aligned}$$

The probability density function of X is equal to $f(x)$, and therefore, the cumulative distribution function of X is equal to $F(x)$.

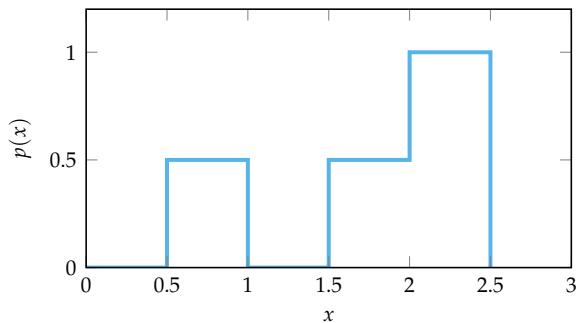
Exercise 2.4. Suppose we have a multivariate Gaussian distribution over two variables (x_1 and x_2) with mean at the origin and the following covariance matrix:

$$\Sigma = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$$

If x_1 is positive, is x_2 more likely to be positive or negative?

Solution: Because the off-diagonal element of the covariance matrix (0.5) is positive, there is a positive correlation between x_1 and x_2 . Therefore, if x_1 is positive, x_2 is more likely to also be positive.

Exercise 2.5. Specify three components (and their corresponding weights) of a mixture model that corresponds to the probability density shown below. Hint: use uniform distributions for the components of the mixture.



Solution: There are multiple possible answers. One answer is $\mathcal{U}(0.5, 1)$ with weight 0.25, $\mathcal{U}(1.5, 2)$ with weight 0.25, and $\mathcal{U}(2, 2.5)$ with weight 0.5.

Exercise 2.6. Consider a continuous random variable X that follows the *Laplace distribution* parameterized by μ and b , with density

$$p(x | \mu, b) = \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right)$$

Compute the maximum likelihood estimates of the parameters of a Laplace distribution given a data set D of m independent observations $x_{1:m}$. Note that $\partial|u|/\partial x = \text{sign}(u)\partial u/\partial x$, where the sign function returns the sign of its argument.

Solution: Since the observations are independent, we can write the log-likelihood function as the summation:

$$\begin{aligned}\ell(\mu, b) &= \sum_{i=1}^m \log \left[\frac{1}{2b} \exp\left(-\frac{|x_i - \mu|}{b}\right) \right] \\ &= -\sum_{i=1}^m \log 2b - \sum_{i=1}^m \frac{|x_i - \mu|}{b} \\ &= -m \log 2b - \frac{1}{b} \sum_{i=1}^m |x_i - \mu|\end{aligned}$$

To obtain the maximum likelihood estimates of the true parameters μ and b , we take the partial derivatives of the log-likelihood with respect to each of the parameters, set them to zero, and solve for each parameter. First, we solve for $\hat{\mu}$:

$$\begin{aligned}\frac{\partial}{\partial \mu} \ell(\mu, b) &= \frac{1}{\hat{b}} \sum_{i=1}^m \text{sign}(x_i - \mu) \\ 0 &= \frac{1}{\hat{b}} \sum_{i=1}^m \text{sign}(x_i - \hat{\mu}) \\ 0 &= \sum_{i=1}^m \text{sign}(x_i - \hat{\mu}) \\ \hat{\mu} &= \text{median}(x_{1:m})\end{aligned}$$

Now, solving for \hat{b} :

$$\begin{aligned}\frac{\partial}{\partial b} \ell(\mu, b) &= -\frac{m}{b} + \frac{1}{b^2} \sum_{i=1}^m |x_i - \hat{\mu}| \\ 0 &= -\frac{m}{\hat{b}} + \frac{1}{\hat{b}^2} \sum_{i=1}^m |x_i - \hat{\mu}| \\ \frac{m}{\hat{b}} &= \frac{1}{\hat{b}^2} \sum_{i=1}^m |x_i - \hat{\mu}| \\ \hat{b} &= \frac{1}{m} \sum_{i=1}^m |x_i - \hat{\mu}|\end{aligned}$$

Thus, the maximum likelihood estimates for the parameters of a Laplace distribution are $\hat{\mu}$, the median of the observations, and \hat{b} , the mean of absolute deviations from the median.

Exercise 2.7. Consider the distribution in example 2.5, but assume that the mean and variance both depend on the input x . Specifically, the mean and variance are both functions parameterized by θ such that

$$p_\theta(y | x) = \mathcal{N}(y | \mu_\theta(x), \sigma_\theta^2(x))$$

Show that the finding the maximum likelihood estimate for the parameters is equivalent to the following optimization problem:

$$\hat{\theta} = \arg \min_{\theta} \sum_{i=1}^m \left[\frac{(y_i - \mu_{\theta}(x_i))^2}{\sigma_{\theta}^2(x_i)} + \log(\sigma_{\theta}^2(x_i)) \right]$$

Solution: We follow a similar process to example 2.5, but we keep terms with the variance in the equation:

$$\begin{aligned}\hat{\theta} &= \arg \max_{\theta} \sum_{i=1}^m \log p_{\theta}(y_i | x_i) \\ &= \arg \max_{\theta} \sum_{i=1}^m \log \mathcal{N}(y_i | \mu_{\theta}(x_i), \sigma_{\theta}^2(x_i)) \\ &= \arg \max_{\theta} \sum_{i=1}^m \log \frac{1}{\sqrt{2\pi\sigma_{\theta}^2(x_i)}} \exp \left(-\frac{(y_i - \mu_{\theta}(x_i))^2}{2\sigma_{\theta}^2(x_i)} \right) \\ &= \arg \max_{\theta} \sum_{i=1}^m \left[\log(1) - \log(\sqrt{2\pi}) - \frac{1}{2} \log(\sigma_{\theta}^2(x_i)) - \frac{(y_i - \mu_{\theta}(x_i))^2}{2\sigma_{\theta}^2(x_i)} \right] \\ &= \arg \max_{\theta} \sum_{i=1}^m \left[-\frac{1}{2} \log(\sigma_{\theta}^2(x_i)) - \frac{(y_i - \mu_{\theta}(x_i))^2}{2\sigma_{\theta}^2(x_i)} \right] \\ &= \arg \min_{\theta} \sum_{i=1}^m \left[\frac{(y_i - \mu_{\theta}(x_i))^2}{\sigma_{\theta}^2(x_i)} + \log(\sigma_{\theta}^2(x_i)) \right]\end{aligned}$$

Exercise 2.8. Suppose that Austin is playing baseball as the designated hitter. Before we see him play, we start with an independent uniform prior over the probability that he gets a hit per at bat. We observe his first three at bats, with one of them resulting in a hit. What is the probability that we assign to him getting a hit in his next at bat?

Solution: We denote the probability of getting a hit as θ . Since we start with a uniform prior Beta(1, 1) and observe one hit and two outs, our posterior is then Beta(1 + 1, 1 + 2) = Beta(2, 3). We want to compute the probability of a basket as follows:

$$P(\text{hit}) = \int P(\text{hit} | \theta) \text{Beta}(\theta | 2, 3) d\theta = \int \theta \text{Beta}(\theta | 2, 3) d\theta$$

This expression is just the expectation (or mean) of a beta distribution, which gives us $P(\text{hit}) = 2/5$.

Exercise 2.9. Suppose we want to fit a model to the following data set of (x, y) pairs:

$$\mathcal{D} = \{(-4, 6), (-3, 3), (0, 0), (1, 4), (4, 4)\}$$

We select a model class of the form $p_{\theta}(y | x) = \mathcal{N}(y | \theta_1|x| + \theta_2, 1^2)$ where $\theta = [\theta_1, \theta_2]$ represents the parameters of the model class. We are considering the following two parameter vectors:

$$\theta_c = [1, 1]$$

$$\theta_d = [0, 2]$$

We want to select the parameter vector with a higher probability density according to the posterior distribution $p(\theta | \mathcal{D})$. For each of the following possible prior distributions, determine whether we should select θ_c or θ_d .

$$1. \ p(\theta) = \begin{cases} 1/16 & -1 \leq \theta_1 \leq 3 \text{ and } 0 \leq \theta_2 \leq 4 \\ 0 & \text{otherwise} \end{cases}$$

$$2. \ p(\theta) = \mathcal{N}(\theta | [0, 2], 0.25^2 I)$$

Solution: Using Bayes' rule, we have

$$p(\theta | \mathcal{D}) = \frac{p(\mathcal{D} | \theta)p(\theta)}{\int p(\mathcal{D} | \theta)p(\theta)d\theta}$$

The denominator is a constant, so we only need to see which potential value for θ has a higher numerator.

$$1. \ p(\theta_c | \mathcal{D}) \propto \left(\prod_{(x,y) \sim \mathcal{D}} \mathcal{N}(y | (1)x + 1, 1^2) \right) \left(\frac{1}{16} \right) \approx 1.157 \times 10^{-5}$$

$$p(\theta_d | \mathcal{D}) \propto \left(\prod_{(x,y) \sim \mathcal{D}} \mathcal{N}(y | (0)x + 2, 1^2) \right) \left(\frac{1}{16} \right) \approx 3.185 \times 10^{-10}$$

Therefore, we should select θ_c .

$$2. \ p(\theta_c | \mathcal{D}) \propto \left(\prod_{(x,y) \sim \mathcal{D}} \mathcal{N}(y | (1)x + 1, 1^2) \right) (\mathcal{N}([1, 1] | [0, 2], 0.25^2 I)) \approx 5.304 \times 10^{-11}$$

$$p(\theta_d | \mathcal{D}) \propto \left(\prod_{(x,y) \sim \mathcal{D}} \mathcal{N}(y | (0)x + 2, 1^2) \right) (\mathcal{N}([0, 2] | [0, 2], 0.25^2 I)) \approx 1.298 \times 10^{-8}$$

Therefore, we should select θ_d .

Exercise 2.10. Suppose we are trying to guess which type of cookie Robert will eat for dessert. The options are chocolate chip, pumpkin chocolate chip, peanut butter, and snickerdoodle. Robert's utility for each cookie is as follows:

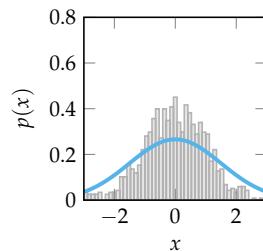
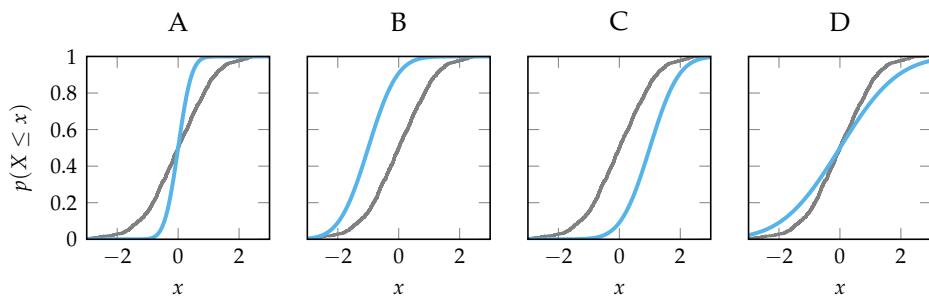
Cookie	Utility
chocolate chip	5
pumpkin chocolate chip	2
peanut butter	4
snickerdoodle	3

We use a softmax response policy with precision parameter $\lambda = 1$ to model Robert's choice. What is the probability that Robert will choose a snickerdoodle?

Solution: Let a_1 be the action in which Robert chooses chocolate chip, a_2 be the action for pumpkin chocolate chip, a_3 be the action for peanut butter, and a_4 be the action for snickerdoodle. The probability that Robert will choose a snickerdoodle is

$$\begin{aligned}\pi(a_4) &= \frac{\exp(\lambda u_4)}{\sum_{i=1}^4 \exp(\lambda u_i)} \\ &= \frac{\exp(3)}{\exp(5) + \exp(2) + \exp(4) + \exp(3)} \\ &\approx 0.087\end{aligned}$$

Exercise 2.11. Consider the probability density comparison plot shown in the margin. The gray histogram shows the probability density of the data, and the blue curve shows the probability density of a model. Which of the cumulative distribution plots below shows the same data and model distributions as the probability density comparison plot?



Solution: D. As we move from left to right in the probability density plot, the area under the blue curve increases faster than the area under the gray histogram at first. As we move to the right, the area under the blue curve increases more slowly than the area under the gray histogram. This behavior is captured in the cumulative distribution plot labeled D.

Exercise 2.12. Because the KL divergence is not symmetric, we should be careful when selecting the order of the arguments. For example, consider the case in which the data comes from distribution $q(\cdot)$ and the model is represented as $p(\cdot)$. In this scenario, the KL divergence from p to q is written as

$$D_{\text{KL}}(q \parallel p) = \int q(x) \log \frac{q(x)}{p(x)} dx$$

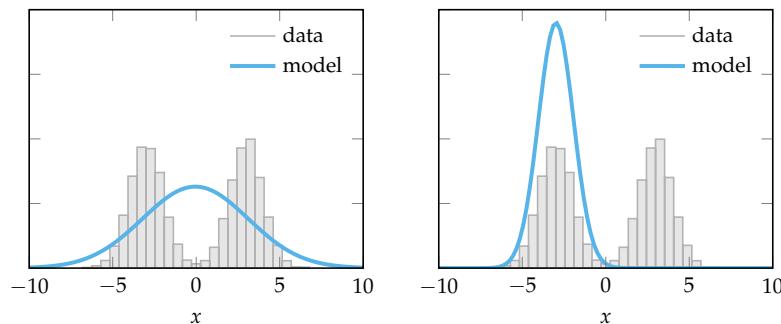
and is referred to as the forward KL divergence. Because the density p appears in the denominator of the fraction, the forward KL divergence will be high if p assigns low probability density to regions where q assigns high probability density. Therefore, using the forward KL divergence to select model parameters tends to result in a model that prioritizes covering all high-density regions of the data.

In contrast, the KL divergence from q to p is written as

$$D_{\text{KL}}(p \parallel q) = \int p(x) \log \frac{p(x)}{q(x)} dx$$

and is referred to as the reverse KL divergence. Because the density q appears in the denominator of the fraction, the reverse KL divergence will be high if q assigns low probability density to regions where p assigns high probability density. Therefore, if the model class used to represent p is not expressive enough to capture all the high-density regions of the data, using the reverse KL divergence to select model parameters will result in a model that prioritizes covering a single high-density region rather than spreading across all high-density regions.

The plots below show a Gaussian model with parameters selected to minimize the KL divergence with respect to the data. One plot minimizes the forward KL divergence, and the other minimizes the reverse KL divergence. Based on the intuition provided, which plot corresponds to the forward KL divergence and which corresponds to the reverse KL divergence?



Solution: The plot on the left corresponds to the forward KL divergence, and the plot on the right corresponds to the reverse KL divergence. The plot on the left prioritizes covering all high-density regions of the data, while the plot on the right prioritizes covering a single high-density region.

Exercise 2.13. Suppose we want to validate the safety of an autonomous vehicle, and we find that the probability of an accident is very sensitive to the model parameters that specify the road conditions. What does this sensitivity tell us about the validation results?

Solution: Since the probability of an accident is very sensitive to the model parameters, the validation results may be unreliable if we are uncertain about our model parameters. Therefore, we may want to gather more data and perform more validation on the model parameters to ensure that the model is accurate.

3 *Property Specification*

In the previous chapter, we focused on creating an accurate model of the system. The final step in defining a validation problem is to formalize the operating requirements of the system as a specification, which is a precise mathematical expression that defines the objectives of a system. Specifications are often derived from metrics, which map the performance of a system to a real number. We begin by discussing common metrics used to measure the performance of stochastic systems. We also discuss how to create composite metrics that capture trade-offs between different performance objectives. We then show how to write specifications as logical formulas using propositional logic, first-order logic, and temporal logic. Finally, we discuss a special case of a temporal specification called a reachability specification and show how to convert temporal logic specifications into reachability specifications.

3.1 *Properties of Systems*

We describe the behavior of a system using *metrics* and *specifications*. A metric is a function that maps system behavior to a real number. For example, a common metric used to evaluate aircraft collision avoidance systems is the miss distance between two aircraft. A specification is a function that maps system behavior to a Boolean value. Therefore, specifications are always either true or false. For example, a specification for the grid world system might be to reach the goal without hitting an obstacle.

Sometimes specifications can be derived from metrics. For example, given a metric that measures the probability of collision for an aircraft collision avoidance system, we can create a specification that requires the probability of collision to be less than a certain threshold. We can also derive metrics from specifications.

Using the grid world specification, we could define a metric that measures the distance between the agent and the goal or obstacle.

We use metrics or specifications to evaluate individual trajectories, sets of trajectories, or probability distributions over trajectories. The miss distance between two aircraft can be used to measure the performance of an aircraft collision avoidance system in a single encounter scenario (figure 3.1), and the net return can be used to measure the performance of a single outcome of a financial trading strategy over time. We can also create metrics or specifications that operate over a set of trajectories. For example, we can compute the average miss distance or net gain over a set of possible trajectories or specify a threshold on the number of trajectories that result in a collision. The remainder of this chapter discusses techniques to formally express metrics and specifications.

3.2 Metrics for Stochastic Systems

For stochastic systems, we often compute metrics over the full distribution of trajectories. Given a function $f(\tau)$ that maps an individual trajectory τ to a real-valued metric, we are interested in summarizing the distribution over the output of $f(\tau)$ (figure 3.2). The remainder of this section outlines several metrics used to summarize distributions.

3.2.1 Expected Value

A common metric used to summarize a distribution is its *expected value*. The expected value represents the average output of a function given a distribution over its inputs. It is defined as

$$\mathbb{E}_{\tau \sim p(\cdot)}[f(\tau)] = \int f(\tau)p(\tau) d\tau \quad (3.1)$$

where $p(\tau)$ is the probability distribution over trajectories. While it is not always possible to evaluate the expected value analytically, we can estimate it using a variety of techniques such as the sampling-based methods discussed in chapter 7.

The expected value of a binary metric represents the probability of the metric being 1. For instance, consider a binary metric $f(\tau)$ that evaluates to 1 if an the agent hits an obstacle and 0 otherwise. The expected value of this metric is the probability that the agent hits an obstacle. In general, the expected value of a

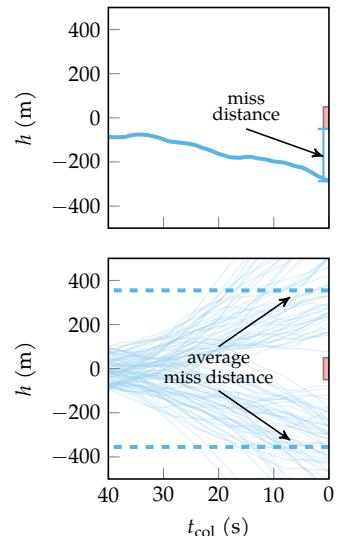


Figure 3.1. Example of a metric for an aircraft collision avoidance system over an individual trajectory (top) and over a set of trajectories (bottom).

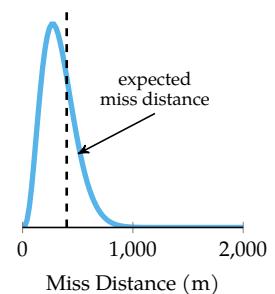


Figure 3.2. Distribution over the miss distance metric for an aircraft collision avoidance system. We can summarize this distribution with another metric such as the expected value of the miss distance.

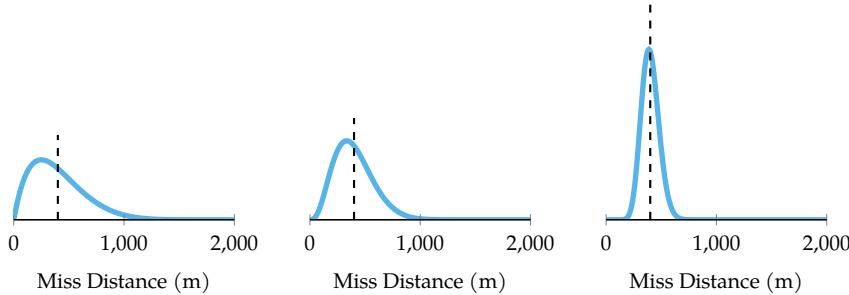


Figure 3.3. Three distributions over the miss distance metric for an aircraft collision avoidance system. While all distributions have the same mean, they have different variances (decreasing from left to right). The distribution with the lowest variance is most likely to operate safely.

binary metric derived from a specification is the probability that a randomly sampled trajectory will satisfy the specification. We could also derive a high-level specification from this probability by requiring that the probability of satisfying the specification is greater than a certain threshold.¹

3.2.2 Variance

Another common summary metric is the *variance*, which measures the spread of the distribution. The variance of a metric $f(\tau)$ is defined as

$$\text{Var}_{\tau \sim p(\cdot)}[f(\tau)] = \mathbb{E}_{\tau \sim p(\cdot)}[(f(\tau) - \mathbb{E}_{\tau \sim p(\cdot)}[f(\tau)])^2] \quad (3.2)$$

Intuitively, the variance measures how much the metric $f(\tau)$ deviates from its expected value. A low variance indicates that the metric tends to be consistent across different trajectories, while a high variance indicates that the metric varies significantly. It is important to consider both the expected value and variance of a metric when evaluating system performance (figure 3.3).

3.2.3 Value at Risk

When we are concerned with safety, we may want to use more conservative metrics that focus on worst-case outcomes. One such metric is the *value at risk* (*VaR*). Suppose we have a metric $f(\tau)$ for individual trajectories in which higher values indicate worse outcomes. This type of metric is often referred to as a *risk metric*. The VaR is the highest risk value that $f(\tau)$ is guaranteed not to exceed with probability α , which corresponds to the α -quantile of the distribution. For a particular value of α , a higher VaR indicates a more risky system.

¹ H. Hansson and B. Jonsson, "A Logic for Reasoning about Time and Reliability," *Formal Aspects of Computing*, vol. 6, pp. 512–535, 1994.

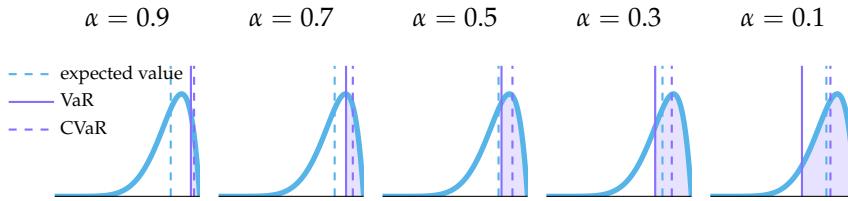


Figure 3.4. Effect of α on VaR and CVaR. Higher values for α correspond to more conservative risk estimates.

3.2.4 Conditional Value at Risk

Another common metric derived from VaR is the *conditional value at risk* (CVaR),² which is the expected value of the metric $f(\tau)$ given that it exceeds the VaR:

$$\text{CVaR}_\alpha[f(\tau)] = \mathbb{E}_{\tau \sim p(\cdot)}[f(\tau) | f(\tau) \geq \text{VaR}_\alpha[f(\tau)]] \quad (3.3)$$

In other words, CVaR is the expected value of the $(1 - \alpha)$ -fraction of worst-case outcomes. A higher CVaR indicates that the system is more likely to perform poorly in the worst-case scenarios. Example 3.1 shows the VaR and CVaR of a risk metric for an aircraft collision avoidance system. Higher values of α push the VaR closer to the worst-case outcome and correspond to more conservative risk estimates. As α approaches 1, the CVaR approaches the risk of the worst-case outcome. As α approaches 0, the CVaR approaches the expected value of the risk metric (figure 3.4).

3.3 Composite Metrics

In many real-world settings, we must select one of several system designs or strategies for final deployment, and metrics allow us to make an informed decision. For example, we might compare the performance of two aircraft collision avoidance systems by computing the probability of collision over a set of aircraft encounters for each system. In these cases, we are often concerned with multiple metrics. For example, an aircraft collision avoidance system should minimize collisions while issuing a small number of alerts to pilots, and a financial trading strategy may aim to maximize return while minimizing risk.

It is often the case that multiple metrics describing system performance are at odds with one another, and some system designs may perform well on one metric but poorly on another. For instance, an aircraft collision avoidance system that

² The conditional value at risk is also known as the *mean excess loss*, *mean shortfall*, and *tail value at risk*. R.T. Rockafellar and S. Uryasev, “Optimization of Conditional Value-at-Risk,” *Journal of Risk*, vol. 2, pp. 21–42, 2000. It is also a kind of *coherent risk measure*, which means that it satisfies some additional mathematical properties. Another coherent risk measure, not discussed here, is the *entropic value at risk*. A. Ahmadi-Javid, “Entropic Value-At-Risk: A New Coherent Risk Measure,” *Journal of Optimization Theory and Applications*, vol. 155, no. 3, pp. 1105–1123, 2011.

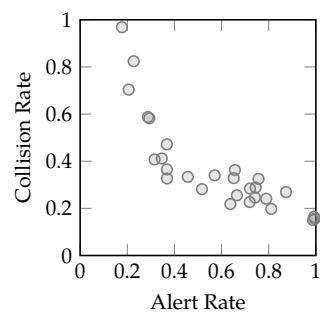
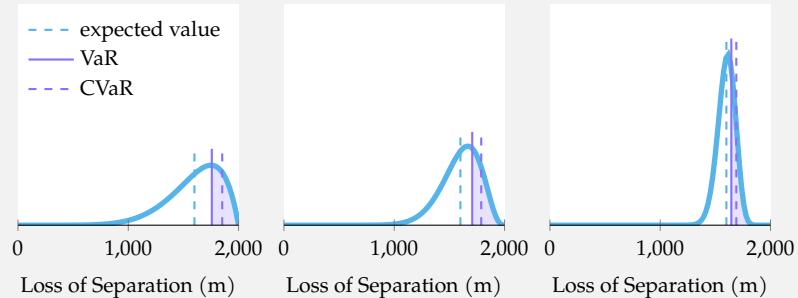


Figure 3.5. Tradeoff between the alert rate and collision rate for an aircraft collision avoidance system. Each point represents a different system design.

Suppose a desired separation for the aircraft in an aircraft collision avoidance environment is 2,000 m. We can define a risk metric $f(\tau)$ to summarize the loss of separation as 2,000 m minus the miss distance. A higher loss of separation indicates higher risk. The plots below show the VaR and CVaR with $\alpha = 0.7$ for the loss of separation metric for three different distributions over outcomes.



Although all three distributions have the same expected value, the VaR and CVaR decrease as we move from the left figure to right figure. The distribution in the figure on the right is the least risky because it has better worst-case outcomes.

Example 3.1. VaR and CVaR for the loss of separation metric for an aircraft collision avoidance system.

minimizes the number of collisions may also increase the number of alerts issued to pilots, while one that minimizes alerts may increase the number of collisions (figure 3.5). In such cases, we can combine multiple metrics into a single *composite metric* that captures the trade-offs between different objectives.

We can compare systems with multiple metrics using the concept of Pareto optimality. A system design is *Pareto optimal*³ if we cannot improve one metric without worsening another. Given a set of system designs, the *Pareto frontier* consists of the subset of designs that are Pareto optimal. The Pareto frontier illustrates the trade-offs between metrics. Figure 3.6 shows the Pareto frontier for the aircraft collision avoidance systems shown in figure 3.5. *Composite metrics* allow system designers to select a single point on the Pareto frontier.

3.3.1 Weighted Metrics

Weighted metrics combine multiple metrics using a vector of weights that reflect the relative importance of each metric. Suppose we have a set of metrics $f_1(\tau), f_2(\tau), \dots, f_n(\tau)$ that we wish to combine into a single metric. The most basic weighted metric is the *weighted sum*, which is defined as

$$f(\tau) = \sum_{i=1}^n w_i f_i(\tau) = \mathbf{w}^\top \mathbf{f}(\tau) \quad (3.4)$$

where $\mathbf{w} = [w_1, \dots, w_n]$ is a vector of weights and $\mathbf{f}(\tau) = [f_1(\tau), \dots, f_n(\tau)]$ is a vector of metrics. The weighted sum allows us to balance the trade-offs between different metrics by adjusting the weights, and each set of weights will correspond to a point or set of points on the Pareto frontier.

3.3.2 Goal Distance Metrics

Another way to combine metrics is to compute the L_p norm⁴ between $\mathbf{f}(\tau)$ and a goal point:

$$f(\tau) = \|\mathbf{f}(\tau) - f_{\text{goal}}\|_p \quad (3.5)$$

where f_{goal} is typically selected to be the *utopia point*. The utopia point is the point in the space of metrics that represents the best possible outcome for each metric. While the utopia point is often unattainable, it provides a reference point for comparing different system designs. Figure 3.7 shows an example of the goal metric for the aircraft collision avoidance problem.

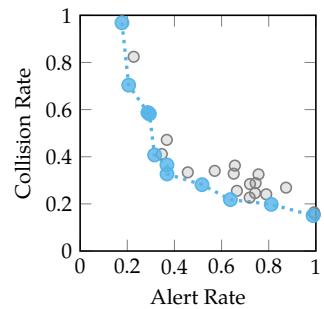


Figure 3.6. Pareto frontier for a set of aircraft collision avoidance system designs. The points that comprise the Pareto frontier are highlighted in blue.

³ Pareto optimality is a topic that was originally explored in the field of economics. It is named after Italian economist Vilfredo Pareto (1848–1923).

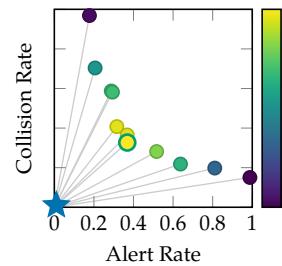
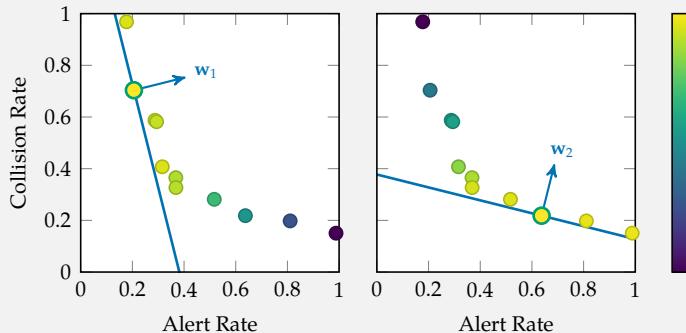


Figure 3.7. Composite metric for an aircraft collision avoidance system using the L_2 norm between the point and the goal point (blue star). The goal point is the utopia point of no alerts and no collisions. The color of each point represents the value of the composite metric with the selected point highlighted in green.

⁴ An overview of the L_p norm operator is provided in appendix B.

Suppose we want to create a composite metric for an aircraft collision avoidance system that balances the alert rate and collision rate. Using the weighted sum method, we define the composite metric as the weighted sum of the alert rate and collision rate. Selecting a weight vector then allows us to choose a point on the Pareto frontier. The plots below show the Pareto frontier for two different weight vectors. The first weight vector ($\mathbf{w}_1 = [0.8, 0.2]$) gives more weight to minimizing the alert rate, while the second weight vector ($\mathbf{w}_2 = [0.2, 0.8]$) gives more weight to minimizing the collision rate. The points are colored according to the value of the composite metric.



The weight vector will be perpendicular to the Pareto frontier at the best design point. The weight vector \mathbf{w}_1 is shown in blue for the first design point and \mathbf{w}_2 is shown in blue for the second design point. The best design points are highlighted in green.

Example 3.2. Using the weighted sum composite metric to select an aircraft collision avoidance system design along the Pareto frontier.

The *weighted exponential sum* is a composite metric that combines the weighted sum and goal metrics as follows:

$$f(\tau) = \sum_{i=1}^n w_i (\mathbf{f}_i(\tau) - f_{\text{goal}})^p \quad (3.6)$$

where $p \geq 1$ is an exponent similar to that used in L_p norms. The weights w_i must be positive and sum to 1. The weighted exponential sum allows us to balance the trade-offs between different metrics while also considering the distance to the utopia point. Other more sophisticated weighting methods such as the weighted min-max metric and the exponential weighted metric build on these ideas.⁵

3.3.3 Preference Elicitation

Creating a composite metric using weights requires us to specify the relative importance of each metric. However, even domain experts may have difficulty translating their preferences to a set of precise numerical weights. *Preference elicitation* allows us to infer a set of weights based on expert responses to a set of preference queries. For example, we might present a domain expert with a pairwise query containing the metrics of two possible system designs and ask them to select the preferred design. By repeating this process for multiple different pairwise queries of system designs, we can infer the weight that the expert assigns to each metric.

In this section, we focus on inferring the weights of a weighted sum composite metric using pairwise queries. There are other schemes for eliciting preferences, such as ranking multiple system designs, but pairwise queries have been shown to pose minimal cognitive burden on the expert.⁶ We will also restrict ourselves to weight vectors with positive entries that sum to 1. Figure 3.8 shows the space of possible weights for the aircraft collision avoidance example.

Suppose we query the expert with a pair of metric vectors \mathbf{f}_1 and \mathbf{f}_2 and find that the expert prefers \mathbf{f}_2 to \mathbf{f}_1 . For the weighted sum metric to be consistent with the preference, we must select a weight vector \mathbf{w} such that

$$\mathbf{w}^\top \mathbf{f}_1 < \mathbf{w}^\top \mathbf{f}_2 \quad (3.7)$$

$$\mathbf{w}^\top (\mathbf{f}_1 - \mathbf{f}_2) < 0 \quad (3.8)$$

⁵ For more information on composite metrics, see T.W. Athan and P.Y. Papalambros, "A Note on Weighted Criteria Methods for Compromise Solutions in Multi-Objective Optimization," *Engineering Optimization*, vol. 27, no. 2, pp. 155–176, 1996.

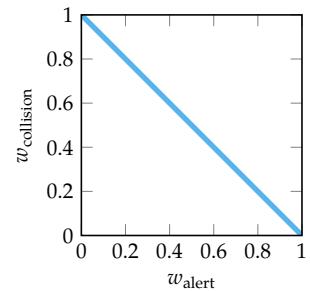


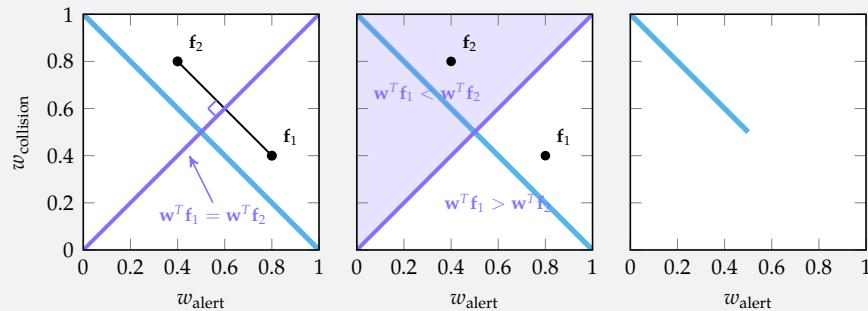
Figure 3.8. The space of possible weights for the aircraft collision avoidance weighted sum metric. The line represents all weights that sum to 1.

⁶ V. Conitzer, "Eliciting Single-Peaked Preferences Using Comparison Queries," *Journal of Artificial Intelligence Research*, vol. 35, pp. 161–191, 2009.

where we assume that lower values for the composite metric are preferable.⁷ In effect, the response to the query further constrains the space of possible weight vectors (example 3.3).

⁷ If higher values are preferable, the inequality in equation (3.8) should be reversed.

Suppose we want to infer the weights for a composite metric that combines the alert rate and collision rate for an aircraft collision avoidance system. When we query a domain expert or stakeholder with system designs $\mathbf{f}_1 = [0.8, 0.4]$ and $\mathbf{f}_2 = [0.4, 0.8]$, we find that the expert prefers \mathbf{f}_2 to \mathbf{f}_1 . In other words, the expert prefers the system design with the higher alert rate and lower collision rate. Since the weight vector must be consistent with this preference (equation (3.8)), we can further constrain the space of possible weight vectors as shown in the figure below.



The purple shaded region in the center plot shows the space of possible weight vectors consistent with the expert's preference. The plot on the right shows the space of possible weight vectors consistent with the expert's preference and the constraint that the weights must sum to 1. We can further refine the space of possible weight vectors by querying the expert with additional pairs of system designs.

By querying the expert with multiple pairs of system designs, we can iteratively refine the space of possible weight vectors (figure 3.9). To minimize the number of times we must query the expert, it is common to select pairs of system designs that will maximally reduce the space of possible weights. For example, one method is to select the query that comes closest to bisecting the space of possible weights.⁸ After querying the expert a desired number of times, we can

⁸ This method is known as Q-Eval. V.S. Iyengar, J. Lee, and M. Campbell, "Q-EVAL: Evaluating Multiple Attribute Items Using Queries," in *ACM Conference on Electronic Commerce*, 2001.

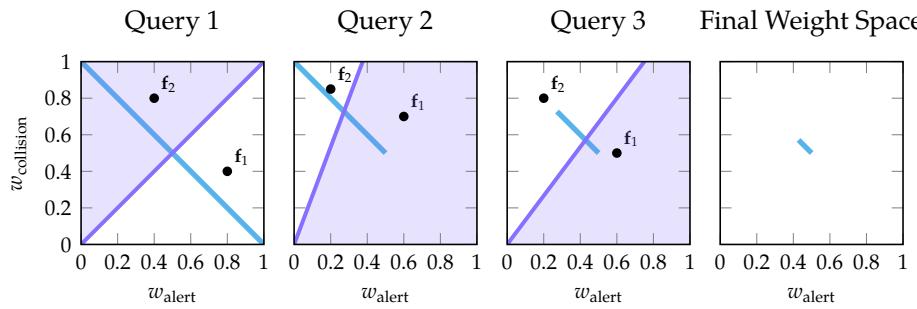


Figure 3.9. The effect of multiple preference queries on the space of possible weight vectors for the aircraft collision avoidance example. The blue lines show the space of possible weight vectors before obtaining the expert’s preference, and the purple shaded regions show the weight vectors consistent with the expert’s preference. The space of possible weight vectors before the next query is the intersection of these regions.

select a set of weights from the refined weight space to create a composite metric that reflects the expert’s preferences. While we could select any value for \mathbf{w} that is consistent with the expert’s responses, it is common to select the weight vector that maximally separates the system designs that were presented to the expert.

3.4 Logical Specifications

A *logical specification* ψ formally defines an operating requirement for a system using a logical formula. A *logical formula* is a precise expression that evaluates to either true or false. Logical specifications can be used to describe requirements for both individual trajectories and trajectory distributions. For example, a logical specification on an individual trajectory for an aircraft collision avoidance system might check whether the aircraft collide at any point in the trajectory. A logical specification over the entire distribution of aircraft collision avoidance trajectories might require that the probability of collision is less than a certain threshold. We can express logical formulas using several different types of logic. This section introduces two common types of logic.

3.4.1 Propositional Logic

Propositional logic constructs logical formulas by connecting propositions using logical operators.⁹ A *proposition* is a statement that is either true or false. The basic building block of propositional logic is an *atomic proposition*, which is a proposition that cannot be further decomposed. The two most basic logical expressions are

⁹ A detailed overview of propositional logic is provided by M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

Suppose we wish to express the following statement using propositional logic: “If the agent is in a safe state, then the agent is not in a collision state.” Let the variable S represent whether the agent is in a safe state and C represent whether the agent is in a collision state. The propositional logic statement is $S \rightarrow \neg C$ (read as “ S implies not C ”). In this statement, S and C are atomic propositions because they cannot be broken down further. The logical formula $S \rightarrow \neg C$ is itself a proposition that can be combined with other propositions to create more complex formulas.

Example 3.4. Constructing a propositional logic formula from a statement.

Expression	Explanation	Construction
$\neg P$	Negation (Not): Inverts a Boolean value.	—
$P \wedge Q$	Conjunction (And): Evaluates to <i>true</i> if both P and Q are <i>true</i> .	—
$P \vee Q$	Disjunction (Or): Evaluates to <i>true</i> if either P or Q are <i>true</i> .	$\neg(\neg P \wedge \neg Q)$
$P \rightarrow Q$	Implication : Evaluates to <i>true</i> unless P is <i>true</i> and Q is <i>false</i> .	$\neg P \vee Q$
$P \leftrightarrow Q$	Biconditional : Evaluates to <i>true</i> when both P and Q are equivalent.	$(P \wedge Q) \vee (\neg P \wedge \neg Q)$

Table 3.1. Propositional logic operators and their equivalent construction using negation and conjunction. The constructions build from previous expressions for convenience (e.g., the use of \vee in implication).

negation (“not”) and *conjunction* (“and”). All other logical expressions such as *disjunction* (“or”), *implication* (“if-then”), and *biconditional* (“if and only if”) can be constructed using negation and conjunction. Example 3.4 demonstrates the construction of a propositional logic formula from a statement.

Table 3.1 shows the propositional logic operators and their construction using negation and conjunction. We can describe propositional logic formulas using *truth tables*, which show the value of the formula as a function of its inputs. Figure 3.10 shows truth tables for each of the basic propositional logic operators. Logical operators can also be illustrated as *logic gates* (figure 3.11), which are fundamental building blocks for digital circuits.¹⁰ Example 3.5 implements the logical operators as functions in Julia.

3.4.2 First-Order Logic

First-order logic extends propositional logic by introducing the notion of *predicates* and *quantifiers*.¹¹ It uses *variables* to represent objects in a domain and *predicate functions* to evaluate propositions over these objects. For example, we could create a variable x to represent the state of an agent and a predicate function $P(x)$ that returns true if the agent is in a safe state and false otherwise. We combine

¹⁰ R. Page and R. Gamboa, *Essential Logic for Computer Science*. MIT Press, 2019.

¹¹ First-order logic is also known as predicate logic. M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

		P	Q	$P \wedge Q$	P		Q	$P \vee Q$
P	$\neg P$	false	false	false	false	false	false	false
false	true	false	true	false	false	true	true	true
true	false	true	false	false	true	false	true	true
		true	true	true	true	true	true	true

P	Q	$P \rightarrow Q$	P	Q	$P \leftrightarrow Q$
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	true	false	false
true	true	true	true	true	true

Figure 3.10. Truth tables for the propositional logic operators using atomic propositions P and Q . The truth tables show the outputs of each logical operator for all possible combinations of Boolean values for P and Q .

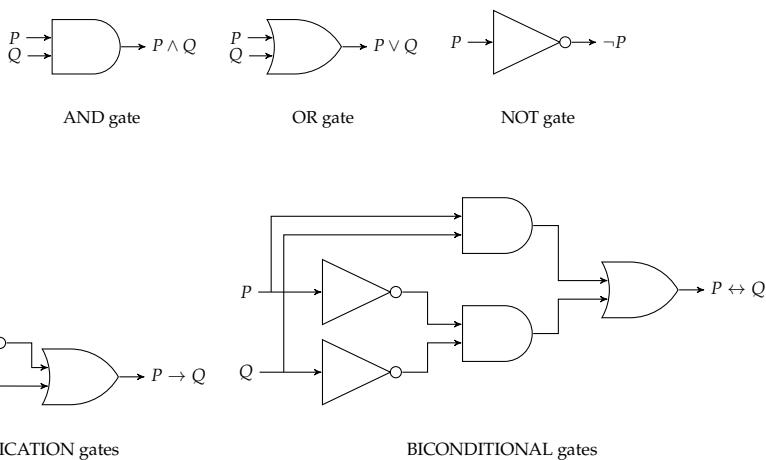


Figure 3.11. Logical operators represented using logic gates.

Consider two atomic propositions, P and Q . The basic operations of negation ($!$), conjunction ($\&\&$), and disjunction ($\|$) are already implemented in most programming languages including Julia. Implication $P \rightarrow Q$ can be defined as the operator \neg given the Boolean values of P and Q :

```
julia>  $\neg(P, Q) = !P \mid\mid Q$  # → produced by \longrightarrow<TAB>
→ (generic function with 1 method)
julia> P = true;
julia> Q = false;
julia> P → Q
false
```

For the biconditional $P \leftrightarrow Q$, we can use the \equiv sign:

```
julia> P = false;
julia> Q = false;
julia> P == Q
true
```

Example 3.5. Julia implementations of propositional logic operators.

predicates to create propositions using logical operators. For instance, if we have a predicate function $Q(x)$ that returns true if the agent is in a collision state, we can create the proposition $P(x) \rightarrow \neg Q(x)$ to express that the agent is not in a collision state when it is in a safe state.

Quantifiers allow us to evaluate propositions over a collection of variables. The *universal quantifier* \forall ("for all") returns true if all variables in the domain satisfy the proposition. The *existential quantifier* \exists ("there exists") returns true if at least one variable in the domain satisfies the proposition. These quantifiers allow us to create specifications over full system trajectories by setting the domain to be the set of all states in the trajectory. Example 3.6 demonstrates the use of quantifiers to define an obstacle avoidance specification over a trajectory.

3.5 Temporal Logic

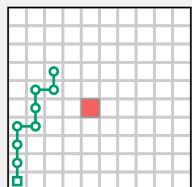
Temporal logic extends first-order logic to specify properties over time. It is particularly useful for specifying properties of dynamical systems because it allows us to describe how trajectories should evolve. This section outlines two common types of temporal logic.

Let x be a variable that represents the state of the agent in the grid world problem where we must avoid an obstacle (red), and define the domain \mathcal{X} as the set of states that comprise a particular trajectory. We define a predicate function $O(x)$ that evaluates to true if x is an obstacle state and false otherwise. To define a specification ψ_1 that states “for all states in the trajectory, the agent does not hit an obstacle,” we can use the formula:

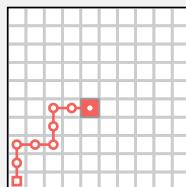
$$\psi_1 = \forall x \neg O(x)$$

The examples below show evaluations of ψ_1 for two different trajectories.

$$\psi_1 = \text{true}$$



$$\psi_1 = \text{false}$$



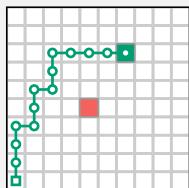
Example 3.6. Universal and existential quantifiers for an obstacle avoidance problem. The red region indicates an obstacle while the green region indicates the goal.

Suppose we also want the agent to reach a goal state while avoiding the obstacle. We can create an additional predicate $G(x)$ that evaluates to true if x is a goal state and false otherwise. We then create ψ_2 to represent the statement “for all states in the trajectory, the agent does not hit an obstacle and there exists a state in the trajectory in which the agent reaches the goal” using the following formula:

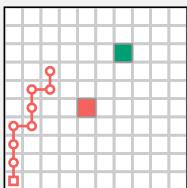
$$\psi_2 = (\forall x \neg O(x)) \wedge (\exists x G(x))$$

The examples below show evaluations of ψ_2 for two different trajectories.

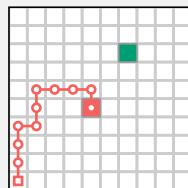
$$\psi_2 = \text{true}$$

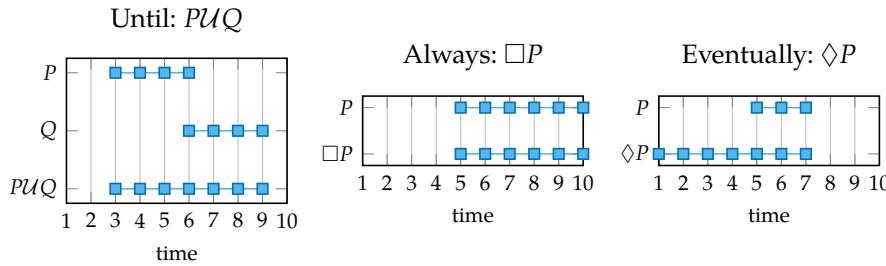


$$\psi_2 = \text{false}$$



$$\psi_3 = \text{false}$$





3.5.1 Linear Temporal Logic

Linear temporal logic (LTL) is a type of temporal logic that assumes a linear sequence of states, meaning that there is a single progression of states over time.¹² It introduces three main temporal operators.¹³ Given a proposition P , the *always* ($\square P$) operator specifies that P must be true at all time steps now and in the future. The *eventually* ($\diamond P$) operator requires that P be true now or at some point in the future. Given another proposition Q , the *until* ($P \cup Q$) operator specifies that P must be true at least until Q becomes true.

Table 3.2 outlines the three LTL operators and their construction, and algorithm 3.1 evaluates LTL specifications over the sequence of states in a trajectory. The until operator can be written using first-order logic quantifiers, and the other two operators build on the until operator. We use the \top symbol to indicate static truth. Figure 3.12 shows the values of these operators over a trajectory, and example 3.7 shows how to construct an LTL specification for the grid world problem.

Expression	Explanation	Construction
$P \cup Q$	Until: Q will be true at some time in the future and P is true at least until Q becomes true.	$\exists t (Q_t \wedge \forall t' (0 \leq t' < t) P_{t'})$
$\diamond P$	Eventually: P will be true at some time in the future.	$\top \cup P$
$\square P$	Always: P is true at every time in the future.	$\neg \diamond (\neg P)$

3.5.2 Signal Temporal Logic

Signal temporal logic (STL) extends LTL to specify properties over signals.¹⁴ A *signal* is a real-valued sequence of points in discrete time that represent the state

Figure 3.12. Examples of the binary temporal operator *until* and unary temporal operators *eventually* and *always*. The temporal operator is defined from time t to the end of the sequence. For the entire trajectory to satisfy the formula, the property must hold at time step 1.

¹² A. Pnueli, “The Temporal Logic of Programs,” in *Symposium on Foundations of Computer Science (SFCS)*, 1977. Computation tree logic (CTL) is another common temporal logic that operates over multiple future paths. A detailed overview is provided in chapter 6 of C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.

¹³ Other common operators include *next*, *weak until*, and *release*.

Table 3.2. LTL operators. The propositions P_t and Q_t represent whether P and Q are true at time t .

¹⁴ STL was first introduced in O. Maler and D. Nickovic, “Monitoring Temporal Properties of Continuous Signals,” in *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, 2004.

```

struct LTLspecification <: Specification
    formula # formula specified using SignalTemporalLogic.jl
end
evaluate(ψ::LTLspecification, τ) = ψ.formula([step.s for step in τ])

```

Algorithm 3.1. Definition of LTL specification. The formula is evaluated over the sequence of states in the trajectory starting at the first time step.

For a navigation problem, let ψ be the LTL property specification that states “*eventually reach the goal after passing through the checkpoint and always avoid the obstacle.*” First, we define the following predicate functions:

- $F(s_t)$: the state s at time t contains an obstacle
- $G(s_t)$: the state s at time t is the goal
- $C(s_t)$: the state s at time t is the checkpoint

The specification can be defined using LTL as follows:

$$\psi = \Diamond G \wedge \neg G \mathcal{U} C \wedge \Box \neg F$$

This formula requires that the agent reaches the goal ($\Diamond G$) but that the goal is not reached before the checkpoint ($\neg G \mathcal{U} C$). Additionally, the agent must always avoid obstacles ($\Box \neg F$). The figure in the caption shows an example trajectory that satisfies this specification. The following code constructs the LTL specification:

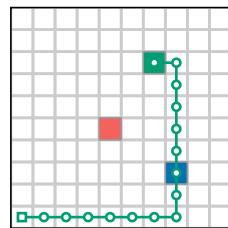
```

F = @formula st → st == [5, 5]
G = @formula st → st == [7, 8]
C = @formula st → st == [8, 3]
ψ = LTLspecification(@formula ♦(G) ∧ ∃(¬G, C) ∧ □(¬F))

```

Example 3.7. LTL formula for an obstacle avoidance problem where a blue checkpoint must be reached before the green goal while avoiding the red obstacle.

$$\psi = \text{true}$$



of a system over time.¹⁵ STL introduces two key extensions to LTL to handle real-valued signals. The first extension is the ability to specify properties over a time interval $[a, b]$. For example, we can write $\Diamond_{[a,b]} P$ to specify that P will eventually be true within the time interval $[a, b]$.¹⁶

The second extension is the introduction of predicates that map real-valued signals to truth values. Specifically, it introduces the predicate $\mu_c(s_t)$ that returns true if

$$\mu(s_t) > c \quad (3.9)$$

where $\mu(\cdot)$ is a real-valued function that operates on the state (example 3.8). Table 3.3 defines the specifications for the continuum world, inverted pendulum, and collision avoidance example problems using STL. Algorithm 3.2 provides a framework for evaluating STL specifications over a trajectory given a time interval.

Suppose we want to implement the following STL formula in code: “eventually the signal will be greater than 1.” We can use the `SignalTemporalLogic.jl` package to define the predicate μ and the formula ψ as follows:

```
julia> using SignalTemporalLogic
julia> τ = [-1.0, -3.2, 2.0, 1.5, 3.0, 0.5, -0.5, -2.0, -4.0, -1.5];
julia> μ = @formula st → st > 1.0;
julia> ψ = @formula ◊(μ);
julia> ψ(τ) # check if formula is satisfied
true
```

The formula is satisfied since the signal eventually becomes greater than 1.

```
struct STLSpecification <: Specification
    formula # formula specified using SignalTemporalLogic.jl
    I       # time interval (e.g. 3:10)
end
evaluate(ψ::STLSpecification, τ) = ψ.formula([step.s for step in τ[ψ.I]])
```

One benefit of expressing properties using STL is the ability to calculate a *robustness* metric using the specification.¹⁷ Robustness provides a quantitative measure of satisfaction. For example, the robustness of the predicate $\mu_c(s_t)$ is defined as

$$\rho(s_t, \mu_c) = \mu(s_t) - c \quad (3.10)$$

¹⁵ These points may be sampled at regular or irregular intervals from a continuous-time function.

¹⁶ When a time range is omitted, we assume the positive time path of $[0, \infty)$.

Example 3.8. Julia implementation of an STL formula.

Algorithm 3.2. Definition of an STL specification for an interval.

¹⁷ A. Donzé and O. Maler, “Robust Satisfaction of Temporal Logic over Real-Valued Signals,” in *International Conference on Formal Modeling and Analysis of Timed Systems*, 2010.

System	Property	Implementation
Continuum World	<p><i>“Reach the goal without hitting the obstacle”</i></p> <p>$G(s_t)$: s_t is in the goal region</p> <p>$F(s_t)$: s_t is in the obstacle region</p> <p>$\psi = \Diamond G \wedge \Box \neg F$</p>	$G = \text{@formula } s \rightarrow \text{norm}(s - [6.5, 7.5]) \leq 0.5$ $F = \text{@formula } s \rightarrow \text{norm}(s - [4.5, 4.5]) \leq 0.5$ $\Psi = \text{@formula } \Diamond(G) \wedge \Box(\neg F)$
Inverted Pendulum	<p><i>“Keep the pendulum balanced”</i></p> <p>$B(s_t)$: $\theta_t \leq \pi/4$</p> <p>$\psi = \Box B$</p>	$B = \text{@formula } s \rightarrow \text{abs}(s[1]) \leq \pi/4$ $\Psi = \text{@formula } \Box(B)$
Aircraft Collision Avoidance	<p><i>“Ensure at least 50 meters relative altitude between 40 and 41 seconds”</i></p> <p>$S(s_t)$: $h_t \geq 50$</p> <p>$\psi = \Box_{[40,41]} S$</p>	$S = \text{@formula } s \rightarrow \text{abs}(s[1]) \geq 50$ $\Psi = \text{@formula } \Box(40:41, S)$

Table 3.3. Signal temporal logic formulas for three of the example problems used throughout the book.

If the predicate is false for s_t , the robustness will be negative, and if it is true, the robustness will be positive. The signal becomes closer to satisfying the specification as the robustness approaches zero and further from not satisfying the specification as the robustness increases from zero.

Given propositions P and Q that correspond to predicates $\mu_c(s_t)$ and $\mu_d(s_t)$, we can also define robustness formulas for the propositional logic operators $\neg P$, $P \wedge Q$, $P \vee Q$, and $P \rightarrow Q$ as follows:

$$\rho(s_t, \neg P) = -\rho(s_t, P) \quad (3.11)$$

$$\rho(s_t, P \wedge Q) = \min(\rho(s_t, P), \rho(s_t, Q)) \quad (3.12)$$

$$\rho(s_t, P \vee Q) = \max(\rho(s_t, P), \rho(s_t, Q)) \quad (3.13)$$

$$\rho(s_t, P \rightarrow Q) = \max(-\rho(s_t, P), \rho(s_t, Q)) \quad (3.14)$$

Intuitively, the robustness of a conjunction is the minimum of the robustness of its components since both components must hold, and the robustness of a disjunction is the maximum of the robustness of its components since only one component must hold.

We can also define robustness over the temporal operators:

$$\rho(s_t, \diamondsuit_{[a,b]} P) = \max_{t' \in [t+a, t+b]} \rho(s_{t'}, P) \quad (3.15)$$

$$\rho(s_t, \square_{[a,b]} P) = \min_{t' \in [t+a, t+b]} \rho(s_{t'}, P) \quad (3.16)$$

$$\rho(s_t, P \cup_{[a,b]} Q) = \max_{t' \in [t+a, t+b]} \min\left(\rho(s_{t'}, Q), \min_{t'' \in [t, t']} \rho(s_{t''}, P)\right) \quad (3.17)$$

In general, the robustness of a temporal operator is the maximum or minimum of the robustness of its components over the specified time interval. We take the maximum over all time steps for the *eventually* operator to get the best-case signal because the signal must satisfy the property at only one time step in the interval. Conversely, we take the minimum over all time steps for the *always* operator because the signal must satisfy the property at all time steps in the interval. Example 3.9 demonstrates this concept.

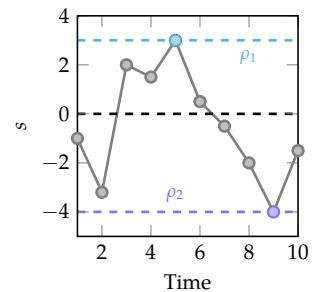
We can use the robustness metric to assess how close a given system trajectory is to a failure. Furthermore, if we are able to compute the gradient of the robustness metric with respect to certain inputs to the system, we can understand how these inputs affect the overall safety of the system. We will use this idea throughout

Let $\mu_0(s_t)$ be a predicate function that is true if s_t is greater than 0. The following code computes the robustness of the formulas $\Diamond\mu_0$ and $\Box\mu_0$ over a signal τ :

```
julia> using SignalTemporalLogic
julia> τ = [-1.0, -3.2, 2.0, 1.5, 3.0, 0.5, -0.5, -2.0, -4.0, -1.5];
julia> μ = @formula st → st > 0.0;
julia> ψ1 = @formula ▯(μ);
julia> ρ1 = ρ(τ, ψ1)
3.0
julia> ψ2 = @formula □(μ);
julia> ρ2 = ρ(τ, ψ2)
-4.0
```

The robustness of the formula $\Diamond\mu_c$ is the maximum difference between the signal and the threshold. We would have to decrease all of our signal values by at least this value to make the formula false. The robustness of the formula $\Box\mu_c$ is the minimum difference between the signal and the threshold. We would have to increase all of our signal values by at least this value to make the formula true. The figure in the caption shows signal values that determine the robustness for each formula.

Example 3.9. Robustness of the formulas $\psi_1 = \Diamond\mu_0$ and $\psi_2 = \Box\mu_0$ over a signal τ .



the book to understand system behavior. For example, we can uncover the failure modes of a system by using the robustness metric to guide the simulator towards a failure trajectory (see chapter 4 for more details).

Taking the gradient of the robustness metrics requires that the robustness formula is differentiable over the input space. However, the min and max functions that commonly occur in STL formulas are not differentiable everywhere. To address this challenge, we can use smooth approximations of the min and max functions, such as the *softmax* and *softmin* functions, respectively.¹⁸ These functions are defined as

$$\text{softmin}(\mathbf{s}; w) = \frac{\sum_i^d s_i \exp(-s_i/w)}{\sum_j^d \exp(-s_j/w)} \quad (3.18)$$

$$\text{softmax}(\mathbf{s}; w) = \frac{\sum_i^d s_i \exp(s_i/w)}{\sum_j^d \exp(s_j/w)} \quad (3.19)$$

where \mathbf{s} is a signal of length d and w is a weight. As w approaches infinity, the softmin and softmax functions approach the mean function. As w approaches zero, the softmin and softmax functions approach the min and max functions (figure 3.13). We call the robustness metric that uses the softmin and softmax functions the *smooth robustness* metric. Figure 3.14 shows the gradient of the smooth robustness metric for different values of w .

3.6 Reachability Specifications

A *reachability specification* is a special type of temporal logic specification that describes a state or set of states that a system should or should not reach during its execution. Let $\mathcal{S}_T \subseteq \mathcal{S}$ represent the target set of states and define the predicate function $R(s_t)$ to be true if $s_t \in \mathcal{S}_T$ and false otherwise. If our goal is to reach the target set, the reachability specification has the following form:

$$\psi = \Diamond R(s_t) \quad (3.20)$$

If our goal is to avoid the target set, we use the negation of the reachability specification as follows:

$$\psi = \neg \Diamond R(s_t) = \Box \neg R(s_t) \quad (3.21)$$

¹⁸ K. Leung, N. Aréchiga, and M. Pavone, “Backpropagation Through Signal Temporal Logic Specifications: Infusing Logical Structure into Gradient-Based Methods,” *The International Journal of Robotics Research*, vol. 42, no. 6, pp. 356–370, 2023.

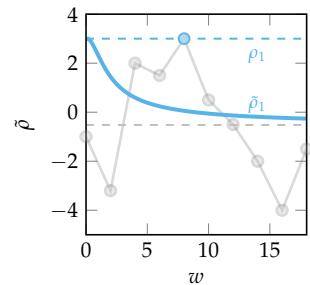


Figure 3.13. Smooth robustness metric $\tilde{\rho}$ (solid blue line) for the formula in example 3.9. The robustness metric ρ_1 is shown as a blue dashed line, and the mean of the points in the trajectory is represented as a dashed gray line. When $w = 0$, the smooth robustness metric $\tilde{\rho}$ is equal to the robustness metric ρ_1 . When w is large, the smooth robustness metric approaches the mean of the trajectory.

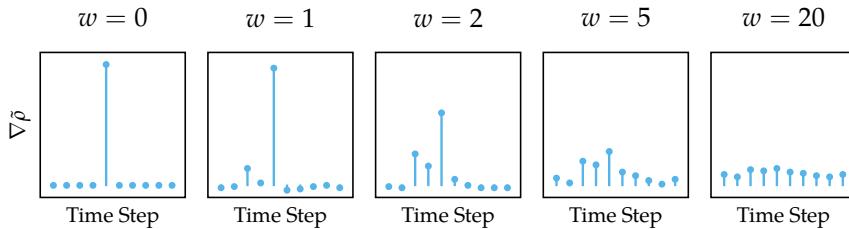


Figure 3.14. The gradient of the smooth robustness function for the formula in example 3.9 with respect to the signal values for different values of w used in the smooth robustness metric $\tilde{\rho}$. When $w = 0$, the gradient is only nonzero at the point corresponding to the maximum robustness. As w increases, the gradient becomes nonzero at all points in the trajectory. Since the smooth robustness approaches the mean of the trajectory as w increases, the gradient becomes more uniform.

Writing specifications in this form is useful because many algorithms related to formal methods and model checking are centered around reachability specifications. For example, the algorithms in chapters 8 to 10 determine whether a system could reach a target set. For some systems, such as the inverted pendulum system in example 3.10, the reachability specification is the most natural way to express the desired behavior. In general, it is possible to solve the model checking problem for other types of specifications by transforming the problem into a reachability problem using various techniques. For systems with LTL specifications, we can create a reachability problem by augmenting the state space of the system.

Let \mathcal{S}_T be the set of states for the inverted pendulum system where the pendulum has tipped over. In other words, \mathcal{S}_T is the set of states where the angle θ is outside the range $[-\pi/4, \pi/4]$. Our goal is to avoid reaching this set of states, so we define the following negated specification as

$$\psi = \neg \Diamond R(s_t) \quad (3.22)$$

where $R(s_t)$ is the predicate function that checks if the state is in the target set.

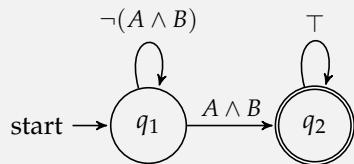
The first step in reducing the problem to a reachability problem is to represent the LTL formula as a *Büchi automaton*.¹⁹ A Büchi automaton consists of a set of states \mathcal{Q} , an initial state $q_1 \in \mathcal{Q}$, a set of atomic propositions Π , a transition function δ , and a set of accepting states.²⁰ The transition function δ maps a state and an instantiation of truth values for the atomic propositions to a set of next states. A Büchi automaton is *deterministic* if for each state and atomic proposition, there is at most one next state. For the remainder of this section, we assume

Example 3.10. Negated reachability specification for the inverted pendulum system.

¹⁹ Büchi automata are named after Swiss mathematician Julius Richard Büchi (1924–1984).

²⁰ In general, Büchi automata are defined over an alphabet. Here, we are referring to Büchi automata derived from LTL specifications, in which case the alphabet is the set of valuations of the atomic propositions.

The figure below shows a simple Büchi automaton that accepts an infinite sequence of states if the sequence satisfies the LTL formula $\Diamond(A \wedge B)$.



The automaton has two states $\mathcal{Q} = \{q_1, q_2\}$, where q_1 is the initial state and q_2 is the accepting state. The automaton has two atomic propositions A and B . The transition function is defined for all possible combinations of truth values for the atomic propositions:

$$\begin{aligned}
 \delta(q_1, A \wedge B) &= q_2 \\
 \delta(q_1, A \wedge \neg B) &= q_1 \\
 \delta(q_1, \neg A \wedge B) &= q_1 \\
 \delta(q_1, \neg A \wedge \neg B) &= q_1 \\
 \delta(q_2, -) &= q_2
 \end{aligned}$$

The diagram above compactly summarizes the transition function as $\delta(q_1, A \wedge B) = q_2$ and $\delta(q_1, \neg(A \wedge B)) = q_1$. The accepting state is denoted using the double circle.

Example 3.11. Example of a Büchi automaton with two states and two atomic propositions.

deterministic automata.²¹ The accepting states of a Büchi automaton are the states that are visited infinitely often when the automaton accepts an infinite sequence of states. Example 3.11 shows a simple Büchi automaton with two states and two propositions.

It is possible to represent any LTL formula as a Büchi automaton (example 3.12).²² Trajectories that visit an accepting state of the Büchi automaton infinitely often satisfy the corresponding LTL formula. To obtain a reachability problem from the Büchi automaton, we must augment the state space of the system of interest. The new state space is the product of the states of the system and the states of the Büchi automaton:

$$(s, q) \in \mathcal{S} \times \mathcal{Q} \quad (3.23)$$

The transition model for the new state space is defined by the transition model of the system T and the transition model of the Büchi automaton δ :

$$T((s', q') \mid (s, q), a) = \begin{cases} T(s' \mid s, a) & \text{if } q' = \delta(q, L(s)) \\ 0 & \text{otherwise} \end{cases} \quad (3.24)$$

where $L(s)$ is a labeling function that maps a state s to values for the atomic propositions of the Büchi automaton. For example, a labeling function for a grid world system with the automaton in example 3.12 would map the state s_t to values that specify whether it is a goal state or checkpoint state.

We refer to the system with the augmented state space as the *product system*. To create a reachability specification for the product system, we must determine the set of augmented states \mathcal{S}_T that are part of a cycle with an accepting state of the Büchi automaton such that we visit the accepting state infinitely often.²³ The reachability specification for the product system is

$$\psi = \Diamond R((s_t, q_t)) \quad (3.25)$$

where $R((s_t, q_t))$ is a predicate function that returns true if $(s_t, q_t) \in \mathcal{S}_T$ and false otherwise. Checking whether a trajectory of the product system satisfies the reachability specification is equivalent to checking whether the equivalent trajectory in the original system satisfies the LTL formula. Figure 3.15 shows the product system with the grid world as the original system and the LTL specification in example 3.12.

²¹ While all LTL specifications can be represented as Büchi automata, they cannot all be represented as deterministic Büchi automata. For specifications that result in nondeterministic Büchi automata, it may be necessary to represent them as *deterministic Rabin automata* instead. More details are provided by M. Bouton, J. Tumova, and M.J. Kochenderfer, “Point-Based Methods for Model Checking in Partially Observable Markov Decision Processes,” in *AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

²² More details are provided in C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008. Open source software packages such as Spot can be used to do the conversion automatically. A. Duret-Lutz, “Manipulating LTL Formulas Using Spot 1.0,” in *Automated Technology for Verification and Analysis*, 2013. The `Spot.jl` package provides an interface to the Spot library.

²³ For many product systems, these states can be determined by manually inspecting the automata. For more complex specification, it may be necessary to systematically search for these states using graph algorithms. C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.

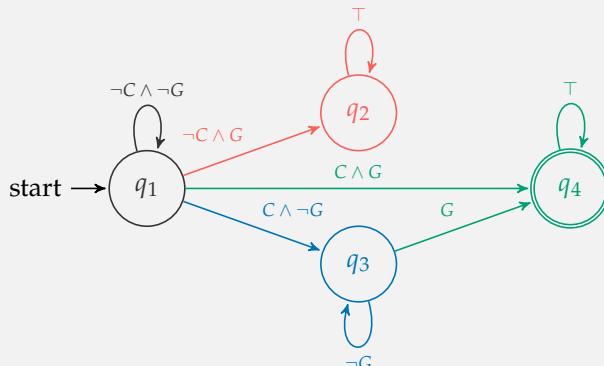
Suppose we have an LTL formula that specifies that we need to visit a checkpoint before reaching a goal, written as

$$\Diamond G \wedge \neg G \mathcal{U} C$$

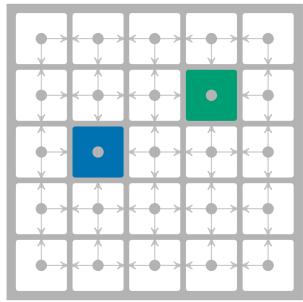
where G is an atomic proposition that represents whether we reach the goal and C is an atomic proposition that represents whether we reach the checkpoint. We can convert this formula into the Büchi automaton using `Spot.jl` as follows:

```
using Spot
a = translate(LTLTranslator(), ltl"◊(G) ∧ ¬G ∥ C")
```

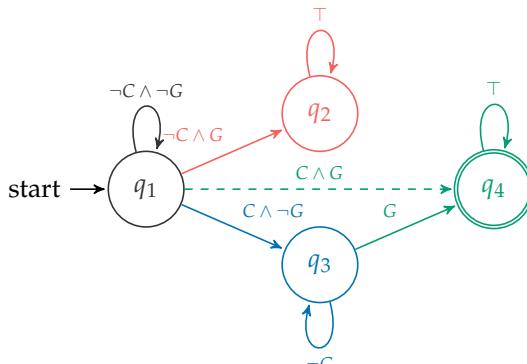
The resulting automaton is shown below. It has 4 states and the same atomic propositions as the LTL formula. The accepting state is q_4 , and a trajectory satisfies the LTL formula if it results in a trace that visits q_4 infinitely often when passed through the automaton. In other words, the LTL formula will be satisfied for trajectories that reach q_4 . The state q_2 represents the state where the agent has reached the goal but has not reached the checkpoint. Once this state has been reached, the agent will remain in this state forever with no chance of reaching the accepting state and satisfying the LTL formula. This state is often omitted in practice to reduce the size of the automaton. In this case, if the agent follows any transition that is not explicitly shown on the automata, we assume that the agent “falls off” the automata, and the property is not satisfied.



Example 3.12. Conversion of an LTL formula to a Büchi automaton.



Original System



Büchi Automaton

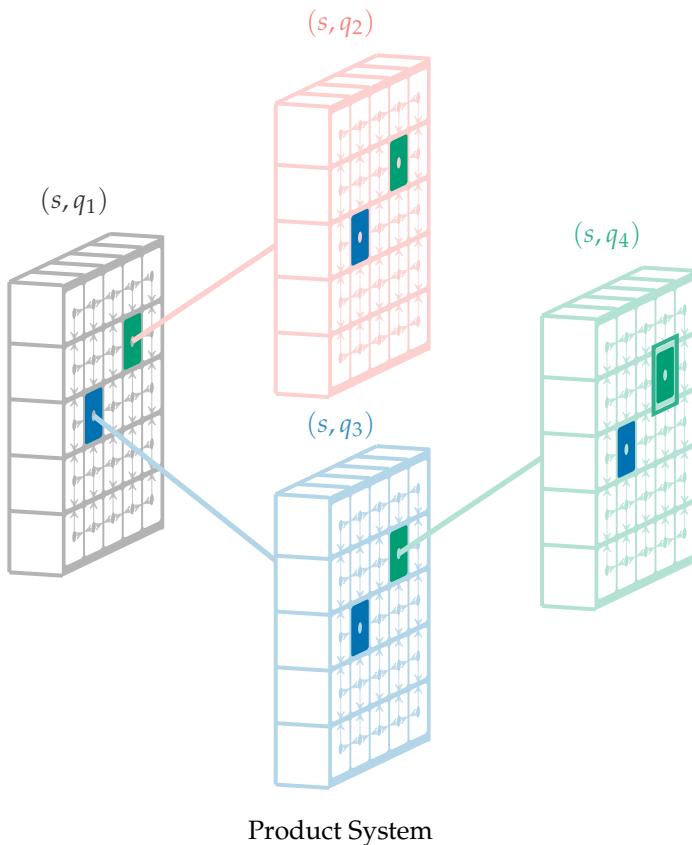


Figure 3.15. Converting an LTL specification (see example 3.12) for the grid world problem to a reachability specification by creating a product system with an augmented state space. The original system is shown on the left, the Büchi automaton is shown in the middle, and the product system is shown on the bottom. We start in the gray grid world until we either reach the checkpoint (blue) or goal (green). If we reach the goal in the gray grid world, we transition to the red grid world and remain there forever. If we reach the checkpoint in the gray grid world, we transition to the blue grid world and remain there until we reach the goal. If we reach the goal in the blue grid world, we transition to the green grid world, which represents an accepting state for the Büchi automaton. After transitioning to this accepting state, we remain there for all time steps in the future. The dashed green line in the Büchi automaton does not appear in the product system because we cannot reach the checkpoint and goal at the same time in the grid world system. The set of target states for the reachability problem is the set of states in the green grid world.

3.7 Summary

- Metrics and specifications allow us to quantify and express the desired behavior of a system.
- For stochastic systems, we often compute metrics over the full distribution of possible outcomes.
- In situations where we are interested in multiple metrics, we can create a composite metric that accounts for the relative importance of each metric.
- Logical specifications allow us to formally express requirements for a system using logical formulas.
- Propositional logic and first-order logic allow us to express properties over a set of propositions.
- Temporal logic extends first-order logic to express properties about how systems evolve over time.
- Linear temporal logic (LTL) and signal temporal logic (STL) are two common temporal logics used in control and verification.
- Reachability specifications are a special type of temporal logic specification that describe a state or set of states that a system should or should not reach during its execution.

3.8 Exercises

Exercise 3.1. Suppose we are creating a system to detect cavities in teeth from X-ray images. Provide an example of a metric and specification that we might use to evaluate the system.

Solution: There are many possible answers. Some example metrics are the false positive rate, the false negative rate, and the accuracy. An example specification might be that the system should have a false positive rate of less than 5%.

Exercise 3.2. Suppose we have a metric for an aircraft collision avoidance system that evaluates to 1 if a collision occurs and 0 otherwise. We evaluate this metric on a set of aircraft encounters and estimate the expected value. What does this estimate represent?

Solution: The expected value of a binary metric represents a probability. In this case, the expected value of the metric represents the probability of collision.

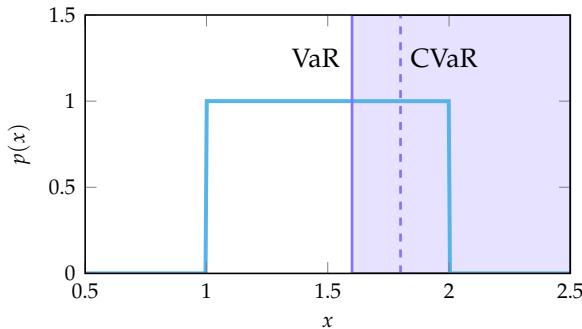
Exercise 3.3. Consider a risk metric that is distributed according to a Gaussian distribution with mean 0 and variance 1^2 . Compute the VaR for $\alpha = 0.8$.

Solution: The VaR is equal to the 0.8-quantile of the distribution. We can compute this value using the `Distributions.jl` package as follows:

```
julia> using Distributions
julia> VaR = quantile(Normal(), 0.8)
0.8416212335729143
```

Exercise 3.4. Consider a risk metric that is distributed according to a uniform distribution between 1 and 2. Compute the VaR and CVaR for $\alpha = 0.6$.

Solution: The figure below show the VaR and CVaR. The VaR (solid purple line) is the 0.6-quantile of the distribution, which is 1.6. The CVaR (dashed purple line) is the expected value of the distribution for values greater than the VaR (shaded region). Since the probability density is uniform for values between 1.6 and 2 and 0 otherwise, the CVaR is the average of the values in this range, which is 1.8.



Exercise 3.5. Suppose we are selecting between five different collision avoidance systems with the following (alert rate, collision rate) pairs: $(0.2, 0.8)$, $(0.3, 0.6)$, $(0.4, 0.4)$, $(0.5, 0.2)$, and $(0.8, 0.1)$. Identify which system we should select for each of the following composite metrics:

1. A weighted sum metric with weights $\mathbf{w} = [0.8, 0.2]$.
2. A goal distance metric with $f_{\text{goal}} = (0, 0)$ using the L_2 norm (Euclidean distance).

Solution:

1. The weighted sum metric results in the following scores for each system: 0.32, 0.36, 0.4, 0.44, and 0.66. In this case, we want to select the system with the lowest weighted sum. Therefore, we should select the system with the $(0.2, 0.8)$ pair.

2. The goal distance metric results in the following scores for each system: 0.82, 0.67, 0.57, 0.54, and 0.81. In this case, we want to select the system that is closest to the goal. Therefore, we should select the system with the (0.5, 0.2) pair.

Exercise 3.6. Suppose we evaluate a financial trading system using two metrics. The first metric is the fraction of trades that are profitable, and the second metric is fraction of trades that result in significant losses. We want to create a composite metric using the goal distance method. What should we select as the goal for the composite metric?

Solution: We should select the goal to be the utopia point of $[1, 0]$, which corresponds to the ideal case where all trades are profitable and none result in significant losses.

Exercise 3.7. Suppose we want to infer a weight vector for a weighted sum composite metric that describes Robert's cookie preferences. We present him with two options. The first option is 2 chocolate chip cookies and 1 snickerdoodle cookie. The second option is 1 chocolate chip cookie and 2 snickerdoodle cookies. Robert chooses the first option. Assume that the weight vector is $\mathbf{w} = [w_1, w_2]$ where w_1 is the weight for chocolate chip cookies and w_2 is the weight for snickerdoodle cookies. Write the constraint that the weight vector must satisfy to match Robert's choice.

Solution: If Robert chose $\mathbf{f}_1 = [2, 1]$ over $\mathbf{f}_2 = [1, 2]$, it must be the case that the composite metric for \mathbf{f}_1 is greater than the composite metric for \mathbf{f}_2 . This constraint can be written as

$$\begin{aligned}\mathbf{w}^\top \mathbf{f}_1 &> \mathbf{w}^\top \mathbf{f}_2 \\ \mathbf{w}^\top (\mathbf{f}_1 - \mathbf{f}_2) &> 0 \\ \mathbf{w}^\top [1, -1] &> 0\end{aligned}$$

Exercise 3.8. Complete the following truth table:

P	Q	$(P \rightarrow Q) \wedge \neg P$
false	false	?
false	true	?
true	false	?
true	true	?

Solution:

P	Q	$(P \rightarrow Q) \wedge \neg P$
false	false	true
false	true	true
true	false	false
true	true	false

Exercise 3.9. Suppose we are creating a specification for a database system that stores user passwords. We only want the system to give the user their password if it is able to authenticate the user using some other method. Write this specification using propositional logic.

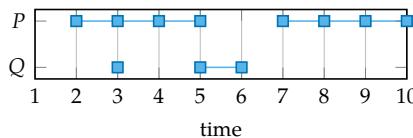
Solution: We define the following two propositions:

A : The user has been authenticated

P : The system has provided the user with their password

The specification can be then be written as $P \rightarrow A$, which states that if the system provides the user with their password, then the user has been authenticated.

Exercise 3.10. Consider the following plot of truth values over time for atomic propositions P and Q :



where time steps marked with a blue square indicate that the corresponding atomic proposition was true at that time step. List all time steps t where the temporal logic formula $\Diamond(P \rightarrow Q)$ holds. Assume that the temporal logic formula is always defined from time t until the end of the sequence.

Solution: The formula holds at time steps 1, 3, 5, and 6. Therefore, $\Diamond(P \rightarrow Q)$ holds for $t \in [1, 6]$.

Exercise 3.11. Suppose we are creating a specification for a database system that stores user passwords. We only want the system to give the user their password if it is able to authenticate the user using some other method. We write the specification using linear temporal logic as $\neg P \wedge A$, where P is the proposition that the system has provided the user with their password and A is the proposition that the user has been authenticated. Why might this specification not be appropriate for this system?

Solution: The specification $\neg P \wedge A$ not only requires that the password is not given to the user until the user is authenticated, but also that the user is eventually authenticated. However, if the user is trying to hack the system, we do not want the system to authenticate the user.

Exercise 3.12. Derive equation (3.14) from equation (3.11) and equation (3.13).

Solution: As shown in table (3.1), we know that $P \rightarrow Q$ is equivalent to $\neg P \vee Q$. Therefore,

$$\begin{aligned}\rho(s_t, P \rightarrow Q) &= \rho(s_t, \neg P \vee Q) \\ &= \max(\rho(s_t, \neg P), \rho(s_t, Q)) \\ &= \max(-\rho(s_t, P), \rho(s_t, Q))\end{aligned}$$

Exercise 3.13. Consider an aircraft collision avoidance trajectory with a relative altitude h of 100 m at time step 40 and a relative altitude of 105 m at time step 41. What is the robustness of the trajectory with respect to the following specification:

$$\psi = \square_{[40,41]} S(s_t)$$

where $S(s_t)$ is true when $|h_t| \geq 50$.

Solution:

$$\begin{aligned}\rho(s_t, \psi) &= \rho(s_t, \square_{[40,41]} S(s_t)) \\ &= \rho(s_t, \square_{[40,41]} |h_t| \geq 50) \\ &= \min_{t \in [40,41]} \rho(s_t, |h_t| - 50) \\ &= \min(|100| - 50, |105| - 50) \\ &= 50\end{aligned}$$

Exercise 3.14. Consider a system with three states s_1 , s_2 , and s_3 that we want to evaluate using the LTL formula in example 3.11. The transition model for the system is as follows. An agent in state s_1 will transition to state s_2 with probability 0.3 and remain in state s_1 with probability 0.7. An agent in state s_2 will transition to state s_3 with probability 0.4 and remain in state s_2 with probability 0.6. An agent in state s_3 will remain in state s_3 with probability 1. The labeling function for the system is as follows:

$$\begin{aligned}L(s_1) &= A \wedge \neg B \\L(s_2) &= \neg A \wedge \neg B \\L(s_3) &= A \wedge B\end{aligned}$$

What are the values of the following transitions in the augmented state space: $T((s_2, q_1) | (s_1, q_1))$, $T((s_2, q_2) | (s_1, q_1))$, and $T((s_3, q_2) | (s_3, q_1))$?

Solution: We need to plug values into equation (3.24). For $T((s_2, q_1) | (s_1, q_1))$, we start by computing $\delta(q_1, L(s_1)) = \delta(q_1, A \wedge \neg B)$. According to example 3.11, the result is q_1 . Therefore, we take the first case in equation (3.24) and compute the transition as follows:

$$\begin{aligned}T((s_2, q_1) | (s_1, q_1)) &= T(s_2 | s_1) \\&= 0.3\end{aligned}$$

For $T((s_2, q_2) | (s_1, q_1))$, we find that $q_2 \neq \delta(q_1, L(s_1))$, so we take the second case in equation (3.24) and the transition probability is 0. Finally, for $T((s_3, q_2) | (s_3, q_1))$, we find that $q_2 = \delta(q_1, L(s_3))$, so we take the first case in equation (3.24) and the transition probability is

$$\begin{aligned}T((s_3, q_2) | (s_3, q_1)) &= T(s_3 | s_3) \\&= 1\end{aligned}$$

4 Falsification through Optimization

The first set of validation algorithms we will explore relate to *falsification*. Falsification is the process of finding trajectories of a system that violate a given specification. Such trajectories are sometimes referred to as *counterexamples*, *failure trajectories*, or *falsifying trajectories*. We will refer to them in this textbook as *failures* for simplicity. The beginning of the chapter introduces a naïve algorithm for finding failures based on direct sampling, with the rest of the chapter focused on more sophisticated algorithms that use optimization techniques to guide the search for failures. Optimization-based falsification relies on the concept of *disturbances*, which control the behavior of the system. We demonstrate how to frame the falsification problem as an optimization over disturbance trajectories and outline several techniques to perform the optimization.

4.1 Direct Sampling

When performing falsification, we want to find any trajectory τ that violates a given specification ψ , written as $\tau \notin \psi$. Algorithm 4.1 uses direct sampling to search for such trajectories.¹ It performs m rollouts and returns all failure trajectories. Figure 4.1 shows an example of direct falsification applied to the grid world problem.

Algorithm 4.1 may struggle for systems with rare failure events. For a system with probability of failure p_{fail} , we will require $1/p_{\text{fail}}$ samples on average to observe a single failure. In fact, we can infer a distribution over the number of samples required to find a failure. The probability of finding the first failure on the k th sample is equivalent to the probability of sampling $k - 1$ successes with probability $1 - p_{\text{fail}}$ and one failure with probability p_{fail} . We therefore write the

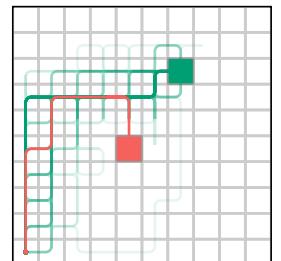


Figure 4.1. Direct falsification applied to the grid world problem with $m = 100$ and $d = 50$. The probability of slipping is set to 0.8. The algorithm samples 96 trajectories before finding a failure. The failure trajectory is shown in red.

¹This type of sampling is often referred to as Monte Carlo sampling, named after the Monte Carlo casino in Monaco. Similar to gambling, the algorithm depends on random chance.

```

struct DirectFalsification
    d # depth
    m # number of samples
end

function falsify(alg::DirectFalsification, sys, ψ)
    d, m = alg.d, alg.m
    ts = [rollout(sys, d=d) for i in 1:m]
    return filter(τ → isfailure(ψ, τ), ts)
end

```

Algorithm 4.1. The direct falsification algorithm for finding failures. The algorithm performs rollouts to a depth d to generate m samples of the system sys . It then filters these samples and returns the ones that violate the specification ψ . If no failures are found, the algorithm returns an empty vector.

probability mass function of the distribution as

$$P(k) = (1 - p_{\text{fail}})^{k-1} p_{\text{fail}} \quad (4.1)$$

where $k \in \mathbb{N}$.

Equation (4.1) corresponds to the probability mass function of a *geometric distribution* with parameter p_{fail} . Figure 4.2 shows an example of a geometric distribution. The expected value of this distribution, $1/p_{\text{fail}}$, corresponds to the average number of samples required to find a failure. Example 4.1 illustrates this relationship for the aircraft collision avoidance problem. Systems with very low failure probabilities will require a large number of samples for direct falsification. For example, some aviation systems have failure probabilities on the order of 10^{-9} . These systems require 1 billion samples on average to observe a single failure event. The remainder of the chapter discusses more efficient falsification techniques.

4.2 Disturbances

We can systematically search for failures by taking control of the sources of randomness in the system. We control these sources of randomness using *disturbances*. To incorporate disturbances into a system, we rewrite its sensor, agent, and environment models by breaking up their stochastic and deterministic elements. For example, the observation model $o \sim O(\cdot | s)$ can be written as a deterministic function of the current state s and a stochastic disturbance x_o such that

$$o = O(s, x_o), x_o \sim D_o(\cdot | s) \quad (4.2)$$

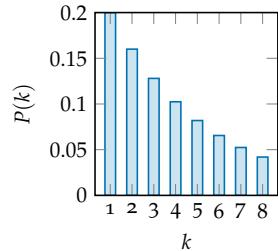
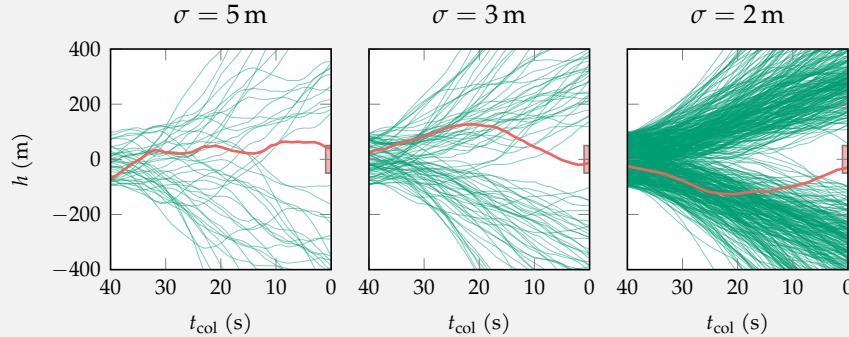


Figure 4.2. The probability mass function of a geometric distribution with parameter $p_{\text{fail}} = 0.2$. The expected value of this distribution is $1/p_{\text{fail}} = 5$.

Suppose we want to find failures of an aircraft collision avoidance system using direct falsification. In this scenario, a failure is a collision between two aircraft, which occurs when the relative altitude to the intruder aircraft h is within ± 50 m and the time to collision t_{col} is zero. The collision avoidance environment applies additive noise with standard deviation σ to the relative vertical rate of the intruder aircraft \dot{h} at each time step. This noise accounts for variation in pilot response to advisories and the intruder flight path. The plots below use different values of σ and show the trajectory samples produced before finding the first failure with the first failure trajectory highlighted in red.



As σ decreases, failures become less likely, and more trajectories are required to find a failure. In this example, the first failure is found after 41 samples with $\sigma = 5$ m, 84 samples with $\sigma = 3$ m, and 522 samples with $\sigma = 2$ m.

Example 4.1. Direct falsification applied to the aircraft collision avoidance problem with different levels of noise applied to the transitions. There are four state variables for the collision avoidance problem. These plots show how two of these state variables evolve for each trajectory. The horizontal axis is the time to collision t_{col} , and the vertical axis is the altitude relative to the intruder aircraft h .

where $O(s, x_o)$ is a deterministic function and $D_o(\cdot | s)$ is a *disturbance distribution*. For example, a disturbance applied to an additive noise sensor controls the amount of sensor noise added to the true state to produce an observation. Example 4.2 demonstrates this concept for a Gaussian noise sensor model.

Suppose we model a sensor using a Gaussian noise model such that $O(o | s) = \mathcal{N}(o | s, \Sigma)$. We can rewrite this sensor model as

$$o = s + x_o, x_o \sim \mathcal{N}(\cdot | 0, \Sigma)$$

We can then define this sensor using the following code:

```
struct GaussianNoiseSensor <: Sensor
    Do # distribution = Do(s)
end
(sensor::GaussianNoiseSensor)(s) = s + rand(sensor.Do(s))
(sensor::GaussianNoiseSensor)(s, xo) = s + xo
```

In this code, `Do` represents the nominal disturbance distribution $D_o(x_o | s) = \mathcal{N}(x_o | 0, \Sigma)$. Since it is a conditional distribution, we represent it as a function that takes in a state `s` and outputs a distribution. The disturbance in this sensor model does not depend on the state, so the function returns the same distribution regardless of its input. The first function represents the original sensor model and adds noise sampled from `Do` to the true state `s` to produce an observation. The second function allows us to deterministically produce an observation for state `s` given a disturbance `xo`.

Example 4.2. Separating the stochastic and deterministic elements of a sensor with a Gaussian noise model.

The agent's policy and the environment's transition model can also be decomposed:

$$a = \pi(o, x_a), x_a \sim D_a(\cdot | o) \quad (4.3)$$

$$s' = T(s, a, x_s), x_s \sim D_s(\cdot | s, a) \quad (4.4)$$

where $\pi(o, x_a)$ and $T(s, a, x_s)$ are deterministic functions and $D_a(\cdot | s)$ and $D_s(\cdot | s, a)$ are disturbance distributions. In this textbook, we will wrap these three components into a single disturbance x and disturbance distribution D . Given a current state and disturbance distribution, we can sample a disturbance and produce an observation, action, and next state (algorithm 4.2). For system components that are modeled using deterministic functions, applying a disturbance has no effect.

```

struct Disturbance
    xa # agent disturbance
    xs # environment disturbance
    xo # sensor disturbance
end

struct DisturbanceDistribution
    Da # agent disturbance distribution
    Ds # environment disturbance distribution
    Do # sensor disturbance distribution
end

function step(sys::System, s, D::DisturbanceDistribution)
    xo = rand(D.Do(s))
    o = sys.sensor(s, xo)
    xa = rand(D.Da(o))
    a = sys.agent(o, xa)
    xs = rand(D.Ds(s, a))
    s' = sys.env(s, a, xs)
    x = Disturbance(xa, xs, xo)
    return (o, a, s', x)
end

```

Algorithm 4.2. Implementation of a disturbance and disturbance distribution. The individual disturbance components are used to control the agent, environment, and sensor, respectively. Since the components of the disturbance distribution are conditional distributions, we assume they are functions that take in the evidence variables and output a sampleable distribution. Given a current state s and disturbance distribution D , the `step` function samples a disturbance and uses it to produce an observation, action, and next state.

4.3 Fuzzing

Unlike direct sampling, which samples from the nominal distribution over system trajectories, we can find failures more efficiently by sampling from a trajectory distribution designed to stress the system. We refer to this process as *fuzzing*.² Before we can perform fuzzing, we need to define the components of a trajectory distribution. There are two sources of randomness in a trajectory rollout: the initial state and the disturbances applied at each time step. Therefore, we can fully capture the distribution over trajectories by specifying an initial state distribution and a disturbance distribution for each time step (algorithm 4.3).

```

abstract type TrajectoryDistribution end
function initial_state_distribution(p::TrajectoryDistribution) end
function disturbance_distribution(p::TrajectoryDistribution, t) end
function depth(p::TrajectoryDistribution) end

```

In the algorithms presented so far, we have been implicitly sampling from the nominal trajectory distribution for a system. We can explicitly construct this distribution for a given system using algorithm 4.4. The nominal trajectory distribution

² Fuzzing is a well-known concept in testing of traditional software. It refers to the generation of off-nominal inputs to a program to uncover potential bugs or failures and was first introduced in B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

Algorithm 4.3. Definition of a trajectory distribution. The `initial_state_distribution` function returns the distribution over initial states. The `disturbance_distribution` function returns the disturbance distribution at time t . The `depth` function returns the number of time steps in the trajectories sampled from the distribution.

uses the default initial state and disturbance distributions for the components of the system. Nominal trajectory distributions are *stationary*, meaning that the disturbance distribution does not depend on time.

```

struct NominalTrajectoryDistribution <: TrajectoryDistribution
    Ps # initial state distribution
    D # disturbance distribution
    d # depth
end

function NominalTrajectoryDistribution(sys::System, d)
    D = DisturbanceDistribution((o) → Da(sys.agent, o),
                                (s, a) → Ds(sys.env, s, a),
                                (s) → Do(sys.sensor, s))
    return NominalTrajectoryDistribution(Ps(sys.env), D, d)
end

initial_state_distribution(p::NominalTrajectoryDistribution) = p.Ps
disturbance_distribution(p::NominalTrajectoryDistribution, t) = p.D
depth(p::NominalTrajectoryDistribution) = p.d

```

Algorithm 4.4. The nominal trajectory distribution for a system. We can construct this distribution for a given system `sys` and depth `d` using the default initial state and disturbance distributions specified by the components of the system. Nominal trajectory distributions are stationary, so the `disturbance_distribution` function returns the same value for any time input `t`.

We sample trajectories from a trajectory distribution by performing rollouts. Algorithm 4.5 implements a trajectory rollout given a trajectory distribution. It returns a trajectory $\tau = (s_1, o_1, a_1, x_1, \dots, s_d, o_d, a_d, x_d)$. If the initial state distribution and disturbance distributions correspond to the nominal distributions for the system, algorithm 4.5 performs the same function as algorithm 1.2. However, algorithm 4.5 also allows us to sample from a different trajectory distribution. We can use it to perform fuzzing by specifying a trajectory distribution that is designed to increase the likelihood of sampling failure trajectories. Example 4.3 demonstrates this technique on the inverted pendulum system.

```

function rollout(sys::System, p::TrajectoryDistribution; d=depth(p))
    s = rand(initial_state_distribution(p))
    τ = []
    for t = 1:d
        o, a, s', x = step(sys, s, disturbance_distribution(p, t))
        push!(τ, (; s, o, a, x))
        s = s'
    end
    return τ
end

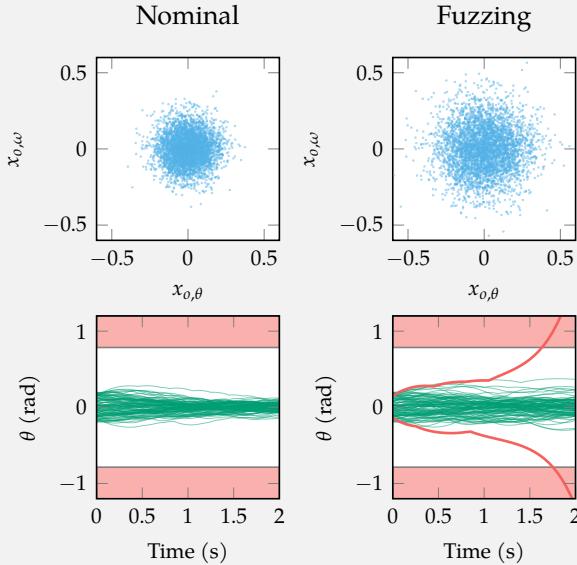
```

Algorithm 4.5. A function that performs a rollout of a system `sys` to a depth `d` given an initial state `s` and trajectory distribution `p`. It repeatedly calls the `step` function, which steps the system forward in time.

Suppose we want to find failures of the inverted pendulum system with an additive noise sensor with $D_o(o | s) = \mathcal{N}(o | 0, \Sigma)$ and $\Sigma = 0.01I$. If we collect 100 samples with this nominal distribution, we do not find any failures. However, if we define a new distribution and increase the standard deviation of the sensor noise on each variable from 0.1 to 0.15 (referred to as fuzzing), we are able to find two failures of the system in the first 100 samples. The following code can be used to define the fuzzing distribution:

```
struct PendulumFuzzingDistribution <: TrajectoryDistribution
    Σ₀ # sensor disturbance covariance
    d # depth
end
function initial_state_distribution(p::PendulumFuzzingDistribution)
    return Product([Uniform(-π / 16, π / 16), Uniform(-1., 1.)])
end
function disturbance_distribution(p::PendulumFuzzingDistribution, t)
    D = DisturbanceDistribution((o)→Deterministic(),
                                (s,a)→Deterministic(),
                                (s)→MvNormal(zeros(2), p.Σ₀))
    return D
end
depth(p::PendulumFuzzingDistribution) = p.d
```

The plots show the disturbances and trajectories for both distributions.



Example 4.3. Fuzzing applied to the inverted pendulum system. The plots in the top row show the sampled disturbances for the sensor noise on each state variable. The initial state distribution is the same as the nominal initial state distribution (algorithm A.6). The plots on the bottom row show the corresponding trajectories for θ with failures highlighted in red. By slightly increasing the standard deviation of the simulated sensor noise, we are able to uncover two failures.

4.4 Falsification through Optimization

The falsification problem can be reformulated as a search over the space of initial states and disturbances. Algorithm 4.6 performs a trajectory rollout given an initial state and a sequence of disturbances. We refer to this sequence of disturbances as a disturbance trajectory $\mathbf{x} = (x_1, \dots, x_d)$. Unlike algorithm 4.5, algorithm 4.6 is deterministic. The initial state s and disturbance trajectory \mathbf{x} fully determine the resulting trajectory τ .

```

function step(sys::System, s, x)
    o = sys.sensor(s, x.xo)
    a = sys.agent(o, x.xa)
    s' = sys.env(s, a, x.xs)
    return (; o, a, s')
end

function rollout(sys::System, s, x; d=length(x))
    τ = []
    for t in 1:d
        x = x[t]
        o, a, s' = step(sys, s, x)
        push!(τ, (; s, o, a, x))
        s = s'
    end
    return τ
end

```

Algorithm 4.6. A function that performs a rollout of a system `sys` to a depth `d` given an initial state `s` and disturbance trajectory `x`. It repeatedly calls the `step` function, which steps the system forward in time. The `step` function takes in the current state `s` and disturbance `x` and deterministically produces an observation `o` from the sensor, gets the action `a` from the agent based on this observation, and determines the next state `s'` from the environment.

To perform falsification in this context, we want to find an initial state s and disturbance trajectory \mathbf{x} that produce a trajectory τ such that $\tau \notin \psi$. Optimization-based falsification techniques use an objective function to guide this search. An objective function $f(\tau)$ maps a trajectory τ to a value related to its level of safety with respect to ψ . We can then search for failures by minimizing this objective over the space of initial states and disturbances as follows:

$$\begin{aligned} & \underset{s,x}{\text{minimize}} \quad f(\tau) \\ & \text{subject to} \quad \tau = \text{Rollout}(s, x) \end{aligned} \tag{4.5}$$

The rest of this chapter discusses different objective functions and optimization techniques for solving the optimization problem in equation (4.5).

4.5 Objective Functions

Objective functions guide the search for failure trajectories. In general, a good objective function should output lower values for trajectories that are closer to a failure. The specific measure of closeness used is dependent on the application. For example, in the aircraft collision avoidance problem, we may use the vertical miss distance between the aircraft as the objective value.

4.5.1 Temporal Logic Robustness

If ψ is specified using a temporal logic formula, we can use its robustness measure (see section 3.5.2) as an objective function such that $f(\tau) = \rho(\tau, \psi)$. Note that τ itself is a function of the initial state and disturbance trajectory, and we can also write this objective function as $f(s, x) = \rho(\text{Rollout}(s, x), \psi)$. Algorithm 4.7 implements this objective function given a system and a specification.

```
function robustness_objective(x, sys, ψ; smoothness=0.0)
    s, x = extract(sys.env, x)
    τ = rollout(sys, s, x)
    s = [step.s for step in τ]
    return robustness(s, ψ.formula, w=smoothness)
end
```

Since most optimization algorithms operate on a vector of real values, algorithm 4.7 takes in a vector of real values containing information about the initial state and disturbances. The first step inside the objective function is to extract the initial state and disturbance trajectories in a way that is system specific. Example 4.4 demonstrates this process for the inverted pendulum system. Given these extracted values, we can perform a rollout of the system using algorithm 4.6 and compute the corresponding robustness. For optimization algorithms that require gradients of the objective function, we use the smoothed robustness instead.

4.5.2 Most Likely Failure

The use of an objective function in optimization-based falsification algorithms allows us to move beyond a simple search for failures and incorporate other objectives into the search. For example, instead of finding any failure, we may want to find the most likely failure of a system.³ Determining the most likely

Algorithm 4.7. Temporal logic robustness objective. The function takes in a vector of real values x , a system sys , and a specification ψ . It returns the smoothed robustness of the resulting trajectory. If $smoothness$ is set to 0, it returns the robustness. The vector x contains information about the initial state and disturbances. The `extract` function extracts an initial state and disturbance trajectory from x and is system specific.

³ Another common objective is to find the most severe failure according to a severity metric. There may also be domain-specific objectives such as obeying traffic laws in a driving scenario.

Suppose we want to compute the robustness objective for the inverted pendulum system where the initial state is always $s = [0, 0]$. We write the `extract` function as follows:

```
function extract(env::InvertedPendulum, x)
    s = [0.0, 0.0]
    x = [Disturbance(0, 0, x[i:i+1]) for i in 1:2:length(x)]
    return s, x
end
```

The function extracts the sensor disturbances from the real-valued vector x to create a disturbance trajectory \mathbf{x} . We set the agent and environment disturbances to zero because the agent and environment are deterministic. It then returns the fixed initial state \mathbf{s} and the disturbance trajectory.

Example 4.4. Extracting an initial state and disturbance trajectory from a vector of real values for the inverted pendulum system.

failure requires specifying the distribution over trajectories and using its probability density function to evaluate likelihoods. Assuming that the initial state and disturbances are sampled independently from one another, the probability density of a trajectory is

$$p(\tau) = p(s_1) \prod_{i=1}^d D(x_i | s_i, a_i, o_i) \quad (4.6)$$

where $D(x | s, a, o) = D_a(x_a | o)D_s(x_s | s, a)D_o(x_o | s)$. Algorithm 4.8 implements equation (4.6).

```
function Distributions.logpdf(D::DisturbanceDistribution, s, o, a, x)
    logp_xa = logpdf(D.Da(o), x.xa)
    logp_xs = logpdf(D.Ds(s, a), x.xs)
    logp_xo = logpdf(D.Do(s), x.xo)
    return logp_xa + logp_xs + logp_xo
end

function Distributions.pdf(p::TrajectoryDistribution, τ)
    logprob = logpdf(initial_state_distribution(p), τ[1].s)
    for (t, step) in enumerate(τ)
        s, o, a, x = step
        logprob += logpdf(disturbance_distribution(p, t), s, o, a, x)
    end
    return exp(logprob)
end
```

Algorithm 4.8. Probability density function of a trajectory distribution p . We perform computations in log space for numerical stability. We first compute the log likelihood of the initial state according the initial state distribution. We then add the log likelihood of each disturbance in the trajectory. The first function evaluates the log likelihood of a disturbance given a disturbance distribution D and the evidence variables.

```

function likelihood_objective(x, sys, ψ; smoothness=0.0)
    s, x = extract(sys.env, x)
    τ = rollout(sys, s, x)
    if isfailure(ψ, τ)
        p = NominalTrajectoryDistribution(sys, length(x))
        return -pdf(p, τ)
    else
        s = [step.s for step in τ]
        return robustness(s, ψ.formula, w=smoothness)
    end
end

```

Algorithm 4.9. Objective function for finding the most likely failure. The function takes in a vector of real values x , a system sys , and a specification ψ . If the resulting trajectory is a failure, it returns the negative likelihood of the trajectory under the nominal trajectory distribution p . Otherwise, it returns the smoothed robustness of the trajectory (or the robustness if $smoothness$ is set to 0).

Given a trajectory distribution p , we define the most likely failure objective (algorithm 4.9) as follows

$$f(\tau) = \begin{cases} -p(\tau) & \text{if } \tau \notin \psi \\ \rho(\tau, \psi) & \text{otherwise} \end{cases} \quad (4.7)$$

If the input trajectory does not produce a failure, equation (4.7) uses the robustness to guide the search toward any failure. If the input does produce a failure trajectory, it uses the negative likelihood of the trajectory to guide the search toward more likely failures. Figure 4.3 compares a search for failures with a search for the most likely failure on the grid world problem. While the robustness objective finds failures that move directly toward the obstacle, the most likely failure objective finds a failure that stays close to the nominal path.

The objective function in equation (4.7) leads to multiple practical challenges. For example, to encourage the optimization algorithm to find failures, we must ensure that failures never have a higher objective value than successes. Since $\rho(\tau, \psi) \geq 0$ when $\tau \in \psi$ and $-p(\tau) \leq 0$, equation (4.7) satisfies this condition. However, $p(\tau)$ can be very small for long trajectories, which can lead to numerical stability issues. Using log likelihood improves numerical stability but breaks the condition that failures never have a higher objective value than successes.

This numerical instability as well as the discontinuity at the point of a failure creates challenges for first- and second-order optimization algorithms (section 4.6). Furthermore, while the global minimum of the objective function in equation (4.7) corresponds to the most likely failure of the system, many optimizers are only guaranteed to find local minima. Due to this fact and the numerical stability

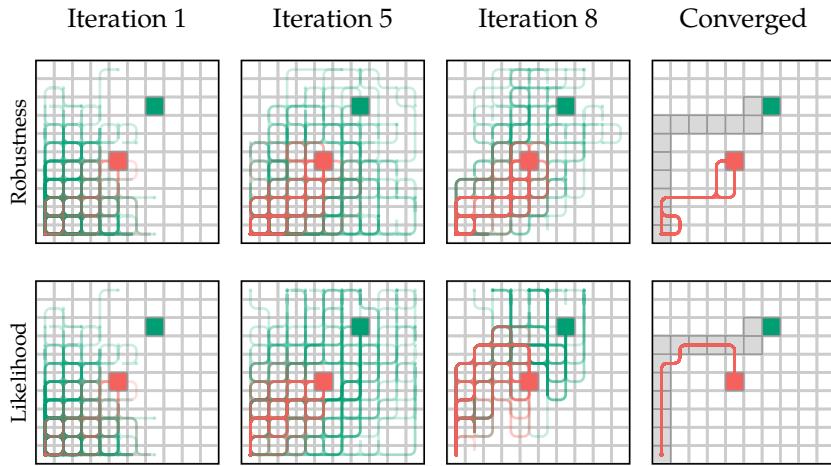


Figure 4.3. A comparison of optimization-based falsification on the grid world using the robustness objective and the likelihood objective. The plots show the progression of a population-based optimization algorithm, which will be discussed in the next section. The shaded gray path on the plots in the final column represents the most likely path for the system. The robustness objective finds failures that quickly move towards the obstacle, while the most likely failure objective find a failure that stays close to the nominal path, only veering toward the obstacle at the end.

issues, other objective functions may lead to the discovery of more likely failures in practice.

Another common objective for most likely failure analysis is

$$f(\tau) = \rho(\tau, \psi) - \lambda \log(p(\tau)) \quad (4.8)$$

where λ is a weighting parameter selected by the user (algorithm 4.10). This objective is smooth and encourages the optimization algorithm to search simultaneously for trajectories that are both likely and close to failure.

```
function weighted_likelihood_objective(x, sys, ψ; smoothness=0.0, λ=1.0)
    s, x = extract(sys.env, x)
    τ = rollout(sys, s, x)
    s = [step.s for step in τ]
    p = NominalTrajectoryDistribution(sys, length(x))
    return robustness(s, ψ.formula, w=smoothness) - λ * log(pdf(p, τ))
end
```

Algorithm 4.10. Objective function that weights the tradeoff between robustness and likelihood. The function takes in a vector of real values x , a system sys , and a specification ψ . It returns a weighted combination of the smoothed robustness (or the robustness if $smoothness$ is set to 0) and the negative likelihood under the nominal trajectory distribution p .

4.6 Optimization Algorithms

We can search for failures by applying a variety of optimization algorithms to the optimization problem in equation (4.5).⁴ Algorithm 4.11 implements

⁴ M.J. Kochenderfer and T.A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019.

```

struct OptimizationBasedFalsification
    objective # objective function
    optimizer # optimization algorithm
end

function falsify(alg::OptimizationBasedFalsification, sys, ψ)
    f(x) = alg.objective(x, sys, ψ)
    return alg.optimizer(f, sys, ψ)
end

```

optimization-based falsification given an objective and optimization algorithm. It computes the system-specific objective function f , runs the optimizer, and returns its output. Example 4.5 applies algorithm 4.11 to find failures in the inverted pendulum problem using an off-the-shelf optimization package.⁵ The choice of optimization algorithm depends on the complexity of the system under test and the level of access to the system's internal model. The rest of this section outlines several categories of optimization algorithms and compares their advantages and disadvantages in the context of falsification.

One category of optimization techniques is *local descent methods*. Local descent methods start from an initial design point and incrementally improve it until some convergence criteria is met. At each iteration, they use a local model of the objective function at the current design point to determine a direction of improvement. They then take a step in this direction to compute the next design point. Some methods use the gradient or Hessian of the objective function with respect to the current design point to create the local model. These methods are called *first-order* and *second-order* methods, respectively. Figure 4.4 shows the result of applying a first-order method called gradient descent to find failures for the inverted pendulum example.

While the gradient and Hessian provide a very powerful signal for optimization algorithms, they are not always available.⁶ Some simulators do not provide access to the internal model of the system, making exact computation of the gradient infeasible. We often refer to such simulators as *black-box simulators*. Another category of optimization algorithms called *direct methods* is better suited for systems with black-box simulators. They traverse the input space using only information from function evaluations, eliminating the need for access to the system's internal model.

Algorithm 4.11. The optimization-based falsification algorithm for finding failures. The algorithm first computes a system-specific objective function f from a generic objective function **objective** (as specified in section 4.5). It then runs the **optimizer** and returns the results.

⁵ Off-the-shelf optimization packages provide implementations of a variety of optimization algorithms. One such package in the Julia ecosystem is `Optim.jl`.

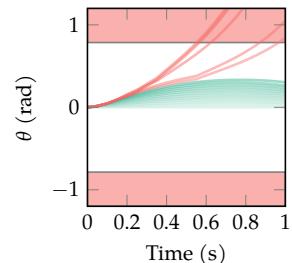


Figure 4.4. First-order method applied to falsify the inverted pendulum example. The plot shows successive iterations of the algorithm, with darker trajectories indicating later iterations. Failures are highlighted in red. The algorithm gets closer to a failure with each iteration until it eventually begins to find failures.

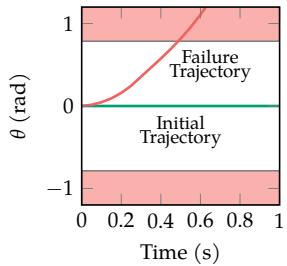
⁶ Gradient information is a strong enough signal to effectively optimize machine learning models with billions of parameters.

The `Optim.jl` package provides implementations of several optimization algorithms. In this example, we show how to use the `Optim.jl` implementation of a second-order method called L-BFGS to falsify the inverted pendulum system. We define the `optimizer` function for algorithm 4.11 and run the algorithm using the robustness objective as follows:

```
using Optim
function lbfgs(f, sys, ψ)
    x₀ = zeros(42)
    alg = Optim.LBFGS()
    options = Optim.Options(store_trace=true, extended_trace=true)
    results = optimize(f, x₀, alg, options; autodiff=:forward)
    ts = [rollout(sys, extract(sys.env, iter.metadata["x"]))...
          for iter in results.trace]
    return filter(t→isfailure(ψ, t), ts)
end
objective(x, sys, ψ) = robustness_objective(x, sys, ψ, smoothness=1.0)
alg = OptimizationBasedFalsification(objective, lbfgs)
failures = falsify(alg, inverted_pendulum, ψ)
```

In this implementation, we are optimizing over a disturbance trajectory with depth $d = 21$. Since each sensor disturbance is two-dimensional, the length of each design point is 42. The `lbfgs` function starts with an initial design point of all zeros, specifies options to store the results of each iteration, and runs the algorithm using `ForwardDiff.jl` to compute gradients. It then extracts the initial state and disturbance trajectory from each iteration and performs a rollout of the system. Finally, it filters the resulting trajectories to return failure trajectories. It is important that we specify the objective as smoothed robustness so that the gradients are well-defined. The plot on the right shows the progression of the algorithm. L-BFGS converges to a failure trajectory after a single iteration.

Example 4.5. Applying a second-order method called L-BFGS to falsify the inverted pendulum example. We use the open-source implementation of L-BFGS in the `Optim.jl` package. The plot shows the trajectory of the pendulum for the initial point (green) and the failure trajectory discovered after one iteration (red). For more information on the L-BFGS algorithm, see J. Nocedal, "Updating Quasi-Newton Matrices with Limited Storage," *Mathematics of Computation*, vol. 35, no. 151, pp. 773–782, 1980.



Local descent methods often get stuck in local optima. *Population methods* attempt to overcome this drawback by performing optimization using a collection of design points. The points in a population are sometimes referred to as *individuals*. Population methods begin with an initial population that is spread out over the design space. At each iteration, they use the current function value of each individual to move the population toward the optimum. Because population methods spread samples over the entire design space rather than incrementally improving a single point, they may find a more diverse set of failures. For example, the population method in figure 4.5 is able to find failures for the pendulum in both directions. High-dimensional problems with long time horizons may require a large number of samples to cover the design space. However, population methods are often easy to parallelize, which can improve efficiency.

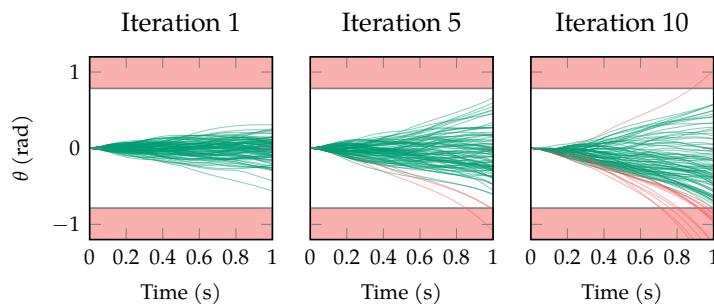


Figure 4.5. Population method applied to falsify the inverted pendulum example. The plots shows the trajectories for the individuals in the population at three iterations. Failures are highlighted in red. The individuals in the population get closer to a failure with each iteration, and the algorithm finds trajectories that fail in both the positive and negative direction.

4.7 Summary

- Monte Carlo falsification requires $1/p_{\text{fail}}$ samples on average to find a failure, which can be computationally expensive for systems with rare failure events.
- Optimization-based falsification algorithms use optimization techniques to find failures more efficiently.
- Disturbances are a useful concept for optimization-based falsification algorithms, and we can reformulate the distribution over trajectories for a system as a distribution over initial states and disturbances.

- We can formulate the falsification problem as an optimization problem by defining an objective function and optimizing over initial states and disturbances.
- We can apply a variety of optimization algorithms to search for failures of a system, and the choice of algorithm depends on the problem complexity and the availability of the system's internal model.

4.8 Exercises

Exercise 4.1. Suppose we have an aircraft collision avoidance system with a true probability of collision of 10^{-6} . We sample 10,000 simulated trajectories of the system. What is the probability that none of the trajectories result in a collision? How many trajectories would we need to sample to have a 95% chance of observing at least one collision?

Solution: The probability of a randomly sampled trajectory does not result in a collision is $1 - 10^{-6}$. Therefore, the probability that 10,000 randomly sampled trajectories do not result in a collision is $(1 - 10^{-6})^{10000} \approx 0.99$. If we want to have a 95% chance of observing at least one collision, we must have a 5% chance of observing no collisions:

$$(1 - 10^{-6})^m = 0.05$$

$$m \approx 3 \times 10^6$$

We would need to run around 3 million simulations to have a 95% chance of observing at least one collision.

Exercise 4.2. Consider a system with an agent that moves along a line where the actions control how far the agent moves in each time step. Specifically, the next state is the current state plus the action with uniform noise added as follows:

$$T(s' | s, a) = \mathcal{U}(s' | s + a - 0.2, s + a + 0.8)$$

Rewrite this transition model in a way that separates the deterministic and stochastic components. Specify the disturbances and the disturbance distribution.

Solution: The transition model can be rewritten as follows:

$$T(s, a, x_a) = s + a + x_s, \quad x_s \sim \mathcal{U}(\cdot | -0.2, 0.8)$$

The disturbance is x_s and the disturbance distribution is $\mathcal{U}(\cdot | -0.2, 0.8)$.

Exercise 4.3. Consider an agent operating in a two-dimensional grid world. The state $s = [x, y]$ represents the current position of the agent, and its actions are

$$\mathcal{A} = \{[-1, 0], [1, 0], [0, -1], [0, 1]\}$$

corresponding to the left, right, down, and up directions, respectively. Each action attempts to steer the agent in the corresponding direction. However, the grid world is slippery such that the agent moves in the intended direction with probability 0.7 and moves in each of the other directions with probability 0.1. Write the transition model for the agent in this grid world in a way that separates the deterministic and stochastic components. Specify the disturbances and the disturbance distribution.

Solution: The transition model can be written as follows:

$$T(s' | s, a) = s + x_s, \quad x_s \sim D_s(\cdot | s, a)$$

where

$$D_s(x_s | s, a) = \begin{cases} 0.7 & \text{if } x_s = a \\ 0.1 & \text{otherwise} \end{cases}$$

The disturbance is x_s and the disturbance distribution is $D_s(\cdot | s, a)$.

Exercise 4.4. Create a fuzzing distribution for the continuum world problem shown in algorithm A.8 following a similar process to the one used in example 4.3.

Solution: The following Julia code creates a fuzzing distribution for the continuum world problem:

```
struct ContinuumWorldFuzzingDistribution <: TrajectoryDistribution
    Σ₀ # environment disturbance covariance
    d # depth
end
function initial_state_distribution(p::ContinuumWorldFuzzingDistribution)
    return SetCategorical([[0.5, 0.5]])
end
function disturbance_distribution(p::ContinuumWorldFuzzingDistribution, t)
    D = DisturbanceDistribution((o)→Deterministic(),
                                (s,a)→MvNormal(zeros(2), p.Σ₀),
                                (s)→Deterministic())
    return D
end
depth(p::ContinuumWorldFuzzingDistribution) = p.d
```

The fuzzing distribution uses the same initial state distribution as the nominal initial state distribution defined in algorithm A.8. Because the only source of randomness for the continuum world system is noise applied to the state transitions, the disturbance distribution for the agent and sensor are both set to `Deterministic()`. The disturbance distribution for the environment is set to a multivariate normal distribution with covariance matrix that we can control to adjust the amount of noise in the environment. To increase the amount of failures we are able to find, we can increase the magnitude of the noise applied by the environment disturbance.

Exercise 4.5. Provide an example of an objective function for the optimization problem in equation (4.5) that we could use to find failures of a financial trading strategy that is required to lose no more than 1 percent of its value on each day of a 10 day period.

Solution: There are multiple possible answers. The objective should be some measure of the closeness to failure for the financial trading strategy. One possible objective function is the average return over the 10 day period. We could also formulate the specification for this system as a signal temporal logic specification as follows:

$$\psi = \square_{[0,10]}(R \geq -0.01)$$

where R is the return of the financial trading strategy. We can then use the robustness measure for this specification as the objective function for the optimization problem.

Exercise 4.6. Suppose we are trying to find the most likely failure of a system by minimizing the objective in algorithm 4.10. We find that the trajectory that the optimizer returns is not a failure. How should we change the weighting parameter λ to address this problem?

Solution: Because the weighting parameter λ is multiplied by the negative log likelihood of the trajectory, a lower value of λ will decrease the relative penalty for trajectories that are unlikely under the nominal model and increase the relative penalty on the robustness. Applying a high relative penalty on the robustness will encourage the optimization process to find failures. Therefore, we should decrease λ .

Exercise 4.7. If we cannot take the gradient of the objective function with respect to the initial state and disturbances, can we apply local descent methods to solve the optimization problem in equation (4.5)?

Solution: We can apply the subcategory of local descent methods called direct methods in this case. These methods traverse the input space using only information from function evaluations and therefore do not require the ability to compute the gradient of the objective function.

Exercise 4.8. Suppose that we are applying local descent methods to solve the optimization problem in equation (4.5) and we find that the optimizer keeps finding failures from a single failure mode. What should we do to encourage the optimizer to find failures from other failure modes?

Solution: There are multiple options. One option is to try initializing the local descent methods to different initial states to encourage the optimizer to explore different regions of the input space. Another option is to switch to a population method for optimization. Because these methods maintain an entire population of candidate solutions, they tend to produce a more diverse set of failures. We can also add a constraint or penalty that prevents converging near solutions that have already been found.

5 Falsification through Planning

The methods in the previous chapter find counterexamples by performing optimization over full trajectories. In many cases, we can increase efficiency by considering a sequence of partial trajectories. In particular, this chapter discusses methods that use planning algorithms to account for the temporal aspect of the problem. Planning techniques break the falsification problem into a sequence of smaller problems. We discuss several categories of planning algorithms that rely on optimization, search, and reinforcement learning.

5.1 Shooting Methods

Shooting methods use optimization to find a feasible path between two points,¹ and they can be used in the context of falsification to produce feasible failure trajectories. These methods break the trajectory optimization problem into a set of smaller problems by optimizing over a sequence of trajectory segments. A trajectory τ can be partitioned into n segments such that $\tau = (\tau_1, \dots, \tau_n)$. Each trajectory segment τ_i is defined by an initial state s_i and a sequence of disturbances x_i of length d_i . Given s_i and x_i , we can compute the resulting trajectory τ_i by performing a rollout.

The *defect* between two trajectory segments is the distance between the final state of the first segment and the initial state of the second segment. A set of trajectory segments forms a feasible trajectory if the defect of all consecutive trajectory segments is 0. In other words, the final state of τ_i must match the initial state of τ_{i+1} for all $i \in \{1, \dots, n - 1\}$. This requirement leads to the following

¹The term shooting method is based on the analogy of shooting at a target from a cannon. Shooting methods start at an initial point and “shoot” trajectories toward a target point until a feasible path between the initial point and target is found. Shooting methods originated from research on boundary value problems. A more detailed review with an implementation can be found in section 18.1 of the reference by W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007.

optimization problem:

$$\begin{aligned} & \underset{s_1, x_1, \dots, s_n, x_n}{\text{minimize}} \quad f(\tau_1, \dots, \tau_n) \\ & \text{subject to} \quad \tau_i = \text{Rollout}(s_i, x_i) \quad \text{for all } i \in \{1, \dots, n\} \\ & \qquad \qquad \qquad \text{Defect}(\tau_i, \tau_{i+1}) = 0 \text{ for all } i \in \{1, \dots, n-1\} \end{aligned} \tag{5.1}$$

where f is the falsification objective (see section 4.5). If $n = 1$, the optimization problem is equivalent to optimizing over the entire trajectory, and equation (5.1) reduces to equation (4.5). This process is referred to as *single shooting*. For $n > 1$, this process is referred to as *multiple shooting*.

Multiple shooting seemingly increases the complexity of the optimization problem by adding more variables and constraints, but it can actually improve efficiency, especially for systems in which small changes in the inputs applied at the beginning of a trajectory have a significant effect on the end of the trajectory. For example, consider the problem of finding a path through a maze where one wrong turn at the beginning of the trajectory could ultimately lead to a dead end. If we use single shooting, we must optimize over the entire path at once. If we use multiple shooting, we can break the path into segments that focus on different regions of the maze.

```
defect(t_i, t_{i+1}) = norm(t_{i+1}[1].s - t_i[end].s)

function shooting_robustness(x, sys, ψ; smoothness=0.0, λ=1.0)
    segments = extract(sys.env, x)
    n = length(segments)
    τ_segments = [rollout(sys, seg.s, seg.x) for seg in segments]
    τ = vcat(τ_segments...)
    s = [step.s for step in τ]
    ρ = smooth_robustness(s, ψ.formula, w=smoothness)
    defects = [defect(τ_segments[i], τ_segments[i+1]) for i in 1:n-1]
    return ρ + λ*sum(defects)
end
```

Algorithm 5.1. Temporal logic robustness objective for multiple shooting. The function takes in a vector of real values x , a system sys , and a specification ψ and returns the objective in equation (5.2). The `smoothness` parameter controls the smoothness of the robustness function, and λ controls the weighting of the defect penalty. The `defect` function computes the defect between two trajectory segments. The `extract` function extracts the trajectory segments from the vector x and is system specific.

The constraint on the defect of consecutive trajectory segments in equation (5.1) poses challenges for many optimization algorithms. In practice, we instead incorporate it as a soft constraint by adding it as a penalty in the objective:

$$\begin{aligned} & \underset{s_1, x_1, \dots, s_n, x_n}{\text{minimize}} \quad f(\tau_1, \dots, \tau_n) + \lambda \sum_{i=1}^{n-1} \text{Defect}(\tau_i, \tau_{i+1}) \\ & \text{subject to} \quad \tau_i = \text{Rollout}(s_i, x_i) \quad \text{for all } i \in \{1, \dots, n\} \end{aligned} \tag{5.2}$$

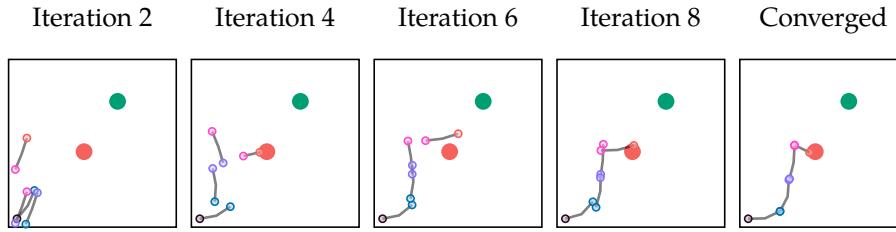


Figure 5.1. Multiple shooting applied to the continuum world example to find a path from an initial point to the obstacle. We use four trajectory segments, and the colors denote which segment end points should connect. The plots show the trajectory segments at different iterations of the L-BFGS optimization algorithm.

where λ is a weighting parameter that controls how heavily the defect is penalized. Algorithm 5.1 implements this objective when f is the temporal logic robustness.

We can apply any of the optimization algorithms discussed in section 4.6 to the optimization problem in equation (5.2). Compared to the optimization problems in the previous chapter, minimizing the defect between the trajectory segments adds complexity to the problem. This added complexity can make it more difficult to find a feasible failure trajectory. Figure 5.1 shows an example that uses a gradient-based optimization technique called L-BFGS² to find a failure trajectory for the continuum world problem. For systems with black-box simulators, the direct methods described in section 4.6 may struggle to find feasible failure trajectories. Instead, we can use direct methods that were designed specifically for multiple shooting.³

5.2 Tree Search

Tree search algorithms iteratively construct a tree structure that represents the space of possible trajectories. Each node in the tree represents a state, and each edge represents a transition between states that is the result of applying a particular disturbance. Each path through the tree corresponds to a feasible trajectory for the system. Tree search algorithms start in an initial state and iteratively grow the tree in an attempt to find feasible failure trajectories.

At each iteration, these algorithms perform the steps illustrated in figure 5.2. They first select a node from the tree to extend. This selection is typically based on a heuristic designed to grow the tree toward failures. Next, they extend the selected node by choosing a disturbance and adding a new child node at the resulting next state. We can terminate the algorithm after a fixed number of iterations or when a failure trajectory is discovered.

²J. Nocedal, “Updating Quasi-Newton Matrices with Limited Storage,” *Mathematics of Computation*, vol. 35, no. 151, pp. 773–782, 1980.

³For an example of a multiple shooting algorithm designed for systems with black-box simulators, see A. Zutshi, J. V. Deshmukh, S. Sankaranarayanan, and J. Kapinski, “Multiple Shooting, CEGAR-Based Falsification for Hybrid Systems,” in *International Conference on Embedded Software*, 2014.

Algorithm 5.2 implements the generic tree search algorithm. It runs for a fixed number of iterations before returning all failures in the tree. Algorithm 5.3 extracts failure trajectories from a tree by enumerating all paths in the tree and checking for failures. Specific implementations of tree search algorithms differ in how they implement the select and extend functions. We discuss two categories of tree search algorithms in the next two sections.

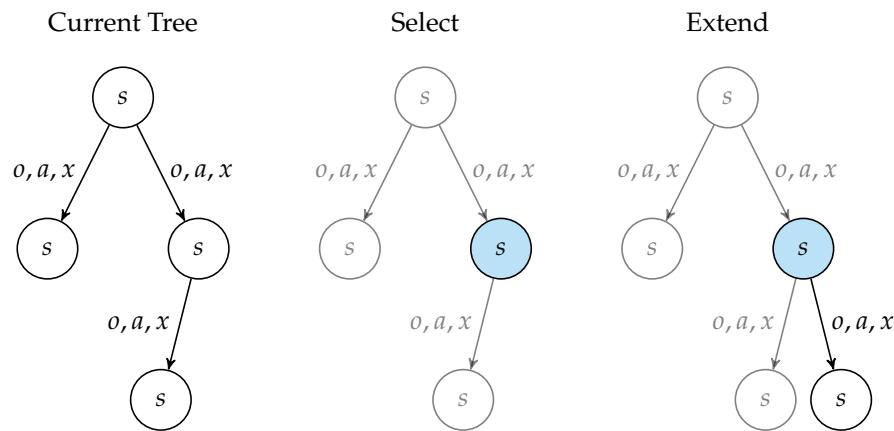


Figure 5.2. One iteration of tree search. The nodes of the tree represent states s . Given a disturbance x , we produce an observation o and an action a that lead us to the next node. The edges represent these transitions. The algorithm first selects a node from the tree to extend. It then chooses a disturbance to extend the selected node and adds the resulting next state as a new child.

```

abstract type TreeSearch end

function falsify(alg::TreeSearch, sys, ψ)
    tree = initialize_tree(alg, sys)
    for i in 1:k_max
        node = select(alg, sys, ψ, tree)
        extend!(alg, sys, ψ, tree, node)
    end
    return failures(tree, sys, ψ)
end

```

Algorithm 5.2. Generic tree search algorithm for finding failure trajectories. The algorithm starts by initializing a tree with a single node. It then iteratively selects a node from the tree using the `select` function and adds to its children using the `extend!` function. After `k_max` iterations, the algorithm returns the set of failure trajectories in the tree.

5.3 Heuristic Search

Some tree search algorithms use heuristics to explore the space of possible trajectories. The *rapidly exploring random trees (RRT)* algorithm, for example, uses heuristics to iteratively extend the search tree toward randomly selected states in

```

function trajectory(node)
    τ = []
    while !isnothing(node.parent)
        pushfirst!(τ, (s=node.parent.state, node.edge...))
        node = node.parent
    end
    return τ
end

function failures(tree, sys, ψ)
    leaves = filter(node → isempty(node.children), tree)
    ts = [trajectory(node) for node in leaves]
    return filter(τ → isfailure(ψ, τ), ts)
end

```

Algorithm 5.3. Functions for extracting failure trajectories from a tree. The `failures` function first finds the leaves of the tree and extracts the corresponding trajectory from each leaf using the `trajectory` function. The `trajectory` function starts at a leaf node and propagates backward through the tree to construct the full trajectory. The `failures` function then filters these trajectories for failures and returns the result.

the state space.⁴ In the context of falsification, we use RRT to efficiently explore the space of possible disturbance trajectories in search of a failure trajectory.

Algorithm 5.4 implements the select and extend steps for the RRT algorithm. In the select step, RRT randomly samples a goal state and computes an objective value for each node in the current tree based on the sampled goal state. This objective is typically related to the distance between each node and the goal state. The algorithm then selects the node with the lowest objective value to pass to the extend step. In the extend step, RRT selects a disturbance, simulates one step forward in time from the selected node, and adds the resulting edge and child node to the tree.

Several variants of RRT differ in how they sample goal states, compute objectives, and select disturbances. Algorithm 5.5 implements a version of the RRT algorithm that samples goal states uniformly from the state space. It then uses the Euclidean distance between the each node and the goal state as the objective. In the extend step, the disturbance is randomly sampled from the nominal disturbance distribution for the system. Example 5.1 applies this algorithm to the continuum world problem.

⁴ RRT was designed to efficiently enumerate trajectories in high-dimensional spaces, particularly for systems with complex dynamics. The algorithm was originally proposed in the context of robotic path planning. For more information on path planning algorithms, see S. LaValle, “Planning Algorithms,” Cambridge University Press, vol. 2, pp. 3671–3678, 2006.

5.3.1 Goal Heuristics

Algorithm 5.5 uses the sampled goal state to select which node in the tree to extend, but it does not use the goal state when selecting the disturbance used to extend the selected node. We can improve the performance of the algorithm by

```

struct RRT <: TreeSearch
    sample_goal      # sgoal = sample_goal(tree)
    compute_objectives # objectives = compute_objectives(tree, sgoal)
    select_disturbance # x = select_disturbance(sys, node)
    k_max           # number of iterations
end

mutable struct RRTNode
    state        # node state
    parent       # parent node
    edge         # (o, a, x)
    children     # vector of child nodes
    goal_state   # current goal state
end

function initialize_tree(alg::RRT, sys)
    return [RRTNode(rand(Ps(sys.env))), nothing, nothing, [], nothing]
end

function select(alg::RRT, sys, ψ, tree)
    sgoal = alg.sample_goal(tree)
    objectives = alg.compute_objectives(tree, sgoal)
    node = tree[argmin(objectives)]
    node.goal_state = sgoal
    return node
end

function extend!(alg::RRT, sys, ψ, tree, node)
    x = alg.select_disturbance(sys, node)
    o, a, s' = step(sys, node.state, x)
    snew = RRTNode(s', node, (; o, a, x), [], nothing)
    push!(node.children, snew)
    push!(tree, snew)
end

```

Algorithm 5.4. The rapidly exploring random trees algorithm. The algorithm is a type of tree search algorithm and implements both the `select` and `extend!` functions. The `select` function samples a goal state according to the `sample_goal` function and computes an objective value for each node in the tree using the `compute_objectives` function. It then selects the node with the lowest objective value, sets its goal state, and returns it. The `extend!` function selects a disturbance according to the `select_disturbance` function, simulates one step forward in time, and adds the results to the tree in the form of a new child node.

```

random_goal(tree, lo, hi) = rand.(Distributions.Uniform.(lo, hi))

function distance_objectives(tree, sgoal)
    return [norm(sgoal .- node.state) for node in tree]
end

function random_disturbance(sys, node)
    D = DisturbanceDistribution(sys)
    o, a, s', x = step(sys, node.state, D)
    return x
end

```

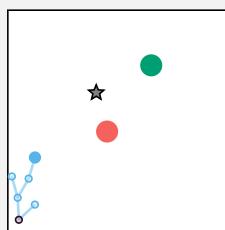
Algorithm 5.5. Functions for the RRT algorithm. The first function samples a goal state uniformly from the state space. The `lo` and `hi` inputs specify the lower and upper bounds of the state variables. The second function computes the Euclidean distance between each node in the tree and the goal state. The third function samples a disturbance from the nominal disturbance distribution for the system.

Suppose we want to apply RRT to search for failures for the continuum world system. We can use the following code to run the basic RRT algorithm for 100 iterations.

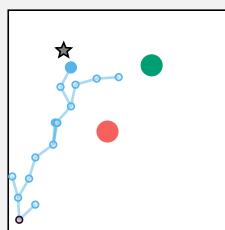
```
select_goal(tree) = random_goal(tree, [0.0, 0.0], [10.0, 10.0])
compute_objectives(tree, sgoal) = distance_objectives(tree, sgoal)
select_disturbance(tree, node) = random_disturbance(tree, node)
alg = RRT(select_goal, compute_objectives, select_disturbance, 100)
failures = falsify(alg, cw, ψ)
```

The plots below show two snapshots of the search tree after 5 and 15 iterations as well as the final tree after 100 iterations. After 100 iterations, RRT did not find any failure trajectories. Although goal states are sampled throughout the state space, the disturbances are sampled from the nominal disturbance distribution. Since the nominal disturbance distribution represents only small deviations from the nominal path, the tree closely follows the nominal path toward the goal. We can improve the performance of the tree search using the heuristics discussed in section 5.3.1.

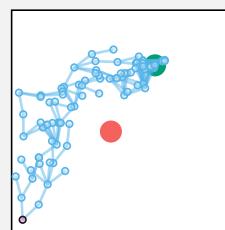
Iteration 5



Iteration 15



Iteration 100



Example 5.1. Basic RRT applied to the continuum world example. The plots show snapshots of the search tree after 5, 15, and 100 iterations. The stars show the next goal state and highlighted nodes show the node selected to extend next.

using the goal state to select the disturbance. Specifically, we want to select the disturbance that leads to the next state that is closest to the goal state.

```
function goal_disturbance(sys, node; m=10)
    D = DisturbanceDistribution(sys)
    steps = [step(sys, node.state, D) for i in 1:m]
    distances = [norm(node.goal_state - step.s') for step in steps]
    return steps[argmin(distances)].x
end
```

Algorithm 5.6 uses sampling to search for a disturbance that results in a next state that is close to the goal state. It draws m samples from the nominal disturbance distribution and simulates one step forward in time from the current node using each sample. It then returns the disturbance that results in the next state that is closest to the goal state. As m increases, the performance of the algorithm improves but at a greater computational cost.⁵ Figure 5.3 demonstrates this process on one step of RRT for the continuum world problem.

Algorithm 5.6. Function for selecting a disturbance that leads to the next state that is closest to the goal state. The algorithm takes m steps using the nominal disturbance distribution. It then computes the distances between the next state from each step and the goal state. It returns the disturbance that resulted in the lowest distance.

⁵ To improve performance and efficiency, more sophisticated optimization algorithms can also be used (see section 4.6).

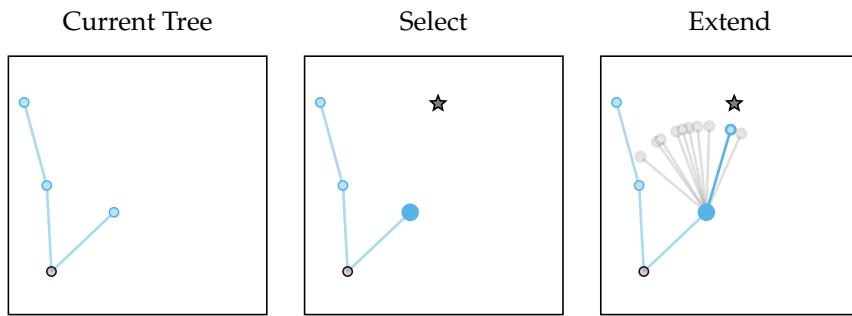


Figure 5.3. One iteration of RRT applied to the continuum world example using algorithm 5.6 with $m = 10$ to select the disturbance in the extend step. The algorithm selects the node that is closest to the goal state to extend and samples 10 disturbances to add to the tree. It then selects the disturbance that results in the next state that is closest to the goal state and adds the resulting edge and child node to the tree.

In addition to improving the extend step of algorithm 5.5, we can improve the select step by modifying how we sample the goal state. Instead of sampling the goal state uniformly from the state space, we can use a heuristic to sample goal states that are more likely to grow the tree toward failures. One technique is to identify a *failure region* in the state space and sample goal states from this region. A failure region is a region such that any trajectory that passes through this region is a failure trajectory. For example, the failure region in the continuum world problem is the set of states within the red obstacle. This technique is limited to systems with specifications that depend only on the state. For specifications of

The failure region for the continuum world example is the set of states within the red obstacle, which is a circle centered at $(4.5, 4.5)$ with radius 0.5. We can uniformly sample from this region using the following code:

```
function failure_goal(tree)
    r = rand(Uniform(0, 0.5))
    θ = rand(Uniform(0, 2π))
    return [4.5, 4.5] .+ [r*cos(θ), r*sin(θ)]
end
```

The code uniformly samples a radius between 0 and 5 and an angle between 0 and 2π . It then converts these samples to a state in the failure region.

Example 5.2. Example of sampling goal states from the failure region of the continuum world problem.

temporal properties, identifying a failure region is not possible without augmenting the state space. Figure 5.4 shows the result of using this heuristic along with algorithm 5.6 to apply RRT to the continuum world problem.

5.3.2 Coverage Heuristics

To uncover a variety of ways in which a system might fail, it is important that we explore a diverse set of trajectories. We incorporate this idea into RRT using heuristics that are designed to maximize *coverage* of the state space. We assess coverage using *coverage metrics*, which measure how well a set of samples fill a given space. In the context of tree search, we are interested in how well the states represented by the nodes in the tree fill the state space. We can then use these coverage metrics in the select step to select the next goal state.

One common coverage metric is related to the concept of *dispersion*. The dispersion of a set of points \mathcal{V} in the bounded region \mathcal{S} is the radius of the largest ball that can be placed in \mathcal{S} such that no point in \mathcal{V} lies within the ball, written as

$$\text{dispersion} = \sup_{s \in \mathcal{S}} \left(\min_{s_i \in \mathcal{V}} \|s - s_i\| \right) \quad (5.3)$$

where the outer optimization represents the *supremum*. A supremum is a generalization of a maximum that allows solutions to exist when the largest ball merely approaches a particular size before containing one of the points in \mathcal{V} . The norm in equation (5.3) can be any norm. A common choice is the ℓ_2 -norm. Coverage is inversely related to dispersion. In other words, a set of points with high dispersion will have low coverage (see figure 5.5).

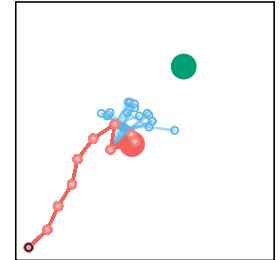


Figure 5.4. RRT applied to the continuum world problem using algorithm 5.6 with $m = 10$ to select the disturbance in the extend step and goal states sampled from the failure region. The algorithm was run for 100 iterations and discovered the failure trajectories highlighted in red.

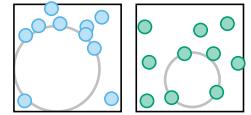


Figure 5.5. Visualization of dispersion for two different sets of 10 points. The blue set does not fill the space as well as the green set. We can find a larger ball that does not contain any points in the blue set than we can for the green set. Therefore, the blue set has higher dispersion and lower coverage.

Since dispersion considers only the largest ball that can be placed in \mathcal{S} , it tends to be a conservative measure of coverage. Furthermore, it is difficult to compute for high-dimensional spaces. An approximate metric called *average dispersion* overcomes these drawbacks.⁶ Average dispersion is computed on a grid of n points with spacing δ in each dimension. It is calculated as

$$\text{average dispersion} = \frac{1}{n} \sum_{j=1}^n \frac{\min(d_j(\mathcal{V}), \delta)}{\delta} \quad (5.4)$$

where $d_j(\mathcal{V})$ is the distance from the j th grid point to the nearest point in \mathcal{V} .

```
function average_dispersion(points, lo, hi, lengths)
    points_norm = [(point .- lo) ./ (hi .- lo) for point in points]
    ranges = [range(0, 1, length) for length in lengths]
    δ = minimum(Float64(r.step) for r in ranges)
    grid_dispersions = []
    for grid_point in Iterators.product(ranges...)
        dmin = minimum(norm(grid_point .- p) for p in points_norm)
        push!(grid_dispersions, min(dmin, δ) / δ)
    end
    return mean(grid_dispersions)
end
```

Algorithm 5.7 computes average dispersion given a set of points and a bounded region. The term in the numerator of equation (5.4) is the radius of the largest ball centered at each grid point that does not contain any points in \mathcal{V} or other grid points. Dividing by δ ensures that the values for average dispersion range between 0 and 1, and subtracting the average dispersion from 1 results in a coverage metric that ranges between 0 and 1.⁷ Figure 5.6 shows the difference between dispersion and average dispersion.

Another common coverage metric is *discrepancy*. The key insight behind discrepancy is that if a set of points covers a space evenly, then a randomly chosen subset of the space should contain a fraction of samples proportional to the fraction of volume occupied by the subset. Discrepancy is defined as the worst case hyperrectangular subset:

$$\text{discrepancy} = \sup_{\mathcal{H} \subseteq \mathcal{S}} \left| \frac{\#(\mathcal{V} \cap \mathcal{H})}{\#(\mathcal{V})} - \frac{\text{vol}(\mathcal{H})}{\text{vol}(\mathcal{S})} \right| \quad (5.5)$$

⁶ J. M. Esposito, J. Kim, and V. Kumar, "Adaptive RRTs for Validating Hybrid Robotic Control Systems," in *Algorithmic Foundations of Robotics*, Springer, 2005, pp. 107–121.

Algorithm 5.7. Algorithm for computing average dispersion of a set of `points` on a space bounded by `lo` and `hi`. It uses a grid specified by `lengths`, which contains the number of grid points in each dimension. The algorithm first normalizes the points to lie in the unit hypercube. It then creates the grid over the unit hypercube and computes the average dispersion using equation (5.4).

⁷ The average dispersion coverage metric will be 1 if \mathcal{V} contains all of the grid points. A finer grid will result in better coverage estimates but at a greater computational cost.

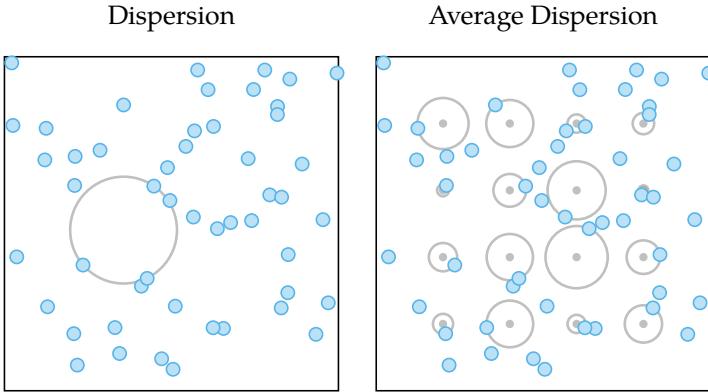


Figure 5.6. Visualization of the difference between dispersion and average dispersion on a set of points. While dispersion finds the largest ball that does not contain any points in the set, average dispersion operates on a grid. The gray dots indicate grid points, and the circles show the largest ball that does not contain any grid points or points in the set. The average dispersion is the average of the radii of these circles normalized by the grid spacing.

where \mathcal{H} is a hyperrectangular subset of \mathcal{S} and $\#(\mathcal{V} \cap \mathcal{H})$ and $\#(\mathcal{V})$ are the number of points in \mathcal{V} that lie in \mathcal{H} and the total number of points in \mathcal{V} respectively. We use $\text{vol}(\mathcal{H})$ and $\text{vol}(\mathcal{S})$ to denote the n -dimensional volume of \mathcal{H} and \mathcal{S} respectively, which can be obtained by multiplying the side lengths.

The worst-case hyperrectangle that determines the discrepancy of a set of points is typically a small region containing many points or a large region with few points. Figure 5.7 visualizes the discrepancy metric. Discrepancy approaches 1 when all points overlap and approaches 0 when all possible hyperrectangular subsets have their proper share of points. In general, discrepancy is difficult to compute exactly, especially in high dimensions.

Star discrepancy is a special case of discrepancy that is easier to compute and is often used in practice. Instead of considering all possible hyperrectangular subsets, star discrepancy considers only hyperrectangular subsets of the unit hypercube that have a vertex at the origin. We can always normalize any hyperrectangular space \mathcal{S} to the unit hypercube by dividing by the side length in each dimension. Given these constraints, it is possible to compute lower and upper bounds on star discrepancy.⁸ We first partition the unit hypercube \mathcal{B} into a finite number of

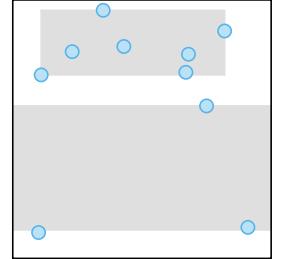


Figure 5.7. Visualization of the discrepancy metric. The rectangles indicate two candidates for the worst case rectangle used to define discrepancy. Discrepancy is determined by a rectangle with small area and many points (top) or a rectangle with large area and few points (bottom).

⁸ E. Thiémard, “An Algorithm to Compute Bounds for the Star Discrepancy,” *Journal of Complexity*, vol. 17, no. 4, pp. 850–880, 2001. Examples of other approximations can be found in Y.-D. Zhou, K.-T. Fang, and J.-H. Ning, “Mixture Discrepancy for Quasi-Random Point Sets,” *Journal of Complexity*, vol. 29, no. 3-4, pp. 283–301, 2013.

subrectangles $h \in \Pi$. We then compute the bounds as

$$\begin{aligned} \text{upper} &= \max_{h \in \Pi} \left(\max \left(\frac{\#(\mathcal{V} \cap h^+)}{\#(\mathcal{V})} - \frac{\text{vol}(h^-)}{\text{vol}(\mathcal{B})}, \frac{\text{vol}(h^+)}{\text{vol}(\mathcal{B})} - \frac{\#(\mathcal{V} \cap h^-)}{\#(\mathcal{V})} \right) \right) \\ \text{lower} &= \max_{h \in \Pi} \left(\max \left(\left| \frac{\#(\mathcal{V} \cap h^-)}{\#(\mathcal{V})} - \frac{\text{vol}(h^-)}{\text{vol}(\mathcal{B})} \right|, \left| \frac{\text{vol}(h^+)}{\text{vol}(\mathcal{B})} - \frac{\#(\mathcal{V} \cap h^+)}{\#(\mathcal{V})} \right| \right) \right) \end{aligned} \quad (5.6)$$

where h^+ and h^- are hyperrectangular subsets derived from subrectangle h as shown in figure 5.8.

The tightness of the upper and lower bounds in equation (5.6) depends on the resolution of the partition. Finer partitions will lead to tighter bounds at a greater computational cost. Algorithm 5.8 computes upper and lower bounds on star discrepancy using equation (5.6) given a set of points and a bounded region. We can subtract the value of star discrepancy from 1 to provide a coverage metric that ranges between 0 and 1. Figure 5.9 shows the upper and lower bounds on star discrepancy for the sets of points in figure 5.5 as the resolution of the partition is increased.

We can use average dispersion or star discrepancy as a metric to select the goal state in RRT. In particular, we want to select the goal state that would result in the greatest increase in coverage if added to the current tree. While it is difficult to determine the goal state exactly, we can approximate this process by drawing samples from the state space, computing the difference in coverage for each sample, and selecting the sample with the largest increase.⁹ The samples may be selected from a grid (figure 5.10) or drawn uniformly from the state space (example 5.3).

Coverage metrics can also be used as termination conditions. It is not always clear when to terminate tree search algorithms, especially if no failures are found. One option is to terminate the search when state space coverage is sufficient. Since not all states are necessarily reachable, coverage will not necessarily approach 1 as the number of tree search iterations increases. We therefore cannot use the magnitude of coverage as a termination condition by itself. Instead, we compute a growth metric such as

$$\text{growth} = \frac{\text{Coverage}(\mathcal{V}') - \text{Coverage}(\mathcal{V})}{\#(\mathcal{V}') - \#(\mathcal{V})} \quad (5.7)$$

where \mathcal{V} and \mathcal{V}' are the sets of points in the tree at the beginning and end of the current iteration.

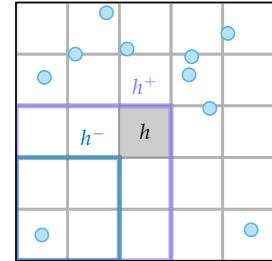


Figure 5.8. Visualization of the hyperrectangular subsets for subrectangle h used to compute upper and lower bounds on star discrepancy.

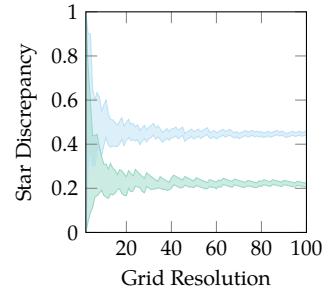


Figure 5.9. Upper and lower bounds on star discrepancy for the sets of points in figure 5.5. The grid resolution is the number of grid points in each dimension. The upper and lower bounds approach each other as the grid resolution increases. The green set of points is more evenly distributed than the blue set, so it has lower star discrepancy.

⁹ High-dimensional problems may require more sophisticated techniques. T. Dang and T. Nahhal, "Coverage-Guided Test Generation for Continuous and Hybrid Systems," *Formal Methods in System Design*, vol. 34, pp. 183–213, 2009.

```

function star_discrepancy(points, lo, hi, lengths)
    n, dim = length(points), length(lo)
    ℙ = [(point .- lo) ./ (hi .- lo) for point in points]
    ranges = [range(0, 1, length)[1:end-1] for length in lengths]
    steps = [Float64(r.step) for r in ranges]
    ℬ = Hyperrectangle(low=zeros(dim), high=ones(dim))
    lbs, ubs = [], []
    for grid_point in Iterators.product(ranges...)
        h⁻ = Hyperrectangle(low=zeros(dim), high=[grid_point...])
        h⁺ = Hyperrectangle(low=zeros(dim), high=grid_point .+ steps)
        ℙh⁻ = length(filter(v → v ∈ h⁻, ℙ))
        ℙh⁺ = length(filter(v → v ∈ h⁺, ℙ))
        push!(lbs, max(abs(ℙh⁻ / n - volume(h⁻) / volume(ℬ)),
                      abs(ℙh⁺ / n - volume(h⁺) / volume(ℬ))))
        push!(ubs, max(ℙh⁺ / n - volume(h⁻) / volume(ℬ),
                      volume(h⁺) / volume(ℬ) - ℙh⁻ / n))
    end
    return maximum(lbs), maximum(ubs)
end

```

Algorithm 5.8. Algorithm for computing upper and lower bounds on the star discrepancy of a set of `points` on a space bounded by `lo` and `hi`. It uses a partition specified by `lengths`, which contains the number of subrectangles in each dimension. The algorithm first normalizes the points to lie in the unit hypercube. It then creates the partition over the unit hypercube and computes the upper and lower bounds on star discrepancy using equation (5.6). We use the `LazySets.jl` package to represent hyperrectangles.

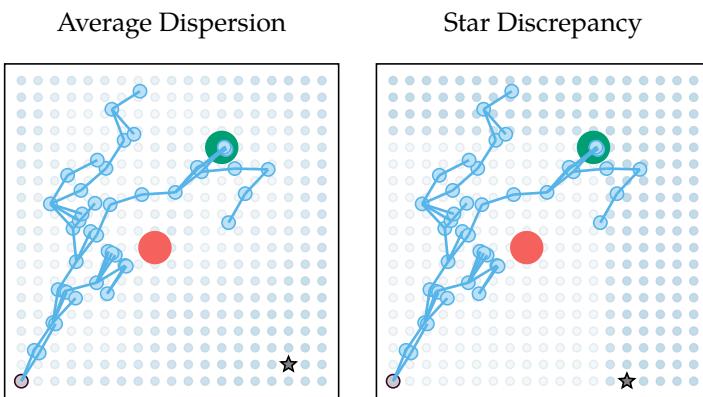
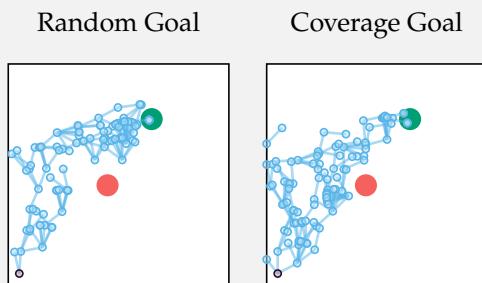


Figure 5.10. Selecting the next goal state for RRT applied to the continuum world problem using average dispersion and star discrepancy coverage metrics. The plots show the grid points used as candidates for the next goal state. The color of each grid point indicates the increase in coverage that would result from adding that grid point to the tree with darker colors indicating a greater increase. For star discrepancy, the colors represent the lower bound. The star indicates the goal state selected by RRT. Because star discrepancy only focuses on the worst-case hyperrectangle, it is not as smooth as average dispersion.

Suppose we want to apply coverage heuristics when using RRT on the continuum world problem. The following code implements a version of the `select_goal` function that uses coverage based on average dispersion to guide the search.

```
function select_goal(tree; m=5)
    a, b, lengths = [0, 0], [10, 10], [10, 10]
    points = [node.state for node in tree]
    sgoals = [rand.(Distributions.Uniform.(a, b)) for _ in 1:m]
    dispersions = [average_dispersion([points..., sgoal], a, b, lengths)
                  for sgoal in sgoals]
    coverages = 1 ./ dispersions
    return sgoals[argmax(coverages)]
end
```

We first collect the states visited so far from the nodes of the tree and sample m potential goal states uniformly from the state space. We then compute the new average dispersion if each goal state were added to the tree. The goal state that results in the greatest increase in coverage is selected. The plots show the resulting trees when using random goals and coverage-based goals. Using the coverage-based goal selection results in a wider tree that covers more of the state space.



Example 5.3. RRT applied to the continuum world problem using coverage heuristics. The plots illustrate the effect of selecting the next goal state based on coverage rather than randomly selecting it.

We can terminate the tree search when the growth metric is sufficiently small. It is important to note that this growth metric does not provide any guarantees about the coverage of the search tree. Even when growth is small, there may still be unexplored regions of the state space that are reachable under rare circumstances. For example, every state in the continuum world problem is reachable through a sequence of disturbances, but the average dispersion coverage metric plateaus at a number less than 1 (see figure 5.11). Some states are extremely unlikely to be reached under the nominal disturbance model.

5.3.3 Alternative Objectives

As noted in the previous chapter, we may want to go beyond a simple search for failures and incorporate other objectives into the search process. For example, we may be interested in finding the shortest path to failure or the most likely failure. We can incorporate these objectives into RRT by modifying how we compute the objectives in the select step (algorithm 5.9).

First, we define a cost function c that maps a node to a cost of transitioning to the node from its parent. For example, the cost might be a measure of the distance between the node's state and its parent's state. To ensure that the tree search algorithm is still encouraged to reach the goal, all costs must be positive. The total cost of a path is the sum of the costs of all nodes in the path. Our goal is to find the path to the goal with the lowest total cost.

We compute an objective for each node consisting of two components: the total cost of the current path from the root to the node and an estimate of the remaining cost to get from the node to the goal state. The remaining cost estimate comes from a heuristic function h . One potential heuristic is the distance from the current node to the goal state. Algorithm 5.9 implements this process given a cost function and heuristic function.¹⁰ It provides default cost and heuristic functions that will guide the search toward the shortest path.

To search for the most likely failure, we can use a cost function related to the negative log likelihood of the disturbance for the current node. We add a constant factor of the maximum possible log likelihood according to the disturbance distribution to ensure that the cost is positive. For the heuristic function, we need to estimate the log likelihood of the remaining path required to reach the goal state. One option is to use the distance to the goal state as a proxy for this value since longer paths tend to result in lower log likelihoods. Adding a scaling factor to the

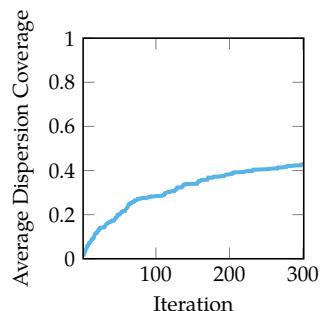


Figure 5.11. The average dispersion coverage metric over iterations of RRT applied to the continuum world problem.

¹⁰This algorithm is a simplified version of the RRT* algorithm. S. Karaman and E. Frazzoli, “Incremental Sampling-Based Algorithms for Optimal Motion Planning,” *Robotics Science and Systems VI*, vol. 104, no. 2, pp. 267–274, 2010.

```

distance_c(node) = norm(node.parent.state .- node.state)
distance_h(node, sgoal) = norm(sgoal .- node.state)

function cost_objectives(tree, sgoal; c=distance_c, h=distance_h)
    costs = Dict()
    queue = [tree[1]]
    while !isempty(queue)
        node = popfirst!(queue)
        if isnothing(node.parent)
            costs[node] = 0.0
        else
            costs[node] = c(node) + costs[node.parent]
        end
        for child in node.children
            push!(queue, child)
        end
    end
    heuristics = [h(sgoal, node) for node in tree]
    objectives = [costs[node] for node in tree] .+ heuristics
    return objectives
end

```

cost function to balance between the heuristic and cost may improve performance. Figure 5.12 shows the results from using RRT to find the shortest path to failure and most likely failure for the continuum world problem.

While algorithm 5.9 will often find a low cost path to failure, it is not necessarily guaranteed to find the path with the lowest possible cost. Certain conditions on the nature of the problem and the heuristic function are required to guarantee optimality. Algorithm 5.9 will converge to the optimal path if the state space and disturbance space are discrete and the heuristic function is *admissible*.¹¹ A heuristic is admissible if it is guaranteed to never overestimate the cost of reaching the goal state. In shortest path problems, the straight-line distance to the goal state is an admissible heuristic. Example 5.4 demonstrates this result on the grid world problem.

5.4 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) (algorithm 5.10) is a tree search algorithm that balances between *exploration* and *exploitation*.¹² It explores by selecting nodes that have not been visited many times and exploits by biasing the search tree toward paths that seem most promising. MCTS determines which paths are most

Algorithm 5.9. Algorithm for computing objectives based on a cost function c and heuristic function h . The algorithm first traverses the tree and accumulates the total cost of each node. It then computes the heuristic for each node and adds it to the total cost to get the objective values. We supply default implementations of the cost and heuristic functions that will encourage RRT to search for the shortest path.

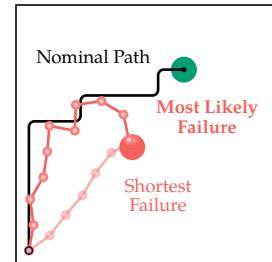


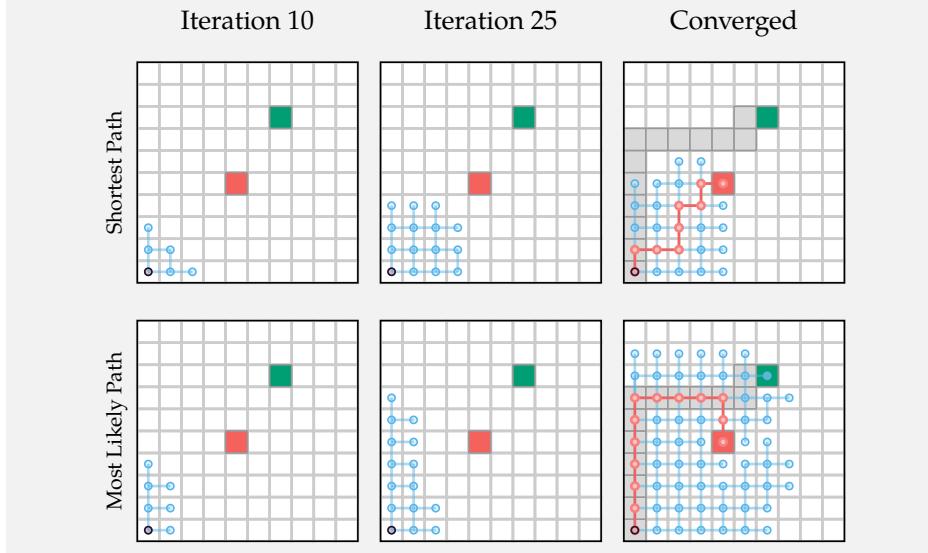
Figure 5.12. The nominal path for the continuum world problem compared to the shortest path to failure and the most likely failure path found by RRT. The most likely failure path stays closer to the nominal path before moving toward the obstacle.

¹¹ When these conditions are met, the algorithm is the same as the A* search algorithm. P. E. Hart, N. J. Nilsson, and B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

¹² For a survey, see C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A Survey of Monte Carlo Tree Search Methods,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

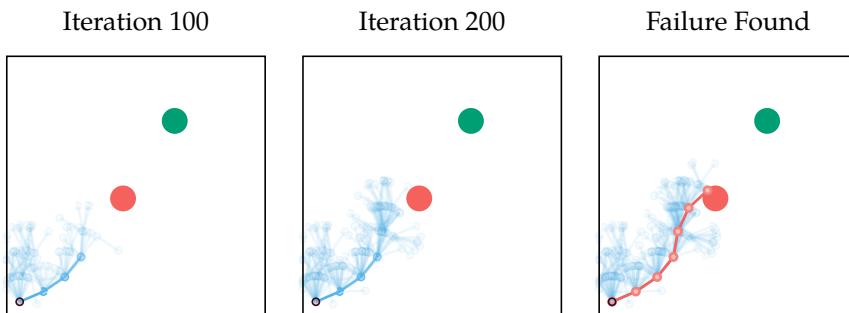
Since the state space and disturbance space for the grid world problem are discrete, we are guaranteed to find the shortest path to failure and the most likely failure path as long as we select an admissible heuristic function. For the shortest path to failure, an admissible heuristic is the Euclidean distance between the current state and the goal state. This distance will always be less than or equal to the actual cost of reaching the goal state since the shortest path between two points is a straight line. For the most likely failure path, we can use the likelihood of a straight line trajectory from the current state to the goal state assuming that it used the most likely disturbance at each step. The plots show the results. As in the continuum world problem (figure 5.12), the most likely failure path stays closer to the nominal path (highlighted in gray) before moving toward the obstacle.

Example 5.4. Example of using RRT to find the shortest path to failure and most likely failure for the grid world problem. The plots show the search tree at different iterations of the algorithm. The most likely failure path stays closer to the nominal path (highlighted in gray) before moving toward the obstacle.



promising by maintaining a value estimate $Q(s, x)$ for each node in the tree. Given a failure objective (section 4.5), $Q(s, x)$ represents the expected future objective value when applying disturbance x from state s . MCTS searches for the path with the lowest objective value.¹³

In the select step, MCTS traverses the tree starting at the root node. At each node, we determine whether to select it for the extend step based on its current number of children and number of visits $N(s)$. Specifically, we extend the node if the number of children is less than or equal to $kN(s)^\alpha$, where k and α are algorithm hyperparameters. This process is referred to as *progressive widening*. If the number of children exceeds this value, we continue to traverse the tree using a heuristic that balances between exploration and exploitation.



A common heuristic is the *lower confidence bound* (*LCB*) (algorithm 5.11), which is defined as

$$Q(s, x) - c \sqrt{\frac{\log N(s)}{N(s, x)}} \quad (5.8)$$

where $N(s, x)$ is the number of times we took the path corresponding to disturbance x from the node corresponding to state s . The first term in equation (5.8) exploits our current estimate of how promising a particular path is based on the value function, and the second term is an exploration bonus. The exploration constant c controls the amount of exploration. Higher values will lead to more exploration. We move to the child node with the lowest LCB value and repeat the process until we reach a node that we can extend.

¹³ When the objective function is the most likely failure objective, this technique is sometimes referred to as *adaptive stress testing*. R. Lee, O. J. Mengshoel, A. Sakseña, R. W. Gardner, D. Genin, J. Silbermann, M. Owen, and M. J. Kochenderfer, “Adaptive Stress Testing: Finding Likely Failure Events with Reinforcement Learning,” *Journal of Artificial Intelligence Research*, vol. 69, pp. 1165–1201, 2020.

Figure 5.13. MCTS applied to find a failure in the continuum world problem. Darker nodes and edges were visited more often. MCTS finds a failure (highlighted in red) after 258 iterations. We estimate the value of each node by performing 10 rollouts of depth 10 from the node and computing the robustness.

```

struct MCTS <: TreeSearch
    estimate_value      # v = estimate_value(sys, ψ, node)
    c                  # exploration constant
    k                  # progressive widening constant
    α                  # progressive widening exponent
    select_disturbance # x = select_disturbance(sys, node)
    k_max              # number of iterations
end

mutable struct MCTSNode
    state      # node state
    parent     # parent node
    edge       # (o, a, x)
    children   # vector of child nodes
    N          # visit count
    Q          # value estimate
end

function initialize_tree(alg::MCTS, sys)
    return [MCTSNode(rand(Ps(sys.env))), nothing, nothing, [], 1, 0]
end

function select(alg::MCTS, sys, ψ, tree)
    c, k, α, node = alg.c, alg.k, alg.α, tree[1]
    while length(node.children) > k * node.N^α
        node = lcb(node, c)
    end
    return node
end

function extend!(alg::MCTS, sys, ψ, tree, node)
    x = alg.select_disturbance(sys, node)
    o, a, s' = step(sys, node.state, x)
    Q = alg.estimate_value(sys, ψ, s')
    snew = MCTSNode(s', node, (; o, a, x), [], 1, Q)
    push!(node.children, snew)
    push!(tree, snew)
    while !isnothing(node)
        node.N += 1
        node.Q += (Q - node.Q) / node.N
        Q, node = node.Q, node.parent
    end
end

```

Algorithm 5.10. The Monte Carlo tree search algorithm. The algorithm is a type of tree search algorithm and implements both the `select` and `extend!` functions. The `select` function traverses the tree using the `lcb` function as a guide until it reaches a node that can be extended based on its number of children. The `extend!` function samples a disturbance according to the `select_disturbance` function and simulates the system one step forward in time from the current node. It then estimates the value at the new node using the `estimate_value` function and adds it to the tree. Finally, it propagates this information back up the tree to update the visit counts and mean value estimate for each node in the path.

```

function lcb(node::MCTSNode, c)
    Qs = [node.Q for node in node.children]
    Ns = [node.N for node in node.children]
    lcbs = [Q - c*sqrt(log(node.N)/N) for (Q, N) in zip(Qs, Ns)]
    return node.children[argmin(lcbs)]
end

```

Algorithm 5.11. The lower confidence bound algorithm. The algorithm computes the LCB for each child node according to equation (5.8) and returns the child node with the lowest LCB.

In the extend step, MCTS samples a disturbance and simulates the system one step forward in time from the current node. It then estimates the value at the new node and adds it to the tree. A common technique to estimate this value is to perform rollouts from the new node and evaluate their robustness. We can also estimate the value using a heuristic such as distance to failure. Finally, we propagate this information back up the tree to update the visit counts and mean value estimate for each node in the path. Figure 5.13 shows the result of using MCTS to find failures in the continuum world problem. The algorithm gradually expands the tree toward the obstacle and visits promising nodes more often.

The tree search algorithms we have presented so far assumed deterministic transitions between nodes. In other words, simulating disturbance x from state s will always lead to the same next state s' . However, we may not have control over all sources of randomness for some real-world simulators, resulting in stochastic transitions between nodes. One advantage of MCTS is that it can handle this stochasticity. A technique called *double progressive widening* can be used to extend the tree in these cases. Double progressive widening applies the progressive widening condition to both the disturbance and next state.¹⁴

5.5 Reinforcement Learning

Reinforcement learning algorithms train agents to perform a task while they interact with an environment.¹⁵ We can use reinforcement learning for falsification by training an agent to cause a system to fail. To avoid confusing the reinforcement learning agent with the agent in the system under test, we call the reinforcement learning agent an *adversary*.

Figure 5.14 shows the overall setup. At each time step, the adversary interacts with the system by selecting a disturbance x . The system then steps forward in time and produces a reward r for the adversary related to the failure objective. We refer to a series of these time steps as an *episode*. Reinforcement learning

¹⁴ A. Couëtoux, J.-B. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard, “Continuous Upper Confidence Trees,” in *Learning and Intelligent Optimization (LION)*, 2011.

¹⁵ For an introduction to reinforcement learning, see R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, Second Edition. MIT Press, 2018.

algorithms train the adversary to maximize reward using data gathered over a series of episodes. Specifically, the adversary learns a policy $\pi_{\text{adv}}(s)$ that maps states to disturbances. Once the adversary is trained, we can use it to search for failures by performing rollouts of the system using disturbances selected by the adversary's policy.

Similar to MCTS, reinforcement learning algorithms balance between exploration and exploitation. The adversary explores by trying different disturbances in each state, and exploits by selecting disturbances that are likely to lead to a failure. Typically, the adversary will explore more at the beginning of training to gather data that it can later on exploit. Reinforcement learning algorithms balance between these two objectives to maximize *sample efficiency*. Sample efficient algorithms require as few episodes as possible to learn an effective policy. A number of sample efficient reinforcement learning algorithms have been developed, and we can use off-the-shelf implementations of them to efficiently find failures of complex systems.¹⁶

Another advantage of a reinforcement learning approach is its ability to generalize. The shooting methods and tree search algorithms discussed in this chapter all required a specific initial state from which to find a failure path. Using reinforcement learning to find failures removes this necessity. Because the adversary learns a policy over the entire state space, we can perform a rollout from any initial state to search for a failure. Example 5.5 demonstrates this result on the continuum world problem using an off-the-shelf reinforcement learning package.

5.6 Simulator Requirements

Selecting an appropriate falsification algorithm for a given system is often dependent on the capabilities of the system simulator. Some commercial simulators, for example, do not provide access to their internal models. Simulators also differ in the aspects of the simulation that the user can control. The falsification algorithms we discussed in this chapter and the previous chapter impose different requirements on the system simulator. Figure 5.15 summarizes these requirements.

To apply any of the algorithms from chapter 4, the simulator must be capable of performing a rollout. For a black-box rollout, the simulator takes as input an initial state s and a vector of disturbances x from the user and outputs the objective value $f(\tau)$. For these simulators, we can perform falsification using direct sampling or fuzzing. We can also use optimization-based falsification algorithms that only rely

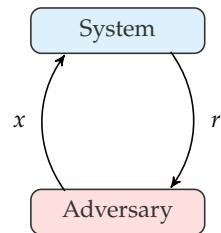


Figure 5.14. Reinforcement learning for falsification. We train an adversary to select disturbances that will cause a system to fail. The adversary receives feedback in the form of a reward signal.

¹⁶ Off-the-shelf reinforcement learning packages provide implementations of a variety of reinforcement learning algorithms. For example, see the Crux.jl package in the Julia ecosystem.

To apply the reinforcement learning algorithms implemented in the `Crux.jl` package to the continuum world problem, we need to define the following:

```
initial_state_dist = Product([Distributions.Uniform(0, 10),
                             Distributions.Uniform(0, 10)])
function interact(s, x, rng)
    -, -, s' = step(cw, s, Disturbance(0, x, 0))
    r = Float32(robustness(s, ψ.formula) - robustness(s', ψ.formula))
    norm(s' - [4.5, 4.5]) < 0.5 ? r += 10.0 : nothing
    return (sp=s', r=r)
end
```

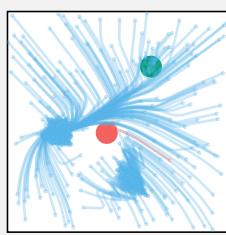
We first define an initial state distribution that covers the entire state space, allowing us to find failures starting from any state. The `interact` function defines how the adversary interacts with the system. Given a state `s` and a disturbance `x`, the function simulates the system one step forward in time and returns a tuple with the next state `s'` and reward `r`. The random number generator `rng` is a required input for `Crux.jl` but is not used in this case since the function is deterministic.

The reward is based on the change in robustness for the current step. We also add a large reward for reaching a failure state. With these definitions, we can apply any of the reinforcement learning algorithms in the `Crux.jl` package to find failures. The plots show rollouts of the adversary policy starting from different initial states after different numbers of training episodes using an algorithm called Proximal Policy Optimization (PPO). The adversary is able to find failures from most initial states after 50,000 training episodes.

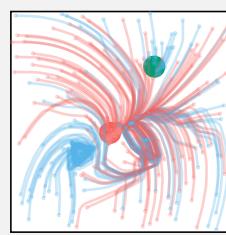
5,000 Episodes



30,000 Episodes



50,000 Episodes



Example 5.5. Example of using reinforcement learning to find failures in the continuum world problem. The plots show rollouts of the adversary policy starting from different initial states after different numbers of training episodes. Failure trajectories are highlighted in red. The adversary is able to find failures from most initial states after 50,000 training episodes. For more information on the solving code, see the `Crux.jl` documentation.

Algorithm Categories	Simulator Requirements
direct sampling fuzzing direct methods population methods	Black-Box Rollout
first-order methods second-order methods	White-Box Rollout
reinforcement learning	Episode
tree search multiple shooting	Extend

Figure 5.15. Overview of simulator requirements for the various categories of falsification algorithms. The first two rows are related to the optimization-based falsification algorithms discussed in chapter 4, and the second two rows relate to the planning algorithms discussed in this chapter. Variables shown in blue are variables that the user of the simulator has control over. We cannot observe any aspects of the simulator shown in gray.

on evaluations of the objective function such as direct methods and population methods. A white-box rollout has the same inputs and outputs as a black-box rollout, but it also allows us to observe the internal model of this system. We can compute gradients and Hessians of the objective function for white-box rollouts, allowing us to apply first- and second-order optimization methods.

The planning algorithms discussed in this chapter require the simulator to be able to perform single steps. Reinforcement learning algorithms operate using episodes, which consist of a series of steps starting from a user-specified initial state. At each step, the reinforcement learning agent observes the next state and reward from the previous step and selects a disturbance. Tree search algorithms and multiple shooting methods require the simulator to be able to take steps from arbitrary states. Given a state s and a disturbance x , the simulator must be able to simulate the system one step forward in time and return the next state s' . For tree search algorithms that use cost functions, the simulator must also return the cost c of taking the step.

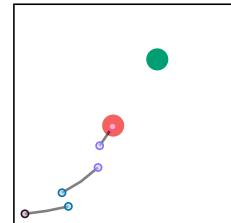
5.7 Summary

- Planning algorithms account for the temporal aspect of the falsification problem and break it into a series of smaller problems.
- Shooting methods perform optimization-based falsification by optimizing over a series of trajectory segments, which may increase efficiency for systems where small changes in the disturbances at the beginning of a trajectory can have a significant effect later.
- Tree search algorithms search the space of possible trajectories as a tree and iteratively grow the tree in search of a failure trajectory.
- Heuristic search algorithms use heuristics such as distance to failure, coverage, and robustness to guide the search.
- Monte Carlo tree search balances between exploration and exploitation to efficiently search the space of possible trajectories.
- Reinforcement learning algorithms can be used to train an adversary to produce failures in a sample-efficient manner.
- The capabilities of a system's simulator determine which falsification algorithms can be applied.

5.8 Exercises

Exercise 5.1. Suppose we apply multiple shooting to find a failure trajectory for the continuum world problem using the optimization problem in equation (5.2), and we find that the optimizer returns the trajectory shown in the margin. The colors represent which trajectory segments should connect. How can we improve this result?

Solution: While the trajectory ends at the obstacle, the trajectory is not a feasible trajectory because the segments do not connect. We can improve this result by increasing the weighting parameter λ to increase the penalty on the defect between the segments.



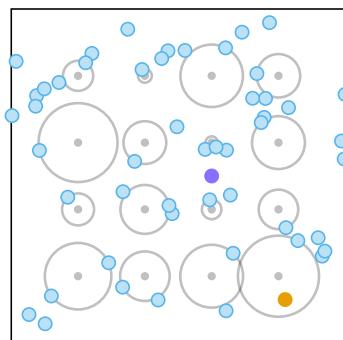
Exercise 5.2. Suppose we want to use the RRT algorithm to find failure trajectories for the inverted pendulum problem with the specification $\psi = \square(|\theta| \leq \pi/4)$. We start by exploring the space of possible trajectories by sampling goal states uniformly throughout the entire state space; however, we notice that the RRT algorithm is not finding any failures. What should we do to encourage the RRT algorithm to find failures?

Solution: We can encourage the RRT algorithm to find failures by identifying a failure region in the state space and sampling goal states uniformly within this region. In this case, the failure region is the set of states where $|\theta| > \pi/4$.

Exercise 5.3. Provide a potential drawback of sampling goal states for RRT only from the failure region. What changes could we make to the algorithm to address this drawback?

Solution: Sampling goal states only from the failure region may result in insufficient exploration of the space of possible trajectories, which may cause the algorithm to miss potential failure modes. There are multiple ways to address this drawback. One option is to sample goal states from a distribution that is biased towards the failure region but that still assigns nonzero probability density to states outside the failure region.

Exercise 5.4. The plot below shows a set of points (blue) for which we have computed the average dispersion on the grid of gray points. How would the average dispersion change if we added the purple point to the set of blue points? How would the average dispersion change if we added the orange point to the set of blue points?

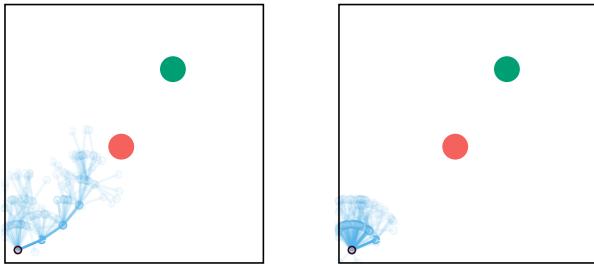


Solution: Adding the purple point would have no effect on the average dispersion because it does not change the minimum distance between points in the set to any of the gray points. Adding the orange point would decrease the distance to the nearest point in the set for the bottom right grid point. This change would decrease the average dispersion.

Exercise 5.5. Suppose we run the RRT algorithm multiple times to find failures for a system, and we find that the algorithm keeps finding the same failure mode. How could we modify the algorithm to encourage the discovery of other failure modes?

Solution: There are multiple options. One option is to modify the select and extend steps to incorporate coverage metrics. These metrics will encourage the algorithm to explore regions of the state space that have not been visited. Another option is to create a cost function that penalizes the algorithm for exploring paths that are similar to the paths of the previously found failure modes.

Exercise 5.6. The plots below show two different runs of MCTS on the continuum world problem for 150 steps. All algorithms parameters are the same except for the value of α used in progressive widening. Based on the plots below, which run used a higher value for α ?



Solution: Increasing α will increase the number of children required to traverse the tree instead of selecting a particular node to extend. Therefore, a higher value of α will result in more children per node, indicating that the MCTS run shown in the plot on the right used a higher value for α .

Exercise 5.7. Suppose we are currently deciding which node to move to next in the select step of MCTS. The current node has been visited 10 times and has two children. The first child has been visited 7 times and has an estimated value of 3. The second child has been visited 3 times and has an estimated value of 5. If our exploration constant is set to $c = 1$, which child should we move to next? Does this result prioritize exploration or exploitation? Which child should we move to if $c = 10$? Does this result prioritize exploration or exploitation?

Solution: When $c = 1$, the lower confidence bound for the first child is

$$3 - (1) \sqrt{\frac{\log 10}{7}} \approx 2.43$$

and the lower confidence bound for the second child is

$$5 - (1) \sqrt{\frac{\log 10}{3}} \approx 4.12$$

Therefore, we should move to the first child. While the first child has been visited more often, it has a lower estimated value, so this result prioritizes exploitation. When $c = 10$, the lower confidence bound for the first child is -2.74 and the lower confidence bound for the second child is -3.76 . Therefore, we should move to the second child, prioritizing exploration.

Exercise 5.8. Suppose we have a simulator of an aircraft collision avoidance system that we want to use to find failures of the system. Each rollout in the simulator is an encounter between two aircraft. For each rollout, we can specify the initial state of the aircraft. As the rollout progresses, we can observe the state of the aircraft and select the sensor noise we would like to apply at the next time step. However, we do not have access to the mathematical equations that the simulator uses to model transitions to the next state, and we cannot initialize the simulator into any arbitrary state. What types of falsification algorithms could we use to find failures of the system?

Solution: We could use any of the algorithms from the first or third row of figure 5.15. Examples of these algorithms include direct sampling, fuzzing, direct methods, population methods, and reinforcement learning.

6 Failure Distribution

While the falsification algorithms in the previous chapters search for single failure events, it is often desirable to understand the distribution over failures for a given system and specification. This distribution is difficult to quantify exactly for many real-world systems. Instead, we can approximate the failure distribution by drawing samples from it. This chapter discusses methods for sampling from the failure distribution. We present two categories of sampling methods. First, we discuss rejection sampling, which produces samples from a target distribution by accepting or rejecting samples from a different distribution. We then present Markov chain Monte Carlo (MCMC) methods. MCMC methods generate samples from a target distribution using a chain of correlated samples. We conclude with a discussion of probabilistic programming, which allows us to scale MCMC methods to complex, high-dimensional systems.

6.1 Distribution over Failures

The distribution over failures for a given system with specification ψ is represented by the conditional probability $p(\tau | \tau \notin \psi)$. We can write this probability as

$$p(\tau | \tau \notin \psi) = \frac{\mathbb{1}\{\tau \notin \psi\} p(\tau)}{\int \mathbb{1}\{\tau \notin \psi\} p(\tau) d\tau} \quad (6.1)$$

where $\mathbb{1}\{\cdot\}$ is the indicator function and $p(\tau)$ is the probability density of the nominal trajectory distribution for trajectory τ . Figure 6.1 shows the failure distribution for a simple system where trajectories consist of only a single state that is sampled from a normal distribution. For most systems, the failure distribution is difficult to compute exactly because doing so requires solving the integral in the denominator of equation (6.1) to compute the normalizing constant. The value of

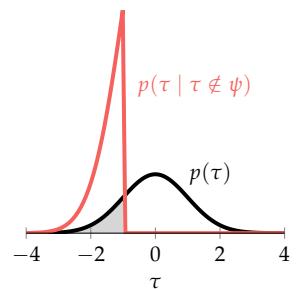


Figure 6.1. The distribution over failures for a simple system where trajectories consist of only a single state that is sampled from a Gaussian distribution (black). A failure occurs when the sampled state is less than -1 . The area of the shaded region corresponds to the integral in equation (6.1). The failure distribution (red) is the probability density function of the nominal distribution in the failure region scaled by this value.

this integral corresponds to the probability of failure for the system. We discuss methods to estimate this quantity in chapter 7.

While we cannot compute the probability density of the failure distribution exactly, we can use its unnormalized probability density $\bar{p}(\tau \mid \tau \notin \psi)$ to draw samples from it. The unnormalized probability density is given by

$$\bar{p}(\tau \mid \tau \notin \psi) = \mathbb{1}\{\tau \notin \psi\} p(\tau) \quad (6.2)$$

Computing this density for a given trajectory only requires determining whether it is a failure trajectory and evaluating its probability density under the nominal trajectory distribution. The rest of this chapter discusses several methods for sampling from this unnormalized distribution.¹ With enough samples, we can implicitly represent the distribution over failures (see figure 6.2).

6.2 Rejection Sampling

Rejection sampling produces samples from a complex target distribution by accepting or rejecting samples from a different distribution that is easier to sample from. It is inspired by the idea of throwing darts uniformly at a rectangular dart board that encloses the graph of the density of the target distribution. If we keep only the darts that land inside the target density, we produce samples that are distributed according to the target distribution (see figure 6.3).

In the dart board example, we are using samples from a uniform distribution to produce samples from an arbitrary target density. The efficiency of this process depends on the area of the dart board that lies outside the target distribution. If there is a large area outside the target distribution, many of the darts will be rejected, and we will require more darts to accurately represent the target distribution. One way to improve efficiency is to use a different dart board that more closely matches the shape of the target distribution. In other words, we may want to draw samples from a different distribution that is still easy to sample from but more closely matches the target distribution. We call this distribution a *proposal distribution*.

Algorithm 6.1 implements the rejection sampling algorithm given a target distribution with density function $\bar{p}(\tau)$ and a proposal distribution with density function $q(\tau)$. At each iteration, we draw a sample τ from the proposal distribution and accept it with probability proportional to $\bar{p}(\tau)/(cq(\tau))$.² To ensure that the proposal distribution fully encloses the target distribution, we require that

¹ For a detailed overview, see C.P. Robert and G. Casella, *Monte Carlo Statistical Methods*. Springer, 1999, vol. 2.

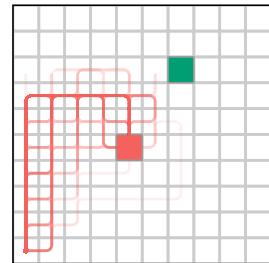


Figure 6.2. Distribution over failures for the grid world problem represented implicitly through samples. The probability of slipping is set to 0.8.

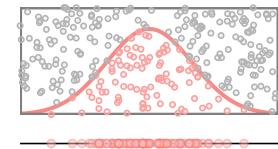


Figure 6.3. Sampling from a truncated normal distribution by throwing darts uniformly at a rectangular dart board that encloses the graph of its density function. The samples on the bottom are obtained by moving all of the darts that land inside the target distribution to the bottom of the dart board. These samples are distributed according to the target distribution.

² In the dart board analogy, we can think of this acceptance criteria as a two step process. First, we sample the x -coordinate of the dart from the proposal distribution. Second, we select its y -coordinate randomly between the bottom of the board and $cq(\tau)$. If it falls under $p(\tau)$, it is accepted.

$q(\tau) > 0$ whenever $\bar{p}(\tau) > 0$ and that c is selected such that $\bar{p}(\tau) \leq cq(\tau)$ for all τ . The density function of the target distribution does not need to be normalized.

```

struct RejectionSampling
    p      # target density
    q      # proposal trajectory distribution
    c      # constant such that  $\bar{p}(\tau) \leq cq(\tau)$ 
    k_max # max iterations
end

function sample_failures(alg::RejectionSampling, sys,  $\psi$ )
    p, q, c, k_max = alg.p, alg.q, alg.c, alg.k_max
    ts = []
    for k in 1:k_max
         $\tau$  = rollout(sys, q)
        if rand() < p( $\tau$ ) / (c * pdf(q,  $\tau$ ))
            push!(ts,  $\tau$ )
        end
    end
    return ts
end

```

To sample from the failure distribution, we use the unnormalized density in equation (6.2) as the target density. A common choice for the proposal distribution is the nominal trajectory distribution. To use this proposal, we must select a value for c such that $\mathbb{1}\{\tau \notin \psi\}p(\tau) \leq cp(\tau)$. Selecting $c = 1$ satisfies this condition and causes the acceptance ratio to reduce to $\mathbb{1}\{\tau \notin \psi\}$. In other words, we will accept a sample if it is a failure trajectory and reject it otherwise. Figure 6.4 shows an example that uses the nominal trajectory distribution to sample from the failure distribution shown in figure 6.1.

If failures are unlikely under the nominal distribution, we will require many samples to produce a representative set of samples from the failure distribution. In this case, we may be able to improve efficiency by using domain knowledge to select a proposal distribution that more closely matches the shape of the failure distribution. For example, failures occur at negative values in the simple system shown in figure 6.1, so we may be able to improve efficiency by shifting the proposal distribution to the left.

When we select the proposal distribution for rejection sampling, we must also select a value for c to ensure that the proposal distribution fully encloses the target distribution for all τ . Figure 6.5 shows an example that uses a shifted proposal distribution to sample from the failure distribution shown in figure 6.1 for two

Algorithm 6.1. The rejection sampling algorithm for sampling from a target distribution. At each iteration, the algorithm performs a rollout using the proposal trajectory distribution, computes the acceptance ratio, and accepts the sample with probability equal to the acceptance ratio.

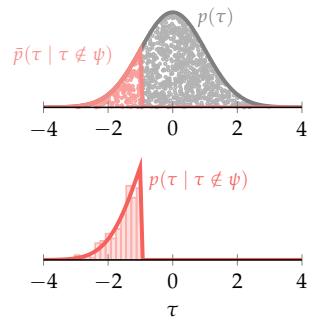


Figure 6.4. Rejection sampling using the nominal trajectory distribution as the proposal distribution to sample from the failure distribution shown in figure 6.1. The plot on the top shows the target density (red) and the proposal density (gray). Accepted samples are highlighted in red. The plot on the bottom shows a histogram of the accepted samples compared to the density function of the failure distribution.

different values of c . We want to select c to be as tight as possible to achieve the highest efficiency. In general, selecting a good proposal distribution and value for c requires domain knowledge and can be challenging for high-dimensional systems with long time horizons. If c is too loose, rejection sampling may be too inefficient to be useful (see example 6.1). The next section discusses techniques that tend to perform better in these cases.

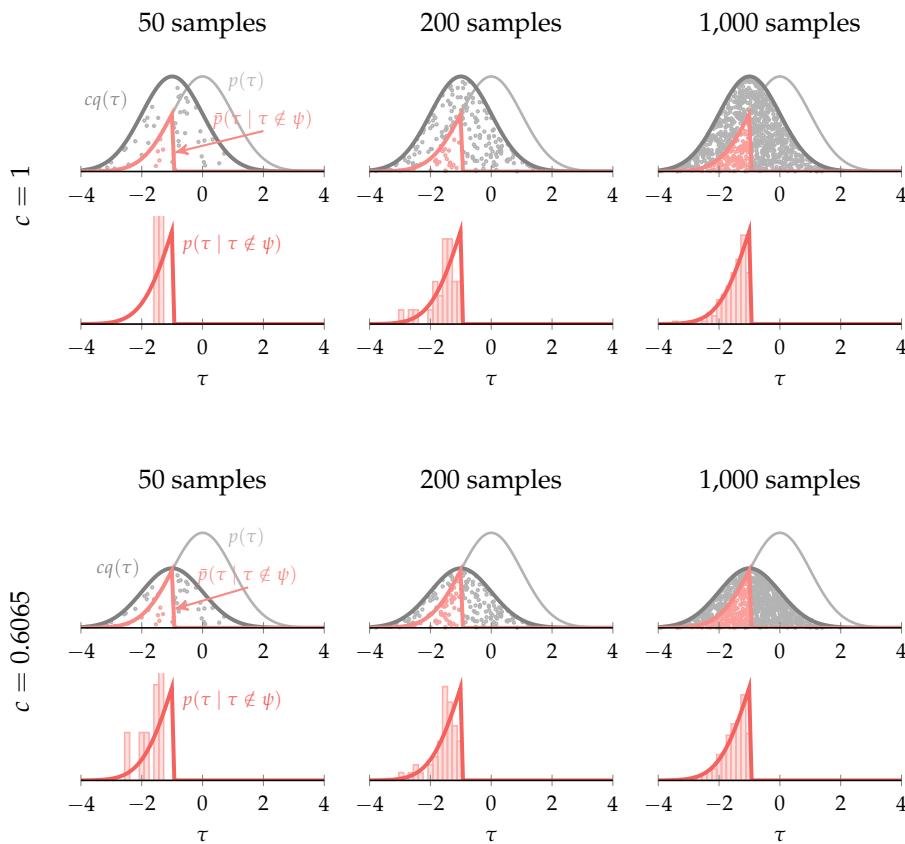


Figure 6.5. Using a hand-designed proposal distribution to apply rejection sampling to the simple system in figure 6.1 for two different values of c . The proposal distribution is a normal distribution shifted to the left ($q(\tau) = \mathcal{N}(\tau | -1, 1^2)$). The top row shows the results for $c = 1$, which is a loose bound. The bottom row shows the results for $c = 0.6065$, which is the tightest possible value for c . More samples are accepted using the tighter value for c resulting in greater efficiency.

Suppose we want to use rejection sampling to sample failures from an inverted pendulum system where the standard deviation of the sensor noise for each state variable is 0.1. From example 4.3, we know that failures are rare under the nominal trajectory distribution, so rejection sampling using the nominal trajectory distribution as a proposal will be inefficient. We also saw in example 4.3 that when we instead sampled trajectories from a distribution where the standard deviation of the sensor noise was 0.15, we were able to find failures. Therefore, we may want to use this distribution as a proposal for rejection sampling.

We must then select a value for c such that

$$\begin{aligned} p(\tau) &\leq cq(\tau) \\ p(s_1) \prod_{t=1}^d \mathcal{N}(x_t | 0, (0.1)^2 I) &\leq cp(s_1) \prod_{t=1}^d \mathcal{N}(x_t | 0, (0.15)^2 I) \\ \prod_{t=1}^d \left(\frac{\mathcal{N}(x_t | 0, 0.01I)}{\mathcal{N}(x_t | 0, 0.0225I)} \right) &\leq c \end{aligned}$$

where we assume that the initial state distribution is the same for the proposal and target. The term in the product will be maximized when $x_t = [0, 0]$ for all t . Plugging this result into the product and assuming a depth of 40, we find that

$$\begin{aligned} \left(\frac{\mathcal{N}(0 | 0, 0.01I)}{\mathcal{N}(0 | 0, 0.0225I)} \right)^{40} &\leq c \\ 1.2226 \times 10^{14} &\leq c \end{aligned}$$

Therefore, the tightest value we can select for c is 1.2226×10^{14} . Using this value, our acceptance probabilities end up being very small (on the order of 10^{-15}), and rejection sampling is inefficient.

Example 6.1. Example of the challenges of using rejection sampling for high-dimensional systems with long time horizons. In this example, we compute the tightest value we can select for c based on domain knowledge for the inverted pendulum system and show that it is prohibitively large.

6.3 Markov Chain Monte Carlo

Markov chain Monte Carlo (MCMC) algorithms generate samples from a target distribution by sampling from a *Markov chain*.³ A Markov chain is a sequence of random variables where each variable depends only on the previous one. MCMC algorithms begin by initializing a Markov chain with an initial sample τ . At each iteration, they use the current sample τ to generate a new sample τ' by sampling from a conditional distribution $g(\cdot | \tau)$. This distribution is sometimes referred to as a *kernel*.⁴ We accept or reject the new sample based on an acceptance criteria. If the new sample is accepted, we set $\tau = \tau'$ and continue to the next iteration. If the new sample is rejected, we keep the previous sample.

Given certain properties of the kernel and acceptance criterion, MCMC algorithms are guaranteed to converge to the target distribution in the limit of infinite samples. However, the initial samples may not be representative of the target distribution. For this reason, we often specify a *burn-in* period in which the initial samples are discarded. Furthermore, unlike rejection sampling, the samples produced by MCMC algorithms are not independent from one another. Each sample in the chain depends on the previous one. Therefore, it is also common to thin the samples by only keeping every h th sample. Several variations of MCMC differ in how they implement the acceptance criteria and the kernel.

6.3.1 Metropolis-Hastings

One of the most common MCMC algorithms is the *Metropolis-Hastings* algorithm.⁵ The Metropolis-Hastings algorithm accepts a new sample τ' given the current sample τ with probability

$$\frac{\bar{p}(\tau')g(\tau | \tau')}{\bar{p}(\tau)g(\tau' | \tau)} \quad (6.3)$$

where \bar{p} is the unnormalized target density. To sample from the failure distribution, we set $\bar{p}(\tau) = \mathbb{1}\{\tau \notin \psi\}p(\tau)$. Since we are taking a ratio of the densities, the target density does not need to be normalized. The kernel $g(\cdot | \tau)$ is often chosen to be a symmetric distribution, meaning that $g(\tau' | \tau) = g(\tau | \tau')$.⁶ In this case, the acceptance criteria reduces to $\bar{p}(\tau')/\bar{p}(\tau)$. Intuitively, if τ' is more likely than τ , it is always accepted. If τ' is less likely than τ , it is accepted with probability proportional to the ratio of the densities.

³ A detailed overview of MCMC techniques is provided in C.P. Robert and G. Casella, *Monte Carlo Statistical Methods*. Springer, 1999, vol. 2.

⁴ This distribution is also sometimes referred to as a proposal distribution. It differs from the proposal distribution used in rejection sampling in that it is conditioned on the previous sample.

⁵ W.K. Hastings, "Monte Carlo Sampling Methods Using Markov Chains and Their Applications," *Biometrika*, vol. 57, no. 1, pp. 97–97, 1970.

⁶ When the kernel is symmetric, the algorithm is called the Metropolis algorithm: N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," *Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953. A common choice of a symmetric kernel is a Gaussian distribution centered at the previous sample.

Algorithm 6.2 implements the Metropolis-Hastings algorithm given a target density, a kernel, and an initial trajectory to begin the Markov chain. The kernel is a conditional distribution that takes in a trajectory and produces a trajectory distribution. Example 6.2 shows an example of a kernel for the inverted pendulum system. The next sample is generated by performing a rollout using this distribution. We then accept or reject the new sample based on the acceptance ratio in equation (6.3). Figure 6.6 shows the result of using the Metropolis-Hastings algorithm to sample from the failure distribution shown in figure 6.1.

```

struct MCMCSampling
     $\bar{p}$       # target density
    g         # kernel:  $\tau' = \text{rollout}(\text{sys}, g(\tau))$ 
     $\tau$        # initial trajectory
    k_max    # max iterations
    m_burnin # number of samples to discard from burn-in
    m_skip    # number of samples to skip for thinning
end

function sample_failures( $\text{alg}:\text{MCMCSampling}$ , sys,  $\psi$ )
     $\bar{p}$ , g,  $\tau$  =  $\text{alg}.\bar{p}$ ,  $\text{alg}.g$ ,  $\text{alg}.\tau$ 
    k_max, m_burnin, m_skip =  $\text{alg}.k\_max$ ,  $\text{alg}.m\_burnin$ ,  $\text{alg}.m\_skip$ 
    ts = []
    for k in 1:k_max
         $\tau' = \text{rollout}(\text{sys}, g(\tau))$ 
        if rand() < ( $\bar{p}(\tau') * \text{pdf}(g(\tau'), \tau)$ ) / ( $\bar{p}(\tau) * \text{pdf}(g(\tau), \tau')$ )
             $\tau = \tau'$ 
        end
        push!(ts,  $\tau$ )
    end
    return ts[m_burnin:m_skip:end]
end

```

Algorithm 6.2. The Metropolis-Hastings algorithm for sampling from a target distribution. The kernel function g must take in a trajectory and return a trajectory distribution. At each iteration, the algorithm generates a new sample by performing a rollout using this distribution. It then accepts or rejects the new sample based on the acceptance ratio in equation (6.3). The algorithm discards the first m_{burnin} samples and thins the remaining samples according to m_{skip} .

6.3.2 Smoothing

When we use algorithm 6.2 to sample from the failure distribution, we will not accept any samples that are not failures because $\bar{p}(\tau) = \mathbb{1}\{\tau \notin \psi\} p(\tau)$ will be 0 for those samples. While this behavior is necessary for the algorithm to converge to the failure distribution in the limit of infinite samples, it can create challenges in practice. For example, if we initialize the Markov chain to a safe trajectory, the algorithm will reject all samples from $g(\cdot | \tau)$ until it samples a failure. Since $g(\cdot | \tau)$ typically produces trajectories similar to τ , we may require many samples

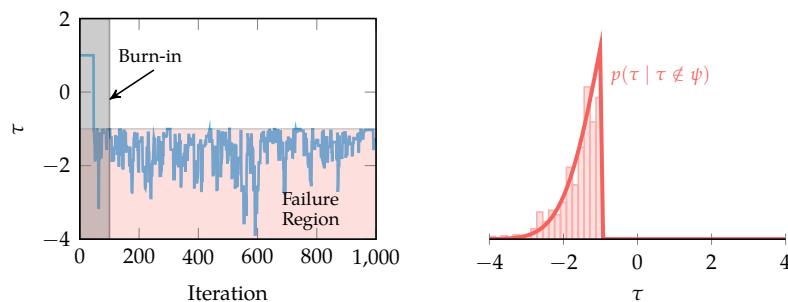
To define a Gaussian kernel for the inverted pendulum system, we must first define a trajectory distribution type (algorithm 4.3) for the pendulum. The following code defines a trajectory distribution for the pendulum system that uses a Gaussian distribution for the initial state and a vector of Gaussian distributions for the observation disturbance distributions:

```
struct PendulumTrajectoryDistribution <: TrajectoryDistribution
    μ₁ # mean of initial state distribution
    Σ₁ # covariance of initial state distribution
    μₛ # vector of means of length d
    Σₛ # vector of covariances of length d
end
function initial_state_distribution(p::PendulumTrajectoryDistribution)
    return MvNormal(p.μ₁, p.Σ₁)
end
function disturbance_distribution(p::PendulumTrajectoryDistribution, t)
    D = DisturbanceDistribution((o)→Deterministic(),
                                (s,a)→Deterministic(),
                                (s)→MvNormal(p.μₛ[t], p.Σₛ[t]))
    return D
end
depth(p::PendulumTrajectoryDistribution) = length(p.μₛ)
```

We can then define a kernel for the pendulum system that returns an instantiation of this distribution as follows:

```
function inverted_pendulum_kernel(τ; Σ=0.01I)
    μ₁ = τ[1].s
    μₛ = [step.x₀ for step in τ]
    return PendulumTrajectoryDistribution(μ₁, Σ, μₛ, [Σ for step in τ])
end
```

The new distribution is centered at the initial state and observation disturbances of the current sample. We can use this kernel with algorithm 6.2 to sample from the failure distribution of the inverted pendulum system.



Example 6.2. Example of a Gaussian kernel for the inverted pendulum system.

before we sample a failure to accept, especially if τ is far from the failure region.⁷ We see this behavior during the burn-in period in figure 6.6.

Another challenge arises when the failure distribution has multiple modes. To move between modes, the algorithm must sample a failure from one failure mode using a kernel conditioned on a trajectory from another. If the failure modes are spread out in the trajectory space, the algorithm may require a large number of samples before moving from one mode to another. Example 6.3 illustrates these challenges on a simple Gaussian system.

Smoothing is a technique that addresses these challenges by modifying the target density to make it easier to sample from.⁸ It relies on a notion of the distance to failure, which we will write as $\Delta(\tau)$ for a given trajectory τ . This distance is a nonnegative number that measures how close τ is to a failure. For failure trajectories, $\Delta(\tau)$ should be 0. We can rewrite the target density in terms of this distance as

$$\bar{p}(\tau \mid \tau \notin \psi) = \mathbb{1}\{\Delta(\tau) = 0\} p(\tau) \quad (6.4)$$

The indicator function causes sharp boundaries between safe and unsafe trajectories. To create a smooth version of this density, we replace the indicator function with a Gaussian distribution with mean 0 and a small standard deviation. The resulting smoothed density is

$$\bar{p}(\tau \mid \tau \notin \psi) \approx \mathcal{N}(\Delta(\tau) \mid 0, \epsilon^2) p(\tau) \quad (6.5)$$

where ϵ is the standard deviation.

For systems with temporal logic specifications, we can specify the distance function using temporal logic robustness (section 3.5.2). Since robustness is positive when the formula is satisfied and negative when it is violated, we can write the distance function as

$$\Delta(\tau) = \max(0, \rho(\tau)) \quad (6.6)$$

where $\rho(\tau)$ is the robustness of the trajectory τ . Figure 6.7 shows the smoothed version of the failure distribution in figure 6.1 for different values of ϵ . As ϵ approaches 0, the smoothed density approaches the shape of the failure distribution. As ϵ approaches infinity, the smoothed density approaches the shape of the nominal distribution.

⁷ One way to avoid this behavior is to ensure that the initial trajectory is a failure. The algorithms in chapters 4 and 5 can be used to search for an initial failure trajectory.

⁸ H. Delecki, A. Corso, and M.J. Kochenderfer, “Model-Based Validation as Probabilistic Inference,” in *Conference on Learning for Dynamics and Control (L4DC)*, 2023.

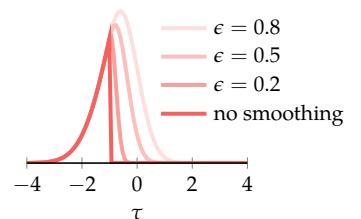


Figure 6.7. Smoothed versions of the failure distribution in figure 6.1 for different values of ϵ . As ϵ decreases, the smoothed distribution approaches the failure distribution.

Suppose we want to sample from the failure distribution shown in the plot on the left and we initialize our Markov chain with $\tau = 1$. We will not accept a new sample until we draw a sample with a value less than -1 . If we use a Gaussian kernel with standard deviation 1, we have that

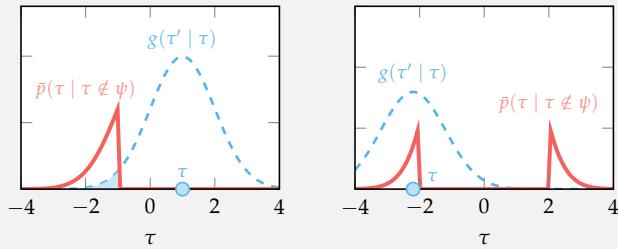
$$g(\tau' | \tau) = \mathcal{N}(\tau' | 1, 1^2)$$

The probability of drawing a sample less than -1 from this distribution is 0.02275 (corresponding to the shaded region in the plot on the left). Therefore, we will require 44 samples on average before the algorithm accepts a sample. If we were to initialize the algorithm with a sample even further from the failure region, we would require even more samples to the point where MCMC may not converge within a finite sample budget.

The plot on the right demonstrates the challenge of using MCMC to sample from a failure distribution with multiple modes. In this case, the current sample is in the mode on the left at -2.2 . Using the same Gaussian kernel, we have

$$g(\tau' | \tau) = \mathcal{N}(\tau' | -2.2, 1^2)$$

The probability of moving to the other mode from this point is 1.3346×10^{-5} . Therefore, we will require a large number of samples before we switch modes.



Example 6.3. Example of the challenges of using MCMC to sample from the failure distribution given a finite sample budget. The plot on the left demonstrates the challenges with initialization, and the plot on the right shows the challenges of sampling from failure distributions with multiple modes.

The smoothed failure distribution assigns a nonzero probability to all trajectories, and it assigns higher probabilities to trajectories that are close to failure. This design allows the MCMC algorithm to more easily move between failure modes. However, because the smoothed distribution will assign a nonzero probability to safe trajectories, the algorithm will accept some samples that are not failures. We can still recover the failure distribution by rejecting these samples after MCMC has terminated. In fact, this process is equivalent to performing rejection sampling with the smoothed density as the proposal distribution. Figure 6.8 and example 6.4 show the benefit of applying MCMC with a smoothed density to sample from a failure distributions with multiple modes.

6.3.3 Metropolis-Adjusted Langevin Algorithm

The performance of MCMC is sensitive to the choice of kernel. While a Gaussian kernel is simple to implement, it does not scale well to high-dimensional systems with complex failure distributions because it randomly explores the target density without taking into account its underlying structure. We can improve performance by selecting a kernel that takes into account this structure. For example, we can use knowledge of the gradient of the target density to guide exploration.

The *Metropolis-Adjusted Langevin Algorithm* (MALA) uses a gradient-based kernel that approximates a process known as Langevin diffusion.⁹ The kernel is defined as

$$g(\tau' | \tau) = \mathcal{N}\left(\tau' | \tau + \alpha \nabla \log \bar{p}(\tau), (\sqrt{2\alpha})^2\right) \quad (6.7)$$

where α is a hyperparameter of the algorithm.¹⁰ The MALA kernel is not symmetric in general. Intuitively, the kernel takes a step in the direction of the greatest increase in log likelihood and samples from a Gaussian distribution centered at the new location. The algorithm then accepts or rejects the new sample based on the Metropolis-Hastings acceptance ratio in equation (6.3). We can run MALA using algorithm 6.2 by implementing a kernel that follows equation (6.7).

Using the gradient to guide the sampling allows the algorithm to explore the target density more efficiently than a random walk. Furthermore, when combined with the smoothing technique in section 6.3.2, the gradient helps to guide the algorithm toward different failure modes. Figure 6.9 compares the path taken by a Gaussian kernel with the path taken by MALA on a simple target density. The

⁹ Langevin dynamics is an idea from physics that was developed to model molecular systems by physicist Paul Langevin (1872–1946). MALA is also referred to as Langevin Monte Carlo. U. Grenander and M. I. Miller, “Representations of Knowledge in Complex Systems,” *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 56, no. 4, pp. 549–581, 1994.

¹⁰ This kernel represents a discrete approximation of the Langevin diffusion process. It approaches the continuous-time Langevin diffusion process as α approaches 0.

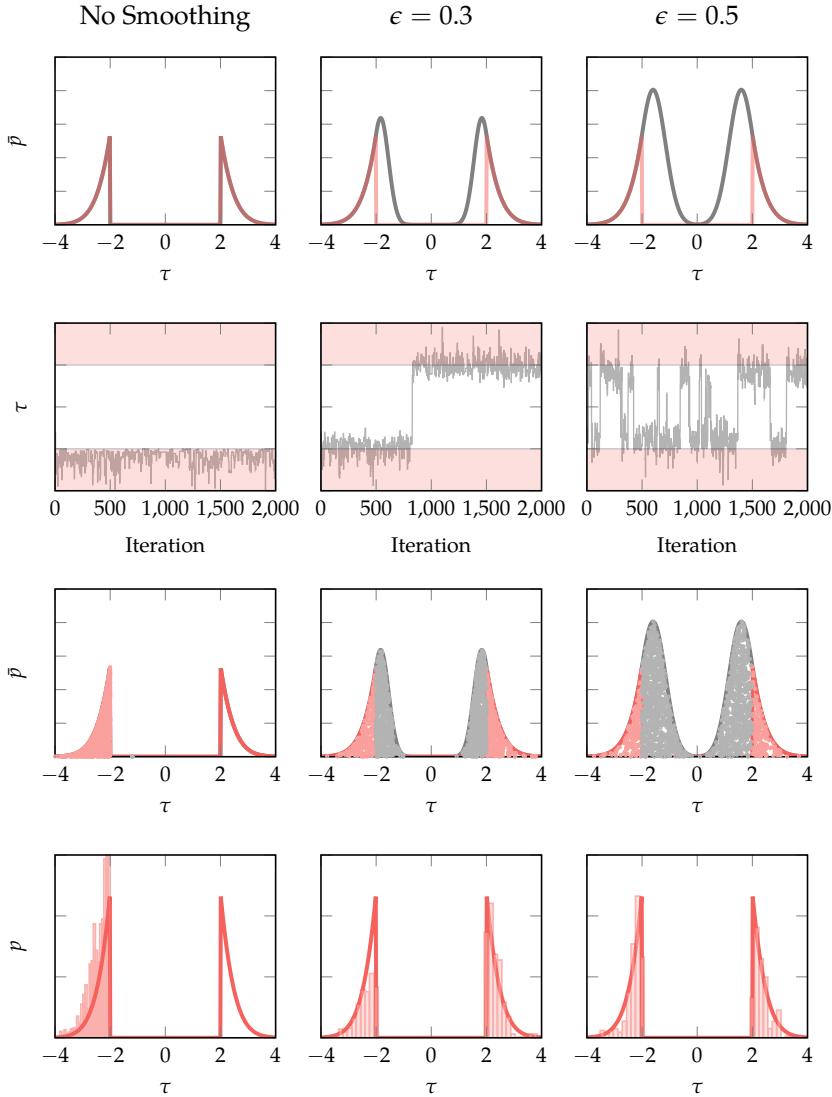
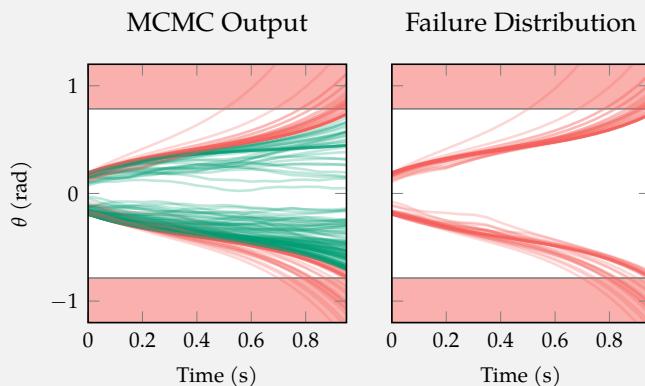


Figure 6.8. The effect of smoothing on the MCMC algorithm for a failure distribution with multiple modes. The first row shows the target density (gray) compared to the density of the true failure distribution (red). The second row shows the MCMC samples over time with the failure regions shaded in red. The third row shows the accepted (red) and rejected samples (gray). The fourth row shows a histogram of the accepted samples compared to the true probability density function of the failure distribution. The first column shows the results without smoothing. The second and third columns show the results with different values of ϵ . Without smoothing, MCMC stays in the same failure mode for all 2,000 iterations and misses the other mode. Applying smoothing allows the algorithm to more easily move between failure modes and results in better estimates of the failure distribution given the sample budget.

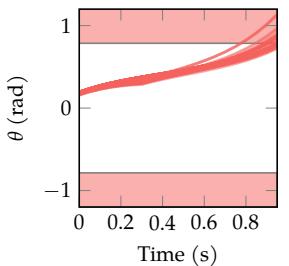
The inverted pendulum system has two main failure modes: tipping over in the negative direction and tipping over in the positive direction. To observe the effect of smoothing on the performance of MCMC, we define the following two unnormalized target densities:

```
p = NominalTrajectoryDistribution(inverted_pendulum, 21) # depth = 21
p(tau) = isfailure(psi, tau) * pdf(p, tau)
function p_smooth(tau; epsilon=0.15)
    Delta = max(robustness([step.s for step in tau], psi.formula), 0)
    return pdf(Normal(0, epsilon), Delta) * pdf(p, tau)
end
```

The plot in the margin shows the results when we run algorithm 6.2 using \bar{p} as the target density. We will not accept any samples that are not failures, and we only observe failures from one failure mode. The plots below show the results when we use \bar{p}_{smooth} combined with rejection sampling. Smoothing allows us to sample failures from both failure modes. However, we now draw some samples that are not failures during the MCMC (left), so we must reject them after the algorithm has terminated to recover the failure distribution (right).



Example 6.4. Applying smoothing to sample from the failure distribution of the inverted pendulum system. Smoothing allows MCMC to sample from both failure modes given a finite sample budget. The plot below shows the result of running MCMC without smoothing.



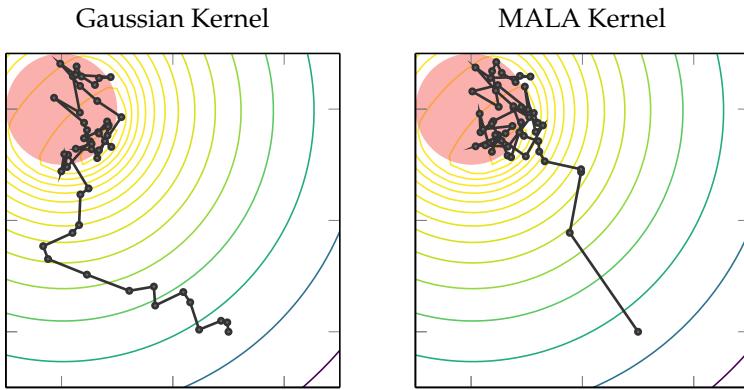


Figure 6.9. Comparison of the paths taken by algorithm 6.2 using the Gaussian kernel and the MALA kernel for a two-dimensional smoothed target density with a failure region shown in red. Brighter contours indicate higher density. The MALA kernel uses the gradient of the log likelihood to guide its steps and requires fewer samples than the Gaussian kernel to move to the failure region. The MALA kernel is also has a higher acceptance rate.

MALA kernel enables MCMC to move more efficiently toward regions of high likelihood.

6.3.4 Metropolis-Hastings Variations

MALA is one of several variations of the Metropolis-Hastings algorithm. Other gradient-based variations include Hamiltonian Monte Carlo¹¹ (HMC) and the No U-Turn Sampler¹² (NUTS). HMC uses a simulation of Hamiltonian dynamics based on the gradient of the log likelihood to guide exploration. NUTS is an extension of HMC that tends to not require as much tuning of its hyperparameters. Another variation of the Metropolis-Hastings algorithms is Gibbs sampling, which updates each variable in the target density one at a time conditioned on the values of the other variables.¹³ Gibbs sampling is particularly beneficial when sampling from high-dimensional target densities where the conditional distributions are easier to sample from than the joint distribution.

6.4 Probabilistic Programming

Probabilistic programming is a technique for specifying probabilistic models as computer programs in a way that allows inference to be performed automatically.¹⁴ By specifying the model of a given system as a probabilistic program, we can apply a variety of MCMC algorithms to sample from the failure distribution.

¹¹ Hamiltonian Monte Carlo is also referred as Hybrid Monte Carlo. S. Duane, A.D. Kennedy, B.J. Pendleton, and D. Roweth, "Hybrid Monte Carlo," *Physics Letters B*, vol. 195, no. 2, pp. 216–222, 1987.

¹² M. D. Hoffman, A. Gelman, et al., "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo," *Journal of Machine Learning Research (JMLR)*, vol. 15, no. 1, pp. 1593–1623, 2014.

¹³ G. Casella and E. I. George, "Explaining the Gibbs Sampler," *The American Statistician*, vol. 46, no. 3, pp. 167–174, 1992.

¹⁴ An in-depth overview of probabilistic programming is provided in G. Barthe, J.-P. Katoen, and A. Silva, *Foundations of Probabilistic Programming*. Cambridge University Press, 2020.

Furthermore, probabilistic programming tools are often combined with autodifferentiation tools, which allows us to automatically compute the gradient of the target density for use in gradient-based MCMC algorithms.¹⁵ These features allow us to sample from the failure distribution of complex systems without the need for significant manual overhead.

```

struct ProbabilisticProgramming
    Δ          # distance function: Δ(s)
    mcmc_alg # e.g. Turing.NUTS()
    k_max     # number of samples
    d         # trajectory depth
    ε         # smoothing parameter
end

function sample_failures(alg::ProbabilisticProgramming, sys, ψ)
    Δ, mcmc_alg = alg.Δ, alg.mcmc_alg
    k_max, d, ε = alg.k_max, alg.d, alg.ε

    @model function rollout(sys, d; xo=fill(missing, d),
                           xa=fill(missing, d),
                           xs=fill(missing, d))
        p = NominalTrajectoryDistribution(sys, d)
        s ~ initial_state_distribution(p)
        s = [s, [zeros(length(s)) for i in 1:d]...]
        for t in 1:d
            D = disturbance_distribution(p, t)
            s = s[t]
            xo[t] ~ D.Do(s)
            o = sys.sensor(s, xo[t])
            xa[t] ~ D.Da(o)
            a = sys.agent(o, xa[t])
            xs[t] ~ D.Ds(s, a)
            s[t+1] = sys.env(s, a, xs[t])
        end
        Turing.@addlogprob! logpdf(Normal(0.0, ε), Δ(s))
    end

    return Turing.sample(rollout(sys, d), mcmc_alg, k_max)
end

```

Algorithm 6.3 writes the rollout function as a probabilistic program that can be used to sample from the smoothed failure distribution.¹⁶ Similar to algorithm 4.5, the probabilistic programming model samples an initial state from the initial state distribution and steps the system forward in time by sampling from the disturbance distribution at each time step. However, rather than explicitly drawing

¹⁵ A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed. SIAM, 2008.

Algorithm 6.3. Probabilistic programming algorithm for sampling from the failure distribution. The algorithm uses the `Turing.jl` package to specify the rollout function as a probabilistic model. It adds a log probability term equivalent to the smoothed indicator function in equation (6.5) to specify that we want to sample failure trajectories. It then generates samples using the specified MCMC algorithm. `Turing.jl` supports a variety of MCMC algorithms, including Metropolis-Hastings, HMC, NUTS, and Gibbs sampling.

¹⁶ We use a probabilistic programming package written for the Julia language called `Turing.jl`.

the samples, the model only specifies the distributions from which the samples are drawn. The probabilistic programming tool handles the sampling and keeps track of the probability associated with each draw automatically.

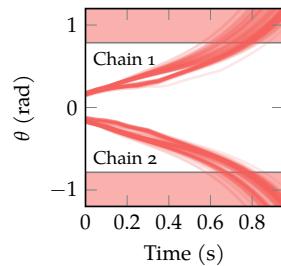
To specify that we want to sample failure trajectories, we add a log probability term for the smoothed indicator function in equation (6.5). Probabilistic programming tools often perform operations in log space for numerical stability. Adding this term in log space is equivalent to multiplying the target density by the smoothed indicator function. Example 6.5 demonstrates how to use algorithm 6.3 to sample from the failure distribution of the inverted pendulum system. It runs the algorithm twice to produce two chains that capture two distinct failure modes. In addition to smoothing, running multiple MCMC chains from different starting points is another method to improve performance of MCMC for failure distributions with multiple modes.

To use probabilistic programming to sample from the failure distribution of the inverted pendulum system, we can use the following code to set up the MCMC algorithm and distance function:

```
mcmc_alg = Turing.NUTS(10, 0.65, max_depth=6)
Δ(s) = max(robustness(s, ψ.formula, w=1.0), 0)
```

The code sets up the No U-Turn Sampler (NUTS) MCMC algorithm. Since NUTS relies on the gradient of the target density, we use smoothed robustness in the distance function so that the gradient exists. The first two parameters in the NUTS constructor are the number of adaptation steps and the target acceptance rate. The plot shows the result of running algorithm 6.3 with the specified parameters. We run the algorithm twice to produce two MCMC chains. Running multiple chains from different starting points is another method to improve performance for failure distributions with multiple modes.

Example 6.5. Sampling from the failure distribution of the inverted pendulum system using algorithm 6.3. The plot shows the result of running the algorithm twice to produce two MCMC chains. The initial samples that are not failures are discarded during the burn-in period.



6.5 Summary

- In general, it is difficult to compute the distribution over failures exactly, but we can compute its unnormalized density.

- Using an unnormalized density over failures, we can apply a variety of algorithms to draw samples.
- Rejection sampling works by drawing independent samples from a proposal distribution and accepting them with probability proportional to the ratio of the target density to the proposal density.
- The performance of rejection sampling depends on the choice of proposal distribution, and it can be difficult to select a good proposal distribution for high-dimensional systems.
- Markov chain Monte Carlo (MCMC) algorithms sample from the target distribution by drawing samples from a Markov chain and scale well to high-dimensional systems.
- MCMC is only guaranteed to converge to the target distribution in the limit of infinite samples, but we cannot generate an infinite number of samples in practice.
- We can use heuristics such as smoothing and gradient-based kernels to improve the performance of MCMC with a finite sample budget.
- Probabilistic programming is a tool that allows us to specify probabilistic models as computer programs and can be used to sample from the failure distribution of complex systems.

6.6 Exercises

Exercise 6.1. Consider the simple system shown in figure 6.1 where trajectories consist of only a single state that is sampled from a Gaussian distribution with mean 0 and variance 1^2 . The specification for this system is $\psi = \square(s \geq -1)$. What is the unnormalized probability density of the failure distribution $\bar{p}(\tau | \tau \notin \psi)$ for $\tau = -1.5$? What is $\bar{p}(\tau | \tau \notin \psi)$ for $\tau = 0$?

Solution: The probability density of the nominal trajectory distribution is $p(\tau) = \mathcal{N}(\tau | 0, 1^2)$. Therefore, $p(\tau | \tau \notin \psi) = \mathbb{1}\{\tau \notin \psi\} \mathcal{N}(\tau | 0, 1^2)$.

$$\begin{aligned}\bar{p}(-1.5 | \tau \notin \psi) &= (1)\mathcal{N}(-1.5 | 0, 1^2) \\ &= 0.13\end{aligned}$$

$$\begin{aligned}\bar{p}(0 \mid \tau \notin \psi) &= (0)\mathcal{N}(0 \mid 0, 1^2) \\ &= 0\end{aligned}$$

Exercise 6.2. Suppose we want to draw samples from a Beta distribution with parameters $\alpha = 5$ and $\beta = 5$. The unnormalized probability density of a Beta distribution is

$$\bar{p}(\tau) = \tau^{\alpha-1}(1-\tau)^{\beta-1} = \tau^4(1-\tau)^4$$

where $\tau \in [0, 1]$. We have access to a random number generator that can draw samples from a uniform distribution $\mathcal{U}(0, 1)$. We want to use rejection sampling with this random number generator as the proposal distribution to draw samples from the Beta distribution. What is the range of possible values we could select for the hyperparameter c ? What value should we select for c to maximize the efficiency of the algorithm?

Solution: We must select c such that

$$\begin{aligned}cq(\tau) &\geq \bar{p}(\tau) \\ c(1) &\geq \tau^4(1-\tau)^4 \\ c &\geq \tau^4(1-\tau)^4\end{aligned}$$

The maximum value of $\tau^4(1-\tau)^4$ occurs when $\tau = 0.5$ and is equal to 0.0039, so we must select $c \geq 0.0039$. To maximize the efficiency of the algorithm, we should select $c = 0.0039$.

Exercise 6.3. Show that the Gaussian kernel $g(\tau \mid \tau') = \mathcal{N}(\tau \mid \tau', \sigma^2)$ is symmetric.

Solution: To show that the kernel is symmetric, we must show that $g(\tau \mid \tau') = g(\tau' \mid \tau)$. We have

$$\begin{aligned}g(\tau \mid \tau') &= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\tau - \tau')^2}{2\sigma^2}\right) \\ g(\tau' \mid \tau) &= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\tau' - \tau)^2}{2\sigma^2}\right)\end{aligned}$$

Because $(\tau - \tau')^2 = (\tau' - \tau)^2$, we can conclude that $g(\tau \mid \tau') = g(\tau' \mid \tau)$.

Exercise 6.4. Consider a system where trajectories consist of only a single state that is sampled from a Beta distribution with parameters $\alpha = 5$ and $\beta = 5$. The unnormalized probability density of a Beta distribution is

$$\bar{p}(\tau) = \tau^{\alpha-1}(1-\tau)^{\beta-1} = \tau^4(1-\tau)^4$$

The specification for this system is $\psi = \square(\tau \leq 0.25)$. We want to use MCMC with a Gaussian kernel to sample from the failure distribution $\bar{p}(\tau \mid \tau \notin \psi)$. Suppose the current sample is $\tau = 0.4$. What is the probability of accepting a proposed sample $\tau' = 0.5$? What if the proposed sample was instead $\tau' = 0.3$ or $\tau' = 0.2$?

Solution: Because a Gaussian kernel is symmetric, the probability of accepting a proposed sample is $\bar{p}(\tau' \mid \tau \notin \psi) / \bar{p}(\tau \mid \tau \notin \psi)$. For $\tau' = 0.5$, this ratio evaluates to

$$\frac{0.5^4(1-0.5)^4}{0.4^4(1-0.4)^4} = 1.18$$

Therefore, we will accept the proposed sample $\tau' = 0.5$ with probability 1. For $\tau' = 0.3$, we will accept the proposed sample with probability

$$\frac{0.3^4(1-0.3)^4}{0.4^4(1-0.4)^4} = 0.586$$

For $\tau' = 0.2$, we will accept the proposed sample with probability

$$\frac{0}{0.4^4(1-0.4)^4} = 0$$

Exercise 6.5. In this exercise, we will prove that if we reject trajectories that are not failures after drawing samples from the smoothed failure distribution with smoothing parameter ϵ , the remaining samples will represent the failure distribution. Specifically, we will show that this process produces the same result we would obtain by applying rejection sampling with the smoothed failure distribution as the proposal distribution. In this case, the proposal distribution is $\bar{q}(\tau) = \mathcal{N}(\Delta\tau \mid 0, \epsilon^2)p(\tau)$ and the target density of the failure distribution is $\bar{p}(\tau \mid \tau \notin \psi) = \mathbb{1}\{\tau \notin \psi\}p(\tau)$. First, determine the smallest value for c such that $c\bar{q}(\tau) \geq \bar{p}(\tau \mid \tau \notin \psi)$ for all τ . Then, show that if we use this value for c , the rejection sampling process will produce the same result as drawing samples from the smoothed failure distribution and rejecting samples that are not failures. (*Hint:* Remember that $\Delta(\tau) = 0$ for all failure trajectories. You may find it helpful to break the proposal and target density into two cases with one case for failure trajectories and the other for success trajectories.)

Solution: We can rewrite the proposal density as

$$\bar{q}(\tau) = \begin{cases} \mathcal{N}(0 \mid 0, \epsilon^2)p(\tau) & \text{if } \tau \notin \psi \\ \mathcal{N}(\tau \mid 0, \epsilon^2)p(\tau) & \text{otherwise} \end{cases}$$

We can also rewrite the target density as

$$\bar{p}(\tau \mid \tau \notin \psi) = \begin{cases} p(\tau) & \text{if } \tau \notin \psi \\ 0 & \text{otherwise} \end{cases}$$

For trajectories that are not failures, we know that $\bar{q}(\tau) \geq \bar{p}(\tau \mid \tau \notin \psi)$ because $\mathcal{N}(\tau \mid 0, \epsilon^2)p(\tau) \geq 0$ for all τ . When $\tau \notin \psi$, we have

$$\begin{aligned} c\mathcal{N}(0 \mid 0, \epsilon^2)p(\tau) &\geq p(\tau) \\ c\mathcal{N}(0 \mid 0, \epsilon^2) &\geq 1 \\ c &\geq \frac{1}{\mathcal{N}(0 \mid 0, \epsilon^2)} \end{aligned}$$

Therefore, we should select $c = 1/\mathcal{N}(0 | 0, \epsilon^2)$.

If we perform rejection sampling with this value for c , we will accept samples with probability

$$\begin{aligned}\frac{\bar{p}(\tau | \tau \notin \psi)}{c\bar{q}(\tau)} &= \frac{\mathbb{1}\{\tau \notin \psi\} p(\tau)}{\frac{1}{\mathcal{N}(0|0,\epsilon^2)} \mathcal{N}(\Delta\tau | 0, \epsilon^2) p(\tau)} \\ &= \frac{\mathbb{1}\{\tau \notin \psi\}}{\frac{1}{\mathcal{N}(0|0,\epsilon^2)} \mathcal{N}(\Delta\tau | 0, \epsilon^2)} \\ &= \begin{cases} \frac{1}{\mathcal{N}(0|0,\epsilon^2)} \mathcal{N}(0|0,\epsilon^2) & \text{if } \tau \notin \psi \\ 0 & \text{otherwise} \end{cases} \\ &= \begin{cases} 1 & \text{if } \tau \notin \psi \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

Therefore, this process is equivalent to accepting all failure trajectories and rejecting all success trajectories.

Exercise 6.6. Why do gradient-based kernels tend to outperform Gaussian kernels?

Solution: Gradient-based kernels tend to outperform Gaussian kernels because they can better capture the structure of the target density. This property allows gradient-based kernels to explore the target density more efficiently than a random walk.

Exercise 6.7. Suppose we are using MCMC with a Gaussian kernel to sample a failure distribution that has multiple disjoint failure modes. We find that the algorithm is getting stuck in one of the failure modes and is not drawing samples from the other failure modes. We decide to switch to a gradient-based kernel to address this problem. Do we expect this change to improve the performance of the algorithm? Why or why not?

Solution: While switching to a gradient-based kernel may improve the sampling within the failure mode that the algorithm is currently stuck in, it may not improve the performance of the algorithm overall. Using the gradient of the failure distribution density to propose the next sample will not guide the algorithm to explore other failure modes because the density is still zero in the regions in between the failure modes. However, if we were to use a gradient-based kernel with a smoothed version of the failure distribution, we would expect the performance of the algorithm to begin sampling from the other failure modes.

7 Failure Probability Estimation

After searching for the potential failure modes of a system, we may also want to estimate its probability of failure. This chapter presents several techniques for estimating this quantity from samples. We begin by discussing a direct estimation approach that uses samples from the nominal trajectory distribution to estimate the probability of failure. If failures are rare, this approach may be inefficient and require a large number of samples to produce a good estimate. The remainder of the chapter discusses more efficient estimation techniques based on importance sampling. Importance sampling techniques artificially increase the likelihood of failure trajectories by sampling from a proposal distribution. We discuss several variations of importance sampling and conclude by presenting a nonparametric algorithm that estimates the probability of failure from a sequence of samples.

7.1 Direct Estimation

The probability of failure for a given system and specification is defined mathematically as

$$p_{\text{fail}} = \mathbb{E}_{\tau \sim p(\cdot)} [\mathbf{1}\{\tau \notin \psi\}] = \int \mathbf{1}\{\tau \notin \psi\} p(\tau) d\tau \quad (7.1)$$

where $\mathbf{1}\{\cdot\}$ is the indicator function. The expectation is taken over the nominal trajectory distribution for the system.¹ Given a set of m trajectories from this distribution, we can produce an estimate \hat{p}_{fail} of the probability of failure by treating the problem as a parameter learning problem, where the parameter of interest is the parameter of a Bernoulli distribution. We can then apply the maximum likelihood or Bayesian methods from chapter 2 to calculate \hat{p}_{fail} .

¹ Note that the right-hand side of equation (7.1) is equivalent to the denominator in equation (6.1). In other words, the probability of failure is the normalizing constant for the failure distribution.

7.1.1 Maximum Likelihood Estimate

The maximum likelihood estimate of the probability of failure is

$$\hat{p}_{\text{fail}} = \frac{1}{m} \sum_{i=1}^m \mathbb{1}\{\tau_i \notin \psi\} = \frac{n}{m} \quad (7.2)$$

where n is the number of samples that resulted in a failure and m is the total number of samples. Algorithm 7.1 uses direct sampling to implement this estimator. It performs m rollouts and computes the probability of failure according to equation (7.2).

```
struct DirectEstimation
    d # depth
    m # number of samples
end

function estimate(alg::DirectEstimation, sys, ψ)
    d, m = alg.d, alg.m
    ts = [rollout(sys, d=d) for i in 1:m]
    return mean(isfailure(ψ, τ) for τ in ts)
end
```

Algorithm 7.1. The direct estimation algorithm for estimating the probability of failure. The algorithm performs rollouts to a depth d to generate m samples from the nominal trajectory distribution. It then applies equation (7.2) to compute \hat{p}_{fail} and returns the result.

We can evaluate the accuracy of an estimator using metrics such as bias, consistency, and variance (example 7.1). Equation (7.2) provides an empirical estimate of the probability of failure by computing the sample mean of a set of samples drawn from a Bernoulli distribution with parameter p_{fail} . The sample mean is an unbiased estimator of the true mean of a Bernoulli distribution, so the estimator is unbiased. We can calculate the variance of this estimator by dividing the variance of a Bernoulli distribution by the number of samples:

$$\text{Var}[\hat{p}_{\text{fail}}] = \frac{p_{\text{fail}}(1 - p_{\text{fail}})}{m} \quad (7.3)$$

The square root of this quantity is known as the *standard error* of the estimator. A lower variance means that the sample mean will be closer to the true mean on average and therefore indicates a more accurate estimator. In the limit of infinite samples, the variance approaches zero, so the estimator is consistent.

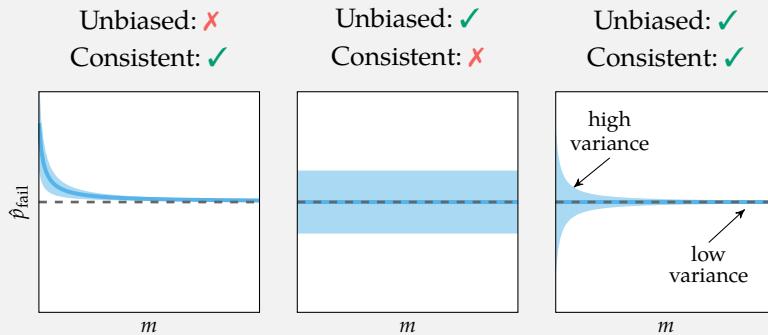
Bias, consistency, and variance are three common properties used to evaluate the quality of an estimator. An estimator that produces \hat{p}_{fail} is *unbiased* if it predicts the true value in expectation:

$$\mathbb{E}[\hat{p}_{\text{fail}}] = p_{\text{fail}}$$

An estimator is *consistent* if it converges to the true value in the limit of infinite samples:

$$\lim_{m \rightarrow \infty} \hat{p}_{\text{fail}} = p_{\text{fail}}$$

For example, given a set of samples drawn independently from the same distribution, the sample mean is an unbiased and consistent estimator of the distribution's true mean. The variance of the estimator quantifies the spread of the estimates around the true value. For the sample mean example, the variance will decrease as the number of samples increases. The plots below illustrate these concepts. The shaded regions reflect the variance of the estimator.

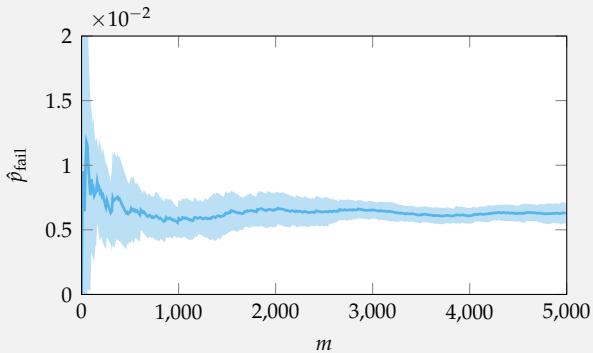


In general, we want to use an estimator that is unbiased, consistent, and has low variance. However, we are sometimes forced to trade off between these metrics to achieve the best efficiency for complex problems.

Example 7.1. Common metrics used to evaluate estimators. The plots show predictions of three different estimators with shaded regions to represent the variance.

Equation (7.3) provides insight on the accuracy of the estimator in equation (7.2). However, it is expressed in terms of the true probability of failure p_{fail} , which is the quantity we want to estimate. Therefore, we cannot apply this equation to directly assess the accuracy of the output of algorithm 7.1. We can instead use it reason about qualitative trends. For example, equation (7.3) indicates that we decrease the variance of our estimator by collecting more samples. Example 7.2 illustrates this trend on the grid world problem.

We demonstrate the effect of equation (7.3) empirically by running 10 trials of algorithm 7.1 on the grid world problem. We compute the empirical mean and variance of \hat{p}_{fail} across all 10 trials after each new sample. The plot below shows the results of this experiment.



As predicted by equation (7.3), the variance decreases as the number of samples m increases.

Example 7.2. The empirical mean and variance of the direct estimator for the grid world problem computed over 10 trials of algorithm 7.1. The depth d is set to 50 and the probability of slipping is set to 0.8. The blue line shows the mean of \hat{p}_{fail} for all 10 trials, and the shaded region represents one standard deviation above and below the mean.

In addition to the number of samples, the true probability of failure p_{fail} also has an impact on the relative accuracy of the estimator. As the true probability of failure decreases, the number of samples required to achieve a given level of accuracy increases (see exercise 7.2). For systems in which failure events are rare, we may require a large number of samples to produce an accurate estimate for the probability of failure using algorithm 7.1. Section 7.2 introduces importance sampling, which can be used to improve the efficiency in these scenarios.

7.1.2 Bayesian Estimate

Bayesian failure probability estimation may improve accuracy in scenarios with limited data or rare failure events. For example, suppose we want to estimate the probability of an aircraft collision from an aviation safety database that contains flight records from the past week. If there are no recorded midair collisions for the past week, the maximum likelihood estimate for the probability of a midair collision would be zero. Believing that there is zero chance of a midair collision is not a reasonable conclusion unless our prior hypothesis was, for example, that all flights were perfectly safe.

Bayesian estimation techniques incorporate a prior belief about the safety of the system and maintain a full distribution over the probability of failure. Since \hat{p}_{fail} is the parameter of a Bernoulli distribution, the distribution over the probability of failure is a beta distribution. The posterior distribution after observing n failures in m samples is

$$p_{\text{fail}} \sim \text{Beta}(\alpha + n, \beta + m - n) \quad (7.4)$$

if we start with a prior of $\text{Beta}(\alpha, \beta)$.

Algorithm 7.2 implements this estimator given a prior distribution. It performs m rollouts and computes the posterior distribution over the probability of failure according to equation (7.4). The prior distribution should be selected to reflect our prior beliefs about the probability of failure based on domain knowledge. If we do not have any reason to believe that one value of p_{fail} is more probable than another value in the absence of data, we can use a uniform prior of $\text{Beta}(1, 1)$. Figure 7.1 shows an example of how the posterior distribution changes as more samples are collected. We can convert the distribution over the probability of failure into a point estimate by computing its mean or mode. The mean of the distribution $\text{Beta}(\alpha, \beta)$ is

$$\frac{\alpha}{\alpha + \beta} \quad (7.5)$$

and the mode is

$$\frac{\alpha - 1}{\alpha + \beta - 2} \quad (7.6)$$

assuming α and β are greater than 1.

Maintaining a distribution over the probability of failure allows us to explicitly quantify the uncertainty in our estimate. For example, suppose we have a target level of safety corresponding to a probability of failure less than or equal to δ .

```

struct BayesianEstimation
    prior::Beta # from Distributions.jl
    d          # depth
    m          # number of samples
end

function estimate(alg::BayesianEstimation, sys, ψ)
    prior, d, m = alg.prior, alg.d, alg.m
    ts = [rollout(sys, d=d) for i in 1:m]
    n, m = sum(isfailure(ψ, t) for t in ts), length(ts)
    return Beta(prior.α + n, prior.β + m - n)
end

```

Algorithm 7.2. The Bayesian estimation algorithm for estimating a distribution over the probability of failure. The algorithm performs rollouts to a depth d to generate m samples from the nominal trajectory distribution. Using the `prior`, it then applies equation (7.4) to compute the posterior distribution over the probability of failure and returns the result.

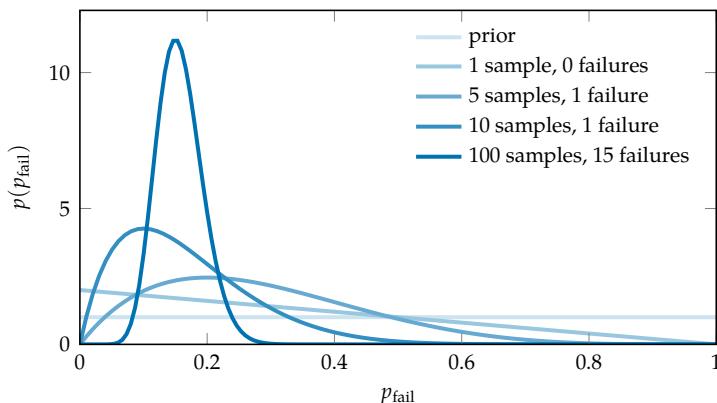


Figure 7.1. Bayesian estimation applied to the grid world problem with a probability of slipping set to 0.5. We begin with a uniform prior of $\text{Beta}(1,1)$ and determine a distribution over the probability of failure by applying algorithm 7.2 with 1, 5, 10, and 100 samples. As we observe more samples, the distribution over the probability of failure becomes more concentrated around a small range of probabilities.

We are interested in the quantity $p(p_{\text{fail}} < \delta)$, which is the probability that the true probability of failure is less than or equal to δ . This quantity is given by the cumulative distribution function of the posterior distribution over the probability of failure.² The quantiles of the posterior distribution can be used to compute confidence intervals in a similar manner. Example 7.3 demonstrates this process.

7.2 Importance Sampling

Importance sampling algorithms increase the efficiency of sampling-based estimation techniques. Instead of sampling from the nominal trajectory distribution p , they sample from a proposal distribution q that assigns higher likelihood to areas of greater “importance.”³ To estimate the probability of failure using these samples, we must transform the expectation in equation (7.1) to an expectation over q :

$$p_{\text{fail}} = \mathbb{E}_{\tau \sim p(\cdot)} [\mathbb{1}\{\tau \notin \psi\}] \quad (7.7)$$

$$= \int p(\tau) \mathbb{1}\{\tau \notin \psi\} d\tau \quad (7.8)$$

$$= \int p(\tau) \frac{q(\tau)}{q(\tau)} \mathbb{1}\{\tau \notin \psi\} d\tau \quad (7.9)$$

$$= \int q(\tau) \frac{p(\tau)}{q(\tau)} \mathbb{1}\{\tau \notin \psi\} d\tau \quad (7.10)$$

$$= \mathbb{E}_{\tau \sim q(\cdot)} \left[\frac{p(\tau)}{q(\tau)} \mathbb{1}\{\tau \notin \psi\} \right] \quad (7.11)$$

For equation (7.11) to be valid, we require that $q(\tau) > 0$ wherever $p(\tau)\mathbb{1}\{\tau \notin \psi\} > 0$. This condition is satisfied as long as the proposal distribution assigns a nonzero likelihood to all failure trajectories that are possible under p .

Given samples from $q(\cdot)$, we can estimate the probability of failure based on equation (7.11) as

$$\hat{p}_{\text{fail}} = \frac{1}{m} \sum_{i=1}^m \frac{p(\tau_i)}{q(\tau_i)} \mathbb{1}\{\tau_i \notin \psi\} \quad (7.12)$$

Algorithm 7.3 implements this estimator. Equation (7.12) is an unbiased estimator of the true probability of failure. It corresponds to a weighted average of samples from the proposal distribution:

$$\hat{p}_{\text{fail}} = \frac{1}{m} \sum_{i=1}^m w_i \mathbb{1}\{\tau_i \notin \psi\} \quad (7.13)$$

² The cumulative distribution function of a Beta distribution is the regularized incomplete beta function. Software packages such as `Distributions.jl` provide implementations of both the cumulative distribution function and the quantile function for the Beta distribution.

³ This proposal distribution has similar properties to the proposal distribution introduced in section 6.2 for rejection sampling.

Suppose that we run algorithm 7.2 on the collision avoidance problem with $m = 100$ samples and observe no failures. Assuming we begin with a uniform prior, the posterior distribution over the probability over failure is Beta(1, 101). Suppose we are also given a safety requirement for the system stating that p_{fail} must not exceed 0.01. We can compute $p(p_{\text{fail}} < 0.01)$ from the cumulative distribution function of the beta distribution using the following code:

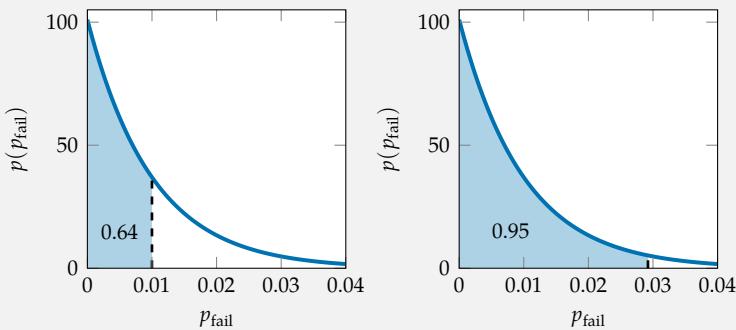
```
using Distributions
posterior = Beta(1, 101)
confidence = cdf(posterior, 0.01)
```

The `confidence` variable is equal to 0.6376, indicating that we are 63.76% confident that the true probability of failure is less than 0.01.

Suppose we instead want to determine a 95% confidence bound on the probability of failure. We can compute this bound using the quantile function of the beta distribution as follows:

```
bound = quantile(posterior, 0.95)
```

The `bound` variable is equal to 0.0292, so we can be 95% confident that the true probability of failure is less than 0.0292. The plots below show these results.



Example 7.3. Quantifying uncertainty in the probability estimate produced by algorithm 7.2. The plots show the posterior distribution Beta(1, 101). The shaded region in the first plot represents the probability that the true probability of failure is less than or equal to 0.01. The shaded region in the second plot shows the 95% confidence bound.

where the weights are $w_i = p(\tau_i)/q(\tau_i)$. These weights are sometimes referred to as *importance weights*. Trajectories that are more likely under the nominal trajectory distribution have higher importance weights.

```

struct ImportanceSamplingEstimation
    p # nominal distribution
    q # proposal distribution
    m # number of samples
end

function estimate(alg::ImportanceSamplingEstimation, sys, ψ)
    p, q, m = alg.p, alg.q, alg.m
    ts = [rollout(sys, q) for i in 1:m]
    ps = [pdf(p, τ) for τ in ts]
    qs = [pdf(q, τ) for τ in ts]
    ws = ps ./ qs
    return mean(w * isfailure(ψ, τ) for (w, τ) in zip(ws, ts))
end

```

Algorithm 7.3. The importance sampling estimation algorithm for estimating the probability of failure. The algorithm generates m samples from the proposal distribution q . It then computes the importance weights for the samples and applies equation (7.12) to compute \hat{p}_{fail} .

7.2.1 Optimal Proposal Distribution

The accuracy and efficiency of importance sampling approaches is highly dependent on the proposal distribution. The variance of the estimator in equation (7.12) is

$$\text{Var}[\hat{p}_{\text{fail}}] = \frac{1}{m} \mathbb{E}_{\tau \sim q(\cdot)} \left[\frac{(p(\tau) \mathbb{1}\{\tau \notin \psi\} - q(\tau)p_{\text{fail}})^2}{q(\tau)} \right] \quad (7.14)$$

In general, we want to select a proposal distribution that makes this variance low, and the optimal proposal distribution is the one that minimizes this variance.

It is evident from equation (7.14) that we can achieve a variance of zero when

$$q^*(\tau) = \frac{p(\tau) \mathbb{1}\{\tau \notin \psi\}}{p_{\text{fail}}} \quad (7.15)$$

This distribution corresponds to the failure distribution $p(\tau \mid \tau \notin \psi)$. As noted in chapter 6, computing this distribution is not possible in practice since we often do not know the full set of failure trajectories and the normalizing constant p_{fail} is the quantity we are trying to estimate. Our goal is therefore to select a proposal distribution that is as close as possible to the failure distribution.

7.2.2 Proposal Distribution Selection

One way to select a proposal distribution for importance sampling is based on domain knowledge. For example, if we know that collisions between aircraft tend to occur more often when they have high vertical rates, we may select a proposal distribution that assigns higher likelihood to high vertical rates. It is important to ensure that the proposal distribution has adequate overlap with the failure distribution. In other words, it should assign high likelihood to likely failure trajectories. A poorly selected proposal distribution can lead to high variance and result in poor performance (see example 7.4).

We can also select a proposal distribution based on samples from the failure distribution. Specifically, we can approximate the failure distribution by fitting a distribution to samples obtained using the algorithms in chapter 6. The resulting distribution will approximate the optimal proposal distribution and may improve the performance over a hand-designed proposal distribution (see figure 7.2). The efficacy of this approach, however, is dependent on our ability to produce a good fit to the failure distribution, which may be difficult in high-dimensional spaces with multiple failure modes.

7.2.3 Multiple Importance Sampling

We can also draw samples from multiple proposal distributions and combine them to form a more robust estimate. This approach is known as *multiple importance sampling* (MIS). Suppose we draw m samples such that

$$\tau_i \sim q_i(\cdot) \text{ for all } i \in \{1, \dots, m\} \quad (7.16)$$

where $q_i(\cdot)$ is the proposal distribution used to generate the i th sample τ_i . We can still use equation (7.13) to estimate the probability of failure for MIS, but we must modify the importance weights to account for multiple proposal distributions.

Several weighting schemes will result in an unbiased estimate. Algorithm 7.4 implements multiple importance sampling with two different weighting schemes.⁴ The first weighting scheme is

$$w_i = \frac{p(\tau_i)}{q_i(\tau_i)} \quad (7.17)$$

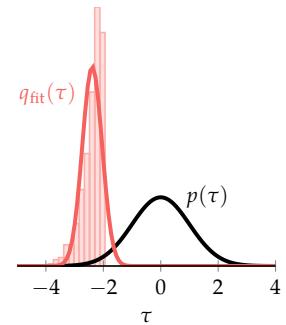


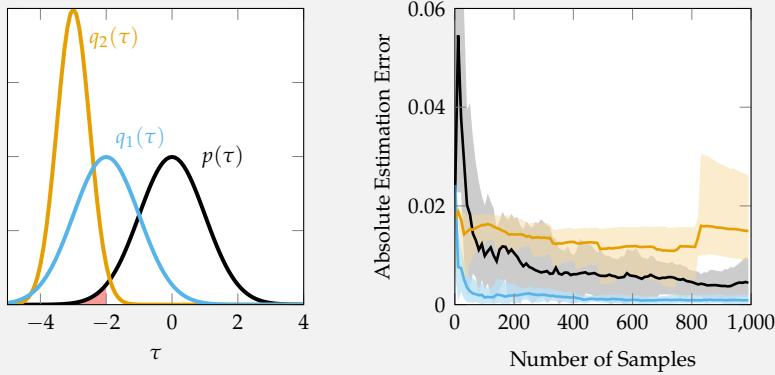
Figure 7.2. Fitting a proposal distribution to samples from the failure distribution. The plot shows a histogram of samples from the failure distribution produced using rejection sampling. We fit a Gaussian distribution to these samples (red) using maximum likelihood estimation to use as a proposal distribution. The nominal distribution $p(\tau)$ is shown in black.

⁴ For a detailed discussion, see V. Elvira, L. Martino, D. Luengo, and M. F. Bugallo, "Generalized Multiple Importance Sampling," *Statistical Science*, vol. 34, no. 1, pp. 129–155, 2019.

Consider the simple Gaussian problem shown below where failures occurs at values less than -2 (red shaded region). The plot on the left shows two proposal distributions we could use for importance sampling. The first proposal distribution q_1 shifts the nominal distribution toward the failure region and assigns high likelihood to likely failure trajectories. The second proposal distribution q_2 is shifted toward the failure region but it still does not assign high likelihood to likely failures. Therefore, we expect q_1 to result in better estimates than q_2 .

The plot on the right shows the estimation error when performing importance sampling with each proposal distribution compared to direct estimation. The shaded region represents the 90% empirical confidence bounds on the error. As expected, q_1 results in a lower estimation error and a lower variance than q_2 and direct estimation. Performing importance sampling with q_2 results in worse performance than direct estimation.

Example 7.4. Performance comparison of two hand-designed proposal distributions for the simple Gaussian problem where failures occur at values less than -2 (red region). The first plot shows the nominal distribution and two possible proposal distributions. The second plot shows the estimation error for direct estimation compared to the estimation error of importance sampling for the two distributions.



```

struct MultipleImportanceSamplingEstimation
    p          # nominal distribution
    qs         # proposal distributions
    weighting # weighting scheme: ws = weighting(p, qs, ts)
end

smis(p, qs, ts) = [pdf(p, τ) / pdf(q, τ) for (q, τ) in zip(qs, ts)]
dmmis(p, qs, ts) = [pdf(p, τ) / mean(pdf(q, τ) for q in qs) for τ in ts]

function estimate(alg::MultipleImportanceSamplingEstimation, sys, ψ)
    p, qs, weighting = alg.p, alg.qs, alg.weighting
    ts = [rollout(sys, q) for q in qs]
    ws = weighting(p, qs, ts)
    return mean(w * isfailure(ψ, τ) for (w, τ) in zip(ws, ts))
end

```

Algorithm 7.4. The multiple importance sampling algorithm for estimating the probability of failure. The algorithm generates a sample for each proposal distribution in `qs`. It then computes the importance weights using the `weighting` function provided and applies equation (7.13) to compute \hat{p}_{fail} . The `smis` and `dmmis` functions implements the s-MIS and DM-MIS weighting schemes respectively.

where the weight for each sample is computed using only the proposal that was used to generate it. This weighting scheme, which we refer to as standard MIS (s-MIS), is most similar to the importance sampling estimator for a single proposal distribution (equation (7.12)).

Instead of considering each proposal individually, we can also view the samples as if they were drawn in a deterministic order from a mixture distribution composed of all proposal distributions. This paradigm leads to the deterministic mixture weighting scheme (DM-MIS):

$$w_i = \frac{p(\tau_i)}{\frac{1}{m} \sum_{j=1}^m q_j(\tau_i)} \quad (7.18)$$

The denominator of equation (7.18) corresponds to the probability density of the mixture distribution evaluated at τ_i .

Figure 7.3 visualizes the weighting schemes, and example 7.5 compares their performance on a two-dimensional Gaussian problem. While both schemes are unbiased, DM-MIS has been shown to have lower variance than s-MIS. However, DM-MIS requires computing the likelihood of each sample under all proposal distributions, which may be computationally expensive if the number of proposal distributions is large.

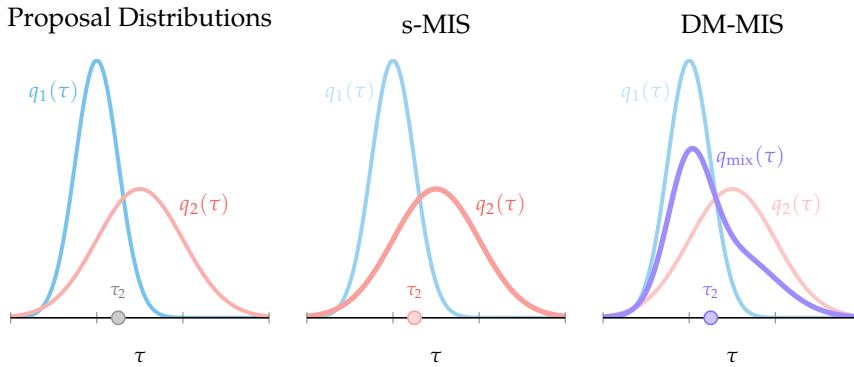


Figure 7.3. Visualization of the two most common multiple importance sampling (MIS) weighting schemes. In this case, we have two proposal distributions, q_1 and q_2 , and we want to determine the importance weight for τ_2 , which was sampled from q_2 . In s-MIS, we consider only q_2 , while in DM-MIS, we consider the mixture distribution of q_1 and q_2 .

7.3 Adaptive Importance Sampling

Adaptive importance sampling algorithms automatically tune a proposal or set of proposals to help alleviate the challenge of designing an effective set of proposals by hand. These algorithms use samples to iteratively adapt the proposal distributions to move toward the failure distribution. In this section, we present two common adaptive importance sampling algorithms: the cross entropy method and population Monte Carlo. The cross entropy method adapts a single proposal distribution, while population Monte Carlo adapts a set of proposal distributions.

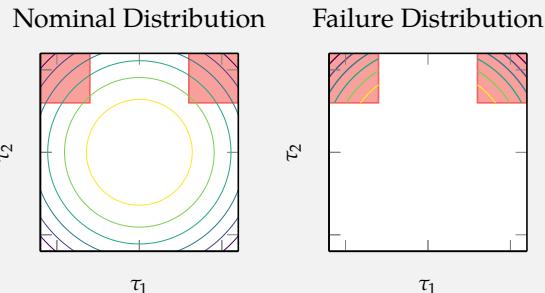
7.3.1 Cross Entropy Method

The *cross entropy method* iteratively fits a proposal distribution using samples.⁵ The algorithm requires selecting a form for the proposal distribution that is described by a set of parameters. A common choice is a multivariate Gaussian distribution, which is parameterized by a mean vector and a covariance matrix. The goal of the cross entropy method is to find the set of parameters that minimizes the *cross entropy* between the proposal distribution and the failure distribution. Cross entropy is an idea used in information theory that provides a measure of distance between two probability distributions.

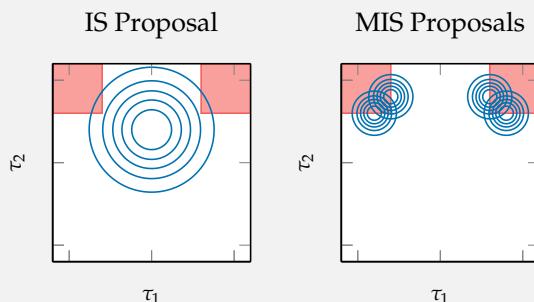
We can determine an approximate solution to this minimization problem using samples drawn from an initial proposal distribution q . For many common distribution types, minimizing the cross entropy is equivalent to computing the weighted maximum likelihood estimate with weights based on the proposal

⁵ For a detailed overview, see P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, "A Tutorial on the Cross-Entropy Method," *Annals of Operations Research*, vol. 134, pp. 19–67, 2005.

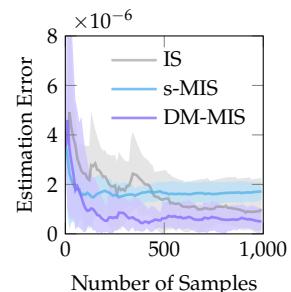
Suppose we want to estimate the probability of failure for the two-dimensional Gaussian system shown below. The nominal distribution is a multivariate Gaussian distribution with a mean at the center of the figure, and the failure region is composed of the two shaded red regions. The plots below show the log density of both distributions.



Most of the probability mass for the failure distribution is concentrated in the central corners of the two modes, and a good proposal distribution should assign high likelihood in those areas. If we only use one multivariate Gaussian proposal distribution, we need to select a wide distribution to ensure that it covers both failure modes (left). We can improve performance by selecting multiple proposal distributions that together cover both failure modes (right). The plot in the caption compares the performance of importance sampling (IS) to multiple importance sampling with the two different weighting schemes. The shaded region represents the 90% empirical confidence bounds on the error. The DM-MIS weighting scheme results in better performance than the s-MIS weighting scheme.



Example 7.5. Performance comparison of importance sampling to multiple importance sampling for a two-dimensional Gaussian problem. The plot on the left shows a single proposal distribution that can be used for importance sampling on this problem, while the plot on the right shows a set of proposal distributions that can be used for MIS. The plot below shows the estimation error for IS compared to the estimation error of MIS with the two different weighting schemes.



density and failure density.⁶ In these cases, the weight for a given sample τ drawn from the distribution q is

$$w = \frac{\mathbb{1}\{\tau \notin \psi\} p(\tau)}{q(\tau)} \quad (7.19)$$

where p is the nominal trajectory distribution.

If failures are rare under the initial proposal distribution, it is possible that no samples will be failures, and the weights computed in equation (7.19) will all be zero. To address this challenge, the cross entropy algorithm iteratively solves a relaxed version of the problem that relies on an objective function f . Similar to the objective functions introduced in section 4.5, the objective function should assess how close a trajectory is to a failure.⁷ The objective value must be greater than zero for trajectories that are not failures and less than or equal to zero for failure trajectories. For systems with temporal logic specifications, we can use the robustness as the objective function.

We can rewrite the goal of the cross entropy method in terms of the objective function as finding the set of parameters that minimizes the cross entropy between the proposal distribution and $p(\tau \mid f(\tau) \leq 0)$. For systems with rare failure events, we gradually make progress toward this goal by solving a series of relaxed problems where we instead minimize the cross entropy between the proposal and $p(\tau \mid f(\tau) \leq \gamma)$ for a given threshold $\gamma > 0$. The weights used in maximum likelihood estimation for the relaxed problem are

$$w = \frac{\mathbb{1}\{f(\tau) \leq \gamma\} p(\tau)}{q(\tau)} \quad (7.20)$$

At each iteration, we select the threshold γ based on our current set of samples to ensure that a fraction of the weights will be nonzero (figure 7.4).

Algorithm 7.5 implements the cross entropy method. At each iteration, we draw samples from the current proposal distribution and compute their objective values. We then select the threshold γ as the highest objective value from a set of *elite samples*. The elite samples are the m_{elite} samples with the lowest objective values. Since our ultimate goal is to approach the failure distribution, we ensure that the threshold does not become negative by clipping it at zero. Given this threshold, we compute the weights using equation (7.20) and fit a new proposal distribution to the samples. After repeating this process for a fixed number of iterations, algorithm 7.5 performs importance sampling (algorithm 7.3) with the

⁶ Minimizing the cross entropy is equivalent to weighted maximum likelihood estimation for distributions in the natural exponential family. The natural exponential family includes many common distributions such as the Gaussian, geometric, exponential, categorical, and Beta distributions.

⁷ For example, an objective function for the aircraft collision avoidance problem might output the miss distance between the two aircraft.

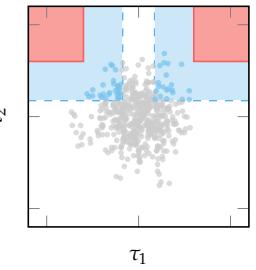


Figure 7.4. Threshold selection for a two-dimensional Gaussian problem with two failure modes. The red shaded region shows the failure region. None of the current samples overlap with the failure region, so we relax the problem by expanding to the blue region that contains a desired fraction of the samples. The blue samples are the top 10% of samples with the lowest objective values.

```

struct CrossEntropyEstimation
    p      # nominal trajectory distribution
    q0   # initial proposal distribution
    f      # objective function f( $\tau$ ,  $\psi$ )
    k_max # number of iterations
    m      # number of samples per iteration
    m_elite # number of elite samples
end

function estimate(alg::CrossEntropyEstimation, sys,  $\psi$ )
    k_max, m, m_elite = alg.k_max, alg.m, alg.m_elite
    p, q, f = alg.p, alg.q0, alg.f
    for k in 1:k_max
        ts = [rollout(sys, q) for i in 1:m]
        Y = [f( $\tau$ ,  $\psi$ ) for  $\tau$  in ts]
        order = sortperm(Y)
        y = max(0, Y[order[m_elite]])
        ps = [pdf(p,  $\tau$ ) for  $\tau$  in ts]
        qs = [pdf(q,  $\tau$ ) for  $\tau$  in ts]
        ws = ps ./ qs
        ws[Y .> y] .= 0
        q = fit(typeof(q), ts, ws=ws)
    end
    return estimate(ImportanceSamplingEstimation(p, q, m), sys,  $\psi$ )
end

```

Algorithm 7.5. The cross entropy method for estimating the probability of failure. At each iteration, the algorithm draws trajectory samples from the current proposal distribution and computes their objective values. It then sorts the samples by objective value and uses the `m_elite` samples with the lowest objective values to compute a threshold value. Using the threshold, the algorithm computes the weights and fits a new proposal distribution to the samples. The `fit` function is specific to the type of proposal distribution used and should perform weighted maximum likelihood estimation. After `k_max` iterations, the algorithm calls algorithm 7.3 to produce an estimate of the probability of failure using the final proposal distribution.

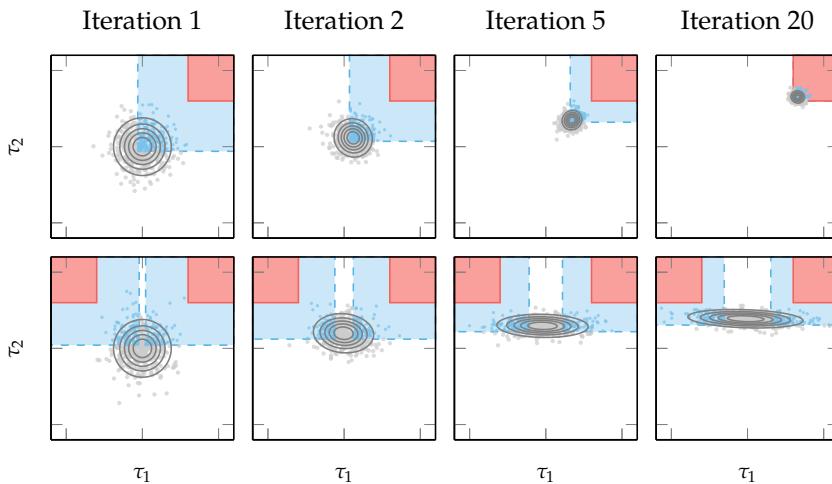


Figure 7.5. Visualization of the cross entropy method on a problem with a single failure mode (top row) and two failure modes (bottom row). The proposal distribution takes the form of a multivariate Gaussian distribution, and the blue samples are the elite samples for each iteration. For a single failure mode, the threshold reaches the failure region and is able to approximate the failure distribution. For two failure modes, the proposal distribution must become wide to capture both failure modes, and the algorithm does not perform as well.

final proposal distribution to produce an estimate of the probability of failure. Figure 7.6 demonstrates the progression of the algorithm on a simple problem.

Some implementations of the cross entropy method increase efficiency by using the samples produced across all iterations of the algorithm to estimate the probability of failure. They keep track of the proposal distribution used to generate the samples at each iteration and view the problem as an instance of MIS. They produce an estimate using the weighting schemes from section 7.2.3. It is important to note in this case, however, that the proposal for each iteration depends on the previous proposal. Since the proposals are not independent from one another, the DM-MIS weighting scheme will no longer be unbiased.

The performance of the cross entropy algorithm is sensitive to the form of the proposal distribution. The algorithm may perform poorly if the proposal distribution is not expressive enough to adequately capture the shape of the failure distribution. This behavior is particularly apparent for complex systems with high-dimensional, multimodal failure distributions. For example, if we select a Gaussian proposal distribution for a system with two failure modes, the algorithm will struggle to find a proposal distribution that captures both failure modes (figure 7.5). One solution is to use a mixture of Gaussians for multimodal failure distributions (figure 7.7), but this approach requires knowing the number of failure modes in advance, which is often not possible in practice. In these cases, an adaptive MIS approach such as population Monte Carlo may perform better.

7.3.2 Population Monte Carlo

Population Monte Carlo (PMC) is an adaptive MIS algorithm that maintains a set, or *population*, of proposal distributions (algorithm 7.6).⁸ Figure 7.8 shows a single step of the algorithm. We begin with an initial population of m proposals that is spread across the space of proposal distributions. For example, we could use a set of multivariate Gaussian distributions with a fixed covariance and different means. It is important to ensure that the initial population is sufficiently diverse to capture all failure modes.

At each iteration, the algorithm draws a single sample from each proposal distribution in the population. It then computes a weight for each sample in the same way the weights are computed for the cross entropy method (equation (7.19)). Samples in regions of high likelihood under the failure distribution will receive higher weights. To adapt the proposal distributions, PMC uses the weights to

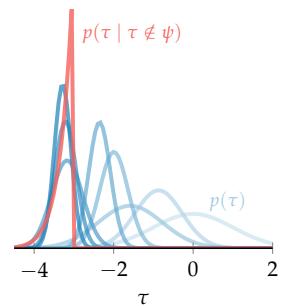


Figure 7.6. The cross entropy method for a one-dimensional Gaussian problem. The plot shows the Gaussian proposal distribution at each iteration of the algorithm with darker distributions representing later iterations. The distributions start at the nominal distribution and gradually move toward the failure distribution (red).

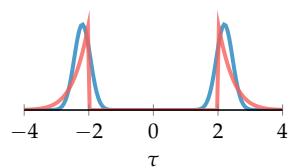


Figure 7.7. Example of a Gaussian mixture model proposal distribution for a one-dimensional problem with two failure modes. The proposal distribution is a mixture of two Gaussians (blue) that approximates the multimodal failure distribution (red).

⁸ O. Cappé, A. Guillin, J.-M. Marin, and C.P. Robert, “Population Monte Carlo,” *Journal of Computational and Graphical Statistics*, vol. 13, no. 4, pp. 907–929, 2004.

```

struct PopulationMonteCarloEstimation
    p          # nominal trajectory distribution
    qs         # vector of initial proposal distributions
    weighting # weighting scheme: ws = weighting(p, qs, τs)
    k_max     # number of iterations
end

function estimate(alg::PopulationMonteCarloEstimation, sys, ψ)
    p, qs, weighting = alg.p, alg.qs, alg.weighting
    k_max, m = alg.k_max, length(qs)
    for k in 1:k_max
        ts = [rollout(sys, q) for q in qs]
        ws = [pdf(p, τ) * isfailure(ψ, τ) / pdf(q, τ)
              for (q, τ) in zip(qs, ts)]
        resampler = Categorical(ws ./ sum(ws))
        qs = [proposal(qs[i], ts[i]) for i in rand(resampler, m)]
    end
    mis = MultipleImportanceSamplingEstimation(p, qs, weighting)
    return estimate(mis, sys, ψ)
end

```

Algorithm 7.6. The population Monte Carlo algorithm for estimating the probability of failure. At each iteration, the algorithm draws trajectory samples from the proposal distributions in the population, computes their weights, and resamples to produce new proposal distributions. The `proposal` function is specific to the trajectory distribution and creates a proposal distribution from a sample. After `k_max` iterations, the algorithm calls algorithm 7.4 using the specified `weighting` scheme to produce an estimate of the probability of failure using the final set of proposal distributions.

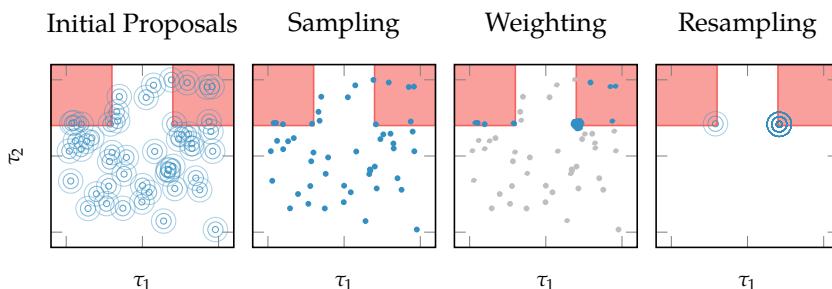


Figure 7.8. One iteration of the population Monte Carlo algorithm. For the weighting step, gray samples have zero weight, and the size of the blue samples is proportional to their weight. The resampling step produces new proposal distributions that are centered in high likelihood regions of the failure distribution.

perform a resampling step. In this step, we redraw m samples from the population of samples with probability proportional to their weights. We then reconstruct the population of proposal distributions using the resulting samples. For example, if we are using proposals in the form of multivariate Gaussian distributions, we could create new proposals with the same fixed covariance and means centered at each sample.

Over time, the population of proposal distributions should cover high likelihood regions of the failure distribution. After a fixed number of iterations, we perform MIS using the final population to estimate the probability of failure. We can use either of the weighting schemes from section 7.2.3 to produce the estimate. Similar to the cross entropy method, we could instead use the samples produced during all iterations of the algorithm to estimate the probability of failure, noting that the estimate in this case may no longer be unbiased.

Using multiple proposal distributions allows us to represent complex, multi-modal failure distributions. However, the performance of PMC is still dependent on the number of proposal distributions and their ability to cover the space of possible proposals. If the number of proposal distributions is too small or the initial proposals are not sufficiently diverse, the algorithm may miss failures modes and produce an inaccurate estimate. Furthermore, the stochastic nature of the resampling procedure can lead to a loss of diversity in the proposal distributions over time. For example, the proposals may collapse to a single failure mode or a subset of the failure modes.

7.4 Sequential Monte Carlo

The sampling, weighting, and resampling components of algorithm 7.6 form the basis for a more general framework used in the field of Bayesian inference called *sequential Monte Carlo* (SMC).⁹ In SMC, we start with samples from the nominal trajectory distribution and gradually adapt these samples to move toward the failure distribution. We then use the path of each sample to estimate the probability of failure.

One way to adapt the samples in SMC is to move them through a sequence of intermediate distributions that gradually transition from the nominal distribution to the failure distribution. Specifically, we create a sequence of distributions g_1, g_2, \dots, g_n where g_1 is the nominal trajectory distribution and g_n is the failure distribution. Figure 7.9 illustrates two methods for selecting the intermediate

⁹ SMC is also known as particle filtering in the context of state estimation. M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A Tutorial on Particle Filters for Online Nonlinear/non-Gaussian Bayesian Tracking," *IEEE Transactions on Signal Processing*, vol. 50, no. 2, pp. 174–188, 2002.

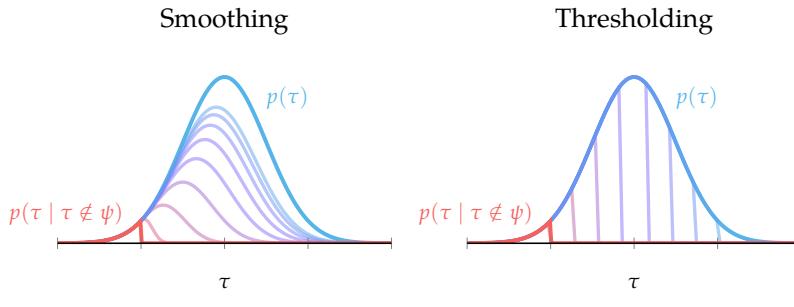


Figure 7.9. Two methods for selecting intermediate distributions for SMC. The distributions gradually transition from the nominal trajectory distribution $p(\tau)$ (blue) to the failure distribution $p(\tau \mid \tau \notin \psi)$ (red). The first method uses the smoothing technique introduced in section 6.3.2. The second method uses the thresholding technique introduced in section 7.3.1 with objective function $f(\tau) = \tau + 2$.

distributions. The first method uses the smoothing technique introduced in section 6.3.2. We can move from the nominal distribution to the failure distribution by gradually decreasing the value of the standard deviation ϵ in the smoothed density.¹⁰ The second method uses the same thresholding technique used in the cross entropy method. The intermediate distributions take the form $p(\tau \mid f(\tau) \leq \gamma)$ where $f(\tau)$ is the objective function and γ is a threshold. We move from the nominal distribution to the failure distribution by gradually decreasing the value of γ .

At each iteration of SMC, our goal is to transition samples from the current distribution g_ℓ to the next distribution in the sequence $g_{\ell+1}$. We typically only have access to the unnormalized densities of the intermediate distributions in practice, so MCMC is commonly used to perform this transition. Specifically, we initialize an MCMC chain at each sample with $g_{\ell+1}$ as the target distribution and run the chain for a fixed number of iterations. We take the final sample in each chain to form the next set of samples. Figure 7.10 demonstrates this process.

To produce an estimate of the probability of failure from the MCMC samples, we derive a set of importance weights using the joint probability distribution over the path of each sample as the proposal distribution.¹¹ The importance weight of the i th trajectory after sampling from the distribution g_ℓ is given by

$$w_i^{(\ell)} = w_i^{(\ell-1)} \frac{\bar{g}_{\ell+1}(\tau_i^{(\ell)})}{\bar{g}_\ell(\tau_i^{(\ell)})} \quad (7.21)$$

where $\bar{g}_\ell(\tau)$ is equal to $p(\tau)$ when $\ell = 1$ and the unnormalized density of the ℓ th intermediate distribution otherwise. We can obtain an estimate for the probability

¹⁰ A similar technique is the exponential tilting barrier presented in A. Sinha, M. O’Kelly, R. Tedrake, and J.C. Duchi, “Neural Bridge Sampling for Evaluating Safety-Critical Autonomous Systems,” *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 6402–6416, 2020.

¹¹ A full derivation can be found in F. Llorente, L. Martino, D. Delgado, and J. Lopez-Santiago, “Marginal Likelihood Computation for Model Selection and Hypothesis Testing: an Extensive Review,” *SIAM Review*, vol. 65, no. 1, pp. 3–58, 2023.

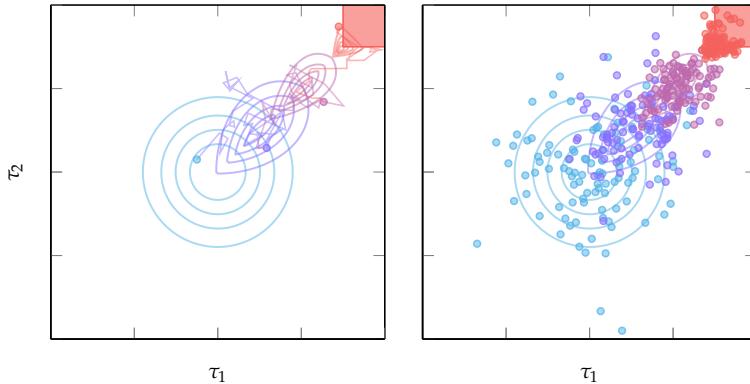


Figure 7.10. Adaptation steps in SMC. The plot on the left shows the MCMC paths of a single sample as it transitions from the nominal distribution (blue) to the failure distribution through a set of smoothed intermediate distributions. The plot on the right shows this process on a set of samples initially drawn from the nominal distribution.

of failure using the mean of the weights at the final iteration:

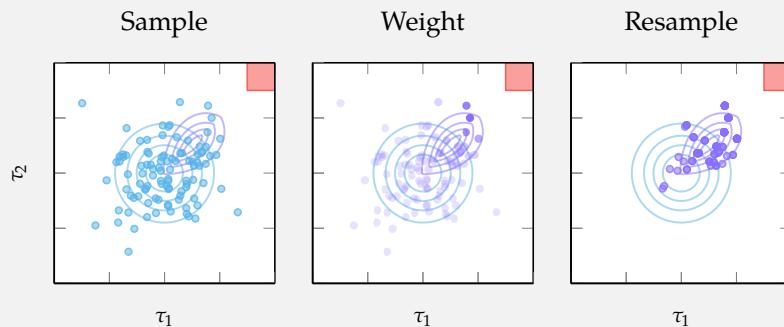
$$\hat{p}_{\text{fail}} = \frac{1}{m} \sum_{i=1}^m w_i^{(n-1)} \quad (7.22)$$

The accuracy of the estimator in equation (7.22) depends on how well the samples at each iteration represent the corresponding intermediate distribution. If the samples are not representative, the weights will be small, and the estimator will be inaccurate. However, we may require a large number of MCMC steps to transition samples from one distribution to the next, especially for samples that are unlikely under the next distribution. One technique used to address this challenge is to resample the trajectories based on their importance weights.¹² This step is similar to the resampling step in PMC and tends to result in better coverage of the intermediate distributions (see example 7.6). After resampling, we reset the weights to the mean of the weights before resampling to ensure that the estimator in equation (7.22) remains accurate.

Algorithm 7.7 implements SMC given a nominal trajectory distribution and a set of intermediate distributions. At each iteration, it perturbs the current set of samples to represent the next distribution in the sequence using MCMC. Example 7.7 provides an implementation of this step for the inverted pendulum problem. The algorithm then updates the importance weights and performs the resampling step. Finally, it returns an estimate of the probability of failure based on equation (7.22).

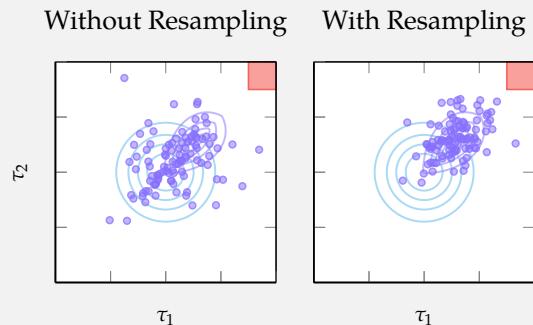
¹² P. Del Moral, A. Doucet, and A. Jasra, "Sequential Monte Carlo Samplers," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 68, no. 3, pp. 411–436, 2006.

Consider the scenario shown below in which we want to transition samples from the blue distribution to the purple distribution using 10 MCMC steps per sample. The plots below illustrate the weighting and resampling steps. The plot in the middle shows the weights of the samples, with darker points having higher weights. The plot on the right shows the samples after resampling according to these weights. After resampling, the samples are more representative of the purple distribution.



Example 7.6. The benefit of resampling in SMC. The first set of plots illustrates the resampling step, and the second set of plots shows the improvement in the samples at the next iteration after performing resampling.

On the next iteration, we perform MCMC starting at these samples with the purple distribution as the target to complete the transition. The plots below show the result of this step with and without resampling. The results without resampling start the MCMC chains at the blue samples shown above. The resampling step results in a set of samples that better represents the target distribution.



```

struct SequentialMonteCarloEstimation
    p      # nominal trajectory distribution
    ġs     # intermediate distributions
    perturb # ts' = perturb(ts, ġ)
    m      # number of samples
end

function estimate(alg::SequentialMonteCarloEstimation, sys, ψ)
    p, ġs, perturb, m = alg.p, alg.ǵs, alg.perturb, alg.m
    āfailure(τ) = isfailure(ψ, τ) * pdf(p, τ)
    ts = [rollout(sys, p) for i in 1:m]
    ws = [ǵs[1](τ) / p(τ) for τ in ts]
    for (ḡ, ḡ') in zip(ǵs, [ǵs[2:end]...; āfailure])
        ts' = perturb(ts, ḡ)
        ws .*= [ḡ'(τ) / ḡ(τ) for τ in ts']
        ts = ts'[rand(Categorical(ws ./ sum(ws)), m)]
        ws .= mean(ws)
    end
    return mean(ws)
end

```

Algorithm 7.7. The sequential Monte Carlo algorithm for estimating the probability of failure. The algorithm iterates through intermediate distributions and perturbs the samples to represent the current distribution at each iteration. It then computes the weights according to the next distribution in the sequence, performs the resampling step, and resets the weights. The algorithm uses a system specific `perturb` function to transition samples from one distribution to the next. It returns an estimate of the probability of failure based on equation (7.22).

Unlike the algorithms presented in section 7.3, algorithm 7.7 is *nonparametric*. We do not need to specify a parametric form for the intermediate distributions. Instead, we represent them using samples. This flexibility allows us to estimate the probability of failure for complex, multimodal failure distributions. However, SMC can run into the same potential problems as PMC. For example, if the MCMC is not run for long enough on each iteration, the samples may be inaccurate and miss potential failure modes. The resampling step can also cause a loss of diversity that may result in a collapse of the samples to a single mode. Other weighting and resampling schemes may be used to maintain diversity.¹³

7.5 Ratio of Normalizing Constants

Importance sampling is a special case of the more general problem of estimating the ratio of the normalizing constants of two distributions.¹⁴ By focusing on the more general problem, we can derive multiple extensions to importance sampling that allow us to use unnormalized proposal densities. Consider two probability distributions g_1 and g_2 with normalizing constants z_1 and z_2 such that $g_1(\tau) = \bar{g}_1(\tau)/z_1$ and $g_2(\tau) = \bar{g}_2(\tau)/z_2$. We have that $z_1 = \int \bar{g}_1(\tau) d\tau$ and

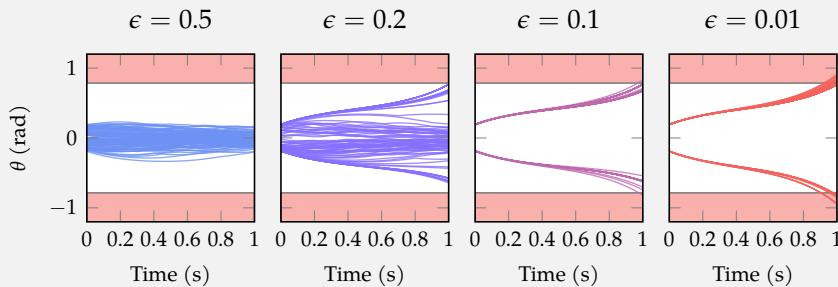
¹³ One common resampling scheme that ensures that the samples remain diverse is called low variance resampling. More details can be found in Section 4.2.4 of S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT Press, 2006.

¹⁴ A detailed survey of techniques relating the estimating the ratio of normalizing constants is provided in F. Llorente, L. Martino, D. Delgado, and J. Lopez-Santiago, “Marginal Likelihood Computation for Model Selection and Hypothesis Testing: an Extensive Review,” *SIAM Review*, vol. 65, no. 1, pp. 3–58, 2023.

To estimate the probability of failure for the inverted pendulum system using SMC, we implement the following function that uses 10 MCMC steps to transition samples between intermediate distributions:

```
function perturb(samples, g)
    function inverted_pendulum_kernel(t; Σ=0.05^2 * I)
        μs, Σs = [step.x_0 for step in t], [Σ for step in t]
        return PendulumTrajectoryDistribution(t[1].s, Σ, μs, Σs)
    end
    k_max, m_burnin, m_skip, new_samples = 10, 1, 1, []
    for sample in samples
        alg = MCMCSampling(g, inverted_pendulum_kernel, sample,
                            k_max, m_burnin, m_skip)
        mcmc_samples = sample_failures(alg, inverted_pendulum, ψ)
        push!(new_samples, mcmc_samples[end])
    end
    return new_samples
end
```

We can use a set of smoothed failure distributions as the intermediate distributions. The plot below shows some of the samples from these intermediate distributions.



Using 1,000 samples per iteration, SMC estimates the probability of failure to be approximately 0.0001. The direct estimate for probability of failure based on one million simulations is approximately 0.0005.

Example 7.7. Application of SMC to the inverted pendulum problem. The plots show the samples from the intermediate smoothed failure distributions.

$z_2 = \int \bar{g}_2(\tau) d\tau$, and our goal is to estimate the ratio of the normalizing constants z_1/z_2 using samples from g_2 .

First, we rewrite z_1 in terms of an expectation over g_2 :

$$z_1 = \int \bar{g}_1(\tau) d\tau \quad (7.23)$$

$$= \int \bar{g}_1(\tau) \frac{g_2(\tau)}{\bar{g}_2(\tau)} d\tau \quad (7.24)$$

$$= \int g_2(\tau) \frac{\bar{g}_1(\tau)}{\bar{g}_2(\tau)/z_2} d\tau \quad (7.25)$$

$$= z_2 \int g_2(\tau) \frac{\bar{g}_1(\tau)}{\bar{g}_2(\tau)} d\tau \quad (7.26)$$

$$= z_2 \mathbb{E}_{\tau \sim g_2(\cdot)} \left[\frac{\bar{g}_1(\tau)}{\bar{g}_2(\tau)} \right] \quad (7.27)$$

Dividing both sides of equation (7.27) by z_2 gives us the ratio of the normalizing constants, which we can approximate using m samples from g_2 :

$$\frac{z_1}{z_2} = \mathbb{E}_{\tau \sim g_2(\cdot)} \left[\frac{\bar{g}_1(\tau)}{\bar{g}_2(\tau)} \right] \approx \frac{1}{m} \sum_{i=1}^m \frac{\bar{g}_1(\tau_i)}{\bar{g}_2(\tau_i)} \quad (7.28)$$

where $\tau_i \sim g_2(\cdot)$ and $g_2(\tau) > 0$ whenever $g_1(\tau) > 0$. Note that the estimator in equation (7.28) only requires evaluating the unnormalized densities $\bar{g}_1(\tau)$ and $\bar{g}_2(\tau)$. Since p_{fail} is the normalizing constant of the failure distribution, we can use equation (7.28) to estimate the probability of failure by setting $\bar{g}_1(\tau)$ equal to the unnormalized failure density and $\bar{g}_2(\tau)$ equal to any normalized proposal density $q(\tau)$. In fact, these choices of $\bar{g}_1(\tau)$ and $\bar{g}_2(\tau)$ cause equation (7.28) to reduce to the importance sampling estimator in equation (7.12) (see exercises 7.12 and 7.13).¹⁵

If g_1 and g_2 have little overlap in terms of probability mass, the estimator in equation (7.28) may perform poorly. One technique to improve performance is called *umbrella sampling* (also known as *ratio importance sampling*). Umbrella sampling introduces a third density, called an umbrella density, that has significant overlap with both g_1 and g_2 . We use this density to estimate the ratio of normalizing constants by applying equation (7.28) twice:

$$\frac{z_1}{z_2} = \frac{z_1/z_u}{z_2/z_u} = \frac{\mathbb{E}_{\tau \sim g_u(\cdot)} \left[\frac{\bar{g}_1(\tau)}{\bar{g}_u(\tau)} \right]}{\mathbb{E}_{\tau \sim g_u(\cdot)} \left[\frac{\bar{g}_2(\tau)}{\bar{g}_u(\tau)} \right]} \approx \frac{\frac{1}{m} \sum_{i=1}^m \frac{\bar{g}_1(\tau_i)}{\bar{g}_u(\tau_i)}}{\frac{1}{m} \sum_{i=1}^m \frac{\bar{g}_2(\tau_i)}{\bar{g}_u(\tau_i)}} \quad (7.29)$$

¹⁵We could also use equation (7.28) to estimate the reciprocal of the probability of failure using samples from the failure distribution by setting $\bar{g}_2(\tau)$ equal to the unnormalized failure density and $\bar{g}_1(\tau)$ equal to any normalized density whose support is contained within the support of failure distribution. However, selecting $\bar{g}_1(\tau)$ to satisfy this condition is often difficult in practice and can lead to estimators with infinite variance. This technique is called reciprocal importance sampling. In general, this estimator should not be used for failure probability estimation.

where \bar{g}_u is the unnormalized umbrella density, z_u is its normalizing constant, and the m samples are drawn from $g_u(\cdot)$. The optimal umbrella density is

$$\bar{g}_u^*(\tau) \propto \left| \bar{g}_1(\tau) - \frac{z_1}{z_2} \bar{g}_2(\tau) \right| \quad (7.30)$$

Similar to the optimal proposal for importance sampling, the optimal umbrella density is expressed in terms of the quantity we are trying to estimate, so we cannot compute it exactly. In general, we want to select an umbrella density that is as close as possible to this density.

Another technique to estimate the ratio of normalizing constants when g_1 and g_2 have little overlap is called *bridge sampling*. Similar to umbrella sampling, bridge sampling introduces a third density called a *bridge density*. However, instead of using samples from this density to estimate the ratio of normalizing constants, bridge sampling uses samples from both g_1 and g_2 . Assuming we produce m_1 samples from g_1 and m_2 samples from g_2 , we again apply equation (7.28) twice to obtain the bridge sampling estimator:

$$\frac{z_1}{z_2} = \frac{z_b/z_2}{z_b/z_1} = \frac{\mathbb{E}_{\tau \sim g_2(\cdot)} \left[\frac{\bar{g}_b(\tau)}{\bar{g}_2(\tau)} \right]}{\mathbb{E}_{\tau \sim g_1(\cdot)} \left[\frac{\bar{g}_b(\tau)}{\bar{g}_1(\tau)} \right]} \approx \frac{\frac{1}{m_2} \sum_{j=1}^{m_2} \frac{\bar{g}_b(\tau_j)}{\bar{g}_2(\tau_j)}}{\frac{1}{m_1} \sum_{i=1}^{m_1} \frac{\bar{g}_b(\tau_i)}{\bar{g}_1(\tau_i)}} \quad (7.31)$$

where $\tau_i \sim g_1(\cdot)$, $\tau_j \sim g_2(\cdot)$, and \bar{g}_b is the bridge density.

The optimal bridge density is

$$\bar{g}_b^*(\tau) \propto \frac{\bar{g}_1(\tau) \bar{g}_2(\tau)}{m_1 \bar{g}_1(\tau) + m_2 \frac{z_1}{z_2} \bar{g}_2(\tau)} \quad (7.32)$$

which is again written in terms of the quantity we are trying to estimate. Given samples from both g_1 and g_2 , we can use a simple iterative procedure to estimate the optimal bridge density (algorithm 7.8). At each iteration, we apply equation (7.31) using the current bridge density to estimate the ratio of normalizing constants. We then plug this ratio into equation (7.32) to obtain a new bridge density. We repeat this process for a fixed number of iterations.

While umbrella sampling and bridge sampling both introduce a third density to improve efficiency, they have different properties. For example, umbrella sampling only requires samples from one density, while bridge sampling requires samples from two different densities. Furthermore, the optimal umbrella density and the optimal bridge density are very different (see figure 7.11). The optimal umbrella

```

function bridge_sampling_estimator(g1ts, ġ1, g2ts, ġ2, ġb)
    ġ1s, ġ2s = ġ1.(g1ts), ġ2.(g2ts)
    ġb1s, ġb2s = ġb.(g1ts), ġb.(g2ts)
    return mean(ġb2s ./ ġ2s) / mean(ġb1s ./ ġ1s)
end

function optimal_bridge(g1ts, ġ1, g2ts, ġ2, k_max)
    ratio = 1.0
    m1, m2 = length(g1ts), length(g2ts)
    ġb(τ) = (ġ1(τ) * ġ2(τ)) / (m1 * ġ1(τ) + m2 * ratio * ġ2(τ))
    for k in k_max
        ratio = bridge_sampling_estimator(g1ts, ġ1, g2ts, ġ2, ġb)
    end
    return ġb
end

```

Algorithm 7.8. Algorithm for estimating the optimal bridge density \bar{g}_b using samples from \bar{g}_1 and \bar{g}_2 . We iteratively apply equation (7.31) to estimate the ratio of normalizing constants and use this ratio to update the bridge density using equation (7.32).

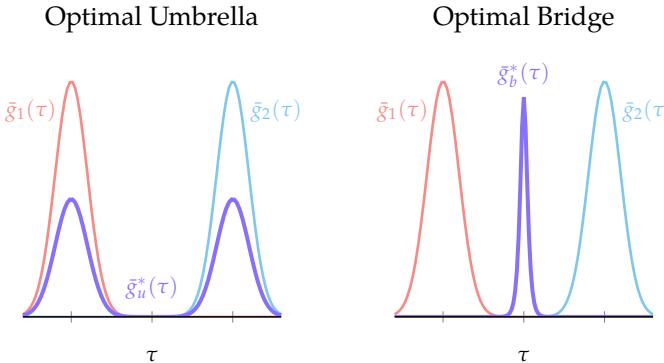


Figure 7.11. Comparison of the optimal umbrella density and optimal bridge density for estimating the ratio of normalizing constants between two example distributions.

density covers regions of high likelihood for both distributions, while the optimal bridge density bridges the gap between the two distributions.

7.5.1 Self-Normalized Importance Sampling

Self-normalized importance sampling (self-IS) is a special case of umbrella sampling that can be used to estimate the probability of failure given samples from an unnormalized density. Specifically, we set $\bar{g}_1(\tau)$ to be the unnormalized failure density, $\bar{g}_2(\tau)$ to be the nominal trajectory distribution, and $\bar{g}_u(\tau)$ to be an unnormalized

proposal density $\bar{q}(\tau)$. These choices lead to the following estimator:

$$\frac{p_{\text{fail}}}{1} \approx \frac{\frac{1}{m} \sum_{i=1}^m \frac{\mathbb{1}\{\tau_i \notin \psi\} p(\tau_i)}{\bar{q}(\tau_i)}}{\frac{1}{m} \sum_{i=1}^m \frac{p(\tau_i)}{\bar{q}(\tau_i)}} = \frac{\frac{1}{m} \sum_{i=1}^m w_i \mathbb{1}\{\tau_i \notin \psi\}}{\frac{1}{m} \sum_{i=1}^m w_i} \quad (7.33)$$

where $w_i = p(\tau_i)/\bar{q}(\tau_i)$ and $\tau_i \sim q(\cdot)$. This estimator (algorithm 7.9) is similar to the estimator in equation (7.13) for normalized proposal distributions with the extra step of dividing the unnormalized importance weights by their sum.

```

struct SelfImportanceSamplingEstimation
    p      # nominal distribution
    q      # unnormalized proposal density
    q_ts # samples from q
end

function estimate(alg::SelfImportanceSamplingEstimation, sys, ψ)
    p, q, q_ts = alg.p, alg.q, alg.q_ts
    ws = [pdf(p, τ) / q(τ) for τ in q_ts]
    ws ./= sum(ws)
    return mean(w * isfailure(ψ, τ) for (w, τ) in zip(ws, q_ts))
end

```

Algorithm 7.9. The self-normalized importance sampling estimation algorithm for estimating the probability of failure. The algorithm takes as input an unnormalized proposal density \bar{q} along with samples drawn from it. It computes the importance weights for the samples and applies equation (7.33) to compute \hat{p}_{fail} .

The optimal proposal for self-IS is different from the optimal proposal for importance sampling. Based on equation (7.30), the optimal proposal for self-IS is

$$q^*(\tau) \propto p(\tau) |\mathbb{1}\{\tau \notin \psi\} - p_{\text{fail}}| \quad (7.34)$$

Sampling from this density should result in half of the samples coming from the failure distribution and half coming from the success distribution. The optimal proposal for IS, on the other hand, is the failure distribution itself. Figure 7.12 shows the optimal proposal distribution for self-IS on a simple Gaussian system.

In practice, we can plug a guess for p_{fail} into equation (7.34) to obtain a proposal distribution that is close to the optimal proposal. However, drawing samples from this proposal is often difficult in practice, especially for systems with rare failure events and multiple failure modes (see example 7.8). Furthermore, the performance of the algorithm tends to be sensitive to incorrect guesses for p_{fail} when creating the proposal distribution. Bridge sampling, which we discuss in the next section, is less sensitive to these choices.

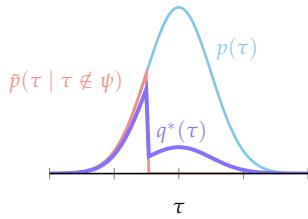
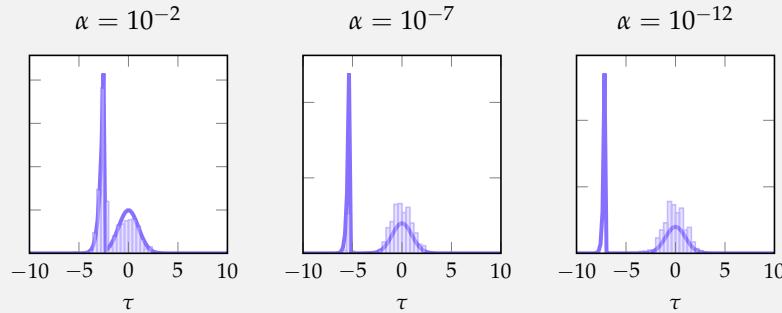


Figure 7.12. The optimal proposal for self-normalized importance sampling for a simple Gaussian problem with a failure threshold of -1 .

Suppose we want to use self-IS to estimate the probability of failure for the simple Gaussian system. We know that the optimal proposal is of the form

$$q^*(\tau) \propto p(\tau) |\mathbb{1}\{\tau \notin \psi\} - \alpha|$$

where α is our guess for the probability of failure. The plots below show the proposal distribution for three different values of α along with histograms of samples drawn from these distributions using MCMC. For each distribution, we use 5,100 MCMC steps with a burn-in of 100 steps, keeping every 10th sample.



As α decreases, the modes of the proposal distribution grow further apart, and the proposal distribution becomes more difficult to accurately sample from. For $\alpha = 10^{-12}$, the MCMC misses the failure region entirely.

Example 7.8. The challenges associated with sampling from the optimal self-IS proposal density. The plots show the proposal distribution for three different failure probabilities along with histograms of samples drawn from these distributions using MCMC. As the probability of failure decreases, the distributions become more difficult to accurately sample from.

7.5.2 Bridge Sampling

We can use the bridge sampling estimator in equation (7.31) to estimate the probability of failure, but it will be inefficient for systems with low failure probabilities. In fact, if we follow the same steps we followed to derive the self-IS estimator, we will arrive at an estimator that can perform no better than the direct estimator in section 7.1. This property of the bridge sampling estimator is a result of the optimal bridge density being zero for all samples that are not failures (see example 7.9).

To improve performance, we can build upon ideas from SMC (section 7.4) by performing bridge sampling on a sequence of distributions that gradually transition from the nominal distribution to the failure distribution.¹⁶ Specifically, we create a sequence of n distributions and represent the ℓ th distribution as $g_\ell(\tau) = \bar{g}_\ell(\tau)/z_\ell$. We set $\bar{g}_1(\tau)$ equal to the density of the nominal trajectory distribution and $\bar{g}_n(\tau)$ equal to the unnormalized failure density.

We then rewrite the probability of failure as a product of ratios of normalizing constants:

$$\frac{p_{\text{fail}}}{1} = \frac{z_n}{z_1} = \left(\frac{z_2}{z_1} \right) \left(\frac{z_3}{z_2} \right) \cdots \left(\frac{z_L}{z_{L-1}} \right) = \prod_{\ell=1}^{L-1} \frac{z_{\ell+1}}{z_\ell} \quad (7.35)$$

These ratios can be estimated using the bridge sampling identity in equation (7.31):

$$p_{\text{fail}} \approx \prod_{\ell=1}^{n-1} \frac{\frac{1}{m_2} \sum_{j=1}^{m_2} \frac{\bar{g}_{b,\ell}(\tau_j)}{\bar{g}_{\ell}(\tau_j)}}{\frac{1}{m_1} \sum_{i=1}^{m_1} \frac{\bar{g}_{b,\ell}(\tau_i)}{\bar{g}_{\ell+1}(\tau_i)}} \quad (7.36)$$

where we draw m_1 samples from $g_{\ell+1}(\cdot)$ and m_2 samples from $g_\ell(\cdot)$. The intermediate distributions in the chain should be chosen such that the ratio of normalizing constants between two consecutive distributions is easy to estimate. In other words, consecutive intermediate distributions should have significant overlap with each other. For example, we can create the intermediate distributions using either of the two methods in figure 7.9.¹⁷

Algorithm 7.10 implements bridge sampling estimation using a sequence of intermediate distributions. It begins by drawing samples from the nominal trajectory distribution. At each iteration, it perturbs the samples to match the next intermediate distribution and estimates the optimal bridge density using algorithm 7.8. It then applies equation (7.36) to compute the ratio of normalizing constants

¹⁶ A. Sinha, M. O'Kelly, R. Tedrake, and J.C. Duchi, "Neural Bridge Sampling for Evaluating Safety-Critical Autonomous Systems," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 6402–6416, 2020.

¹⁷ If we use the thresholding technique, the algorithm reduces to the multilevel splitting algorithm presented in section 7.6.

To estimate the probability of failure from equation (7.31), we set $\bar{g}_1(\tau)$ equal to the unnormalized failure density and $\bar{g}_2(\tau)$ equal to the density of the nominal trajectory distribution:

$$\frac{p_{\text{fail}}}{1} \approx \frac{\frac{1}{m_2} \sum_{j=1}^{m_2} \frac{\bar{g}_b(\tau_j)}{p(\tau_j)}}{\frac{1}{m_1} \sum_{i=1}^{m_1} \frac{\bar{g}_b(\tau_i)}{\mathbb{1}\{\tau_i \notin \psi\} p(\tau_i)}}$$

Plugging in to equation (7.32) to get the optimal bridge density gives:

$$g_b^*(\tau) \propto \frac{\mathbb{1}\{\tau \notin \psi\} p(\tau)^2}{m_1 \mathbb{1}\{\tau \notin \psi\} p(\tau) + m_2 p(\tau)} \propto \begin{cases} p(\tau) & \text{if } \tau \notin \psi \\ 0 & \text{otherwise} \end{cases}$$

The optimal bridge density is zero for all samples that are not failures. Since all the samples from the failure density will be failure samples, we have

$$\frac{1}{m_1} \sum_{i=1}^{m_1} \frac{\bar{g}_b(\tau_i)}{\mathbb{1}\{\tau_i \notin \psi\} p(\tau_i)} = \frac{1}{m_1} \sum_{i=1}^{m_1} \frac{p(\tau_i)}{p(\tau_i)} = 1$$

We also have that

$$\frac{1}{m_2} \sum_{j=1}^{m_2} \frac{p(\tau_j)}{\mathbb{1}\{\tau_j \notin \psi\} p(\tau_j)} = \frac{n_2}{m_2}$$

where n_2 is the number of samples from the nominal trajectory distribution that were failures. In this case, the bridge sampling estimator reduces to the direct estimator in equation (7.2). Since we produced this result using the optimal bridge density, we can conclude that this estimator will not perform any better than the direct estimator for estimating the probability of failure.

Example 7.9. Proof that a bridge sampling estimator that sets $\bar{g}_1(\tau)$ to the unnormalized failure density and $\bar{g}_2(\tau)$ to the nominal trajectory distribution can perform no better than the direct estimator for estimating the probability of failure.

between the two distributions. Finally, the algorithm applies equation (7.35) to compute an estimate for the probability of failure.

```

struct BridgeSamplingEstimation
    p      # nominal trajectory distribution
    ġs     # intermediate distributions
    perturb # samples' = perturb(samples, ġ')
    m      # number of samples from each intermediate distribution
    kb     # number of iterations for estimating optimal bridge
end

function estimate(alg::BridgeSamplingEstimation, sys, ψ)
    p, ġs, perturb, m = alg.p, alg.ġs, alg.perturb, alg.m
    īfailure(τ) = isfailure(ψ, τ) * pdf(p, τ)
    ts = [rollout(sys, p) for i in 1:m]
    īfail = 1.0
    for (ġ, ġ') in zip([p; ġs...], [ġs...; īfailure])
        ws = [ġ'(τ) / ġ(τ) for τ in ts]
        ts' = ts[rand(Categorical(ws ./ sum(ws)), m)]
        ts' = perturb(ts', ġ')
        ġb = optimal_bridge(ts', ġ', ts, ġ, kb)
        ratio = bridge_sampling_estimator(ts', ġ', ts, ġ, ġb)
        īfail *= ratio
        ts = ts'
    end
    return īfail
end

```

Algorithm 7.10. The bridge sampling algorithm for estimating the probability of failure. The algorithm takes as input a nominal trajectory distribution p and a sequence of intermediate densities \bar{g}_s . At each iteration, the algorithm performs resampling and uses the system-specific `perturb` function to produce samples from the next distribution. It then estimates the optimal bridge density and applies equation (7.31) from algorithm 7.8 to compute the ratio of normalizing constants. The final iteration draws samples from the failure distribution and produces an estimate of the failure probability.

The perturb step in algorithm 7.10 produces samples from the next distribution in the sequence and can be performed using the MCMC algorithms presented in chapter 6. However, these distributions may be difficult to sample from, especially as we get closer to the failure distribution. In practice, we can greatly increase efficiency by using the samples from the previous distribution as a starting point to produce samples from the next distribution. This process is similar to the process used in SMC (algorithm 7.7), in which we weight and resample the trajectories from the previous distribution before applying MCMC. The ability to adapt samples from the previous distribution is another benefit of using a sequence of intermediate distributions. Figure 7.13 shows the samples from the intermediate distributions for the continuum world problem.

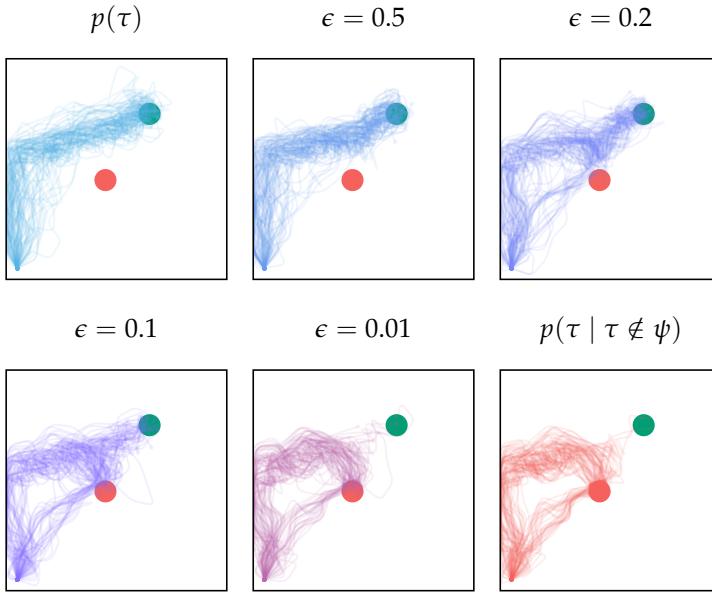


Figure 7.13. Samples from the smoothed intermediate distributions when applying bridge sampling to estimate the probability of failure for the continuum world problem. These samples result in a failure probability estimate of 4.93×10^{-4} . The direct estimate from one million samples is 2.47×10^{-4} .

7.6 Multilevel Splitting

Multilevel splitting algorithms estimate the probability of failure using a series of conditional distributions.¹⁸ Similar to the cross entropy method (section 7.3.1), multilevel splitting relies on an objective function f with properties such that the probability of failure can be written as $p(f(\tau) \leq 0)$. Given a series of thresholds $\gamma_1 > \gamma_2 > \dots > \gamma_n$ where $\gamma_1 = \infty$ and $\gamma_n = 0$, we can write the probability of failure as

$$p_{\text{fail}} = p(f(\tau) \leq \gamma_n) = \prod_{\ell=2}^n p(f(\tau) \leq \gamma_\ell \mid f(\tau) \leq \gamma_{\ell-1}) \quad (7.37)$$

As long as the thresholds gradually decrease in a way that ensures that the conditional probabilities remain large, we can efficiently estimate these intermediate probabilities using direct estimation.

To ensure that the conditional probabilities remain large, it is common to select the thresholds adaptively.¹⁹ Algorithm 7.11 implements adaptive multilevel splitting. Adaptive multilevel splitting begins by drawing samples from the nominal trajectory distribution. At each iteration, it computes the objective value for

¹⁸ H. Kahn and T. E. Harris, "Estimation of Particle Transmission by Random Sampling," *National Bureau of Standards Applied Mathematics Series*, vol. 12, pp. 27–30, 1951.

¹⁹ F. Cérou and A. Guyader, "Adaptive Multilevel Splitting for Rare Event Analysis," *Stochastic Analysis and Applications*, vol. 25, no. 2, pp. 417–443, 2007.

```

struct AdaptiveMultilevelSplitting
    p      # nominal trajectory distribution
    m      # number of samples
    m_elite # number of elite samples
    k_max  # maximum number of iterations
    f      # objective function f( $\tau$ ,  $\psi$ )
    perturb #  $\tau_s' = \text{perturb}(\tau_s, \bar{p}\gamma)$ 
end

function estimate(alg::AdaptiveMultilevelSplitting, sys,  $\psi$ )
    p, m, m_elite, k_max = alg.p, alg.m, alg.m_elite, alg.k_max
    f, perturb = alg.f, alg.perturb
    ts = [rollout(sys, p) for i in 1:m]
     $\hat{p}_{\text{fail}} = 1.0$ 
    for i in 1:k_max
        Y = [f( $\tau$ ,  $\psi$ ) for  $\tau$  in ts]
        order = sortperm(Y)
        y = i == k_max ? 0 : max(0, Y[order[m_elite]])
         $\hat{p}_{\text{fail}} *= \text{mean}(Y \leq y)$ 
        y == 0 && break
        ts = rand(ts[order[1:m_elite]], m)
         $\bar{p}\gamma(\tau) = p(\tau) * (f(\tau, \psi) \leq y)$ 
        ts = perturb(ts,  $\bar{p}\gamma$ )
    end
    return  $\hat{p}_{\text{fail}}$ 
end

```

Algorithm 7.11. The adaptive multilevel splitting algorithm for estimating the probability of failure. At each iteration, the algorithm computes the objective value for each sample and selects the next threshold. It then estimates the current conditional probability and perturbs the samples to represent the next distribution. The `perturb` function is system specific. The algorithm iterates until the threshold reaches zero. If we reach the maximum number of iterations before this criterion is met, the algorithm will force the final threshold to be zero.

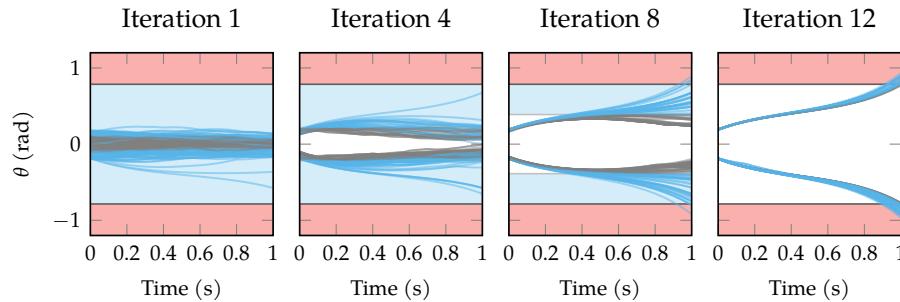


Figure 7.14. Adaptive multilevel splitting applied to the inverted pendulum system. The shaded blue region represents the region where the objective function is less than the current threshold. The samples gradually transition from the nominal trajectory distribution to the failure distribution as the algorithm progresses. The algorithm terminates on iteration 12 when all elite samples are failures.

each sample and selects a threshold γ such that a fixed number of samples have objective values less than γ . It then uses this threshold and the current samples to estimate $p(f(\tau) \leq \gamma_\ell | f(\tau) \leq \gamma_{\ell-1})$.

The algorithm produces the next set of samples by perturbing the current samples to represent the distribution $p(\tau | f(\tau) \leq \gamma_\ell)$. As with SMC and bridge sampling, this step can be performed using the MCMC algorithms presented in chapter 6. To improve the efficiency of the MCMC, we first resample by drawing m samples uniformly from the elite samples.

To accurately estimate the probability of failure, the last iteration of the algorithm must use a threshold of zero. Algorithm 7.11 iterates until the threshold reaches zero, at which point all elite samples are failures. If we reach the maximum number of iterations before this criterion is met, the algorithm will force the final threshold to be zero. However, if there are no failure samples in the final iteration, the final conditional probability will be zero, causing the algorithm to return an estimate of zero. Therefore, it is important to ensure that we allow enough iterations for the algorithm to reach the final threshold.

Multilevel splitting is considered a nonparametric algorithm in that we estimate the probability of failure without assuming a specific form for the conditional distributions. This feature allows multilevel splitting to extend to systems with complex, multimodal failure distributions. Furthermore, the adaptive nature of algorithm 7.11 allows us to smoothly transition from the nominal trajectory distribution to the failure distribution without specifying the intermediate distributions ahead of time. Figure 7.14 shows an example of adaptive multilevel splitting applied to the inverted pendulum system, which has two modes in its failure distribution.

7.7 Summary

- We can view the problem of estimating the probability of failure as a parameter estimation problem and apply maximum likelihood or Bayesian methods.
- For systems with rare failure events, we can use importance sampling to estimate the probability of failure by sampling from a proposal distribution that assigns higher likelihood to failure trajectories.
- The performance of importance sampling algorithms depends on the choice of proposal distribution, and we want to select a proposal distribution that is as close as possible to the optimal proposal distribution, which is the failure distribution.
- We use adaptive importance sampling algorithms to automatically tune a proposal or set of proposals based on samples.
- Sequential Monte Carlo is a nonparametric algorithm that can be applied to estimate the probability of failure for complex, multimodal failure distributions.
- By viewing the failure probability estimation problem as a special case of a more general problem of estimating ratios of normalizing constants, we can derive estimators that allow us to use complex proposal distributions for which we only know the unnormalized density.
- Umbrella sampling and bridge sampling increase efficiency by introducing a third density into the ratio of normalizing constants.
- Multilevel splitting is a nonparametric algorithm that estimates the probability of failure using a series of conditional distributions.

7.8 Exercises

Exercise 7.1. Show that equation (7.2) is an unbiased estimator of the probability of failure.

Solution: To show that the estimator is unbiased, we must show $\mathbb{E}[\hat{p}_{\text{fail}}] = p_{\text{fail}}$:

$$\begin{aligned}\mathbb{E}[\hat{p}_{\text{fail}}] &= \mathbb{E}\left[\frac{1}{m} \sum_{i=1}^m \mathbb{1}\{\tau_i \notin \psi\}\right] \\ &= \frac{1}{m} \sum_{i=1}^m \mathbb{E}[\mathbb{1}\{\tau_i \notin \psi\}] \\ &= \frac{1}{m} \sum_{i=1}^m p_{\text{fail}} \\ &= p_{\text{fail}}\end{aligned}$$

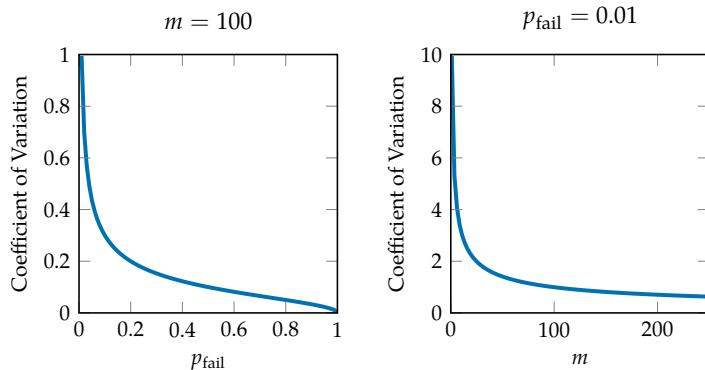
The second to last step follows from the fact that $\mathbb{1}\{\tau_i \notin \psi\}$ is a Bernoulli random variable with expectation p_{fail} .

Exercise 7.2. The *coefficient of variation* of a random variable is defined as the ratio of the standard deviation to the mean and is a measure of relative variability. Compute the coefficient of variation for the estimator in equation (7.2). For a fixed sample size m , how does the coefficient of variation change as p_{fail} increases? For a fixed p_{fail} , how does the coefficient of variation change as the sample size m increases?

Solution: The coefficient of variation for the estimator in equation (7.2) is

$$\begin{aligned}\frac{\text{standard deviation}}{\text{mean}} &= \frac{\sqrt{\frac{p_{\text{fail}}(1-p_{\text{fail}})}{m}}}{p_{\text{fail}}} \\ &= \sqrt{\frac{1-p_{\text{fail}}}{mp_{\text{fail}}}}\end{aligned}$$

For a fixed m , the coefficient of variation will decrease as the true probability of failure p_{fail} increases. For a fixed p_{fail} , the coefficient of variation will decrease as the sample size m increases. The plots below show an example of these relationships.



Exercise 7.3. Suppose we have a system with a requirement that the probability of failure must not exceed 0.01. We sample m trajectories of the system and observe no failures. We can calculate our confidence that the probability of failure is less than 0.01 using the Bayesian estimation techniques in section 7.1.2. Another way to compute this confidence is to use *Hoeffding's inequality*,²⁰ which states that

$$P(p_{\text{fail}} < \delta) \geq 1 - e^{-2m\delta^2}$$

where δ represents the target probability of failure.²¹ Another common bound is the *Clopper-Pearson bound*,²² which states that

$$P(p_{\text{fail}} < \delta) \geq \frac{1}{m} \log \frac{1}{\delta}$$

Plot the 95% confidence bound on the true probability of failure given by Bayesian estimation (assume a uniform prior) for values of m between 1 and 100 and compare it to the 95% confidence bound computed using Hoeffding's inequality and the Clopper-Pearson bound. How do the confidence bounds change as m increases?

Solution: A 95% confidence bound means we want to compute the value of δ such that $P(p_{\text{fail}} < \delta)$ is at least 0.95. Plugging this quantity into the equation for Hoeffding's inequality provided in the problem and rearranging gives

$$\delta = \sqrt{\frac{-\log(1 - 0.95)}{2m}}$$

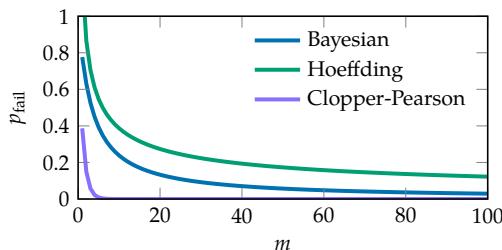
Plugging this quantity into the equation for the Clopper-Pearson bound provided in the problem and rearranging gives

$$\delta = \exp(-0.95m)$$

We can then compute the bounds using the following code:

```
bayesian_bounds = [quantile(Beta(1, 1+m), 0.95) for m in 1:100]
hoeffding_bounds = [sqrt(-log(0.05) / (2m)) for m in 1:100]
clopper_pearson_bounds = [exp(-0.95m) for m in 1:100]
```

The plot shows the results.



²⁰ W. Hoeffding, "Probability Inequalities for Sums of Bounded Random Variables," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 13–30, 1963.

²¹ This equation represents a specific instantiation of Hoeffding's inequality that assumes the samples are drawn independently from Bernoulli distributions and no failures are observed

²² C. J. Clopper and E. S. Pearson, "The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial," *Biometrika*, vol. 26, no. 4, pp. 404–413, 1934.

All three bounds decrease as m increases.

Exercise 7.4. Suppose we are using Bayesian estimation with a uniform prior to bound the probability of collision for an aircraft collision avoidance system. We observe 10,000 consecutive aircraft encounters without a collision. Compute the value α for which we are 99% confident that the true probability of failure is less than α .

Solution: The posterior distribution over the probability of collision is Beta(1, 10001). We can obtain a 99% confidence bound by taking the 0.99-quantile of the posterior. The 0.99-quantile of the Beta(1, 10001) distribution is 0.00046.

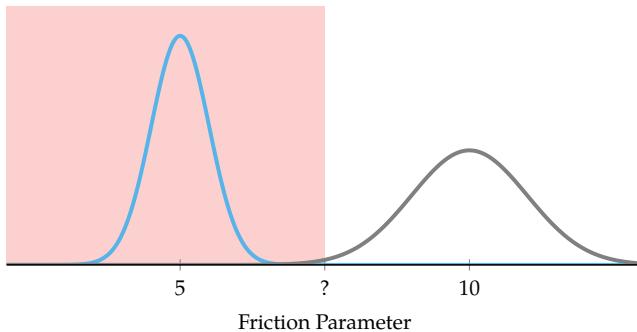
Exercise 7.5. Show that the estimator in equation (7.12) is an unbiased estimator of the probability of failure.

Solution: To show that the estimator is unbiased, we must show that $\mathbb{E}[\hat{p}_{\text{fail}}] = p_{\text{fail}}$:

$$\begin{aligned}\mathbb{E}[\hat{p}_{\text{fail}}] &= \mathbb{E}\left[\frac{1}{m} \sum_{i=1}^m \frac{p(\tau_i)}{q(\tau_i)} \mathbb{1}\{\tau_i \notin \psi\}\right] \\ &= \frac{1}{m} \sum_{i=1}^m \mathbb{E}\left[\frac{p(\tau_i)}{q(\tau_i)} \mathbb{1}\{\tau_i \notin \psi\}\right] \\ &= \frac{1}{m} \sum_{i=1}^m \int q(\tau_i) \frac{p(\tau_i)}{q(\tau_i)} \mathbb{1}\{\tau_i \notin \psi\} d\tau_i \\ &= \frac{1}{m} \sum_{i=1}^m \int p(\tau_i) \mathbb{1}\{\tau_i \notin \psi\} d\tau_i \\ &= \frac{1}{m} \sum_{i=1}^m p_{\text{fail}} \\ &= p_{\text{fail}}\end{aligned}$$

Exercise 7.6. We want to estimate the probability of failure for a braking system on a vehicle. We have a simulator that takes in a parameter that controls the amount of friction on the road (higher values indicate more friction) and outputs the distance the vehicle travels before coming to a stop. A lower friction parameter will result in a larger stopping distance. The specification for the system requires that the vehicle must come to a stop within 10 m. We know the nominal distribution over the friction parameter is a Gaussian distribution with a mean of 5 and a standard deviation of 1. We perform importance sampling with a proposal distribution that is a Gaussian distribution with a mean of 10 and a standard deviation of 1. However, we find empirically that even though most of the samples from the proposal distribution result in a failure, the importance sampling estimator has a high variance. What might cause this outcome and how could we modify the proposal distribution to reduce the variance of the estimator?

Solution: The failures with the highest likelihood will occur at a friction parameter that is as close as possible to the mean of the nominal distribution while still causing the vehicle to stop within a distance greater than 10 m. However, we do not know the threshold on the friction parameter for which failures begin to occur. As shown below, we likely guessed the threshold incorrectly, resulting in a proposal distribution that assigns high likelihood to low-likelihood regions of the failure distribution and low likelihood to high-likelihood regions of the failure distribution. We can reduce the variance of the estimator by decreasing the mean of the proposal distribution or increasing the variance of the proposal distribution.



Exercise 7.7. One quantity that is often used to understand the performance of an importance sampling estimator is the *effective sample size (ESS)*. Suppose we perform importance sampling on a system with a nominal trajectory distribution of $p(\cdot)$ using a proposal distribution $q(\cdot)$. The effective sample size is defined as

$$\text{ESS} = \frac{(\sum_{i=1}^m w_i)^2}{\sum_{i=1}^m w_i^2}$$

where $w_i = p(\tau_i) \mathbb{1}\{\tau_i \notin \psi\} / q(\tau_i)$, τ_i is the i th sample from the proposal distribution, and m is the number of samples used for importance sampling. The effective sample size can be interpreted as the number of failures that would be required to achieve similar performance with direct sampling. Let n represent the number of failure samples. Show that the effective sample size is equal to n if we use a proposal distribution of the nominal trajectory distribution. Next, show that the effective sample size would be equal to m if we were able to use the failure distribution as the proposal distribution.

Solution: When the proposal distribution is the nominal trajectory distribution, the weights are equal to $p(\tau_i) \mathbb{1}\{\tau_i \notin \psi\} / p(\tau_i) = \mathbb{1}\{\tau_i \notin \psi\}$. Therefore, $\sum_{i=1}^m w_i = n$. The effective sample size is then

$$\text{ESS} = \frac{n^2}{n} = n$$

When the proposal distribution is the failure distribution, the weights are equal to $p(\tau_i) \mathbb{1}\{\tau_i \notin \psi\} / p(\tau_i) \mathbb{1}\{\tau_i \notin \psi\} = 1$. Therefore, $\sum_{i=1}^m w_i = m$. The effective sample size is then

$$\text{ESS} = \frac{m^2}{m} = m$$

Exercise 7.8. Consider a system with a nominal trajectory distribution $\mathcal{N}(0, 1^2)$. Suppose we perform multiple importance sampling by drawing the first 50 samples from a Gaussian distribution with a mean of 1 and a variance of 1^2 . We draw the next 50 samples from a Gaussian distribution with a mean of -1 and a variance of 1^2 . The first sample has a value of $\tau_1 = 0.75$. Calculate the importance weight assigned to the first sample using

1. standard multiple importance sampling (s-MIS)
2. deterministic mixture multiple importance sampling (DM-MIS)

Solution:

1. $w_1 = \frac{p(\tau_1)}{q_1(\tau_1)} = \frac{\mathcal{N}(0.75|0,1^2)}{\mathcal{N}(0.75|1,1^2)} = 0.7788$
2. $w_1 = \frac{p(\tau_1)}{\frac{1}{100} \sum_{i=1}^{100} q_i(\tau_1)} = \frac{\mathcal{N}(0.75|0,1^2)}{\frac{1}{100} [50\mathcal{N}(0.75|1,1^2) + 50\mathcal{N}(0.75|-1,1^2)]} = 1.273$

Exercise 7.9. Suppose we are using robustness as the objective for the cross entropy method. What might happen if we do not clip the threshold at each iteration to zero?

Solution: If we do not clip the threshold to zero, we will encourage the algorithm to fit a proposal distribution to the failure samples with the lowest robustness values. However, we want to find a proposal that represents the entire failure distribution rather than the failures with the lowest robustness.

Exercise 7.10. Suppose we try to run the first iteration of population Monte Carlo to estimate the probability of failure of a system, but we notice that the weights of the samples are all zero. What might cause this outcome and how could we modify the algorithm to prevent this from happening?

Solution: If the weights of the samples are all zero, it means that none of the samples from the initial population of proposal distributions were failures. There are multiple modifications we could try to prevent this from happening. One option is to increase the number of proposals in the initial population. Another option is to select the initial proposals that are more likely to produce failure samples.

Exercise 7.11. What is one advantage of sequential Monte Carlo over the cross entropy method for estimating the probability of failure?

Solution: There are multiple possible answers. One advantage is that sequential Monte Carlo is nonparametric, allowing it to better capture the structure of complex multimodal failure distributions.

Exercise 7.12. Show that equation (7.28) reduces to equation (7.2) when $\bar{q}_1(\tau) = \mathbb{1}\{\tau \notin \psi\}p(\tau)$ and $\bar{q}_2(\tau) = p(\tau)$.

Solution: Since \bar{q}_1 is the unnormalized failure distribution, its normalizing constant is the probability of failure ($z_1 = p_{\text{fail}}$). Since \bar{q}_2 is the normalized nominal distribution, its normalizing constant is $z_2 = 1$. Plugging these values into equation (7.28) gives

$$\frac{p_{\text{fail}}}{1} = \mathbb{E}_{\tau \sim p(\cdot)} \left[\frac{\mathbb{1}\{\tau \notin \psi\}p(\tau)}{p(\tau)} \right]$$

$$p_{\text{fail}} = \mathbb{E}_{\tau \sim p(\cdot)} [\mathbb{1}\{\tau \notin \psi\}]$$

Given samples from the nominal distribution $\tau_i \sim p(\cdot)$, we can approximate the above equation as

$$\hat{p}_{\text{fail}} = \frac{1}{m} \sum_{i=1}^m \mathbb{1}\{\tau_i \notin \psi\}$$

Exercise 7.13. Show that equation (7.28) reduces to equation (7.12) when $\bar{q}_1(\tau) = \mathbb{1}\{\tau \notin \psi\}p(\tau)$ and $\bar{q}_2(\tau) = q(\tau)$.

Solution: Since \bar{q}_1 is the unnormalized failure distribution, its normalizing constant is the probability of failure ($z_1 = p_{\text{fail}}$). Since \bar{q}_2 is a normalized proposal distribution, its normalizing constant is $z_2 = 1$. Plugging these values into equation (7.28) gives

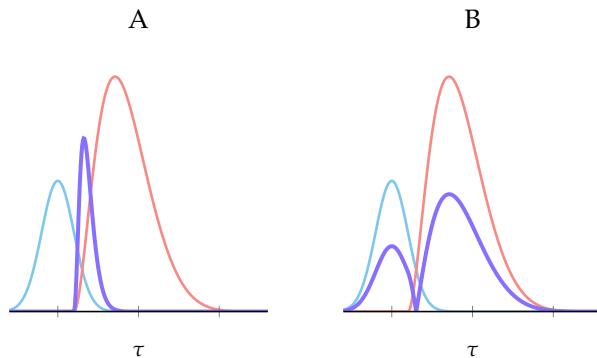
$$\frac{p_{\text{fail}}}{1} = \mathbb{E}_{\tau \sim p(\cdot)} \left[\frac{\mathbb{1}\{\tau \notin \psi\}p(\tau)}{q(\tau)} \right]$$

$$p_{\text{fail}} = \mathbb{E}_{\tau \sim p(\cdot)} \left[\frac{p(\tau)}{q(\tau)} \mathbb{1}\{\tau \notin \psi\} \right]$$

Given samples from the proposal distribution $\tau_i \sim q(\cdot)$, we can approximate the above equation as

$$\hat{p}_{\text{fail}} = \frac{1}{m} \sum_{i=1}^m \frac{p(\tau_i)}{q(\tau_i)} \mathbb{1}\{\tau_i \notin \psi\}$$

Exercise 7.14. One of the plots below shows the optimal umbrella density (purple) for the two distributions shown in red and blue. The other plot shows the optimal bridge density. Which plot shows the optimal umbrella density and which plot shows the optimal bridge density? Explain your reasoning.



Solution: The optimal umbrella density should cover high-likelihood regions for both distributions, while the optimal bridge density should bridge the gap between the two distributions. Therefore, the plot labeled A shows the optimal bridge density, and the plot labeled B shows the optimal umbrella density.

Exercise 7.15. What is one advantage of multilevel splitting over sequential Monte Carlo for estimating the probability of failure?

Solution: One advantage is that the adaptive nature of the algorithm allows us to smoothly transition from the nominal trajectory distribution to the failure distribution without the need to specify the intermediate distributions ahead of time, which we must do for sequential Monte Carlo.

8 Reachability for Linear Systems

In chapters 4 to 7, we covered a variety of sampling-based validation algorithms. We now transition to formal methods, which can provide mathematical guarantees that a system satisfies a given specification. In contrast with sampling-based algorithms, which evaluate properties based on a finite sampling of trajectories, formal methods consider the full set of possible trajectories. We first focus on the task of reachability. In this chapter, we discuss algorithms for *forward reachability* of linear systems. Forward reachability algorithms start with a set of initial states and compute the set of states that the system reaches as it progresses forward in time. This chapter begins by defining linear systems and the corresponding reachability problem. We then discuss set propagation techniques for computing reachable sets. Set propagation techniques can be computationally expensive for high-dimensional systems with long time horizons, so we also discuss overapproximation techniques that allow us to reduce the computational complexity. Finally, we outline an optimization-based approach to reachability analysis.

8.1 Forward Reachability

Forward reachability algorithms compute the set of states a system could reach over a given time horizon. To perform this analysis, we need to make some assumptions about the initial state and disturbances for the system. In the previous chapters, we sampled initial states and disturbances from probability distributions, often with support over the entire real line. However, to perform reachability computations, we need to restrict the initial states and disturbances to bounded sets.¹ We assume that the initial state comes from a bounded set \mathcal{S} and that the disturbances at each time step come from a bounded set \mathcal{X} . The disturbance set

¹ One way to convert a probability distribution to a bounded set is to use the support of the distribution. If the support of the distribution spans the entire real line, we can select a region that contains most of the probability mass.

\mathcal{X} is defined as follows:

$$\mathcal{X} = \left\{ \begin{bmatrix} \mathbf{x}_a \\ \mathbf{x}_o \\ \mathbf{x}_s \end{bmatrix} \mid \mathbf{x}_a \in \mathcal{X}_a, \mathbf{x}_o \in \mathcal{X}_o, \mathbf{x}_s \in \mathcal{X}_s \right\} \quad (8.1)$$

where \mathcal{X}_a , \mathcal{X}_s , and \mathcal{X}_o are the disturbance sets for the agent, environment, and sensor, respectively.

Given an initial state \mathbf{s} and a disturbance trajectory $\mathbf{x}_{1:d} = (\mathbf{x}_1, \dots, \mathbf{x}_d)$ with depth d , we can compute the state of the system at time step d by performing a rollout (algorithm 4.6) and taking the final state. We denote this operation as $\mathbf{s}_d = \text{Reach}(\mathbf{s}, \mathbf{x}_{1:d})$. By performing this operation on various initial states and disturbances sampled from \mathcal{S} and \mathcal{X} , we find a set of points in the state space that the system could reach at time step d . Figure 8.1 demonstrates this process on the mass-spring-damper system.

We define the *reachable set* at depth d as the set of all states that the system could reach at time step d given all possible initial states and disturbances. We write this set as

$$\mathcal{R}_d = \{\mathbf{s}_d \mid \mathbf{s}_d = \text{Reach}(\mathbf{s}, \mathbf{x}_{1:d}), \mathbf{s} \in \mathcal{S}, \mathbf{x}_t \in \mathcal{X}_t, t \in 1 : d\} \quad (8.2)$$

where \mathcal{X}_t represents the set of possible disturbances at time step t . We are often interested in the full set of states that the system might reach in a given time horizon rather than at a specific depth d . We denote this set as $\mathcal{R}_{1:h}$ and represent it as the union of the reachable sets at each depth up to the time horizon:

$$\mathcal{R}_{1:h} = \bigcup_{d=1}^h \mathcal{R}_d \quad (8.3)$$

Figure 8.2 shows the reachable sets in $\mathcal{R}_{1:4}$ for the mass-spring-damper system.

Computing reachable sets allows us to understand the behavior of a system over time. For example, we can use reachable sets to determine if a system remains within a safe region of the state space.² We call the set of states that make up the unsafe region of the state space the *unsafe set* and use this set to define a specification for the system (algorithm 8.1). If the reachable set intersects with the unsafe set, the system violates the specification.

The reachability algorithms we discuss in this chapter apply to *linear systems*. Linear systems are a class of systems for which the sensor, agent, and environment

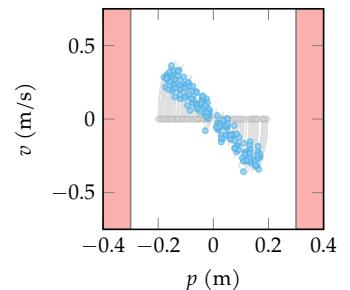


Figure 8.1. Samples from \mathcal{R}_5 for the mass-spring-damper system with initial position between -0.2 and 0.2 and initial velocity set to zero. The disturbance sets for the observation noise are bounded between -1 and 1 . The gray points represent the initial states, the gray lines show the trajectories, and the blue points represent the states after 5 time steps.

² We could also determine if the system reaches a goal region in the state space. In this case, we would want to check if the reachable set is contained within the goal region.

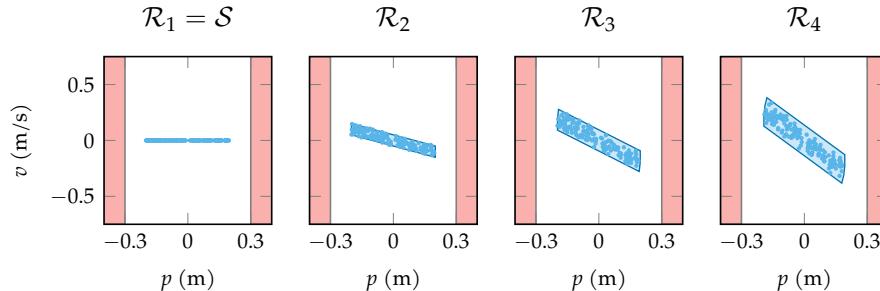


Figure 8.2. The reachable sets that make up $\mathcal{R}_{1:4}$ for the mass-spring-damper system. The blue points are samples from the reachable set generated using the Reach operator. The shaded blue regions show the true reachable sets, and the red regions make up the avoid set. Since the reachable sets do not intersect with the avoid set, the system satisfies the specification.

```
struct AvoidSetSpecification <: Specification
    set # avoid set
end
evaluate( $\psi$ ::AvoidSetSpecification,  $\tau$ ) = all(step.s  $\notin$   $\psi$ .set for step in  $\tau$ )
```

models are linear functions of the continuous state \mathbf{s} , action \mathbf{a} , observation \mathbf{o} , and disturbance \mathbf{x} . We define these models mathematically as follows:

$$O(\mathbf{s}, \mathbf{x}_o) = \mathbf{O}_s \mathbf{s} + \mathbf{x}_o \quad (8.4)$$

$$\pi(\mathbf{o}, \mathbf{x}_a) = \mathbf{\Pi}_o \mathbf{o} + \mathbf{x}_a \quad (8.5)$$

$$T(\mathbf{s}, \mathbf{a}, \mathbf{x}_s) = \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a} + \mathbf{x}_s \quad (8.6)$$

where \mathbf{O}_s , $\mathbf{\Pi}_o$, \mathbf{T}_s , and \mathbf{T}_a are matrices of the appropriate dimensions.³ Example 8.1 outlines a common linear system that we will refer to throughout this chapter.

A naïve approach to computing reachable sets would involve applying the Reach operator to all possible initial states and disturbances. However, this approach is not feasible for systems with continuous states and disturbances since there are an infinite number of possibilities. Instead, we rely on other mathematical analysis techniques to reason about the reachable set. The remainder of the chapter discusses set propagation and optimization techniques that can be used to compute reachable sets for linear systems.

Algorithm 8.1. A specification that checks if a trajectory avoids a given set. The set can be any type that supports the \notin operator. A common package for defining sets in Julia is `LazySets.jl`. M. Forets and C. Schilling, “LazySets.jl: Scalable Symbolic-Numeric Set Computations,” *Proceedings of the JuliaCon Conferences*, vol. 1, no. 1, pp. 1–11, 2021.

³ We could also multiply the disturbances by matrices, but we omit this step for simplicity.

8.2 Set Propagation Techniques

Set propagation techniques perform reachability by converting the operations in equations (8.4) to (8.6) to set operations. To introduce these techniques, we will

A common example of a linear system is a mass-spring-damper system (see diagram in caption), which can be used to model mechanical systems such as a car suspension or a bridge. The state of the system is the position (*p*) and velocity (*v*) of the mass ($\mathbf{s} = [p, v]$), the action is the force β applied to the mass, and the observation is a noisy measurement of state. The equations of motion for a mass-spring-damper system are

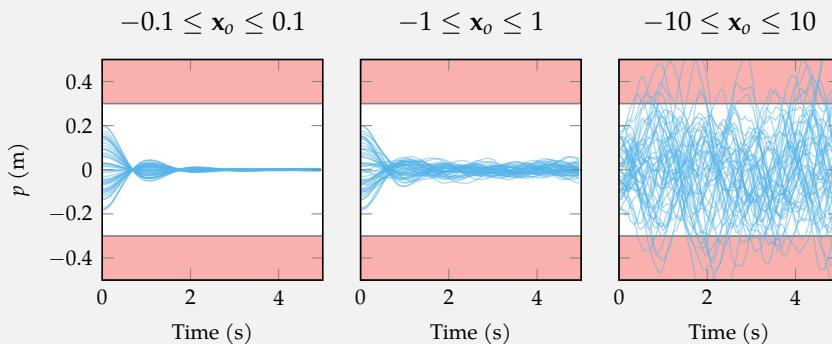
$$\begin{aligned} p' &= p + (v)\Delta t \\ v' &= v + \left(-\frac{k}{m}p - \frac{c}{m}v + \frac{1}{m}\beta \right) \Delta t \end{aligned}$$

where m is the mass, k is the spring constant, c is the damping coefficient, and Δt is the discrete time step. Rewriting the dynamics in the form of equation (8.6), we have

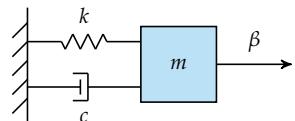
$$T(\mathbf{s}, \mathbf{a}, \mathbf{x}_s) = \begin{bmatrix} 1 & \Delta t \\ -\frac{k}{m}\Delta t & 1 - \frac{c}{m}\Delta t \end{bmatrix} \begin{bmatrix} p \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m}\Delta t \end{bmatrix} \beta + \mathbf{x}_s = \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a} + \mathbf{x}_s$$

We control the mass-spring-damper using a proportional controller such that Π_o in equation (8.5) is the gain matrix. Similarly, we model the sensor as an additive noise sensor such that \mathbf{O}_s in equation (8.4) is the identity matrix and \mathbf{x}_o is the additive noise distributed uniformly within specified bounds.

Typically, trajectories for this system with oscillate back and forth before coming to rest. In general, we want to ensure that the system remains stable, meaning that the position does not exceed some magnitude. The plots below show simulated trajectories of the system for different levels of observation noise. With enough noise, the system becomes unstable.



Example 8.1. A common example of a linear system. The mass-spring-damper system consists of a mass m attached to a wall by a spring with spring constant k and a damper with damping coefficient c . The system is controlled by a force β applied to the mass. The plots show simulated trajectories of the system for different levels of observation noise. With enough noise, the system becomes unstable.



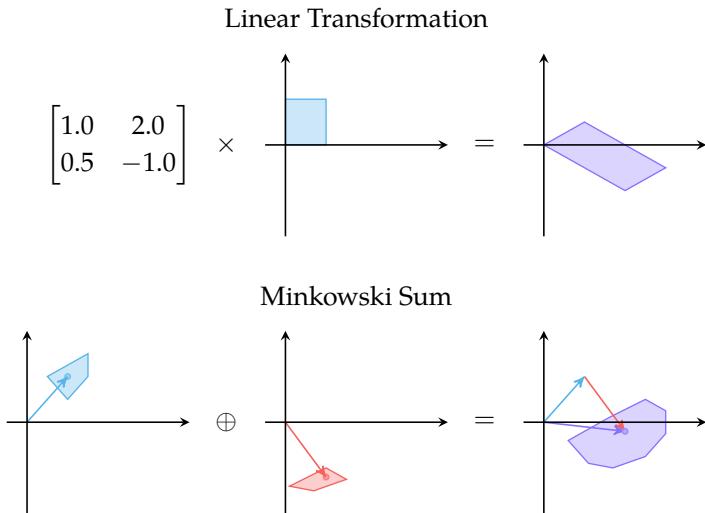


Figure 8.3. Visualization of linear set operations. Applying a linear transformation has the effect of rotating, stretching, and translating the set. The Minkowski sum of two sets is the set of all points obtained by adding each point in the first set to each point in the second set.

first focus on set propagation for the one-step reachability problem. We assume we are given a set of initial states \mathcal{S} and a set of disturbances \mathcal{X} . Our goal is to compute the set of states \mathcal{S}' that the system could reach at the next time step. Given a single initial state \mathbf{s} and disturbance \mathbf{x} , we can compute the next state \mathbf{s}' by applying equations (8.4) to (8.6) sequentially:

$$\mathbf{s}' = T(\mathbf{s}, \pi(O(\mathbf{s}, \mathbf{x}_o), \mathbf{x}_a), \mathbf{x}_s) \quad (8.7)$$

$$= \mathbf{T}_s \mathbf{s} + \mathbf{T}_a (\Pi_o (\mathbf{O}_s \mathbf{s} + \mathbf{x}_o) + \mathbf{x}_a) + \mathbf{x}_s \quad (8.8)$$

$$= (\mathbf{T}_s + \mathbf{T}_a \Pi_o \mathbf{O}_s) \mathbf{s} + \mathbf{T}_a \Pi_o \mathbf{x}_o + \mathbf{T}_a \mathbf{x}_a + \mathbf{x}_s \quad (8.9)$$

We can compute the reachable set at the next time step by applying equation (8.9) to \mathcal{S} and \mathcal{X} . To perform this computation, we must define the operations in equation (8.9) as set operations. In particular, we must be able to apply a linear transformation, or matrix multiplication, to a set and add two sets together.

The multiplication of a set \mathcal{P} by a matrix \mathbf{A} is defined as

$$\mathbf{A}\mathcal{P} = \{\mathbf{Ap} \mid \mathbf{p} \in \mathcal{P}\} \quad (8.10)$$

where the result is the set of all points obtained by multiplying each point in \mathcal{P} by \mathbf{A} . The addition of two sets \mathcal{P} and \mathcal{Q} is defined as

$$\mathcal{P} \oplus \mathcal{Q} = \{\mathbf{p} + \mathbf{q} \mid \mathbf{p} \in \mathcal{P}, \mathbf{q} \in \mathcal{Q}\} \quad (8.11)$$

where the result is the set of all points obtained by adding each point in \mathcal{P} to each point in \mathcal{Q} . This operation is referred to as the *Minkowski sum* of two sets and is often denoted using the \oplus symbol.⁴ Figure 8.3 shows these operations in two-dimensional space. As we will discuss in the next section, we can efficiently compute linear transformations and Minkowski sums for many common set types such as hyperrectangles and polytopes.⁵

With these definitions in place, we can rewrite equation (8.9) using set operations as

$$\mathcal{S}' = (\mathbf{T}_s + \mathbf{T}_a \boldsymbol{\Pi}_o \mathbf{O}_s) \mathcal{S} \oplus \mathbf{T}_a \boldsymbol{\Pi}_o \mathcal{X}_o \oplus \mathbf{T}_a \mathcal{X}_a \oplus \mathcal{X}_s \quad (8.12)$$

where \mathcal{S}' is the one-step reachable set. It is important that we simplify the system dynamics into the form of equation (8.9) before applying set operations. If we apply the equations without simplification, we may encounter a phenomenon called the *dependency effect*, which occurs when a variable appears more than once in a formula. Set operations fail to model this dependency, leading to conservative reachable sets (see example 8.2). Algorithm 8.2 implements equation (8.12).

```
function get_matrices(sys)
    return Ts(sys.env), Ta(sys.env), Pi_o(sys.agent), Os(sys.sensor)
end

function linear_set_propagation(sys, S, X)
    Ts, Ta, Pi_o, Os = get_matrices(sys)
    return (Ts + Ta * Pi_o * Os) * S ⊕ Ta * Pi_o * X.xo ⊕ Ta * X.xa ⊕ X_xs
end
```

⁴ The Minkowski sum is named after Polish mathematician Hermann Minkowski (1864–1909).

⁵ The `LazySets.jl` package in Julia provides implementations of these operations for many common sets.

Algorithm 8.2. Algorithm for computing the one-step reachable set for a linear system with initial states from \mathcal{S} and disturbances from \mathcal{X} . We use the `LazySets.jl` package to perform the set operations in equation (8.12).

To compute reachable sets over a given time horizon using set propagation techniques, we rely on the fact that the reachable set at time step d is a function of the reachable set at time step $d - 1$. Specifically, we can compute the reachable set at time step d by applying equation (8.12) to the reachable set at time step $d - 1$:

$$\mathcal{R}_d = (\mathbf{T}_s + \mathbf{T}_a \boldsymbol{\Pi}_o \mathbf{O}_s) \mathcal{R}_{d-1} \oplus \mathbf{T}_a \boldsymbol{\Pi}_o \mathcal{X}_o \oplus \mathbf{T}_a \mathcal{X}_a \oplus \mathcal{X}_s \quad (8.13)$$

Algorithm 8.3 implements this recursive algorithm for computing the reachable set at each time step. The algorithm terminates when it reaches the desired time horizon h and returns $\mathcal{R}_{1:h}$.

In addition to gaining insight into the behavior of a system, we can use reachable sets to verify that a system satisfies a given specification. For a given specification, we want to ensure that the reachable set does not intersect with its

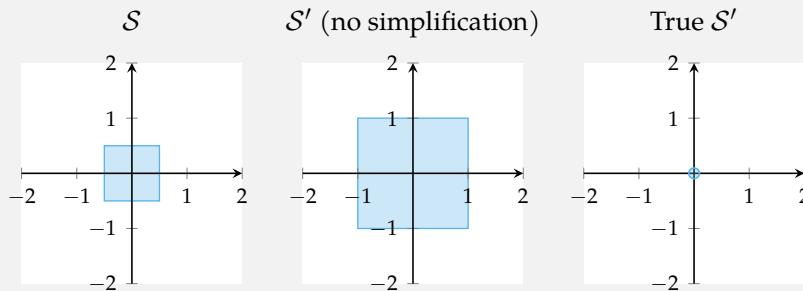
Consider a simple system with the following component models:

$$O(\mathbf{s}, \mathbf{x}_0) = \mathbf{s}$$

$$\pi(\mathbf{o}, \mathbf{x}_a) = -\mathbf{I}\mathbf{o}$$

$$T(\mathbf{s}, \mathbf{a}, \mathbf{x}_s) = \mathbf{s} + \mathbf{a}$$

where the state, action, and observation are two-dimensional and \mathbf{I} is the identity matrix. Suppose we want to compute the one-step reachable set \mathcal{S}' when the initial set is a square centered at the origin with side length 1. If we apply the sensor, agent, and environment models on the initial set without simplification, we get $\mathcal{O} = \mathcal{S}$, $\mathcal{A} = -\mathbf{I}\mathcal{O} = -\mathbf{I}\mathcal{S}$, and $\mathcal{S}' = \mathcal{S} \oplus \mathcal{A} = \mathcal{S} \oplus -\mathbf{I}\mathcal{S}$. The resulting set $\mathcal{S} \oplus -\mathbf{I}\mathcal{S}$ is a square with side length 2 centered at the origin. However, if we first simplify before switching to set operations, we get that $\mathbf{s}' = \mathbf{s} - \mathbf{s} = \mathbf{0}$. Thus, the true reachable set contains only the origin. The plots below show this result.



This mismatch is due to an effect called the dependency effect, which leads to conservative reachable sets. Because applying the set operations in order does not account for the fact that the action depends on the state, it considers worst-case behavior. For this reason, it is important to simplify the system models into the form of equation (8.9) before applying set operations to avoid unnecessary conservativeness. While this simplification is always possible for linear systems, it is not always possible for the nonlinear systems we discuss in the next chapter.

Example 8.2. Example of the dependency effect on a simple system.

```

abstract type ReachabilityAlgorithm end

struct SetPropagation <: ReachabilityAlgorithm
    h # time horizon
end

function reachable(alg::SetPropagation, sys)
    h = alg.h
    S, X = S₁(sys.env), disturbance_set(sys)
    R = S
    for t in 1:h
        S = linear_set_propagation(sys, S, X)
        R = R ∪ S
    end
    return R
end

```

Algorithm 8.3. Linear forward reachability using set propagation. The `S₁` and `disturbance_set` functions are system-specific functions that return the initial state set and disturbance set, respectively. We assume the disturbance set is the same at each time step. At each iteration, the algorithm computes the reachable set at the next time step by calling algorithm 8.2. It then adds this set to the union of reachable sets. The algorithm terminates when it reaches the desired time horizon `h` and returns $\mathcal{R}_{1:h}$.

```

~(ψ::AvoidSetSpecification) = ψ.set
function satisfies(alg::SetPropagation, sys, ψ)
    R = reachable(alg, sys)
    return !isempty(R ∩ ~ψ)
end

```

Algorithm 8.4. Checking whether a system satisfies a given specification using set propagation. The algorithm computes the reachable set $\mathcal{R}_{1:h}$ using algorithm 8.3 and checks whether its intersection with the avoid set $\neg\psi$ is empty.

Suppose we want to compute $\mathcal{R}_{1:20}$ for the mass-spring-damper system with initial position between -0.2 and 0.2 and initial velocity set to zero. We assume the observation noise is bounded between -0.2 and 0.2 . To perform reachability, we must implement the following functions for the system:

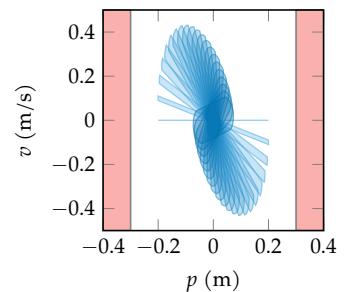
```

S₁(env::MassSpringDamper) = Hyperrectangle(low=[-0.2,0], high=[0.2,0])
function disturbance_set(sys)
    Do = sys.sensor.Do
    low = [support(d).lb for d in Do.v]
    high = [support(d).ub for d in Do.v]
    return Disturbance(ZeroSet(1), ZeroSet(2), Hyperrectangle(;low,high))
end

```

The `disturbance_set` function uses the support of the disturbance distribution from the sensor to define the observation disturbance set. We use `ZeroSet` from `LazySets.jl` for the agent and environment disturbances since they are deterministic. We can then compute the reachable set using algorithm 8.3 and visualize the reachable sets in $\mathcal{R}_{1:20}$ (see figure on the right).

Example 8.3. Computing the reachable sets for the mass-spring-damper system over a time horizon of 20 steps. The reachable sets are shown below, switching from light blue to dark blue over time.



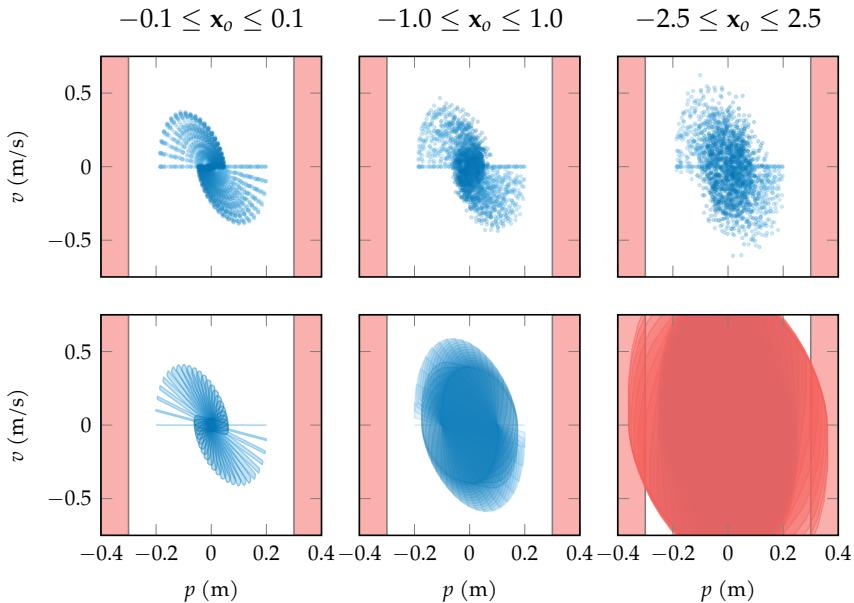


Figure 8.4. Reachable sets (bottom row) for the mass-spring-damper system with varying levels of observation noise compared to samples from a finite number of trajectory rollouts (top row). As the noise bounds increase, the reachable sets move closer to the avoid set. For the largest noise bound, the reachable set intersects with the avoid set, indicating that the system violates the specification. However, the finite number of trajectory samples do not capture this behavior. Formal methods such as reachability are able to identify this violation by considering the entire reachable set.

complement such that

$$\mathcal{R}_{1:h} \cap \neg\psi = \emptyset \quad (8.14)$$

For a specification that is defined as an avoid set, we can check if the system satisfies the specification by verifying that the reachable set does not intersect with the avoid set (algorithm 8.4). Figure 8.4 shows the reachable sets for the mass-spring-damper system with varying levels observation noise compared to a finite sampling of reachable points. When the noise becomes large enough, the reachable sets intersect with the avoid set, indicating that the system violates the specification.

In general, the safety guarantee derived from equation (8.14) only holds up to the horizon h . In other words, there is no guarantee that the system will not enter the avoid set after the time horizon. However, if we observe certain convergence properties of the reachable set, we can extend the safety guarantee to infinite time. Specifically, if at any point in algorithm 8.3 we find that $\mathcal{R}_d \subseteq \mathcal{R}_{d-1}$ (figure 8.5), we can conclude that \mathcal{R}_d is an *invariant set*, meaning that the system will stay within this set indefinitely.⁶

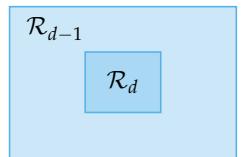


Figure 8.5. Example of an invariant set \mathcal{R}_d such that $\mathcal{R}_d \subseteq \mathcal{R}_{d-1}$.

⁶We can use `LazySets.jl` to check this property for convex sets. It is also the case that if $\mathcal{R}_d \subseteq \mathcal{R}_{1:d-1}$, then $\mathcal{R}_{1:d}$ is an invariant set. However, this property is generally more difficult to check since it requires checking whether a set is a subset of a nonconvex set.

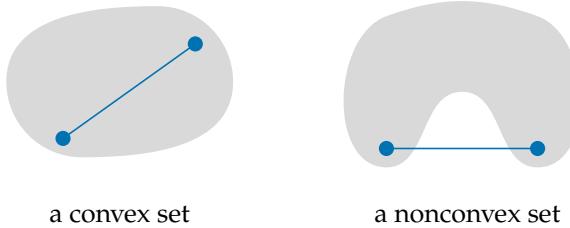


Figure 8.6. An example of a convex and nonconvex set. For the nonconvex set, it is possible to draw a line segment connecting two points in the set that is not entirely contained within the set.

8.3 Set Representations

To ensure that algorithms 8.2 to 8.4 are tractable, we must select set representations that are computationally efficient. Desirable properties include:

- *Finite representations*: We should be able to specify the points that are contained in the set without needing to enumerate all of them.
- *Efficient set operations*: We should be able to perform set operations such as linear transformations, Minkowski sums, and intersection efficiently.
- *Closure under set operations*: A set representation is closed under a particular set operation if applying the operation results in a set of the same type.

In this chapter, we will focus on *convex* set representations, which tend to have these properties.⁷ A *convex set* is a set for which a line drawn between any two points in the set is contained entirely within the set. Mathematically, a set \mathcal{P} is convex if we have

$$\alpha \mathbf{p} + (1 - \alpha) \mathbf{q} \in \mathcal{P} \quad (8.15)$$

for all $\mathbf{p}, \mathbf{q} \in \mathcal{P}$ and $\alpha \in [0, 1]$. Figure 8.6 illustrates this property. The rest of this section discusses a common convex set representation called *polytopes*.

8.3.1 Polytopes

A *polytope* is defined as the bounded intersection of a set of *linear inequalities*.⁸ A linear inequality has the form $\mathbf{a}^\top \mathbf{x} \leq b$ where \mathbf{a} is a vector of coefficients, \mathbf{x} is a vector of variables, and b is a scalar. We refer to the set of points that satisfy a given linear inequality as a *half space*. A *polyhedron* is the intersection of a finite number of half spaces. If the polyhedron is bounded, we call it a polytope.⁹ Figure 8.7 illustrates these concepts in two dimensions.

⁷ Some nonconvex sets can also be efficiently represented and manipulated. A detailed overview is provided in M. Althoff, G. Frehse, and A. Girard, “Set Propagation Techniques for Reachability Analysis,” *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 4, pp. 369–395, 2021.

⁸ We can also define convex sets such as ellipsoids using nonlinear inequalities. O. Maler, “Computing Reachable Sets: An Introduction,” *French National Center of Scientific Research*, pp. 1–8, 2008.

⁹ Other definitions for polyhedron and polytope are used in different contexts. In some cases, an unbounded intersection of halfspaces is referred to as an unbounded polytope, and a polyhedron refers to three-dimensional shape with planar faces.

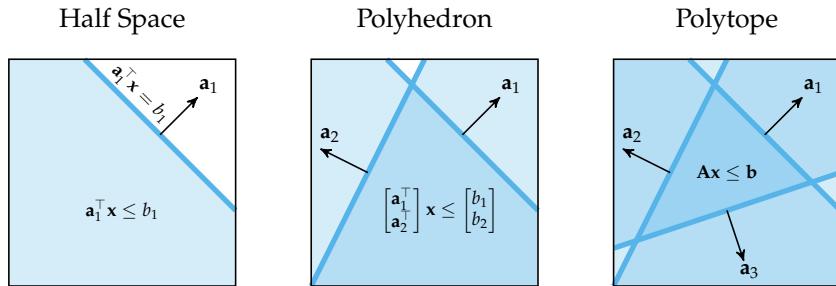


Figure 8.7. Example of a half space, polyhedron, and polytope in two dimensions. A half space is defined by a single linear inequality, a polyhedron is the intersection of multiple half spaces, and a polytope is a bounded polyhedron.

We can represent polytopes in different ways. An \mathcal{H} -polytope is a polytope represented as a set of half spaces. It is written in the form $\mathbf{Ax} \leq \mathbf{b}$ where \mathbf{A} and \mathbf{b} are formed by stacking the linear inequalities from the half spaces. A \mathcal{V} -polytope is a polytope represented as the *convex hull* of a set of vertices \mathcal{V} , written as $\text{conv}(\mathcal{V})$. The convex hull of a set of points \mathcal{V} is the set of all possible *convex combinations* of the points. A convex combination of a set of points $\{\mathbf{v}_1, \dots, \mathbf{v}_n\}$ is a linear combination of the form

$$\lambda_1 \mathbf{v}_1 + \dots + \lambda_n \mathbf{v}_n \quad (8.16)$$

such that $\sum_{i=1}^n \lambda_i = 1$ and $\lambda_i \geq 0$ for all i . Intuitively, the convex hull of a set of points is the smallest convex set that contains all the points (figure 8.8).

It is always possible to convert between the two polytope representations; however, the calculation is nontrivial.¹⁰ Each representation has different advantages. For example, \mathcal{H} -polytopes are more efficient for checking whether a point belongs to the set because we can simply check if it satisfies all the linear inequalities. In contrast, \mathcal{V} -polytopes are more efficient for set operations such as linear transformations. To compute a linear transformation of a polytope represented as a \mathcal{V} -polytope, we can apply the transformation to each vertex to obtain the vertices of the transformed polytope.

The Minkowski sum of two \mathcal{V} -polytopes is

$$\mathcal{P}_1 \oplus \mathcal{P}_2 = \text{conv}(\{\mathbf{v}_1 + \mathbf{v}_2 \mid \mathbf{v}_1 \in \mathcal{V}_1, \mathbf{v}_2 \in \mathcal{V}_2\}) \quad (8.17)$$

where \mathcal{V}_1 and \mathcal{V}_2 are the vertices of \mathcal{P}_1 and \mathcal{P}_2 , respectively. In other words, we can obtain all candidates for the vertices of the Minkowski sum by taking the sum

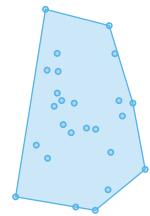
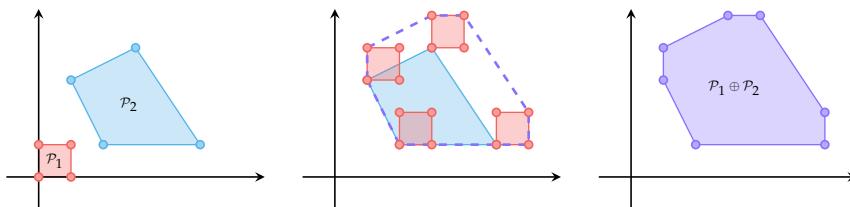


Figure 8.8. The convex hull of a set of points.

¹⁰ A detailed overview is provided in G. M. Ziegler, *Lectures on Polytopes*. Springer Science & Business Media, 2012, vol. 152. In Julia, `LazySets.jl` provides functionality to convert between the two representations.

Figure 8.9. Computing the Minkowski sum of two \mathcal{V} -polytopes.

of all pairs of vertices from the two polytopes. To determine which candidates are actual vertices, we must determine which candidate vertices are on the boundary of the convex hull. Figure 8.9 illustrates this process.

We can use these results to reason about the complexity of algorithm 8.3 if we were to represent our sets as polytopes. We apply equation (8.12) at each iteration, which involves three linear transformations and three Minkowski sums. The number of candidate vertices resulting from computing the one-step reachable set using equation (8.12) is $|\mathcal{S}_1||\mathcal{X}_o||\mathcal{X}_a||\mathcal{X}_s|$ where $|\mathcal{P}|$ represents the number of vertices in polytope \mathcal{P} . The number of candidate vertices for the reachable set at depth d is then $|\mathcal{S}_1|(|\mathcal{X}_o||\mathcal{X}_a||\mathcal{X}_s|)^d$. We can prune the candidate vertices that are not actual vertices by computing the convex hull of the candidate vertices, but this operation can be expensive.¹¹ Therefore, the exponential growth in the number of candidate vertices creates tractability challenges for high-dimensional systems with long time horizons.¹²

8.3.2 Zonotopes

A *zonotope* is a special type of polytope that avoids the exponential growth in candidate vertices for Minkowski sums. It is defined as the Minkowski sum of a set of line segments centered at a point \mathbf{c} :

$$\mathcal{Z} = \{\mathbf{c} + \sum_{i=1}^m \alpha_i \mathbf{g}_i \mid \alpha_i \in [-1, 1]\} \quad (8.18)$$

where $\mathbf{g}_{1:m}$ are referred to as the *generators* of the zonotope.¹³ We represent zonotopes by a center point and list of generators:

$$\mathcal{Z} = (\mathbf{c}, \langle \mathbf{g}_{1:m} \rangle) \quad (8.19)$$

¹¹ The most efficient algorithms for computing the vertices of the convex hull of a set of points have a complexity of $O(mv)$ where m is the number of candidate vertices and v is the number of actual vertices. In general, the number of actual vertices grows super-linearly. For more details, see R. Seidel, “Convex Hull Computations,” in *Handbook of Discrete and Computational Geometry*, Chapman and Hall, 2017, pp. 687–703.

¹² Other polytope representations such as the \mathcal{Z} -representation and \mathcal{M} -representation perform Minkowski sums more efficiently. More details are provided in S. Sigl and M. Althoff, “M-Representation of Polytopes,” *ArXiv:2303.05173*, 2023.

¹³ Zonotopes can also be viewed as linear transformations of the unit hypercube.

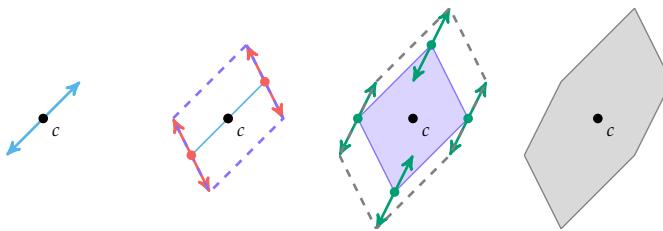


Figure 8.10. Iterative construction of a zonotope centered at a point c by taking the Minkowski sum of its generators. The generators are shown below.



Figure 8.10 shows the construction of a zonotope from its generators.

To apply a linear transformation to a zonotope, we apply the transformation to the center and each generator:

$$\mathbf{A}\mathcal{Z} = (\mathbf{Ac}, \langle \mathbf{Ag}_{1:m} \rangle) \quad (8.20)$$

To compute the Minkowski sum of two zonotopes, we sum the centers and concatenate the generators:

$$\mathcal{Z} \oplus \mathcal{Z}' = (c + c', \langle \mathbf{g}_{1:m}, \mathbf{g}'_{1:m'} \rangle) \quad (8.21)$$

Note that the number of generators in the resulting zonotope grows linearly with the number of generators in each zonotope. Therefore, if we represent our sets as zonotopes, the number of generators for the reachable set at depth d is $|\mathcal{S}_1| + d(|\mathcal{X}_o| + |\mathcal{X}_a| + |\mathcal{X}_s|)$. This linear growth represents a significant improvement over the exponential growth in candidate vertices for generic polytopes.

8.3.3 Hyperrectangles

A *hyperrectangle* is a generalization of a rectangle to higher dimensions (figure 8.12). It is a special type of zonotope in which the generators are aligned with the axes. We may also work with linear transformations of hyperrectangles, which can always be transformed back to an axis-aligned representation. All hyperrectangles are zonotopes, and all zonotopes are polytopes; however, the reverse does not hold (figure 8.11). Hyperrectangles can be compactly represented as a center point and a vector of half-widths. They can also be represented as a set of *intervals* with one for each dimension. Unlike zonotopes, hyperrectangles are not closed under linear transformations and Minkowski sums.

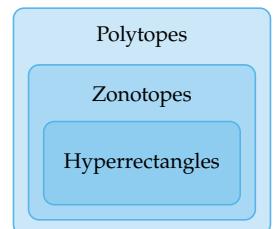


Figure 8.11. Zonotopes are a subclass of polytopes, and hyperrectangles are a subclass of zonotopes.



Figure 8.12. Example of a hyperrectangle in one, two, and three dimensions. In one dimension, a hyperrectangle is equivalent to an interval.

8.4 Reducing Computational Cost

As noted in section 8.3.1, the number of candidate vertices for the reachable sets in algorithm 8.3 grows exponentially with the time horizon and causes computational challenges for high-dimensional systems. There are multiple ways to reduce this computational burden. One way is to represent the initial state and disturbance sets using zonotopes since the number of generators scales linearly with the time horizon (see section 8.3.2). In this section, we will discuss another technique to reduce the computational cost that relies on *overapproximation*.

The set $\tilde{\mathcal{P}}$ represents an overapproximation of the set \mathcal{P} if $\mathcal{P} \subseteq \tilde{\mathcal{P}}$. Typically, we select the overapproximated set $\tilde{\mathcal{P}}$ such that it is easier to compute or represent. For example, we can use overapproximation to reduce the computational cost of algorithm 8.3 by overapproximating the reachable set at each iteration with a set that has fewer vertices (figure 8.13). We can then use this overapproximated set as the initial set for the next iteration.

As long as the overapproximated reachable set does not intersect with the avoid set, we can still use it to make claims about the safety of the system. However, if the overapproximated reachable set does intersect with the avoid set, the results are inconclusive. The violation could be due to unsafe behavior or the overapproximation itself. In this case, we could move to a tighter overapproximation or use a different method to verify safety (example 8.4).

Algorithm 8.5 modifies algorithm 8.3 to include overapproximation. Depending on the complexity of the reachable sets, we may not need to overapproximate at every iteration, so we set a frequency parameter to control how often we overapproximate. Figure 8.14 demonstrates this idea on the mass-spring-damper system. A more frequent overapproximation will result in greater computational efficiency at the cost of extra *overapproximation error* in the reachable sets. We define overapproximation error as the difference in volume between the overapproximated reachable set and the true reachable set.

The overapproximation tolerance ϵ places a bound on the *Hausdorff distance* between the overapproximated set and the original set.¹⁴ The Hausdorff distance

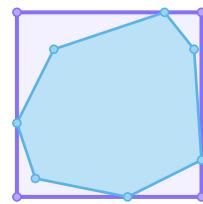


Figure 8.13. Overapproximating the blue polytope with the purple polytope. The purple polytope has fewer vertices.

¹⁴ The Hausdorff distance is named after German mathematician Felix Hausdorff (1868–1942).

```

struct OverapproximateSetPropagation <: ReachabilityAlgorithm
    h      # time horizon
    freq  # overapproximation frequency
    ε     # overapproximation tolerance
end

function reachable(alg::OverapproximateSetPropagation, sys)
    h, freq, ε = alg.h, alg.freq, alg.ε
    S, X = S₁(sys.env), disturbance_set(sys)
    R = S
    for t in 1:h
        S = linear_set_propagation(sys, S, X)
        R = R ∪ S
        S = t % freq == 0 ? overapproximate(S, ε) : S
    end
    return R
end

```

Algorithm 8.5. Overapproximate linear forward reachability using set propagation. At each iteration, the algorithm calls algorithm 8.2 to compute the reachable set at the next time step. If the current time step matches up with the overapproximation frequency, the algorithm calls the `overapproximate` function from `LazySets.jl` to compute an ϵ -close overapproximation of the reachable set for use at the next time step. Section 8.4.2 describes how the overapproximation function works.

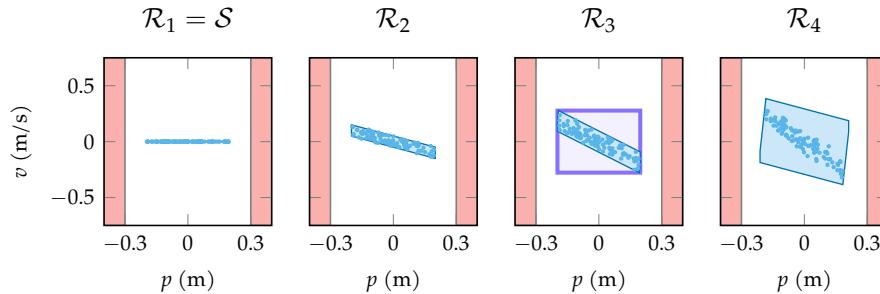


Figure 8.14. An overapproximation of $\mathcal{R}_{1:4}$ for the mass-spring-damper system. We reduce the number of vertices in \mathcal{R}_3 by overapproximating it with the purple polytope. This overapproximation results in fewer vertices for \mathcal{R}_4 but causes it to produce a conservative estimate of the reachable set.

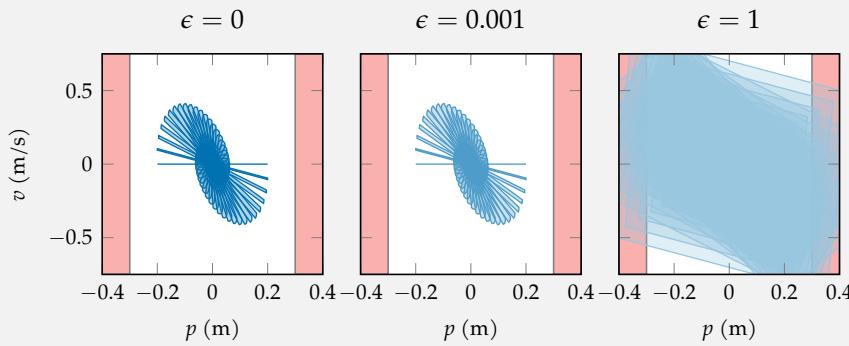
between two sets \mathcal{P} and $\tilde{\mathcal{P}}$ is the maximum distance from a point in \mathcal{P} to the nearest point in $\tilde{\mathcal{P}}$. A lower value for ϵ results in a less conservative overapproximation but may require more computation and result in a more complex representation. The rest of this section discusses a technique for computing this overapproximation.

8.4.1 Support Functions

We can overapproximate convex sets by sampling their *support function*. The support function ρ of a set $\mathcal{P} \subset \mathbb{R}^n$ is defined as

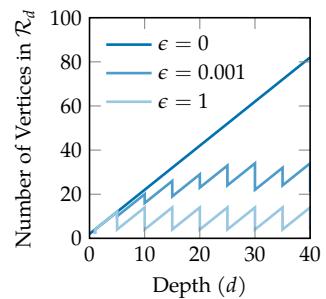
$$\rho(\mathbf{d}) = \max_{\mathbf{p} \in \mathcal{P}} \mathbf{d}^\top \mathbf{p} \quad (8.22)$$

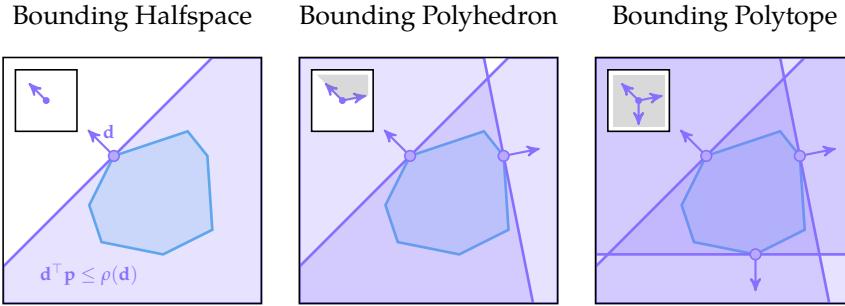
Suppose we want to determine if the mass-spring-damper system could reach the avoid set within 40 time steps. To reduce computational cost, we use algorithm 8.5 with an overapproximation frequency of 5 time steps. The plots below show the reachable set $\mathcal{R}_{1:40}$ using three different overapproximation tolerances ϵ . The plot on the right shows the number of vertices in \mathcal{R}_d for each depth d .



The first tolerance of $\epsilon = 0$ results in no overapproximation, but the number of vertices grows quickly. The highest tolerance of $\epsilon = 1$ results in significantly fewer vertices, but it is too conservative to the point where the reachable set overlaps with the avoid set. Therefore, the results of the analysis with $\epsilon = 1$ are inconclusive. The middle tolerance of $\epsilon = 0.001$ strikes a balance between the two extremes. With this tolerance, we are still able to verify safety while reducing the computational cost.

Example 8.4. The effect of overapproximation on accuracy and computational cost for the mass-spring-damper system. The plots show the reachable sets $\mathcal{R}_{1:40}$ using three different overapproximation tolerances. The plot below shows the number of vertices in the reachable sets at each depth. If the tolerance is too high, the reachable set may overlap with the avoid set, and the analysis is inconclusive.





where \mathbf{d} is a direction vector. The maximizer of the support function is called the *support vector*:

$$\sigma(\mathbf{d}) = \arg \max_{\mathbf{p} \in \mathcal{P}} \mathbf{d}^\top \mathbf{p} \quad (8.23)$$

For polytopes, there will always be a support vector in a given direction \mathbf{d} that corresponds to one of its vertices (figure 8.16). In fact, evaluating the support function of a \mathcal{V} -polytope at a direction \mathbf{d} involves computing $\mathbf{d}^\top \mathbf{v}$ for each vertex $\mathbf{v} \in \mathcal{V}$ and taking the maximum. Evaluating the support function of an \mathcal{H} -polytope requires solving a linear program. The support function of a zonotope can be computed in closed form as a function of its generators.¹⁵

The support function of a set \mathcal{P} can be used to define a half space that contains the set:

$$\{\mathbf{p} \mid \mathbf{d}^\top \mathbf{p} \leq \rho(\mathbf{d})\} \quad (8.24)$$

By evaluating the support function on a set of directions $\mathcal{D} = \{\mathbf{d}_1, \dots, \mathbf{d}_m\}$ and taking the intersection of the resulting half spaces, we obtain a polyhedral overapproximation of the set \mathcal{P} :

$$\tilde{\mathcal{P}} = \bigcap_{\mathbf{d} \in \mathcal{D}} \{\mathbf{p} \mid \mathbf{d}^\top \mathbf{p} \leq \rho(\mathbf{d})\} \quad (8.25)$$

For the overapproximation to be a polytope, the set \mathcal{D} must be a *positive spanning set*. The set of directions \mathcal{D} represents a positive spanning set if we can construct any point in \mathbb{R}^n as a convex combination of the directions in \mathcal{D} .¹⁶ Figure 8.15 demonstrates this concept in \mathbb{R}^2 .

Figure 8.15. Overapproximating a polytope by evaluating its support function in various directions. When we only use one direction (left), the overapproximation is a half space. By using multiple directions (center), we can construct a polyhedral overapproximation. However, the two vectors only positively span the shaded cone, resulting in an unbounded overapproximation. By adding a third direction (right), we can construct a set of directions that positively span \mathbb{R}^2 and produce a bounded overapproximate set.

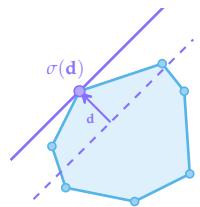


Figure 8.16. Support vector of a polytope in a given direction \mathbf{d} . The support vector is the vertex of the polytope that maximizes the support function.

¹⁵ M. Althoff and G. Frehse, “Combining Zonotopes and Support Functions for Efficient Reachability Analysis of Linear Systems,” in *IEEE Conference on Decision and Control (CDC)*, 2016.

¹⁶ R. G. Regis, “On the Properties of Positive Spanning Sets and Positive Bases,” *Optimization and Engineering*, vol. 17, no. 1, pp. 229–262, 2016.

The choice of directions in \mathcal{D} affects the tightness of the overapproximation. In general, adding more direction vectors to \mathcal{D} will decrease overapproximation error. As we approach all possible direction vectors, the overapproximation converges to the set itself. However, more direction vectors will require more computation to create the overapproximated set and will result in a more complex overapproximate representation.

We want to select the directions in \mathcal{D} to balance between overapproximation error and computational cost. If we have no prior information about the shape of the set, a common choice is to add a direction in the positive and negative direction of each axis. This choice will result in a hyperrectangular overapproximation. However, other choices of directions may result in a tighter overapproximation (figure 8.17). Section 8.4.2 discusses an iterative algorithm for intelligently selecting these directions.

8.4.2 Iterative Refinement

One way to select the directions in \mathcal{D} is to use a process called *iterative refinement*.¹⁷ The algorithm proceeds as follows:

1. Begin with a positive spanning set of template directions \mathcal{D} and compute the corresponding overapproximate polytope by evaluating the support function in each direction. A common choice is the positive and negative directions of the axes.
2. Compute an inner approximation by taking the convex hull of the corresponding support vectors.
3. Compute the distance between each facet of the inner approximation and the nearest vertex of the outer approximation.
4. Add the direction of the face that is furthest from the nearest vertex to \mathcal{D} and return to step 1.

The process is repeated until the maximum distance between the inner and outer approximations is less than a specified tolerance ϵ . Figure 8.18 shows the steps involved in a single iteration of the algorithm, and figure 8.19 demonstrates the process over multiple iterations.

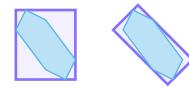


Figure 8.17. Two different overapproximations of the blue polytope that each use four evaluations of the support function. The first overapproximation evaluates the support function in the positive and negative directions of the axes. The second overapproximation uses the directions of the diagonals of the unit square. The choice of directions affects the tightness of the overapproximation.

¹⁷ This method is implemented in the `LazySets.jl` package as the `overapproximate` function. For more details, see G. K. Kamenev, “An Algorithm for Approximating Polyhedra,” *Computational Mathematics and Mathematical Physics*, vol. 4, no. 36, pp. 533–544, 1996.

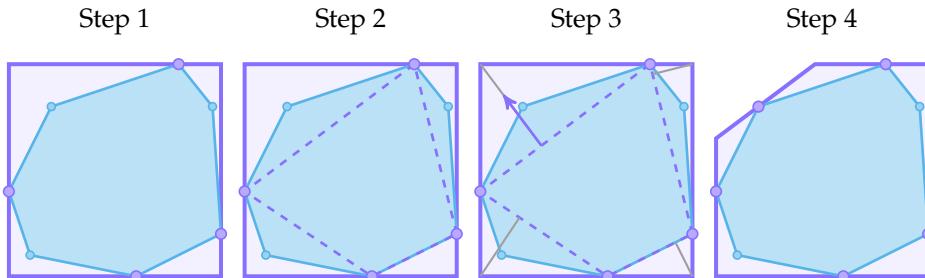


Figure 8.18. Illustration of the steps involved in a single iteration of the iterative refinement algorithm. In this example, the initial overapproximation is a hyperrectangle. The distance between the inner and outer approximations is computed for each face of the inner approximation. The direction of the face that is furthest from the nearest vertex (purple arrow) is added to the template directions.

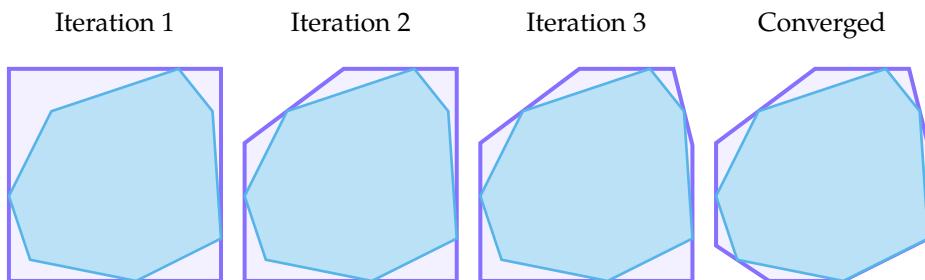


Figure 8.19. The resulting overapproximated polytope for various iterations of the iterative refinement algorithm. The Hausdorff distance between the overapproximated set and the true set decreases with each iteration until it is within the specified tolerance $\epsilon = 0.7$.

8.5 Linear Programming

Another technique for computing overapproximate reachable sets of linear systems is to directly evaluate the support function of the reachable set at a desired depth d :

$$\rho_d(\mathbf{d}) = \max_{\mathbf{s} \in \mathcal{R}_d} \mathbf{d}^\top \mathbf{s} \quad (8.26)$$

Similar to the support function of a polytope, the support function of a reachable set can be used to construct an overapproximation of the reachable set. We form the overapproximation by evaluating the support function in a set of directions \mathcal{D} and taking the intersection of the resulting half spaces.

We can solve the optimization problem in equation (8.26) using a *linear program* solver. A linear program is an optimization problem where the objective function

and constraints are all linear. The linear program for equation (8.26) is

$$\begin{aligned} & \underset{\mathbf{s}_{1:d}, \mathbf{x}_{1:d}}{\text{maximize}} \quad \mathbf{d}^\top \mathbf{s}_d \\ & \text{subject to} \quad \mathbf{s}_1 \in \mathcal{S} \\ & \quad \mathbf{x}_t \in \mathcal{X}_t \text{ for all } t \in 1 : d \\ & \quad \mathbf{s}_{t+1} = \text{Step}(\mathbf{s}_t, \mathbf{x}_t) \text{ for all } t \in 1 : d - 1 \end{aligned} \tag{8.27}$$

where

$$\text{Step}(\mathbf{s}, \mathbf{x}) = (\mathbf{T}_s + \mathbf{T}_a \boldsymbol{\Pi}_o \mathbf{O}_s) \mathbf{s} + \mathbf{T}_a \boldsymbol{\Pi}_o \mathbf{x}_o + \mathbf{T}_a \mathbf{x}_a + \mathbf{x}_s \tag{8.28}$$

The decision variables in equation (8.27) are the state and disturbances at each time step. The constraints enforce that the state and disturbances are within their respective sets and that the state evolves according to equation (8.9). The optimization problem in equation (8.27) can be solved efficiently using a variety of algorithms.¹⁸

For the optimization problem in equation (8.27) to be a linear program, the sets \mathcal{S} and \mathcal{X}_t must be polytopes. We can write them as a set of linear inequalities using their \mathcal{H} -polytope representations. Algorithm 8.6 implements the linear program for computing the support function of a reachable set at a particular depth d . Given a desired time horizon h and a set of directions \mathcal{D} , we can compute an overapproximation of $\mathcal{R}_{1:h}$ by evaluating the support function at each direction for each depth. Algorithm 8.7 implements this process.

Similar to the polytope overapproximation in section 8.4, the choice of the directions in \mathcal{D} affects the tightness of the reachable set overapproximation. We could select the directions to align with the axes or use more sophisticated methods like the iterative refinement algorithm in section 8.4.2. Since linear program solvers are computationally efficient, another option is to simply evaluate the support function at many randomly sampled directions. We could also select the directions using trajectory samples. Given a set of samples from the reachable set, we can use principal component analysis (PCA)¹⁹ to determine the directions that best capture the shape of the set.²⁰

The overapproximate reachable sets improve our understanding of the behavior of the system. However, if our ultimate goal is to check intersection with a convex avoid set \mathcal{U} , we can solve the problem exactly without the need for

¹⁸ Modern linear programming solvers can solve problems with thousands of variables and constraints. H. Karloff, *Linear Programming*. Springer, 2008.

¹⁹ H. Abdi and L. J. Williams, "Principal Component Analysis," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 4, pp. 433–459, 2010.

²⁰ O. Stursberg and B. H. Krogh, "Efficient Representation and Computation of Reachable Sets for Hybrid Systems," in *Hybrid Systems: Computation and Control*, 2003.

```

Ab(p) = tosimplehrep(constraints_list(p))

function constrained_model(sys, d, S, X)
    model = Model(SCS.Optimizer)
    @variable(model, s[1:dim(S),1:d])
    @variable(model, xo[1:dim(X.xo),1:d])
    @variable(model, xs[1:dim(X.xs),1:d])
    @variable(model, xa[1:dim(X.xa),1:d])

    As, bs = Ab(S)
    (Axo, bxo), (Axs, bxs), (Axa, bxa) = Ab(X.xo), Ab(X.xs), Ab(X.xa)
    @constraint(model, As * s[:, 1] .≤ bs)
    for i in 1:d
        @constraint(model, Axo * xo[:, i] .≤ bxo)
        @constraint(model, Axs * xs[:, i] .≤ bxs)
        @constraint(model, Axa * xa[:, i] .≤ bxa)
    end

    Ts, Ta, Πo, Os = get_matrices(sys)
    for i in 1:d-1
        @constraint(model, (Ts + Ta*Πo*Os) * s[:, i] + Ta*Πo * xo[:, i]
                        + Ta * xa[:, i] + xs[:, i] .== s[:, i+1])
    end
    return model
end

function ρ(model, d, d)
    s = model.obj_dict[:s]
    @objective(model, Max, d' * s[:, d])
    optimize!(model)
    return objective_value(model)
end

```

Algorithm 8.6. Computing the support function of a reachable set at a desired depth d . The `constrained_model` function constructs an optimization model with the constraints in equation (8.27) that is compatible with the `JuMP.jl` package. It uses the `Ab` function to convert a polytope to a set of linear inequalities. Given this model and a direction vector d , the ρ function solves the linear program and returns the value of the support function.

```

struct LinearProgramming <: ReachabilityAlgorithm
    h # time horizon
    D # set of directions to evaluate support function
    tol # tolerance for checking satisfaction
end

function reachable(alg::LinearProgramming, sys)
    h, D = alg.h, alg.D
    S, X = S1(sys.env), disturbance_set(sys)
    R = S
    for d in 2:h
        model = constrained_model(sys, d, S, X)
        ps = [p(model, d, d) for d in D]
        R = R ∪ HPolytope([HalfSpace(d, p) for (d, p) in zip(D, ps)])
    end
    return R
end

```

Algorithm 8.7. Linear forward reachability using linear programming. The system-specific S_1 and disturbance_set functions return the initial state set and disturbance set, respectively. For each depth, the algorithm creates the constrained model and evaluates the support function at each direction in D . It then constructs a polytope from the results and takes its union with the current reachable set. The algorithm returns $R_{1:h}$. The `tol` input is a tolerance used by algorithm 8.8.

overapproximation. Specifically, we solve the following optimization problem:

$$\begin{aligned}
 & \underset{\mathbf{s}_{1:d}, \mathbf{x}_{1:D}}{\text{minimize}} \quad \|\mathbf{s}_d - \mathbf{u}\| \\
 & \text{subject to} \quad \mathbf{u} \in \mathcal{U} \\
 & \quad \mathbf{s}_1 \in \mathcal{S} \\
 & \quad \mathbf{x}_t \in \mathcal{X}_t \quad \text{for all } t \in \{1, \dots, d\} \\
 & \quad \mathbf{s}_{t+1} = \text{Step}(\mathbf{s}_t, \mathbf{x}_t) \quad \text{for all } t \in \{1, \dots, d-1\}
 \end{aligned} \tag{8.29}$$

The solution to the optimization problem in equation (8.29) is the minimum distance between any point in the reachable set and the avoid set. If this distance is greater than zero, we can conclude that the reachable set does not intersect the avoid set at depth d . If the distance is equal to zero, we can conclude that the reachable set intersects the avoid set.

The norm in the objective function of equation (8.29) means that it is no longer a linear program. It is, however, a convex program as long as the avoid set is convex. If the avoid set is a union of convex sets, we can check intersection with each component separately (see example 8.5). Convex programs can be solved efficiently using a variety of algorithms.²¹ Algorithm 8.8 implements this check for a given time horizon. For each depth, the algorithm solves the optimization problem in equation (8.29) and checks if the objective value is within some tolerance of zero. If the objective value is zero at any depth, the system does not satisfy the specification.

²¹ S. P. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

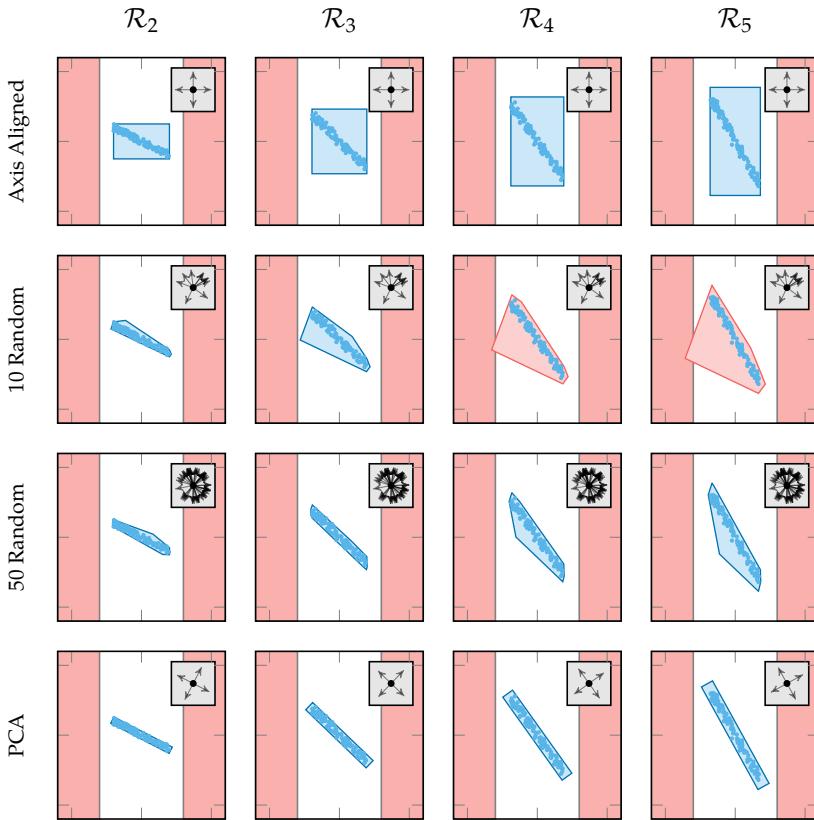


Figure 8.20. Overapproximate reachable sets for the mass-spring-damper system using linear programming. On each plot, the x -axis is position, and the y -axis is velocity. Each row uses a different strategy for selecting \mathcal{D} . The first row uses directions aligned with the axes, the second row uses 10 randomly sampled directions, the third row uses 50 randomly sampled directions, and the fourth row uses directions selected based on the principal components of the trajectory samples. When we randomly sample only 10 directions, the overapproximation is too conservative to verify safety.

8.6 Summary

- While the sampling-based methods in the previous chapters draw conclusions based on a finite sampling of trajectories, formal methods such as reachability analysis consider the entire set of possible trajectories.
- We can compute reachable sets for linear systems by propagating sets through the system dynamics.
- We can efficiently propagate convex sets such as polytopes, zonotopes, and hyperrectangles through linear equations.

The avoid set for the mass-spring-damper system can be written as the union of two convex sets. Specifically, we require that $|p| < 0.3$. The first set is therefore represented by the linear inequality $[1, 0]^\top s \leq -0.3$, and the second set is represented by the linear inequality $[-1, 0]^\top s \leq -0.3$. To check whether the system could reach the avoid set, we run algorithm 8.8 for each component of the avoid set. The system does not satisfy the specification if the algorithm returns false for either component.

Example 8.5. Checking whether the mass-spring-damper system can reach the avoid set using convex programming.

```

function satisfies(alg::LinearProgramming, sys, ψ)
    S, X = S₁(sys.env), disturbance_set(sys)
    for d in 1:alg.h
        model = constrained_model(sys, d, S, X)
        @variable(model, u[1:dim(S)])
        Au, bu = Ab(¬ψ)
        @constraint(model, Au * u .≤ bu)
        s = model.obj_dict[:s]
        @objective(model, Min, sum((s[i, d] - u[i])^2 for i in 1:dim(S)))
        optimize!(model)
        if isapprox(objective_value(model), 0.0, atol=alg.tol)
            return true
        end
    end
    return false
end

```

Algorithm 8.8. Checking whether a system could reach a convex avoid set using convex programming. For each depth, the algorithm constructs a constrained model that considers the initial state, disturbances, and system dynamics. It then adds a variable for the avoid set and minimizes the squared distance between the reachable set and the avoid set (equivalent to minimizing the norm). If the distance is zero (within the numerical tolerance) at any depth, the system does not satisfy the specification.

- If the number of vertices in the reachable set grows too large, we can produce overapproximate representations by evaluating the support function on a set of directions.
- We can overapproximate the reachable set directly by solving a linear program to evaluate the support function.

8.7 Exercises

Exercise 8.1. The mass-spring-damper system is described by the following linear equations:

$$\begin{aligned} O(\mathbf{s}, \mathbf{x}) &= \mathbf{O}_s \mathbf{s} + \mathbf{x} \\ \pi(\mathbf{o}) &= \mathbf{\Pi}_o \mathbf{o} \\ T(\mathbf{s}, \mathbf{a}) &= \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a} \end{aligned}$$

Show that the state at time step 3 is a linear function of \mathbf{s}_1 , \mathbf{x}_1 , and \mathbf{x}_2 .

Solution: First, calculate the state at time step 2:

$$\begin{aligned} \mathbf{s}_2 &= T(\mathbf{s}_1, \pi(O(\mathbf{s}_1, \mathbf{x}_1))) \\ &= \mathbf{T}_s \mathbf{s}_1 + \mathbf{T}_a \mathbf{\Pi}_o (\mathbf{O}_s \mathbf{s}_1 + \mathbf{x}_1) \\ &= (\mathbf{T}_s + \mathbf{T}_a \mathbf{\Pi}_o \mathbf{O}_s) \mathbf{s}_1 + \mathbf{T}_a \mathbf{\Pi}_o \mathbf{x}_1 \end{aligned}$$

Next, calculate the state at time step 3:

$$\begin{aligned} \mathbf{s}_3 &= T(\mathbf{s}_2, \pi(O(\mathbf{s}_2, \mathbf{x}_2))) \\ &= \mathbf{T}_s \mathbf{s}_2 + \mathbf{T}_a \mathbf{\Pi}_o (\mathbf{O}_s \mathbf{s}_2 + \mathbf{x}_2) \\ &= (\mathbf{T}_s + \mathbf{T}_a \mathbf{\Pi}_o \mathbf{O}_s) \mathbf{s}_2 + \mathbf{T}_a \mathbf{\Pi}_o \mathbf{x}_2 \end{aligned}$$

We can substitute the expression for \mathbf{s}_2 into the equation for \mathbf{s}_3 to obtain

$$\begin{aligned} \mathbf{s}_3 &= (\mathbf{T}_s + \mathbf{T}_a \mathbf{\Pi}_o \mathbf{O}_s) [(\mathbf{T}_s + \mathbf{T}_a \mathbf{\Pi}_o \mathbf{O}_s) \mathbf{s}_1 + \mathbf{T}_a \mathbf{\Pi}_o \mathbf{x}_1] + \mathbf{T}_a \mathbf{\Pi}_o \mathbf{x}_2 \\ &= (\mathbf{T}_s + \mathbf{T}_a \mathbf{\Pi}_o \mathbf{O}_s)^2 \mathbf{s}_1 + (\mathbf{T}_s + \mathbf{T}_a \mathbf{\Pi}_o \mathbf{O}_s) \mathbf{T}_a \mathbf{\Pi}_o \mathbf{x}_1 + \mathbf{T}_a \mathbf{\Pi}_o \mathbf{x}_2 \end{aligned}$$

This result is a linear function of \mathbf{s}_1 , \mathbf{x}_1 , and \mathbf{x}_2 .

Exercise 8.2. Are hyperrectangles closed under linear transformations? If so, explain why. If not, provide a counterexample.

Solution: No. Hyperrectangles are not closed under linear transformations. Figure 8.3 shows a counterexample in which a linear transformation of a hyperrectangle results in a shape that is no longer a hyperrectangle.

Exercise 8.3. Compute the resulting \mathcal{V} -polytope after applying the linear transformation

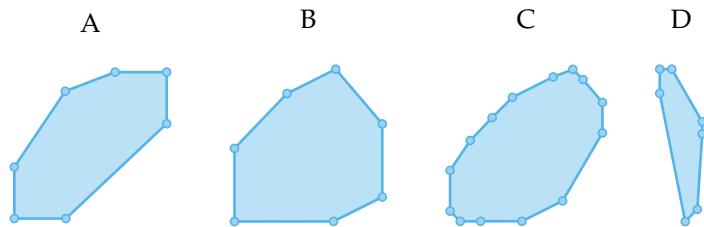
$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix} \text{ to } \mathcal{P} = \text{conv}(\{[-1, -1], [2, 1], [3, 5]\}).$$

Solution: The resulting \mathcal{V} -polytope is $\mathcal{P}' = \text{conv}(\{[-2, 2], [3, -3], [8, -8]\})$.

Exercise 8.4. Suppose we are using set propagation to compute \mathcal{R}_d from \mathcal{R}_{d-1} for a linear system. Our current representation of \mathcal{R}_{d-1} is a \mathcal{V} -polytope with 12 vertices. The agent disturbance set has 4 vertices, the environment disturbance set has 3 vertices, and the sensor disturbance set has 2 vertices. How many candidate vertices will the resulting \mathcal{V} -polytope for \mathcal{R}_d have?

Solution: Due to the three Minkowski sums in equation (8.12), the resulting \mathcal{V} -polytope will have $12 \times 4 \times 3 \times 2 = 288$ candidate vertices.

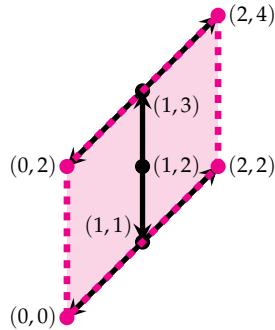
Exercise 8.5. Let \mathcal{X}_1 be a two-dimensional polytope with 3 vertices and \mathcal{X}_2 be a two-dimensional polytope with 4 vertices. Which of the following polytopes could result from the linear operation $M\mathcal{X}_1 \oplus \mathcal{X}_2$? Assume M is a 2×2 matrix.



Solution: A, B, and D. The number of candidate vertices resulting from the operation is $4 \times 3 = 12$. Polytopes A, B, and D all have fewer than 12 vertices.

Exercise 8.6. Calculate the vertices of a zonotope with center $\mathbf{c} = [1, 2]$ and generators $\mathbf{g}_1 = [0, 1]$ and $\mathbf{g}_2 = [1, 1]$.

Solution: The vertices of the zonotope are $\{(0,2), (2,4), (2,2), (0,0)\}$.



Exercise 8.7. Suppose we are using set propagation to compute \mathcal{R}_d from \mathcal{R}_{d-1} for a linear system. Our current representation of \mathcal{R}_{d-1} is a zonotope with 12 generators. The agent, environment, and sensor disturbance sets are also represented as zonotopes. The agent disturbance set has 4 generators, the environment disturbance set has 3 generators, and the sensor disturbance set has 2 generators. How many generators will the resulting zonotope for \mathcal{R}_d have?

Solution: Due to the three Minkowski sums in equation (8.12), the resulting zonotope will have $12 + 4 + 3 + 2 = 21$ generators.

Exercise 8.8. Compute the support vector for the \mathcal{V} -polytope $\mathcal{P} = \text{conv}(\{[1,1], [2,3], [3,2]\})$ in the direction $[-4,1]$.

Solution: We can compute the support vector of a \mathcal{V} -polytope by evaluating the dot product of the direction vector with each vertex of the polytope and selecting the vertex with the maximum dot product:

$$[-4,1]^\top [1,1] = -3$$

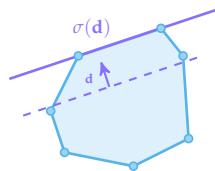
$$[-4,1]^\top [2,3] = -5$$

$$[-4,1]^\top [3,2] = -10$$

Therefore, the support vector is $[1,1]$.

Exercise 8.9. Is the support vector of a polytope in a given direction necessarily unique? If so, explain why. If not, provide a counterexample.

Solution: No, the support vector of a polytope in a given direction is not necessarily unique. If the direction vector is orthogonal to one of the edges of the polytope, all points on the edge will be valid support vectors. The plot below shows an example of this case.



Exercise 8.10. Suppose we want to compute the support vector for the reachable set of a linear system by solving the optimization problem in equation (8.27). The initial state set S for the system is represented as a zonotope with center \mathbf{c} and generators $\mathbf{g}_{1:2}$. Rewrite the constraint that $\mathbf{s}_1 \in S$ as a set of linear constraints. You may introduce new variables if necessary.

Solution: We can rewrite the initial state constraint as a set of linear constraints as follows:

$$\mathbf{s}_1 = \mathbf{c} + \alpha_1 \mathbf{g}_1 + \alpha_2 \mathbf{g}_2$$

$$-1 \leq \alpha_1 \leq 1$$

$$-1 \leq \alpha_2 \leq 1$$

where α_1 and α_2 are new decision variables for the optimization problem.

9 Reachability for Nonlinear Systems

This chapter extends the set propagation and optimization techniques discussed in chapter 8 to perform reachability on nonlinear systems. A system is nonlinear if its agent, environment, or sensor model contains nonlinear functions. The reachable sets of nonlinear systems are often nonconvex and difficult to compute exactly. This chapter begins by discussing several set propagation techniques for nonlinear systems that overapproximate the reachable set.¹ We then discuss optimization-based nonlinear reachability methods. To minimize the overapproximation error introduced by these methods, we introduce a technique for overapproximation error reduction that involves partitioning the state space. We conclude by discussing reachability techniques for nonlinear systems represented by a neural network.

9.1 Interval Arithmetic

For nonlinear systems, the reachability function $r(\mathbf{s}, \mathbf{x}_{1:d})$ is a nonlinear function. In contrast with the linear systems in chapter 8, we cannot directly propagate arbitrary polytopes through nonlinear systems. We can, however, propagate hyperrectangular sets² using a technique called *interval arithmetic*.³ Interval arithmetic extends traditional arithmetic operations and other elementary functions to intervals. An interval is a set of real numbers written as

$$[x] = [\underline{x}, \bar{x}] = \{x \mid \underline{x} \leq x \leq \bar{x}\} \quad (9.1)$$

where \underline{x} and \bar{x} are the lower and upper bounds of the interval, respectively. A hyperrectangle, also known as an *interval box*, is the Cartesian product of a set of n intervals:

$$[\mathbf{x}] = [x_1] \times [x_2] \times \cdots \times [x_n] \quad (9.2)$$

¹ For more details on set propagation through nonlinear systems, refer to M. Althoff, G. Frehse, and A. Girard, “Set Propagation Techniques for Reachability Analysis,” *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 4, pp. 369–395, 2021.

² We can also propagate sets that are linear transformations of hyperrectangles by reversing the linear transformation to obtain an axis-aligned hyperrectangle and performing the analysis in the transformed space.

³ L. Jaulin, M. Kieffer, O. Didrit, and É. Walter, *Interval Analysis*. Springer, 2001.

where $[x_i] = [\underline{x}_i, \bar{x}_i]$ for $i \in 1 : n$ (figure 9.1).

Given two intervals $[x]$ and $[y]$, we define the *interval counterpart* of elementary arithmetic functions as

$$[x] \circ [y] = \{x \circ y \mid x \in [x], y \in [y]\} \quad (9.3)$$

where \circ represents the addition, subtraction, multiplication, and division operations. We evaluate the interval counterparts of these functions as follows:

$$[x] + [y] = [\underline{x} + \underline{y}, \bar{x} + \bar{y}] \quad (9.4)$$

$$[x] - [y] = [\underline{x} - \bar{y}, \bar{x} - \underline{y}] \quad (9.5)$$

$$[x] \times [y] = [\min(\underline{x}\underline{y}, \underline{x}\bar{y}, \bar{x}\underline{y}, \bar{x}\bar{y}), \max(\underline{x}\underline{y}, \underline{x}\bar{y}, \bar{x}\underline{y}, \bar{x}\bar{y})] \quad (9.6)$$

$$[x] / [y] = [\min(x/\underline{y}, \underline{x}/\bar{y}, \bar{x}/\underline{y}, \bar{x}/\bar{y}), \max(x/\underline{y}, \underline{x}/\bar{y}, \bar{x}/\underline{y}, \bar{x}/\bar{y})] \quad (9.7)$$

where the division operation is only defined when $0 \notin [y]$.

In general, we define the interval counterpart of a given function $f(x)$ as

$$f([x]) = [\{f(x) \mid x \in [x]\}] \quad (9.8)$$

where the $[\cdot]$ operation takes the *interval hull* of the resulting set. The interval hull of a set is the smallest interval that contains the set. Therefore, the interval counterpart of a function returns the smallest interval that contains all possible function evaluations of the points in the input interval.

We can define an interval counterpart for a variety of elementary functions.⁴ For monotonically increasing functions such as \exp , \log , and square root, the interval counterpart is

$$f([x]) = [f(\underline{x}), f(\bar{x})] \quad (9.9)$$

The interval counterpart for monotonically decreasing functions is similarly defined. Nonmonotonic elementary functions such as \sin , \cos , and square require multiple cases to define their interval counterparts. For example, the interval counterpart for the square function is

$$[x]^2 = \begin{cases} [\min(\underline{x}^2, \bar{x}^2), \max(\underline{x}^2, \bar{x}^2)] & \text{if } 0 \notin [x] \\ [0, \max(\underline{x}^2, \bar{x}^2)] & \text{otherwise} \end{cases} \quad (9.10)$$

Figure 9.2 shows example evaluations of the interval counterparts for the \exp , square, and \sin functions.

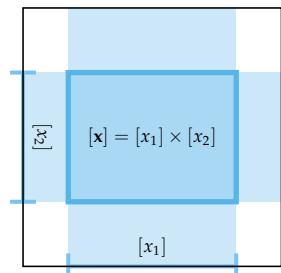


Figure 9.1. The Cartesian product of two intervals forms a hyperrectangle in \mathbb{R}^2 .

⁴ `IntervalArithmetic.jl` defines the interval counterpart of many elementary functions such as `sin`, `cos`, `exp`, and `log` in Julia.

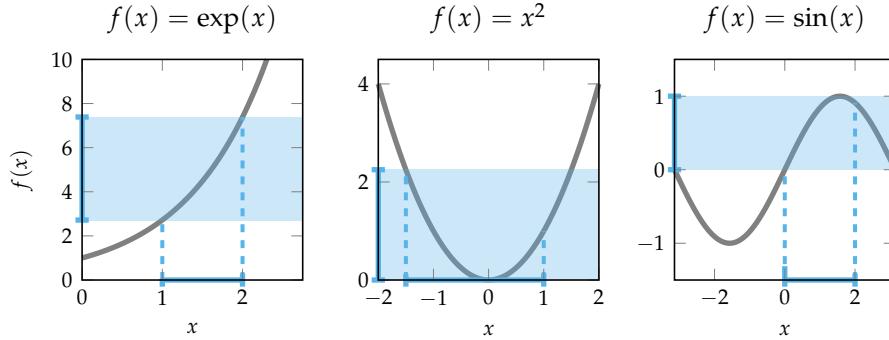


Figure 9.2. Example of the interval counterparts for the exp, square, and sin functions.

9.2 Inclusion Functions

For complex functions, it is not always possible to define a tight interval counterpart. In these cases, we instead define an *inclusion function*. An inclusion function $[f]([x])$ outputs an interval that is guaranteed to contain the interval from the interval counterpart:

$$f([x]) \subseteq [f]([x]) \quad (9.11)$$

In other words, inclusion functions output overapproximate intervals. We can also define an inclusion function for multivariate functions that map from \mathbb{R}^k to \mathbb{R} where $k \geq 1$.

For reachability analysis, our goal is to propagate intervals through the function $r(\mathbf{s}, \mathbf{x}_{1:d})$, which maps its inputs to \mathbb{R}^n where n is the dimension of the state space. We can rewrite $r(\mathbf{s}, \mathbf{x}_{1:d})$ as a vector of functions that map to \mathbb{R} as follows:

$$\mathbf{s}' = r(\mathbf{s}, \mathbf{x}_{1:d}) = \begin{bmatrix} r_1(\mathbf{s}, \mathbf{x}_{1:d}) \\ \vdots \\ r_n(\mathbf{s}, \mathbf{x}_{1:d}) \end{bmatrix} \quad (9.12)$$

where $r_i(\mathbf{s}, \mathbf{x}_{1:d})$ outputs the value of the i th component of \mathbf{s}' . We can then define the inclusion function for each $r_i(\mathbf{s}, \mathbf{x}_{1:d})$ as $[r_i]([\mathbf{s}], [\mathbf{x}_{1:d}])$. By evaluating each inclusion function for the input intervals $[\mathbf{s}]$ and $[\mathbf{x}_{1:d}]$, we obtain an overapproximate hyperrectangular reachable set. The rest of this section discusses techniques to create these inclusion functions.

9.2.1 Natural Inclusion Functions

One simple way to create an inclusion function from a complex function is to replace each elementary function with its interval counterpart. This type of inclusion function is known as a *natural inclusion function*. For example, the natural inclusion function for $f(x) = x - \sin(x)$ is $[f](x) = [x] - \sin([x])$ (figure 9.3).

By replacing the elementary nonlinear components of the agent, environment, and sensor models with their interval counterparts, we can create the natural inclusion function for $r_i(s, x_{1:d})$. We can then use interval arithmetic to propagate hyperrectangular sets through the natural inclusion function. This computation will result in overapproximate reachable sets for nonlinear systems. Algorithm 9.1 implements the natural inclusion reachability algorithm and computes overapproximate reachable sets up to a desired time horizon. Example 9.1 applies algorithm 9.1 to the inverted pendulum problem.

As shown in figure 9.3 and example 9.1, natural inclusion functions tend to be overly conservative. This property is due to the dependency effect, in which multiple occurrences of the same variable are treated independently (see example 8.2). In chapter 8, we were able to eliminate this effect by simplifying equations to algebraically combine all repeated instances of a variable. However, this simplification is not always possible for nonlinear functions such as the one shown in figure 9.3. We can instead mitigate the dependency effect by using more sophisticated techniques for generating inclusion functions, which we discuss in the remainder of this section.

9.2.2 Mean Value Inclusion Functions

For functions that are continuous and differentiable, we can use the *mean value theorem* to create an inclusion function. The mean value theorem states that for a function $f(x)$ that is continuous and differentiable on the interval $[x]$, there exists a point $x' \in [x]$ such that

$$\frac{f(\bar{x}) - f(\underline{x})}{\bar{x} - \underline{x}} = f'(x') \quad (9.13)$$

In other words, there exists a point in $[x]$ where the slope of the tangent line is equal to the slope of the secant line between the endpoints of the interval (figure 9.4).

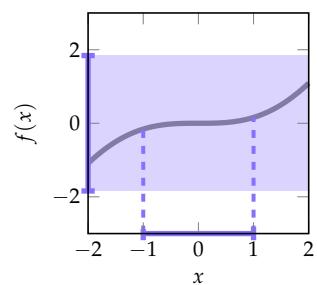


Figure 9.3. Example evaluation of the natural inclusion function for $f(x) = x - \sin(x)$. The inclusion function produces an overapproximate interval.

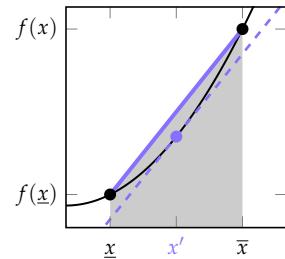


Figure 9.4. Illustration of the mean value theorem on the function $f(x) = x^2$ over the interval $[x] = [1, 4]$.

```

struct NaturalInclusion <: ReachabilityAlgorithm
    h # time horizon
end

function r(sys, x)
    s, x = extract(sys.env, x)
    t = rollout(sys, s, x)
    return t[end].s
end

to_hyperrectangle(I) = Hyperrectangle(low=[i.lo for i in I],
                                         high=[i.hi for i in I])

function reachable(alg::NaturalInclusion, sys)
    I's = []
    for d in 1:alg.h
        I = intervals(sys, d)
        push!(I's, r(sys, I))
    end
    return UnionSetArray([to_hyperrectangle(I') for I' in I's])
end

```

Suppose we want to compute reachable sets for the pendulum system with bounded sensor noise on the angle and angular velocity using algorithm 9.1. We define the `intervals` and `extract` functions as follows:

```

function intervals(sys, d)
    disturbance_mag = 0.01
    θmin, θmax = -π/16, π/16
    ωmin, ωmax = -1.0, 1.0
    I = [interval(θmin, θmax), interval(ωmin, ωmax)]
    for i in 1:2d
        push!(I, interval(-disturbance_mag, disturbance_mag))
    end
    return I
end

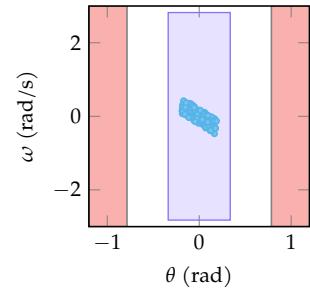
function extract(env::InvertedPendulum, x)
    s = x[1:2]
    x = [Disturbance(0, 0, x[i:i+1]) for i in 3:2:length(x)]
    return s, x
end

```

The `intervals` function returns the initial state intervals followed by the disturbance intervals for each time step. The `extract` function extracts these intervals into the state and disturbance components. The plot in the caption shows the overapproximated reachable set after two time steps.

Algorithm 9.1. Nonlinear forward reachability using natural inclusion functions. For each depth, the algorithm gets the input intervals using the system specific `intervals` function and computes the output intervals using the natural inclusion function of $r(s, x_{1:d})$. The `IntervalArithmetic.jl` package replaces functions with their interval counterparts so that we can propagate the intervals directly through the `rollout` function. The algorithm returns $\mathcal{R}_{1:h}$ as the union of the output intervals.

Example 9.1. Computing reachable sets for the inverted pendulum system using its natural inclusion function. The plot shows the overapproximated reachable set \mathcal{R}_2 computed using algorithm 9.1 compared to a set of samples from \mathcal{R}_2 .



The mean value theorem implies that for any subinterval of $[x]$, there exists a point in $[x]$ where the slope of the tangent line is equal to the slope of the secant line between the endpoints of the subinterval. Therefore, given the center c of the interval $[x]$, there exists a point $x' \in [x]$ such that

$$\frac{f(x) - f(c)}{x - c} = f'(x') \quad (9.14)$$

for any $x \in [x]$ (figure 9.5). Rearranging equation (9.14) gives

$$f(x) = f(c) + f'(x')(x - c) \quad (9.15)$$

Because we know that $x' \in [x]$, we can use equation (9.15) to create an inclusion function for $f(x)$ as follows:

$$[f]([x]) = f(c) + [f']([x])([x] - c) \quad (9.16)$$

where $[f']([x])$ is an inclusion function for $f'(x)$. It is common to define $[f']([x])$ as the natural inclusion function for $f'(x)$. For multivariate functions, equation (9.16) generalizes to

$$[f]([x]) = f(\mathbf{c}) + [\nabla f]([x])^\top ([x] - \mathbf{c}) \quad (9.17)$$

where \mathbf{c} is the center of the interval $[x]$ and $[\nabla f]([x])$ is an inclusion function for the gradient of $f(x)$.

Equation (9.17) is a *linearization* of the nonlinear function $f(x)$. Therefore, mean value inclusion functions tend to perform well when the input interval covers a region of the input space for which the function is nearly linear. Figure 9.6 shows an evaluation of the mean value inclusion function for the function in figure 9.3. Because the function is roughly linear over the input interval, the mean value inclusion function provides a tighter overapproximation. However, if we expand the input interval to include nonlinear regions, the mean value inclusion function produces more conservative results (figure 9.7).

9.2.3 Taylor Inclusion Functions

Natural inclusion functions and mean value inclusion functions are special cases of a more general type of inclusion function known as a *Taylor inclusion function*. These inclusion functions use Taylor series expansions about the center of the

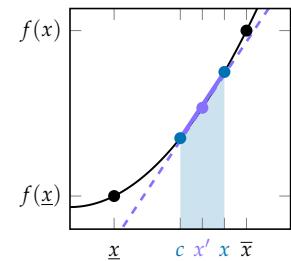


Figure 9.5. For a given subinterval $[c, x]$, there exists a point in $[x]$ where the slope of the tangent line is equal to the slope of the secant line between the endpoints of the subinterval.

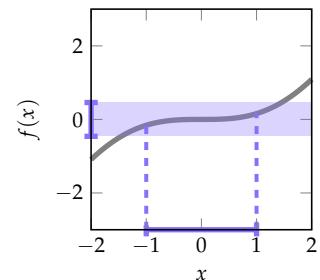


Figure 9.6. Mean value inclusion function for $f(x) = x - \sin(x)$ over the same interval as figure 9.3.

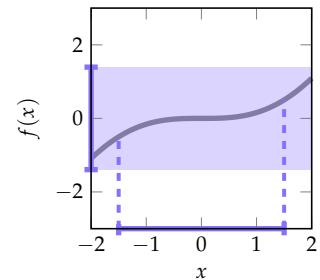


Figure 9.7. Mean value inclusion function for $f(x) = x - \sin(x)$ over a wider interval.

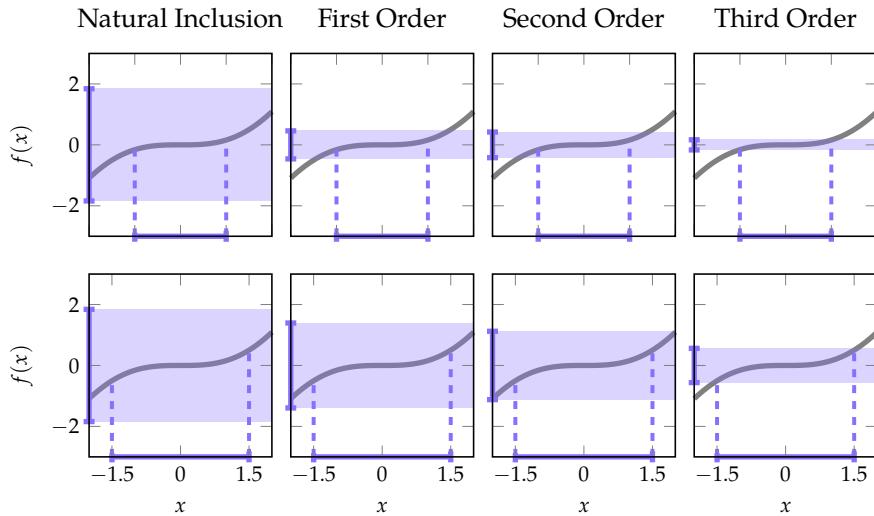


Figure 9.8. Evaluation of Taylor inclusion functions of different orders for $f(x) = x - \sin(x)$ over the interval $[x] = [-1, 1]$ (top row) and $[x] = [-1.5, 1.5]$ (bottom row). The natural inclusion function is equivalent to the zeroth-order Taylor inclusion function. As the order increases, overapproximation error decreases, especially over the wider input interval.

input interval. In one dimension, a Taylor inclusion function of order n for a function $f(x)$ is defined as

$$[f]([x]) = f(c) + f'(c)([x] - c) + \frac{f''(c)}{2!}([x] - c)^2 + \cdots + \frac{[f^{(n)}]([x])}{n!}([x] - c)^n \quad (9.18)$$

where c is the center of the interval $[x]$ and $[f^{(n)}]([x])$ is an inclusion function for the n th-order derivative of $f(x)$.⁵

Taylor inclusion functions can be similarly defined for multivariate functions. The second-order Taylor inclusion function for a multivariate function $f(\mathbf{x})$ is

$$[f]([\mathbf{x}]) = f(\mathbf{c}) + \nabla f(\mathbf{c})^\top ([\mathbf{x}] - \mathbf{c}) + \frac{1}{2}([\mathbf{x}] - \mathbf{c})^\top [\nabla^2 f]([\mathbf{x}])([\mathbf{x}] - \mathbf{c}) \quad (9.19)$$

where \mathbf{c} is the center of the interval $[\mathbf{x}]$ and $[\nabla^2 f]([\mathbf{x}])$ is an inclusion function for the Hessian of $f(\mathbf{x})$.⁶ A zero-order Taylor inclusion function is equivalent to the natural inclusion function, and a first-order Taylor inclusion function is equivalent to the mean value inclusion function.

In general, higher-order Taylor inclusion functions provide tighter overapproximations (figure 9.8). However, the benefit of using higher-order terms depends on the behavior of the function over the input interval. If the function is nearly linear over the input interval, moving beyond a first-order model may not be

⁵ It is possible to create a Taylor inclusion function centered around any point in the interval. However, choosing the center of the interval minimizes overapproximation error.

⁶ For higher order models, see R. Neidinger, "Directions for Computing Truncated Multivariate Taylor Series," *Mathematics of Computation*, vol. 74, no. 249, pp. 321–340, 2005.

worth the additional computational cost. In contrast, if the function is highly nonlinear over the input interval, a higher-order model may significantly decrease overapproximation error.

Algorithm 9.2 implements first- and second-order Taylor inclusion functions for reachability analysis. The algorithm computes overapproximate reachable sets up to a desired time horizon by evaluating the Taylor inclusion function for each subfunction $r_i(\mathbf{s}, \mathbf{x}_{1:d})$. Taylor inclusion functions can be used to create tighter overapproximations of the reachable set than natural inclusion functions, especially for short time horizons (figure 9.9).⁷ However, the nonlinearities compound for each time step, so Taylor inclusion functions can be computationally expensive and result in significant overapproximation error for long time horizons (example 9.2).

```

struct TaylorInclusion <: ReachabilityAlgorithm
    h      # time horizon
    order # order of Taylor inclusion function (supports 1 or 2)
end

function taylor_inclusion(sys, I, order)
    c = mid.(I)
    fc = r(sys, c)
    if order == 1
        I' = [fc[i] + gradient(x->r(sys, x)[i], I)' * (I - c)
              for i in eachindex(fc)]
    else
        I' = [fc[i] + gradient(x->r(sys, x)[i], c)' * (I - c) +
               (I - c)' * hessian(x->r(sys, x)[i], I) * (I - c)
              for i in eachindex(fc)]
    end
    return I'
end

function reachable(alg::TaylorInclusion, sys)
    I's = []
    for d in 1:alg.h
        I = intervals(sys, d)
        I' = taylor_inclusion(sys, I, alg.order)
        push!(I's, I')
    end
    return UnionSetArray([to_hyperrectangle(I') for I' in I's])
end
```

⁷ Because Taylor inclusion functions can only be applied to functions that are continuous and differentiable, we use a modified version of the pendulum problem in this chapter that does not apply clamping in the environment model.

Algorithm 9.2. Nonlinear forward reachability using first- or second-order Taylor inclusion functions. For each depth, the algorithm gets the input intervals using the system specific `intervals` function and applies either equation (9.17) or equation (9.19) to each subfunction $r_i(\mathbf{s}, \mathbf{x}_{1:d})$ of $r(\mathbf{s}, \mathbf{x}_{1:d})$. The `IntervalArithmetic.jl` and `ForwardDiff.jl` packages are compatible, which allows us to evaluate gradients and Hessians over intervals. The algorithm returns $\mathcal{R}_{1:h}$ as the union of the output intervals.

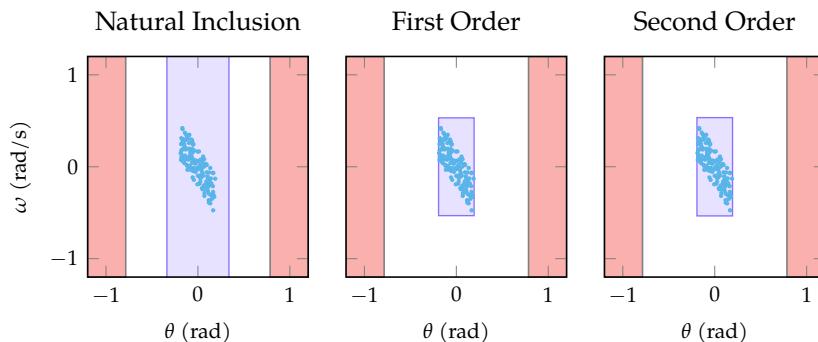
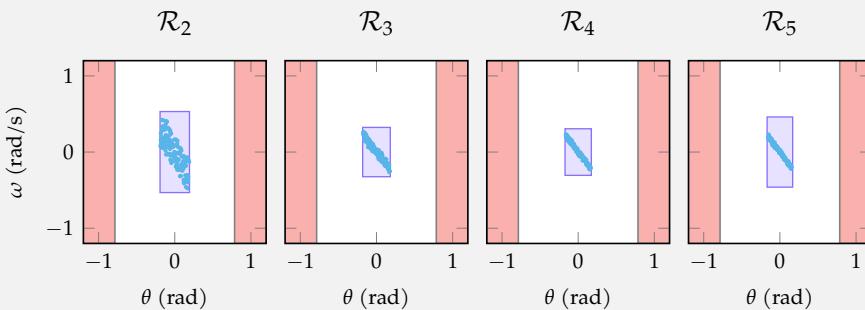


Figure 9.9. Comparison of the one-step overapproximated reachable sets for the inverted pendulum system using natural, first-order Taylor, and second-order Taylor inclusion functions. In this particular example, a first-order Taylor inclusion function provides a significantly tighter overapproximation than the natural inclusion function. The second-order Taylor inclusion function does not provide a significant benefit over the first-order Taylor inclusion function, indicating that the dynamics are roughly linear over the input space.

The plots below show the overapproximate reachable sets for the inverted pendulum system produced by a first-order Taylor inclusion function at different depths. As the depth increases, the overapproximation error increases. This result is due to the increasing presence of nonlinearities in the system dynamics as we increase the depth. For the one-step reachable set (\mathcal{R}_2), the only nonlinearity present is the sine function in the pendulum dynamics. As the depth increases, this nonlinearity will be repeated for each time step, leading to larger overapproximation error.



Example 9.2. Overapproximate reachable sets for the inverted pendulum system using first-order Taylor inclusion functions at different depths. As the depth increases, the overapproximation error increases.

9.3 Taylor Models

While inclusion functions only operate over interval inputs and output reachable sets in the form of hyperrectangles, *Taylor models* operate over other types of input sets and are able to represent more expressive reachable sets.⁸ Similar to Taylor inclusion functions, Taylor models are based on Taylor series expansions. An n th-order Taylor model is a set represented as

$$\mathcal{T} = \{p(\mathbf{x}) + \boldsymbol{\alpha} \mid \mathbf{x} \in \mathcal{X}, \boldsymbol{\alpha} \in [\boldsymbol{\alpha}]\} \quad (9.20)$$

where \mathcal{X} is the input set, $p(\mathbf{x})$ is a polynomial of degree $n - 1$, and $[\boldsymbol{\alpha}]$ is an interval remainder term. In one dimension, the polynomial of an n th-order Taylor model for the function $f(x)$ over an input interval $[x]$ is defined as

$$p(x) = f(c) + f'(c)(x - c) + \frac{f''(c)}{2!}(x - c)^2 + \cdots + \frac{f^{(n-1)}(c)}{(n-1)!}(x - c)^{(n-1)} \quad (9.21)$$

where c is the center of the input interval. The interval remainder term, also known as the *Lagrange remainder*, bounds the sum of the rest of the terms in the Taylor expansion over the input interval $[x]$ so that the Taylor model is guaranteed to contain the true output of the function. It is calculated as

$$[\boldsymbol{\alpha}] = \frac{[f^{(n)}]([x])}{n!}([x] - c)^n \quad (9.22)$$

and is equivalent to the last term in a Taylor inclusion function of order n . In fact, passing an interval through a Taylor model performs the same computation as a Taylor inclusion function of the same order.

As the order of a Taylor model increases, overapproximation error tends to decrease (figure 9.10). Producing a zero-order Taylor model is equivalent to evaluating the natural inclusion function, while producing a first-order Taylor model is equivalent to evaluating the mean value inclusion function. Taylor models begin to deviate from inclusion functions for orders of two or higher. Second-order Taylor models represent arbitrary polytopes, while second-order inclusion functions only produce hyperrectangles. Higher-order Taylor models correspond to nonconvex sets, which are more difficult to understand and manipulate.⁹ For this reason, we focus the remainder of this section on second-order Taylor models.

Creating a second-order Taylor model for a function $f(\mathbf{x})$ is a process known as *conservative linearization*.¹⁰ Given an input set \mathcal{X} and a center point \mathbf{c} , the second-

⁸ K. Makino and M. Berz, "Taylor Models and Other Validated Functional Inclusion Methods," *International Journal of Pure and Applied Mathematics*, vol. 4, no. 4, pp. 379–456, 2003.

⁹ One way to handle this nonconvexity is to represent sets using an extension of zonotopes called *polynomial zonotopes*. More details can be found in M. Althoff, "Reachability Analysis of Nonlinear Systems Using Conservative Polynomialization and Non-Convex Sets," in *International Conference on Hybrid Systems: Computation and Control*, 2013. Another representation called *star sets* can also be used to represent nonconvex sets and has been used for reachability. H.-D. Tran, D. Manzanas Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, and T. T. Johnson, "Star-Based Reachability Analysis of Deep Neural Networks," in *International Symposium on Formal Methods*, 2019.

¹⁰ M. Althoff, O. Stursberg, and M. Buss, "Reachability Analysis of Nonlinear Systems with Uncertain Parameters Using Conservative Linearization," in *IEEE Conference on Decision and Control (CDC)*, 2008.

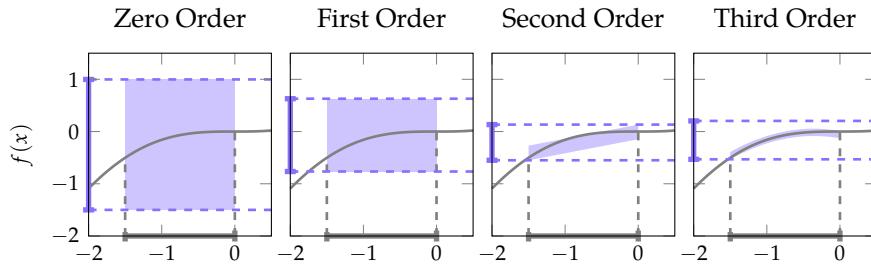


Figure 9.10. Taylor models of different orders for $f(x) = x - \sin(x)$ over the interval $[x] = [-1.5, 0.0]$. The dashed purple lines show results from a Taylor inclusion function of the same order.

order Taylor model is

$$\mathcal{T} = \{f(\mathbf{c}) + \mathbf{J}(\mathbf{x} - \mathbf{c}) + \boldsymbol{\alpha} \mid \mathbf{x} \in \mathcal{X}, \boldsymbol{\alpha} \in [\boldsymbol{\alpha}]\} \quad (9.23)$$

where \mathbf{J} is the Jacobian of f evaluated at \mathbf{c} and $[\boldsymbol{\alpha}]$ is the interval remainder term. The Jacobian is a generalization of the gradient to functions with multidimensional outputs and is computed as

$$\mathbf{J} = \begin{bmatrix} \nabla f_1(\mathbf{c})^\top \\ \vdots \\ \nabla f_n(\mathbf{c})^\top \end{bmatrix} \quad (9.24)$$

where $\nabla f_i(\mathbf{c})$ is the gradient of the i th component of f evaluated at \mathbf{c} . The interval remainder term is calculated using interval arithmetic as

$$[\boldsymbol{\alpha}] = \frac{1}{2}([\mathcal{X}] - \mathbf{c})^\top [\nabla^2 f](\mathcal{X}) ([\mathcal{X}] - \mathbf{c}) \quad (9.25)$$

where $[\mathcal{X}]$ is the interval hull of \mathcal{X} .¹¹

Equation (9.23) represents a linear approximation of the nonlinear function f with a remainder term that bounds the error of the approximation. Because all of the operations in equation (9.23) are linear, we can use it to propagate convex sets. In other words, if \mathcal{X} is convex, we can rewrite the Taylor model in terms of linear transformations and Minkowski sums as

$$\mathcal{T} = f(\mathbf{c}) + \mathbf{J}(\mathcal{X} \oplus -\mathbf{c}) \oplus [\boldsymbol{\alpha}] \quad (9.26)$$

¹¹ If the input set \mathcal{X} is represented as a zonotope, it is also possible to overapproximate the remainder term directly without taking the interval hull. This approach can reduce overapproximation error. M. Althoff, O. Stursberg, and M. Buss, “Reachability Analysis of Nonlinear Systems with Uncertain Parameters Using Conservative Linearization,” in *IEEE Conference on Decision and Control (CDC)*, 2008.

Algorithm 9.3 computes overapproximate reachable sets using conservative linearization. Since Taylor models can be applied to functions with multidimensional outputs, we can apply conservative linearization directly to the reachability function $r(\mathbf{s}, \mathbf{x}_{1:d})$ without the need to break it into subfunctions. Example 9.3 demonstrates algorithm 9.3 on the inverted pendulum system. Conservative linearization using Taylor models performs better than second-order Taylor inclusion functions because it is able to output more expressive reachable sets. However, for higher orders, Taylor models output nonconvex sets that are difficult to manipulate. In contrast, Taylor inclusion functions always output hyperrectangles and do not suffer from this added complexity.

```

struct ConservativeLinearization <: ReachabilityAlgorithm
    h # time horizon
end

to_intervals(P) = [interval(lo, hi) for (lo, hi) in zip(low(P), high(P))]

function conservative_linearization(sys, P)
    I = to_intervals(interval_hull(P))
    c = mid(I)
    fc = r(sys, c)
    J = ForwardDiff.jacobian(x→r(sys, x), c)
    α = to_hyperrectangle([(I - c)'*hessian(x→r(sys, x))[i], I)*(I - c)
        for i in eachindex(fc)]
    return fc + J * (P ⊕ -c) ⊕ α
end

function reachable(alg::ConservativeLinearization, sys)
    Rs = []
    for d in 1:alg.h
        S, X = sets(sys, d)
        S' = conservative_linearization(sys, S × X)
        push!(Rs, S')
    end
    return UnionSetArray([Rs...])
end

```

Algorithm 9.3. Nonlinear forward reachability using conservative linearization. At each depth, the algorithm gets the input sets for the initial states and disturbances using the system specific **sets** function and applies equation (9.23) to $r(\mathbf{s}, \mathbf{x}_{1:d})$. It uses the interval hull of the input set to calculate the interval remainder term. The algorithm returns $\mathcal{R}_{1:h}$ as the union of the output sets.

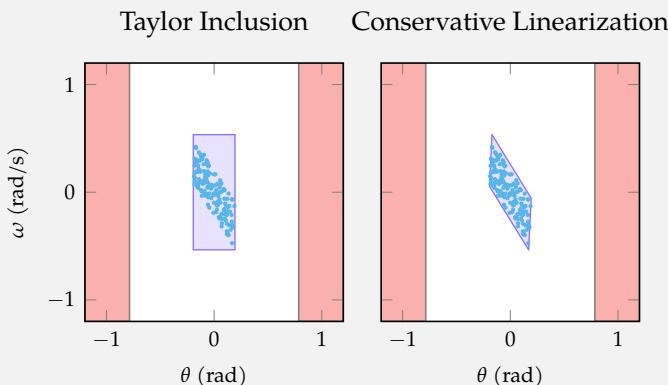
9.4 Concrete Reachability

Algorithms 9.2 and 9.3 tend to be computationally expensive when computing reachable sets over long time horizons. As the depth d increases, the input dimension for the reachability function $r(\mathbf{s}, \mathbf{x}_{1:d})$ also increases. This increase in

Suppose we want to compute reachable sets for the pendulum system with bounded sensor noise on the angle and angular velocity using algorithm 9.3. We define the `sets` function as follows:

```
function sets(sys, d)
    disturbance_mag = 0.01
    θmin, θmax = -π/16, π/16
    ωmin, ωmax = -1.0, 1.0
    S = Hyperrectangle(low=[θmin, ωmin], high=[θmax, ωmax])
    low = fill(-disturbance_mag, 2d)
    high = fill(disturbance_mag, 2d)
    X = Hyperrectangle(low=low, high=high)
    return S, X
end
```

The `sets` function returns the initial state set followed by the disturbance sets for each time step. The plots below compare the one-step reachable set produced by conservative linearization with the set produced by a second-order Taylor inclusion function. While conservative linearization still produces an overapproximation, it captures the shape of the true reachable set better than a Taylor inclusion function.



Example 9.3. Computing the one-step reachable set for the inverted pendulum system using conservative linearization. Conservative linearization better approximates the reachable set than a second-order Taylor inclusion function.

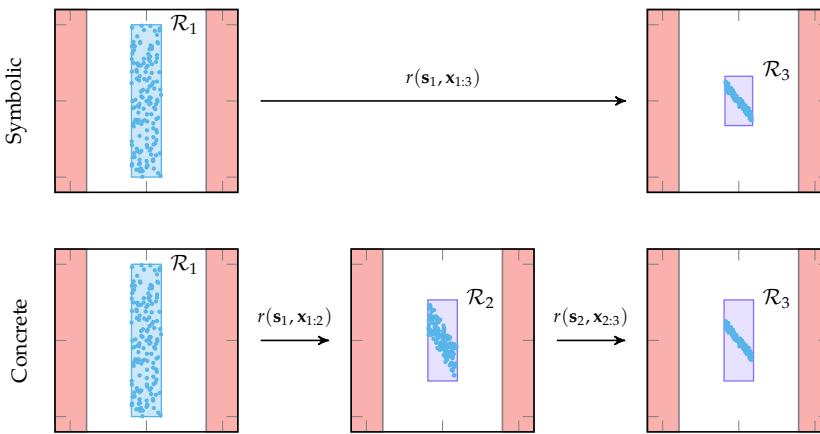


Figure 9.11. Comparison of symbolic and concrete reachability algorithms when computing \mathcal{R}_3 for the inverted pendulum system. The symbolic reachability algorithm directly computes \mathcal{R}_3 without explicitly computing \mathcal{R}_2 by considering $r(\mathbf{s}_1, \mathbf{x}_{1:3})$ as a single function. The concrete reachability algorithm computes \mathcal{R}_2 and \mathcal{R}_3 separately by considering $r(\mathbf{s}_1, \mathbf{x}_{1:2})$ and $r(\mathbf{s}_2, \mathbf{x}_{2:3})$ as separate functions. It uses \mathcal{R}_2 as the input set for computing \mathcal{R}_3 .

input dimension causes the size of the gradient and Hessian to increase, leading to more expensive computations. Furthermore, the nonlinearities in the agent, environment, and sensor models compound over time, causing the accuracy of a linearized model to degrade as the depth increases.

Concrete reachability algorithms address these issues by decomposing the reachability function into a sequence of simpler functions. Instead of overapproximating the reachable set over the entire depth at once, they compute the overapproximate reachable set for each time step individually. At each iteration, they use the overapproximate reachable set from the previous time step as the input set for the next time step. We refer to this process as concrete reachability because we *concretize* the reachable set at each time step by explicitly computing an overapproximate representation. In contrast, the algorithms presented thus far maintain a symbolic representation of the reachable set at each time step and only concretize the reachable set at depth d . For this reason, we refer to these algorithms as *symbolic reachability* algorithms. Figure 9.11 illustrates the difference between symbolic and concrete reachability algorithms.

Algorithms 9.4 and 9.5 implement concrete versions of the symbolic reachability algorithms presented in algorithms 9.2 and 9.3, respectively. For each depth in the time horizon, they compute the overapproximate reachable set for the next step using the overapproximate reachable set from the previous step. Algorithm 9.4 concretizes the reachable set into a hyperrectangle at each time

```

struct ConcreteTaylorInclusion <: ReachabilityAlgorithm
    h # time horizon
    order # order of Taylor inclusion function (supports 1 or 2)
end

function reachable(alg::ConcreteTaylorInclusion, sys)
    I = intervals(sys, 2)
    s, _ = extract(sys.env, I)
    I's = [s]
    for d in 2:alg.h
        I' = taylor_inclusion(sys, I, alg.order)
        push!(I's, I')
        s, _ = extract(sys.env, I)
        I[1:length(s)] = s
    end
    return UnionSetArray([to_hyperrectangle(I') for I' in I's])
end

```

Algorithm 9.4. Nonlinear forward reachability using Taylor inclusion functions, concretizing the reachable set at each time step. The algorithm first gets the intervals for a depth of 2, which correspond to the intervals for a one-step reachability computation. At each depth, it computes the intervals for the next time step and creates the input for the next time step by extracting the new state. The algorithms return $\mathcal{R}_{1:h}$ as the union of the output sets.

```

struct ConcreteConservativeLinearization <: ReachabilityAlgorithm
    h # time horizon
end

function reachable(alg::ConcreteConservativeLinearization, sys)
    S, X = sets(sys, 2)
    Rs = []
    push!(Rs, S)
    for d in 2:alg.h
        S = conservative_linearization(sys, S × X)
        push!(Rs, S)
    end
    return UnionSetArray([Rs...])
end

```

Algorithm 9.5. Nonlinear forward reachability using conservative linearization, concretizing the reachable set at each time step. The algorithm first gets the state and disturbance sets for a depth of 2, which correspond to the sets required for a one-step reachability computation. At each depth, it computes the state set for the next time step and uses it to compute the next reachable set. The algorithms return $\mathcal{R}_{1:h}$ as the union of the output sets.

step, while algorithm 9.5 concretizes the reachable set into a polytope at each time step.

Concrete reachability algorithms are generally more computationally efficient than symbolic reachability algorithms. However, it is not always clear whether they will produce tighter overapproximations because there are multiple factors that contribute to the overapproximation error. The only source of overapproximation error in symbolic reachability algorithms is the error introduced by linearizing the reachability function and bounding the remainder term. We expect this linearization error to be smaller for concrete reachability algorithms because they linearize over a single time step rather than the entire time horizon.

While concrete reachability algorithms reduce overapproximation error due to linearization, they introduce additional overapproximation error by concretizing the reachable set at each time step into an overapproximate reachable set (figure 9.11). This error compounds over time, and the accumulation of this error is often referred to as the *wrapping effect*.

The decrease in linearization error and introduction of the wrapping effect for concrete reachability algorithms result in a tradeoff between concrete and symbolic reachability (figures 9.12 and 9.13). The choice of which type of algorithm to use depends on the specific system, the reachability algorithm, and the desired tradeoff between computational efficiency and overapproximation error. For example, if we are using linearized models for reachability and the one-step reachability function is nearly linear, concrete reachability algorithms may produce tighter overapproximations than symbolic reachability algorithms. It is common to mix concrete and symbolic reachability algorithms to take advantage of the strengths of each approach. For example, instead of concretizing the reachable set at each time step, we can concretize the reachable set every k time steps to reduce the wrapping effect.

Another benefit of using concrete reachability algorithms is that we can use them to check for invariant sets. Similar to the check for invariance described for the set propagation techniques in section 8.2, if we find that the reachable set at a given time step is contained within the concrete reachable set at the previous time step, we can conclude that the reachable set is invariant. For example, the concrete versions of \mathcal{R}_6 in figures 9.12 and 9.13 are contained within the concrete versions of \mathcal{R}_5 . Therefore, we can conclude that \mathcal{R}_6 is an invariant set in both cases, meaning that the system will remain within the set for all future time steps.

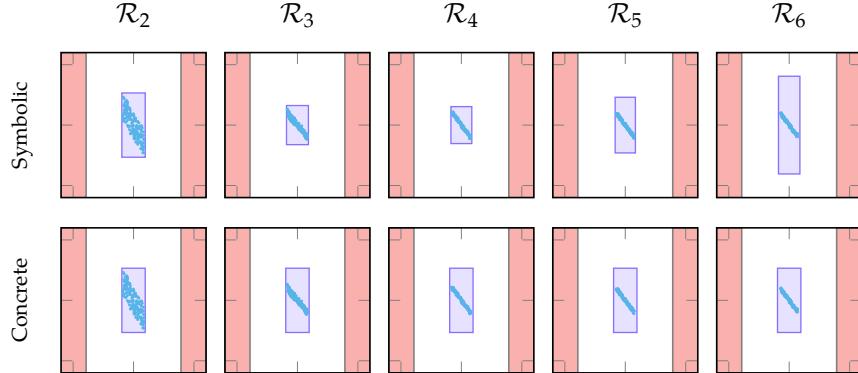


Figure 9.12. Comparison of symbolic and concrete Taylor inclusion algorithms when computing $\mathcal{R}_{1:6}$ for the inverted pendulum system. Up to a depth of 5, the concretization error dominates, so the symbolic algorithm produces tighter overapproximations. However, at a depth of 6, the linearization error dominates, and the concrete algorithm produces a tighter overapproximation.

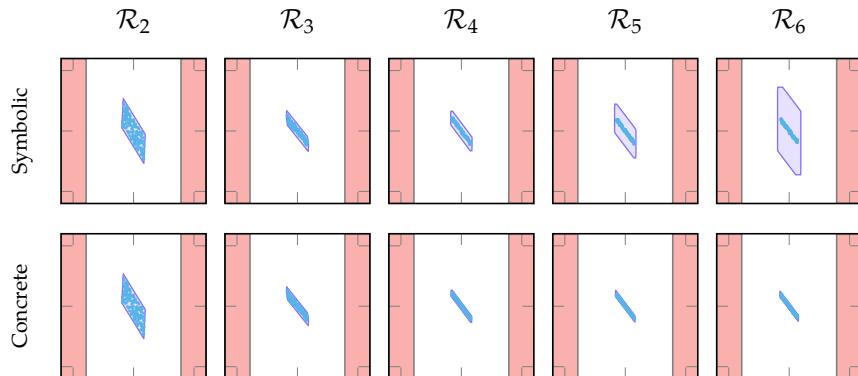


Figure 9.13. Comparison of symbolic and concrete conservative linearization algorithms when computing $\mathcal{R}_{1:6}$ for the inverted pendulum system. Because concretization using polytopes does not introduce as much error as the hyperrectangles used by the Taylor inclusion functions in figure 9.12, the linearization error dominates, and the concrete algorithm produces tighter overapproximations.

We cannot draw this conclusion using symbolic reachability algorithms because the property requires consecutive concrete steps.

9.5 Optimization-Based Nonlinear Reachability

Similar to the ideas in section 8.5, we can overapproximate the reachable set of nonlinear systems by sampling the support function. For symbolic reachability,

we rewrite the optimization problem in equation (8.27) as

$$\begin{aligned} & \underset{\mathbf{s}_d, \mathbf{x}_{1:d}}{\text{minimize}} \quad \mathbf{d}^\top \mathbf{s}_d \\ & \text{subject to} \quad \mathbf{s}_1 \in \mathcal{S} \\ & \quad \mathbf{x}_t \in \mathcal{X}_t \text{ for all } t \in 1 : d \\ & \quad \mathbf{s}_d = r(\mathbf{s}_1, \mathbf{x}_{1:d}) \end{aligned} \tag{9.27}$$

For concrete reachability, we replace the last constraint with a constraint for each time step as follows:

$$\mathbf{s}_{t+1} = r(\mathbf{s}_t, \mathbf{x}_{t:t+1}) \text{ for all } t \in 1 : d - 1 \tag{9.28}$$

The optimization problem in equation (9.27) is a nonlinear program because $r(\mathbf{s}_1, \mathbf{x}_{1:d})$ is a nonlinear function of the state and disturbance. However, to ensure that the overapproximation of the reachable set holds, we must solve this optimization problem exactly. In general, we cannot find exact solutions for nonlinear programs, so we must introduce further overapproximations. The rest of this section discusses these methods.

9.5.1 Linear Programming through Conservative Linearization

We can transform the nonlinear program in equation (9.27) into a linear program using the conservative linearization technique introduced in section 9.3. Specifically, we create the following linear program:

$$\begin{aligned} & \underset{\mathbf{s}_d, \mathbf{x}_{1:d}, \boldsymbol{\alpha}}{\text{minimize}} \quad \mathbf{d}^\top \mathbf{s}_d \\ & \text{subject to} \quad \mathbf{s}_1 \in \mathcal{S} \\ & \quad \mathbf{x}_t \in \mathcal{X}_t \text{ for all } t \in 1 : d \\ & \quad \mathbf{s}_d = r(\mathbf{s}_c, \mathbf{x}_c) + \mathbf{J} \begin{bmatrix} \mathbf{s}_1 - \mathbf{s}_c \\ \mathbf{x}_{1:d} - \mathbf{x}_c \end{bmatrix} + \boldsymbol{\alpha} \\ & \quad \boldsymbol{\alpha} \in [\boldsymbol{\alpha}] \end{aligned} \tag{9.29}$$

where \mathbf{s}_c and \mathbf{x}_c are the centers of the state and disturbance sets and \mathbf{J} is the Jacobian of the reachability function evaluated at \mathbf{s}_c and \mathbf{x}_c . We introduce another decision variable $\boldsymbol{\alpha}$ to represent the remainder term and constrain it to belong to the Lagrange remainder interval $[\boldsymbol{\alpha}]$ (equation (9.25)). The concrete version

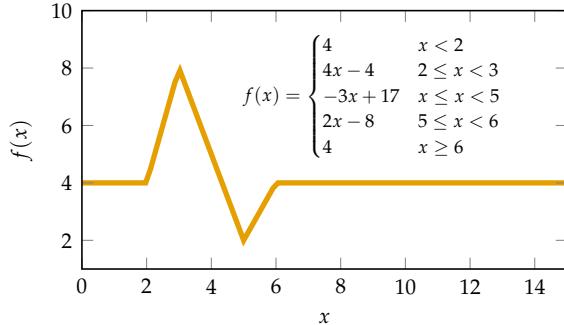


Figure 9.14. Example of a piecewise linear function.

of this linear program is similarly defined by replacing equation (9.28) with a conservative linearization for each time step.

9.5.2 Piecewise Linear Models

If the reachability function $r(\mathbf{s}, \mathbf{x}_{1:d})$ is piecewise linear, we can formulate the optimization problem as a *mixed-integer linear program* (MILP). A piecewise linear function is a function that comprises multiple linear functions that are activated based on the region of the input space (figure 9.14). A mixed-integer linear program is a linear program that includes some design variables that are constrained to a set of integers. We can convert piecewise linear functions to mixed-integer constraints by introducing binary variables that activate the appropriate linear function based on the input.

The process of encoding a piecewise linear function as a set of constraints begins by writing the function in terms of max and min functions (example 9.4). It is possible to write any piecewise linear function in this form.¹² We can then convert the max and min functions to mixed-integer constraints.¹³ Example 9.5 shows the conversion of the ReLU function. Encoding the piecewise linear reachability function as a set of mixed integer constraints turns equation (9.27) into a MILP, which we can solve using a variety of algorithms.¹⁴

While many real-world nonlinear systems do not have piecewise linear reachability functions, we can overapproximate them with piecewise linear bounds. First, we decompose the reachability function into a conjunction of elementary nonlinear functions (see example 9.6). For each nonlinear elementary function,

¹² C. Sidrane, A. Maleki, A. Irfan, and M. J. Kochenderfer, “OVERT: An Algorithm for Safety Verification of Neural Network Control Policies for Nonlinear Systems,” *Journal of Machine Learning Research*, vol. 23, no. 117, pp. 1–45, 2022.

¹³ More details can be found in V. Tjeng, K. Y. Xiao, and R. Tedrake, “Evaluating Robustness of Neural Networks with Mixed Integer Programming,” in *International Conference on Learning Representations (ICLR)*, 2018.

¹⁴ A detailed overview of integer programming can be found in L. A. Wolsey, *Integer Programming*. Wiley, 2020. Modern solvers, such as Gurobi and CPLEX, can routinely handle problems with millions of variables. There are packages for Julia that provide access to Gurobi, CPLEX, and a variety of other solvers.

Consider the following piecewise linear function (shown in the caption):

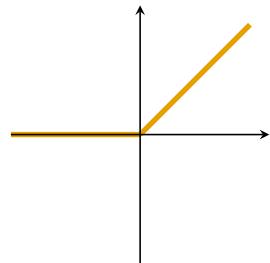
$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

This function is often referred to as the *rectified linear unit* (ReLU) function and is commonly used in neural networks. We can rewrite this function in terms of the max function as follows:

$$f(x) = \max(0, x)$$

If $x < 0$, the max function will return 0, and if $x \geq 0$, the max function will return x .

Example 9.4. Writing the ReLU function in terms of the max function.



we can derive piecewise linear lower and upper bounds over a given interval. We can then convert those bounds to mixed-integer constraints and solve the resulting MILP to overapproximate the reachable set.¹⁵

9.6 Partitioning

The methods presented in this chapter tend to result in less overapproximation error when computing reachable sets over smaller regions of the input space. For example, Taylor approximations are more accurate for points near the center of the region and become less accurate as we move away from the center (figure 9.15). Therefore, we want to keep the input set for Taylor inclusion functions and Taylor models as small as possible to minimize overapproximation error.

Based on this property, we can improve the performance of reachability algorithms by *partitioning* the input set into smaller regions and computing the reachable set for each region separately. Specifically, we divide the input set \mathcal{S} into a set of smaller regions $\mathcal{S}^{(1)}, \mathcal{S}^{(2)}, \dots, \mathcal{S}^{(m)}$ such that

$$\mathcal{S} = \bigcup_{i=1}^m \mathcal{S}^{(i)} \quad (9.30)$$

To compute the reachable set at depth d , we compute the reachable set $\mathcal{R}_d^{(i)}$ for each region $\mathcal{S}^{(i)}$ separately and then combine the results to form the reachable

¹⁵ For more details on the process of deriving the bounds and converting to constraints, see C. Sidrane, A. Maleki, A. Irfan, and M. J. Kochenderfer, "OVERT: An Algorithm for Safety Verification of Neural Network Control Policies for Nonlinear Systems," *Journal of Machine Learning Research*, vol. 23, no. 117, pp. 1–45, 2022.

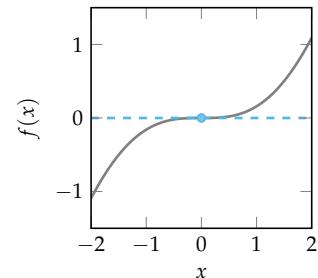


Figure 9.15. First-order Taylor approximation (dashed blue line) for the function $f(x) = x - \sin(x)$ (gray) at centered $x = 0$. The approximation is more accurate near the center.

Suppose we want to solve an optimization problem with the following piecewise linear constraint:

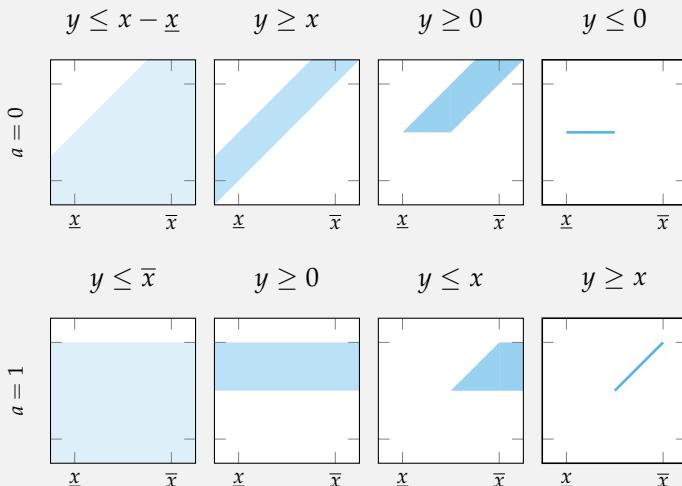
$$y = \max(0, x)$$

We will also assume that we know that x lies in the interval $[\underline{x}, \bar{x}]$. We can encode this constraint using a set of mixed-integer constraints as follows:

$$\begin{aligned} y &\leq x - \underline{x}(1 - a) \\ y &\geq x \\ y &\leq \bar{x}a \\ y &\geq 0 \\ a &\in \{0, 1\} \end{aligned}$$

Example 9.5. Mixed-integer formulation of the ReLU function.

The plots below iteratively build up the constrained region for each possible value of a .



When $a = 0$, y must be 0 and x must be between \underline{x} and 0. When $a = 1$, y must be equal to x and x must be between 0 and \bar{x} .

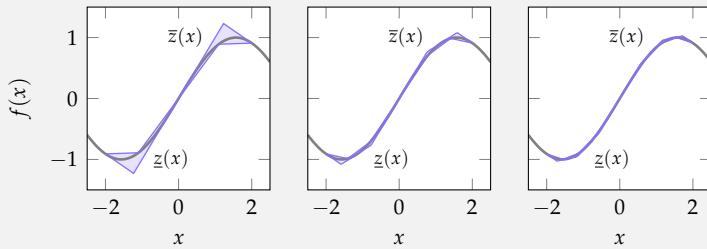
Consider the nonlinear constraint $y = x - \sin(x)$ over the region $-2 \leq x \leq 2$. We can convert this constraint into a set of piecewise linear constraints by first decomposing the function into its elementary functions:

$$\begin{aligned}y &= x - z \\z &= \sin(x) \\-2 &\leq x \leq 2\end{aligned}$$

We then derive a piecewise linear lower bound \underline{z} and upper bound \bar{z} for $\sin(x)$ and rewrite the constraints as

$$\begin{aligned}y &= x - z \\\underline{z} &\leq z \leq \bar{z} \\z &= \underline{z}(x) \\\bar{z} &= \bar{z}(x) \\-2 &\leq x \leq 2\end{aligned}$$

The plots below show the overapproximations of $\sin(x)$ using different numbers of linear segments.



The final step is to convert the piecewise linear functions $\underline{z}(x)$ and $\bar{z}(x)$ into their corresponding mixed-integer constraints. The overapproximations become tighter as the number of segments increases, but the computational cost and the number of mixed integer constraints required to represent the piecewise linear bounds also increases.

Example 9.6. Converting a nonlinear equality constraint into a set of mixed-integer constraints using piecewise linear bounds. We use the `OVERT.jl` package to compute the overapproximations.

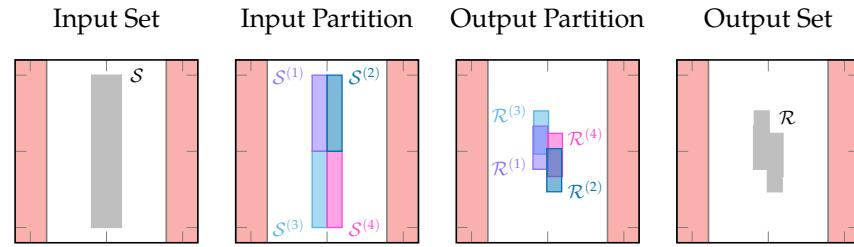


Figure 9.16. Computing the one-step reachable set for the inverted pendulum system using partitioning. The input set S is partitioned into four regions, and the reachable set for each region is computed separately using a first order Taylor inclusion function. The union of the resulting output sets forms the full reachable set.

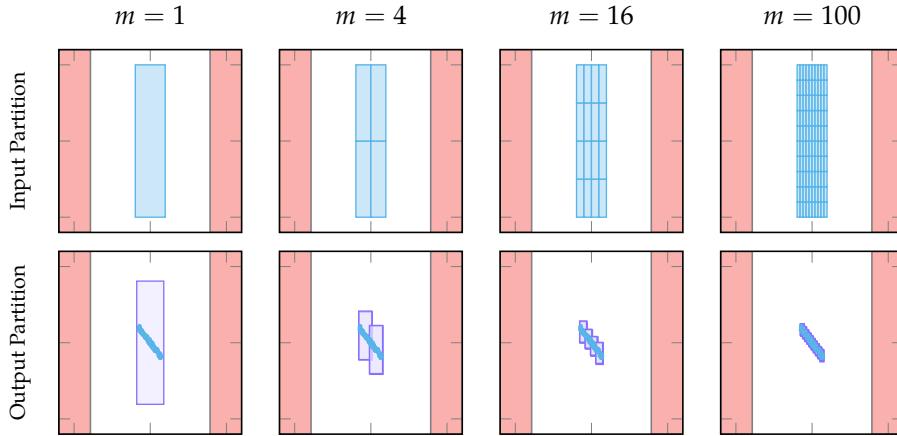


Figure 9.17. The effect of the number of subsets m in the partition on the overapproximation error in \mathcal{R}_6 for the inverted pendulum system. The reachability algorithm applied to each subset uses a first-order Taylor inclusion function. As m increases, the overapproximation error decreases.

set for the entire input set:

$$\mathcal{R}_d = \bigcup_{i=1}^m \mathcal{R}_d^{(i)} \quad (9.31)$$

Figure 9.16 demonstrates this process. The union of the reachable sets for each region is often nonconvex, which results in a benefit when performing reachability analysis for nonlinear systems with nonconvex reachable sets.

As shown in figure 9.17, partitioning the input set can significantly reduce overapproximation error. Finer partitions tend to result in more accurate reachable sets. In general, the performance is highly dependent on the partitioning strategy. Figure 9.17 uses a uniform partitioning strategy, in which the input set is divided into m equal-sized regions. However, this strategy may be intractable for systems with high-dimensional input spaces because the number of subsets

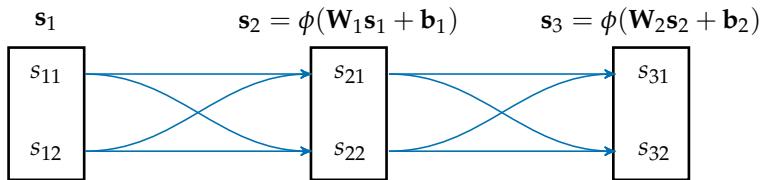


Figure 9.18. Example of a two layer neural network with two neurons in each layer.

in a uniform partition grows exponentially with the dimension. In such cases, we can use more sophisticated partitioning strategies, such as adaptive partitioning based on samples, to improve the accuracy of the reachable set while keeping the computational cost manageable.¹⁶

9.7 Neural Networks

We can use the techniques discussed in the previous sections to verify properties of neural networks. Neural networks are a class of functions that are widely used in machine learning and could be used to represent the agent, environment, or sensor model. They are composed of a series of layers, each of which applies an affine transformation followed by a nonlinear activation function.¹⁷ Given a set of inputs to a neural network, we are often interested in understanding the possible outputs.¹⁸ For example, we may want to ensure that an aircraft collision avoidance system will always output an alert when other aircraft are nearby.

Evaluating a neural network is similar to performing a rollout of a system. However, instead of computing s_{t+1} by passing s_t through the sensor, agent, and environment models, we compute it by passing s_t through the t th layer of the neural network. If s_t is the input to layer t , then the output s_{t+1} is computed as

$$s_{t+1} = \phi(\mathbf{W}_t s_t + \mathbf{b}_t) \quad (9.32)$$

where \mathbf{W}_t is a matrix of weights, \mathbf{b}_t is a bias vector, and $\phi(\cdot)$ is a nonlinear activation function. Common activation functions include ReLU, sigmoid, and hyperbolic tangent. Figure 9.18 shows an example of a two-layer neural network. In this context, we can check properties of the neural network by computing the reachable set of the output layer given an input set.

For piecewise linear activation functions, we can compute the exact reachable set by partitioning the input space into different activation sets and computing

¹⁶ M. Everett, G. Habibi, C. Sun, and J. P. How, "Reachability Analysis of Neural Feedback Loops," *IEEE Access*, vol. 9, pp. 163938–163953, 2021.

¹⁷ More details about the structure and training of neural networks are found in appendix C.

¹⁸ This process is sometimes referred to as *neural network verification*. A detailed overview of neural network verification can be found in C. Liu, T. Arnon, C. Lazarus, C. Strong, C. Barrett, and M. J. Kochenderfer, "Algorithms for Verifying Deep Neural Networks," *Foundations and Trends in Optimization*, vol. 4, no. 3–4, pp. 244–404, 2021.

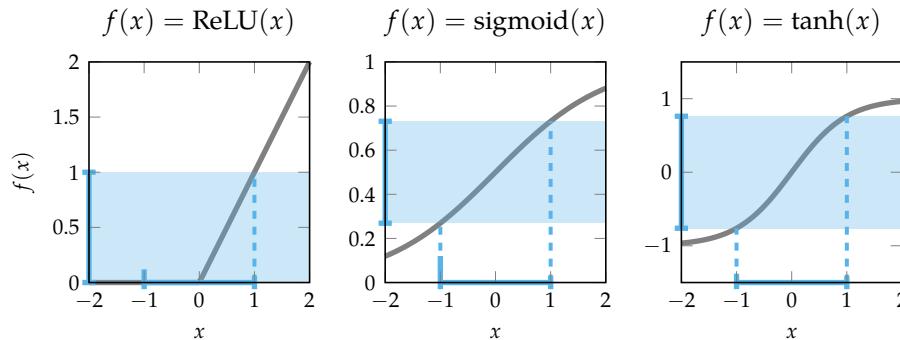


Figure 9.19. Example evaluations of the interval counterparts for three common neural network activation functions.

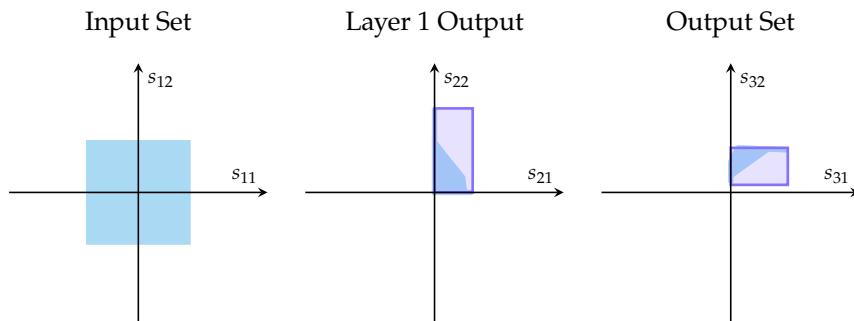


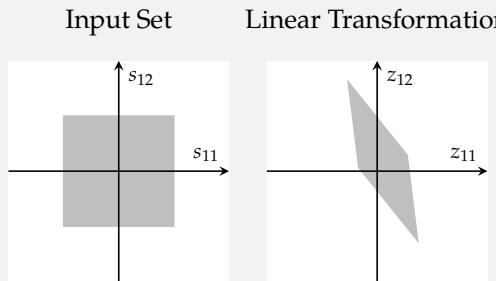
Figure 9.20. Computing the over-approximate reachable set of a two-layer neural network using natural inclusion functions. The true reachable set for each layer is shown in blue, and the interval overapproximation is shown in purple.

the reachable set for each subset separately.¹⁹ For example, we can compute exact reachable sets for neural networks with ReLU activation functions (example 9.7). However, the number of subsets grows exponentially with the number of nodes in the network. Therefore, exact reachability analysis is often intractable for large neural networks, so it is common to instead use overapproximation techniques to bound the output set.

Similar to the nonlinear systems discussed earlier, we can use inclusion functions to overapproximate the output set of neural networks. By replacing each activation function with its interval counterpart, we obtain the natural inclusion function for a neural network. Figure 9.19 shows an example evaluation of the interval counterpart for the ReLU function. Evaluating the natural inclusion function for the network on a set of input intervals provides an overapproximation of the possible network outputs (figure 9.20).

¹⁹ W. Xiang, H.-D. Tran, J. A. Rosenfeld, and T. T. Johnson, “Reachable Set Estimation and Safety Verification for Piecewise Linear Systems with Neural Network Controllers,” in *American Control Conference (ACC)*, 2018.

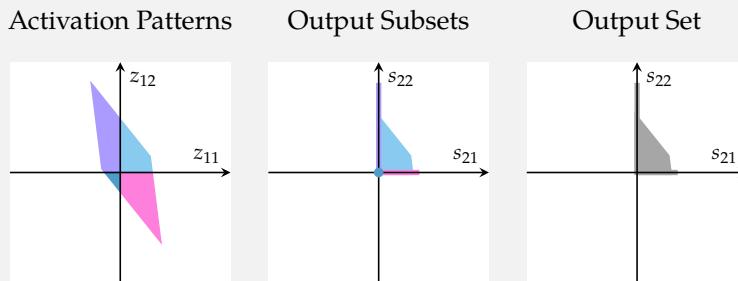
Suppose we want to propagate the input set \mathcal{S}_1 shown below through the first layer of the neural network in figure 9.18. We first apply the linear transformation to obtain the pre-activation region $\mathcal{Z}_2 = \mathbf{W}_1 \mathcal{S}_1 \oplus \mathbf{b}_1$:



Example 9.7. Exact reachability for a two-layer neural network with ReLU activation functions.

Next, we need to apply the nonlinear ReLU activation function to \mathcal{Z}_2 to compute \mathcal{S}_2 . We divide \mathcal{Z}_2 into four subsets for which the ReLU function is linear. Each subset corresponds to a different *activation pattern* for the first layer. An activation pattern describes which nodes in the layer are active for a given input. A node is considered active if its input is greater than zero.

Each quadrant in \mathcal{Z}_2 corresponds to a different activation pattern and maps to a different subset in the output set \mathcal{S}_2 . For example, in the first quadrant, both nodes are active, so the ReLU has no affect on the output. In the second quadrant, only the second node is active, so the inputs get mapped to a line. The output set \mathcal{S}_2 is the union of the four subsets. The plots below demonstrate this process.



To compute the final output set of the neural network in figure 9.18, we would repeat this process for each of the subsets that comprise \mathcal{S}_2 . The final output set will therefore be the union of 16 subsets.

As noted in section 9.2.1, natural inclusion functions tend to be overly conservative. Some techniques apply partitioning strategies to reduce overapproximation error.²⁰ Note that we cannot use Taylor inclusion functions or conservative linearization for ReLU networks because the ReLU function is not differentiable at zero. However, we can use other techniques to create inclusion functions that are tighter than the natural inclusion function.²¹

We can also evaluate the support function of the output set of a neural network with d layers by solving the following optimization problem:

$$\begin{aligned} & \underset{\mathbf{s}_d}{\text{minimize}} \quad \mathbf{d}^\top \mathbf{s}_d \\ & \text{subject to} \quad \mathbf{s}_1 \in \mathcal{S} \\ & \quad \mathbf{s}_d = f_n(\mathbf{s}_1) \end{aligned} \tag{9.33}$$

where $f_n(\mathbf{s}_1)$ is the neural network function and \mathcal{S} is the input set. For ReLU networks, it is possible to write this optimization problem as a MILP by converting each ReLU activation function into its corresponding mixed-integer constraints (see example 9.5). To create the mixed integer constraints, we need an upper and lower bound on the input to each ReLU. We can either select a sufficiently large bound for all nodes²² or compute specific bounds by evaluating the natural inclusion function.²³ To compute an overapproximation of the output set, we evaluate the support function in multiple directions.

In addition to evaluating the support function, we can use the MILP formulation to check other properties of the neural network by changing the objective function or adding constraints.²⁴ For example, we can check if the output set intersects with a given avoid set or find the maximum disturbance that causes the network to change its output. In general, neural network verification approaches can be combined with the techniques discussed in this chapter to verify closed-loop properties of systems that contain neural networks.²⁵

9.8 Summary

- Reachable sets for nonlinear systems are often nonconvex and difficult to compute exactly.
- We can apply a variety of techniques to overapproximate the reachable sets of nonlinear systems.

²⁰ W. Xiang, H.-D. Tran, and T.T. Johnson, "Output Reachable Set Estimation and Verification for Multilayer Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 11, pp. 5777–5783, 2018.

²¹ H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel, "Efficient Neural Network Robustness Certification with General Activation Functions," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 31, 2018.

²² M. Akintunde, A. Lomuscio, L. Maganti, and E. Pirovano, "Reachability Analysis for Neural Agent-Environment Systems," in *International Conference on Principles of Knowledge Representation and Reasoning*, 2018.

²³ V. Tjeng, K.Y. Xiao, and R. Tedrake, "Evaluating Robustness of Neural Networks with Mixed Integer Programming," in *International Conference on Learning Representations (ICLR)*, 2018.

²⁴ C. A. Strong, H. Wu, A. Zeljic, K.D. Julian, G. Katz, C. Barrett, and M.J. Kochenderfer, "Global Optimization of Objective Functions Represented by ReLU Networks," *Machine Learning*, vol. 112, pp. 3685–3712, 2023.

²⁵ M. Everett, G. Habibi, C. Sun, and J.P. How, "Reachability Analysis of Neural Feedback Loops," *IEEE Access*, vol. 9, pp. 163938–163953, 2021.

- Interval arithmetic allows us to propagate intervals through elementary functions.
- We can use interval arithmetic to create inclusion functions that provide overapproximate output intervals for nonlinear functions.
- Taylor inclusion functions overapproximate nonlinear functions by passing intervals through their Taylor series approximations.
- An n th order Taylor models represent sets using a Taylor approximation of degree $n - 1$ and an interval remainder term that bounds the sum of the remaining terms in the Taylor series.
- While Taylor inclusion functions always output hyperrectangular sets, Taylor models more expressive reachable sets and tend to produce tighter overapproximations.
- We can sample the support function of the reachable set for nonlinear systems by solving an overapproximate linear program or mixed-integer linear program.
- Because nonlinear reachability methods tend to produce tighter overapproximations on smaller input sets, we can reduce overapproximation error by partitioning the input space into smaller sets and computing the reachable set for each smaller set.
- We can extend some of the techniques outlined in this chapter to analyze the output sets of neural networks.

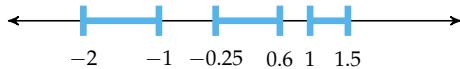
9.9 Exercises

Exercise 9.1. What is the interval counterpart for a function $f(x)$ that is monotonically decreasing?

Solution: The interval counterpart for a monotonically decreasing function can be defined in terms of function evaluations at its upper and lower bounds as follows:

$$f([x]) = [f(\bar{x}), f(\underline{x})]$$

Exercise 9.2. What is the interval hull of the blue set?



Solution: The interval hull of the blue set is $[-2, 1.5]$.

Exercise 9.3. Determine an overapproximate output interval for the function $f(x) = x \sin(\frac{1}{2}x)$ over the input interval $[-1, 1]$ using the natural inclusion function.

Solution: We evaluate the natural inclusion function for f over the interval $[-1, 1]$ as follows:

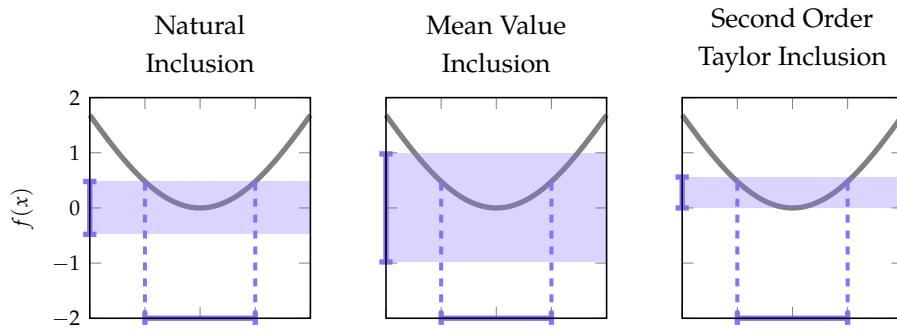
$$\begin{aligned}[f]([-1, 1]) &= [-1, 1] \sin(0.5[-1, 1]) \\ &= [-1, 1] \sin([-0.5, 0.5]) \\ &= ([-1, 1])([-0.479, 0.479]) \\ &= [-0.479, 0.479]\end{aligned}$$

Exercise 9.4. Determine an overapproximate output interval for the function $f(x) = x \sin(\frac{1}{2}x)$ over the input interval $[-1, 1]$ using the mean value inclusion function. Compare it to the result from exercise 9.3. Which function provides a tighter overapproximation? Why? How could we tighten the overapproximation from the mean value inclusion function?

Solution: Using equation (9.16) and the derivative of f , we have

$$[f]([x]) = f(c) + \left(\frac{1}{2}[x] \cos\left(\frac{1}{2}[x]\right) + \sin\left(\frac{1}{2}[x]\right) \right) ([x] - c)$$

where c is center of interval. Evaluating this expression at $c = 0$ and $[x] = [-1, 1]$ results in an overapproximate output interval of $[-0.979, 0.979]$. This interval is larger than the interval obtained using the natural inclusion function. This extra overapproximation is due to the fact that the mean value inclusion function relies on a linearization of the function; however, the function f is highly nonlinear over the interval $[-1, 1]$. We could tighten the overapproximation from the mean value inclusion function by using a higher-order Taylor inclusion function. The plots below show these results.



Exercise 9.5. Why does the overapproximation error of Taylor inclusion functions tend to increase as the time horizon increases?

Solution: The overapproximation error of Taylor inclusion functions tends to increase as the time horizon increases because the nonlinearities in the step function compound over time. This results in a function that is more difficult to approximate using only a few terms of the Taylor series.

Exercise 9.6. Write the polynomial and the interval remainder for the third order Taylor model for the function $f(x) = x - \sin x$ over the interval $[-1.5, 0]$ (shown in figure 9.10).

Solution: The polynomial of a third order Taylor model corresponds to the first three terms of the Taylor series expansion of the function centered at $c = -0.75$:

$$\begin{aligned} p(x) &= f(c) + f'(c)(x - c) + \frac{1}{2}f''(c)(x - c)^2 \\ &= -0.75 - \sin(-0.75) + (1 - \cos(-0.75))(x + 0.75) + \frac{1}{2}\sin(-0.75)(x + 0.75)^2 \\ &= -0.341x^2 - 0.243x - 0.059 \end{aligned}$$

We can calculate the interval remainder using equation (9.22):

$$\begin{aligned} [\alpha] &= \frac{[f^{(3)}]([x])}{3!} ([x] - c)^3 \\ &= \frac{\cos([-1.5, 0])}{6} ([-1.5, 0] + 0.75)^3 \\ &= [-0.07, 0.07] \end{aligned}$$

This polynomial and interval remainder results in the nonconvex set shown in figure 9.10.

Exercise 9.7. Provide one advantage of using concrete reachability compared the symbolic reachability for nonlinear systems.

Solution: There are multiple possible answers. One advantage is that concrete reachability tends to be more computationally efficient than symbolic reachability. Another advantage is that concrete reachability may decrease overapproximation error because it does not suffer from compounding nonlinearities in the way that symbolic reachability does. Concrete reachability also allows us to check for invariant sets.

Exercise 9.8. What is an advantage of Taylor inclusion functions over Taylor models?

Solution: There are multiple possible answers. One answer is that Taylor inclusion functions always output convex sets, while Taylor models only output convex sets for orders less than 3.

Exercise 9.9. Provide one advantage of using symbolic reachability compared to concrete reachability for nonlinear systems.

Solution: There are multiple possible answers. One advantage is that unlike concrete reachability, symbolic reachability does not suffer from the wrapping effect caused by concretizing the set at each time step.

Exercise 9.10. Why is it important that we find the global minimum of the optimization problem in equation (9.27) when computing reachable sets for nonlinear systems?

Solution: It is important to find the global minimum of the optimization problem in equation (9.27) because only the global minimum is guaranteed to produce a bounding halfspace for the reachable set.

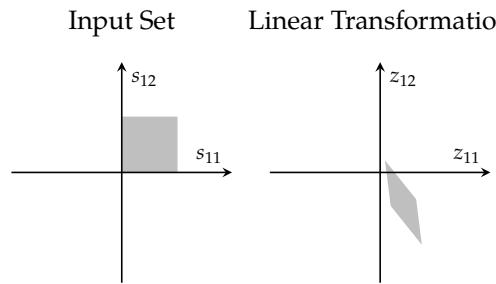
Exercise 9.11. In addition to minimizing overapproximation error by approximating over smaller regions, partitioning the state space also allows us to better represent reachable sets with complex shapes. What feature of partitioning creates this advantage?

Solution: When computing the final reachable set, we use the union of the reachable sets over each partition. This union often results in a nonconvex set that better represents the true reachable set.

Exercise 9.12. What might happen if we apply Taylor inclusion functions or Taylor models to determine the output set of a neural network that uses ReLU activation functions?

Solution: Because the ReLU function is not differentiable at zero, the assumptions required to soundly apply Taylor inclusion functions and Taylor models do not hold. Therefore, the results may not provide valid overapproximations of the output set.

Exercise 9.13. Suppose we want to propagate the input set S_1 shown below through the first layer of the neural network shown in figure 9.18. The linear transformation of the first layer results in the set shown on the right. What is the minimum number of polytopes required to define the output set of the first layer after the ReLU activation function is applied?



Solution: The output set of the first layer after the ReLU activation function is applied can be represented by the union of the two polytopes. The first polytope is the set of points of the linear transformation that are in the first quadrant, and the second polytope will be a line segment on the z_{11} axis.

10 Reachability for Discrete Systems

While the techniques in chapters 8 and 9 focus on reachability for systems with continuous states, this chapter focuses on reachability for systems with discrete states. We begin by representing the transitions of a discrete system as a directed graph. This formulation allows us to use graph search algorithms to perform reachability analysis. Next, we discuss techniques for probabilistic reachability analysis, in which we calculate the probability of reaching a particular state or set of states. We conclude by discussing a method to apply these techniques to continuous systems by abstracting them into discrete systems.

10.1 Graph Formulation

Directed graphs are a natural way to represent the transitions of a discrete system. A directed graph consists of a set of nodes and a set of directed edges connecting the nodes. For discrete systems, each node represents a state of the system, and each edge represents a transition between states (figure 10.1). We can also associate a probability with each edge to represent the likelihood of the transition occurring.

Algorithm 10.1 creates a directed graph from a discrete system. For each discrete state, it computes the set of possible next states and their corresponding probabilities. It then adds an edge to the graph for each possible transition. Figure 10.2 shows the graph representation of the grid world system. For systems with large state spaces, it may be inefficient to store the full graph in memory. In these cases, we can represent the graph implicitly using a function that takes in a state and returns its successors and their probabilities.

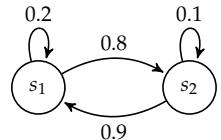


Figure 10.1. Graph representation of a discrete system with two states, s_1 and s_2 . The graph has a node for each state and an edge originating from each state for each possible transition. Each edge is labeled with the probability of the transition. For example, when we are in s_1 , we have a 0.8 probability of transitioning to s_2 .

```

function to_graph(sys)
    S = states(sys.env)
    g = WeightedGraph(S)
    for s in S
        S', ws = successors(sys, s)
        for (s', w) in zip(S', ws)
            add_edge!(g, s, s', w)
        end
    end
    return g
end

```

Algorithm 10.1. Converting a discrete system to a directed weighted graph using an extension of the `Graphs.jl` package (see appendix D for more details). For each state returned by the `states` function, the algorithm calls the system-specific `successors` function to determine the set of possible next states and their corresponding probabilities. It then adds an edge to the graph for each possible transition.

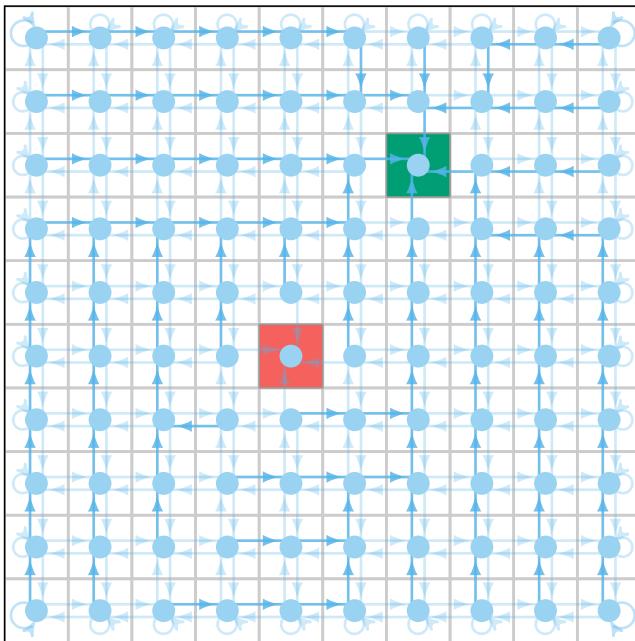


Figure 10.2. Graph representation of the grid world system. Each node represents a grid world state, and each edge represents a possible transition between states. Darker edges have higher probabilities associated with them.

10.2 Reachable Sets

To compute reachable sets, we ignore the probabilities associated with the edges of the graph and focus only on its connectivity. The reachable sets are represented as collections of discrete states. We focus on two types of reachability analysis: *forward reachability* and *backward reachability*. Forward reachability analysis determines the set of states that can be reached from a given set of initial states within a specified time horizon.¹ Backward reachability analysis determines the set of states from which a given set of target states can be reached within a specified time horizon. Figure 10.3 demonstrates the difference between the two types of reachability analysis, and the rest of this section presents algorithms for each type.

¹This process is sometimes referred to as *bounded model checking*

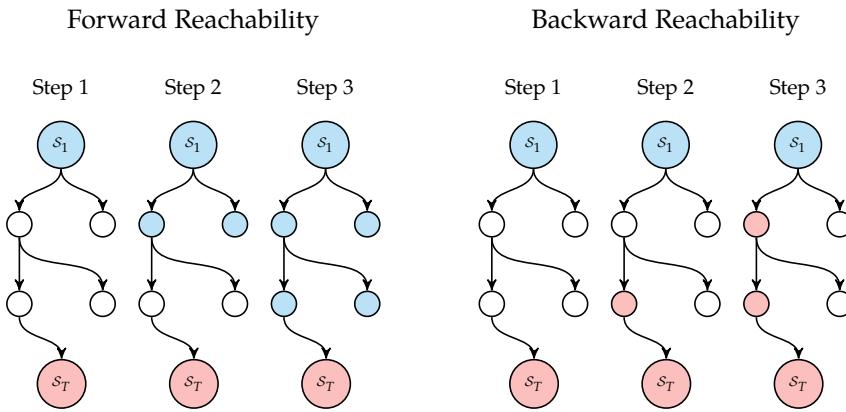


Figure 10.3. Example of forward and backward reachability on a discrete system with six states. The forward reachability algorithm determines starts from the initial set S_1 and progresses forward through the graph to determine the set of states that can be reached within a specified time horizon. The backward reachability algorithm starts from the target set S_T and progresses backward through the graph to determine the set of states from which the target state can be reached within a specified time horizon.

10.2.1 Forward Reachability

To compute the forward reachable set from a set of initial states, we perform a *breadth-first search* on the graph. We start with the initial set of states $S_1 = \mathcal{R}_1$ and iteratively add the set of states reachable from the current set of states at the next time step. In other words, \mathcal{R}_d is the set of states for which the graph contains an edge originating from a state in \mathcal{R}_{d-1} . We repeat this process for a specified time horizon or until convergence. Algorithm 10.2 implements this technique, and figure 10.4 shows the results on the grid world problem.

```

struct DiscreteForward <: ReachabilityAlgorithm
    h # time horizon
end

function reachable(alg::DiscreteForward, sys)
    g = to_graph(sys)
    S = S1(sys.env)
    R = S
    for d in 2:alg.h
        S = Set(reduce(vcat, [outneighbors(g, s) for s in S]))
        R == (R ∪ S) && break
        R = R ∪ S
    end
    return R
end

```

Algorithm 10.2. Forward reachability for discrete systems. The algorithm first creates the graph representation of the system by calling algorithm 10.1. For each depth d , it computes \mathcal{R}_d by finding the set of states reachable from \mathcal{R}_{d-1} according to the edges in the graph and checks for convergence. The `outneighbors` function returns all nodes connected to the current node through an outgoing edge. The algorithm returns the union of all sets, which corresponds to $\mathcal{R}_{1:h}$.

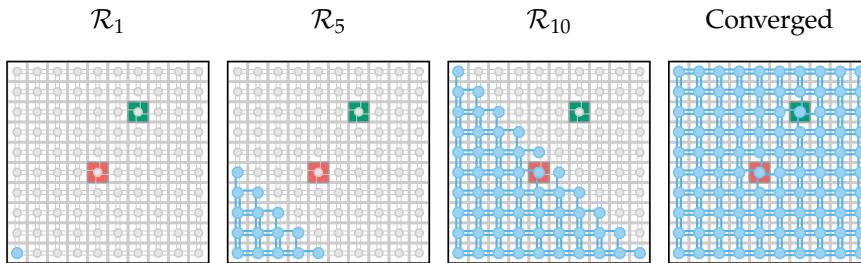


Figure 10.4. Forward reachable sets for the grid world system. Reachable states and their corresponding edges are highlighted in blue. In this example, the reachable set converges after 19 steps and shows that all states are reachable from the initial state.

The reachable set has converged once it no longer changes. If we find that $\mathcal{R}_{1:d} = \mathcal{R}_{1:d-1}$, the reachable set has converged, and $\mathcal{R}_{1:\infty} = \mathcal{R}_{1:d}$.² We can also check for invariant sets by relaxing this condition. Specifically, if $\mathcal{R}_d \subseteq \mathcal{R}_{1:d-1}$, we can conclude that $\mathcal{R}_{1:d}$ is an invariant set and that the system will remain within this set for all future time steps ($\mathcal{R}_{1:\infty} \subseteq \mathcal{R}_{1:d}$). Performing this check on discrete sets is straightforward because we can directly compare the states contained in each set.

10.2.2 Backward Reachability

In contrast with forward reachability, which starts from a set of initial states and progresses forward through the graph, backward reachability starts from a set of target states \mathcal{S}_T and progresses backward through the graph. The target set is often determined based on a specification for the system. For example, the target set may represent a set of goal states or a set of states that should be avoided. The backward reachable set $\mathcal{B}_{1:h}$ represents the set of states from which the target set can be reached within the time horizon h .

Algorithm 10.3 computes backwards reachable sets for discrete systems given a reachability specification. It has a structure similar to algorithm 10.2. However, instead of starting with the initial state set, it starts with the target set. It then iteratively computes \mathcal{B}_d as the set of states for which the graph contains an edge originating from a state in \mathcal{B}_{d-1} and ending at a state in \mathcal{B}_d . We can check for convergence and invariance using the same conditions we use for forward reachability. Figure 10.5 shows the results of applying the algorithm to the grid world problem to compute the backward reachable sets from the goal and obstacle states.

10.3 Satisfiability

We can use the forward and backward reachable sets of discrete systems to determine whether they satisfy a reachability specification (figure 10.6). For forward reachability, we check whether the target set intersects with the forward reachable set. For backward reachability, we check whether the initial set intersects with the backward reachable set. In both cases, these checks require us to compute the full forward or backward reachable set. This process can be computationally expensive, especially for systems with large state spaces.

²This condition allows us to perform *unbounded model checking*, in which the output holds over all possible trajectories.

```

struct DiscreteBackward <: ReachabilityAlgorithm
    h # time horizon
end

function backward_reachable(alg::DiscreteBackward, sys, ψ)
    g = to_graph(sys)
    S = ψ.set
    B = S
    for d in 2:alg.h
        S = Set(reduce(vcat, [inneighbors(g, s) for s in S]))
        B == (B ∪ S) && break
        B = B ∪ S
    end
    return B
end

```

Algorithm 10.3. Backward reachability for discrete systems. The algorithm first creates the graph representation of the system by calling algorithm 10.1. For each depth d , it computes \mathcal{B}_d by finding the set of states from which \mathcal{B}_{d-1} can be reached according to the edges in the graph and checks for convergence. The `inneighbors` function returns all nodes connected to the current node through an incoming edge. The algorithm returns the union of all sets, which corresponds to $\mathcal{B}_{1:h}$.

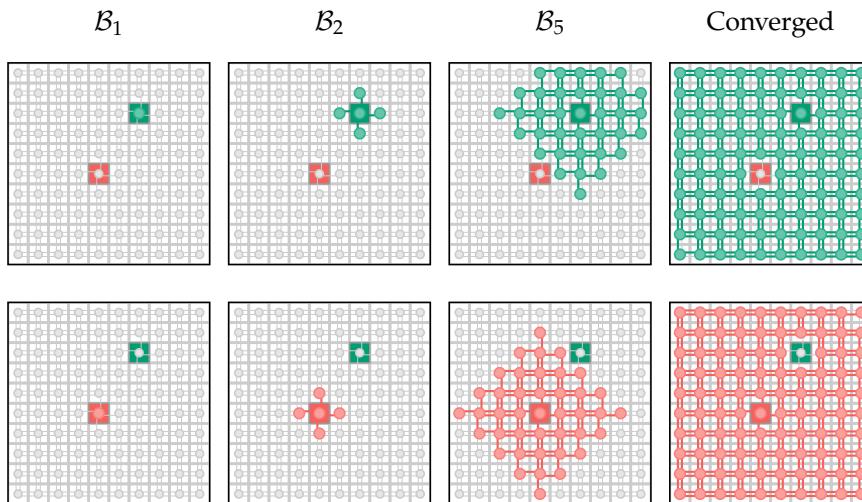


Figure 10.5. Backward reachable sets for the grid world system. The top row shows the backward reachable sets from the goal state (green), and the bottom row shows the backward reachable sets from the obstacle state (red). The reachable sets from the goal state converge after 14 steps, while the reachable sets from the obstacle state converge after 11 steps. The results show that the goal can be reached from any state outside the obstacle and that the obstacle can be reached from any state outside the goal.

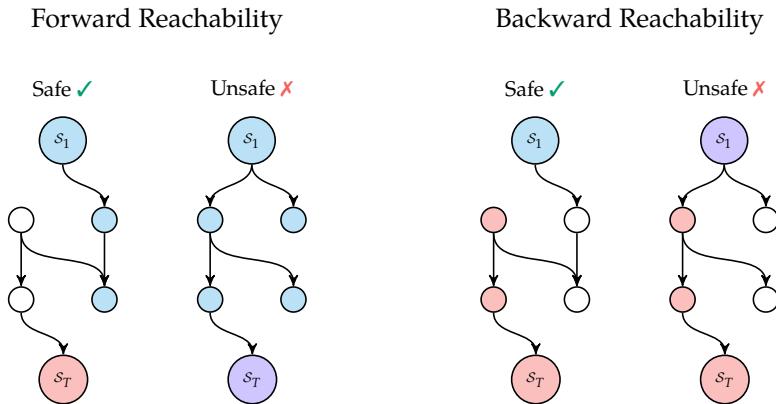


Figure 10.6. Checking whether a discrete system satisfies a reachability specification using forward (blue) and backward (red) reachable sets. If the forward reachable set overlaps (purple) with the avoid set S_T , the system is unsafe. Furthermore, if the backward reachable set overlaps (purple) with the initial set S_1 , the system is unsafe.

10.3.1 Counterexample Search

If our only goal is to check whether a system satisfies a reachability specification, we can use more efficient techniques that do not require us to compute the full reachable set. For example, we could perform the same breadth-first search we perform in algorithms 10.2 and 10.3 while only storing the states in the current and previous reachable sets, \mathcal{R}_d and \mathcal{R}_{d-1} , and performing a check for overlap with the target set at each iteration. This approach tends to be more memory-efficient than storing the full reachable set.

When the target set is an avoid set, we call this analysis *counterexample search* because reaching the target set represents a *counterexample* that proves that the system does not satisfy the specification.³ If the analysis converges without reaching any states in the avoid set, we can conclude that the avoid set is not reachable and the system satisfies the specification. Conversely, if we reach a state in the avoid set, we can terminate the search early and return the counterexample.

If a counterexample exists, we can save computation by finding it early and terminating the search. In these cases, we may want to use a different graph traversal algorithm such as depth-first search. Depth-first search explores the graph by following a single path to its maximum depth before backtracking.⁴ It therefore allows us to more quickly begin searching over full trajectories, which could be more efficient for finding counterexamples.

The use of more sophisticated graph search algorithms may further increase efficiency.⁵ Heuristic search algorithms such as the one introduced in section 5.3

³ The term *counterexample* is another word for failure that is commonly used in formal verification.

⁴ We can also perform depth first search to a fixed depth and increase the depth if no counterexamples are found. This process is known as *iterative deepening*.

⁵ The `Graphs.jl` package in Julia implements a variety of graph search algorithms such as Dijkstra's algorithm, the Floyd-Warshall algorithm, and heuristic search. More details on graph search are provided in S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021.

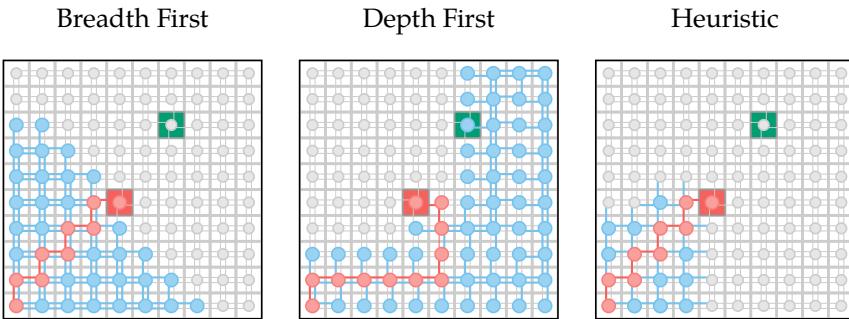


Figure 10.7. Comparison of breadth-first, depth-first, and heuristic search for finding counterexamples in the grid world system. We use A* search as the heuristic search algorithm, which significantly improves efficiency over breadth-first and depth-first search.

further increase efficiency by using heuristics to prioritize paths that are more likely to lead to a counterexample. In cases where the system satisfies the specification and no counterexample exists, these algorithms have the same computational complexity as breadth-first search. Figure 10.7 compares the performance of breadth-first search, depth-first search, and heuristic search for finding counterexamples in the grid world problem.

10.3.2 Boolean Satisfiability

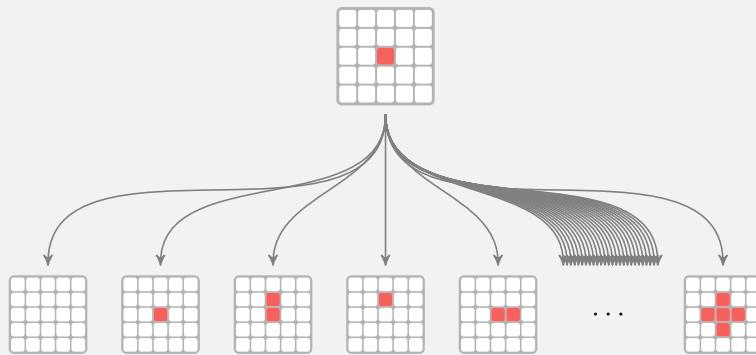
Graph search algorithms may be inefficient for systems with large state spaces, especially when each state has many neighboring states (example 10.1). In these cases, it may be more efficient to formulate the reachability problem as a *Boolean satisfiability* (SAT) problem. Solving a SAT problem involves searching for a satisfying assignments of the Boolean variables, or propositions, in a propositional logic formula (see section 3.4.1).⁶

In the context of reachability analysis, the Boolean variables in the SAT problem represent the discrete states of the system at each time step. The propositional logic formula encodes the possible initial states, transitions between states, and the failure condition. We can then pass the formula to a variety of different SAT solvers, which use heuristics to efficiently search for a satisfying assignment.⁷ If the SAT solver finds a satisfying assignment, the assignment corresponds to a counterexample, and the system does not satisfy the specification. If the SAT solver does not find a satisfying assignment, we can conclude that system satisfies the specification.

⁶ Boolean satisfiability is also sometimes referred to as propositional satisfiability.

⁷ More information on SAT solvers is provided in A. Biere, *Handbook of Satisfiability*. IOS Press, 2009, vol. 185. The `Satisfiability.jl` package provides a Julia interface for many common SAT solvers. E. Soroka, M. J. Kochenderfer, and S. Lall, “`Satisfiability.jl`: Satisfiability Modulo Theories in Julia,” *Journal of Open Source Software*, vol. 9, no. 100, p. 6757, 2024.

The wildfire problem is an example of a problem in which graph search is intractable. Consider a wildfire scenario modeled as an $n \times n$ grid where each cell is either burning or not burning. At each time step, a burning cell has a nonzero probability of spreading the fire to each of its neighboring cells. A burning cell will also remain burning at the next time step with some probability. This problem has 2^{n^2} possible states, and a state with b burning cells has as many as 2^{5b} possible successors. For a 5×5 grid, the state space has $2^{25} = 3.4 \times 10^7$ states. For a 10×10 grid, that number increases to $2^{100} = 1.27 \times 10^{30}$ possible states. The example below shows the successors for a state where only the cell in the center is burning.



Even though only one cell is burning, there are still 32 successor states. This number only increases as we increase the number of burning cells. A state with 10 burning cells has as many as $2^{50} = 1.13 \times 10^{15}$ successors. For most grid sizes, even partially computing and storing the graph for the wildfire problem is intractable. For this reason, we cannot use graph search algorithms for this problem and must turn to other methods such as Boolean satisfiability.

Example 10.1. Demonstration of difficulties that arise when applying graph search algorithms to the wildfire problem.

For a system with states represented as Boolean vectors⁸ of length m and a given horizon h , we define $\mathbf{s}_{1:h}$ as a set of Boolean variables each of length m representing the states at each time step. The initial state set is encoded as a propositional logic formula $I(\mathbf{s}_1)$, which returns true if \mathbf{s}_1 is in the initial state set and false otherwise. Example 10.2 demonstrates how to encode the initial state of the wildfire problem. Next, we define a propositional logic formula for each state transition $T(\mathbf{s}_t, \mathbf{s}_{t+1})$ that returns true if \mathbf{s}_{t+1} is a successor of \mathbf{s}_t and false otherwise (see example 10.3 for the wildfire problem). The failure condition is the negation of the specification ψ .

Consider a wildfire problem with a 10×10 grid and a time horizon of $h = 20$. The state at a particular time step is represented as a set of Boolean variables that represent whether each grid cell is burning. The SAT problem will therefore have $100 \times 20 = 2000$ Boolean variables representing the states at each time step. We can represent the initial state as a propositional logic formula that evaluates to true when the bottom left cell is burning and all other cells are not burning. The following code implements this formula:

```
n = 10 # grid is n x n
h = 20 # time horizon
@satvariable(burning[1:n, 1:n, 1:h], Bool)
init = burning[1, 1, 1] # bottom left cell is burning
for i in 1:n, j in 1:n
    if i ≠ 1 || j ≠ 1 # all other cells are not burning
        init = init ∧ ¬burning[i, j, 1]
    end
end
```

⁸ SAT solvers require Boolean variables. *Satisfiability modulo theories (SMT)* solvers extend SAT solvers to continuous variables. More information about SMT is provided in A. Biere, *Handbook of Satisfiability*. IOS Press, 2009, vol. 185.

Example 10.2. Encoding the initial state of the wildfire problem as a propositional logic formula using the `Satisfiability.jl` package.

Combining the initial state and transition formulas with the failure condition, we can create a single propositional logic formula that represents the reachability problem:

$$I(\mathbf{s}_1) \wedge (T(\mathbf{s}_1, \mathbf{s}_2) \wedge T(\mathbf{s}_2, \mathbf{s}_3) \dots \wedge T(\mathbf{s}_{h-1}, \mathbf{s}_h)) \wedge \neg\psi(\mathbf{s}_{1:h}) \quad (10.1)$$

A SAT solver will search the space of possible values for the Boolean variables $\mathbf{s}_{1:h}$ to find an assignment that satisfies the formula. A satisfying assignment corresponds to a feasible trajectory that satisfies the failure condition. Therefore, if the SAT solver determines that there are no satisfying assignments, we can conclude that the system satisfies the specification. Example 10.4 demonstrates how to use Boolean satisfiability to check reachability specifications for the wildfire problem.

The following code implements the propositional logic formula for the transitions of the wildfire problem:

```
transition = true
for i in 1:n, j in 1:n, t in 1:h-1
    transition = transition ∧ (
        burning[i, j, t+1] ⇒
        (burning[i, j, t] ∨
         burning[max(1, i-1), j, t] ∨
         burning[min(n, i+1), j, t] ∨
         burning[i, max(1, j-1), t] ∨
         burning[i, min(n, j+1), t])
    )
end
```

If a particular cell is burning at time $t + 1$, it must be the case that either it was burning at time t or one of its neighbors was burning at time t . The examples below show two evaluations of the transition proposition.

$$T \left(\begin{array}{|c|c|} \hline & \textcolor{red}{\blacksquare} \\ \hline \end{array}, \begin{array}{|c|c|} \hline & \textcolor{red}{\blacksquare} \\ \hline \end{array} \right) = \text{true}$$

$$T \left(\begin{array}{|c|c|} \hline & \textcolor{red}{\blacksquare} \\ \hline \end{array}, \begin{array}{|c|c|} \hline & \\ \hline \end{array} \right) = \text{false}$$

In the first case, both cells burning at time $t + 1$ were either burning at time t or had a neighbor that was burning at time t . In the second case, the cell at $(3, 4)$ was not burning at time t , and none of its neighbors were burning at time t .

Example 10.3. Encoding the transitions of the wildfire problem as a propositional logic formula using the `Satisfiability.jl` package.

Suppose there is a densely populated area in the top right cell of the wildfire grid, and we want to determine whether it might burn. We can encode the failure condition as a propositional logic formula that evaluates to true when the top right cell is burning. We can then combine this formula with the initial state and transition formulas from equation (10.1) and pass it to a SAT solver to determine whether the top right cell is reachable. The following code demonstrates this process:

```
ψ = ¬burning[n, n, t]
reachable = sat!(init ∧ transition ∧ ¬ψ)
```

For a 10×10 grid with a horizon of 20, the `burning` variable has $10 \times 10 \times 20 = 2000$ Boolean variables and therefore 2^{2000} possible assignments. However, the SAT solver can efficiently search this space to find a satisfying assignment in a few seconds. If we decrease the time horizon to 18, the SAT solver is able to determine in a similar amount of time that none of the 2^{1800} possible assignments satisfy the formula. This result indicates that the top right cell is not reachable within 18 time steps.

Example 10.4. Checking reachability specifications for the wildfire problem using Boolean satisfiability.

10.4 Probabilistic Reachability

Probabilistic reachability analysis computes the probability of reaching a target set by taking into account the probability of each transition between states.⁹ In some cases, the results allow us to build more confidence in a system than the reachable sets alone. For example, if the avoid set overlaps with the reachable set, our reachability analysis will conclude that the system is unsafe even if the probability of reaching the avoid set is very low. Example 10.5 demonstrates this property on the grid world problem. Probabilistic reachability analysis allows us to uncover these scenarios and provide a more useful safety assessment that focuses on actual risk.

10.4.1 Probability of Occupancy

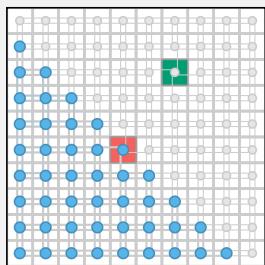
Determining the probability of occupancy involves computing a distribution over reachable states at each time step. We denote this distribution as P_t , where $P_t(s)$ is the probability of occupying state s at time step t . The algorithm begins with an

⁹ When we consider the transition probabilities, the system represents a discrete-time Markov chain. More information on the analysis of Markov chains is provided in J. R. Norris, *Markov Chains*. Cambridge University Press, 1998. Open source software packages such as PRISM (M. Kwiatkowska, G. Norman, and D. Parker, “PRISM 4.0: Verification of Probabilistic Real-Time Systems,” in *International Conference on Computer Aided Verification*, 2011.) and STORM C. Hensel, S. Junges, J.-P. Katoen, T. Quatmann, and M. Volk, “The Probabilistic Model Checker Storm,” *International Journal on Software Tools for Technology Transfer*, pp. 1–22, 2022. implement these analysis techniques.

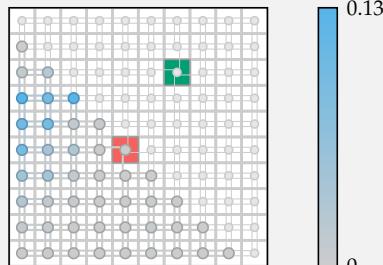
Consider the grid world problem with a slip probability of 0.3. Running algorithm 10.2 with a time horizon $h = 9$ leads to the conclusion that the system is unsafe because the obstacle is included in the forward reachable set. However, the probability of reaching the obstacle after 9 steps when following the optimal policy is only 0.0004, and the system is more likely to be in a state near its nominal path to the goal. In this scenario, the probabilistic reachability provides a more useful assessment of the actual safety of the system. The plots below show the reachable set (left) and the results of a probabilistic reachability analysis (right).

Example 10.5. Comparison of reachable set analysis and probabilistic forward reachability analysis on the grid world problem.

Reachable Set



Probabilistic Reachability



initial state distribution P_1 . It then computes the distribution at each subsequent time step using the distribution from the previous time step and the transition probabilities between states as follows

$$P_{t+1}(s) = \sum_{s' \in \mathcal{S}} T(s', s) P_t(s') \quad (10.2)$$

where $T(s', s)$ is the probability of transitioning from state s' to state s .

Algorithm 10.4 implements probabilistic forward reachability using the graph representation of the system. The weights in the graph correspond to $T(s', s)$ in equation (10.2). The algorithm also uses the fact that the only nonzero terms in the sum in equation (10.2) are the terms corresponding to the incoming neighbors of s in the graph. The algorithm terminates after a desired time horizon h . Example 10.5 demonstrates this technique on the grid world problem.

```
struct ProbabilisticOccupancy <: ReachabilityAlgorithm
    h # time horizon
end

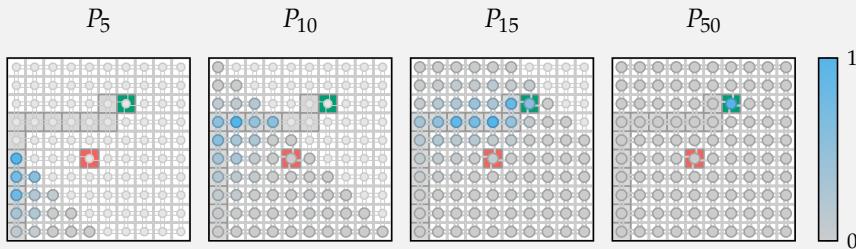
function reachable(alg::ProbabilisticOccupancy, sys)
    S, g, dist = states(sys.env), to_graph(sys), Ps(sys.env)
    P = Dict(s => pdf(dist, s) for s in S)
    for t in 2:alg.h
        P = Dict(s => sum(get_weight(g, s', s) * P[s']
                            for s' in inneighbors(g, s)) for s in S)
    end
    return SetCategorical(P)
end
```

Algorithm 10.4. Determining the probability of occupancy for discrete systems. The algorithm first creates the graph representation of the system using algorithm 10.1. It then begins with the initial state distribution and iteratively computes the distribution at each time step using equation (10.2). The `inneighbors` function returns all nodes in the graph that are connected to the current node through an incoming edge. The algorithm terminates when it has reached the time horizon.

10.4.2 Finite-Horizon Reachability

In addition to the distribution over states at a particular time step, we may also be interested in the probability that target state or set of states is reached within a given time horizon. We denote this probability as R_t , where $R_t(s)$ is the probability of reaching the target set \mathcal{S}_T when starting from state s within t time steps. Unlike the output of probabilistic occupancy analysis, R_t is not a distribution over states, and the probability values for all states will not sum to 1.

The plots below show the results from probabilistic occupancy analysis on the grid world problem with a slip probability of 0.3. They show the distribution over reachable states at different time steps with reachable states appearing larger and darker states indicating a higher probability of reaching them. The nominal path is highlighted in gray.



Example 10.6. Determining occupancy probabilities for the grid world problem.

While the obstacle state is reachable in three of the plots, the probability of occupying the obstacle state is low and the probability is much higher for states near the nominal path. After 50 time steps, most of the probability mass is in the goal state with a small portion in the obstacle state and the other grid cells. At this point, the probability of being in the goal state is 0.981 and the probability of being in the obstacle state is 0.018. We can use these numbers to draw conclusions about the overall safety of the system.

Similar to P_t , we can derive a recursive relationship to compute R_t such that

$$R_{t+1}(s) = \begin{cases} 1 & \text{if } s \in \mathcal{S}_T \\ \sum_{s' \in \mathcal{S}} T(s, s') R_t(s') & \text{otherwise} \end{cases} \quad (10.3)$$

In other words, for states in the target set, the probability of reaching the target set is 1. For all other states, the probability of reaching the target set within $t + 1$ time steps is sum of the probability of transitioning to each of its successors times the probability that they reach the target set within t time steps. We initialize R_1 to be 1 for states in the target set and 0 otherwise.

We can use the results of this analysis to identify dangerous states for the system. Furthermore, if we know the initial state distribution P_1 for the system, we can determine the probability of reaching the target set within a given time horizon by summing the probability of reaching the target set from each state weighted by the probability of occupying that state at time $t = 1$:

$$P_{\text{reach}} = \sum_{s \in \mathcal{S}} R_h(s) P_1(s) \quad (10.4)$$

Algorithm 10.5 implements finite-horizon probabilistic reachability using the graph representation of the system given a reachability specification, and figure 10.8 demonstrates this technique on the grid world problem.

```

struct ProbabilisticFiniteHorizon <: ReachabilityAlgorithm
    h # time horizon
end

function reachable(alg::ProbabilisticFiniteHorizon, sys,  $\psi$ )
    S, g, dist = states(sys.env), to_graph(sys), Ps(sys.env)
    ST =  $\psi$ .set
    R = Dict(s  $\Rightarrow$  s  $\in$  ST ? 1.0 : 0.0 for s in S)
    for d in 2:alg.h
        R = Dict(s  $\Rightarrow$  s  $\in$  ST ? 1.0 : sum(get_weight(g, s, s') * R[s'] for s' in outneighbors(g, s) for s in S)
    end
    return sum(R[s] * pdf(dist, s) for s in S)
end
```

Algorithm 10.5. Finite-horizon probabilistic reachability for discrete systems. The algorithm first creates the graph representation of the system using algorithm 10.1. It then initializes the probability of reaching the target set from each state and iteratively computes the probability at each time step using equation (10.3). The algorithm terminates when it has reached the time horizon. It returns the probability of reaching the target set in h time steps given the initial state distribution, which is computed according to equation (10.4).

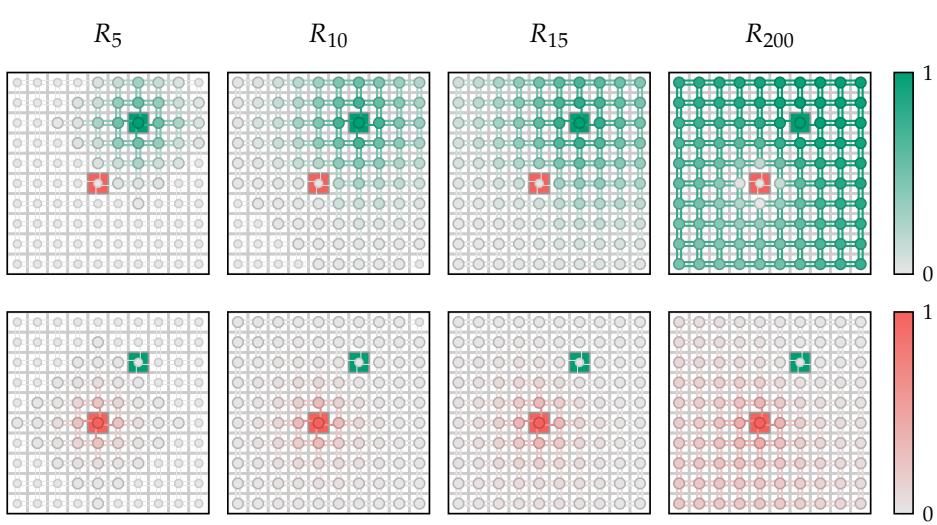


Figure 10.8. Finite-horizon probabilistic reachability for the grid world problem with a slip probability of 0.4. The top row shows the result when we set the target set to the goal state, and the bottom row shows the result when we set the target set to the obstacle state. States with nonzero probability are colored according to the probability of reaching the target set. Given an initial state distribution that places all probability on the state in the bottom left corner, the probability of reaching the goal state within 200 time steps is 0.777, and the probability of reaching the obstacle state is 0.222.

10.4.3 Infinite-Horizon Reachability

If we run finite-horizon reachability analysis over a large horizon, the probability of reaching the target set will begin to converge (see figure 10.9). Therefore, in many scenarios, running the analysis for a sufficiently long time horizon is enough to draw conclusions about the overall safety of the system. However, it is also possible to compute the probability of reaching the target set in the limit as the time horizon approaches infinity. This probability is known as the infinite-horizon reachability probability, and we denote it as $R_\infty(s)$.

To compute this probability, we rewrite the recursive relationship in equation (10.3) as

$$R_{t+1}(s) = R_t(s) + \sum_{s' \in \mathcal{S}} T_R(s, s') R_t(s') \quad (10.5)$$

where

$$T_R(s, s') = \begin{cases} 0 & \text{if } s \in \mathcal{S}_T \\ T(s, s') & \text{otherwise} \end{cases} \quad (10.6)$$

While this formulation is equivalent to equation (10.3), it allows us to compute the infinite-horizon reachability probability by solving a system of linear equations.

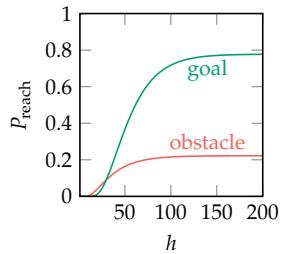


Figure 10.9. Probability of reaching the goal state and the obstacle state in the grid world problem with a slip probability of 0.6 as a function of the time horizon. We assume the system is initialized in the bottom left corner. As the horizon increases, the probabilities begin to converge.

We can write this system in matrix form as

$$\mathbf{R}_{t+1} = \mathbf{R}_1 + \mathbf{T}_R \mathbf{R}_t \quad (10.7)$$

where \mathbf{R}_t is a vector of length $|\mathcal{S}|$ such that the i th entry corresponds to $R_t(s_i)$, and \mathbf{T}_R is a matrix of size $|\mathcal{S}| \times |\mathcal{S}|$ such that entry in the i th row and j th column corresponds to $T_R(s_i, s_j)$.¹⁰

For an infinite horizon, we have that

$$\mathbf{R}_\infty = \mathbf{R}_1 + \mathbf{T}_R \mathbf{R}_\infty \quad (10.8)$$

We can solve for \mathbf{R}_∞ by rearranging the terms in equation (10.8) to get

$$\mathbf{R}_\infty - \mathbf{T}_R \mathbf{R}_\infty = \mathbf{R}_1 \quad (10.9)$$

$$(\mathbf{I} - \mathbf{T}_R) \mathbf{R}_\infty = \mathbf{R}_1 \quad (10.10)$$

$$\mathbf{R}_\infty = (\mathbf{I} - \mathbf{T}_R)^{-1} \mathbf{R}_1 \quad (10.11)$$

¹⁰ This formulation is equivalent to a Markov reward process with an immediate reward of 1 for all states in the target set and 0 otherwise. The states in the target set are terminal states.

Algorithm 10.6 implements infinite-horizon probabilistic reachability by converting the graph representation of the system to matrix form and solving the system of linear equations in equation (10.11). Example 10.7 shows the results on the grid world problem for different slip probabilities.

```
struct ProbabilisticInfiniteHorizon <: ReachabilityAlgorithm end

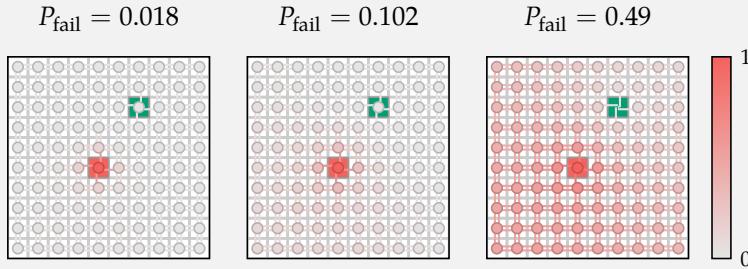
function reachable(alg::ProbabilisticInfiniteHorizon, sys, ψ)
    S, g, dist = states(sys.env), to_graph(sys), Ps(sys.env)
    STi = [index(g, s) for s in ψ.set]
    R1 = [i ∈ STi ? 1.0 : 0.0 for i in eachindex(S)]
    TR = to_matrix(g)
    TR[STi, :] .= 0
    R∞ = (I - TR) \ R1
    return sum(R∞[i] * pdf(dist, state(g, i)) for i in eachindex(S))
end
```

Algorithm 10.6. Infinite-horizon probabilistic reachability for discrete systems. The algorithm creates \mathcal{R}_1 and \mathcal{T}_R from the graph representation of the system. The `to_matrix` function converts the graph to a transition matrix representation. The transition matrix can be represented as a sparse matrix if memory is constrained. The algorithm uses equation (10.11) to compute the infinite-horizon reachability probability from each state and uses the initial state distribution to compute the overall probability of reaching the target set.

10.5 Discrete State Abstractions

The methods discussed in this chapter apply only to discrete systems. However, we can use them to produce overapproximate reachability results for continuous systems by creating a *discrete state abstraction* (DSA). To create a discrete state

Suppose we want to understand the probability of reaching the obstacle state for grid world problems with different slip probabilities. The plots below show the results of infinite-horizon reachability analysis with the obstacle as the target set for slip probabilities of 0.3, 0.5, and 0.7. For each slip probability, we compute P_{fail} assuming we start in the bottom left corner of the grid.



As the probability of slipping increases, the probability of reaching the obstacle state also increases, especially for states near the obstacle.

abstraction, we partition the continuous state space into a finite number of smaller regions. We then create a graph where the nodes correspond to the regions, and the edges correspond to transitions between regions. Figure 10.10 shows the process of creating a DSA for the inverted pendulum problem.

10.5.1 Reachable Sets

To obtain overapproximate reachable sets of continuous systems using a DSA, it is important to ensure that we overapproximate the transitions between regions. In other words, if there exists a state in region $S^{(i)}$ that can transition to a state in region $S^{(j)}$ in one step, we add an edge between the nodes corresponding to $S^{(i)}$ and $S^{(j)}$ in the graph. This rule creates an overapproximation since there may be some states in $S^{(i)}$ that cannot reach $S^{(j)}$ in one step.

For continuous systems with bounded disturbances, we can calculate the reachable set using the algorithms in chapters 8 and 9 to determine the connectivity of the graph. For each region in the partition $S^{(i)} \in \mathcal{S}$, we use a forward reachability algorithm to compute the exact or overapproximate one-step reachable set $\mathcal{R}^{(i)}$. For any region $S^{(j)} \in \mathcal{S}$ that intersects with $\mathcal{R}^{(i)}$, we add an edge from

Example 10.7. Infinite-horizon probability of reaching the obstacle for different slip probabilities in the grid world problem.

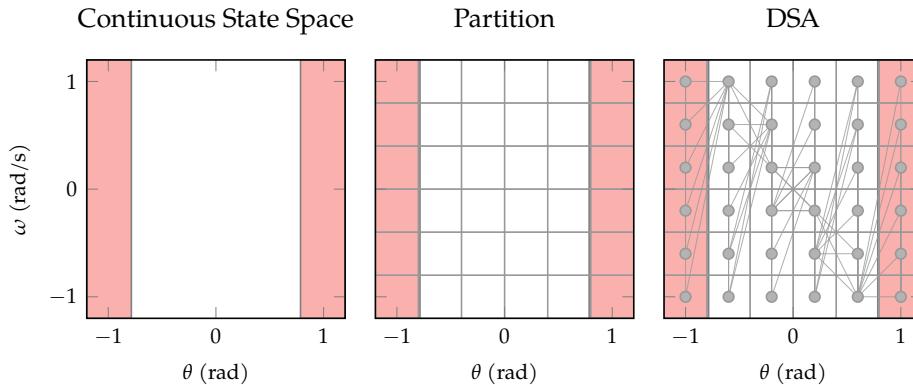


Figure 10.10. Process of creating a discrete state abstraction for the inverted pendulum problem. In this particular example, we partition the continuous state space uniformly into 36 regions. We then create a graph where the nodes correspond to the regions and the edges correspond to the possible transitions between regions.

the node corresponding to $\mathcal{S}^{(i)}$ to the node corresponding to $\mathcal{S}^{(j)}$. Example 10.8 implements this process to create a DSA for the inverted pendulum problem using algorithm 10.1.

Once we have the graph representation of the DSA, we can apply algorithms 10.2 and 10.3 to determine its forward and backward reachable sets. We can then use these results to determine overapproximate reachable sets for the continuous system. Specifically, the overapproximate reachable set for the continuous system is the union of all regions that correspond to a reachable node in the DSA. Figure 10.11 shows this process for the inverted pendulum system.

The choice of partition in the DSA affects the amount of overapproximation error in the reachable sets. In general, a finer partition will result in less overapproximation error at the cost of increased computational complexity (figure 10.12). The examples in this chapter use a uniform partitioning strategy, which may be computationally prohibitive for high-dimensional systems. Adaptive partitioning strategies reduce the number of regions while maintaining a desired level of accuracy.¹¹

10.5.2 Probabilistic Reachability

For probabilistic reachability, the edges in the graph representation of the DSA correspond to overapproximate transition probabilities. Specifically, the weight on the edge from region $\mathcal{S}^{(i)}$ to $\mathcal{S}^{(j)}$ must be greater than equal to the probability that any state in $\mathcal{S}^{(i)}$ transitions to a state in $\mathcal{S}^{(j)}$. The calculation of these

¹¹ S. M. Katz, K. D. Julian, C. A. Strong, and M. J. Kochenderfer, “Generating Probabilistic Safety Guarantees for Neural Network Controllers,” *Machine Learning*, vol. 112, pp. 2903–2931, 2023.

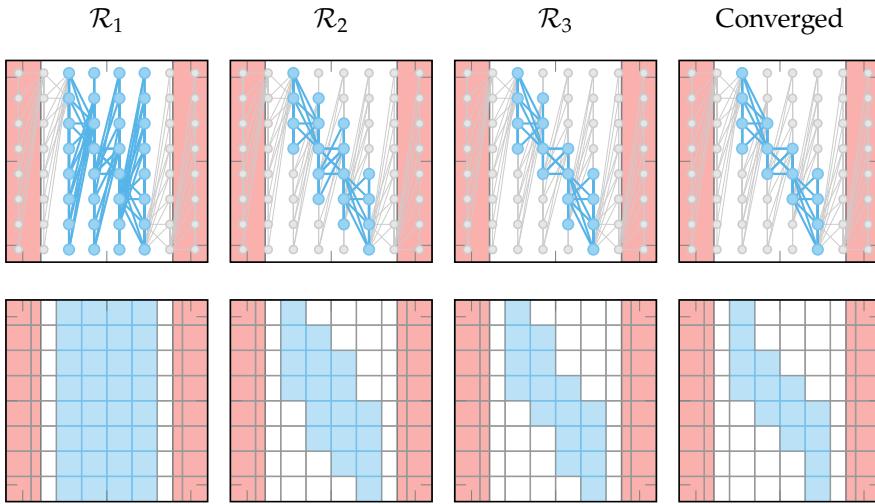


Figure 10.11. Forward reachability for the inverted pendulum system using a discrete state abstraction. The top row shows the reachable sets (blue) of the DSA, and the bottom row shows the corresponding reachable sets (blue) in the continuous system. The x -axis represents the angle of the pendulum, and the y -axis represents the angular velocity.

overapproximated probabilities is system specific. Example 10.9 demonstrates this process for a continuum world problem with Gaussian disturbances on its transitions. Given these transition probabilities, we can apply algorithm 10.4 or algorithm 10.5 to determine the overapproximate probabilities of occupying or reaching a set of target states.¹²

10.6 Summary

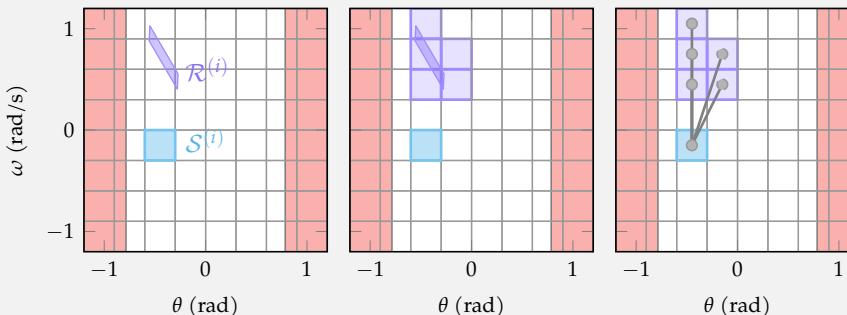
- We can represent discrete systems as directed graphs where the nodes represent states and the edges represent transitions between states.
- Forward reachable sets for discrete systems can be computed by applying breadth-first search from a set of initial states.
- Backwards reachability algorithms begin with a set of target states and calculate the set of states that can reach the target set in a given time horizon.
- If our only goal is check whether a system satisfies a reachability specification, we may be able to use more efficient algorithms that do not directly compute reachable sets such as heuristic search or Boolean satisfiability.

¹² Since the transition probabilities are overapproximations, we may calculate intermediate overapproximate probabilities that are greater than 1. In these cases, these probabilities should be clamped to a value of 1.

We can create a DSA for the inverted pendulum system using algorithm 10.1 by defining the `states` function to partition the state space into a grid of regions and the `successors` function to determine the connectivity of the graph using a nonlinear forward reachability technique such as conservative linearization. Example implementations are as follows:

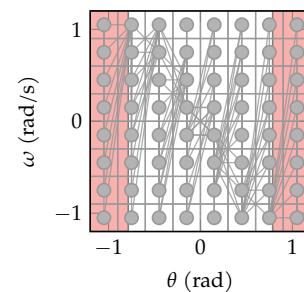
```
function states(env::InvertedPendulum; nθ=8, nw=8)
    θs, ws = range(-1.2, 1.2, length=nθ+1), range(-1.2, 1.2, length=nw+1)
    S = [Hyperrectangle(low=[θlo, ωlo], high=[θhi, ωhi])
        for (θlo, θhi) in zip(θs[1:end-1], θs[2:end])
        for (ωlo, ωhi) in zip(ws[1:end-1], ws[2:end])]
    return S
end
function successors(sys, S(i))
    - , X = sets(sys, 2)
    R(i) = conservative_linearization(sys, S(i) × X)
    R(i) = VPolytope([clamp.(v, -1.2, 1.2) for v in vertices_list(R(i))])
    S(j)s = filter(S(j) → !isempty(R(i) ∩ S(j)), states(sys.env))
    return S(j)s, ones(length(S(j)s))
end
```

The plots below demonstrate the `successors` function on an example state $S^{(i)}$. The function first computes $\mathcal{R}^{(i)}$ using conservative linearization (left). It then determines the regions $S^{(j)}$ that intersect with $\mathcal{R}^{(i)}$ (middle). Finally, the function returns these regions so that they can be connected in the graph (right). The edge weights can be ignored when computing reachable sets.



Algorithm 10.1 calls the `successors` function for each region in the partition to determine the connectivity of the graph. The result is shown in the caption.

Example 10.8. Creating a DSA for the inverted pendulum system using algorithm 10.1. The plots show the process of determining the connectivity of the graph for a single region $S^{(i)}$. The plot below shows the graph for the final DSA with a uniform partition of the state space into 64 regions.



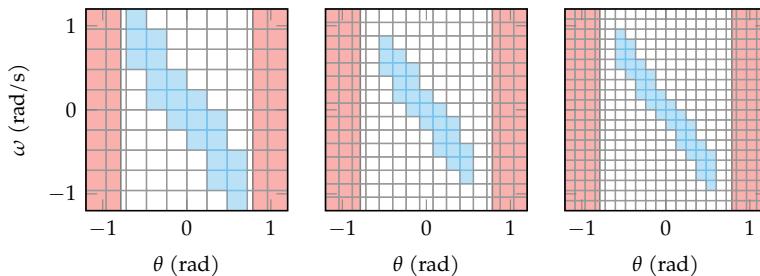


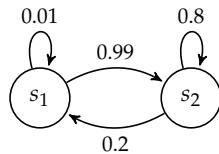
Figure 10.12. Converged overapproximate forward reachable sets for the inverted pendulum using different resolutions of the DSA. As the resolution increases, the overapproximation error decreases.

- Probabilistic reachability analysis allows us to compute the probability of reaching a set of target states in a finite or infinite time horizon.
- We can convert continuous systems into discrete systems by producing a discrete state abstraction of the continuous system.
- We can apply the reachability algorithms for discrete systems to a DSA to determine overapproximate reachable sets for its corresponding continuous system.

10.7 Exercises

Exercise 10.1. Consider a system with two states $\mathcal{S} = \{s_1, s_2\}$ and two actions $\mathcal{A} = \{a_1, a_2\}$. When the agent takes action a_1 , the agent remains in its current state with probability 0.01 and transitions to the other state with probability 0.99. When the agent takes action a_2 , the agent remains in its current state with probability 0.8 and transitions to the other state with probability 0.2. The agent takes action a_1 in state s_1 and action a_2 in state s_2 . Create a graph to represent this system.

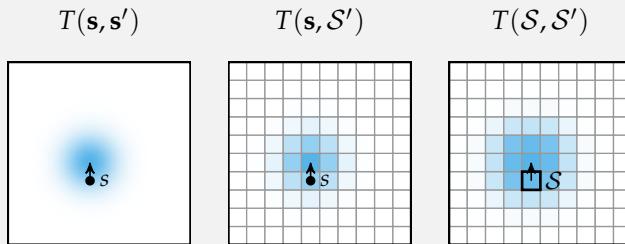
Solution: The following graph represents the system:



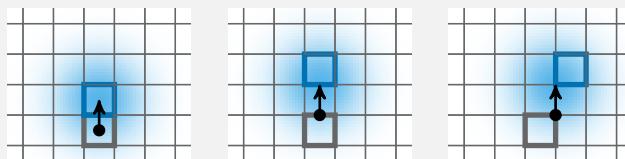
Exercise 10.2. In what situations might we prefer backward reachability to forward reachability?

Suppose we have a continuum world problem with Gaussian disturbances on its transitions. For example, if the agent takes the up action, its next position is sampled from a Gaussian distribution with a mean 1 unit above its current state and a standard deviation of 1 in each direction. In other words, $T(\mathbf{s}, \mathbf{s}') = \mathcal{N}(\mathbf{s}' | \mathbf{s} + \mathbf{d}, \mathbf{I})$ where \mathbf{d} is the direction vector corresponding to the action taken in the state \mathbf{s} . Our goal is to determine the overapproximated transition probabilities $T(\mathcal{S}, \mathcal{S}')$ for a DSA of the continuous system.

To obtain the probability of transitioning from a specific state \mathbf{s} to a region in the partition \mathcal{S}' , we integrate the transition function such that $T(\mathbf{s}, \mathcal{S}') = \int_{\mathcal{S}'} T(\mathbf{s}, \mathbf{s}') d\mathbf{s}'$. To obtain an overapproximation of the transition probabilities, we select the transition from the current region \mathcal{S} that results in the highest probability of reaching the target region \mathcal{S}' such that $T(\mathcal{S}, \mathcal{S}') = \max_{\mathbf{s} \in \mathcal{S}} T(\mathbf{s}, \mathcal{S}')$. The plots below show the transition probabilities for a single state s to the regions in the DSA. The plots below demonstrate this process.



The maximization in the formula for $T(\mathcal{S}, \mathcal{S}')$ finds the state in \mathcal{S} that puts the highest amount of probability mass in \mathcal{S}' . The plots below demonstrate this maximization for three different next regions \mathcal{S}' . This process produces an overapproximation of the transition probabilities since we assume all states in \mathcal{S} transition to the worst-case next state.



Example 10.9. Overapproximation the transition probabilities for the DSA of the continuum world system.

Solution: We might prefer backward reachability to forward reachability if we do not know the exact set of initial states for the system. Backward reachability will determine whether the target set is reachable for all states.

Exercise 10.3. Suppose we are applying discrete forward reachability on a system with states $\mathcal{S} = \{A, B, C, D, E, F, G\}$. We want the system to reach the goal state G when starting from initial state A . We apply the reachability algorithm for 4 time steps and obtain the following results:

$$\begin{aligned}\mathcal{R}_1 &= \{A\} \\ \mathcal{R}_2 &= \{B, C, E\} \\ \mathcal{R}_3 &= \{B, C\} \\ \mathcal{R}_4 &= \{B\}\end{aligned}$$

Is it possible for the system to reach the goal state G from state A ? Why or why not?

Solution: It is not possible to reach the goal state G from state A because G is not reached during any of the first 4 time steps and $\mathcal{R}_4 = \{B\}$ is an invariant set because $\mathcal{R}_4 \subseteq \mathcal{R}_3$. Therefore, the system will remain in state B for all time in the future after time step 4 and will not reach the goal.

Exercise 10.4. When performing counterexample search for systems with avoid set specifications, we can terminate our reachability analysis early and conclude that the system is unsafe if we find a reachable state that is contained in the avoid set. Could we apply similar logic when proving that a system will always reach a target set? Why or why not?

Solution: We cannot necessarily terminate early when proving that the system will always reach a target set. Finding a reachable state in the target set only tells us that there is at least one path from the initial state set to the target set. However, it does not tell us that all possible paths from the initial state set will reach the target set.

Exercise 10.5. Suppose we perform 20 steps of finite-horizon probabilistic reachability analysis using algorithm 10.5 on the grid world problem and find that the probability of reaching the obstacle state from an initial state in the lower left corner is 0.005. We also perform 1,000 rollouts of depth 20 from the same initial state in the bottom left corner and observe 4 trajectories that reach the obstacle, resulting in a probability of failure estimate of 0.004 using the methods from chapter 7. Why do these numbers not match? Which number should we trust more?

Solution: The methods in chapter 7 only provide estimates of the probability of failure based on a set of samples and are not guaranteed to result in the exact probability of failure. In contrast, the finite-horizon technique for probabilistic reachability discussed in this chapter provides an exact calculation of the probability of failure. Therefore, we should trust the results from the finite-horizon analysis more.

Exercise 10.6. Consider the system shown in figure 10.1 and assume the initial state for the system is s_1 . What is the probability that the system is in state s_2 at time step 3?

Solution: Since the system begins at time step 1 in state s_1 , $P_1(s_1) = 1$ and $P_1(s_2) = 0$. We can calculate $P_t(s_1)$ and $P_t(s_2)$ recursively as follows:

$$\begin{aligned} P_2(s_1) &= T(s_1, s_1)P_1(s_1) + T(s_2, s_1)P_1(s_2) \\ &= (0.2)(1) + (0.9)(0) \\ &= 0.2 \end{aligned}$$

$$\begin{aligned} P_2(s_2) &= T(s_1, s_2)P_1(s_1) + T(s_2, s_2)P_1(s_2) \\ &= (0.8)(1) + (0.1)(0) \\ &= 0.8 \end{aligned}$$

$$\begin{aligned} P_3(s_1) &= T(s_1, s_1)P_2(s_1) + T(s_2, s_1)P_2(s_2) \\ &= (0.2)(0.2) + (0.9)(0.8) \\ &= 0.76 \end{aligned}$$

$$\begin{aligned} P_3(s_2) &= T(s_1, s_2)P_2(s_1) + T(s_2, s_2)P_2(s_2) \\ &= (0.8)(0.2) + (0.1)(0.8) \\ &= 0.24 \end{aligned}$$

Therefore, the probability that the system is in state 2 at time step 3 is 0.24.

Exercise 10.7. Consider the system shown in figure 10.1 and assume the initial state for the system is s_1 . What is the probability that the system reaches state s_2 within the first 3 time steps?

Solution: Our target set is $\{s_2\}$, so $R_1(s_1) = 0$ and $R_1(s_2) = 1$. Our goal is to find $R_3(s_1)$. We can calculate $R_t(s_1)$ and $R_t(s_2)$ recursively as follows:

$$\begin{aligned} R_2(s_1) &= T(s_1, s_1)R_1(s_1) + T(s_1, s_2)R_1(s_2) \\ &= (0.2)(0) + (0.8)(1) \\ &= 0.8 \end{aligned}$$

$$R_2(s_2) = 1$$

$$\begin{aligned} R_3(s_1) &= T(s_1, s_1)R_2(s_1) + T(s_1, s_2)R_2(s_2) \\ &= (0.2)(0.8) + (0.8)(1) \\ &= 0.96 \end{aligned}$$

Therefore, the probability that the system reaches state s_2 within the first three time steps is 0.96.

Exercise 10.8. When we compute the connections for a discrete state abstraction, we need to compute the one-step reachable set for each cell and determine which cells intersect the reachable set. Suppose we have an \mathcal{H} -polytope representation of the reachable set for a particular cell, and we want to determine if it overlaps with cell \mathcal{C} , which is also represented as an \mathcal{H} -polytope. Write a linear program we could solve to perform this check.

Solution: We need to check if there is a point that satisfies the linear inequalities for both polytopes. Let $\mathbf{A}_r \mathbf{s} \leq \mathbf{b}_r$ be the set of linear inequalities for the \mathcal{H} -polytope representation of the reachable set \mathcal{R} , and $\mathbf{A}_c \mathbf{s} \leq \mathbf{b}_c$ be the set of linear inequalities for the \mathcal{H} -polytope representation of cell \mathcal{C} . We can solve the following linear program to check for overlap:

$$\begin{aligned} & \text{minimize} && 0 \\ & \text{subject to} && \mathbf{A}_r \mathbf{s} \leq \mathbf{b}_r \\ & && \mathbf{A}_c \mathbf{s} \leq \mathbf{b}_c \end{aligned}$$

If the linear program is feasible, then the two polytopes overlap.

11 Explainability

This chapter focuses on understanding system behavior through *explanations*. An explanation is a description of a system’s behavior that helps a human understand why it behaves in a particular manner. In this chapter, we discuss several types of explanations. We begin by discussing policy visualization techniques that allow us to interpret an agent’s policy. We then discuss feature importance techniques to help us understand the input features that are most important to the behavior of a system. For systems with complex policies, we discuss ways to create interpretable surrogate models that approximate the system’s behavior. We also discuss techniques for generating counterfactual explanations that change the outcome of a particular scenario by making small changes to important features. We conclude by introducing methods to categorize the failure modes of a system.

11.1 Explanations

We often desire explanations of system behavior when metrics such as failure probabilities or reachable sets are insufficient for an adequate understanding of the system. For example, it may be impossible to capture all possible edge-case behaviors when creating a model of a complex system, which may cause us to miss potential failure modes. We also may not be able to fully specify our objectives for a system using metrics or logical specifications. This incompleteness may lead to an alignment problem (section 1.1) in which the metrics and specifications used to evaluate a system do not perfectly capture our true objectives. In this case, explanations of system behavior may provide a better understanding of whether the behavior is aligned with our objectives. The results can be used to debug the system by informing changes to the policy, model, or specifications.

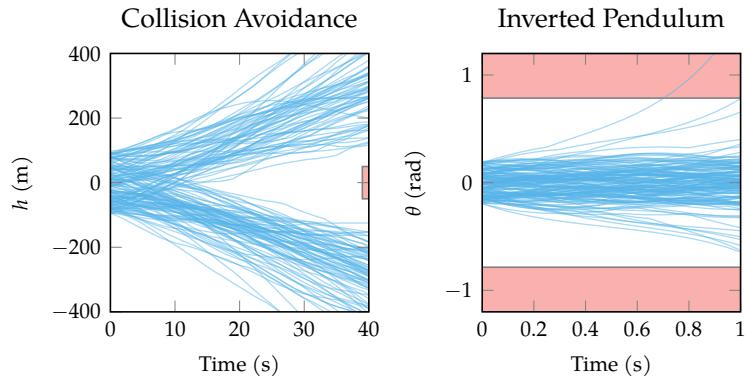


Figure 11.1. Visualization of the policies for the collision avoidance and inverted pendulum systems by plotting trajectory rollouts. We plot time on the horizontal axis and one of the state variables on the vertical axis.

Explanations are also important to the stakeholders of a system. They can be used to calibrate trust of end users by providing insight into a system’s decision-making process. This insight helps users understand the strengths of a system as well as its potential weaknesses. Explanations can also help stakeholders check the fairness of a system by identifying the factors that influence its decisions. Moreover, the end users of high-stakes decision-making systems such as loan approval systems often have a right to an explanation. The General Data Protection Regulation (GDPR) in the European Union requires that users be provided with an explanation of automated decisions that significantly affect them.¹

The algorithms presented in this chapter provide descriptions of system behavior to human operators or stakeholders. A good description should be interpretable to humans in a way that allows them to explain and predict the system’s behavior.² The *interpretability* of a description is the degree to which it can be readily parsed by humans. For example, a small decision tree with only a few nodes tends to be more interpretable than a large decision tree with hundreds of nodes. The *explainability* of a description is the degree to which it helps humans understand why a system behaves in a particular way.³

11.2 Policy Visualization

One way to understand the behavior of an agent is to visualize its policy in a way that is readily interpretable by humans. For example, we might visualize how the policy affects the state of the system over time. We can generate these

¹ B. Goodman and S. Flaxman, “European Union Regulations on Algorithmic Decision-Making and a ‘Right to Explanation’,” *AI Magazine*, vol. 38, no. 3, pp. 50–57, 2017.

² J. Colin, T. Fel, R. Cadène, and T. Serre, “What I Cannot Predict, I Do Not Understand: A Human-Centered Evaluation Framework for Explainability Methods,” *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 2832–2845, 2022.

³ These definitions are often used interchangeably depending on the context.

plots by performing rollouts of the policy and plotting the state of the system at each time step. This visualization can help us understand how the policy behaves in different scenarios and identify potential failure modes. Figure 11.1 shows an example of this visualization technique for the aircraft collision avoidance and inverted pendulum policies.

If the policy is Markov and therefore depends only on the current state, we can also visualize it directly by plotting the action taken by the agent in each state. If the state space is two-dimensional as in the inverted pendulum example, we can plot the action taken by the agent as a two-dimensional heatmap (figure 11.2). For higher-dimensional state spaces, we often need to apply dimensionality reduction techniques to visualize the policy. One common technique is to fix all but two of the state variables, which become associated with the vertical and horizontal axes. We can indicate the action for every state with a color. Example 11.1 demonstrates this technique for the collision avoidance policy.

Instead of fixing the hidden state variables, we could also use various techniques to aggregate over them (figure 11.3). One method involves partitioning the state space into a set of regions and keeping track of the actions taken in each region over a series of rollouts. We can then aggregate over these actions by plotting the mean or mode of the actions taken in each region. One benefit of this technique is that it relies only on rollouts of the policy and therefore extends to non-Markovian policies. Because all states may not be reachable in practice, some areas of the policy plot may have no data associated with it.

11.3 Feature Importance

Feature importance algorithms allow us to understand the contribution of various input features to the overall behavior of a system. We can use this analysis, for example, to identify the features of an observation that are most important to the agent’s decision or to identify the disturbances in a trajectory that have the greatest effect on its outcome. In this section, we use the term *feature* to refer to any component of a system trajectory that might affect the outcome. Features could include the states, observations, actions, or disturbances of the system. We can also derive more complex features by combining these basic features. For example, we could create features for an aircraft collision avoidance system by grouping states together into different configurations that represent different relative positions of our aircraft and the intruder.

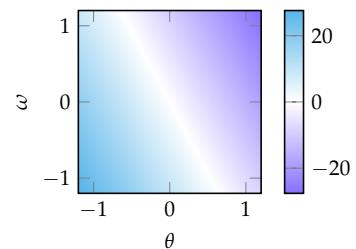
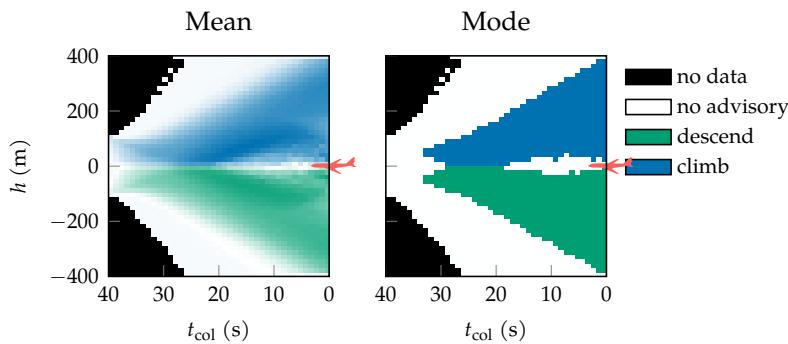
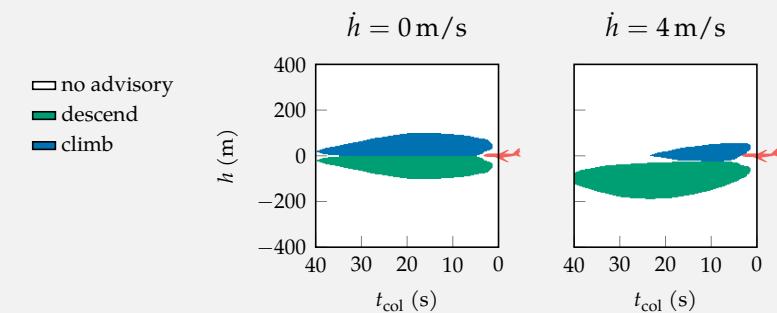


Figure 11.2. Visualization of the actions taken by the inverted pendulum policy. The colors represent the torque applied in each state.

The figure in the caption shows policy plots for the aircraft collision avoidance policy when the relative vertical rate is fixed at 0 m/s and 4 m/s, and the previous action is fixed at no advisory. The red aircraft represents the relative location of the intruder aircraft. We can use these plots to explain the behavior of the policy in these scenarios. For example, we can see that when the relative vertical rate is fixed at zero, the policy advises our aircraft to climb when it is above the intruder and descend when it is below the intruder. This behavior is aligned with our objective of avoiding collisions.

The plot on the left also reveals some potentially unexpected behaviors. For example, when the time to collision is near zero and a collision is imminent, the policy results in no advisory. This behavior may prompt us to perform further analysis. For example, a counterfactual analysis (see section 11.5) reveals that a collision is inevitable in this scenario regardless of the action taken by the agent due to limits on the vertical rate of the aircraft.



Example 11.1. Aircraft collision avoidance policy when the relative vertical rate is fixed at 0 m/s (left) and 4 m/s (right), and the previous action is fixed at no advisory. The colors represent the action taken by the agent in each state.

Figure 11.3. Result of different aggregation methods for plotting the four-dimensional collision avoidance policy using data from 10,000 rollouts. On the left, we plot the mean of the vertical rate action taken by the agent in each state. On the right, we plot the action taken most frequently by the agent in each state.

The results of a feature importance analysis can lead to explanations of system behavior that allow us to check fairness and calibrate trust. For example, we could use feature importance to ensure that a loan approval system is not focusing on protected characteristics such as race or gender when making decisions. We could also use feature importance to calibrate trust by ensuring that the agent is focusing on the features required to make a decision. For instance, we could check that an image classifier is not focusing on irrelevant portions of the image when making decisions.⁴ We can also use feature importance to inform the design of future systems by identifying the features that are most important in causing the system to fail.

There are a variety of ways to define the importance of a particular feature. One definition involves holding all features other than the feature of interest constant and observing the effect on the system's behavior when varying that feature. Techniques that use this definition are often referred to as *sensitivity analysis* techniques.⁵ This definition, however, focuses only on the effect of the feature of interest by itself and does not consider its interaction with other features. Another definition of feature importance involves examining the effect of the feature of interest in the context of the other features. Example 11.2 provides a scenario where considering the interactions between features produces a different result. This section presents techniques for determining feature importance using both definitions.

11.3.1 Sensitivity Analysis

Sensitivity analysis techniques allow us to understand how a particular output changes when a single feature is changed. Examples of possible outputs include the robustness of a trajectory or the agent's decision at a single time step. If we change the value of an input that has high sensitivity, we expect a large change in the output, while if we change an input with low sensitivity, we expect a small change in the output. Figure 11.4 illustrates this concept.

One way to approximate sensitivity is to randomly perturb a single feature and observe the effect on the system's behavior. By repeating this process multiple times and taking the standard deviation or some other variability metric of the outcome of each trial, we obtain a measure of sensitivity. To evaluate the sensitivity of a decision at a single point in time, we often perturb one component of the observation and observe the effect on the decision (example 11.3). To evaluate the

⁴ We want to ensure that the classifier is not exploiting *spurious correlations* between the input and output. Neural networks are prone to this type of behavior. R. Geirhos, J.-H. Jacobsen, C. Michaelis, R. Zemel, W. Brendel, M. Bethge, and F. A. Wichmann, "Shortcut Learning in Deep Neural Networks," *Nature Machine Intelligence*, vol. 2, no. 11, pp. 665–673, 2020.

⁵ We could also imagine corrupting all features except the feature of interest and observing the effect on the system's behavior. This technique is sometimes referred to as causal mediation analysis. J. Pearl, "Direct and Indirect Effects," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2001.

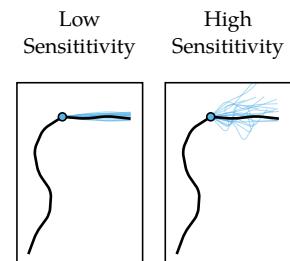
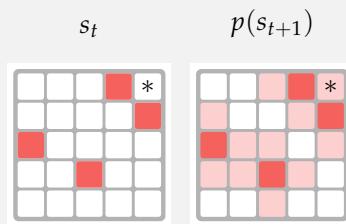
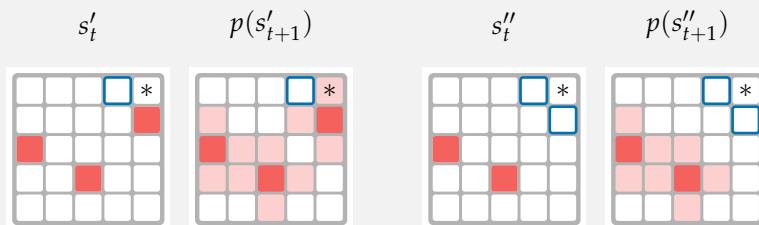


Figure 11.4. The trajectories show the effect of randomly changing the disturbance at a single time step on the rest of the trajectory (blue). The system on the right has a higher sensitivity to the disturbance applied at the time step than the system on the left, resulting in a wider variety of outcomes.

Consider a wildfire scenario modeled as a grid where each cell is either burning or not burning. At each time step, there is a 30% chance that a cell that was not burning at the previous time step will be burning if at least one of its neighbors was burning. The plots below show an example of a current state s_t and the probability that each cell is burning at the next time step $p(s_{t+1})$ (darker cells indicate higher probability). Suppose we are interested in understanding the features that are most important in determining the probability that the cell in the upper right corner will burn.



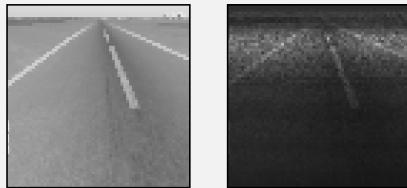
For this example, we will focus specifically on the feature that indicates whether the cell directly to the left of the upper right cell is burning. We can test the first definition of feature importance by changing that cell to not burning while holding all other cells constant and observing the effect on the probability that the upper right cell will burn. In this case (leftmost plots), the probability that the upper right cell will be burning at the next time step does not change. Therefore, we will conclude that this cell has no contribution to the output. However, if we remove fire from both this cell and the cell below the upper right cell (rightmost plots), the upper right cell changes to zero probability of burning at the next time step. The second definition of feature importance considers the interaction between these two features and would conclude that the cell does contribute to the output.



Example 11.2. Motivation for considering the interactions between features when determining feature importance.

Suppose we have an agent that selects a steering angle for an aircraft based on runway images from a camera mounted on its wing. Given a particular input image, we can generate a sensitivity map to identify the pixels that are most important in determining the steering angle by fixing all but the pixel of interest and checking its effect on the steering angle output. The results are shown below, where the left image is the original image and the right image is the sensitivity map. This analysis indicates that the agent is focusing on the portion of the runway in front of it where the lines are most visible.

Original Image Sensitivity Map



Example 11.3. Sensitivity analysis at a single time step. Brighter pixels in the sensitivity map indicate pixels with higher sensitivity.

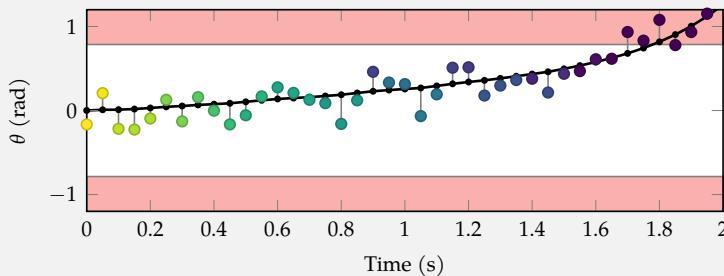
sensitivity of a trajectory, we can perturb the disturbance, observation, or action at one time step and measure the effect on the rest of the trajectory (example 11.4).

Algorithm 11.1 estimates the sensitivity of the robustness of a particular trajectory with respect to the disturbance at each time step. For each time step in the trajectory, the algorithm perturbs the disturbance m times and computes the robustness of the resulting trajectories. The algorithm then returns the standard deviation of the change in robustness for each time step.⁶ The sensitivity of the robustness of a trajectory with respect to its disturbances can be used to identify the disturbances that have the greatest effect on the outcome of the trajectory.

Because algorithm 11.1 requires performing multiple rollouts for each time step, it tends to be inefficient for high-dimensional systems with many disturbances and long time horizons. If the output of interest is differentiable with respect to the input features, we can reduce computational cost by calculating the sensitivity using *saliency maps*. Saliency maps are a type of sensitivity map that use gradients to identify inputs that are most important, or salient, in determining a particular outcome. We can apply saliency maps to measures the sensitivity of both individual decisions and the outcomes of full trajectories.

⁶ This direct sampling algorithm is one of many different approaches to quantify the uncertainty in the output of a function given uncertainty in the input. Other methods such as those that use Taylor series approximations and polynomial chaos can be used instead. See the “Uncertainty Propagation” chapter in M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019.

We can use sensitivity analysis to understand the effect of disturbances on the outcome of a trajectory. For example, consider an inverted pendulum system in which the agent's observation of its current angle is subject to a noise disturbance. We can estimate the sensitivity of the robustness of a trajectory with respect to its disturbances by perturbing the disturbances at each time step and observing the effect on the robustness of the trajectory. The results on a given failure trajectory are shown below. This analysis indicates that small changes in the disturbances at the beginning of the trajectory have a large effect on the robustness of the trajectory. Furthermore, the disturbances applied towards the end of the failure trajectory have little to no effect because the controller is saturated and the system cannot recover.



Example 11.4. Sensitivity analysis over a full trajectory. Brighter colors in the sensitivity map of the inverted pendulum trajectory indicate higher sensitivity. The black line shows the true angle of the pendulum at each time step and the colored markers indicate the noisy observation of the current angle at each time step.

```

struct Sensitivity
    x          # vector of trajectory inputs (s, x = extract(sys.env, x))
    perturb # x' = perturb(x, t)
    m          # number of samples per time step
end

function describe(alg::Sensitivity, sys,  $\psi$ )
    m, x, perturb = alg.m, alg.x, alg.perturb
    s, x = extract(sys.env, x)
    τ = rollout(sys, s, x)
     $\rho_0$  = robustness([step.s for step in τ],  $\psi.formula$ )
    sensitivities = zeros(length(τ))
    for t in eachindex(τ)
        x's = [perturb(x, t) for i in 1:m]
        τ's = [rollout(sys, extract(sys.env, x')... for x' in x's]
         $\rho_s$  = [robustness([st.s for st in τ'],  $\psi.formula$ ) for τ' in τ's]
        sensitivities[t] = std(abs.( $\rho_s$  .-  $\rho_0$ ))
    end
    return sensitivities
end

```

Algorithm 11.1. Algorithm for estimating the sensitivity of the robustness of a trajectory with respect to its disturbances. It takes as input a vector of trajectory features for the current trajectory we want to evaluate. These feature can be converted to a trajectory using the system specific `extract` function. The `perturb` function generates a new trajectory feature vector by perturbing the feature at a particular time step. The algorithm then computes the robustness of the perturbed trajectories and returns the standard deviation of the resulting change in robustness for each time step.

```

struct GradientSensitivity
    x # vector of trajectory inputs (s, x = extract(sys.env, x))
end

function describe(alg::GradientSensitivity, sys,  $\psi$ )
    function current_robustness(x)
        s, x = extract(sys.env, x)
        t = rollout(sys, s, x)
        return robustness([step.s for step in t],  $\psi$ .formula)
    end

    return ForwardDiff.gradient(current_robustness, alg.x)
end

```

Algorithm 11.2. Algorithm for approximating the sensitivity of the robustness of a trajectory with respect to its disturbances using gradients. It takes as input a vector of trajectory features for the current trajectory we want to evaluate. These feature can be converted to a trajectory using the system specific `extract` function. The algorithm computes the robustness of the trajectory and returns the gradient of the robustness with respect to the input features.

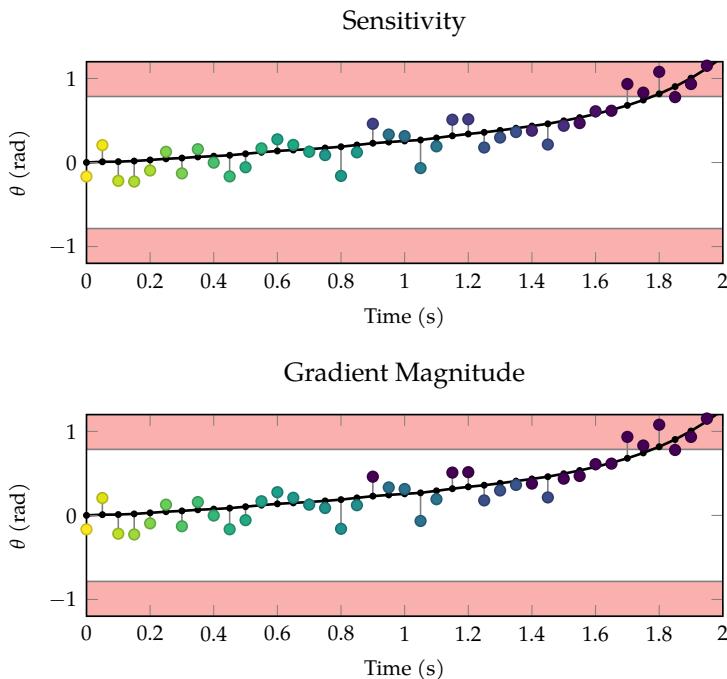


Figure 11.5. Sensitivity of the robustness of a trajectory with respect to its disturbances for an inverted pendulum system calculated using algorithm 11.2 compared to using algorithm 11.1. The black line shows the true angle of the pendulum at each time step and the colored markers indicate the noisy observation of the current angle at each time step. Brighter colors indicate higher sensitivity. The gradient calculation provides values similar to the sensitivity estimate from algorithm 11.1.

A simple way to produce a saliency map given a set of inputs is to take the gradient of the output of interest with respect to the inputs.⁷ The saliency of a particular input is related to the magnitude of the gradient at that input. A high gradient magnitude indicates that small changes in the input will result in large changes in the output. In other words, inputs with high gradient values are more salient and indicate higher sensitivity. This method is often used to determine the components of an observation (such as the pixels of an image) that contribute most to an agent's decision.⁸ We can also use it to approximate sensitivity over a full trajectory by taking the gradient of a performance measure with respect to input features such as actions or disturbances. Algorithm 11.2 measures the sensitivity of the robustness of a trajectory with respect to its disturbances, and figure 11.5 shows an example on the inverted pendulum system.

While algorithm 11.2 is more computationally efficient than algorithm 11.1, it is limited by its local nature. Important input features, for example, often saturate the output function of interest, causing the gradient to be small even when the feature is important.⁹ The *integrated gradients*¹⁰ algorithm addresses this limitation by averaging the gradient along the path between a baseline input and the input of interest (figure 11.6). The choice of baseline depends on the context. For images, a common choice is a black image (figure 11.7). For disturbances, we can set all disturbances to zero.

Algorithm 11.3 calculates the sensitivity of the robustness of a trajectory with respect to the disturbances at each time step using integrated gradients. It takes m steps along the path between the baseline and the current input and computes the gradient of the robustness at each step. The algorithm then returns the av-

⁷ D. Baehrens, T. Schroeter, S. Harmeling, M. Kawanabe, K. Hansen, and K.-R. Müller, "How to Explain Individual Classification Decisions," *Journal of Machine Learning Research*, vol. 11, pp. 1803–1831, 2010.

⁸ K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps," in *International Conference on Learning Representations (ICLR)*, 2014.

⁹ For image inputs in particular, it has also been shown that there are sometimes meaningless local variations in gradients that can lead to noisy sensitivity maps. D. Smilkov, N. Thorat, B. Kim, F. Viégas, and M. Wattenberg, "Smoothgrad: Removing Noise by Adding Noise," in *International Conference on Machine Learning (ICML)*, 2017.

¹⁰ M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic Attribution for Deep Networks," in *International Conference on Machine Learning (ICML)*, 2017.

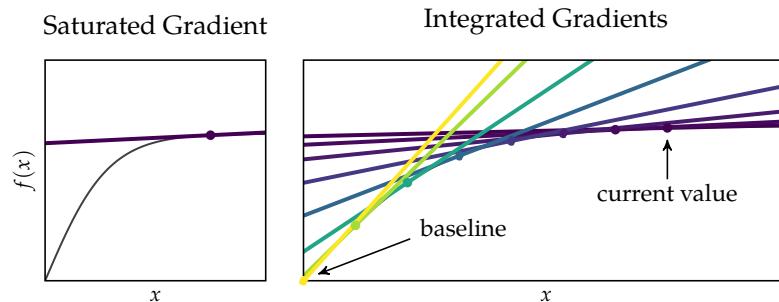


Figure 11.6. Example of the integrated gradients algorithm for an input feature x . While x has a significant effect on the output function $f(x)$, the gradient at the current value of x is small because $f(x)$ is saturated (left). If we average the gradient of the output function $f(x)$ along the path between a baseline input and the current value of the input of interest, we can capture this effect (right). Brighter colors of the gradient lines indicate higher magnitudes.

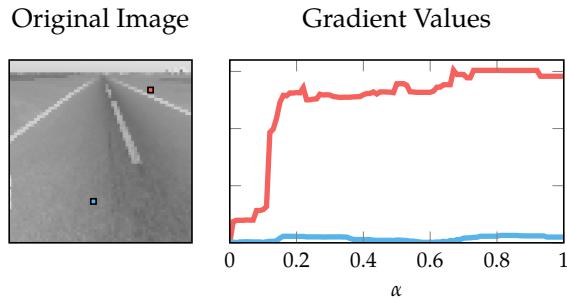


Figure 11.7. Comparison of the gradients of the output of an aircraft taxi network that selects a steering angle based on runway images from a camera mounted on its wing. As α moves from 0 to 1, the image moves from a baseline black image to the original image. The gradients are much higher for the pixel marked in red indicating that it has a larger effect on the output of the network.

```

struct IntegratedGradients
    x # vector of trajectory inputs (s, x = extract(sys.env, x))
    b # vector of baseline inputs
    m # number of steps for numerical integration
end

function describe(alg::IntegratedGradients, sys, ψ)
    function current_robustness(x)
        s, x = extract(sys.env, x)
        τ = rollout(sys, s, x)
        return robustness([step.s for step in τ], ψ.formula)
    end
    as = range(0, stop=1, length=alg.m)
    xs = [(1 - α) * alg.b + α * alg.x for α in as]
    grads = [ForwardDiff.gradient(current_robustness, x) for x in xs]
    return mean(hcat(grads...), dims=2)
end

```

Algorithm 11.3. Algorithm for approximating the sensitivity of the robustness of a trajectory with respect to its disturbances using integrated gradients. It takes as input a vector of trajectory features for the current trajectory we want to evaluate, a vector of baseline features, and the number of steps for numerical integration. For each step along the path between the baseline and the current input, the algorithm computes the gradient of the robustness. It then returns the average gradient at each time step.

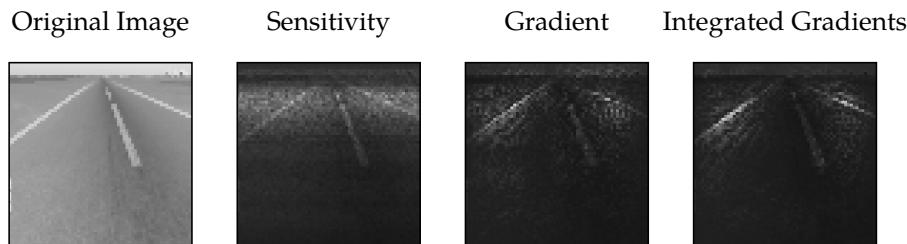


Figure 11.8. Comparison of the sensitivity descriptions using algorithms 11.1 to 11.3 for an aircraft taxi system that selects a steering angle from an image observation. The sensitivity map focuses on the portion where the edge and center lines are most apparent, while the gradient-based methods focus only on the edges of the runway. The integrated gradients method provides a smoother map than the single gradient approach.

erage gradient at each time step. As m approaches infinity, the average gradient approaches the integral of the gradient along the path. Figure 11.8 compares the sensitivity estimates from algorithms 11.1 to 11.3 for an aircraft taxi system. All three methods produce slightly different descriptions of the agent’s behavior, and in general, the most appropriate sensitivity estimate is application dependent.

11.3.2 Shapley Values

Computing the *Shapley value* of a feature allows us to evaluate its importance in the context of its interaction with other features. For example, it may be a combination of multiple disturbances that leads to a failure rather than a single disturbance. While sensitivity analysis techniques miss this interaction because they only vary one feature at a time, Shapley values capture it by varying all possible subsets of features.¹¹

Suppose we have a set of feature indices $\mathcal{I} = \{1, \dots, n\}$, and let $\mathcal{I}_s \subseteq \mathcal{I}$ be a subset of these features. Given a set of values for the features \mathbf{x} and a function f that maps these values to an outcome, we define the following function to represent the expectation of the outcome while holding the features in \mathcal{I}_s constant:

$$f_{\mathcal{I}_s}(\mathbf{x}) = \mathbb{E}[f(\mathbf{x}') \mid x'_i = x_i, i \in \mathcal{I}_s] \quad (11.1)$$

The Shapley value ϕ_i of feature i is then defined as the average marginal contribution of feature i to the expectation of the outcome over all possible subsets of features:

$$\phi_i(\mathbf{x}) = \sum_{\mathcal{I}_s \subseteq \mathcal{I} \setminus \{i\}} \frac{|\mathcal{I}_s|!(n - |\mathcal{I}_s| - 1)!}{n!} (f_{\mathcal{I}_s \cup \{i\}}(\mathbf{x}) - f_{\mathcal{I}_s}(\mathbf{x})) \quad (11.2)$$

Intuitively, computing the Shapley value of feature i involves looping over all possible subsets of features that do not include i and computing the difference in the expectation of the outcome when adding i to the subset. The constant factor in equation (11.2) ensures that subsets of different sizes are weighted equally. In general, Shapley values are expensive (often intractable) to compute due to the large number of possible subsets. For example, a function with 100 input features has 6.3×10^{29} possible subsets.

¹¹ Shapley values were originally developed in the context of game theory in economics and are named for American mathematician and economist Lloyd Shapley (1923–2016). L. S. Shapley, “Notes on the N-Person Game—II: The Value of an N-Person Game,” 1951.

We can approximate the Shapley value using randomly sampled subsets.¹² First, we rewrite equation (11.2) as follows:

$$\phi_i(\mathbf{x}) = \frac{1}{n!} \sum_{\mathcal{P} \in \pi(n)} [f_{\mathcal{P}_{1:j}}(\mathbf{x}) - f_{\mathcal{P}_{1:j-1}}(\mathbf{x})] \quad (11.3)$$

where $\pi(n)$ represents the set of all possible permutations of n elements, j is the index in the permutation \mathcal{P} that corresponds to feature i , and $\mathcal{P}_{1:j}$ represents the first j elements of \mathcal{P} . We can then approximate the Shapley value using sampling. For each sample, we randomly permute the features and compute the difference in the expectation of the outcome when adding feature i to the features before it in the permutation.

Algorithm 11.4 estimates the Shapley values for the disturbances in a trajectory to determine their contribution to the robustness of the trajectory. It takes in a current trajectory τ with disturbance trajectory \mathbf{x} and a number of samples per time step m . For each time step in the trajectory, the algorithm randomly samples another disturbance trajectory \mathbf{w} by performing a rollout using the nominal trajectory distribution.¹³ It then samples a random permutation \mathcal{P} of the time steps and performs a rollout in which the disturbances are taken from \mathbf{x} for the time steps in $\mathcal{P}_{1:j}$ and from \mathbf{w} for all other time steps. It similarly performs a rollout in which the disturbances are taken from \mathbf{x} for the time steps in $\mathcal{P}_{1:j-1}$ and from \mathbf{w} for all other time steps. The algorithm then computes the difference in the robustness of the two rollouts and averages the differences over m sampled permutations to estimate the Shapley value of each disturbance.

Figure 11.9 shows the Shapley values for the disturbances of the inverted pendulum trajectory used in example 11.3 and figure 11.5. The Shapley values differ from the sensitivity estimates because they account for interactions between disturbances. If we remove groups of disturbances with high Shapley values, it produces a large change in the outcome.

11.4 Policy Explanation through Surrogate Models

For agents with complex policies, it may be difficult to understand the reasoning behind their decisions. In such cases, we can build *surrogate models* to approximate the policy with a model that is easier to interpret. A good surrogate model should have the following characteristics:

¹² E. Štrumbelj and I. Kononenko, "Explaining Prediction Models and Individual Predictions with Feature Contributions," *Knowledge and Information Systems*, vol. 41, pp. 647–665, 2014.

¹³ This step requires that the disturbances sampled at each time step are independent of one another. This assumption may break if the disturbances depend on the states, actions, or observations.

```

struct Shapley
    τ # current trajectory
    m # number of samples per time step
end

function shapley_rollout(sys, s, x, w, inds)
    τ = []
    for t in 1:length(x)
        x = t ∈ inds ? x[t] : w[t]
        o, a, s' = step(sys, s, x)
        push!(τ, (; s, o, a, x))
        s = s'
    end
    return τ
end

function describe(alg::Shapley, sys, ψ)
    τ, m = alg.τ, alg.m
    p = NominalTrajectoryDistribution(sys, length(alg.τ))
    x = [step.x for step in τ]
    φs = zeros(length(τ))
    for t in eachindex(τ)
        for _ in 1:m
            w = [step.x for step in rollout(sys, p)]
            ℙ = randperm(length(τ))
            j = findfirst(ℙ .== t)
            τ_+ = shapley_rollout(sys, τ[1].s, x, w, ℙ[1:j])
            τ_- = shapley_rollout(sys, τ[1].s, x, w, ℙ[1:j-1])
            φs[t] += robustness([step.s for step in τ_+], ψ.formula) -
                      robustness([step.s for step in τ_-], ψ.formula)
        end
        φs[t] /= m
    end
    return φs
end

```

Algorithm 11.4. Estimating the Shapley values of the disturbances in a trajectory. The algorithm takes as input a trajectory τ and a number of samples per time step m . For each time step in the trajectory, the algorithm samples a random vector of disturbances by performing a rollout using the nominal trajectory distribution. It then samples a random permutation of the time steps and locates the current time step in the permutation. Using the `shapley_rollout` function, the algorithm computes the difference in the robustness of the trajectory when adding the disturbance at the current time step to the subset of disturbances in the permutation. It then averages the differences over m sampled permutations to estimate the Shapley value of each disturbance.

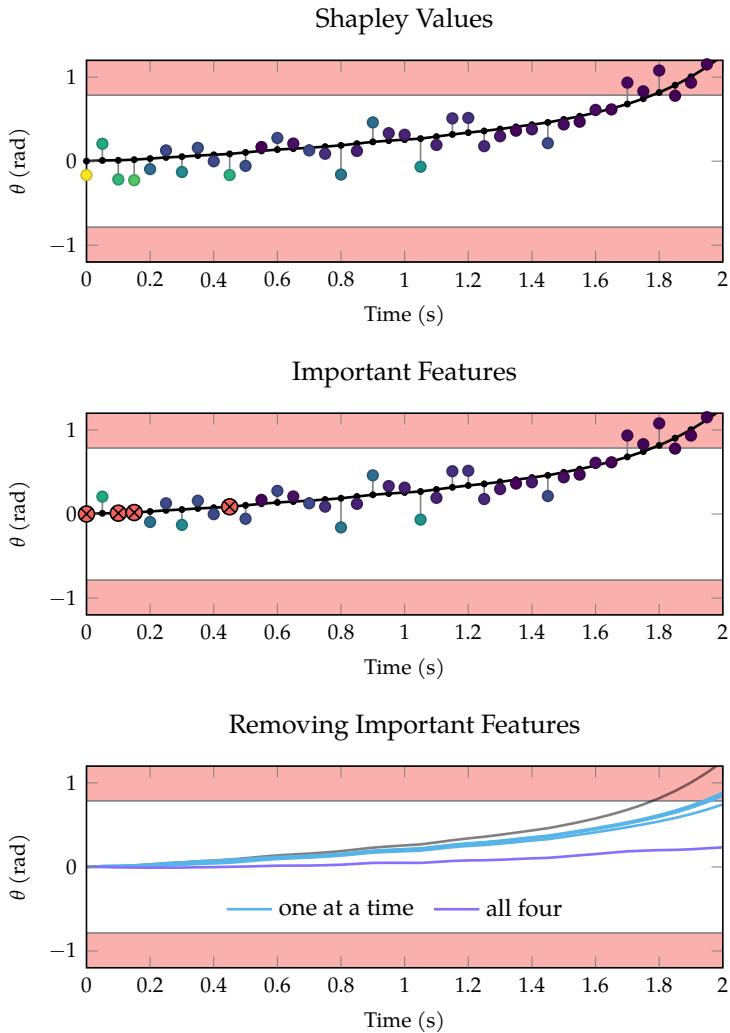


Figure 11.9. Shapley values for the disturbances in an inverted pendulum failure trajectory. The black line shows the true angle of the pendulum at each time step, and the colored markers indicate the noisy observation of the current angle at each time step. Brighter colors indicate higher Shapley values. The Shapley values are highest for disturbances that cause the agent to think that the pendulum is further from tipping over than it actually is, causing it to apply too small of a torque to move toward upright. The second plot shows the trajectory with the four disturbances with the highest Shapley values marked in red. If we remove these disturbances one at a time (blue trajectories in the third plot), we cause small changes in the outcome. However, if we remove all four disturbances (purple trajectory in the third plot) at once, we cause a large change in the outcome.

- *High Fidelity:* The surrogate model should accurately represent the policy. If the surrogate model does not adequately represent the policy, the explanations it provides may be misleading.
- *High Interpretability:* The surrogate model should be easily interpretable by humans. If the surrogate model is too complex, it may be difficult to understand the reasoning behind the decisions.

In general, there is a tradeoff between fidelity and interpretability. A more complex model may be higher fidelity but less interpretable, while a simpler model may be more interpretable but lower fidelity.

Surrogate models can provide *local explanations* or *global explanations* of a policy. Local explanations provide insight into a single decision, while global explanations provide insight into the full policy. To create a local surrogate model, we create a dataset of observations and corresponding decisions near the observation of interest and fit a model to this dataset.¹⁴ For global surrogate models, we gather data across the entire observation space. When selecting a model class to fit the data, we must consider the tradeoff between fidelity and interpretability. This section discusses this tradeoff for two common model classes used as surrogate models.

11.4.1 Linear Models

One common choice for a surrogate model is a *linear model*. Linear models have the form

$$f(\mathbf{x}) = \sum_{i=1}^n w_i x_i + b \quad (11.4)$$

where x_i is a feature of the observation, w_i is a weight for feature i , and b is the bias term. If the action space is discrete, we may apply the logistic or softmax function to the output of the linear model to obtain probabilities for each action. Linear surrogate models can be used to determine feature importance. The magnitudes of the weights of the linear model indicate the contribution of each feature to the agent's decision. Figure 11.10 demonstrates how to use a linear surrogate model to describe the behavior of a collision avoidance policy in two different regions of the observation space. This technique is particularly useful for high-dimensional observations, where it may be difficult to visualize the policy directly.

¹⁴ It is common to weight these data points with higher weights for observations that are closer to the observation of interest. M. T. Ribeiro, S. Singh, and C. Guestrin, "Why Should I Trust You?" Explaining the Predictions of Any Classifier," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

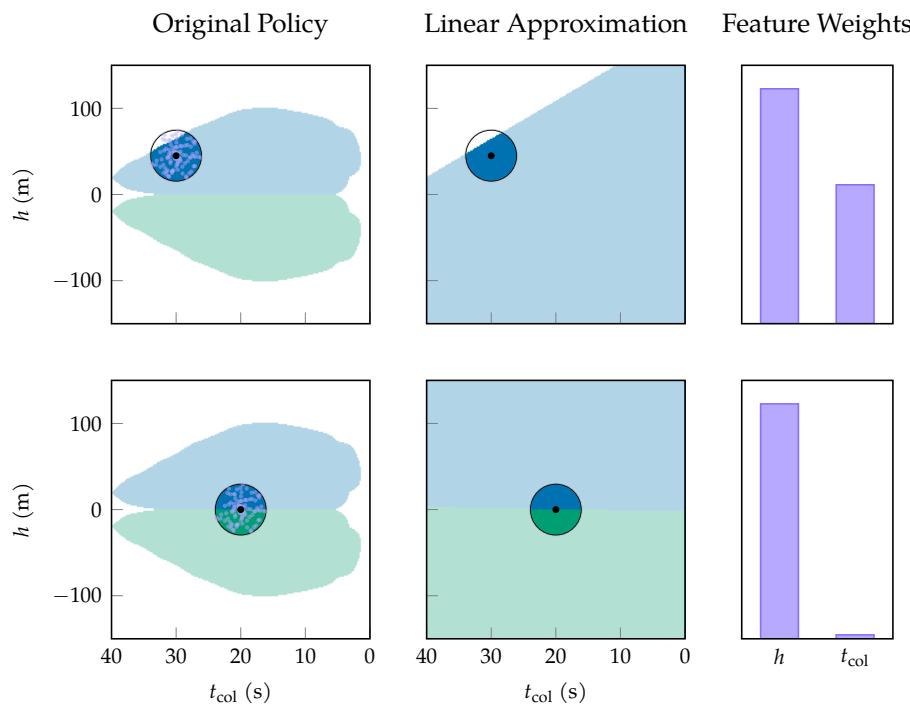


Figure 11.10. Linear surrogate model fit to samples in two different local regions (highlighted circles) of the observation space for a collision avoidance policy. The left column shows the original policy where blue corresponds to the climb action, green corresponds to the descend action, and white corresponds to no advisory. The column in the middle shows the linear surrogate model fit to the samples in the highlighted circle indicated by the purple dots. The right column shows the feature weights of the linear surrogate model for each state variable. The linear surrogate model is accurate in the local region where it was fit, but may not be accurate in other regions of the observation space. Based on the feature weights, the linear surrogate model indicates that the agent’s decision depends on both the relative altitude and time to collision when the relative altitude is around 50 m. In contrast, when the relative altitude is around 0 m, the agent’s decision primarily depends on the relative altitude.

For complex policies, a model that is simply a linear function of observation variables may not provide sufficient fidelity. We may need to add more complex features of the observation to the model. For example, we could add polynomial features of the observation to the linear model to capture non-linear relationships between the features. Alternatively, we can train a neural network to learn a set of nonlinear features that can be linearly combined, but these features are generally not interpretable. Figure 11.11 shows the tradeoff between fidelity and interpretability for a linear model with polynomial features. A common technique to simplify linear models with many features is to encourage sparsity in the weights using a technique called LASSO regression.¹⁵ The features with nonzero weights in a sparse linear model tend to be more important to the output.

¹⁵ R. Tibshirani, “Regression Shrinkage and Selection via the Lasso,” *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 58, no. 1, pp. 267–288, 1996.

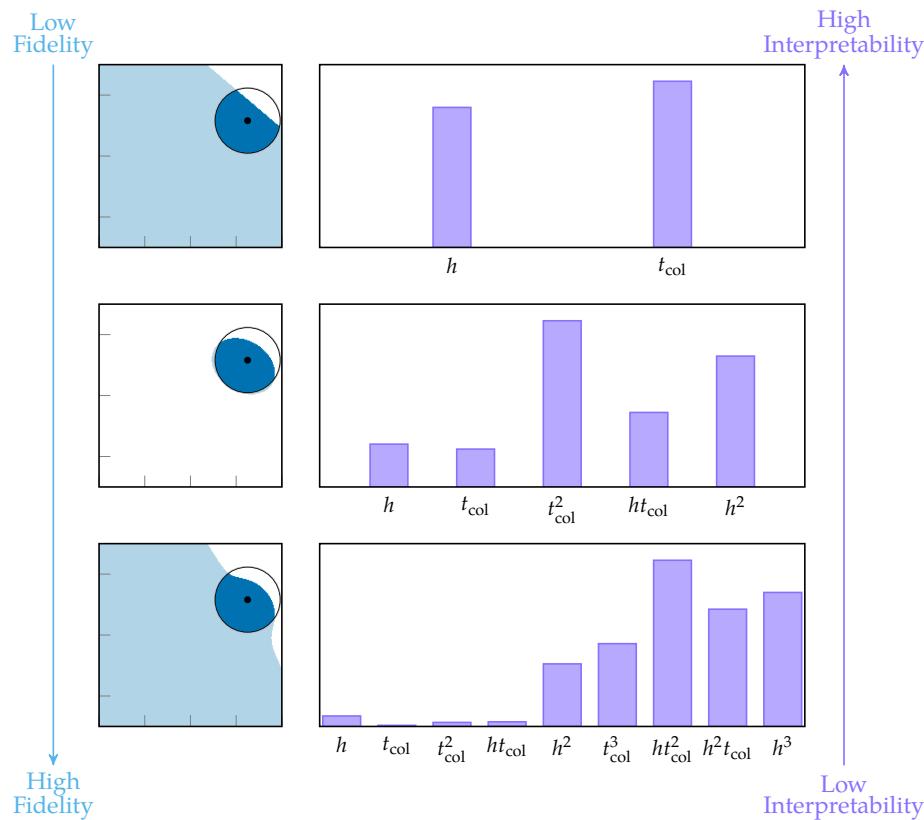
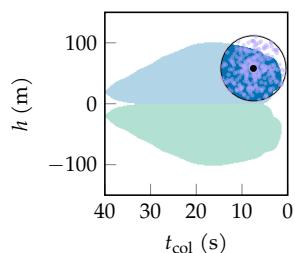


Figure 11.11. Tradeoff between interpretability and fidelity in a linear surrogate model. Each row corresponds to a linear model with first, second, and third order features, respectively. The left column shows the decision boundary of the surrogate model, with the black point indicating the state for which the model is evaluated. The right column shows the feature weights of the surrogate model. The plot below shows the original policy and the sampled points. As the order of the polynomial features increases, the model becomes more accurate in the local region where it was fit at the cost of lower interpretability.



11.4.2 Decision Trees

Decision trees model policies as a series of simple decisions.¹⁶ Each node in the tree represents a decision based on a feature of the observation, and the leaf nodes represent the action taken by the agent. Decisions are typically represented as binary splits, where we follow the left branch in the tree if the feature value is less than a threshold and the right branch if the feature value is greater than or equal to the threshold. Example 11.5 shows a simple decision tree for a slice of the collision avoidance policy.

The maximum depth of the decision tree controls the tradeoff between fidelity and interpretability. Shallow decision trees tend to be more interpretable because they do not require many decisions to make a prediction. However, shallow trees are less expressive and may miss important features of the policy. Figure 11.12 shows the tradeoff between fidelity and interpretability for decision trees.

11.5 Counterfactual Explanations

A *counterfactual* is a hypothetical scenario that describes how an outcome would change if events had unfolded differently. *Counterfactual explanations* explain the behavior of a model by identifying the smallest change to the input that would result in a different outcome. We can frame the problem of generating a counterfactual explanation as a multiobjective optimization problem, in which our goal is to maximize the following four objectives:¹⁷

1. *Change in outcome:* The counterfactual input should result in an outcome different from that obtained with the original input. If our goal is to change a trajectory τ from a failure to a success, we can use the temporal logic robustness metric as the objective:

$$f_{\text{outcome}}(\tau') = \text{robustness}(\tau', \psi) \quad (11.5)$$

where τ' is the counterfactual trajectory. By maximizing robustness, we move toward a trajectory that satisfies the safety property ψ .

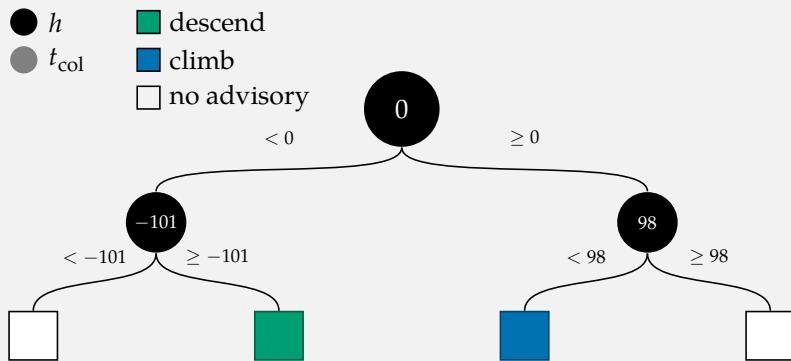
2. *Distance to original input:* The counterfactual input should be close to the original input τ to ensure that the change is minimal, resulting in the following objective:

$$f_{\text{close}}(\tau') = -\|\tau' - \tau\|_p \quad (11.6)$$

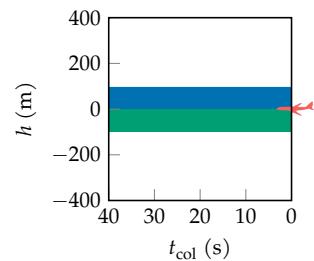
¹⁶ The `DecisionTree.jl` package can be used to train a decision tree model from data.

¹⁷ S. Dandl, C. Molnar, M. Binder, and B. Bischl, “Multi-Objective Counterfactual Explanations,” in *International Conference on Parallel Problem Solving from Nature*, 2020.

Suppose we want to train a decision tree to approximate the slice of the collision avoidance policy shown in example 11.1. The following decision tree was trained on a dataset of 100,000 randomly sampled states from the policy slice. The decision tree has a maximum depth of 2 and uses the state variables to make decisions. Nodes that split using h are shown in black, nodes that split using t_{col} are shown in gray, and the color of the square leaf nodes are the actions taken by the agent.



Example 11.5. Simple decision tree for the collision avoidance policy. The policy represented by the decision tree is shown below.



With a maximum depth of 2, the decision tree only makes decisions based on h . If h is positive, the tree selects whether to climb or issue no advisory based on the magnitude of the relative altitude. Similarly, if h is negative, the tree selects whether to descend or issue no advisory based on the magnitude of the relative altitude. The policy represented by the decision tree is shown in the caption. This decision tree provides a simple, interpretable model of the agent's policy. However, the fidelity of the decision tree is limited by its depth, and it misses some key features of the policy that depend on the time to collision.

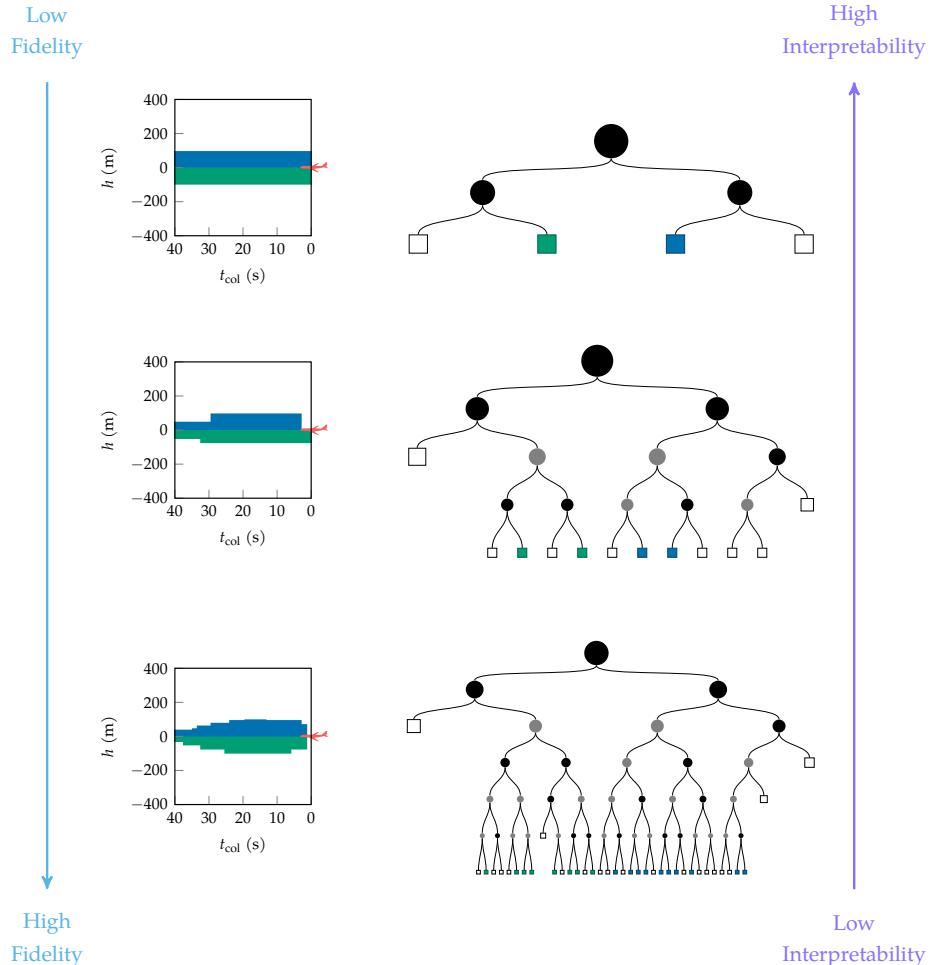


Figure 11.12. Tradeoff between fidelity and interpretability when training a decision tree surrogate model on the slice of the policy shown in example 11.1. Each row corresponds to a decision tree with a maximum depth of 2, 4, and 6, respectively. The left column shows the decision boundary of the surrogate model. The colors correspond to the colorscheme shown in example 11.5. As the maximum depth of the decision tree increases, the model becomes more accurate at the cost of lower interpretability.

where τ' is the counterfactual input and $\|\cdot\|_p$ is the p norm.

3. *Sparsity of the change:* The difference between the original input and the counterfactual input should be sparse. In other words, the counterfactual input should differ in only a few features. We can use the following objective

$$f_{\text{sparsity}}(\tau') = -\|\tau' - \tau\|_0 \quad (11.7)$$

where $\|\cdot\|_0$ returns the number of nonzero elements.¹⁸ This objective presents challenges for gradient-based optimization algorithms because its derivative is zero almost everywhere. To use gradient-based optimization, we can use the L_1 norm, which encourages sparsity in the final solution.

4. *Plausibility:* The new input should be a plausible input. We can check plausibility using the likelihood of the counterfactual trajectory as follows:

$$f_{\text{plaus}}(\tau') = p(\tau') \quad (11.8)$$

The four counterfactual objectives are often at odds with one another. For example, only making small changes to the input is unlikely to change the outcome. We can use multiobjective optimization techniques to find counterfactual inputs that balance these objectives.¹⁹ Algorithm 11.5 creates a single objective function using a weighted sum of the objectives. We can apply a variety of optimization algorithms to find the counterfactual input that maximizes the objective function (see section 4.6).²⁰ To ensure compatibility with gradient-based optimization techniques, we omit the objective in equation (11.7) and instead set $p = 1$ in equation (11.6) to encourage sparsity. Figure 11.13 shows the generation of a counterfactual explanation for a failure of the inverted pendulum system.

```

function counterfactual_objective(x, sys, ψ, x₀; ws=ones(3))
    s, x = extract(sys.env, x)
    τ = rollout(sys, s, x)
    foutcome = robustness([step.s for step in τ], ψ.formula)
    fclose = -norm(x - x₀, 1)
    fplaus = logpdf(NominalTrajectoryDistribution(sys, length(x)), τ)
    return ws' * [foutcome, fclose, fplaus]
end

```

We are often interested in producing counterfactual explanations for inputs that we can control. For example, a counterfactual explanation for a loan approval

¹⁸ This operation is sometimes referred to as the L_0 norm; however, it is not a proper norm because it does not scale properly when multiplied by a scalar.

¹⁹ An overview of multiobjective optimization is provided in chapter 12 of M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019.

²⁰ It is also common to use genetic algorithms that encourage diversity in the solutions to find a diverse set of counterfactual explanations. S. Dandl, C. Molnar, M. Binder, and B. Bischl, "Multi-Objective Counterfactual Explanations," in *International Conference on Parallel Problem Solving from Nature*, 2020.

Algorithm 11.5. Counterfactual objective function that combines equations (11.5), (11.6) and (11.8). The objective function takes in the counterfactual input x , the system sys , the specification ψ , the original input x_0 , and a vector of weights ws . The algorithm computes each individual objective and returns their weighted sum. We take the logarithm of f_{plaus} for numerical stability.

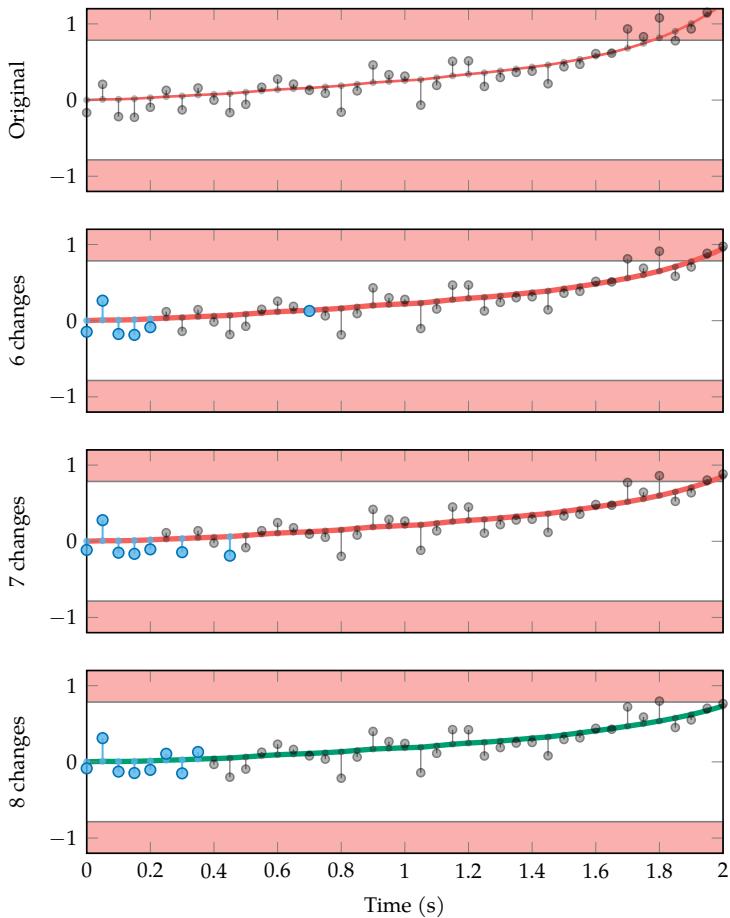
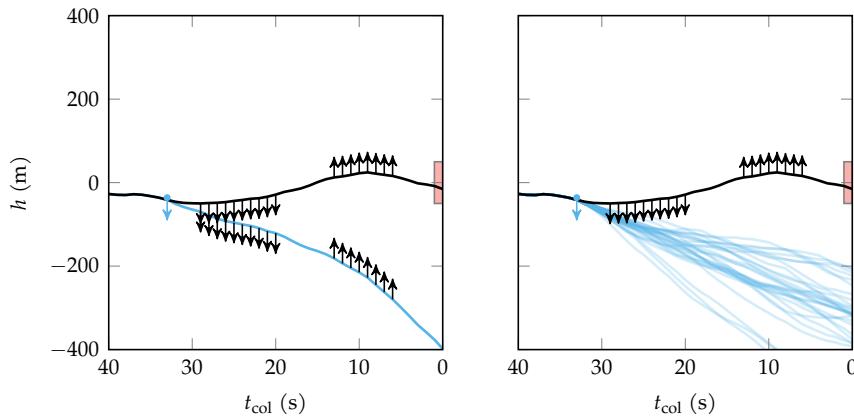


Figure 11.13. Generation of a counterfactual trajectory for the inverted pendulum system by changing the disturbances on the measurement of θ . The original trajectory is shown in the plot on the top with the disturbance at each time step shown in black. The remaining plots show the counterfactual trajectories with different numbers of disturbances changed. The disturbances that differ from the original trajectory are shown in blue. We can create these trajectories by decreasing the relative weighting of the closeness objective in the counterfactual objective until we generate a trajectory that is no longer a failure. As noted in example 11.4, the disturbances at the beginning of the trajectory have the most significant impact on the outcome because the controller is saturated at the end of the trajectory.



system that involves a change in the income of the applicant is more useful than an explanation that requires a change in their age. While we have control over the actions of an agent, we often do not have control over the disturbances that affect the system.

We can generate several different types of counterfactual explanations over the actions of the agent. The simplest type of counterfactual explanation involves changing an action at a particular time step or set of time steps while keeping all other actions constant. This technique is most similar to algorithm 11.5. A key assumption of this method is that the components of the input are independent of one another. However, changing the action at one time affects the state at the next time step, which in turn affects the action at the next time step. This cascading effect breaks the independence assumption and can lead to counterfactual explanations that are not plausible.

One way to account for the cascading effect is to select actions that maximize the expected change in the outcome given all possible future actions and disturbances. If we are searching for counterfactuals that only change the action at one time step, we can produce a set of counterfactual explanations by performing rollouts of the policy for the remaining time steps. We can then select the action that maximizes the expected change in the outcome. Figure 11.14 shows this technique for the aircraft collision avoidance example. Understanding the effects of changing multiple actions requires more sophisticated techniques.²¹

Figure 11.14. Counterfactual explanations (blue) for a failure of the collision avoidance system (black). The arrows represent the direction of the commanded collision avoidance advisory at each time step. No arrow indicated no advisory. The black arrows represent the original trajectory, while the blue arrow represents the action change used to generate the counterfactual trajectories. The plot on the left shows the counterfactual explanation when holding all other actions and disturbances constant. The plot on the right shows the counterfactual explanations when rolling out the trajectory for all other time steps. In this scenario, we can conclude that issuing a descend advisory a few time steps earlier would have prevented the failure.

²¹ S. Dandl, C. Molnar, M. Binder, and B. Bischl, "Multi-Objective Counterfactual Explanations," in *International Conference on Parallel Problem Solving from Nature*, 2020.

11.6 Failure Mode Characterization

Another way to explain the behavior of a system is to characterize its failure modes. We can use *clustering* algorithms to create groupings of failure trajectories that are similar to one another. Identifying the similarities and differences between failures helps us understand their underlying causes. One common clustering algorithm is *k-means*²² (algorithm 11.6), which groups data points into k clusters based on their similarity to one another.²³

To apply *k-means*, we must first extract a set of real-valued features from each failure trajectory to use for clustering. Let \mathbf{x} represent the set of features from trajectory τ and ϕ be a feature extraction function such that $\mathbf{x} = \phi(\tau)$. To represent the clusters \mathcal{C} , *k-means* keeps track of k cluster centroids $\mu_{1:k}$ in feature space and assigns each trajectory to the cluster with the closest centroid to its features. We begin by initializing the centroids to the features of k random trajectories. At each iteration, *k-means* performs the following steps:

1. Assign each trajectory to the cluster with the closest centroid to its feature vector. In other words, τ_i is assigned to cluster \mathcal{C}_j when

$$j = \arg \min_{j \in 1:k} d(\mathbf{x}_i, \mu_j) \quad (11.9)$$

where $d(\cdot, \cdot)$ is a distance metric such as the L_2 norm.

2. Update the centroids to the mean of the feature vectors of the trajectories in each cluster such that

$$\mu_j = \frac{1}{|\mathcal{C}_j|} \sum_{\tau \in \mathcal{C}_j} \phi(\tau) \quad (11.10)$$

where $|\mathcal{C}_j|$ is the number of trajectories in cluster \mathcal{C}_j .

The algorithm repeats until the centroids converge or a maximum number of iterations is reached. The *k-means* algorithm may converge to a local minimum depending on the initialization of the centroids, so it is common to run the algorithm multiple times with different initializations. Figure 11.15 shows the progression of the *k-means* algorithm on failure trajectories of the inverted pendulum system using the average angle and angular velocity of each trajectory as the features.

²² This algorithm is also referred to as Lloyd's algorithm, named after Stuart P. Lloyd (1923–2007). S. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

²³ A detailed overview of clustering algorithms is provided in D. Xu and Y. Tian, "A Comprehensive Survey of Clustering Algorithms," *Annals of Data Science*, vol. 2, pp. 165–193, 2015.

```

struct Kmeans
    ts      # trajectories to cluster
    ϕ       # feature extraction function ( $x = \phi(\tau)$ )
    d       # distance metric function ( $d(x[i], \mu_j)$ )
    k       # number of clusters
    max_iter # maximum number of iterations
end

function describe(alg::Kmeans, sys, ψ)
    x = [alg.ϕ(τ) for τ in alg.ts]
    μ = x[randperm(length(x))[1:alg.k]]
    C = [Int[] for j in 1:alg.k]
    for _ in 1:alg.max_iter
        C = [Int[] for j in 1:alg.k]
        for i in eachindex(x)
            push!(C[argmin([alg.d(x[i], μ_j) for μ_j in μ])], i)
        end
        for j in 1:alg.k
            if !isempty(C[j])
                μ[j] = mean(x[i] for i in C[j])
            end
        end
    end
    return C, μ
end

```

Algorithm 11.6. The k -means algorithm for clustering failure trajectories. The algorithm takes in a set of trajectories `ts`, a feature extraction function `ϕ`, the number of clusters `k`, and the maximum number of iterations `max_iter`. The algorithm first extracts the features from each trajectory and initializes the centroids to random trajectories. At each iteration, it assigns each trajectory to the cluster with the closest centroid based on the L_2 norm and updates the centroids to the mean of the feature vectors of the trajectories in each cluster.

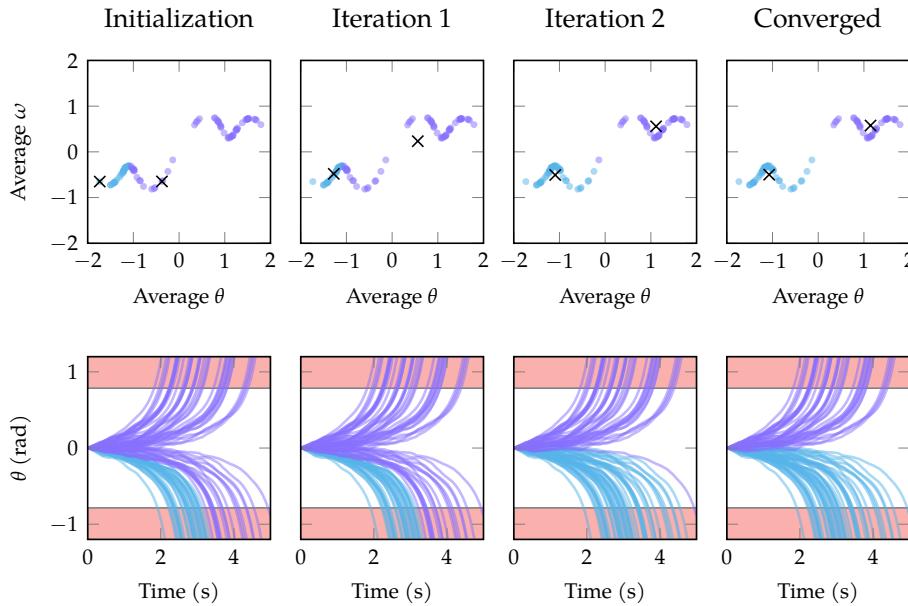


Figure 11.15. Progression of the k -means algorithm on failure trajectories of the inverted pendulum system with $k = 2$ and the average angle and angular velocity of each trajectory as the features. The colors represent the different clusters at each iteration and the black crosses represent the cluster centroids. The algorithm converges to a set of clusters that represent two distinct failure modes. One failure mode corresponds to trajectories in which the pendulum falls to the left, and the other cluster corresponds to trajectories in which the pendulum falls to the right.

The clustering results help us understand the failure modes of the system. One way to interpret the clusters is to create a *prototypical example* for each cluster. The prototypical example for a given cluster is the trajectory that is closest to its centroid in feature space. By examining the prototypical examples, we can understand the characteristics of each failure mode. Figure 11.16 shows the prototypical examples for the final clusters in figure 11.15. At runtime, we can assign new failure trajectories to the cluster with the closest centroid to their features and use the prototypical examples to explain the failure mode of the trajectory.

Algorithm 11.6 requires us to select the number of clusters k , the distance function d , and the feature extraction function ϕ . The clustering results are highly dependent on these choices. However, selecting the number of clusters and the features is often a subjective process that requires domain knowledge. To select the number of clusters, we can try different values for k and select the one that results in the most interpretable clusters or that minimizes a clustering objective such as the sum of the squared distances between each point and its cluster centroid.

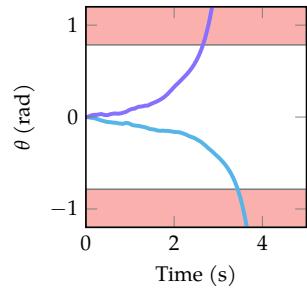


Figure 11.16. Prototypical examples of failure modes for the inverted pendulum system using the clusters in figure 11.15. The prototypes reveal that one failure mode involves the pendulum falling to the left, while the other involves the pendulum falling to the right.

We can also use domain knowledge to select features that are likely to capture the underlying causes of the failures. A simple way to select features is to create a feature vector by concatenating all of the states in the trajectory. We could create similar feature vectors for the actions, observations, and disturbances. However, these feature vectors will be high-dimensional and may not result in interpretable clusters (figure 11.17).

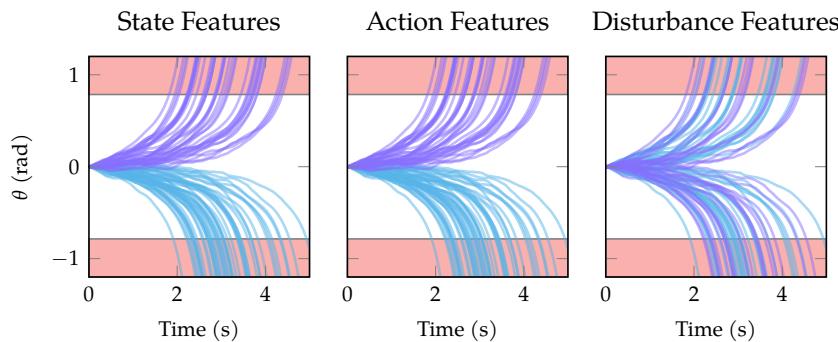


Figure 11.17. Clustering failure trajectories of the inverted pendulum system using features consisting of the states, actions, and disturbances of each trajectory, respectively. The colors represent the different clusters. The clusters based on the state and action features show interpretable failure modes, while the clusters based on the disturbance features are less interpretable.

To improve interpretability of the clusters, we can cluster the trajectories based on temporal logic features. Specifically, we use the parameters of a *parametric signal temporal logic* (*PSTL*) formula as the features for clustering.²⁴ *PSTL* is an extension of signal temporal logic (section 3.5.2), in which the time constants in the temporal operators and signal values in the atomic propositions may be replaced by parameters. *PSTL* expressions represent template formulas that can be instantiated to *STL* formulas with specific values for the parameters.

To perform clustering with *PSTL* features, we first select a template formula. We then set the features of each trajectory to the values of the parameters for which the formula is *marginally satisfied*. An *STL* formula is marginally satisfied by a trajectory if the robustness of the trajectory with respect to the formula is zero. By plugging these parameters into the template formula, we obtain a temporal logic formula that describes the behavior of the trajectory.

We can use optimization methods to find the values of the parameters that marginally satisfy the formula for each trajectory by finding the ϕ that minimizes $\|\rho(\tau, \psi_\phi)\|$ where ρ is the robustness function and ψ_ϕ is the instantiated *STL* formula with parameters ϕ . Example 11.6 applies this idea to the inverted pendulum system. We can then perform clustering on the extracted *PSTL* features to identify

²⁴ M. Vazquez-Chanlatte, J. V. Deshmukh, X. Jin, and S. A. Seshia, "Logical Clustering and Learning for Time-Series Data," in *International Conference on Computer Aided Verification*, 2017.

failure modes. Figure 11.18 shows the clusters of failure trajectories of the inverted pendulum system using the PSTL template in example 11.6.

Clustering using PSTL features requires us to select a template formula. The template formula should capture the key aspects of the system that are relevant to the failure modes. For systems with complex failure modes, it may be difficult to hand-design a template formula that captures all the failure modes. In these cases, we can use more sophisticated techniques that build decision trees using a grammar based on temporal logic.²⁵

11.7 Summary

- Interpretable descriptions of system behavior are essential for understanding and calibrating trust.
- Policy visualization allows us to interpret the policy that a particular agent uses to make decisions.
- Feature importance methods such as saliency maps and Shapley values allow us to understand the impact of different features on the behavior of a system.
- Surrogate models allow us to explain the policy of a complex system using a simpler model and must balance between fidelity and interpretability.
- Counterfactual explanations provide insights into the decision-making process of a system by showing how changes in the input affect the output.
- We can characterize the failure modes of a system by clustering them using interpretable features.

11.8 Exercises

Exercise 11.1. Suppose we want to evaluate the fairness of an automated hiring assistant using sensitivity analysis. What résumé features should we expect the assistant to have high sensitivity toward? What features should we expect the assistant to have low sensitivity toward?

Solution: There are multiple possible answers. The assistant should be sensitive to features such as education level, years of experience, and relevant skills. The assistant should have low sensitivity to features such as gender, race, and age.

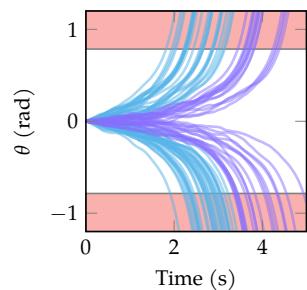


Figure 11.18. Clusters of failure trajectories of the inverted pendulum system using the PSTL template in example 11.6 and $k = 2$. The algorithm results in two clusters. The pendulum falls over earlier in the blue trajectories compared to the purple trajectories.

²⁵R. Lee, M. J. Kochenderfer, O. J. Mengshoel, and J. Silbermann, “Interpretable Categorization of Heterogeneous Time Series Data,” in *SIAM International Conference on Data Mining*, 2018.

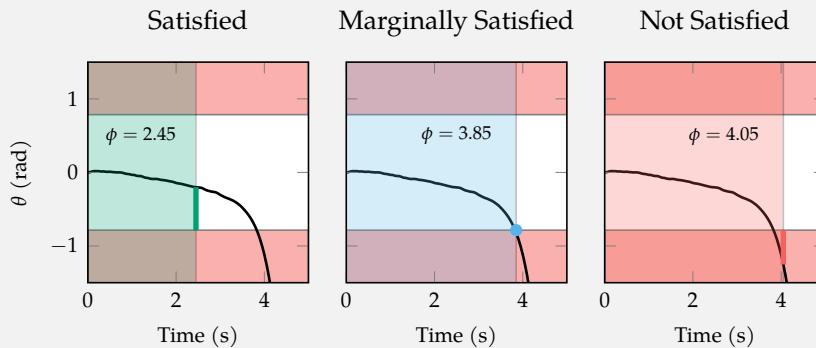
The following STL formula specifies that the angle of the pendulum should not exceed $\pi/4$ for the first 200 time steps:

$$\psi = \square_{[0,200]} \left(\theta < \frac{\pi}{4} \right)$$

If we replace the time bound with a parameter ϕ , we obtain the following PCTL template formula:

$$\psi_\phi = \square_{[0,\phi]} \left(\theta < \frac{\pi}{4} \right)$$

The plots below show the robustness of the formula for different values of ϕ . The plot on the left shows a value for ϕ such that the trajectory satisfies the formula, the plot in the middle shows a value for ϕ that marginally satisfies the formula, and the plot on the right shows a value for ϕ such that the trajectory does not satisfy the formula.



We can find the value of ϕ that marginally satisfies ψ_ϕ by searching for the value that causes the robustness to be as close as possible to zero. For this simple formula, we can solve the optimization problem using a grid search over the values of ϕ . The value of ϕ that marginally satisfies the formula will be the time just before the magnitude of the angle of the pendulum exceeds $\pi/4$.

Example 11.6. Example of a PCTL template formula for the inverted pendulum system. The plots show the robustness of the formula for different values of ϕ . Our goal is to find the value of ϕ that causes a given trajectory to marginally satisfy the formula.

Exercise 11.2. Consider the function $f(x) = \sigma(10x)$ where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function. Compute the sensitivity of f with respect to the input feature x when $x = 1$ by taking the derivative of f with respect to x .

Solution: We can compute the derivative using the following code:

```
julia> using ForwardDiff: derivative
julia> derivative(x→1/(1+exp(-10x)), 1)
0.00045395807735951676
```

Exercise 11.3. Consider the function $f(x) = \sigma(10x)$ where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function. Compute the sensitivity of f with respect to the input feature x when $x = 1$ using the integrated gradients technique. Use a baseline of $x = 0$ and $m = 10$ steps of numerical integration. How does this result compare to the result in the previous question?

Solution: We can compute the integrated gradients sensitivity using the following code:

```
julia> using ForwardDiff: derivative
julia> sum(derivative(x→1/(1+exp(-10x)), x) for x in range(0,1,length=10))/10
0.5749783367160306
```

This value is higher than the sensitivity computed in the previous question. In the previous question, we only considered the gradient of the current value of x , which is in a saturated region of the sigmoid function. In contrast, the integrated gradients technique considers the gradient of the function over the entire range of x values between the baseline and the current value.

Exercise 11.4. Suppose we want to use the inner loop in algorithm 11.4 that iterates over samples of the term in the summation in equation (11.2) to estimate the Shapley value $\phi_i(\mathbf{x})$. This value represents the Shapley value for input feature i of the function $f(\mathbf{x})$ when $\mathbf{x} = [0.1, 0.2, -0.1, 0.8, -0.4]$. During the first iteration, we compute

$$f([0.1, 0.2, -0.1, 0.2, 0.5]) - f([0.1, 0.3, -0.1, 0.2, 0.5])$$

What is the value of i ? What subset of features does this sample consider?

Solution: For each term in the summation, we consider the difference in the function output when we include the current value of feature of interest to the subset of fixed features compared to when we do not include it. Since the second element of the input is different between the two input vectors, the value of i is 2. The first and third features are fixed to their current values in this sample. Therefore, the subset of features considered in this sample is $\{1, 3\}$.

Exercise 11.5. What is an advantage of using a linear surrogate model to determine feature importance compared to gradient-based sensitivity analysis techniques?

Solution: There are multiple possible answers. One advantage is that the surrogate modeling approach can be applied to any type of model and does not require computing the gradient of the model.

Exercise 11.6. Suppose we want to produce a counterfactual explanation for a model that predicts the likelihood of a patient having a particular disease. We want the counterfactual explanation to be actionable for the patient to prevent developing the disease in the future. What features should we consider in the counterfactual explanation? What features should we avoid?

Solution: We should consider features that the patient can change, such as diet, exercise, and lifestyle. We should avoid features that the patient cannot change, such as their age or genetic background.

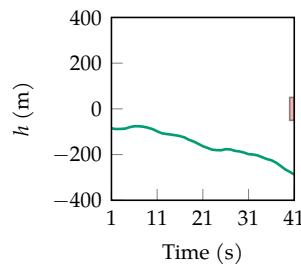
Exercise 11.7. What is a drawback of characterizing the failure modes of a system by clustering the disturbance trajectories?

Solution: For systems with long time horizons, the disturbance trajectories will be high-dimensional and may not result in interpretable clusters. To improve this approach, we could use dimensionality reduction techniques to reduce the dimensionality of the disturbance trajectories before clustering.

Exercise 11.8. Consider the following PCTL template formula for the aircraft collision avoidance example:

$$\psi_\phi = \square_{[40,41]} |h| \geq \phi$$

For what value of ϕ does the trajectory shown below marginally satisfy the formula?



Solution: The maximum value of $|h|$ in the trajectory is around 300 m. Therefore, the trajectory marginally satisfies the formula when $\phi = 300$ m.

12 Runtime Monitoring

While the validation algorithms in the previous chapters are typically applied *offline* prior to deployment, *runtime monitoring* techniques perform *online* assessments that check the safety of the system during operation. The goal of a runtime monitor is to flag situations that may be hazardous when they occur so that we can trigger fallback mechanisms such as alerting a human operator, switching to a safe mode or fallback policy, or updating the system's model of the world. Offline validation algorithms rely on a set of modeling assumptions about the environment and disturbances that the system will encounter during operation. If these models are incorrect or the environment changes, the validation results may no longer be valid. This chapter begins by discussing techniques to identify when we are operating outside the assumptions made during offline validation. We then discuss techniques to monitor uncertainty in the behavior of the system. Finally, we present techniques to monitor for potential failures in the system.

12.1 Operational Design Domain Monitoring

The *operational design domain (ODD)* of a system is the set of conditions under which it is designed to operate safely. For example, the operational design domain for an image-based aircraft taxi system may consist of the set of weather conditions, times of day, and taxiways for which the system was designed. A good system model should cover the ODD so that the validation results from the previous chapters are valid within the ODD. If the system is operating outside the ODD, the validation results may no longer be valid, and we cannot provide any guarantees on the system's safety. Therefore, it is important for us to monitor at runtime whether a system is operating within its ODD.

There are multiple ways to represent the ODD of a system. One option is to specify the ODD as a set of hand-designed conditions. For example, we could write down the exact weather conditions, times of day, and taxiways that the aircraft taxi system is designed to operate under (figure 12.1). We can also specify the ODD in terms of acceptable ranges for the model parameters. For example, we might expect the variance of our sensor measurements to stay within a particular bound. A drawback of this approach is that can require specialized domain knowledge to properly specify these conditions.

We could also represent the ODD using data-driven approaches. These approaches rely on a data set of trajectory features that can be monitored at runtime and adequately capture the characteristics of the ODD. These features are problem dependent. For example, if the characteristics of the ODD are well-described by the state, the data could be the set of states observed during offline validation or training (figure 12.2).¹ Some problems may require additional features to adequately represent the ODD. For example, the aircraft taxi system may perform differently depending on the image observation it receives. In this case, the data could be the set of images observed during offline validation or training. We could also use trajectory segments or full trajectories as the data.

Data-driven approaches to ODD monitoring use the data to define a set representation of the ODD. When the system encounters new data points at runtime, it can check whether they belong to the set and flag potential dangerous behavior if they do not. The remainder of this section discusses techniques to define this set given a representative data set.

12.1.1 Nearest Neighbors Representation

One way to define the ODD given a data set is to use a nearest neighbors representation. Specifically, we define the ODD as the set of points whose nearest neighbor in the data set is within a certain threshold distance γ according to a distance metric. A common distance metric is the Euclidean distance. The threshold γ controls the conservatism of the set representation. A smaller γ results in a more conservative representation in the sense of being less likely to include situations that should not be included in the ODD. However, a value for γ that is too small may be too conservative such that it misses out on situations that should be included in the ODD. Figure 12.3 shows the ODD for the data in figure 12.3 defined using different threshold values.



Figure 12.1. An example of a hand-designed operational design domain for an aircraft taxi system.

¹ Many modern systems are trained using a data set that is representative of the distribution over features we expect to observe when operating in the ODD. Therefore, we often want to detect when we are operating outside this distribution. This process is often referred to as *out-of-distribution detection*. J. Yang, K. Zhou, Y. Li, and Z. Liu, "Generalized Out-Of-Distribution Detection: A Survey," *International Journal of Computer Vision*, vol. 132, no. 12, pp. 5635–5662, 2024.

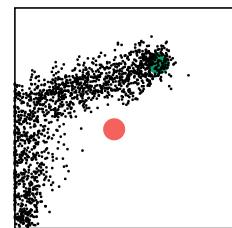


Figure 12.2. For the continuum world problem, we can use the states visited during rollouts used for offline validation to derive a representation of the operational design domain.

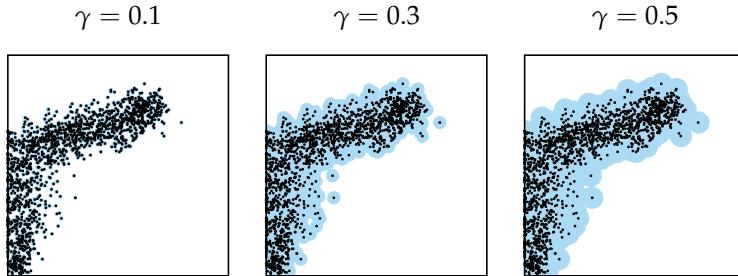


Figure 12.3. ODD (blue) defined using nearest neighbors with different threshold values for the data in figure 12.2. As the threshold value increases, the ODD covers more space and becomes less conservative.

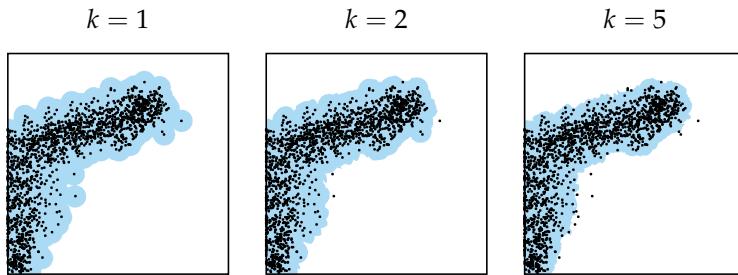


Figure 12.4. ODD (blue) defined using k nearest neighbors with different values of k for the data in figure 12.2. The distance threshold is held constant at $\gamma = 0.5$. As k increases, the ODD covers less space and becomes more conservative.

In addition to the threshold γ , we can also control the number of neighbors used to define the ODD. Instead of only considering the nearest neighbor, we can define the ODD as the set of points whose k -nearest neighbors in the data set are within a certain threshold distance γ . Increasing k ensures that we do not include points in the ODD that are near outliers in the data set. Figure 12.4 shows the ODD defined using different values of k for the data in figure 12.4.

Algorithm 12.1 implements a runtime monitor that uses the nearest neighbors representation to check whether a new input is within the ODD. It computes the k -nearest neighbors to the input and checks if they are within the threshold γ . If all of the neighbors are within the threshold, the monitor returns true, indicating that the input is within the ODD.

A drawback of the nearest neighbors representation is that it requires storing the entire data set in memory at runtime, which may be infeasible for systems with memory limitations or large data sets. It may also require a significant amount of time to compute the nearest neighbors for each input, though a *spatial index* can help improve computational efficiency.² One way to improve the memory

² H. Samet, *Foundations of Multi-dimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

```

struct KNNMonitor
    data # ODD data matrix (each column is a datapoint)
    k    # number of neighbors
    y    # threshold
end

function monitor(alg::KNNMonitor, input)
    kdtree = KDTTree(alg.data)
    neighbors, distances = knn(kdtree, input, alg.k)
    return all(distances .< alg.y)
end

```

Algorithm 12.1. ODD monitoring using a set defined by k -nearest neighbors. The algorithm creates a k -d tree from the ODD data using the `NearestNeighbors.jl` package. A k -d tree is a data structure that allows for efficient nearest neighbor queries. The monitor then uses this data structure to find the k -nearest neighbors to the input and checks if they are within a threshold distance γ .

efficiency of the nearest neighbors representation is to cluster the data into groups of similar points and compute the nearest neighbors to the center of each cluster (figure 12.5). A common clustering algorithm called k -means is described in section 11.6. We can increase the threshold γ to account for the distance between the input and the center of the cluster.

12.1.2 Polytope Representation

To avoid storing the entire data set in memory, we can use a more compact set representation such as a polytope (see section 8.3.1). One way to define the ODD using a polytope is with the convex hull of the data. We can then check if a new input is within the convex hull to determine if it is within the ODD. However, polytopes are not as expressive as the nearest neighbors representation and may produce representations that are insufficiently conservative when the ODD is nonconvex (figure 12.6).

We can represent the ODD using a more expressive set by defining it as the union of multiple polytopes. For example, we could cluster the data set into k clusters using a clustering algorithm and take the union of the convex hulls of the clusters. Algorithm 12.2 implements this monitoring technique given a data set and a clustering of the data. Figure 12.7 shows the ODD for the data in figure 12.2 using different numbers of clusters. This approach, however, may still produce an ODD that contains regions of low data density if outliers are present.

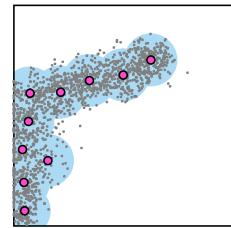


Figure 12.5. Improving the efficiency of the nearest neighbors representation (blue) by clustering the data in figure 12.2 into 10 clusters and computing the nearest neighbors to the center of each cluster.

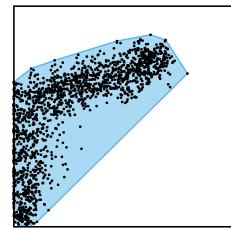


Figure 12.6. The ODD (blue) defined using the convex hull of the data in figure 12.2. This ODD representation is underconservative because it includes a large area for which there is very little data.

```

struct HullMonitor
    data # ODD data matrix (each column is a datapoint)
    C    # collection of vectors containing cluster column indices
end

function monitor(alg::HullMonitor, input)
    for (k, v) in alg.C
        hull = convex_hull([alg.data[:, i] for i in v])
        if input ∈ VPolytope(hull)
            return true
        end
    end
    return false
end

```

Algorithm 12.2. ODD monitoring using a set defined by the convex hull of clustered data. For each cluster, the algorithm computes the convex hull and checks if the input is within it. If the input is within any of the convex hulls, the monitor returns true, indicating that the input is within the ODD.

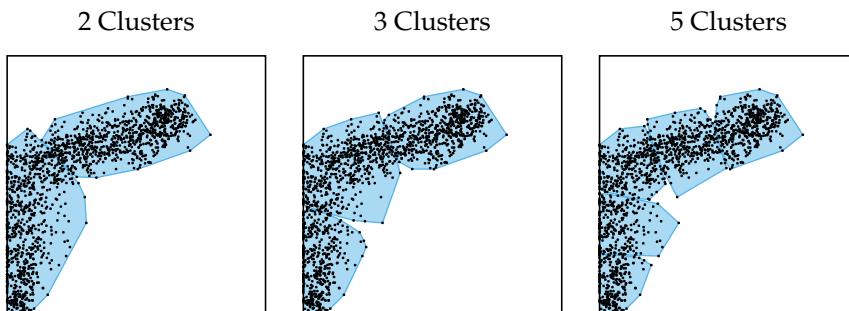


Figure 12.7. The ODD defined using the convex hulls of multiple clusters for the data in figure 12.2. These ODDs are more expressive than the ODD that uses the convex hull of the entire data set. As we increase the number of clusters, the ODD more closely approximates the data at the expense of increased memory usage.

12.1.3 Superlevel Set Representation

Another way to define the ODD is to use a *superlevel set*. A superlevel set of a function is the set of points for which the function is greater than a threshold. For example, we could fit a distribution to the data set and define the ODD as a superlevel set of its probability density function. Algorithm 12.3 implements this monitoring technique given a distribution and a likelihood threshold.

```
struct SuperlevelSetMonitor
    dist # distribution
    γ    # likelihood threshold
end

function monitor(alg::SuperlevelSetMonitor, input)
    return pdf(alg.dist, input) > alg.γ
end
```

When fitting a distribution to the data, it is important that we select a model class that is expressive enough to capture the characteristics of the ODD (see chapter 2). For example, figure 12.8 demonstrates a scenario in which fitting a mixture of Gaussians to the data results in a better representation of the ODD than fitting a single Gaussian. Because the distributions of many model classes can be fully specified using a small number of parameters, this approach is more memory efficient than the nearest neighbors representation. The method is also robust to outliers because the likelihood will be high where the data is dense.

Instead of using the superlevel set of a distribution fit to the data, we can also use the superlevel set of a function that outputs the likelihood of the input being in the ODD. For example, we can train a classifier to predict the likelihood of a point being in the ODD and define the ODD as the set of points for which the model outputs a likelihood greater than a threshold. Figure 12.9 shows an example of this approach. One drawback of this approach is that training the model typically requires data from inside and outside the ODD, which may be difficult to obtain.

Algorithm 12.3. ODD monitoring using a set defined by the superlevel set of a distribution. The algorithm checks whether the probability density function of the distribution evaluated at the input is greater than a threshold γ . If the probability density function is greater than the threshold, the monitor returns true, indicating that the input is within the ODD.

12.1.4 High-Dimensional Data

The methods presented in this section may struggle with high-dimensional data such as image data due to the *curse of dimensionality*. As the dimension of the data

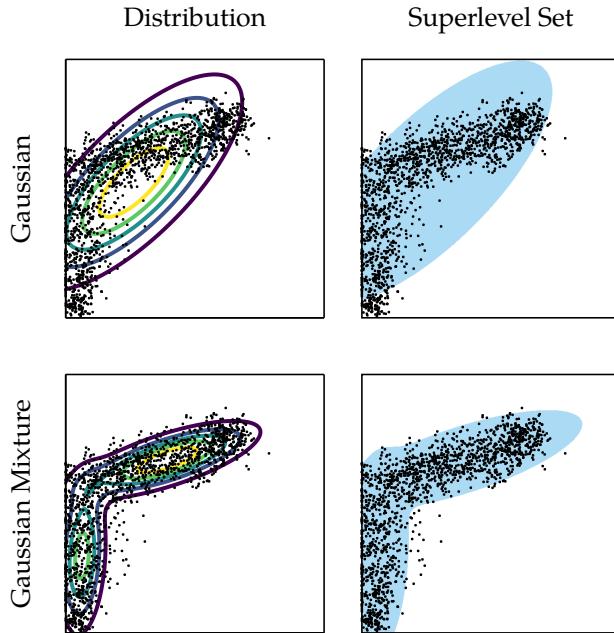


Figure 12.8. The ODD (blue, right column) defined using the superlevel set of a distribution (left column) for the data in figure 12.2. The top rows shows the ODD defined using a Gaussian distribution, and the bottom row shows the ODD defined using a mixture of Gaussians. The mixture of Gaussians is a more expressive distribution that better captures the data and results in a more accurate ODD.

increases, the volume of the space the data must cover increases exponentially. This increase in volume makes it difficult to adequately cover the space with a limited amount of data and can cause distance metrics to lose meaning.

One way to address the curse of dimensionality is to gather more data and use more expressive models. For example, we could represent high-dimensional distributions using an expressive model such as a normalizing flow. However, this approach may lead to overfitting or poor generalization.³ Another approach is to assume that the data lies on a lower-dimensional manifold and to use dimensionality reduction techniques⁴ to find this manifold.

Given a lower dimensional projection of the data, we can use the methods described in this section to define the ODD. When we get new data at runtime, we can project it onto this lower dimensional manifold and check whether it fits within the ODD. Figure 12.10 shows an example of a two-dimensional manifold for the aircraft taxi image observations. When creating a lower-dimensional representation of the data, it is important to ensure that the projection captures the relevant features of the data that define the ODD and that data outside the

³ For example, researchers have shown that normalizing flows trained on images often assign higher likelihoods to images outside the ODD. P. Kirichenko, P. Izmailov, and A.G. Wilson, “Why Normalizing Flows Fail to Detect Out-Of-Distribution Data,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 20578–20589, 2020.

⁴ Common approaches for dimensionality reduction include principle component analysis and autoencoders. A detailed overview is provided by B. Ghojogh, M. Crowley, F. Karay, and A. Ghodsi, *Elements of Dimensionality Reduction and Manifold Learning*. Springer, 2023.

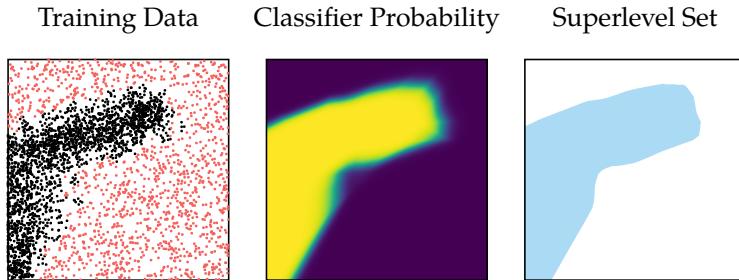


Figure 12.9. The ODD (blue, right plot) defined using the superlevel set of a classifier (middle plot) trained on the data in figure 12.2 as well as additional data sampled uniformly outside the ODD. The classifier outputs the probability that a point is in the ODD. The superlevel set is defined as the set of points for which the classifier outputs a probability greater than 0.5, though this can be treated as a free parameter to control conservatism.

ODD in the original space is projected outside the ODD in the lower-dimensional space. A representation that is not expressive enough may result in *feature collapse*, where far points in the original space are projected to nearby points in the lower-dimensional space.⁵ Figure 12.11 shows an example of feature collapse in the two-dimensional manifold for the aircraft taxi example.

12.2 Uncertainty Quantification

Another important aspect of runtime monitoring is *uncertainty quantification*, which allows us to understand our uncertainty in various aspects of the current and future behavior of a system. If we have high uncertainty, we may want to be more cautious and take actions to reduce this uncertainty. We can quantify uncertainty by creating models that predict some aspect of the current or future behavior of the system and understanding the uncertainty in their predictions. For example, we could train a model to predict the next state of the system and quantify the uncertainty in this prediction.

We may encounter two different types of uncertainty when monitoring a system. The first type of uncertainty is *output uncertainty*, which occurs when a single input can lead to multiple different outputs.⁶ For example, the output of the transition model of a system may be uncertain given the current state due to various sources of randomness in the real world such as the behavior of other agents. Another common cause of output uncertainty is sensor noise, which results in different possible observations for the same state. In previous chapters, we modeled this type of uncertainty using disturbance distributions.

⁵J. Postels, M. Segù, T. Sun, L. D. Sieber, L. Van Gool, F. Yu, and F. Tombari, “On the Practicality of Deterministic Epistemic Uncertainty,” in *International Conference on Learning Representations (ICLR)*, 2022.

⁶This type of uncertainty occurs due to inherent stochasticity in the real world and is also referred to as *aleatoric uncertainty* or *irreducible uncertainty*.

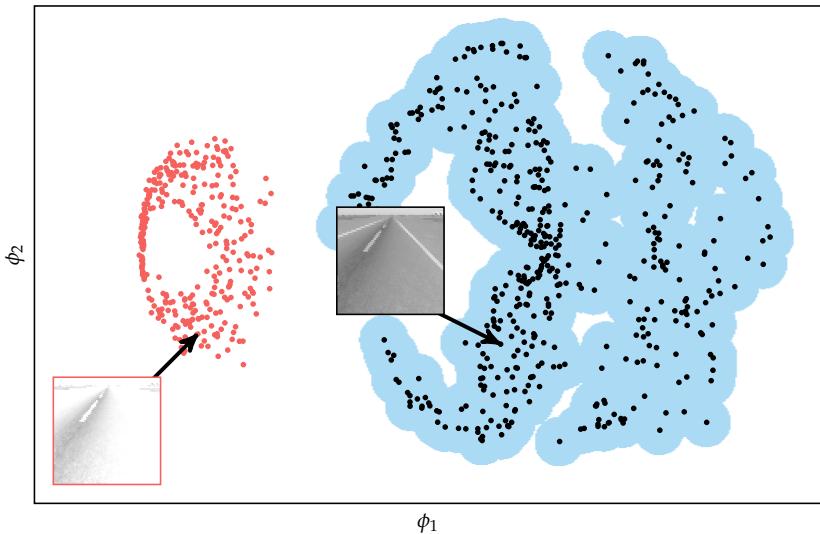


Figure 12.10. Projection of the high-dimensional image data for the aircraft taxi example onto a two-dimensional manifold. The black points represent the projection of runway images used for validation. The red points represent the projection runway images with different lighting conditions. The blue region shows a nearest neighbors representation of the ODD defined using the black points. The images with different lighting conditions are projected outside the ODD. The projection uses the encoder of a variational autoencoder, which is a common dimensionality reduction technique for image data. Y. Pu, Z. Gan, R. Henao, X. Yuan, C. Li, A. Stevens, and L. Carin, “Variational Autoencoder for Deep Learning of Images, Labels and Captions,” *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 29, 2016.

⁷ This type of uncertainty is also referred to as *epistemic uncertainty* or *reducible uncertainty*.

The second type of uncertainty is *model uncertainty*, which arises due to limitations of the model we are using to predict system behavior.⁷ A common limitation of data-driven models is a lack of data in certain regions of the input space. For example, if we train a model to predict system behavior using data from the ODD of the system, we cannot expect it to make accurate predictions outside the ODD. In other words, the model should have high uncertainty in regions of the input space with no data. This section outlines data-driven approaches to quantify both types of uncertainty. Outcome uncertainty is present in the data used to create the model, while model uncertainty occurs in regions of no data (figure 12.12).

12.2.1 Predicting Output Uncertainty

Because output uncertainty is inherent to the system, it will be present in the data we gather from the system. Therefore, we can use data to quantify outcome uncertainty by training a model to output the parameters of a distribution over its prediction. For example, if our goal is to predict the next state of the system, we can learn a model that outputs the mean and standard deviation of a Gaussian distribution conditioned on the input.

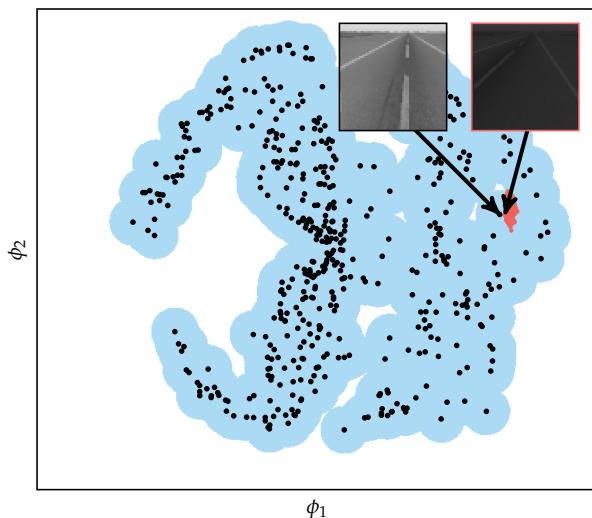


Figure 12.11. Example of feature collapse for the aircraft taxi system. The black points show the projection of runway images used for validation, and the blue region shows the nearest neighbors representation of the OOD. The red points show the projection of runway images with dark lighting conditions. While these images are outside the OOD in the original space, they are projected inside the OOD in the lower-dimensional space.

To learn the parameters of this model, we must minimize a *proper scoring rule*. A scoring rule is a function that takes as input both the true values and the predicted distribution and outputs a score that quantifies the quality of the predicted distribution. A proper scoring rule is a scoring rule that is minimized when the true distribution is equal to the predicted distribution. The negative log-likelihood is a common proper scoring rule for parameter learning.⁸

A common model used to predict uncertainty in continuous outputs is the conditional Gaussian model, which outputs the parameters of a Gaussian distribution conditioned on the input.⁹ Example 12.1 derives the negative log-likelihood objective for learning the parameters of a conditional Gaussian model. Figure 12.13 shows the result of fitting a model using this objective to the data set in figure 12.12. The variance of the model will be larger in regions where the output data is more spread out, indicating higher output uncertainty.

⁸ More details on proper scoring rules are provided by T. Gneiting and M. Katzfuss, "Probabilistic Forecasting," *Annual Review of Statistics and Its Application*, vol. 1, no. 1, pp. 125–151, 2014.

⁹ We could also select a different continuous distribution to predict or train a model to predict quantiles of a distribution in a process known as quantile regression. R. Koenker and K. F. Hallock, "Quantile Regression," *Journal of Economic Perspectives*, vol. 15, no. 4, pp. 143–156, 2001.

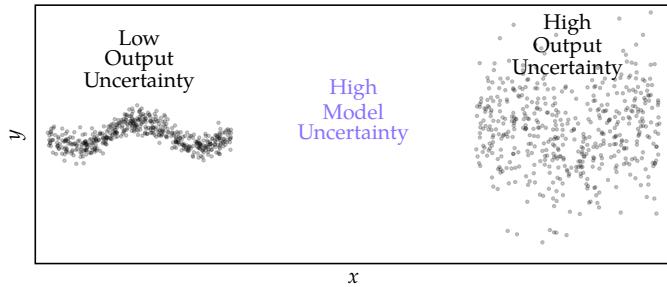


Figure 12.12. Output and model uncertainty in a data set. Output uncertainty is inherent uncertainty present in the data, while model uncertainty occurs where there is no data.

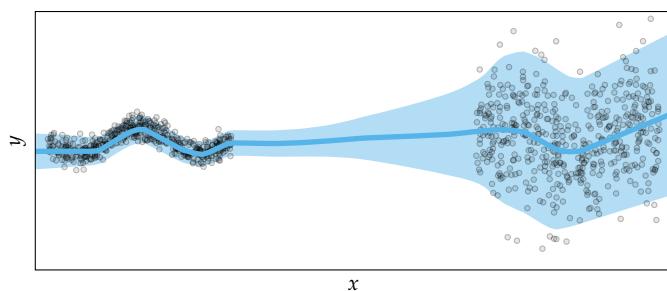


Figure 12.13. Fitting a conditional Gaussian model to the data set in figure 12.12 to quantify output uncertainty using the objective derived in example 12.1. We represent $p_\theta(y \mid x)$ as a neural network that outputs a mean and log standard deviation. The solid blue line shows the mean prediction, and the shaded region shows 2σ around the mean and represents the output uncertainty in the prediction. The model correctly predicts higher output uncertainty for the data on the right.

Suppose we want to quantify the outcome uncertainty when predicting a continuous variable y given an input x from a data set of (x, y) pairs. We can learn the parameters of the following conditional Gaussian model:

$$p_{\theta}(y | x) = \mathcal{N}(y | \mu_{\theta}(x), \sigma_{\theta}^2(x))$$

where $\mu_{\theta}(x)$ and $\sigma_{\theta}^2(x)$ are functions of x parameterized by θ . This model is similar to the model introduced in examples 2.4 and 2.5. However, instead of allowing only the mean to depend on the input x , we allow both the mean and variance to depend on x . We can learn the parameters θ by maximizing the likelihood of the data using the following optimization problem:

$$\begin{aligned}\hat{\theta} &= \arg \max_{\theta} \sum_{i=1}^m \log p_{\theta}(y_i | x_i) \\&= \arg \max_{\theta} \sum_{i=1}^m \log \mathcal{N}(y_i | \mu_{\theta}(x_i), \sigma_{\theta}^2(x_i)) \\&= \arg \max_{\theta} \sum_{i=1}^m \log \frac{1}{\sqrt{2\pi\sigma_{\theta}^2(x_i)}} \exp \left(-\frac{(y_i - \mu_{\theta}(x_i))^2}{2\sigma_{\theta}^2(x_i)} \right) \\&= \arg \max_{\theta} \sum_{i=1}^m \left[\log(1) - \log(\sqrt{2\pi}) - \frac{1}{2} \log(\sigma_{\theta}^2(x_i)) - \frac{(y_i - \mu_{\theta}(x_i))^2}{2\sigma_{\theta}^2(x_i)} \right] \\&= \arg \max_{\theta} \sum_{i=1}^m \left[-\frac{1}{2} \log(\sigma_{\theta}^2(x_i)) - \frac{(y_i - \mu_{\theta}(x_i))^2}{2\sigma_{\theta}^2(x_i)} \right] \\&= \arg \min_{\theta} \sum_{i=1}^m \left[\frac{(y_i - \mu_{\theta}(x_i))^2}{\sigma_{\theta}^2(x_i)} + \log(\sigma_{\theta}^2(x_i)) \right]\end{aligned}$$

Intuitively, the first term in the final objective encourages the model to predict a high variance when the squared error is high, and the second term penalizes high variances. This objective is commonly used in machine learning and is referred to as the Gaussian negative log-likelihood loss function. Figure 12.13 shows the result of fitting a model using this objective to the data set in figure 12.12.

Example 12.1. Learning the parameters of a conditional Gaussian model to quantify outcome uncertainty.

For discrete outputs, we can train a model $h_\theta(x)$ to output a vector \mathbf{y} that specifies the relative likelihood of each possible output given the input. To ensure that the model outputs valid probabilities, we apply a *softmax* function to the output of the model to obtain a vector of probabilities \mathbf{p} such that

$$p_i = \frac{\exp(y_i)}{\sum_{j=1}^k \exp(y_j)} \quad (12.1)$$

where k is the total number of possible outputs. We can learn the parameters θ of the model by maximizing the likelihood of the data given this model.

Given a distribution over predicted outputs, we can quantify our uncertainty using the *entropy* of the distribution. Higher entropy indicates higher uncertainty in the prediction, and we may want to flag situations that result in outputs with high entropy as potentially dangerous. The entropy of a conditional Gaussian distribution is a function of the variance of the distribution. A higher variance results in a higher entropy. For discrete distributions, the entropy is defined as

$$-\sum_{i=1}^k p_i \log p_i \quad (12.2)$$

where p_i is the probability of the i th output. A model that assigns equal probability to all outputs will have maximum entropy, indicating high uncertainty in the prediction (figure 12.14). We can use a threshold on entropy to create a runtime monitor that flags uncertain situations.

12.2.2 Calibrating Output Uncertainty

We can measure the performance of our uncertainty quantification model using the visual diagnostics and summary metrics outlined in sections 2.5.1 and 2.5.2. Because negative log-likelihood is a proper scoring rule, if we were able to perfectly minimize the negative log-likelihood of the data, the resulting model would be perfectly calibrated. However, in practice, we have limited model capacity and imperfect optimization, so the model distribution may not exactly match the true distribution. Therefore, it is common to *calibrate* the model after training to better match the true distribution.

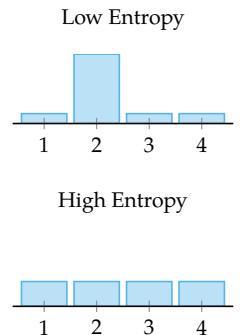


Figure 12.14. Entropy of a discrete distribution over four possible outputs. If the distribution assigns high probability to a single output, the entropy will be low. If the distribution assigns equal probability to all outputs, the entropy will be high.

Calibration techniques typically rely on a separate set of calibration data. The calibration data should consist of pairs of inputs and outputs that are sampled independently from the data distribution the model will encounter at runtime. We can use this data to assess baseline performance by comparing the model distribution to the distribution of the calibration data (figure 12.15). We can then apply calibration techniques to adjust the model distribution to better match the calibration data distribution.

A common technique to calibrate a model is to perform *histogram binning* on the desired uncertainty metric. For discrete outputs, a common uncertainty metric for calibration is the predicted probability of the correct output. For continuous outputs predicted using a Gaussian distribution, a common uncertainty metric for calibration is the predicted variance of the distribution. The first step in histogram binning is to divide the calibration data into bins using the predicted values of the uncertainty metric. We typically select the bin boundaries to create bins of equal width or equal number of samples.

After binning the data, we can calculate the actual value of the uncertainty metric for each bin. For example, for discrete outputs, we can calculate the average predicted probability of the correct output for each bin. We can then adjust the model predictions to match the actual values of the uncertainty metric in each bin. For example, if the model is underconfident in its predictions, we can increase the predicted probability of the model in the corresponding bins. When we obtain a new data point, we calculate its predicted uncertainty metric to determine which bin it belongs to. We can then use the actual value of the uncertainty metric in that bin to adjust the model prediction. Example 12.2 provides an example of this process.

Histogram binning requires storing the bin edges and corresponding calibration values at runtime. Other techniques focus on instead fitting a single calibration parameter to the data. For example, a common calibration technique for models that output probabilities of discrete outputs using a softmax function is to introduce a *precision parameter* λ such that

$$p_i = \frac{\exp(\lambda y_i)}{\sum_{j=1}^k \exp(\lambda y_j)} \quad (12.3)$$

This model is similar to the softmax response model introduced in section 2.4.2. We can select the precision parameter lambda that minimizes a proper scoring rule such as the negative log-likelihood of the calibration data.¹⁰ Figure 12.16

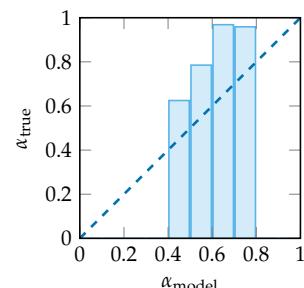
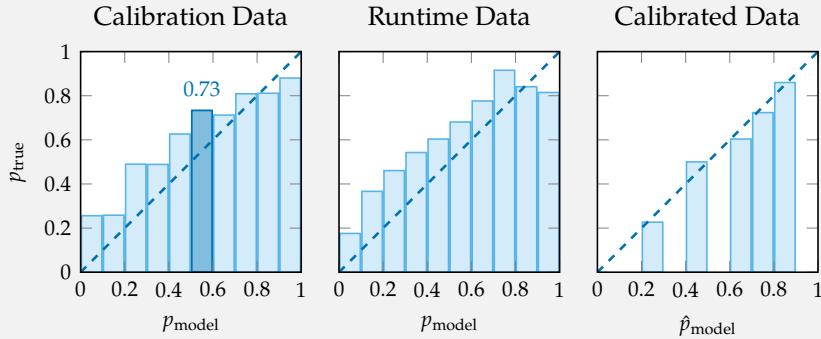


Figure 12.15. Calibration plot of a neural network model trained to predict the discrete actions of the continuum world agent using the data in figure 12.2. A calibrated model should match the dashed line. The model is poorly calibrated and tends to be underconfident in its predictions.

¹⁰ Because we are fitting a single parameter, the risk of overfitting is low, so we do not necessarily need a separate set of calibration data. This method is also sometimes referred to as *temperature scaling* with temperature $1/\lambda$. C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, “On Calibration of Modern Neural Networks,” in *International Conference on Machine Learning (ICML)*, 2017.

Suppose we have a model that predicts binary outputs for a system by predicting a probability p that the output is 1. Given a data set of calibration data, we can see that the model tends to be underconfident in its predictions (left). For example, for data points where the model predicts that the probability of the output being 1 is between 0.5 and 0.6, the actual probability of the output being 1 according to the calibration data is around 0.73. If we were to deploy this model at runtime without additional calibration, we would observe a similar trend (center).



We can use the bins of the model probability and their corresponding actual probabilities from the calibration data to adjust the model predictions. For example, suppose we get a new input at runtime that has a predicted probability according to the model of 0.52. This probability falls into the highlighted bin above where the actual probability is 0.73. We should therefore adjust that probability such that $\hat{p} = 0.73$ and use this probability to make runtime monitoring decisions. The plot on the right shows the result if we apply this calibration technique to runtime data in the plot on the right. The model is now well-calibrated and follows the dashed line.

Example 12.2. Histogram binning calibration for a model that predicts a binary discrete output. The rightmost plot shows the calibrated runtime data after applying the calibration technique. A well-calibrated model should follow the dashed line.

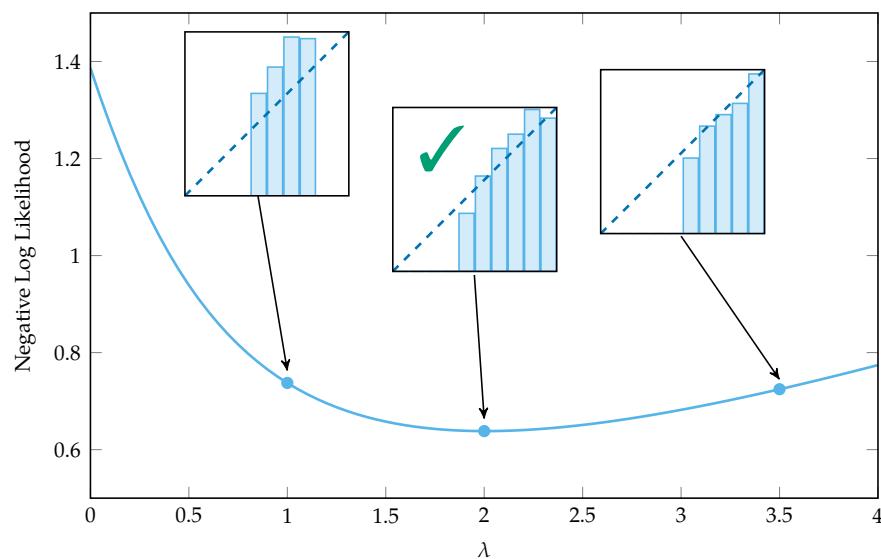


Figure 12.16. Calibrating to continuum world action prediction model shown in figure 12.15 using a precision parameter λ in the softmax model. The plot shows the negative log-likelihood of the calibration data for different values of λ . The calibration plots show the calibrated model predictions for different values of λ . We should select the value of λ that minimizes the negative log-likelihood of the calibration data (shown in the center).

shows the result of applying this calibration technique to calibrate the model in figure 12.15. For Gaussian models, we can apply a similar technique by introducing a single scaling parameter to the predicted variance of the model.

While fitting a single calibration parameter reduces the complexity of the calibration procedure, it is not as expressive as histogram binning. For example, there may not be a single precision parameter λ that adequately calibrates all bins of the model. Other calibration techniques use more complex models with multiple parameters to adjust the model predictions.¹¹

12.2.3 Prediction Sets

Another approach to quantifying uncertainty is to predict a set of possible outcomes rather than a single outcome. To create a prediction set, we must choose a desired level of *coverage*. The coverage of a prediction set is the probability that the true output lies within the set. For example, a prediction set with coverage of 0.95 should contain the true output 95% of the time. A large prediction set indicates high uncertainty, while a small prediction set indicates low uncertainty. For example, a large prediction set for the location of an autonomous vehicle indicates that we have high uncertainty in its true location.

¹¹ These techniques include Platt scaling and isotonic regression. A. Niculescu-Mizil and R. Caruana, “Predicting Good Probabilities with Supervised Learning,” in *International Conference on Machine Learning (ICML)*, 2005.

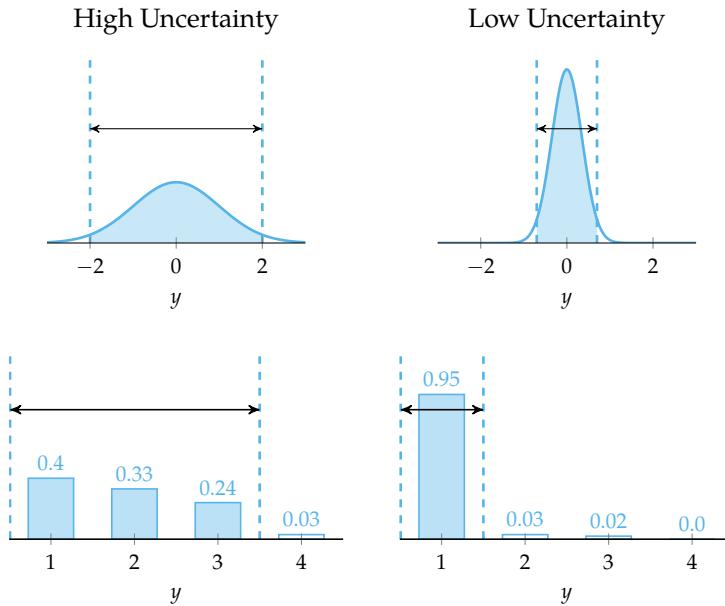


Figure 12.17. Prediction sets with $\alpha = 0.95$ for models that predict continuous and discrete outputs. The prediction sets are the regions between the dashed blue lines. The column on the left shows large prediction sets that indicate high uncertainty, while the column on the right shows small prediction sets that indicate low uncertainty.

Given a desired level of coverage c , we can derive a prediction set from a model trained using the methods in section 12.2.1 to predict a distribution over the output. For models that predict the parameters of a Gaussian distribution, it is common to create a prediction set centered around the predicted mean. We create this set by extending outward from the mean until the set includes c probability mass.¹² For models that predict the probabilities of discrete outputs, we can create a prediction set by adding outputs to the set in order of decreasing predicted probability until the sum of the probabilities of the outputs in the set exceeds c .

Figure 12.17 shows small and large prediction sets for both discrete and continuous models. Before generating prediction sets, it is important to ensure that the model is well-calibrated. If the model is not well-calibrated, the prediction sets may be too small or too large. For example, if the model is overconfident in its predictions, the prediction sets may be too small. We can calibrate the model using the techniques described in section 12.2.2.

We can also generate accurate prediction sets from an uncalibrated uncertainty measure using a technique known as *conformal prediction*.¹³ Similar to the techniques described in section 12.2.2, conformal prediction uses a calibration set

¹² Prediction sets do not need to be centered around the mean. Any prediction set that occupies c probability mass is a valid prediction set. However, the centered prediction set is the smallest possible set that contains c probability mass.

¹³ A detailed overview of conformal prediction is provided in A. N. Angelopoulos, S. Bates, et al., “Conformal Prediction: A Gentle Introduction,” *Foundations and Trends in Machine Learning*, vol. 16, no. 4, pp. 494–591, 2023.

to adjust the uncertainty measure. The data points in the calibration set must be *exchangeable*, meaning that the joint distribution of the data points does not change based on the order they appear.¹⁴ However, conformal prediction does not require a model that predicts a distribution over the output. Instead, it uses the calibration set to adjust any heuristic uncertainty measure.

The first step of conformal prediction involves identifying a heuristic notion of uncertainty. For example, we can use the parameters of an uncalibrated output distribution from a model trained using the methods in section 12.2.1. Next, conformal prediction requires a score function $s(x, y)$ that encodes how well the predicted uncertainty in the output conditioned on the input x matches the true output y . The score should be lower when there is good agreement between the prediction and the true output. Example 12.3 shows a score function for a Gaussian model, and example 12.4 shows a score function for a model that predicts discrete outputs.

The final step of conformal prediction involves computing the score for each point in the calibration set. We then find the score q that corresponds to the $\lceil(n+1)c\rceil/n$ quantile of the calibration scores, where $\lceil \cdot \rceil$ is the ceiling function and n is the number of points in the calibration set. Given a new input at runtime, the prediction set that guarantees a coverage of at least c is

$$\{y \mid s(x, y) \leq q\} \tag{12.4}$$

as long as the new input is exchangeable with the calibration set.¹⁵ Example 12.5 shows an example of a prediction set for a Gaussian model, and example 12.6 shows an example of a prediction set for a model that predicts discrete outputs.

The practicality of conformal prediction is highly dependent on the heuristic notion of uncertainty and the score function. If these choices do not accurately reflect the true uncertainty in the model, the prediction sets may be too large to be useful. It is also important to note that conformal prediction provides guarantees on the *marginal coverage* of the prediction sets and not the *conditional coverage*. In other words, the prediction sets will have a coverage of at least c on average, but the coverage may vary for different inputs (figure 12.18). Example 12.7 highlights a limitation of conformal prediction caused by this property.

¹⁴ Exchangeability is a more relaxed condition than independence. Any set of variables that are independent and identically distributed are also exchangeable.

¹⁵ A proof of this property is provided by A.N. Angelopoulos, S. Bates, et al., "Conformal Prediction: A Gentle Introduction," *Foundations and Trends in Machine Learning*, vol. 16, no. 4, pp. 494–591, 2023.

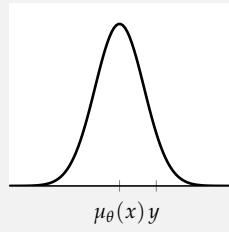
An example of a score function for a Gaussian model described in example 12.1 is

$$s(x, y) = \frac{|y - \mu_\theta(x)|}{\sigma_\theta(x)}$$

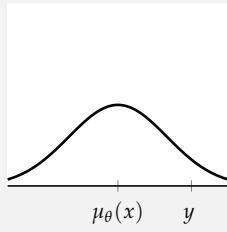
where $\mu_\theta(x)$ and $\sigma_\theta(x)$ are the predicted mean and standard deviation. The score will be low for inputs where the predicted mean is close to the true output. The score will also be low if the predicted output is far from the true output but the predicted standard deviation is large enough to account for this variation.

The plots below show the score function for different predicted distributions and corresponding true outputs. Although the true output is further from the predicted mean in the center plot than it is the left plot, the two data points produce the same score because the predicted standard deviation is larger in the center plot. In the plot on the right, the predicted standard deviation does not account for the increased gap between the predicted mean and the true output, resulting in a higher score.

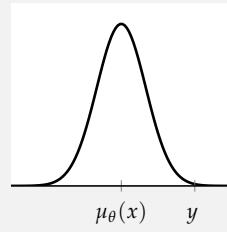
$$s(x, y) = 1.5$$



$$s(x, y) = 1.5$$



$$s(x, y) = 3.0$$



Example 12.3. Score function for conformal prediction using a conditional Gaussian model as a heuristic notion of uncertainty.

Suppose we want to use the softmax probabilities of a model $f_{\theta}(x)$ that predicts discrete outputs as a heuristic notion of uncertainty. We first define the function $\pi_j(x)$ to return the index of the output with the j th highest predicted probability. The score function can then be defined as

$$s(x, y) = \sum_{j=1}^k f_{\theta}(x)_{\pi_j(x)}$$

where k is selected such that $y = \pi_k(x)$. In other words, the procedure for determining the score is as follows. We first compute the predicted probabilities of the model for the input x . We then add the predicted probabilities to the score in decreasing order until we have added the probability for the true class.

The plots below show an example computation of the score function for a model that predicts probabilities of the action that a continuum world agent will take. The correct output is $y = \text{LEFT}$. The left plot shows the predicted probabilities of the model for each action. The right plot shows the predicted probabilities sorted in decreasing order. The score function is the sum of the probabilities in the sorted list until the true action is reached, which results in a score of 0.7.



Example 12.4. Score function for conformal prediction using a model that predicts probabilities of discrete outputs.

Plugging the score function from example 12.3 into equation (12.4) provides us with the following prediction set for a Gaussian model:

$$\begin{aligned}\{y \mid s(x, y) \leq q\} &= \left\{y \mid \frac{|y - \mu_{\theta}(x)|}{\sigma_{\theta}(x)} \leq q\right\} \\ &= \{y \mid |y - \mu_{\theta}(x)| \leq q\sigma_{\theta}(x)\}\end{aligned}$$

where q is the quantile of the calibration scores that corresponds to the desired coverage c . This prediction set is centered around the predicted mean and extends outward q standard deviations from the mean.

Intuitively, conformal prediction scales the standard deviation based on the results of the calibration data. For example, suppose we want to create a prediction set with 95% coverage. If the model was perfectly calibrated, we would expect $q \approx 2$ because 95% of the data should lie within two standard deviations of the mean. However, if the model was overconfident, we would expect $q > 2$ to produce larger prediction sets, and if the model was underconfident, we would expect $q < 2$ to produce smaller prediction sets.

Example 12.5. Prediction set from conformal prediction using a Gaussian model and the score function from example 12.3.

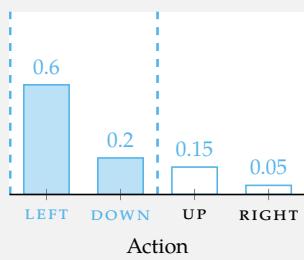
Plugging the score function from example 12.4 into equation (12.4) provides us with the following prediction set for a model that predicts probabilities of discrete outputs:

$$\{y \mid s(x, y) \leq q\} = \left\{ y \mid \sum_{j=1}^k f_\theta(x)_{\pi_j(x)} \leq q \right\}$$

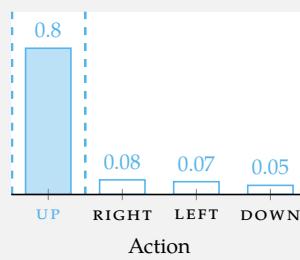
where q is the quantile of the calibration scores that corresponds to the desired coverage c . In other words, this prediction set is the set of outputs with the highest predicted probabilities that have a cumulative probability mass of at least c .

Suppose we found that the score q that corresponds to the 95% quantile of the calibration scores is 0.7. The prediction set for two different inputs is shown in the plots below. The input on the left results in a prediction set with two actions ($\{\text{LEFT}, \text{DOWN}\}$), while the input on the right results in a prediction set with one action ($\{\text{UP}\}$).

Prediction Set = $\{\text{LEFT}, \text{DOWN}\}$



Prediction Set = $\{\text{UP}\}$



Example 12.6. Prediction set from conformal prediction using a model that predicts probabilities of discrete outputs and the score function from example 12.4.

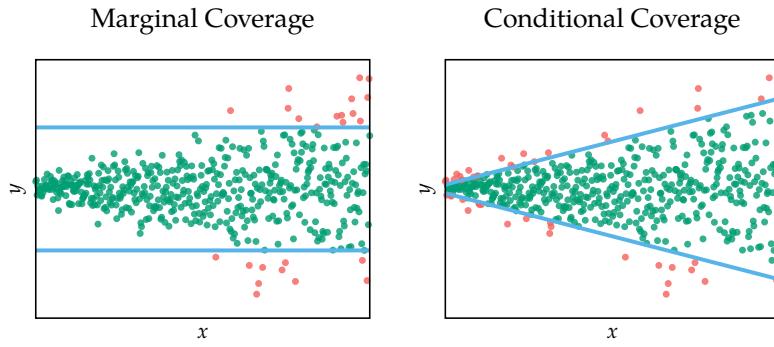


Figure 12.18. Comparison of marginal and conditional coverage of prediction sets. The blue lines indicate the prediction sets with 95% coverage. Green points are inside the prediction set, while red points are outside the prediction set. While 95% of the points are inside the prediction set on average in both plots, the plot on the left only provides marginal coverage, while the plot on the right provides both marginal and conditional coverage.

Suppose we perform conformal prediction on a model that predicts the location of an aircraft from runway images. We use calibration data in which 95% of the images are taken during the day, and the remaining 5% are taken at night. If we use conformal prediction to produce prediction sets with 95% coverage, the prediction sets will have a coverage of at least 95% on average. However, it is possible that the prediction sets for nighttime images may have a coverage of 0%, while the prediction sets for daytime images may have a coverage of 95%. In this case, if the aircraft is operating at night, the prediction sets will be inaccurate, resulting in potentially dangerous behavior. Therefore, it is important to consider the conditional coverage of the prediction sets when using conformal prediction.

Example 12.7. Example of the limitations of the coverage guarantees provided by conformal prediction.

12.2.4 Model Uncertainty

Model uncertainty arises from a lack of knowledge about the behavior of the system, which may arise when the system is operating outside its ODD. In these scenarios, we do not have data on the system's behavior, and we therefore cannot expect data-driven output uncertainty estimates to be accurate (figure 12.19). For this reason, we need other techniques to estimate model uncertainty.

One approach to estimating model uncertainty is to use a Bayesian approach in which we maintain a distribution over possible models. The key insight behind this approach is that there are many possible models that could have generated the data (figure 12.20), and we should account for this uncertainty when making predictions. We represent the distribution over possible models as $p(\theta | D)$, where θ are the parameters of the model and D is the data.

Given a new input, we can compute a distribution output using a process known as *Bayesian model averaging*. Bayesian model averaging uses the following equation to make predictions:

$$p(y | x, D) = \int p(y | x, \theta) p(\theta | D) d\theta \quad (12.5)$$

where $p(y | x, \theta)$ is the distribution over the prediction given the input and a specific instantiation of the parameters of the model. Intuitively, this equation computes the distribution over the output by averaging the predictions of all possible models weighted by the probability of each model given the data.

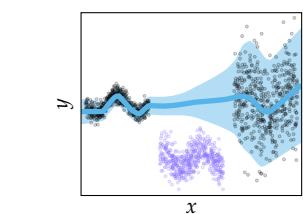
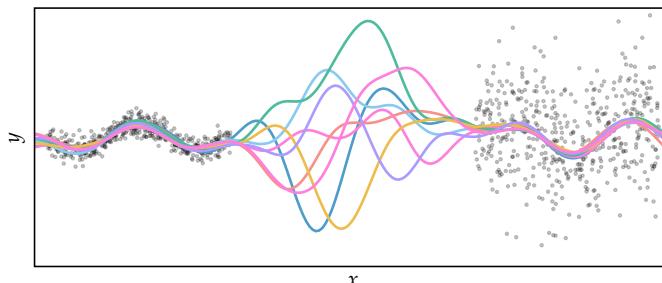


Figure 12.19. We cannot expect data-driven uncertainty estimates to be calibrated in regions of the input space where we do not have data. For example, it is possible that the data we were missing (purple) when we trained the model in figure 12.13 lies well outside the 2σ region of the model's predictions.

Figure 12.20. There are many possible models that could have generated the black data points. Each line represents a different model.

In general, equation (12.5) is intractable to compute because it requires integrating over the entire parameter space. However, we can use a variety of techniques to approximate this integral.¹⁶ One approach is to use MCMC to sample from

¹⁶ A. G. Wilson and P. Izmailov, "Bayesian Deep Learning and a Probabilistic Perspective of Generalization," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 4697–4708, 2020.

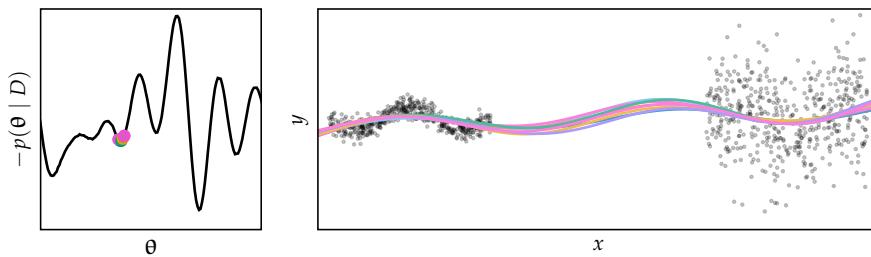


Figure 12.21. An example in which all models in the ensemble converge to the same local minima. In this case, we will underestimate our uncertainty.

the posterior distribution over the parameters of the model $p(\theta | D)$ (see sections 2.3.2 and 6.3). We can then use these samples to approximate the integral in equation (12.5). One drawback of this approach is that it requires running MCMC to make predictions at runtime, which can be computationally expensive.

Another approach to approximating the integral in equation (12.5) is to create an ensemble consisting of a set of models \mathcal{M} that all have high likelihood according to $p(\theta | D)$.¹⁷ One way to create these models is to train the same model multiple times with different initializations. We then approximate the integral as an equally weighted mixture of the predictions of each model as follows:

$$p(y | x, D) \approx \frac{1}{|\mathcal{M}|} \sum_{\theta \in \mathcal{M}} p(y | x, \theta) \quad (12.6)$$

Figure 12.23 shows an example of an ensemble trained to predict the actions of an agent in the continuum world using the data in figure 12.2.

It is important to ensure that the models in the ensemble have sufficient diversity. By starting from different initializations, we encourage each model to find a different local minima in the loss function (figure 12.22). However, this property is not guaranteed. It is possible that all models in the ensemble will still converge to the same local minima, which results in overconfident uncertainty estimates (figure 12.21). Therefore, it may be necessary to incorporate other heuristics into the training to ensure that the models in the ensemble are diverse.¹⁸

12.3 Failure Monitoring

Even when a system is operating with low uncertainty within its ODD, it may still end up in situations that lead to failure. Therefore, it is important to monitor

¹⁷ B. Lakshminarayanan, A. Pritzel, and C. Blundell, “Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles,” *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.

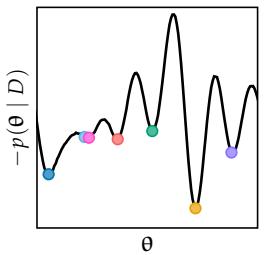


Figure 12.22. By training models with different initializations for θ , we hope to arrive at different local minima in the loss function. The colored points represent local minima for each mode in figure 12.20.

¹⁸ V. Dwaracherla, Z. Wen, I. Osband, X. Lu, S. M. Asghari, and B. Van Roy, “Ensembles for Uncertainty Estimation: Benefits of Prior Functions and Bootstrapping,” *Transactions on Machine Learning Research*, 2022.

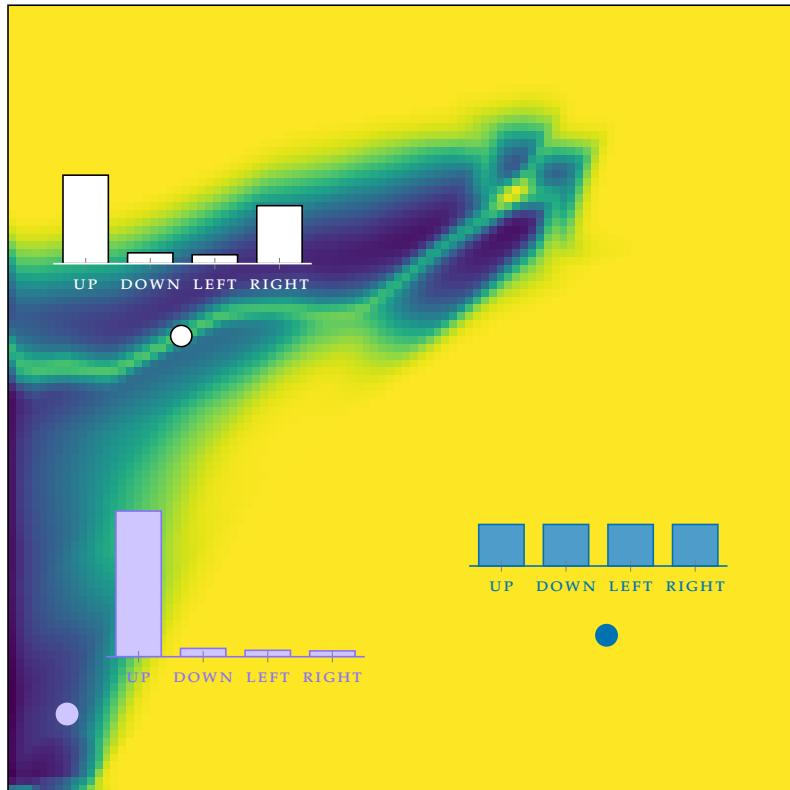


Figure 12.23. Ensemble trained to predict the actions of the continuum world agent using the data in figure 12.2. Brighter colors indicate higher entropy (and therefore high uncertainty) in the ensemble output. The bar plots show the distribution over actions for three different inputs. When the input is in the ODD (purple), the ensemble is confident in a single action. When the input is outside the ODD (blue), the ensemble assigns equal probability to all actions. The white point is on the decision boundary between the `UP` and `RIGHT` actions, so it assigns roughly equal probability to both actions and low probability to the others.

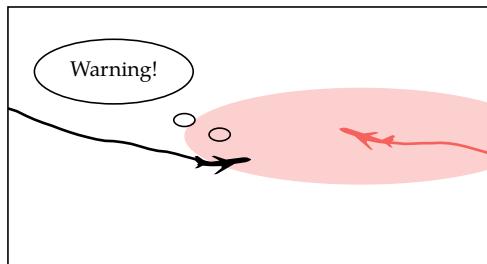


Figure 12.24. A simple failure monitoring system for the aircraft collision avoidance problem that issues a warning if the aircraft (black) enters a buffer region (red ellipse) around the intruder aircraft (red).

systems for potential dangerous situations within their ODD. A simple approach to failure monitoring is to create a set of heuristic rules or properties that describe dangerous scenarios using information that can be monitored at runtime. For example, in an aircraft collision avoidance scenarios, we can monitor the distance between the two aircraft and issue a warning if the distance falls below a certain threshold (figure 12.24). We typically want to set this threshold to be conservative so that we have time to take corrective actions mitigate the likelihood of a potential failure.

In addition to heuristic rules, we can make predictions about whether a particular situation is likely to lead to failure by performing some additional computation at runtime. Specifically, we can use a model of the system to run validation algorithms online during deployment. For example, we could use one of the reachability algorithms discussed in chapters 8 to 10 to determine whether failure states are reachable from the current state (first row of figure 12.25). If the specification for the system can be written as an LTL formula, we can use the techniques discussed in section 3.6 to convert the specification into a reachability specification using an automaton. We can then monitor the system at runtime by traversing the automaton.¹⁹

Monitors that use reachability analysis may be overly conservative, and we may instead only want to flag situations that have a significant probability of leading to failure. In this case, we can use the techniques discussed in chapter 7 to compute the probability of reaching a failure state from the current state. We can then use this probability to determine whether to issue a warning. The second row of figure 12.25 shows an example of a failure monitoring system that uses a probabilistic model to predict the likelihood of failure at each time step. In

¹⁹ A. Bauer, M. Leucker, and C. Schallhart, “Runtime Verification for LTL and TLTL,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, pp. 1–64, 2011.

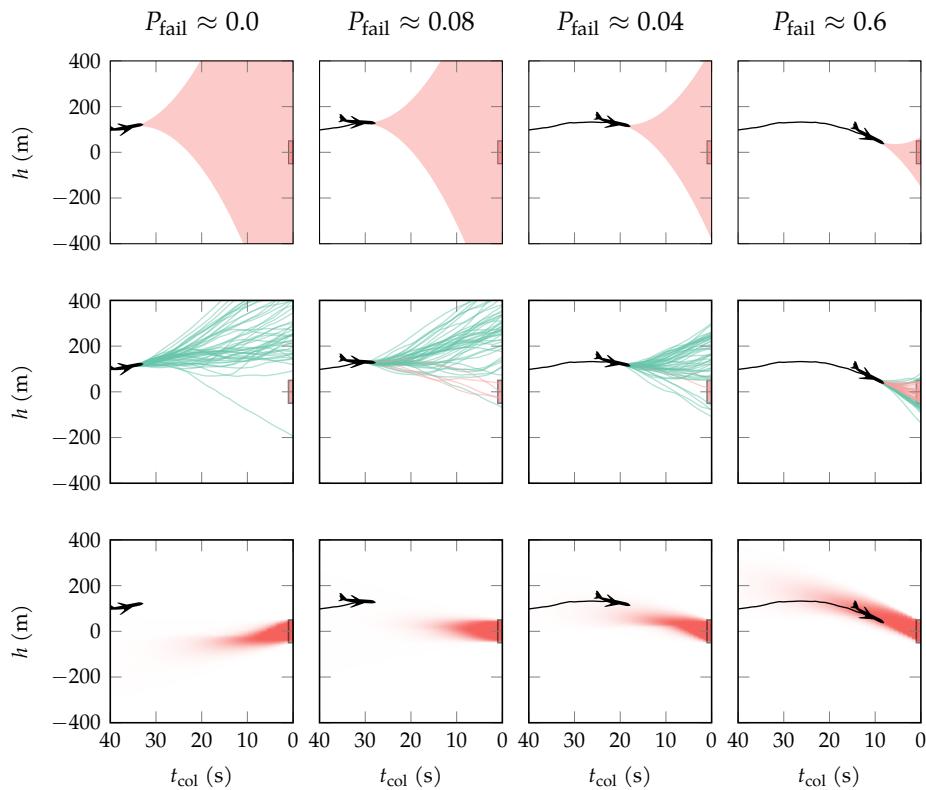


Figure 12.25. Online estimation of reachable set and the probability of failure for the aircraft collision avoidance problem. The reachable set estimation (top row) assumes a maximum acceleration and disturbance magnitude, and the online probability of failure estimation (second row) uses 50 roll-outs for algorithm 7.1. The third row shows the result of computing failure probabilities over the entire state space offline and looking up the results during operation. Brighter red indicates a higher failure probability. Using only the reachable sets to check safety may be overly conservative. We may instead only want to issue a warning if the probability of failure exceeds a threshold.

addition to failure probabilities, we can also predict other quantities of interest, such as the distribution over STL robustness values.

When we run validation algorithms offline, we often need to run them over the entire state space of the system because we do not know which states the system will encounter at runtime. In contrast, when we run validation algorithms online, we can focus only on the state that the system reaches. This property of online validation often allows us to save computation. However, running validation algorithms online can still be computationally expensive.

One way to further reduce computational expense is to store the results of offline validation algorithms and query them at runtime. For example, we could compute the probability of failure from all states in the state space offline and

store the results in a lookup table. At runtime, we can then query the lookup table to determine the probability of failure for the current state. The third row of figure 12.25 shows an example of a failure monitoring system that uses a lookup table to determine the probability of failure at each time step. For systems that have memory limitations, we can use a variety of compression techniques to store the results in a more compact form. For example, we could train a neural network to approximate the failure probability from any state in the state space.

12.4 Summary

- Runtime monitors allow us to monitor the safety of a system during deployment and take corrective action to avoid unsafe situations.
- We can use runtime monitors to check whether the assumptions we used during offline validation are still valid at runtime by checking whether the system is operating within its operational design domain.
- We can represent the operational design domain using a variety of set representations such as sets defined by nearest neighbors, convex hulls, or level sets.
- Output uncertainty and model uncertainty are two types of uncertainty we should monitor at runtime.
- Output uncertainty can be learned from data by training models to output uncertainty measures.
- Calibration techniques can be used to ensure that the uncertainty measures output by the model are accurate.
- Model uncertainty can be represented using an ensemble of models.
- Failure monitoring systems can be implemented using heuristic rules, reachability analysis, or probabilistic models.

12.5 Exercises

Exercise 12.1. What issue might arise when using a nearest neighbor ODD monitor with value of the distance threshold γ that is too small? What issue might arise when using a value of γ that is too large?

Solution: If γ is too small, the monitor may be overconservative and produce too many false alarms. If γ is too large, the monitor may be underconservative and fail to detect points outside the ODD.

Exercise 12.2. In what situations might we prefer to represent the ODD as a superlevel set of a distribution rather than a convex hull?

Solution: We might prefer to represent the ODD as a superlevel set of the probability density of a distribution when the data set used to define the ODD has outliers or regions of low data density. While a distribution will assign lower probability density to regions of lower data density, the convex hull approach has no way to ensure that regions of low data density are not included in the ODD.

Exercise 12.3. Suppose we have created an operational design domain monitor for an imaged-based pedestrian detection system on an autonomous vehicle designed to operate only during sunny conditions. The monitor projects the data onto a lower-dimensional manifold and applies the superlevel set approach to define the ODD in the lower-dimensional space. As we are testing our monitor, we find that the monitor fails to flag images taken during a cloudy day as outside the ODD. What might be the cause of this failure?

Solution: One possible cause of this failure is feature collapse during the projection onto a lower-dimensional manifold. It is likely the case that cloudy images are projected onto the same region of the lower-dimensional space as sunny images, causing the monitor to fail to detect the difference between the two.

Exercise 12.4. In addition to the Gaussian negative log-likelihood loss function, another way to train a regression model to quantify its output uncertainty is to train it to predict quantiles of the output distribution using a process known as quantile regression. We can create a function $f_\theta(x)$ that outputs the α -quantile of the output distribution over y given a dataset D by solving the following minimization problem:

$$\hat{\theta} = \arg \min_{\theta} \sum_{(x,y) \in D} \begin{cases} \alpha|y - f_\theta(x)| & \text{if } f_\theta(x_i) \leq y \\ (1 - \alpha)|y - f_\theta(x)| & \text{otherwise} \end{cases}$$

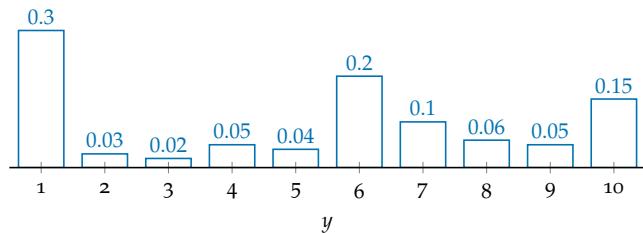
Suppose we want to find the 0.2-quantile of the data. Will we receive a higher penalty in the objective function if the model underestimates or overestimates the true value of y ? What about for the 0.8-quantile?

Solution: For the 0.2-quantile, the model will receive a higher penalty if it overestimates the true value of y . This result is consistent with our intuition because the 0.2-quantile of the data should be lower than the observed value of the output y more often than it is higher. For the 0.8-quantile, the model will receive a higher penalty if it underestimates the true value of y .

Exercise 12.5. Suppose we train a model to predict a distribution over the next word in a sentence by applying a softmax function to the output of the model. When we measure the calibration of the model, we notice that the model is overconfident in its predictions. We introduce a precision parameter λ to improve the calibration performance of the model. Should we try values for λ that are greater than or less than 1?

Solution: We should try values for λ that are less than 1. Since the model is overconfident in its predictions, we want to increase the entropy of the output distribution to make the model less confident in its predictions.

Exercise 12.6. Provide a prediction set with 50% coverage for the following distribution over the output of a model:



Solution: There are multiple possible answers. One prediction set with 50% coverage is $\{1, 6\}$. Another prediction set with 50% coverage is $\{2, 3, 4, 6, 9, 10\}$.

Exercise 12.7. Suppose we create an ensemble of m models that each predict a Gaussian distribution over the output. Show that the expected value of the output given the ensemble distribution is the average of the means of the individual models.

Solution: Using equation (12.6), we can calculate the output distribution for the ensemble as

$$\begin{aligned} p(y | x, D) &= \frac{1}{m} \sum_{i=1}^m p(y | x, \theta_i) \\ &= \frac{1}{m} \sum_{i=1}^m \mathcal{N}(y | \mu_i, \sigma_i^2) \end{aligned}$$

Therefore, the expected value of the output is

$$\begin{aligned}\mathbb{E}[y] &= \mathbb{E} \left[\frac{1}{m} \sum_{i=1}^m \mathcal{N}(y | \mu_i, \sigma_i^2) \right] \\ &= \frac{1}{m} \sum_{i=1}^m \mathbb{E} [\mathcal{N}(y | \mu_i, \sigma_i^2)] \\ &= \frac{1}{m} \sum_{i=1}^m \mu_i\end{aligned}$$

APPENDICES

A Systems

This appendix summarizes some of the systems used as examples in this book.

A.1 Default Implementations

Each component of the system must take in both its typical inputs as well as a disturbance. For components that do not use the disturbance, the disturbance is ignored using the default implementation in algorithm A.1.

```
(env::Environment)(s, a, x) = env(s, a)
(sensor::Sensor)(s, x) = sensor(s)
(agent::Agent)(o, x) = agent(o)
```

Algorithm A.1. Default implementation for components that do not use the disturbance.

Each component must also have a disturbance distribution that takes in the necessary inputs and returns a distribution over disturbances. Algorithm A.2 provides a default implementation for components that do not use the disturbance. It returns a distribution object that specifies that the component is deterministic. The environment component for each system must also have a function that returns the default distribution over initial states.

```
Ds(env::Environment, s, a) = Deterministic()
Da(agent::Agent, o) = Deterministic()
Do(sensor::Sensor, s) = Deterministic()
```

Algorithm A.2. Default implementation of the disturbance distribution for components that do not use the disturbance.

A.2 Simple Gaussian System

The simple Gaussian system consists of an environment with a single state variable that is sampled from a Gaussian distribution with mean 0 and standard deviation 1. After sampling an initial state, the system will remain in that state for all time regardless of the action. In other words, the system has no agent, and the state is fully observable. Algorithm A.3 defines each component of the system. A typical specification (chapter 3) for the simple Gaussian system is

$$\psi = \square(s > \gamma) \quad (\text{A.1})$$

which requires that the state be above a specified threshold γ .

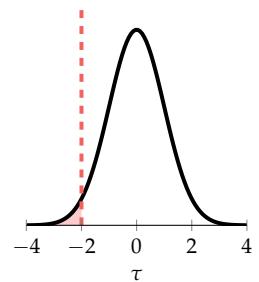


Figure A.1. The simple Gaussian system with failure threshold $\gamma = -2$. If the sampled state is below the threshold, the system fails.

```

struct SimpleGaussian <: Environment end
(env::SimpleGaussian)(s, a) = s
Ps(env::SimpleGaussian) = Normal()

struct NoAgent <: Agent end
(c::NoAgent)(s) = nothing

struct IdealSensor <: Sensor end
(sensor::IdealSensor)(s) = s
```

Algorithm A.3. Components of the simple Gaussian system. It samples an initial state and stays in that state for all time. The system has no agent, and the state is fully observable.

A.3 Multivariate Gaussian System

The multivariate Gaussian system is an extension to the simple Gaussian system where the state is two dimensions. The state is sampled from a multivariate Gaussian distribution with its mean at the origin and a covariance of the identity matrix. Algorithm A.4 defines the environment for this system. Similar to the simple Gaussian system, the multivariate Gaussian system has no agent, and the state is fully observable. A typical specification for the multivariate Gaussian distribution is

$$\psi = \square(s_1 > \gamma_1 \wedge s_2 > \gamma_2) \quad (\text{A.2})$$

where s_1 and s_2 are the first and second components of the state. We can also string together multiple specifications of this form to create multiple possible failure modes (figure A.2).

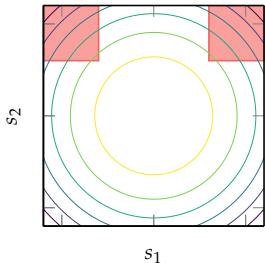


Figure A.2. Multivariate Gaussian environment with two possible failure modes (red shaded regions). The contours show the log density of the distribution over states with brighter colors indicating higher density.

```

struct MvGaussian <: Environment end
(env::MvGaussian)(s, a) = s
Ps(env::MvGaussian) = MvNormal(zeros(2), I)

```

Algorithm A.4. The environment for the multivariate Gaussian system. It uses the same agent and sensor as the simple Gaussian system.

A.4 Mass-Spring-Damper System

A common example of a linear system is a mass-spring-damper system, which can be used to model a wide range of physical systems such as a car suspension or a bridge. The state of the system is the position (relative to the resting point) p and velocity v of the mass ($s = [p, v]$), the action is the force β applied to the mass, and the observation is a noisy measurement of state. The equations of motion for a mass-spring-damper system are

$$\begin{aligned} p' &= p + v\Delta t \\ v' &= v + \left(-\frac{k}{m}p - \frac{c}{m}v + \frac{1}{m}\beta \right) \Delta t \end{aligned}$$

where m is the mass, k is the spring constant, c is the damping coefficient, and Δt is the discrete time step.

For linear reachability analysis, it is helpful to write the dynamics in the form of equation (8.6) as

$$T(\mathbf{s}, \mathbf{a}, \mathbf{x}_s) = \begin{bmatrix} 1 & \Delta t \\ -\frac{k}{m}\Delta t & 1 - \frac{c}{m}\Delta t \end{bmatrix} \begin{bmatrix} p \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m}\Delta t \end{bmatrix} \beta + \mathbf{x}_s = \mathbf{T}_s \mathbf{s} + \mathbf{T}_a \mathbf{a} + \mathbf{x}_s \quad (\text{A.3})$$

We control the mass-spring-damper using a proportional controller such that $\alpha = \Pi_o^\top = [0, -1]$ in equation (8.5) is the gain matrix. Similarly, we model the sensor as an additive noise sensor such that \mathbf{O}_s in equation (8.4) is the identity matrix and \mathbf{x}_o is the additive noise. Algorithm A.5 defines the components of the mass-spring-damper system. Typically, trajectories for this system will oscillate back and forth before coming to rest. In general, we want to ensure that the system remains stable, meaning that the position does not exceed some magnitude. We can write this specification in STL as

$$\psi = \square(|p| < \gamma) \quad (\text{A.4})$$

where γ is the maximum position magnitude of the mass. If the noise becomes too large, the system may become unstable.

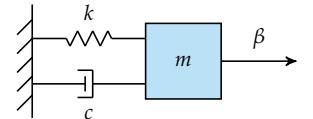


Figure A.3. A mass-spring-damper system with a mass m attached to a wall by a spring with spring constant k and a damper with damping coefficient c . The system is controlled by a force β applied to the mass.

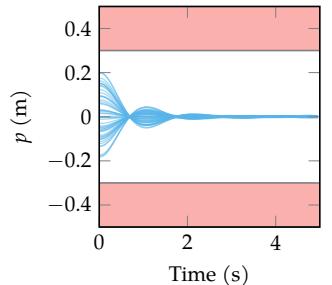


Figure A.4. Example trajectories of the mass-spring-damper system. The system oscillates back and forth before coming to rest.

```

@with_kw struct MassSpringDamper <: Environment
    m = 1.0    # mass
    k = 10.0   # spring constant
    c = 2.0    # damping coefficient
    dt = 0.05  # time step
end

Ts(env::MassSpringDamper) = [1 env.dt;
                             -env.k*env.dt/env.m 1-env.c*env.dt/env.m]
Ta(env::MassSpringDamper) = [0 env.dt/env.m]'
```

$$\text{function } (\text{env}::\text{MassSpringDamper})(\text{s}, \text{a})$$

$$\quad \text{return } \text{Ts}(\text{env}) * \text{s} + \text{Ta}(\text{env}) * \text{a}$$

$$\text{end}$$

$$\text{Ps}(\text{env}::\text{MassSpringDamper}) = \text{Product}([\text{Uniform}(-0.2, 0.2),$$

$$\quad \text{Uniform}(-1e-12, 1e-12)])$$

$$\text{struct AdditiveNoiseSensor <: Sensor}$$

$$\quad \text{Do} \# \text{ noise distribution}$$

$$\text{end}$$

$$(\text{sensor}::\text{AdditiveNoiseSensor})(\text{s}) = \text{sensor}(\text{s}, \text{rand}(\text{Do}(\text{sensor}), \text{s}))$$

$$(\text{sensor}::\text{AdditiveNoiseSensor})(\text{s}, \text{x}) = \text{s} + \text{x}$$

$$\text{Do}(\text{sensor}::\text{AdditiveNoiseSensor}, \text{s}) = \text{sensor}.\text{Do}$$

$$\text{Os}(\text{sensor}::\text{AdditiveNoiseSensor}) = \text{I}$$

$$\text{struct ProportionalController <: Agent}$$

$$\quad \alpha \# \text{ gain matrix } (c = \alpha^\top * o)$$

$$\text{end}$$

$$(\text{c}::\text{ProportionalController})(\text{o}) = \text{c}.\alpha^\top * \text{o}$$

$$\text{No}(\text{agent}::\text{ProportionalController}) = \text{agent}.\alpha^\top$$

Algorithm A.5. Components of the mass-spring-damper system. The environment implements equation (A.3). We implement the initial state distribution Ps to sample the initial position from an interval around the resting point and a small initial velocity. The sensor is an additive noise sensor, which samples noise from Do and adds it to the state. The agent uses a proportional controller, which multiplies a gain matrix by the observation.

A.5 Inverted Pendulum System

The inverted pendulum system is a classic nonlinear system in which we balance a pendulum by applying torques at its base. The state of the system is the angle θ of the pendulum and its angular velocity ω , the action is the torque applied, and the observation is a noisy measurement of the state. The next state for the inverted pendulum system is calculated as follows:

$$\begin{aligned}\omega' &= \omega + \left(\frac{3g}{2\ell} \sin \theta + \frac{3a}{m\ell^2} \right) \Delta t \\ \theta' &= \theta + \omega' \Delta t\end{aligned}\tag{A.5}$$

where m is the mass of the pendulum, ℓ is the length of the pendulum, g is the acceleration due to gravity, and Δt is the discrete time step. Algorithm A.6 implements these dynamics. The magnitude of the angular velocity of the pendulum is limited to ω_{\max} and the torque applied is limited to a_{\max} . The initial state for the pendulum is sampled uniformly from angles near upright with a small angular velocity.

```
@with_kw struct InvertedPendulum <: Environment
    m::Float64 = 1.0      # mass of the pendulum
    l::Float64 = 1.0      # length of the pendulum
    g::Float64 = 10.0     # acceleration due to gravity
    dt::Float64 = 0.05    # time step
    ω_max::Float64 = 8.0 # maximum angular velocity
    a_max::Float64 = 2.0 # maximum torque
end

function (env::InvertedPendulum)(s, a)
    θ, ω = s[1], s[2]
    dt, g, m, l = env.dt, env.g, env.m, env.l
    a = clamp(a, -env.a_max, env.a_max)
    ω = ω + (3g / (2 * l) * sin(θ) + 3 * a / (m * l^2)) * dt
    θ = θ + ω * dt
    ω = clamp(ω, -env.ω_max, env.ω_max)
    return [θ, ω]
end
Ps(env::InvertedPendulum) = Product([Uniform(-π / 16, π / 16),
                                         Uniform(-1.0, 1.0)])
```

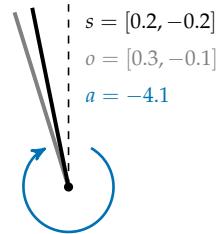


Figure A.5. State, action, and observation for the inverted pendulum system. The goal is to apply a torque at each time step to balance the pendulum upright. The observation is a noisy measurement of the state.

Algorithm A.6. Environment for the inverted pendulum system. The initial state distribution `Ps` samples the initial angle from an interval near upright and the angular velocity from an interval near zero.

Throughout the book, we use sensor noise as the main source of randomness in the inverted pendulum system. Therefore, the inverted pendulum system uses the

same additive noise sensor as the mass-spring-damper system in algorithm A.5. The pendulum also uses the proportional controller from algorithm A.5 with $\alpha \propto [-15, -8]$. The specification for the pendulum system requires that the angle of the pendulum remain within a specified range and is written in STL as

$$\psi = \square(|\theta| < \pi/4) \quad (\text{A.6})$$

Figure A.6 shows an example plot of both a success and failure trajectory for the inverted pendulum system.

A.6 Grid World System

The grid world system is an example of a system with discrete states, actions, and disturbances. The state of the system is a two-dimensional position on a grid, the action is a movement in one of four cardinal directions. Most of the time, the agent will move in the direction specified by the action, but with some probability, it will slip and move in a random direction. If the agent tries to move in a direction outside the grid, it will remain in its current cell. For example, if the agent moves to the left when it is in a cell on the leftmost side of the grid, it will remain in its current cell.

Algorithm A.7 implements the agent and environment for the grid world system. The policy of the agent is a lookup table that maps states to actions. Figure A.7 shows the action taken in each state for an agent that is trained to reach the green goal while avoiding the red obstacle. The system is fully observable and therefore uses the ideal sensor from algorithm A.3. The specification for the grid world system requires that the agent avoid the obstacle states and reach a goal state. We can write the specification in LTL as

$$\psi = \diamond G(s_t) \wedge \square \neg F(s_t) \quad (\text{A.7})$$

where $G(s_t)$ returns true if s_t is a goal state and $F(s_t)$ returns true if s_t is an obstacle state. Figure A.8 shows an example of the grid world system with an obstacle state in the center of the grid and a goal state near the upper right corner.

A.7 Continuum World System

The continuum world system is an extension of the grid world system with continuous states and disturbances. The action is still selected as one of the

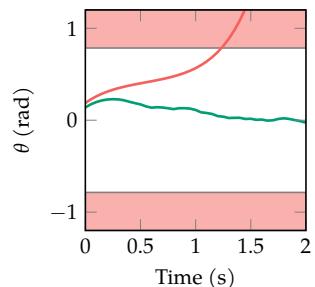


Figure A.6. Example of a success (green) and failure (red) trajectory for the inverted pendulum system. The failure trajectory enters the red failure region when its angle θ exceeds $\pi/4$.

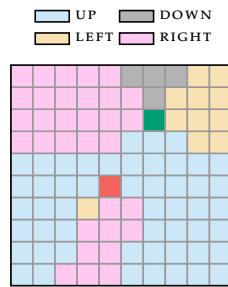


Figure A.7. Policy for the grid world agent.

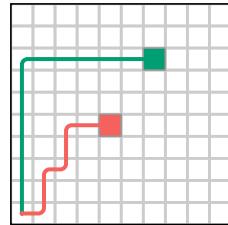


Figure A.8. Example of a success (green) and failure (red) trajectory for the grid world system. The failure trajectory enters the cell with the obstacle.

```

@with_kw struct GridWorld <: Environment
    size = (10, 10)                                # dimensions of the grid
    terminal_states = [[5,5],[7,8]]                # goal and obstacle states
    directions = [[0,1],[0,-1],[-1,0],[1,0]]        # up, down, left, right
    tprob = 0.7                                     # probability do not slip
end

function Ds(env::GridWorld, s, a)
    slip_prob = (1 - env.tprob) / (length(env.directions) - 1)
    probs = fill(slip_prob, length(env.directions))
    probs[a] = env.tprob
    return Categorical(probs)
end
(env::GridWorld)(s, a) = env(s, a, rand(Ds(env, s, a)))
function (env::GridWorld)(s, a, x)
    if s in env.terminal_states
        return s
    else
        dir = env.directions[x]
        return clamp.(s .+ dir, [1, 1], env.size)
    end
end
Ps(env::GridWorld) = SetCategorical([[1, 1]])

struct DiscreteAgent <: Agent
    policy # dictionary mapping states to actions
end
(c::DiscreteAgent)(o) = c.policy[o]

```

Algorithm A.7. Environment and agent for the grid world system. The actions correspond to indices in the `directions` field of the environment. The agent will move in the direction specified by the action with probability `tprob` and slip with probability $1 - \text{tprob}$. The agent uses a vector of actions called `policy` and a function that maps states to their corresponding indices in the `policy` vector.

cardinal directions; however, instead of slipping in one of the other cardinal directions, the agents slips in a random direction on the unit circle with higher probability of slipping in directions close to the desired direction. Specifically, we model this process by first adding a random vector x sampled from a multivariate Gaussian distribution to the desired direction and then normalizing the result to have a magnitude of 1.

The agent for the continuum world problem maps continuous states to discrete actions by interpolating a policy defined on a grid of discrete points. Specifically, each state in the grid corresponds to a set of values that represent the expected future return when taking each action from the state.¹ Given a new state that is not part of the grid, the agent uses multilinear interpolation to estimate the expected future return for each action. It then takes the action with the highest expected return. Figure A.9 shows the resulting policy for an agent trained to reach the green goal while avoiding the red obstacle. Algorithm A.8 defines the agent and environment for the continuum world problem.

```

@with_kw struct ContinuumWorld <: Environment
    size = [10, 10] # dimensions
    terminal_centers = [[4.5,4.5],[6.5,7.5]] # obstacle and goal centers
    terminal_radii = [0.5, 0.5] # radius of obstacle and goal
    directions = [[0,1],[0,-1],[-1,0],[1,0]] # up, down, left, right
    Σ = 0.5 * I(2)
end

Ds(env::ContinuumWorld, s, a) = MvNormal(zeros(2), env.Σ)
(env::ContinuumWorld)(s, a) = env(s, a, rand(Ds(env, s, a)))
function (env::ContinuumWorld)(s, a, x)
    is_terminal = [norm(s .- c) ≤ r
                  for (c, r) in zip(env.terminal_centers, env.terminal_radii)]
    if any(is_terminal)
        return s
    else
        dir = normalize(env.directions[a] .+ x)
        return clamp.(s .+ dir, [0, 0], env.size)
    end
end
Ps(env::ContinuumWorld) = SetCategorical([[0.5, 0.5]])

struct InterpAgent <: Agent
    grid # grid of discrete states using GridInterpolations.jl
    Q # corresponding state-action values
end
(c::InterpAgent)(s) = argmax(interpolate(c.grid, q, s) for q in c.Q)

```



Figure A.9. Policy for the continuum world agent.

¹These values make up the state-action value function. More details are provided by M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray, *Algorithms for Decision Making*. MIT Press, 2022.

Algorithm A.8. Environment and agent for the continuum world system. Instead of slipping in one of the cardinal directions, the agents slips in a random direction in the unit circle. The agent interpolates on a grid of discrete states using the `GridInterpolations.jl` package to determine the expected future return for each action. It then takes the action with the highest expected return.

The continuum world agent uses the same specification as the grid world system with continuous states instead of discrete states. The obstacles and goal for the continuum world system are represented as balls.

A.8 Aircraft Collision Avoidance System

The aircraft collision avoidance system involves issuing climb or descend advisories to an aircraft to avoid an intruder aircraft.² There are three actions corresponding to no advisory, commanding a 5 m/s descend, and commanding a 5 m/s climb. The intruder is approaching us head on, with a constant horizontal closing speed. The state is specified by the altitude h of our aircraft measured relative to the intruder aircraft, our vertical rate \dot{h} measured relative to the intruder aircraft, the previous action a_{prev} , and the time to potential collision t_{col} . Figure A.11 illustrates the problem scenario.

Given action a , the state variables are updated as follows

$$h' = h + \dot{h}\Delta t \quad (\text{A.8})$$

$$\dot{h}' = \dot{h} + (\ddot{h} + x_s) \quad (\text{A.9})$$

$$a'_{\text{prev}} = a \quad (\text{A.10})$$

$$t'_{\text{col}} = t_{\text{col}} - \Delta t \quad (\text{A.11})$$

where $\Delta t = 1$ s and x_s is noise added to the relative vertical rate to account for variations in intruder behavior. The value \ddot{h} is given by

$$\ddot{h} = \begin{cases} 0 & \text{if } a = \text{no advisory} \\ a/\Delta t & \text{if } |a - \dot{h}|/\Delta t < \ddot{h}_{\text{limit}} \\ \text{sign}(a - \dot{h})\ddot{h}_{\text{limit}} & \text{otherwise} \end{cases} \quad (\text{A.12})$$

where \ddot{h}_{limit} is the maximum allowable vertical acceleration.

The agent for the aircraft collision avoidance problem is similar to the continuum world agent in that it interpolates on a grid of discrete states to determine the action to take. We determine that the aircraft have nearly collided if their relative altitude is less than 50 m when the time to collision is zero. A time to collision of zero occurs at time step 41, resulting in the following STL specification:

$$\psi = \square_{[40,41]} |h| > 50 \quad (\text{A.13})$$

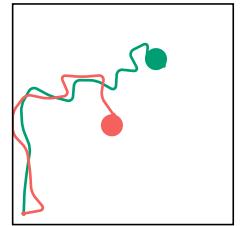


Figure A.10. Example of a success (green) and failure (red) trajectory for the continuum world system. The failure trajectory enters the circle with the obstacle.

²This formulation is a highly simplified version of the problem described by M. J. Kochenderfer and J. P. Chryssanthacopoulos, "Robust Airborne Collision Avoidance Through Dynamic Programming," Massachusetts Institute of Technology, Lincoln Laboratory, Project Report ATC-371, 2011.

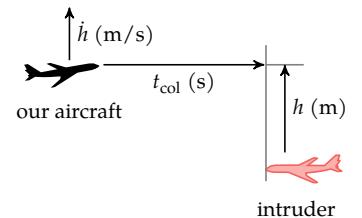


Figure A.11. State variables for the aircraft collision avoidance system.

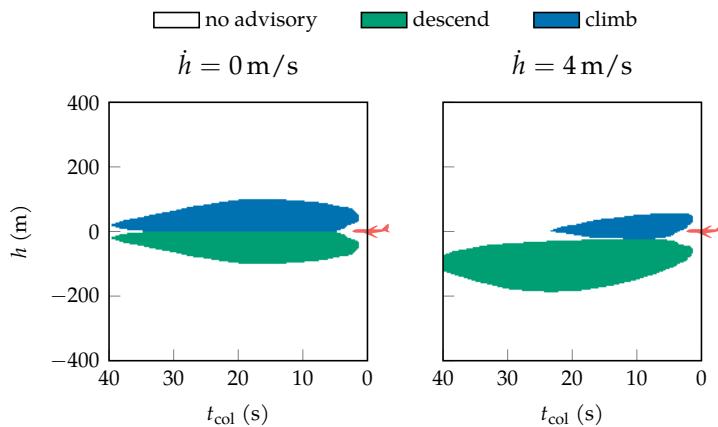


Figure A.12. Aircraft collision avoidance policy when the relative vertical rate is fixed at 0 m/s (left) and 4 m/s (right), and the previous action is fixed at no advisory. The colors represent the action taken by the agent in each state. The red aircraft represents the location of the intruder aircraft.

```

@with_kw struct CollisionAvoidance <: Environment
    dddh_max::Float64 = 1.0           # maximum vertical acceleration
    A::Vector{Float64} = [-5.0, 0.0, 5.0] # vertical rate commands
    Ds::Sampleable = Normal()          # vertical rate noise
end

Ds(env::CollisionAvoidance, s, a) = env.Ds
(env::CollisionAvoidance)(s, a) = env(s, a, rand(Ds(env, s, a)))
function (env::CollisionAvoidance)(s, a, x)
    a = env.A[a]
    h, dh, a_prev, τ = s
    h = h + dh
    if a != 0.0
        if abs(a - dh) < env.dddh_max
            dh += a
        else
            dh += sign(a - dh) * env.dddh_max
        end
    end
    a_prev = a
    τ = max(τ - 1.0, -1.0)
    return [h, dh + x, a_prev, τ]
end
Ps(env::CollisionAvoidance) = product_distribution(Uniform(-100, 100),
                                                    Uniform(-10, 10),
                                                    DiscreteNonParametric([0], [1.0]),
                                                    DiscreteNonParametric([40], [1.0]))

```

Algorithm A.9. Collision avoidance environment. The initial state distribution P_s samples the altitude h from a uniform distribution over $[-100, 100]$, the vertical rate \dot{h} from a uniform distribution over $[-10, 10]$. The previous action is fixed at no advisory, and we always bin with 40 seconds until a potential collision.

B Mathematical Concepts

This appendix provides a brief overview of some of the mathematical concepts used in this book.

B.1 Measure Spaces

Before introducing the definition of a measure space, we will first discuss the notion of a sigma-algebra over a set Ω . A sigma-algebra is a collection Σ of subsets of Ω such that

1. $\Omega \in \Sigma$.
2. If $E \in \Sigma$, then $\Omega \setminus E \in \Sigma$ (*closed under complementation*).
3. If $E_1, E_2, E_3, \dots \in \Sigma$, then $E_1 \cup E_2 \cup E_3 \dots \in \Sigma$ (*closed under countable unions*).

An element $E \in \Sigma$ is called a *measurable set*.

A *measure space* is defined by a set Ω , a sigma-algebra Σ , and a *measure* $\mu : \Omega \rightarrow \mathbb{R} \cup \{\infty\}$. For μ to be a measure, the following properties must hold:

1. If $E \in \Sigma$, then $\mu(E) \geq 0$ (*nonnegativity*).
2. $\mu(\emptyset) = 0$.
3. If $E_1, E_2, E_3, \dots \in \Sigma$ are pairwise disjoint, then $\mu(E_1 \cup E_2 \cup E_3 \dots) = \mu(E_1) + \mu(E_2) + \mu(E_3) + \dots$ (*countable additivity*).

B.2 Probability Spaces

A *probability space* is a measure space (Ω, Σ, μ) with the requirement that $\mu(\Omega) = 1$. In the context of probability spaces, Ω is called the *sample space*, Σ is called the *event space*, and μ (or, more commonly, P) is the *probability measure*. The *probability axioms*¹ refer to the nonnegativity and countable additivity properties of measure spaces, together with the requirement that $\mu(\Omega) = 1$.

B.3 Metric Spaces

A set with a *metric* is called a *metric space*. A metric d , sometimes called a *distance metric*, is a function that maps pairs of elements in X to nonnegative real numbers such that for all $x, y, z \in X$:²

1. $d(x, y) = 0$ if and only if $x = y$ (*identity of indiscernibles*).
2. $d(x, y) = d(y, x)$ (*symmetry*).
3. $d(x, y) \leq d(x, z) + d(z, y)$ (*triangle inequality*).

¹ These axioms are sometimes called the *Kolmogorov axioms*. A. Kolmogorov, *Foundations of the Theory of Probability*, 2nd ed. Chelsea, 1956.

² In section 3.2, we use a less precise definition of *metric* to refer to any function that maps system behavior to a real value.

B.4 Normed Vector Spaces

A *normed vector space* consists of a *vector space* X and a norm $\|\cdot\|$ that maps elements of X to nonnegative real numbers such that for all scalars α and vectors $\mathbf{x}, \mathbf{y} \in X$:

1. $\|\mathbf{x}\| = 0$ if and only if $\mathbf{x} = \mathbf{0}$.
2. $\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|$ (*absolutely homogeneous*).
3. $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ (*triangle inequality*).

The L_p norms are a commonly used set of norms parameterized by a scalar $p \geq 1$. The L_p norm of vector \mathbf{x} is

$$\|\mathbf{x}\|_p = \lim_{\rho \rightarrow p} (|x_1|^p + |x_2|^p + \dots + |x_n|^p)^{\frac{1}{p}} \quad (\text{B.1})$$

where the limit is necessary for defining the infinity norm, L_∞ . Several L_p norms are shown in figure B.1.

Norms can be used to induce distance metrics in vector spaces by defining the metric $d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|$. We can then, for example, use an L_p norm to define distances.

$$L_1: \|\mathbf{x}\|_1 = |x_1| + |x_2| + \cdots + |x_n|$$

This metric is often referred to as the *taxicab norm*.

$$L_2: \|\mathbf{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

This metric is often referred to as the *Euclidean norm*.

$$L_\infty: \|\mathbf{x}\|_\infty = \max(|x_1|, |x_2|, \dots, |x_n|)$$

This metric is often referred to as the *max norm*, *Chebyshev norm*, or *chessboard norm*. The latter name comes from the minimum number of moves that a king needs to move between two squares in chess.

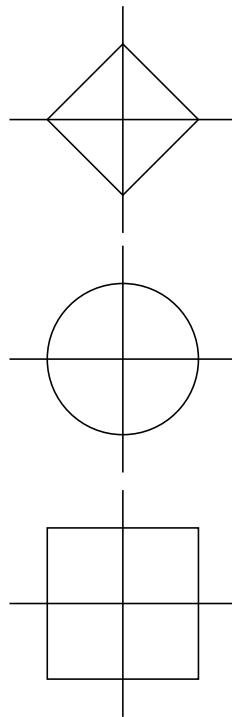


Figure B.1. Common L_p norms. The illustrations show the shape of the norm contours in two dimensions. All points on the contour are equidistant from the origin under that norm.

B.5 Positive Definiteness

A symmetric matrix \mathbf{A} is *positive definite* if $\mathbf{x}^\top \mathbf{A} \mathbf{x}$ is positive for all points other than the origin. In other words, $\mathbf{x}^\top \mathbf{A} \mathbf{x} > 0$ for all $\mathbf{x} \neq \mathbf{0}$. A symmetric matrix \mathbf{A} is *positive semidefinite* if $\mathbf{x}^\top \mathbf{A} \mathbf{x}$ is always nonnegative. In other words, $\mathbf{x}^\top \mathbf{A} \mathbf{x} \geq 0$ for all \mathbf{x} .

B.6 Information Content

If we have a discrete distribution that assigns probability $P(x)$ to value x , the *information content*³ of observing x is given by

$$I(x) = -\log P(x) \quad (\text{B.2})$$

The unit of information content depends on the base of the logarithm. We generally assume natural logarithms (with base e), making the unit *nat*, which is short for *natural*. In information theoretic contexts, the base is often 2, making the unit *bit*. We can think of this quantity as the number of bits required to transmit the value x according to an optimal message encoding when the distribution over messages follows the specified distribution.

³ Sometimes information content is referred to as *Shannon information*, in honor of Claude Shannon, the founder of the field of information theory. C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, no. 4, pp. 623–656, 1948.

B.7 Entropy

Entropy is an information theoretic measure of uncertainty. The entropy associated with a discrete random variable X is the expected information content:

$$H(X) = \mathbb{E}_x[I(x)] = \sum_x P(x)I(x) = -\sum_x P(x)\log P(x) \quad (\text{B.3})$$

where $P(x)$ is the mass assigned to x . For a continuous distribution where $p(x)$ is the density assigned to x , the *differential entropy* (also known as *continuous entropy*) is defined to be

$$h(X) = \int p(x)I(x)dx = -\int p(x)\log p(x)dx \quad (\text{B.4})$$

B.8 Cross Entropy

The *cross entropy* of one distribution relative to another can be defined in terms of expected information content. If we have one discrete distribution with mass function $P(x)$ and another with mass function $Q(x)$, then the cross entropy of P relative to Q is given by

$$H(P, Q) = -\mathbb{E}_{x \sim P}[\log Q(x)] = -\sum_x P(x) \log Q(x) \quad (\text{B.5})$$

For continuous distributions with density functions $p(x)$ and $q(x)$, we have

$$H(p, q) = -\int p(x) \log q(x) dx \quad (\text{B.6})$$

B.9 Relative Entropy

Relative entropy, also called the *Kullback-Leibler (KL) divergence*, is a measure of how one probability distribution is different from a reference distribution.⁴ If $P(x)$ and $Q(x)$ are mass functions, then the KL divergence from Q to P is the expectation of the logarithmic differences, with the expectation using P :

$$D_{\text{KL}}(P \parallel Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)} = -\sum_x P(x) \log \frac{Q(x)}{P(x)} \quad (\text{B.7})$$

This quantity is defined only if the support of P is a subset of that of Q . The summation is over the support of P to avoid division by zero.

For continuous distributions with density functions $p(x)$ and $q(x)$, we have

$$D_{\text{KL}}(p \parallel q) = \int p(x) \log \frac{p(x)}{q(x)} dx = -\int p(x) \log \frac{q(x)}{p(x)} dx \quad (\text{B.8})$$

Similarly, this quantity is defined only if the support of p is a subset of that of q . The integral is over the support of p to avoid division by zero.

⁴ Named for the two American mathematicians who introduced this measure, Solomon Kullback (1907–1994) and Richard A. Leibler (1914–2003). S. Kullback and R. A. Leibler, “On Information and Sufficiency,” *Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951. S. Kullback, *Information Theory and Statistics*. Wiley, 1959.

B.10 Taylor Expansion

The *Taylor expansion*,⁵ also called the *Taylor series*, of a function is important to many approximations used in this book. From the *first fundamental theorem of calculus*,⁶ we know that

$$f(x + h) = f(x) + \int_0^h f'(x + a) da \quad (\text{B.9})$$

⁵ Named for the English mathematician Brook Taylor (1685–1731) who introduced the concept.

⁶ The first fundamental theorem of calculus relates a function to the integral of its derivative:

$$f(b) - f(a) = \int_a^b f'(x) dx$$

Nesting this definition produces the Taylor expansion of f about x :

$$f(x+h) = f(x) + \int_0^h \left(f'(x) + \int_0^a f''(x+b) db \right) da \quad (\text{B.10})$$

$$= f(x) + f'(x)h + \int_0^h \int_0^a f''(x+b) db da \quad (\text{B.11})$$

$$= f(x) + f'(x)h + \int_0^h \int_0^a \left(f''(x) + \int_0^b f'''(x+c) dc \right) db da \quad (\text{B.12})$$

$$= f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \int_0^h \int_0^a \int_0^b f'''(x+c) dc db da \quad (\text{B.13})$$

$$\vdots \quad (\text{B.14})$$

$$= f(x) + \frac{f'(x)}{1!}h + \frac{f''(x)}{2!}h^2 + \frac{f'''(x)}{3!}h^3 + \dots \quad (\text{B.15})$$

$$= \sum_{n=0}^{\infty} \frac{f^{(n)}(x)}{n!} h^n \quad (\text{B.16})$$

In the formulation given here, x is typically fixed and the function is evaluated in terms of h . It is often more convenient to write the Taylor expansion of $f(x)$ about a point a such that it remains a function of x :

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x-a)^n \quad (\text{B.17})$$

The Taylor expansion represents a function as an infinite sum of polynomial terms based on repeated derivatives at a single point. Any analytic function can be represented by its Taylor expansion within a local neighborhood.

A function can be locally approximated by using the first few terms of the Taylor expansion. Figure B.2 shows increasingly better approximations for $\cos(x)$ about $x = 1$. Including more terms increases the accuracy of the local approximation, but error still accumulates as one moves away from the expansion point.

A linear *Taylor approximation* uses the first two terms of the Taylor expansion:

$$f(x) \approx f(a) + f'(a)(x-a) \quad (\text{B.18})$$

A quadratic Taylor approximation uses the first three terms:

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 \quad (\text{B.19})$$

and so on.

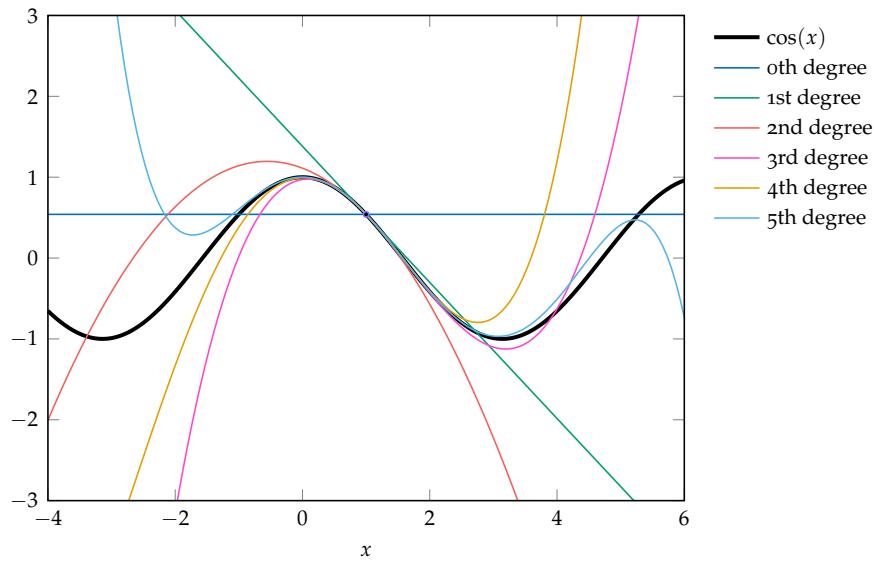


Figure B.2. Successive approximations of $\cos(x)$ about 1 based on the first n terms of the Taylor expansion.

In multiple dimensions, the Taylor expansion about \mathbf{a} generalizes to

$$f(\mathbf{x}) = f(\mathbf{a}) + \nabla f(\mathbf{a})^\top (\mathbf{x} - \mathbf{a}) + \frac{1}{2}(\mathbf{x} - \mathbf{a})^\top \nabla^2 f(\mathbf{a})(\mathbf{x} - \mathbf{a}) + \dots \quad (\text{B.20})$$

The first two terms form the tangent plane at \mathbf{a} . The third term incorporates local curvature. This book will use only the first three terms shown here.

C Neural Representations

Neural networks are parametric representations of nonlinear functions.¹ The function represented by a neural network is differentiable, allowing gradient-based optimization algorithms such as stochastic gradient descent to optimize their parameters to better approximate desired input-output relationships.² Neural representations can be helpful in a variety of contexts related to validation, especially for tasks that require expressive, flexible models. We also may want to validate systems with neural network components, as discussed in section 9.7

A *neural network* is a differentiable function $y = f_\theta(x)$ that maps inputs x to produce outputs y and is parameterized by θ . Modern neural networks may have millions of parameters and can be used to convert inputs in the form of high-dimensional images or video into high-dimensional outputs like multidimensional classifications or speech.

The parameters of the network θ are generally tuned to minimize a scalar *loss function* $\ell(f_\theta(x), y)$ that is related to how far the network output is from the desired output. Both the loss function and the neural network are differentiable, allowing us to use the gradient of the loss function with respect to the parameterization $\nabla_\theta \ell$ to iteratively improve the parameterization. This process is often referred to as neural network *training* or *parameter tuning*. It is demonstrated in example C.1.

Neural networks are typically trained on a data set of input-output pairs D . In this case, we tune the parameters to minimize the aggregate loss over the data set:

$$\arg \min_{\theta} \sum_{(x,y) \in D} \ell(f_\theta(x), y) \quad (\text{C.1})$$

Data sets for modern problems tend to be very large, making the gradient of equation (C.1) expensive to evaluate. It is common to sample random subsets of the training data in each iteration, using these *batches* to compute the loss gradient.

¹ The name derives from the loose inspiration of networks of neurons in biological brains. We will not discuss these biological connections, but an overview and historical perspective is provided by B. Müller, J. Reinhardt, and M. T. Strickland, *Neural Networks*, Springer, 1995.

² This optimization process when applied to neural networks with many layers, as we will discuss shortly, is often called *deep learning*. Several textbooks are dedicated entirely to these techniques, including I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016. The Julia package `Flux.jl` provides efficient implementations of various learning algorithms.

Consider a very simple neural network, $f_{\theta}(x) = \theta_1 + \theta_2 x$. We wish our neural network to take the square footage x of a home and predict its price y_{pred} . We want to minimize the squared deviation between the predicted housing price and the true housing price by the loss function $\ell(y_{\text{pred}}, y_{\text{true}}) = (y_{\text{pred}} - y_{\text{true}})^2$. Given a training pair, we can compute the gradient:

$$\begin{aligned}\nabla_{\theta} \ell(f(x), y_{\text{true}}) &= \nabla_{\theta} (\theta_1 + \theta_2 x - y_{\text{true}})^2 \\ &= \begin{bmatrix} 2(\theta_1 + \theta_2 x - y_{\text{true}}) \\ 2(\theta_1 + \theta_2 x - y_{\text{true}})x \end{bmatrix}\end{aligned}$$

If our initial parameterization were $\theta = [10,000, 123]$ and we had the input-output pair ($x = 2,500, y_{\text{true}} = 360,000$), then the loss gradient would be $\nabla_{\theta} \ell = [-85,000, -2.125 \times 10^8]$. We would take a small step in the opposite direction to improve our function approximation.

In addition to reducing computation, computing gradients with smaller batch sizes introduces some stochasticity to the gradient, which helps training to avoid getting stuck in local minima.

Neural networks are typically constructed to pass the input through a series of layers.³ Networks with many layers are often called *deep*. In *feedforward networks*, each layer applies an affine transform, followed by a nonlinear *activation function* applied elementwise:⁴

$$\mathbf{x}' = \phi.(\mathbf{W}\mathbf{x} + \mathbf{b}) \quad (\text{C.2})$$

where matrix \mathbf{W} and vector \mathbf{b} are parameters associated with the layer. A fully connected layer is shown in figure C.1. The dimension of the output layer is different from that of the input layer when \mathbf{W} is nonsquare. Figure C.2 shows a more compact depiction of the same network.

Example C.1. The fundamentals of neural networks and parameter tuning.

³ A sufficiently large, single-layer neural network can, in theory, approximate any function. See A. Pinkus, “Approximation Theory of the MLP Model in Neural Networks,” *Acta Numerica*, vol. 8, pp. 143–195, 1999.

⁴ The nonlinearity introduced by the activation function provides something analogous to the activation behavior of biological neurons, in which input buildup eventually causes a neuron to fire. A. L. Hodgkin and A. F. Huxley, “A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in Nerve,” *Journal of Physiology*, vol. 117, no. 4, pp. 500–544, 1952.

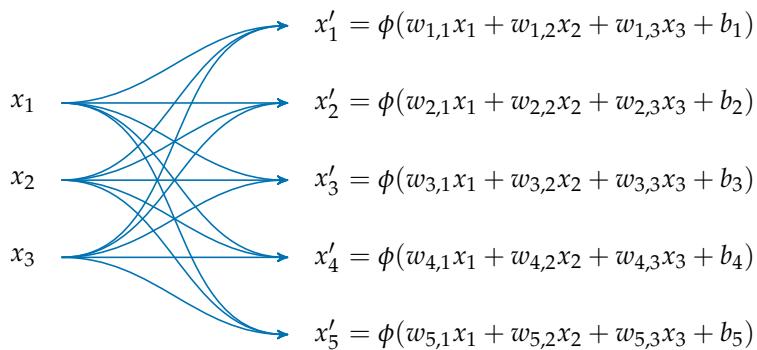


Figure C.1. A fully connected layer with a three-component input and a five-component output.

If there are no activation functions between them, multiple successive affine transformations can be collapsed into a single, equivalent affine transform:

$$\mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = \mathbf{W}_2\mathbf{W}_1\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2) \quad (\text{C.3})$$

These nonlinearities are necessary to allow neural networks to adapt to fit arbitrary target functions. To illustrate, figure C.3 shows the output of a neural network trained to approximate a nonlinear function.

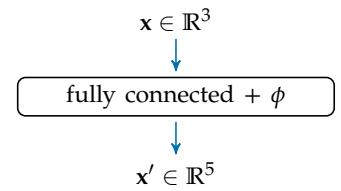
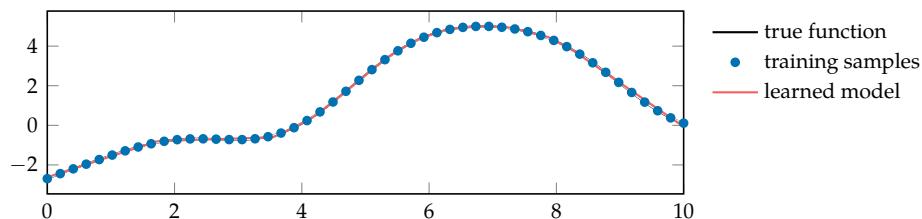


Figure C.2. A more compact depiction of figure C.1. Neural network layers are often represented as blocks or slices for simplicity.

Figure C.3. A deep neural network fit to samples from a nonlinear function so as to minimize the squared error. This neural network has four affine layers, with 10 neurons in each intermediate representation.

There are many types of activation functions that are commonly used. Similar to their biological inspiration, they tend to be close to zero when their input is low and large when their input is high. Some common activation functions are shown in figure C.5.

Sometimes special layers are incorporated to achieve certain effects. For example, in figure C.4, we used a *softmax* layer at the end to force the output to represent a two-element categorical distribution. The softmax function applies the exponential function to each element, which ensures that they are positive

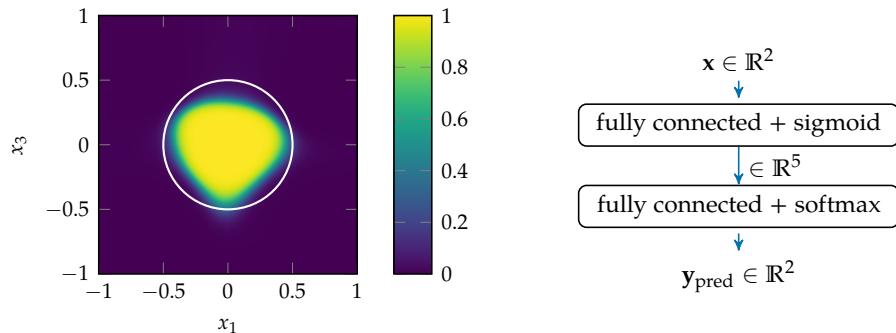


Figure C.4. A simple, two-layer, fully connected network trained to classify whether a given coordinate lies within a circle (shown in white). The nonlinearities allow neural networks to form complicated, nonlinear decision boundaries.

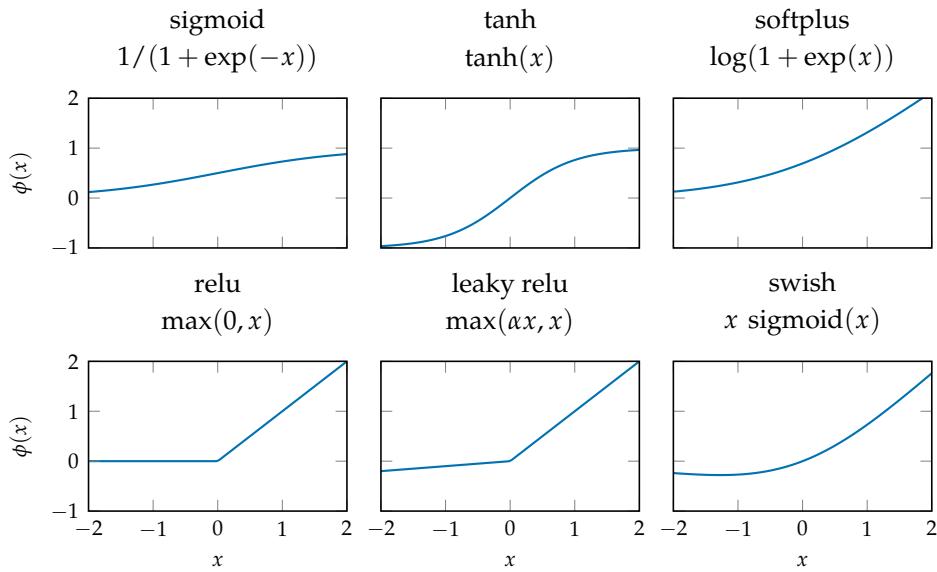


Figure C.5. Several common activation functions.

and then renormalizes the resulting values:

$$\text{softmax}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \quad (\text{C.4})$$

The outputs of the softmax function can be interpreted as probabilities (section 12.2.1).

Gradients for neural networks are typically computed using *reverse accumulation*.⁵ The method begins with a forward step, in which the neural network is evaluated using all input parameters. In the backward step, the gradient of each term of interest is computed working from the output back to the input. Reverse accumulation uses the chain rule for derivatives:

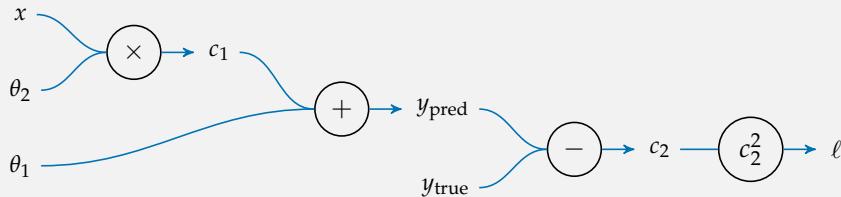
$$\frac{\partial f(g(h(x)))}{\partial x} = \frac{\partial f(g(h))}{\partial h} \frac{\partial h(x)}{\partial x} = \left(\frac{\partial f(g)}{\partial g} \frac{\partial g(h)}{\partial h} \right) \frac{\partial h(x)}{\partial x} \quad (\text{C.5})$$

Example C.2 demonstrates this process. Many deep learning packages compute gradients using such automatic differentiation techniques.⁶ Users rarely have to provide their own gradients.

⁵ This process is commonly called *backpropagation*, which specifically refers to reverse accumulation applied to a scalar loss function. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning Representations by Back-Propagating Errors,” *Nature*, vol. 323, pp. 533–536, 1986.

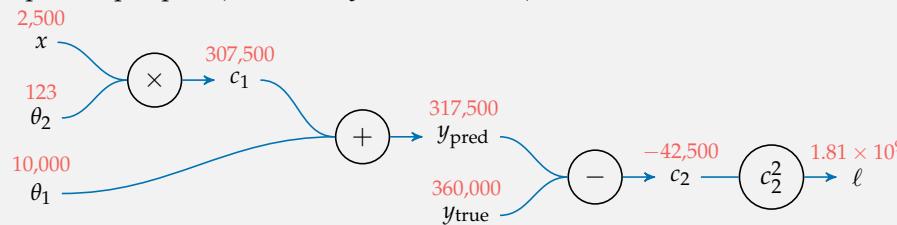
⁶ A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed. SIAM, 2008.

Recall the neural network and loss function from example C.1. Here we have drawn the computational graph for the loss calculation:

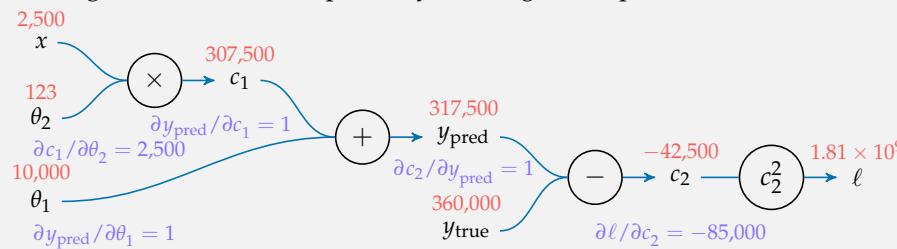


Example C.2. How reverse accumulation is used to compute parameter gradients given training data.

Reverse accumulation begins with a forward pass, in which the computational graph is evaluated. We will again use $\theta = [10,000, 123]$ and the input-output pair ($x = 2,500$, $y_{\text{true}} = 360,000$) as follows:



The gradient is then computed by working back up the tree:



Finally, we compute:

$$\frac{\partial \ell}{\partial \theta_1} = \frac{\partial \ell}{\partial c_2} \frac{\partial c_2}{\partial y_{\text{pred}}} \frac{\partial y_{\text{pred}}}{\partial \theta_1} = -85,000 \cdot 1 \cdot 1 = -85,000$$

$$\frac{\partial \ell}{\partial \theta_2} = \frac{\partial \ell}{\partial c_2} \frac{\partial c_2}{\partial y_{\text{pred}}} \frac{\partial y_{\text{pred}}}{\partial c_1} \frac{\partial c_1}{\partial \theta_2} = -85,000 \cdot 1 \cdot 1 \cdot 2500 = -2.125 \times 10^8$$

D Julia

Julia is a scientific programming language that is free and open source.¹ It is a relatively new language that borrows inspiration from languages like Python, MATLAB, and R. It was selected for use in this book because it is sufficiently high level² so that the algorithms can be compactly expressed and readable while also being fast. This book is compatible with Julia version 1.11. This appendix introduces the concepts necessary for understanding the included code, omitting many of the advanced features of the language.

D.1 Types

Julia has a variety of basic types that can represent data given as truth values, numbers, strings, arrays, tuples, and dictionaries. Users can also define their own types. This section explains how to use some of the basic types and how to define new types.

D.1.1 Booleans

The *Boolean* type in Julia, written as `Bool`, includes the values `true` and `false`. We can assign these values to variables. Variable names can be any string of characters, including Unicode, with a few restrictions.

```
α = true  
done = false
```

The variable name appears on the left side of the equal sign; the value that variable is to be assigned is on the right side.

¹ Julia may be obtained from <http://julialang.org>.

² In contrast with languages like C++, Julia does not require programmers to worry about memory management and other lower-level details, yet it allows low-level control when needed.

We can make assignments in the Julia console. The console, or REPL (for read, eval, print, loop), will return a response to the expression being evaluated. The `#` symbol indicates that the rest of the line is a comment.

```
julia> x = true
true
julia> y = false; # semicolon suppresses the console output
julia> typeof(x)
Bool
julia> x == y # test for equality
false
```

The standard Boolean operators are supported:

```
julia> !x      # not
false
julia> x && y # and
false
julia> x || y # or
true
```

D.1.2 Numbers

Julia supports integer and floating-point numbers, as shown here:

```
julia> typeof(42)
Int64
julia> typeof(42.0)
Float64
```

Here, `Int64` denotes a 64-bit integer, and `Float64` denotes a 64-bit floating-point value.³ We can perform the standard mathematical operations:

```
julia> x = 4
4
julia> y = 2
2
julia> x + y
6
julia> x - y
2
julia> x * y
8
julia> x / y
2.0
```

³ On 32-bit machines, an integer literal like `42` is interpreted as an `Int32`.

```
julia> x ^ y # exponentiation
16
julia> x % y # remainder from division
0
julia> div(x, y) # truncated division returns an integer
2
```

Note that the result of `x / y` is a `Float64`, even when `x` and `y` are integers. We can also perform these operations at the same time as an assignment. For example, `x += 1` is shorthand for `x = x + 1`.

We can also make comparisons:

```
julia> 3 > 4
false
julia> 3 >= 4
false
julia> 3 ≥ 4    # unicode also works, use \ge[tab] in console
false
julia> 3 < 4
true
julia> 3 <= 4
true
julia> 3 ≤ 4    # unicode also works, use \le[tab] in console
true
julia> 3 == 4
false
julia> 3 < 4 < 5
true
```

D.1.3 Strings

A *string* is an array of characters. Strings are not used very much in this textbook except to report certain errors. An object of type `String` can be constructed using `"` characters. For example:

```
julia> x = "optimal"
"optimal"
julia> typeof(x)
String
```

D.1.4 Symbols

A *symbol* represents an identifier. It can be written using the : operator or constructed from strings:

```
julia> :A
:A
julia> :Battery
:Battery
julia> Symbol("Failure")
:Failure
```

D.1.5 Vectors

A *vector* is a one-dimensional array that stores a sequence of values. We can construct a vector using square brackets, separating elements by commas:

```
julia> x = [];
           # empty vector
julia> x = trues(3);
           # Boolean vector containing three trues
julia> x = ones(3);
           # vector of three ones
julia> x = zeros(3);
           # vector of three zeros
julia> x = rand(3);
           # vector of three random numbers between 0 and 1
julia> x = [3, 1, 4];
           # vector of integers
julia> x = [3.1415, 1.618, 2.7182];
           # vector of floats
```

An *array comprehension* can be used to create vectors:

```
julia> [sin(x) for x in 1:5]
5-element Vector{Float64}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
 -0.7568024953079282
 -0.9589242746631385
```

We can inspect the type of a vector:

```
julia> typeof([3, 1, 4])
           # 1-dimensional array of Int64s
Vector{Int64} (alias for Array{Int64, 1})
julia> typeof([3.1415, 1.618, 2.7182])
           # 1-dimensional array of Float64s
Vector{Float64} (alias for Array{Float64, 1})
julia> Vector{Float64} # alias for a 1-dimensional array
Vector{Float64} (alias for Array{Float64, 1})
```

We index into vectors using square brackets:

```
julia> x[1]      # first element is indexed by 1
3.1415
julia> x[3]      # third element
2.7182
julia> x[end]    # use end to reference the end of the array
2.7182
julia> x[end-1]  # this returns the second to last element
1.618
```

We can pull out a range of elements from an array. Ranges are specified using a colon notation:

```
julia> x = [1, 2, 5, 3, 1]
5-element Vector{Int64}:
1
2
5
3
1
julia> x[1:3]      # pull out the first three elements
3-element Vector{Int64}:
1
2
5
julia> x[1:2:end]  # pull out every other element
3-element Vector{Int64}:
1
5
1
julia> x[end:-1:1] # pull out all the elements in reverse order
5-element Vector{Int64}:
1
3
5
2
1
```

We can perform a variety of operations on arrays. The exclamation mark at the end of function names is used to indicate that the function *mutates* (i.e., changes) the input:

```
julia> length(x)
5
julia> [x, x]          # concatenation
2-element Vector{Vector{Int64}}:
```

```
[1, 2, 5, 3, 1]
[1, 2, 5, 3, 1]
julia> push!(x, -1)      # add an element to the end
6-element Vector{Int64}:
1
2
5
3
1
-1
julia> pop!(x)          # remove an element from the end
-1
julia> append!(x, [2, 3]) # append [2, 3] to the end of x
7-element Vector{Int64}:
1
2
5
3
1
2
3
julia> sort!(x)          # sort the elements, altering the same vector
7-element Vector{Int64}:
1
1
2
2
3
3
5
julia> sort(x);          # sort the elements as a new vector
julia> x[1] = 2; print(x) # change the first element to 2
[2, 1, 2, 2, 3, 3, 5]
julia> x = [1, 2];
julia> y = [3, 4];
julia> x + y            # add vectors
2-element Vector{Int64}:
4
6
julia> 3x - [1, 2]       # multiply by a scalar and subtract
2-element Vector{Int64}:
2
4
julia> using LinearAlgebra
```

```
julia> dot(x, y)          # dot product available after using LinearAlgebra
11
julia> x·y              # dot product using unicode character, use \cdot[tab] in console
11
julia> prod(y)          # product of all the elements in y
12
```

It is often useful to apply various functions elementwise to vectors. This is a form of *broadcasting*. With infix operators (e.g., `+`, `*`, and `^`), a dot is prefixed to indicate elementwise broadcasting. With functions like `sqrt` and `sin`, the dot is postfix:

```
julia> x .* y    # elementwise multiplication
2-element Vector{Int64}:
 3
 8
julia> x .^ 2    # elementwise squaring
2-element Vector{Int64}:
 1
 4
julia> sin.(x)  # elementwise application of sin
2-element Vector{Float64}:
 0.8414709848078965
 0.9092974268256817
julia> sqrt.(x) # elementwise application of sqrt
2-element Vector{Float64}:
 1.0
 1.4142135623730951
```

D.1.6 Matrices

A *matrix* is a two-dimensional array. Like a vector, it is constructed using square brackets. We use spaces to delimit elements in the same row and semicolons to delimit rows. We can also index into the matrix and output submatrices using ranges:

```
julia> X = [1 2 3; 4 5 6; 7 8 9; 10 11 12];
julia> typeof(X)  # a 2-dimensional array of Int64s
Matrix{Int64} (alias for Array{Int64, 2})
julia> X[2]        # second element using column-major ordering
4
julia> X[3,2]      # element in third row and second column
8
```

```
julia> X[1,:]      # extract the first row
3-element Vector{Int64}:
1
2
3
julia> X[:,2]      # extract the second column
4-element Vector{Int64}:
2
5
8
11
julia> X[:,1:2]    # extract the first two columns
4x2 Matrix{Int64}:
1  2
4  5
7  8
10 11
julia> X[1:2,1:2] # extract a 2x2 submatrix from the top left of x
2x2 Matrix{Int64}:
1  2
4  5
julia> Matrix{Float64} # alias for a 2-dimensional array
Matrix{Float64} (alias for Array{Float64, 2})
```

We can also construct a variety of special matrices and use array comprehensions:

```
julia> Matrix(1.0I, 3, 3)                      # 3x3 identity matrix
3x3 Matrix{Float64}:
1.0  0.0  0.0
0.0  1.0  0.0
0.0  0.0  1.0
julia> Matrix(Diagonal([3, 2, 1]))            # 3x3 diagonal matrix with 3, 2, 1 on diagonal
3x3 Matrix{Int64}:
3  0  0
0  2  0
0  0  1
julia> zeros(3,2)                             # 3x2 matrix of zeros
3x2 Matrix{Float64}:
0.0  0.0
0.0  0.0
0.0  0.0
julia> rand(3,2)                               # 3x2 random matrix
3x2 Matrix{Float64}:
0.714382  0.164401
```

```

0.19441  0.12398
0.610099 0.616684
julia> [sin(x + y) for x in 1:3, y in 1:2] # array comprehension
3x2 Matrix{Float64}:
 0.909297  0.14112
 0.14112   -0.756802
 -0.756802 -0.958924

```

Matrix operations include the following:

```

julia> X'          # complex conjugate transpose
3x4 adjoint(::Matrix{Int64}) with eltype Int64:
 1  4  7  10
 2  5  8  11
 3  6  9  12
julia> 3X .+ 2      # multiplying by scalar and adding scalar
4x3 Matrix{Int64}:
 5   8   11
 14  17  20
 23  26  29
 32  35  38
julia> X = [1 3; 3 1]; # create an invertible matrix
julia> inv(X)        # inversion
2x2 Matrix{Float64}:
 -0.125  0.375
 0.375  -0.125
julia> pinv(X)       # pseudoinverse (requires LinearAlgebra)
2x2 Matrix{Float64}:
 -0.125  0.375
 0.375  -0.125
julia> det(X)         # determinant (requires LinearAlgebra)
-8.0
julia> [X X]          # horizontal concatenation, same as hcat(X, X)
2x4 Matrix{Int64}:
 1  3  1  3
 3  1  3  1
julia> [X; X]          # vertical concatenation, same as vcat(X, X)
4x2 Matrix{Int64}:
 1  3
 3  1
 1  3
 3  1
julia> sin.(X)         # elementwise application of sin
2x2 Matrix{Float64}:
 0.841471  0.14112

```

```

0.14112  0.841471
julia> map(sin, X)      # elementwise application of sin
2×2 Matrix{Float64}:
 0.841471  0.14112
 0.14112   0.841471
julia> vec(X)          # reshape an array as a vector
4-element Vector{Int64}:
 1
 3
 3
 1

```

D.1.7 Tuples

A *tuple* is an ordered list of values, potentially of different types. They are constructed with parentheses. They are similar to vectors, but they cannot be mutated:

```

julia> x = ()    # the empty tuple
()
julia> isempty(x)
true
julia> x = (1,) # tuples of one element need the trailing comma
(1,)
julia> typeof(x)
Tuple{Int64}
julia> x = (1, 0, [1, 2], 2.5029, 4.6692) # third element is a vector
(1, 0, [1, 2], 2.5029, 4.6692)
julia> typeof(x)
Tuple{Int64, Int64, Vector{Int64}, Float64, Float64}
julia> x[2]
0
julia> x[end]
4.6692
julia> x[4:end]
(2.5029, 4.6692)
julia> length(x)
5
julia> x = (1, 2)
(1, 2)
julia> a, b = x;
julia> a
1
julia> b
2

```

D.1.8 Named Tuples

A *named tuple* is like a tuple, but each entry has its own name:

```
julia> x = (a=1, b=-Inf)
(a = 1, b = -Inf)
julia> x isa NamedTuple
true
julia> x.a
1
julia> a, b = x;
julia> a
1
julia> (; :a=>10)
(a = 10,)
julia> (; :a=>10, :b=>11)
(a = 10, b = 11)
julia> merge(x, (d=3, e=10)) # merge two named tuples
(a = 1, b = -Inf, d = 3, e = 10)
```

D.1.9 Dictionaries

A *dictionary* is a collection of key-value pairs. Key-value pairs are indicated with a double arrow operator `=>`. We can index into a dictionary using square brackets, just as with arrays and tuples:

```
julia> x = Dict(); # empty dictionary
julia> x[3] = 4 # associate key 3 with value 4
4
julia> x = Dict(3=>4, 5=>1) # create a dictionary with two key-value pairs
Dict{Int64, Int64} with 2 entries:
  5 => 1
  3 => 4
julia> x[5]           # return the value associated with key 5
1
julia> haskey(x, 3) # check whether dictionary has key 3
true
julia> haskey(x, 4) # check whether dictionary has key 4
false
```

D.1.10 Composite Types

A *composite type* is a collection of named fields. By default, an instance of a composite type is immutable (i.e., it cannot change). We use the `struct` keyword and then give the new type a name and list the names of the fields:

```
struct A
    a
    b
end
```

Adding the keyword `mutable` makes it so that an instance can change:

```
mutable struct B
    a
    b
end
```

Composite types are constructed using parentheses, between which we pass in values for each field:

```
x = A(1.414, 1.732)
```

The double-colon operator can be used to specify the type for any field:

```
struct A
    a::Int64
    b::Float64
end
```

These type annotations require that we pass in an `Int64` for the first field and a `Float64` for the second field. For compactness, this book does not use type annotations, but it is at the expense of performance. Type annotations allow Julia to improve runtime performance because the compiler can optimize the underlying code for specific types.

D.1.11 Abstract Types

So far we have discussed *concrete types*, which are types that we can construct. However, concrete types are only part of the type hierarchy. There are also *abstract types*, which are supertypes of concrete types and other abstract types.

We can explore the type hierarchy of the `Float64` type shown in figure D.1 using the `supertype` and `subtypes` functions:

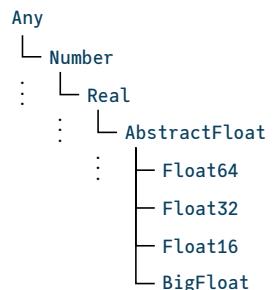


Figure D.1. The type hierarchy for the `Float64` type.

```
julia> supertype(Float64)
AbstractFloat
julia> supertype(AbstractFloat)
Real
julia> supertype(Real)
Number
julia> supertype(Number)
Any
julia> supertype(Any)           # Any is at the top of the hierarchy
Any
julia> using InteractiveUtils # required for using subtypes in scripts
julia> subtypes(AbstractFloat) # different types of AbstractFloats
4-element Vector{Any}:
BigFloat
Float16
Float32
Float64
julia> subtypes(Float64)       # Float64 does not have any subtypes
Type[]
```

We can define our own abstract types:

```
abstract type C end
abstract type D <: C end # D is an abstract subtype of C
struct E <: D # E is a composite type that is a subtype of D
    a
end
```

D.1.12 Parametric Types

Julia supports *parametric types*, which are types that take parameters. The parameters to a parametric type are given within braces and delimited by commas. We have already seen a parametric type with our dictionary example:

```
julia> x = Dict(3⇒1.4, 1⇒5.9)
Dict{Int64, Float64} with 2 entries:
 3 => 1.4
 1 => 5.9
```

For dictionaries, the first parameter specifies the key type, and the second parameter specifies the value type. The example has `Int64` keys and `Float64` values, making the dictionary of type `Dict{Int64,Float64}`. Julia was able to infer these types based on the input, but we could have specified it explicitly:

```
julia> x = Dict{Int64,Float64}(3⇒1.4, 1⇒5.9);
```

While it is possible to define our own parametric types, we do not need to do so in this text.

D.2 Functions

A *function* maps its arguments, given as a tuple, to a result that is returned.

D.2.1 Named Functions

One way to define a *named function* is to use the `function` keyword, followed by the name of the function and a tuple of names of arguments:

```
function f(x, y)
    return x + y
end
```

We can also define functions compactly using assignment form:

```
julia> f(x, y) = x + y;
julia> f(3, 0.1415)
3.1415
```

D.2.2 Anonymous Functions

An *anonymous function* is not given a name, though it can be assigned to a named variable. One way to define an anonymous function is to use the arrow operator:

```
julia> h = x → x^2 + 1 # assign anonymous function with input x to a variable h
#1 (generic function with 1 method)
julia> h(3)
10
julia> g(f, a, b) = [f(a), f(b)]; # applies function f to a and b and returns array
julia> g(h, 5, 10)
2-element Vector{Int64}:
 26
 101
julia> g(x→sin(x)+1, 10, 20)
2-element Vector{Float64}:
 0.4559788891106302
 1.9129452507276277
```

D.2.3 Callable Objects

We can define a type and associate functions with it, allowing objects of that type to be *callable*:

```
julia> (x::A)() = x.a + x.b    # adding a zero-argument function to the type A defined earlier
julia> (x::A)(y) = y*x.a + x.b # adding a single-argument function
julia> x = A(22, 8);
julia> x()
30
julia> x(2)
52
```

D.2.4 Optional Arguments

We can assign a default value to an argument, making the specification of that argument optional:

```
julia> f(x=10) = x^2;
julia> f()
100
julia> f(3)
9
julia> f(x, y, z=1) = x*y + z;
julia> f(1, 2, 3)
5
julia> f(1, 2)
3
```

D.2.5 Keyword Arguments

Functions may use *keyword arguments*, which are arguments that are named when the function is called. Keyword arguments are given after all the positional arguments. A semicolon is placed before any keywords, separating them from the other arguments:

```
julia> f(; x = 0) = x + 1;
julia> f()
1
julia> f(x = 10)
11
julia> f(x, y = 10; z = 2) = (x + y)*z;
julia> f(1)
```

```

22
julia> f(2, z = 3)
36
julia> f(2, 3)
10
julia> f(2, 3, z = 1)
5

```

D.2.6 Dispatch

The types of the arguments passed to a function can be specified using the double colon operator. If multiple methods of the same function are provided, Julia will execute the appropriate method. The mechanism for choosing which method to execute is called *dispatch*:

```

julia> f(x::Int64) = x + 10;
julia> f(x::Float64) = x + 3.1415;
julia> f(1)
11
julia> f(1.0)
4.141500000000001
julia> f(1.3)
4.441500000000004

```

The method with a type signature that best matches the types of the arguments given will be used:

```

julia> f(x) = 5;
julia> f(x::Float64) = 3.1415;
julia> f([3, 2, 1])
5
julia> f(0.00787499699)
3.1415

```

D.2.7 Splatting

It is often useful to *splat* the elements of a vector or a tuple into the arguments to a function using the ... operator:

```
julia> f(x,y,z) = x + y - z;
julia> a = [3, 1, 2];
julia> f(a...)
2
julia> b = (2, 2, 0);
julia> f(b...)
4
julia> c = ([0,0],[1,1]);
julia> f([2,2], c...)
2-element Vector{Int64}:
 1
 1
```

D.3 Control Flow

We can control the flow of our programs using conditional evaluation and loops. This section provides some of the syntax used in the book.

D.3.1 Conditional Evaluation

Conditional evaluation will check the value of a Boolean expression and then evaluate the appropriate block of code. One of the most common ways to do this is with an `if` statement:

```
if x < y
    # run this if x < y
elseif x > y
    # run this if x > y
else
    # run this if x == y
end
```

We can also use the *ternary operator* with its question mark and colon syntax. It checks the Boolean expression before the question mark. If the expression evaluates to true, then it returns what comes before the colon; otherwise, it returns what comes after the colon:

```
julia> f(x) = x > 0 ? x : 0;
julia> f(-10)
0
julia> f(10)
10
```

D.3.2 Loops

A *loop* allows for repeated evaluation of expressions. One type of loop is the while loop, which repeatedly evaluates a block of expressions until the specified condition after the `while` keyword is met. The following example sums the values in the array `X`:

```
X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
while !isempty(X)
    s += pop!(X)
end
```

Another type of loop is the for loop, which uses the `for` keyword. The following example will also sum over the values in the array `X` but will not modify `X`:

```
X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
for y in X
    s += y
end
```

The `in` keyword can be replaced by `=` or `∈`. The following code block is equivalent:

```
X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
for i = 1:length(X)
    s += X[i]
end
```

D.3.3 Iterators

We can iterate over collections in contexts such as for loops and array comprehensions. To demonstrate various iterators, we will use the `collect` function, which returns an array of all items generated by an iterator:

```
julia> X = ["feed", "sing", "ignore"];
julia> collect(enumerate(X)) # return the count and the element
3-element Vector{Tuple{Int64, String}}:
 (1, "feed")
 (2, "sing")
 (3, "ignore")
julia> collect(eachindex(X)) # equivalent to 1:length(X)
3-element Vector{Int64}:
```

```

1
2
3
julia> Y = [-5, -0.5, 0];
julia> collect(zip(X, Y))    # iterate over multiple iterators simultaneously
3-element Vector{Tuple{String, Float64}}:
 ("feed", -5.0)
 ("sing", -0.5)
 ("ignore", 0.0)
julia> import IterTools: subsets
julia> collect(subsets(X))  # iterate over all subsets
8-element Vector{Vector{String}}:
 []
 ["feed"]
 ["sing"]
 ["feed", "sing"]
 ["ignore"]
 ["feed", "ignore"]
 ["sing", "ignore"]
 ["feed", "sing", "ignore"]
julia> collect(eachindex(X)) # iterate over indices into a collection
3-element Vector{Int64}:
 1
 2
 3
julia> Z = [1 2; 3 4; 5 6];
julia> import Base.Iterators: product
julia> collect(product(X,Y)) # iterate over Cartesian product of multiple iterators
3x3 Matrix{Tuple{String, Float64}}:
 ("feed", -5.0)   ("feed", -0.5)   ("feed", 0.0)
 ("sing", -5.0)   ("sing", -0.5)   ("sing", 0.0)
 ("ignore", -5.0)  ("ignore", -0.5)  ("ignore", 0.0)

```

D.4 Packages

A *package* is a collection of Julia code and possibly other external libraries that can be imported to provide additional functionality. This section briefly reviews a few of the key packages that we build upon in this book. To add a registered package like `Distributions.jl`, we can run

```
using Pkg
Pkg.add("Distributions")
```

To update packages, we use

```
Pkg.update()
```

To use a package, we use the keyword `using` as follows:

```
using Distributions
```

D.4.1 *Distributions.jl*

We use the `Distributions.jl` package (version 0.25) to represent, fit, and sample from probability distributions:

```
julia> using Distributions
julia> dist = Categorical([0.3, 0.5, 0.2]) # create a categorical distribution
Distributions.Categorical{Float64, Vector{Float64}}(support=Base.OneTo(3), p=[0.3, 0.5, 0.2])
julia> data = rand(dist) # generate a sample
3
julia> data = rand(dist, 2) # generate two samples
2-element Vector{Int64}:
 2
 3
julia> μ, σ = 5.0, 2.5; # define parameters of a normal distribution
julia> dist = Normal(μ, σ) # create a normal distribution
Distributions.Normal{Float64}(μ=5.0, σ=2.5)
julia> rand(dist) # sample from the distribution
5.301718264898578
julia> data = rand(dist, 3) # generate three samples
3-element Vector{Float64}:
 7.8074946194990815
 4.40160104223275
 4.0094530628735345
julia> data = rand(dist, 1000); # generate many samples
julia> Distributions.fit(Normal, data) # fit a normal distribution to the samples
Distributions.Normal{Float64}(μ=4.995186963615839, σ=2.5486524236118573)
julia> μ = [1.0, 2.0];
julia> Σ = [1.0 0.5; 0.5 2.0];
julia> dist = MvNormal(μ, Σ) # create a multivariate normal distribution
FullNormal(
dim: 2
μ: [1.0, 2.0]
Σ: [1.0 0.5; 0.5 2.0]
)
julia> rand(dist, 3) # generate three samples
2×3 Matrix{Float64}:
 -0.0987709 -0.282818  0.472134
  0.910944   4.77778   1.082
```

```
julia> dist = Dirichlet(ones(3))      # create a Dirichlet distribution Dir(1,1,1)
Distributions.Dirichlet{Float64, Vector{Float64}, Float64}(alpha=[1.0, 1.0, 1.0])
julia> rand(dist)                  # sample from the distribution
3-element Vector{Float64}:
 0.2973383158106437
 0.3034604891594059
 0.3992011950299505
```

D.4.2 JuMP.jl

We use the `JuMP.jl` package (version 1.23) to specify optimization problems that we can then solve using a variety of solvers, such as those included in `GLPK.jl` and `Ipopt.jl`:

```
julia> using JuMP
julia> using GLPK
julia> model = Model(GLPK.Optimizer)      # create model and use GLPK as solver
A JuMP Model
|- solver: GLPK
|- objective_sense: FEASIBILITY_SENSE
|- num_variables: 0
|- num_constraints: 0
└ Names registered in the model: none
julia> @variable(model, x[1:3])          # define variables x[1], x[2], and x[3]
3-element Vector{JuMP.VariableRef}:
x[1]
x[2]
x[3]
julia> @objective(model, Max, sum(x) - x[2]) # define maximization objective
x[1] + 0 x[2] + x[3]
julia> @constraint(model, x[1] + x[2] ≤ 3)   # add constraint
x[1] + x[2] ≤ 3
julia> @constraint(model, x[2] + x[3] ≤ 2)   # add another constraint
x[2] + x[3] ≤ 2
julia> @constraint(model, x[2] ≥ 0)           # add another constraint
x[2] ≥ 0
julia> optimize!(model)                   # solve
julia> value.(x)                        # extract optimal values for elements in x
3-element Vector{Float64}:
 3.0
 0.0
 2.0
```

D.4.3 Optim.jl

We use the `Optim.jl` package (version 1.9) to solve unconstrained optimization problems using a variety of techniques:

```
julia> using Optim
julia> f(x) = x[1]^4+x[1]^2-x[1]+x[2]^2-20*x[1]^2*x[2]^2; # function to minimize
julia> x₀ = [0.0, 0.0]; # initial guess
julia> result = optimize(f, x₀, LBFGS()); # minimize f starting at x₀ using LBFGS
* Status: success
* Candidate solution
  Final objective value: -2.148047e-01
* Found with
  Algorithm: L-BFGS
* Convergence measures
  |x - x'| = 1.94e-04 ⪻ 0.0e+00
  |x - x'|/|x'| = 5.04e-04 ⪻ 0.0e+00
  |f(x) - f(x')| = 7.13e-08 ⪻ 0.0e+00
  |f(x) - f(x')|/|f(x')| = 3.32e-07 ⪻ 0.0e+00
  |g(x)| = 5.12e-09 ⪻ 1.0e-08
* Work counters
  Seconds run: 0 (vs limit Inf)
  Iterations: 4
  f(x) calls: 9
  ∇f(x) calls: 9
julia> result.minimizer # extract the optimal value for x
2-element Vector{Float64}:
 0.3854584971606701
 0.0
julia> result.minimum # extract the optimal value of the function
-0.21480474685286194
```

D.4.4 SimpleWeightedGraphs.jl

We extend the `SimpleWeightedGraphs.jl` package (version 1.4) to represent graphs with weighted edges for discrete reachability analysis in chapter 10. Specifically, we extend the package to create a `WeightedGraph` type that can represent a graph with weighted edges between states. The code for this extension is provided in the ancillaries for this book, and the following code demonstrates its usage:

```
julia> using SimpleWeightedGraphs
julia> states = [:s1, :s2, :s3]; # define states
julia> g = WeightedGraph(states); # weighted graph with a node for each state
```

```
julia> add_edge!(g, :s1, :s2, 1.0); # add an edge from s1 to s2 with weight 1.0
julia> get_weight(g, :s1, :s2)      # get the weight of the edge from s1 to s2
1.0
julia> inneighbors(g, :s2)         # get the nodes with an edge pointing to s2
1-element Vector{Symbol}:
:s1
julia> outneighbors(g, :s1)        # get the nodes with an edge starting at s1
1-element Vector{Symbol}:
:s2
```

D.5 Convenience Functions

We define `SetCategorical` to represent distributions over discrete sets:

```
struct SetCategorical{S}
    elements::Vector{S} # Set elements (could be repeated)
    distr::Categorical # Categorical distribution over set elements

    function SetCategorical(elements::AbstractVector{S}) where S
        weights = ones(length(elements))
        return new{S}(elements, Categorical(normalize(weights, 1)))
    end

    function SetCategorical(
        elements::AbstractVector{S},
        weights::AbstractVector{Float64}
    ) where S
        ℓ₁ = norm(weights, 1)
        if ℓ₁ < 1e-6 || isinf(ℓ₁)
            return SetCategorical(elements)
        end
        distr = Categorical(normalize(weights, 1))
        return new{S}(elements, distr)
    end
end

Distributions.rand(D::SetCategorical) = D.elements[rand(D.distr)]
Distributions.rand(D::SetCategorical, n::Int) = D.elements[rand(D.distr, n)]
function Distributions.pdf(D::SetCategorical, x)
    sum(e == x ? w : 0.0 for (e,w) in zip(D.elements, D.distr.p))
end
```

```
julia> D = SetCategorical(["up", "down", "left", "right"],[0.4, 0.2, 0.3, 0.1]);  
julia> rand(D)  
"left"  
julia> rand(D, 5)  
5-element Vector{String}:  
"down"  
"left"  
"left"  
"left"  
"down"  
julia> pdf(D, "up")  
0.3999999999999999
```

References

1. P. Abbeel and A. Y. Ng, "Apprenticeship Learning via Inverse Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, 2004 (cit. on p. 40).
2. H. Abdi and L. J. Williams, "Principal Component Analysis," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 4, pp. 433–459, 2010 (cit. on p. 222).
3. A. Ahmadi-Javid, "Entropic Value-At-Risk: A New Coherent Risk Measure," *Journal of Optimization Theory and Applications*, vol. 155, no. 3, pp. 1105–1123, 2011 (cit. on p. 64).
4. M. Akintunde, A. Lomuscio, L. Maganti, and E. Pirovano, "Reachability Analysis for Neural Agent-Environment Systems," in *International Conference on Principles of Knowledge Representation and Reasoning*, 2018 (cit. on p. 257).
5. M. Althoff, "Reachability Analysis of Nonlinear Systems Using Conservative Polynomialization and Non-Convex Sets," in *International Conference on Hybrid Systems: Computation and Control*, 2013 (cit. on p. 240).
6. M. Althoff and G. Frehse, "Combining Zonotopes and Support Functions for Efficient Reachability Analysis of Linear Systems," in *IEEE Conference on Decision and Control (CDC)*, 2016 (cit. on p. 219).
7. M. Althoff, G. Frehse, and A. Girard, "Set Propagation Techniques for Reachability Analysis," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 4, pp. 369–395, 2021 (cit. on p. 212).
8. M. Althoff, G. Frehse, and A. Girard, "Set Propagation Techniques for Reachability Analysis," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 4, pp. 369–395, 2021 (cit. on p. 231).
9. M. Althoff, O. Stursberg, and M. Buss, "Reachability Analysis of Nonlinear Systems with Uncertain Parameters Using Conservative Linearization," in *IEEE Conference on Decision and Control (CDC)*, 2008 (cit. on pp. 240, 241).
10. A. N. Angelopoulos, S. Bates, et al., "Conformal Prediction: A Gentle Introduction," *Foundations and Trends in Machine Learning*, vol. 16, no. 4, pp. 494–591, 2023 (cit. on pp. 339, 340).

11. T. L. Arel, *Safety Management System Manual*, Air Traffic Organization, Federal Aviation Administration, 2022 (cit. on p. 13).
12. D. Ariely, *Predictably Irrational: The Hidden Forces That Shape Our Decisions*. Harper, 2008 (cit. on p. 41).
13. M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A Tutorial on Particle Filters for Online Nonlinear/non-Gaussian Bayesian Tracking," *IEEE Transactions on Signal Processing*, vol. 50, no. 2, pp. 174–188, 2002 (cit. on p. 177).
14. T. W. Athan and P. Y. Papalambros, "A Note on Weighted Criteria Methods for Compromise Solutions in Multi-Objective Optimization," *Engineering Optimization*, vol. 27, no. 2, pp. 155–176, 1996 (cit. on p. 68).
15. D. Baehrens, T. Schroeter, S. Harmeling, M. Kawanabe, K. Hansen, and K.-R. Müller, "How to Explain Individual Classification Decisions," *Journal of Machine Learning Research*, vol. 11, pp. 1803–1831, 2010 (cit. on p. 300).
16. C. Baier and J.-P. Katoen, "Principles of Model Checking," in MIT Press, 2008, ch. 1 (cit. on p. 2).
17. C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008 (cit. on pp. 75, 84).
18. G. Barthe, J.-P. Katoen, and A. Silva, *Foundations of Probabilistic Programming*. Cambridge University Press, 2020 (cit. on p. 152).
19. A. Bauer, M. Leucker, and C. Schallhart, "Runtime Verification for LTL and TLTL," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, pp. 1–64, 2011 (cit. on p. 349).
20. R. Bhattacharyya, S. Jung, L. Kruse, R. Senanayake, and M. J. Kochenderfer, "A Hybrid Rule-Based and Data-Driven Approach to Driver Modeling Through Particle Filtering," *IEEE Transactions on Intelligent Transportation Systems*, no. 2108.12820, 2021 (cit. on p. 48).
21. A. F. Bielajew, "History of Monte Carlo," in *Monte Carlo Techniques in Radiation Therapy*, CRC Press, 2021, pp. 3–15 (cit. on p. 4).
22. A. Biere, *Handbook of Satisfiability*. IOS Press, 2009, vol. 185 (cit. on pp. 270, 272).
23. C. M. Bishop and H. Bishop, *Deep Learning: Foundations and Concepts*. Springer Nature, 2023 (cit. on p. 28).
24. A. T. Borchers, F. Hagie, C. L. Keen, and M. E. Gershwin, "The History and Contemporary Challenges of the US Food and Drug Administration," *Clinical Therapeutics*, vol. 29, no. 1, pp. 1–16, 2007 (cit. on p. 5).
25. M. Bouton, J. Tumova, and M. J. Kochenderfer, "Point-Based Methods for Model Checking in Partially Observable Markov Decision Processes," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2020 (cit. on p. 84).

26. G. E. Box, "Science and Statistics," *Journal of the American Statistical Association*, vol. 71, no. 356, pp. 791–799, 1976 (cit. on p. 20).
27. S. P. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004 (cit. on p. 224).
28. C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfsagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012 (cit. on p. 126).
29. C. F. Camerer, *Behavioral Game Theory: Experiments in Strategic Interaction*. Princeton University Press, 2003 (cit. on p. 41).
30. O. Cappé, A. Guillin, J.-M. Marin, and C. P. Robert, "Population Monte Carlo," *Journal of Computational and Graphical Statistics*, vol. 13, no. 4, pp. 907–929, 2004 (cit. on p. 175).
31. G. Casella and E. I. George, "Explaining the Gibbs Sampler," *The American Statistician*, vol. 46, no. 3, pp. 167–174, 1992 (cit. on p. 152).
32. F. Cérou and A. Guyader, "Adaptive Multilevel Splitting for Rare Event Analysis," *Stochastic Analysis and Applications*, vol. 25, no. 2, pp. 417–443, 2007 (cit. on p. 191).
33. P. Chaudhuri, "On a Geometric Notion of Quantiles for Multivariate Data," *Journal of the American Statistical Association*, vol. 91, no. 434, pp. 862–872, 1996 (cit. on p. 47).
34. J. K. Choi and Y. G. Ji, "Investigating the Importance of Trust on Adopting an Autonomous Vehicle," *International Journal of Human-Computer Interaction*, vol. 31, no. 10, pp. 692–702, 2015 (cit. on p. 7).
35. B. Christian, *The Alignment Problem: Machine Learning and Human Values*. W. W. Norton & Company, 2020 (cit. on p. 2).
36. J. P. Chryssanthacopoulos and M. J. Kochenderfer, "Collision Avoidance System Optimization with Probabilistic Pilot Response Models," in *American Control Conference (ACC)*, 2011 (cit. on p. 51).
37. C. J. Clopper and E. S. Pearson, "The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial," *Biometrika*, vol. 26, no. 4, pp. 404–413, 1934 (cit. on p. 196).
38. J. Colin, T. Fel, R. Cadène, and T. Serre, "What I Cannot Predict, I Do Not Understand: A Human-Centered Evaluation Framework for Explainability Methods," *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 2832–2845, 2022 (cit. on p. 292).
39. V. Conitzer, "Eliciting Single-Peaked Preferences Using Comparison Queries," *Journal of Artificial Intelligence Research*, vol. 35, pp. 161–191, 2009 (cit. on p. 68).

40. A. Couëtoux, J.-B. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard, "Continuous Upper Confidence Trees," in *Learning and Intelligent Optimization (LION)*, 2011 (cit. on p. 130).
41. S. Dandl, C. Molnar, M. Binder, and B. Bischl, "Multi-Objective Counterfactual Explanations," in *International Conference on Parallel Problem Solving from Nature*, 2020 (cit. on pp. 309, 312, 314).
42. T. Dang and T. Nahhal, "Coverage-Guided Test Generation for Continuous and Hybrid Systems," *Formal Methods in System Design*, vol. 34, pp. 183–213, 2009 (cit. on p. 122).
43. P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, "A Tutorial on the Cross-Entropy Method," *Annals of Operations Research*, vol. 134, pp. 19–67, 2005 (cit. on p. 171).
44. P. Del Moral, A. Doucet, and A. Jasra, "Sequential Monte Carlo Samplers," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 68, no. 3, pp. 411–436, 2006 (cit. on p. 179).
45. H. Delecki, A. Corso, and M. J. Kochenderfer, "Model-Based Validation as Probabilistic Inference," in *Conference on Learning for Dynamics and Control (L4DC)*, 2023 (cit. on p. 147).
46. W. M. Dickie, "A Comparison of the Scientific Method and Achievement of Aristotle and Bacon," *The Philosophical Review*, vol. 31, no. 5, pp. 471–494, 1922 (cit. on p. 3).
47. A. Donzé and O. Maler, "Robust Satisfaction of Temporal Logic over Real-Valued Signals," in *International Conference on Formal Modeling and Analysis of Timed Systems*, 2010 (cit. on p. 77).
48. M. Dowson, "The Ariane 5 Software Failure," *Software Engineering Notes*, vol. 22, no. 2, p. 84, 1997 (cit. on p. 7).
49. S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth, "Hybrid Monte Carlo," *Physics Letters B*, vol. 195, no. 2, pp. 216–222, 1987 (cit. on p. 152).
50. A. Duret-Lutz, "Manipulating LTL Formulas Using Spot 1.0," in *Automated Technology for Verification and Analysis*, 2013 (cit. on p. 84).
51. V. Dwaracherla, Z. Wen, I. Osband, X. Lu, S. M. Asghari, and B. Van Roy, "Ensembles for Uncertainty Estimation: Benefits of Prior Functions and Bootstrapping," *Transactions on Machine Learning Research*, 2022 (cit. on p. 347).
52. EASA AI Task Force, "Concepts of Design Assurance for Neural Networks," *EASA*, 2020 (cit. on p. 5).
53. B. Efron, "Bootstrap Methods: Another Look at the Jackknife," in *Breakthroughs in Statistics: Methodology and Distribution*, Springer, 1992, pp. 569–593 (cit. on p. 39).

54. V. Elvira, L. Martino, D. Luengo, and M. F. Bugallo, "Generalized Multiple Importance Sampling," *Statistical Science*, vol. 34, no. 1, pp. 129–155, 2019 (cit. on p. 168).
55. A. Engel, *Verification, Validation, and Testing of Engineered Systems*. John Wiley & Sons, 2010, vol. 73 (cit. on p. 1).
56. J. M. Esposito, J. Kim, and V. Kumar, "Adaptive RRTs for Validating Hybrid Robotic Control Systems," in *Algorithmic Foundations of Robotics*, Springer, 2005, pp. 107–121 (cit. on p. 120).
57. M. Everett, G. Habibi, C. Sun, and J. P. How, "Reachability Analysis of Neural Feedback Loops," *IEEE Access*, vol. 9, pp. 163 938–163 953, 2021 (cit. on p. 254).
58. M. Everett, G. Habibi, C. Sun, and J. P. How, "Reachability Analysis of Neural Feedback Loops," *IEEE Access*, vol. 9, pp. 163 938–163 953, 2021 (cit. on p. 257).
59. M. Forets and C. Schilling, "LazySets.jl: Scalable Symbolic-Numeric Set Computations," *Proceedings of the JuliaCon Conferences*, vol. 1, no. 1, pp. 1–11, 2021 (cit. on p. 205).
60. K. Forsberg and H. Mooz, "The Relationship of System Engineering to the Project Cycle," *Center for Systems Management*, vol. 5333, 1991 (cit. on p. 5).
61. D. Fudenberg and J. Tirole, *Game Theory*. MIT Press, 1991 (cit. on p. 42).
62. H. Ge, K. Xu, and Z. Ghahramani, "Turing: a Language for Flexible Probabilistic Inference," in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2018 (cit. on p. 35).
63. R. Geirhos, J.-H. Jacobsen, C. Michaelis, R. Zemel, W. Brendel, M. Bethge, and F. A. Wichmann, "Shortcut Learning in Deep Neural Networks," *Nature Machine Intelligence*, vol. 2, no. 11, pp. 665–673, 2020 (cit. on p. 295).
64. J. W. Gelder, "Air Law: The Federal Aviation Act of 1958," *Michigan Law Review*, vol. 57, no. 8, pp. 1214–1227, 1959 (cit. on p. 5).
65. B. Ghojogh, M. Crowley, F. Karay, and A. Ghodsi, *Elements of Dimensionality Reduction and Manifold Learning*. Springer, 2023 (cit. on p. 329).
66. T. Gneiting and M. Katzfuss, "Probabilistic Forecasting," *Annual Review of Statistics and Its Application*, vol. 1, no. 1, pp. 125–151, 2014 (cit. on p. 332).
67. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016 (cit. on p. 375).
68. I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 27, 2014 (cit. on p. 24).
69. B. Goodman and S. Flaxman, "European Union Regulations on Algorithmic Decision-Making and a 'Right to Explanation,'" *AI Magazine*, vol. 38, no. 3, pp. 50–57, 2017 (cit. on p. 292).

70. M. A. Gosavi, B. B. Rhoades, and J. M. Conrad, "Application of Functional Safety in Autonomous Vehicles Using ISO 26262 Standard: A Survey," in *SoutheastCon*, 2018 (cit. on p. 5).
71. U. Grenander and M. I. Miller, "Representations of Knowledge in Complex Systems," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 56, no. 4, pp. 549–581, 1994 (cit. on p. 149).
72. A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed. SIAM, 2008 (cit. on p. 153).
73. A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed. SIAM, 2008 (cit. on p. 379).
74. C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On Calibration of Modern Neural Networks," in *International Conference on Machine Learning (ICML)*, 2017 (cit. on p. 336).
75. H. Hansson and B. Jonsson, "A Logic for Reasoning about Time and Reliability," *Formal Aspects of Computing*, vol. 6, pp. 512–535, 1994 (cit. on p. 63).
76. P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968 (cit. on p. 126).
77. T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. Springer Series in Statistics, 2001 (cit. on p. 28).
78. W. K. Hastings, "Monte Carlo Sampling Methods Using Markov Chains and Their Applications," *Biometrika*, vol. 57, no. 1, pp. 97–97, 1970 (cit. on p. 144).
79. C. Hensel, S. Junges, J.-P. Katoen, T. Quatmann, and M. Volk, "The Probabilistic Model Checker Storm," *International Journal on Software Tools for Technology Transfer*, pp. 1–22, 2022 (cit. on p. 274).
80. J. Herkert, J. Borenstein, and K. Miller, "The Boeing 737 MAX: Lessons for Engineering Ethics," *Science and Engineering Ethics*, vol. 26, pp. 2957–2974, 2020 (cit. on p. 7).
81. A. L. Hodgkin and A. F. Huxley, "A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in Nerve," *Journal of Physiology*, vol. 117, no. 4, pp. 500–544, 1952 (cit. on p. 376).
82. W. Hoeffding, "Probability Inequalities for Sums of Bounded Random Variables," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 13–30, 1963 (cit. on p. 196).
83. M. D. Hoffman, A. Gelman, et al., "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo," *Journal of Machine Learning Research (JMLR)*, vol. 15, no. 1, pp. 1593–1623, 2014 (cit. on pp. 37, 152).

84. A. L. Hunkenschroer and A. Kriebitz, "Is AI Recruiting (Un)ethical? A Human Rights Perspective on the Use of AI for Hiring," *AI and Ethics*, vol. 3, no. 1, pp. 199–213, 2023 (cit. on p. 6).
85. M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004 (cit. on pp. 70, 71).
86. K. Ishikawa and J. H. Loftus, "Introduction to Quality Control," in Springer, 1990, vol. 98, ch. 1 (cit. on p. 3).
87. V. S. Iyengar, J. Lee, and M. Campbell, "Q-EVAL: Evaluating Multiple Attribute Items Using Queries," in *ACM Conference on Electronic Commerce*, 2001 (cit. on p. 69).
88. L. Jaulin, M. Kieffer, O. Didrit, and É. Walter, *Interval Analysis*. Springer, 2001 (cit. on p. 231).
89. E. T. Jaynes, *Probability Theory: The Logic of Science*. Cambridge University Press, 2003 (cit. on p. 20).
90. P. Jorion, "Risk Management Lessons from Long-Term Capital Management," *European Financial Management*, vol. 6, no. 3, pp. 277–300, 2000 (cit. on p. 7).
91. H. Kahn and T. E. Harris, "Estimation of Particle Transmission by Random Sampling," *National Bureau of Standards Applied Mathematics Series*, vol. 12, pp. 27–30, 1951 (cit. on p. 191).
92. G. K. Kamenev, "An Algorithm for Approximating Polyhedra," *Computational Mathematics and Mathematical Physics*, vol. 4, no. 36, pp. 533–544, 1996 (cit. on p. 220).
93. S. Karaman and E. Frazzoli, "Incremental Sampling-Based Algorithms for Optimal Motion Planning," *Robotics Science and Systems VI*, vol. 104, no. 2, pp. 267–274, 2010 (cit. on p. 125).
94. H. Karloff, *Linear Programming*. Springer, 2008 (cit. on p. 222).
95. S. M. Katz, K. D. Julian, C. A. Strong, and M. J. Kochenderfer, "Generating Probabilistic Safety Guarantees for Neural Network Controllers," *Machine Learning*, vol. 112, pp. 2903–2931, 2023 (cit. on p. 282).
96. S. M. Katz, A.-C. LeBihan, and M. J. Kochenderfer, "Learning an Urban Air Mobility Encounter Model from Expert Preferences," in *Digital Avionics Systems Conference (DASC)*, 2019 (cit. on p. 28).
97. P. Kirichenko, P. Izmailov, and A. G. Wilson, "Why Normalizing Flows Fail to Detect Out-Of-Distribution Data," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20 578–20 589, 2020 (cit. on p. 329).
98. J. Kleinberg, S. Mullainathan, and M. Raghavan, "Inherent Trade-Offs in the Fair Determination of Risk Scores," in *Innovations in Theoretical Computer Science (ITCS) Conference*, 2017 (cit. on p. 7).

99. I. Kobyzev, S. J. Prince, and M. A. Brubaker, "Normalizing Flows: An Introduction and Review of Current Methods," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 11, pp. 3964–3979, 2020 (cit. on p. 24).
100. M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019 (cit. on pp. 104, 297, 312).
101. M. J. Kochenderfer and J. P. Chryssanthacopoulos, "Robust Airborne Collision Avoidance Through Dynamic Programming," Massachusetts Institute of Technology, Lincoln Laboratory, Project Report ATC-371, 2011 (cit. on p. 365).
102. M. J. Kochenderfer, M. W. M. Edwards, L. P. Espindle, J. K. Kuchar, and J. D. Griffith, "Airspace Encounter Models for Estimating Collision Risk," *AIAA Journal on Guidance, Control, and Dynamics*, vol. 33, no. 2, pp. 487–499, 2010 (cit. on p. 44).
103. M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray, *Algorithms for Decision Making*. MIT Press, 2022 (cit. on pp. 2, 10, 25, 27, 34, 41, 364).
104. R. Koenker and K. F. Hallock, "Quantile Regression," *Journal of Economic Perspectives*, vol. 15, no. 4, pp. 143–156, 2001 (cit. on p. 332).
105. D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009 (cit. on p. 25).
106. A. Kolmogorov, *Foundations of the Theory of Probability*, 2nd ed. Chelsea, 1956 (cit. on p. 368).
107. A. Kossiakoff, S. M. Biemer, S. J. Seymour, and D. A. Flanigan, *Systems Engineering Principles and Practice*. John Wiley & Sons, 2020 (cit. on p. 2).
108. J. Kuchar and A. C. Drumm, "The Traffic Alert and Collision Avoidance System," *Lincoln Laboratory Journal*, vol. 16, no. 2, p. 277, 2007 (cit. on p. 6).
109. S. Kullback and R. A. Leibler, "On Information and Sufficiency," *Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951 (cit. on p. 371).
110. S. Kullback, *Information Theory and Statistics*. Wiley, 1959 (cit. on p. 371).
111. S. Kullback and R. A. Leibler, "On Information and Sufficiency," *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951 (cit. on p. 46).
112. M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of Probabilistic Real-Time Systems," in *International Conference on Computer Aided Verification*, 2011 (cit. on p. 274).
113. B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017 (cit. on p. 347).
114. S. LaValle, "Planning Algorithms," *Cambridge University Press*, vol. 2, pp. 3671–3678, 2006 (cit. on p. 115).

115. R. Lee, M. J. Kochenderfer, O. J. Mengshoel, and J. Silbermann, "Interpretable Categorization of Heterogeneous Time Series Data," in *SIAM International Conference on Data Mining*, 2018 (cit. on p. 319).
116. R. Lee, O. J. Mengshoel, A. Saksena, R. W. Gardner, D. Genin, J. Silbermann, M. Owen, and M. J. Kochenderfer, "Adaptive Stress Testing: Finding Likely Failure Events with Reinforcement Learning," *Journal of Artificial Intelligence Research*, vol. 69, pp. 1165–1201, 2020 (cit. on p. 128).
117. J. Lehrer, *How We Decide*. Houghton Mifflin, 2009 (cit. on p. 41).
118. K. Leung, N. Aréchiga, and M. Pavone, "Backpropagation Through Signal Temporal Logic Specifications: Infusing Logical Structure into Gradient-Based Methods," *The International Journal of Robotics Research*, vol. 42, no. 6, pp. 356–370, 2023 (cit. on p. 81).
119. N. G. Leveson and C. S. Turner, "An Investigation of the Therac-25 Accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993 (cit. on p. 6).
120. F. Liese and I. Vajda, "On Divergences and Informations in Statistics and Information Theory," *IEEE Transactions on Information Theory*, vol. 52, no. 10, pp. 4394–4412, 2006 (cit. on p. 46).
121. C. Liu, T. Arnon, C. Lazarus, C. Strong, C. Barrett, and M. J. Kochenderfer, "Algorithms for Verifying Deep Neural Networks," *Foundations and Trends in Optimization*, vol. 4, no. 3–4, pp. 244–404, 2021 (cit. on p. 254).
122. F. Llorente, L. Martino, D. Delgado, and J. Lopez-Santiago, "Marginal Likelihood Computation for Model Selection and Hypothesis Testing: an Extensive Review," *SIAM Review*, vol. 65, no. 1, pp. 3–58, 2023 (cit. on pp. 178, 181).
123. S. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982 (cit. on p. 315).
124. K. Makino and M. Berz, "Taylor Models and Other Validated Functional Inclusion Methods," *International Journal of Pure and Applied Mathematics*, vol. 4, no. 4, pp. 379–456, 2003 (cit. on p. 240).
125. O. Maler, "Computing Reachable Sets: An Introduction," *French National Center of Scientific Research*, pp. 1–8, 2008 (cit. on p. 212).
126. O. Maler and D. Nickovic, "Monitoring Temporal Properties of Continuous Signals," in *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, 2004 (cit. on p. 75).
127. R. L. McCarthy, "Autonomous Vehicle Accident Data Analysis: California OL 316 Reports: 2015–2020," *ASCE-ASME Journal of Risk and Uncertainty in Engineering Systems, Part B: Mechanical Engineering*, vol. 8, no. 3, p. 034502, 2022 (cit. on p. 6).
128. S. B. McGrayne, *The Theory That Would Not Die*. Yale University Press, 2011 (cit. on p. 34).

129. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," *Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953 (cit. on p. 144).
130. B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990 (cit. on p. 97).
131. B. Müller, J. Reinhardt, and M. T. Strickland, *Neural Networks*. Springer, 1995 (cit. on p. 375).
132. C. N. Murphy and J. Yates, *The International Organization for Standardization (ISO): Global Governance Through Voluntary Consensus*. Routledge, 2009 (cit. on p. 5).
133. K. P. Murphy, *Probabilistic Machine Learning: An Introduction*. MIT Press, 2022 (cit. on p. 28).
134. R. Neidinger, "Directions for Computing Truncated Multivariate Taylor Series," *Mathematics of Computation*, vol. 74, no. 249, pp. 321–340, 2005 (cit. on p. 237).
135. A. Niculescu-Mizil and R. Caruana, "Predicting Good Probabilities with Supervised Learning," in *International Conference on Machine Learning (ICML)*, 2005 (cit. on p. 338).
136. J. Nocedal, "Updating Quasi-Newton Matrices with Limited Storage," *Mathematics of Computation*, vol. 35, no. 151, pp. 773–782, 1980 (cit. on pp. 106, 113).
137. J. R. Norris, *Markov Chains*. Cambridge University Press, 1998 (cit. on p. 274).
138. R. Page and R. Gamboa, *Essential Logic for Computer Science*. MIT Press, 2019 (cit. on p. 71).
139. J. Pearl, "Direct and Indirect Effects," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2001 (cit. on p. 295).
140. D. Phillips-Donaldson, "100 Years of Juran," *Quality Progress*, vol. 37, no. 5, pp. 25–31, 2004 (cit. on p. 4).
141. A. Pinkus, "Approximation Theory of the MLP Model in Neural Networks," *Acta Numerica*, vol. 8, pp. 143–195, 1999 (cit. on p. 376).
142. A. Pnueli, "The Temporal Logic of Programs," in *Symposium on Foundations of Computer Science (SFCS)*, 1977 (cit. on p. 75).
143. J. Postels, M. Segù, T. Sun, L. D. Sieber, L. Van Gool, F. Yu, and F. Tombari, "On the Practicality of Deterministic Epistemic Uncertainty," in *International Conference on Learning Representations (ICLR)*, 2022 (cit. on p. 330).
144. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007 (cit. on p. 111).

145. Y. Pu, Z. Gan, R. Henao, X. Yuan, C. Li, A. Stevens, and L. Carin, "Variational Autoencoder for Deep Learning of Images, Labels and Captions," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 29, 2016 (cit. on p. 331).
146. J. Reason, "Human Error: Models and Management," *British Medical Journal*, vol. 320, no. 7237, pp. 768–770, 2000 (cit. on p. 14).
147. R. G. Regis, "On the Properties of Positive Spanning Sets and Positive Bases," *Optimization and Engineering*, vol. 17, no. 1, pp. 229–262, 2016 (cit. on p. 219).
148. M. T. Ribeiro, S. Singh, and C. Guestrin, "'Why Should I Trust You?' Explaining the Predictions of Any Classifier," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016 (cit. on p. 306).
149. L. Rierson, *Developing Safety-Critical Software: a Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2017 (cit. on p. 5).
150. C. P. Robert and G. Casella, *Monte Carlo Statistical Methods*. Springer, 1999, vol. 2 (cit. on pp. 140, 144).
151. R. T. Rockafellar and S. Uryasev, "Optimization of Conditional Value-at-Risk," *Journal of Risk*, vol. 2, pp. 21–42, 2000 (cit. on p. 64).
152. S. Ross, G. J. Gordon, and J. A. Bagnell, "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning," in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, vol. 15, 2011 (cit. on p. 40).
153. W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," *IEEE WESCON*, 1970 (cit. on p. 5).
154. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-Propagating Errors," *Nature*, vol. 323, pp. 533–536, 1986 (cit. on p. 379).
155. S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021 (cit. on p. 269).
156. H. Samet, *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006 (cit. on p. 325).
157. R. Seidel, "Convex Hull Computations," in *Handbook of Discrete and Computational Geometry*, Chapman and Hall, 2017, pp. 687–703 (cit. on p. 214).
158. C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, no. 4, pp. 623–656, 1948 (cit. on p. 370).
159. L. S. Shapley, "Notes on the N-Person Game—II: The Value of an N-Person Game," 1951 (cit. on p. 302).
160. Y. Shoham and K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game Theoretic, and Logical Foundations*. Cambridge University Press, 2009 (cit. on p. 42).

161. C. Sidrane, A. Maleki, A. Irfan, and M. J. Kochenderfer, "OVERT: An Algorithm for Safety Verification of Neural Network Control Policies for Nonlinear Systems," *Journal of Machine Learning Research*, vol. 23, no. 117, pp. 1–45, 2022 (cit. on pp. 249, 250).
162. J. Siegel and G. Pappas, "Morals, Ethics, and the Technology Capabilities and Limitations of Automated and Self-Driving Vehicles," *AI & Society*, vol. 38, no. 1, pp. 213–226, 2023 (cit. on p. 7).
163. S. Sigl and M. Althoff, "M-Representation of Polytopes," *ArXiv:2303.05173*, 2023 (cit. on p. 214).
164. K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps," in *International Conference on Learning Representations (ICLR)*, 2014 (cit. on p. 300).
165. A. Sinha, M. O'Kelly, R. Tedrake, and J. C. Duchi, "Neural Bridge Sampling for Evaluating Safety-Critical Autonomous Systems," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 6402–6416, 2020 (cit. on pp. 178, 188).
166. D. Smilkov, N. Thorat, B. Kim, F. Viégas, and M. Wattenberg, "Smoothgrad: Removing Noise by Adding Noise," in *International Conference on Machine Learning (ICML)*, 2017 (cit. on p. 300).
167. E. Soroka, M. J. Kochenderfer, and S. Lall, "Satisfiability.jl: Satisfiability Modulo Theories in Julia," *Journal of Open Source Software*, vol. 9, no. 100, p. 6757, 2024 (cit. on p. 270).
168. D. O. Stahl and P. W. Wilson, "Experimental Evidence on Players' Models of Other Players," *Journal of Economic Behavior & Organization*, vol. 25, no. 3, pp. 309–327, 1994 (cit. on p. 42).
169. C. A. Strong, H. Wu, A. Zeljic, K. D. Julian, G. Katz, C. Barrett, and M. J. Kochenderfer, "Global Optimization of Objective Functions Represented by ReLU Networks," *Machine Learning*, vol. 112, pp. 3685–3712, 2023 (cit. on p. 257).
170. E. Štrumbelj and I. Kononenko, "Explaining Prediction Models and Individual Predictions with Feature Contributions," *Knowledge and Information Systems*, vol. 41, pp. 647–665, 2014 (cit. on p. 303).
171. O. Stursberg and B. H. Krogh, "Efficient Representation and Computation of Reachable Sets for Hybrid Systems," in *Hybrid Systems: Computation and Control*, 2003 (cit. on p. 222).
172. M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic Attribution for Deep Networks," in *International Conference on Machine Learning (ICML)*, 2017 (cit. on p. 300).
173. R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, Second Edition. MIT Press, 2018 (cit. on p. 130).

174. E. Thiémard, "An Algorithm to Compute Bounds for the Star Discrepancy," *Journal of Complexity*, vol. 17, no. 4, pp. 850–880, 2001 (cit. on p. 121).
175. S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT Press, 2006 (cit. on p. 181).
176. R. Tibshirani, "Regression Shrinkage and Selection via the Lasso," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 58, no. 1, pp. 267–288, 1996 (cit. on p. 308).
177. V. Tjeng, K. Y. Xiao, and R. Tedrake, "Evaluating Robustness of Neural Networks with Mixed Integer Programming," in *International Conference on Learning Representations (ICLR)*, 2018 (cit. on pp. 249, 257).
178. H.-D. Tran, D. Manzanas Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, and T. T. Johnson, "Star-Based Reachability Analysis of Deep Neural Networks," in *International Symposium on Formal Methods*, 2019 (cit. on p. 240).
179. W. M. Tsutsui, "W. Edwards Deming and the Origins of Quality Control in Japan," *Journal of Japanese Studies*, vol. 22, no. 2, pp. 295–325, 1996 (cit. on p. 4).
180. A. M. Turing, "Computing Machinery and Intelligence," *Mind*, vol. 59, pp. 433–460, 1950 (cit. on p. 48).
181. M. Vazquez-Chanlatte, J. V. Deshmukh, X. Jin, and S. A. Seshia, "Logical Clustering and Learning for Time-Series Data," in *International Conference on Computer Aided Verification*, 2017 (cit. on p. 318).
182. A. G. Wilson and P. Izmailov, "Bayesian Deep Learning and a Probabilistic Perspective of Generalization," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 4697–4708, 2020 (cit. on p. 346).
183. L. A. Wolsey, *Integer Programming*. Wiley, 2020 (cit. on p. 249).
184. B. Wong, "Points of View: Color Blindness," *Nature Methods*, vol. 8, no. 6, pp. 441–442, 2011 (cit. on p. xiii).
185. W. Xiang, H.-D. Tran, and T. T. Johnson, "Output Reachable Set Estimation and Verification for Multilayer Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 11, pp. 5777–5783, 2018 (cit. on p. 257).
186. W. Xiang, H.-D. Tran, J. A. Rosenfeld, and T. T. Johnson, "Reachable Set Estimation and Safety Verification for Piecewise Linear Systems with Neural Network Controllers," in *American Control Conference (ACC)*, 2018 (cit. on p. 255).
187. D. Xu and Y. Tian, "A Comprehensive Survey of Clustering Algorithms," *Annals of Data Science*, vol. 2, pp. 165–193, 2015 (cit. on p. 315).
188. J. Yang, K. Zhou, Y. Li, and Z. Liu, "Generalized Out-Of-Distribution Detection: A Survey," *International Journal of Computer Vision*, vol. 132, no. 12, pp. 5635–5662, 2024 (cit. on p. 324).

189. H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel, "Efficient Neural Network Robustness Certification with General Activation Functions," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 31, 2018 (cit. on p. 257).
190. Y.-D. Zhou, K.-T. Fang, and J.-H. Ning, "Mixture Discrepancy for Quasi-Random Point Sets," *Journal of Complexity*, vol. 29, no. 3-4, pp. 283–301, 2013 (cit. on p. 121).
191. B. D. Ziebart, A. Maas, J. A. Bagnell, and A. K. Dey, "Maximum Entropy Inverse Reinforcement Learning," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2008 (cit. on p. 41).
192. G. M. Ziegler, *Lectures on Polytopes*. Springer Science & Business Media, 2012, vol. 152 (cit. on p. 213).
193. A. Zutshi, J. V. Deshmukh, S. Sankaranarayanan, and J. Kapinski, "Multiple Shooting, CEGAR-Based Falsification for Hybrid Systems," in *International Conference on Embedded Software*, 2014 (cit. on p. 113).

Index

- \mathcal{H} -polytope, 213
 \mathcal{V} -polytope, 213
k-means, 315
- absolutely homogeneous, 368
abstract types, 392
activation function, 376
activation pattern, 256
Adaptive importance sampling, 171
adaptive stress testing, 128
admissible, 126
adversary, 130
aleatoric uncertainty, *see* output uncertainty
alignment problem, 2, 291
anonymous function, 394
array comprehension, 384
atomic proposition, 70
average dispersion, 120
avoid set, 204
- backpropagation, 379
backward reachability, 265
batch, 375
Bayesian model averaging, 346
Bayesian parameter learning, 34
behavior model, 41
behavioral cloning, 39
best response, 41
biconditional, 71
- bit, 370
black-box simulators, 105
Boolean, 381
Boolean satisfiability, 270
bootstrap method, 39
bounded model checking, 265
breadth-first search, 265
bridge density, 184
bridge sampling, 184
broadcasting, 387
burn-in, 144
Büchi automaton, 82
- calibration, 335
calibration plot, 44
callable, 395
cascading errors, 40
Chebyshev norm, 369
chessboard norm, 369
Clopper-Pearson bound, 196
closed under complementation, 367
closed under countable unions, 367
clustering, 315
coefficient of variation, 195
coherent risk measure, 64
composite metric, 66
composite metrics, 66
composite type, 392
Computation tree logic, 75
- Concrete reachability, 244
concrete types, 392
concretize, 244
conditional coverage, 340
conditional distribution, 25
conditional Gaussian distribution, 27
conditional value at risk, 64
conformal prediction, 339
conjugate prior, 34
conjunction, 71
conservative linearization, 240
consistent, 161
continuous entropy, *see* differential entropy
convex, 212
convex combinations, 213
convex hull, 213
convex set, 212
countable additivity, 367
counterexample, 269
counterexample search, 269
counterexamples, 93
counterfactual, 309
counterfactual explanation, 309
coverage, 119, 338
coverage metrics, 119
cross entropy, 171, 371
cross entropy method, 171
CTL, *see* Computation tree logic

- curse of dimensionality, 328
CVaR, *see* conditional value at risk
- DAgger, *see* data set aggregation
 data set aggregation, 40
 decision tree, 309
 deep learning, 375
 deep neural network, 376
 defect, 111
 dependency effect, 208
 depth of rationality, 42
 deterministic, 82
 deterministic Rabin automata, 84
 dictionary, 391
 differential entropy, 370
 direct methods, 105
 discrepancy, 120
 discrete state abstraction, 280
 disjunction, 71
 dispatch, 396
 dispersion, 119
 distance metric, 368
 disturbance distribution, 96
 disturbances, 93, 94
 double progressive widening, 130
- earth mover's distance, 46
 ECE, *see* expected calibration error
 effective sample size, 198
 elite samples, 173
 EM, *see* expectation-maximization
 entropic value at risk, 64
 entropy, 335, 370
 environment, 8
 episode, 130
 epistemic uncertainty, 331
 ESS, *see* effective sample size
 Euclidean norm, 369
 event space, 368
 exchangeable, 340
 existential quantifier, 73
 expectation-maximization, 34
- expected calibration error, 46
 expected value, 62
 explainability, 292
 explanation, 291
 exploitation, 126
 exploration, 126
- F-divergences, 46
 failure region, 118
 failure trajectories, 93
 failures, 93
 Falsification, 13
 falsification, 93
 falsifying trajectories, 93
 feature, 293
 feature collapse, 330
 feature importance, 293
 feedforward network, 376
 first fundamental theorem of calculus, 371
 first-order, 105
 first-order logic, 71
 forward reachability, 203, 265
 function, 394
 fuzzing, 97
- Gaussian distribution, 21, 22
 Gaussian mixture model, 23
 generalization performance, 35
 generative adversarial networks, 24
 generative models, 24
 generators, 214
 global explanation, 306
- half space, 212
 Hausdorff distance, 216
 hierarchical, 42
 hierarchical softmax, 42
 histogram binning, 336
 Hoeffding's inequality, 196
 holdout method, 39
 hyperrectangle, 215
- identity of indiscernibles, 368
 imitation game, 48
 imitation learning, 39
 implication, 71
 Importance sampling, 165
 inclusion function, 233
 independently and identically distributed, 28
 individuals, 107
 information content, 370
 integrated gradients, 300
 interaction models, 42
 interpretability, 292
 interval arithmetic, 231
 interval box, *see* hyperrectangle
 interval counterpart, 232
 interval hull, 232
 intervals, 215
 invariant set, 211
 inverse reinforcement learning, 40
 inverse transform sampling, 53
 irreducible uncertainty, *see* output uncertainty
 iterative deepening, 269
 iterative refinement, 220
- joint distribution, 25
- k-fold cross validation, 39
 K-S statistic, *see* Kolmogorov-Smirnov statistic
 kernel, 144
 keyword argument, 395
 KL divergence, *see* Kullback-Leibler divergence
 Kolmogorov-Smirnov statistic, 46
 Kolmogorov axioms, 368
 Kullback-Leibler divergence, 44, *see* relative entropy
- Lagrange remainder, 240
 Laplace distribution, 54

- least-squares, 28, 29
 linear inequalities, 212
 linear model, 306
 linear program, 221
 linear systems, 204
 Linear temporal logic, 75
 linearization, 236
 local descent methods, 105
 local explanation, 306
 log-likelihood, 28
 logic gates, 71
 logical formula, 70
 logical specification, 70
 logit response, 41
 logit-level-k, 42
 loop, 398
 loss function, 375
 lower confidence bound (LCB), 128
 LTL, *see* Linear temporal logic
- machine learning, 28
 marginal coverage, 340
 marginal satisfaction, 318
 Markov chain, 144
 Markov chain Monte Carlo, 144
 matrix, 387
 max norm, 369
 maximum a posteriori, 34
 maximum calibration error, 46
 maximum entropy inverse
 reinforcement learning, 40
 maximum likelihood estimate, 28
 maximum likelihood parameter
 learning, 28
 maximum margin inverse
 reinforcement learning, 40
 MCE, *see* maximum calibration error
 mean excess loss, 64
 mean shortfall, 64
 mean value theorem, 234
 measurable set, 367
 measure, 367
 measure space, 367
 metric, 368
 metric space, 368
 metrics, 61
 Metropolis-Adjusted Langevin
 Algorithm, 149
 Metropolis-Hastings, 144
 Minkowski sum, 208
 mixed-integer linear program, 249
 mixture models, 21
 mode, 34
 model class, 19
 model uncertainty, 331
 Monte Carlo tree search (MCTS), 126
 multimodal, 21
 multiple importance sampling, 168
 multiple shooting, 112
 multivariate distribution, 25
 multivariate Gaussian distribution, 25
 mutate, 385
 named function, 394
 named tuple, 391
 nat, 370
 natural, 370
 natural inclusion function, 234
 negation, 71
 neural network, 375
 neural network verification, 254
 nonnegativity, 367
 nonparametric, 181
 normal distribution, 21, 22
 normalizing flows, 23
 normed vector space, 368
 ODD, *see* operational design domain
 offline, 323
 online, 323
 operational design domain, 323
 out of distribution detection, 324
 output uncertainty, 330
 overapproximation, 216
 overapproximation error, 216
 overfitting, 39
 package, 399
 parameter learning, 27
 parameter tuning, 375
 parametric signal temporal logic, 318
 parametric types, 393
 Pareto frontier, 66
 Pareto optimal, 66
 partially observable Markov decision
 process, 10
 partitioning, 250
 polyhedron, 212
 polynomial zonotopes, 240
 polytope, 212
 polytopes, 212
 population, 175
 Population methods, 107
 Population Monte Carlo, 175
 positive definite, 370
 positive semidefinite, 370
 positive spanning set, 219
 posterior distribution, 34
 precision parameter, 41, 336
 predicate function, 71
 predicates, 71
 preference elicitation, 68
 prior distribution, 34
 Probabilistic programming, 152
 probabilistic programming, 35
 probability, 20
 probability axioms, 368
 probability density functions, 21
 probability distribution, 20
 probability mass function, 20
 probability measure, 368
 probability space, 368
 product system, 84
 progressive widening, 128

proper scoring rule, 332
 proposal distribution, 140
 proposition, 70
 propositional logic, 70
 prototypical example, 317
 pseudoinverse, 33
 pseudorandom number generator, 24
PSTL, *see* parametric signal temporal logic
 Q-Q plot, *see* quantile-quantile plot
 quantal response, 41
 quantal-level-k, 42
 quantifiers, 71
 quantile-quantile plot, 44
 rapidly exploring random trees, 114
 ratio importance sampling, 183
 reachability specification, 81
 reachable set, 204
 rectified linear unit, 250
 reducible uncertainty, *see* model uncertainty
 Reinforcement learning, 130
 Rejection sampling, 140
 relative entropy, 371
 reverse accumulation, 379
 risk metric, 63
 robustness, 77
 rollout, 10
 rotation estimation, 39
RRT, *see* rapidly exploring random trees
 runtime monitoring, 323

safety case, 14
 saliency maps, 297
 sample efficiency, 131
 sample space, 368
SAT, *see* Boolean satisfiability
 Satisfiability modulo theories, 272
 second-order, 105

Self-normalized importance sampling, 185
 sensitivity analysis, 295
 sensor, 9
 sequential interactive demonstration, 40
 sequential Monte Carlo, 177
 Set propagation, 205
 Shannon information, 370
 Shapley value, 302
 Shooting methods, 111
 sigmoid models, 27
 signal, 75
 Signal temporal logic, 75
 single shooting, 112
 smooth robustness, 81
 Smoothing, 147
SMT, *see* Satisfiability modulo theories
 softmax, 81, 335, 377
 softmax response, 41
 softmin, 81
 spatial index, 325
 specification, 8, 11
 specifications, 61
 splat, 396
 spurious correlations, 295
 standard error, 160
 Star discrepancy, 121
 star sets, 240
 stationary, 98
STL, *see* Signal temporal logic
 string, 383
 superlevel set, 328
 support, 21
 support function, 217
 support vector, 219
 supremum, 119
 surrogate model, 303
 Swiss cheese model, 14
 symbol, 384
 symbolic reachability, 244
 symmetry, 368
 system, 8
 tail value at risk, 64
 taxicab norm, 369
 Taylor approximation, 372
 Taylor expansion, 371
 Taylor inclusion function, 236
 Taylor models, 240
 Taylor series, 371
 temperature scaling, 336
 temporal logic, 73
 ternary operator, 397
 test set, 39
 testing, 1
 training, 27, 375
 training set, 39
 trajectory, 10
 triangle inequality, 368
 truthtable, 71
 tuple, 390
 Turing test, 48
 umbrella sampling, 183
 unbiased, 161
 unbounded model checking, 267
 uncertainty quantification, 330
 uniform distribution, 52
 unimodal, 21
 univariate distribution, 25
 universal quantifier, 73
 utopia point, 66

V model, 5
 validation, 1
 value at risk, 63
VaR, *see* value at risk
 variable, 71
 variance, 63
 vector, 384
 vector space, 368
 verification, 1

Wasserstein distance, 46

waterfall model, 4

weighted exponential sum, 68

weighted metrics, 66

weighted sum, 66

wrapping effect, 246

zonotope, 214