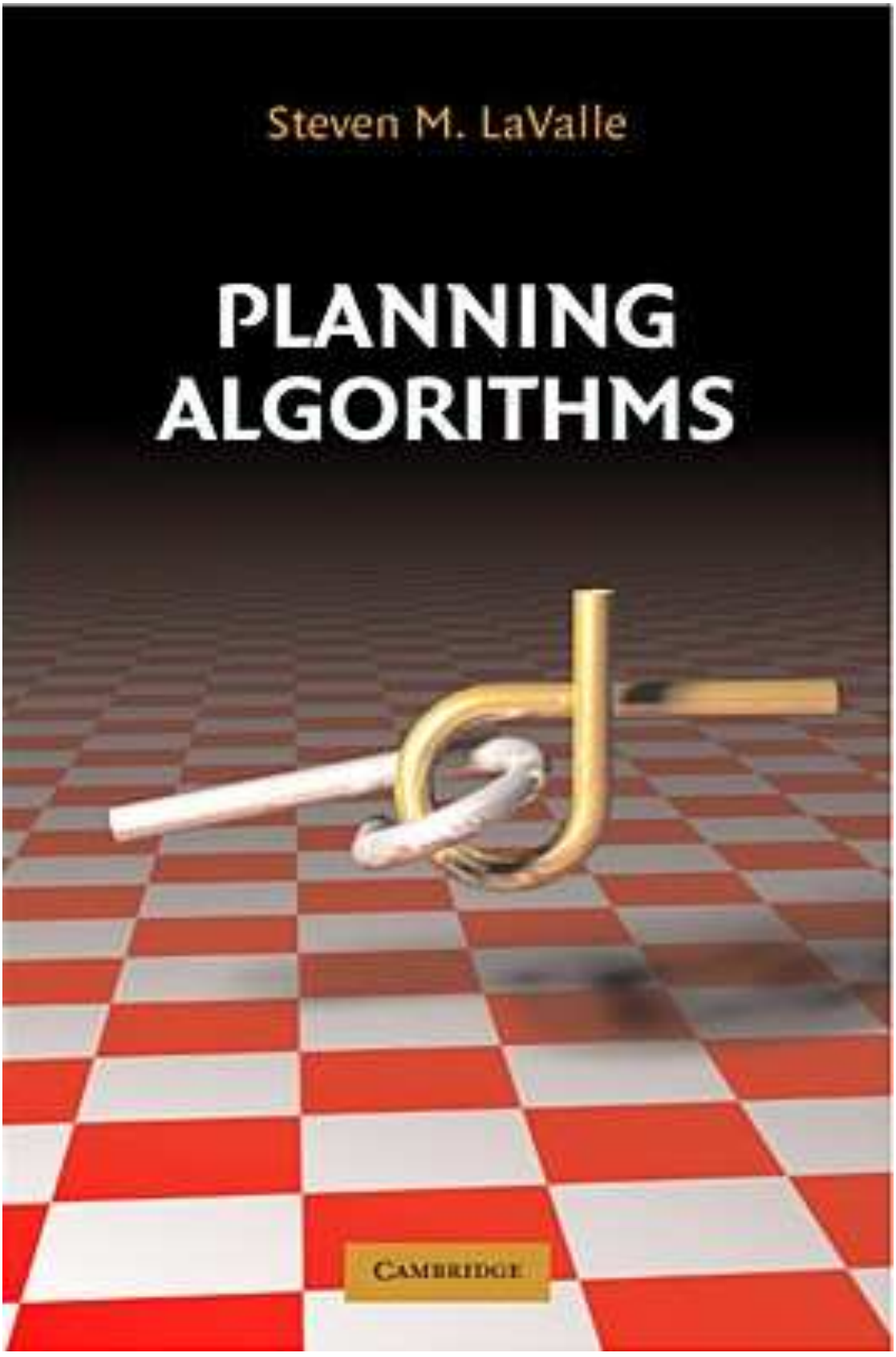

Steven M. LaValle
PLANNING
ALGORITHMS
CAMBRIDGE

PLANNING ALGORITHMS

Steven M. LaValle

University of Illinois

Available for downloading at **http://planning.cs.uiuc.edu/**

Published by Cambridge University Press

For Tammy, and my sons, Alexander and Ethan

Contents

Preface

What Is Meant by "Planning Algorithms"?

Due to many exciting developments in the fields of robotics, artificial intelligence, and control theory, three topics that were once quite distinct are presently on a collision course. In robotics, motion planning was originally concerned with problems such as how to move a piano from one room to another in a house without hitting anything. The field has grown, however, to include complications such as uncertainties, multiple bodies, and dynamics. In artificial intelligence, planning originally meant a search for a sequence of logical operators or actions that transform an initial world state into a desired goal state. Presently, planning extends beyond this to include many decision-theoretic ideas such as Markov decision processes, imperfect state information, and game-theoretic equilibria. Although control theory has traditionally been concerned with issues such as stability, feedback, and optimality, there has been a growing interest in designing algorithms that find feasible open-loop trajectories for nonlinear systems. In some of this work, the term "motion planning" has been applied, with a different interpretation from its use in robotics. Thus, even though each originally considered different problems, the fields of robotics, artificial intelligence, and control theory have expanded their scope to share an interesting common ground.

In this text, I use the term *planning* in a broad sense that encompasses this common ground. This does not, however, imply that the term is meant to cover everything important in the fields of robotics, artificial intelligence, and control theory. The presentation focuses on *algorithm* issues relating to planning. Within robotics, the focus is on designing algorithms that generate useful motions by processing complicated geometric models. Within artificial intelligence, the focus is on designing systems that use decision-theoretic models to compute appropriate actions. Within control theory, the focus is on algorithms that compute feasible trajectories for systems, with some additional coverage of feedback and optimality. Analytical techniques, which account for the majority of control theory literature, are not the main focus here.

The phrase "planning and control" is often used to identify complementary issues in developing a system. Planning is often considered as a higher level process than control. In this text, I make no such distinctions. Ignoring historical connotations that come with the terms, "planning" and "control" can be used

interchangeably. Either refers to some kind of decision making in this text, with no associated notion of "high" or "low" level. A hierarchical approach can be developed, and either level could be called "planning" or "control" without any difference in meaning.

Who Is the Intended Audience?

The text is written primarily for computer science and engineering students at the advanced-undergraduate or beginning-graduate level. It is also intended as an introduction to recent techniques for researchers and developers in robotics, artificial intelligence, and control theory. It is expected that the presentation here would be of interest to those working in other areas such as computational biology (drug design, protein folding), virtual prototyping, manufacturing, video game development, and computer graphics. Furthermore, this book is intended for those working in industry who want to design and implement planning approaches to solve their problems.

I have attempted to make the book as self-contained and readable as possible. Advanced mathematical concepts (beyond concepts typically learned by undergraduates in computer science and engineering) are introduced and explained. For readers with deeper mathematical interests, directions for further study are given.

Where Does This Book Fit?

Here is where this book fits with respect to other well-known subjects:

Robotics: This book addresses the planning part of robotics, which includes motion planning, trajectory planning, and planning under uncertainty. This is only one part of the big picture in robotics, which includes issues not directly covered here, such as mechanism design, dynamical system modeling, feedback control, sensor design, computer vision, inverse kinematics, and humanoid robotics.

Artificial Intelligence: Machine learning is currently one of the largest and most successful divisions of artificial intelligence. This book (perhaps along with [382]) represents the important complement to machine learning, which can be thought of as "machine planning." Subjects such as reinforcement learning and decision theory lie in the boundary between the two and are covered in this book. Once learning is being successfully performed, what decisions should be made? This enters into planning.

Control Theory: Historically, control theory has addressed what may be considered here as planning in continuous spaces under differential constraints. Dynamics, optimality, and feedback have been paramount in control theory. This book is complementary in that most of the focus is on open-loop control laws, feasibility as opposed to optimality, and dynamics may or may not be important.

Nevertheless, feedback, optimality, and dynamics concepts appear in many places throughout the book. However, the techniques in this book are mostly algorithmic, as opposed to the analytical techniques that are typically developed in control theory.

Computer Graphics: Animation has been a hot area in computer graphics in recent years. Many techniques in this book have either been applied or can be applied to animate video game characters, virtual humans, or mechanical systems. Planning algorithms allow users to specify tasks at a high level, which avoids having to perform tedious specifications of low-level motions (e.g., key framing).

Algorithms: As the title suggests, this book may fit under algorithms, which is a discipline within computer science. Throughout the book, typical issues from combinatorics and complexity arise. In some places, techniques from computational geometry and computational real algebraic geometry, which are also divisions of algorithms, become important. On the other hand, this is not a pure algorithms book in that much of the material is concerned with characterizing various decision processes that arise in applications. This book does not focus purely on complexity and combinatorics.

Other Fields: At the periphery, many other fields are touched by planning algorithms. For example, motion planning algorithms, which form a major part of this book, have had a substantial impact on such diverse fields as computational biology, virtual prototyping in manufacturing, architectural design, aerospace engineering, and computational geography.

Suggested Use

The ideas should flow naturally from chapter to chapter, but at the same time, the text has been designed to make it easy to skip chapters. The dependencies between the four main parts are illustrated in Figure 1.

If you are only interested in robot motion planning, it is only necessary to read Chapters 3–8, possibly with the inclusion of some discrete planning algorithms from Chapter 2 because they arise in motion planning. Chapters 3 and 4 provide the foundations needed to understand basic robot motion planning. Chapters 5 and 6 present algorithmic techniques to solve this problem. Chapters 7 and 8 consider extensions of the basic problem. If you are additionally interested in nonholonomic planning and other problems that involve differential constraints, then it is safe to jump ahead to Chapters 13–15, after completing Part II.

Chapters 11 and 12 cover problems in which there is sensing uncertainty. These problems live in an *information space*, which is detailed in Chapter 11. Chapter 12 covers algorithms that plan in the information space.

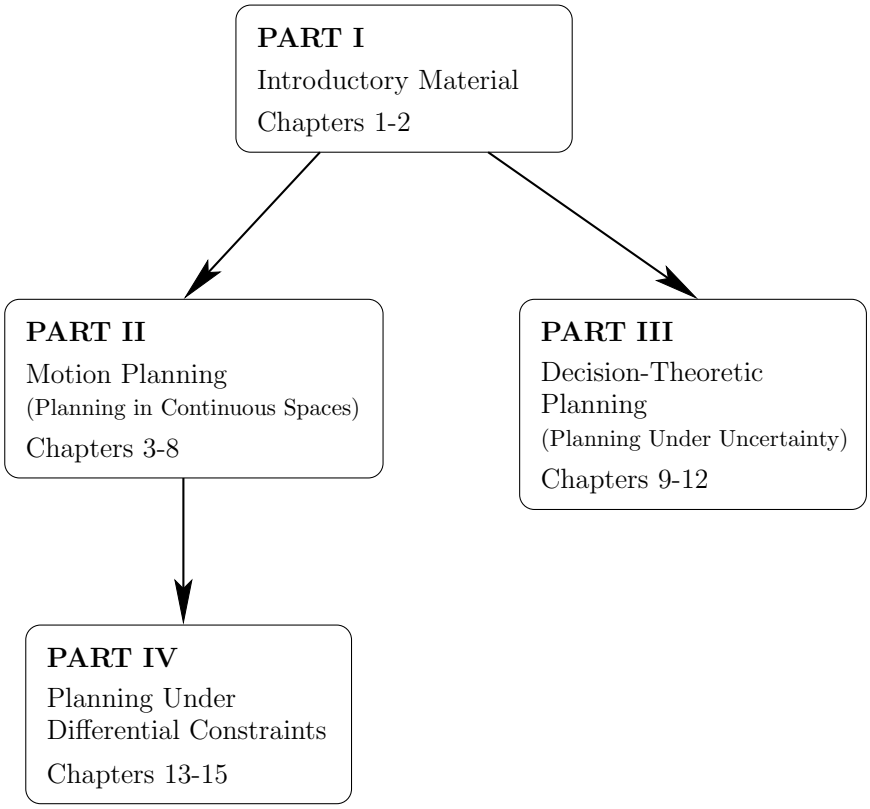


Figure 1: The dependencies between the four main parts of the book.

If you are interested mainly in decision-theoretic planning, then you can read Chapter 2 and then jump straight to Chapters 9–12. The material in these later chapters does not depend much on Chapters 3–8, which cover motion planning. Thus, if you are not interested in motion planning, the chapters may be easily skipped.

There are many ways to design a semester or quarter course from the book material. Figure 2 may help in deciding between core material and some optional topics. For an advanced undergraduate-level course, I recommend covering one core and some optional topics. For a graduate-level course, it may be possible to cover a couple of cores and some optional topics, depending on the initial background of the students. A two-semester sequence can also be developed by drawing material from all three cores and including some optional topics. Also, two independent courses can be made in a number of different ways. If you want to avoid continuous spaces, a course on discrete planning can be offered from Sections 2.1–2.5, 9.1–9.5, 10.1–10.5, 11.1–11.3, 11.7, and 12.1–12.3. If you are interested in teaching some game theory, there is roughly a chapter's worth of material in Sections 9.3–9.4, 10.5, 11.7, and 13.5. Material that contains the most prospects for future research appears in Chapters 7, 8, 11, 12, and 14. In particular, research on information spaces is still in its infancy.

Motion planning

Core:	2.1-2.2, 3.1-3.3, 4.1-4.3, 5.1-5.6, 6.1-6.3
Optional:	3.4-3.5, 4.4, 6.4-6.5, 7.1-7.7, 8.1-8.5

Planning under uncertainty

Core:	2.1-2.3, 9.1-9.2, 10.1-10.4, 11.1-11.6, 12.1-12.3
Optional:	9.3-9.5, 10.5-10.6, 11.7, 12.4-12.5

Planning under differential constraints

Core:	8.3, 13.1-13.3, 14.1-14.4, 15.1, 15.3-15.4
Optional:	13.4-13.5, 14.5-14.7, 15.2, 15.5

Figure 2: Based on Parts II, III, and IV, there are three themes of core material and optional topics.

To facilitate teaching, there are more than 500 examples and exercises throughout the book. The exercises in each chapter are divided into written problems and implementation projects. For motion planning projects, students often become bogged down with low-level implementation details. One possibility is to use the Motion Strategy Library (MSL):

http://msl.cs.uiuc.edu/msl/

as an object-oriented software base on which to develop projects. I have had great success with this for both graduate and undergraduate students.

For additional material, updates, and errata, see the Web page associated with this book:

http://planning.cs.uiuc.edu/

You may also download a free electronic copy of this book for your own personal use.

For further reading, consult the numerous references given at the end of chapters and throughout the text. Most can be found with a quick search of the Internet, but I did not give too many locations because these tend to be unstable over time. Unfortunately, the literature surveys are shorter than I had originally planned; thus, in some places, only a list of papers is given, which is often incomplete. I have tried to make the survey of material in this book as impartial as possible, but there is undoubtedly a bias in some places toward my own work. This was difficult to avoid because my research efforts have been closely intertwined with the development of this book.

Acknowledgments

I am very grateful to many students and colleagues who have given me extensive feedback and advice in developing this text. It evolved over many years through the development and teaching of courses at Stanford, Iowa State, and the University of Illinois. These universities have been very supportive of my efforts.

Many ideas and explanations throughout the book were inspired through numerous collaborations. For this reason, I am particularly grateful to the helpful insights and discussions that arose through collaborations with Michael Branicky, Francesco Bullo, Jeff Erickson, Emilio Frazzoli, Rob Ghrist, Leo Guibas, Seth Hutchinson, Lydia Kavraki, James Kuffner, Jean-Claude Latombe, Rajeev Motwani, Rafael Murrieta, Rajeev Sharma, Thierry Siméon, and Giora Slutzki. Over years of interaction, their ideas helped me to shape the perspective and presentation throughout the book.

Many valuable insights and observations were gained through collaborations with students, especially Peng Cheng, Hamid Chitsaz, Prashanth Konkimalla, Jason O'Kane, Steve Lindemann, Stjepan Rajko, Shai Sachs, Boris Simov, Benjamin Tovar, Jeff Yakey, Libo Yang, and Anna Yershova. I am grateful for the opportunities to work with them and appreciate their interaction as it helped to develop my own understanding and perspective.

While writing the text, at many times I recalled being strongly influenced by one or more technical discussions with colleagues. Undoubtedly, the following list is incomplete, but, nevertheless, I would like to thank the following colleagues for their helpful insights and stimulating discussions: Pankaj Agarwal, Srinivas Akella, Nancy Amato, Devin Balkcom, Tamer Başar, Antonio Bicchi, Robert Bohlin, Joel Burdick, Stefano Carpin, Howie Choset, Juan Cortés, Jerry Dejong, Bruce Donald, Ignacy Duleba, Mike Erdmann, Roland Geraerts, Malik Ghallab, Ken Goldberg, Pekka Isto, Vijay Kumar, Andrew Ladd, Jean-Paul Laumond, Kevin Lynch, Matt Mason, Pascal Morin, David Mount, Dana Nau, Jean Ponce, Mark Overmars, Elon Rimon, and Al Rizzi.

Many thanks go to Karl Bohringer, Marco Bressan, John Cassel, Stefano Carpin, Peng Cheng, Hamid Chitsaz, Ignacy Duleba, Claudia Esteves, Brian Gerkey, Ken Goldberg, Björn Hein, Sanjit Jhala, Marcelo Kallmann, James Kuffner, Olivier Lefebvre, Mong Leng, Steve Lindemann, Dennis Nieuwenhuisen, Jason O'Kane, Neil Petroff, Mihail Pivtoraiko, Stephane Redon, Gildardo Sanchez, Wiktor Schmidt, Fabian Schöfeld, Robin Schubert, Sanketh Shetty, Mohan Sirchabesan, James Solberg, Domenico Spensieri, Kristian Spoerer, Tony Stentz, Morten Strandberg, Ichiro Suzuki, Benjamin Tovar, Zbynek Winkler, Anna Yershova, Jingjin Yu, George Zaimes, and Liangjun Zhang for pointing out numerous mistakes in the on-line manuscript. I also appreciate the efforts of graduate students in my courses who scribed class notes that served as an early draft for some parts. These include students at Iowa State and the University of Illinois: Peng Cheng, Brian George, Shamsi Tamara Iqbal, Xiaolei Li, Steve Lindemann, Shai Sachs, Warren Shen, Rishi Talreja, Sherwin Tam, and Benjamin Tovar.

I sincerely thank Krzysztof Kozlowski and his staff, Joanna Gawecka, Wirginia Król, and Marek Lawniczak, at the Politechnika Poznańska (Technical University of Poznan) for all of their help and hospitality during my sabbatical in Poland. I also thank Heather Hall for managing my U.S.-based professional life while I lived in Europe. I am grateful to the National Science Foundation, the Office of

Naval Research, and DARPA for research grants that helped to support some of my sabbatical and summer time during the writing of this book. The Department of Computer Science at the University of Illinois was also very generous in its support of this huge effort.

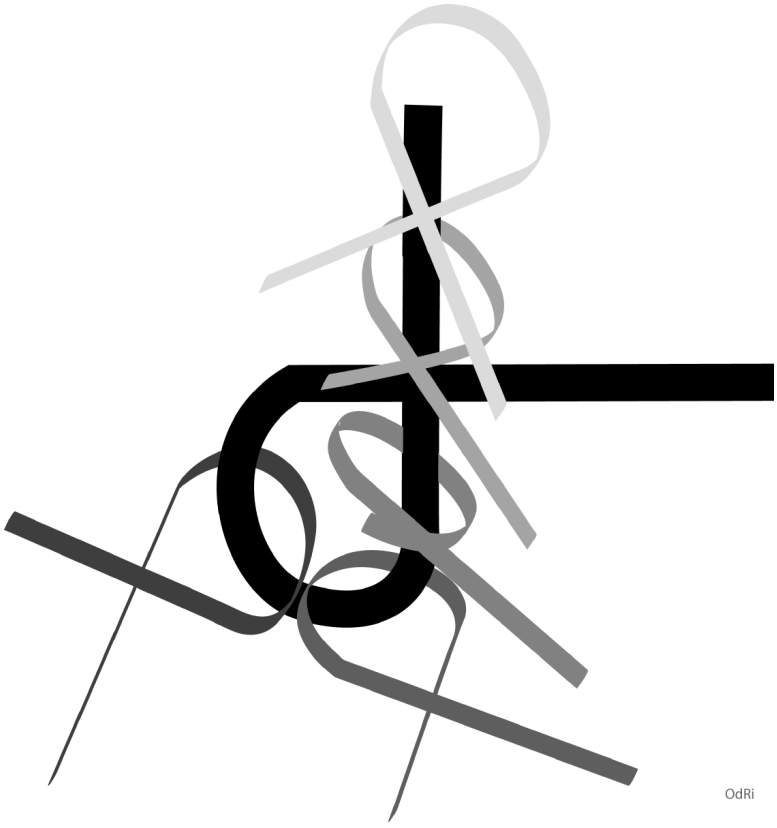
I am very fortunate to have artistically talented friends. I am deeply indebted to James Kuffner for creating the image on the front cover and to Audrey de Malmazet de Saint Andeol for creating the art on the first page of each of the four main parts.

Finally, I thank my editor, Lauren Cowles, my copy editor, Elise Oranges, and the rest of the people involved with Cambridge University Press for their efforts and advice in preparing the manuscript for publication.

Steve LaValle
Urbana, Illinois, U.S.A.

Part I

Introductory Material



Chapter 1

Introduction

1.1 Planning to Plan

Planning is a term that means different things to different groups of people. *Robotics* addresses the automation of mechanical systems that have sensing, actuation, and computation capabilities (similar terms, such as *autonomous systems* are also used). A fundamental need in robotics is to have algorithms that convert high-level specifications of tasks from humans into low-level descriptions of how to move. The terms *motion planning* and *trajectory planning* are often used for these kinds of problems. A classical version of motion planning is sometimes referred to as the *Piano Mover's Problem*. Imagine giving a precise computer-aided design (CAD) model of a house and a piano as input to an algorithm. The algorithm must determine how to move the piano from one room to another in the house without hitting anything. Most of us have encountered similar problems when moving a sofa or mattress up a set of stairs. Robot motion planning usually ignores dynamics and other differential constraints and focuses primarily on the translations and rotations required to move the piano. Recent work, however, does consider other aspects, such as uncertainties, differential constraints, modeling errors, and optimality. Trajectory planning usually refers to the problem of taking the solution from a robot motion planning algorithm and determining how to move along the solution in a way that respects the mechanical limitations of the robot.

Control theory has historically been concerned with designing inputs to physical systems described by differential equations. These could include mechanical systems such as cars or aircraft, electrical systems such as noise filters, or even systems arising in areas as diverse as chemistry, economics, and sociology. Classically, control theory has developed *feedback policies*, which enable an adaptive response during execution, and has focused on *stability*, which ensures that the dynamics do not cause the system to become wildly out of control. A large emphasis is also placed on optimizing criteria to minimize resource consumption, such as energy or time. In recent control theory literature, *motion planning* sometimes refers to the construction of inputs to a nonlinear dynamical system that drives it from an initial state to a specified goal state. For example, imagine trying to operate a

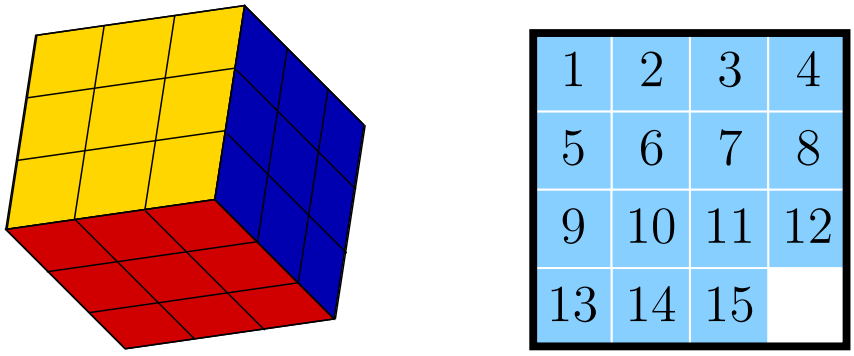
remote-controlled hovercraft that glides over the surface of a frozen pond. Suppose we would like the hovercraft to leave its current resting location and come to rest at another specified location. Can an algorithm be designed that computes the desired inputs, even in an ideal simulator that neglects uncertainties that arise from model inaccuracies? It is possible to add other considerations, such as uncertainties, feedback, and optimality; however, the problem is already challenging enough without these.

In *artificial intelligence*, the terms *planning* and *AI planning* take on a more discrete flavor. Instead of moving a piano through a continuous space, as in the robot motion planning problem, the task might be to solve a puzzle, such as the Rubik's cube or a sliding-tile puzzle, or to achieve a task that is modeled discretely, such as building a stack of blocks. Although such problems could be modeled with continuous spaces, it seems natural to define a finite set of actions that can be applied to a discrete set of states and to construct a solution by giving the appropriate sequence of actions. Historically, planning has been considered different from *problem solving*; however, the distinction seems to have faded away in recent years. In this book, we do not attempt to make a distinction between the two. Also, substantial effort has been devoted to representation language issues in planning. Although some of this will be covered, it is mainly outside of our focus. Many decision-theoretic ideas have recently been incorporated into the AI planning problem, to model uncertainties, adversarial scenarios, and optimization. These issues are important and are considered in detail in Part III.

Given the broad range of problems to which the term planning has been applied in the artificial intelligence, control theory, and robotics communities, you might wonder whether it has a specific meaning. Otherwise, just about anything could be considered as an instance of planning. Some common elements for planning problems will be discussed shortly, but first we consider planning as a branch of algorithms. Hence, this book is entitled *Planning Algorithms*. The primary focus is on algorithmic and computational issues of planning problems that have arisen in several disciplines. On the other hand, this does not mean that planning algorithms refers to an existing community of researchers within the general algorithms community. This book it not limited to combinatorics and asymptotic complexity analysis, which is the main focus in pure algorithms. The focus here includes numerous concepts that are not necessarily algorithmic but aid in modeling, solving, and analyzing planning problems.

Natural questions at this point are, What is a plan? How is a plan represented? How is it computed? What is it supposed to achieve? How is its quality evaluated? Who or what is going to use it? This chapter provides general answers to these questions. Regarding the user of the plan, it clearly depends on the application. In most applications, an algorithm executes the plan; however, the user could even be a human. Imagine, for example, that the planning algorithm provides you with an investment strategy.

In this book, the user of the plan will frequently be referred to as a *robot* or a *decision maker*. In artificial intelligence and related areas, it has become popular



(a) (b)

Figure 1.1: The Rubik's cube (a), sliding-tile puzzle (b), and other related puzzles are examples of discrete planning problems.

in recent years to use the term *agent*, possibly with adjectives to yield an *intelligent agent* or *software agent*. Control theory usually refers to the decision maker as a *controller*. The plan in this context is sometimes referred to as a *policy* or *control law*. In a game-theoretic context, it might make sense to refer to decision makers as *players*. Regardless of the terminology used in a particular discipline, this book is concerned with planning algorithms that find a strategy for one or more decision makers. Therefore, remember that terms such as *robot*, *agent*, and *controller* are interchangeable.

1.2 Motivational Examples and Applications

Planning problems abound. This section surveys several examples and applications to inspire you to read further.

Why study planning algorithms? There are at least two good reasons. First, it is fun to try to get machines to solve problems for which even humans have great difficulty. This involves exciting challenges in modeling planning problems, designing efficient algorithms, and developing robust implementations. Second, planning algorithms have achieved widespread successes in several industries and academic disciplines, including robotics, manufacturing, drug design, and aerospace applications. The rapid growth in recent years indicates that many more fascinating applications may be on the horizon. These are exciting times to study planning algorithms and contribute to their development and use.

Discrete puzzles, operations, and scheduling Chapter 2 covers discrete planning, which can be applied to solve familiar puzzles, such as those shown in Figure 1.1. They are also good at games such as chess or bridge [898]. Discrete planning techniques have been used in space applications, including a rover that traveled on Mars and the Earth Observing One satellite [207, 382, 896]. When

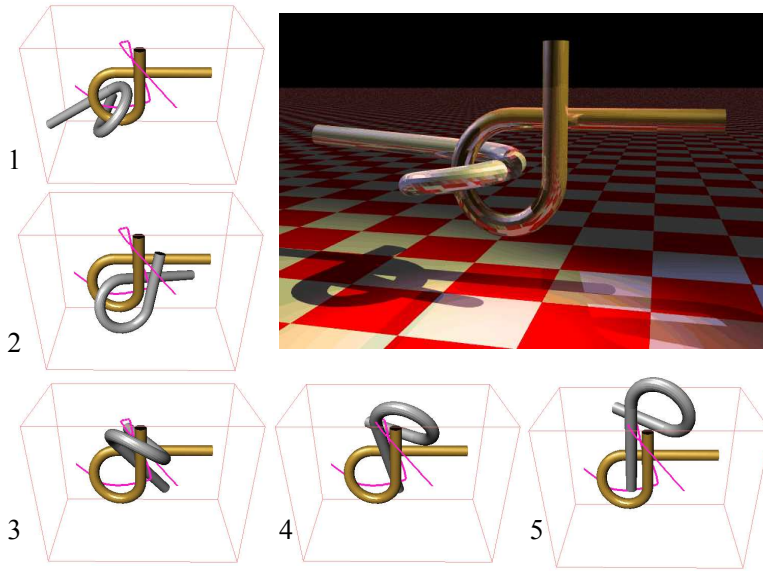


Figure 1.2: Remember puzzles like this? Imagine trying to solve one with an algorithm. The goal is to pull the two bars apart. This example is called the Alpha 1.0 Puzzle. It was created by Boris Yamrom and posted as a research benchmark by Nancy Amato at Texas A&M University. This solution and animation were made by James Kuffner (see [558] for the full movie).

combined with methods for planning in continuous spaces, they can solve complicated tasks such as determining how to bend sheet metal into complicated objects [419]; see Section 7.5 for the related problem of folding cartons.

A motion planning puzzle The puzzles in Figure 1.1 can be easily discretized because of the regularity and symmetries involved in moving the parts. Figure 1.2 shows a problem that lacks these properties and requires planning in a continuous space. Such problems are solved by using the motion planning techniques of Part II. This puzzle was designed to frustrate both humans and motion planning algorithms. It can be solved in a few minutes on a standard personal computer (PC) using the techniques in Section 5.5. Many other puzzles have been developed as benchmarks for evaluating planning algorithms.

An automotive assembly puzzle Although the problem in Figure 1.2 may appear to be pure fun and games, similar problems arise in important applications. For example, Figure 1.3 shows an automotive assembly problem for which software is needed to determine whether a wiper motor can be inserted (and removed) from the car body cavity. Traditionally, such a problem is solved by constructing physical models. This costly and time-consuming part of the design process can be virtually eliminated in software by directly manipulating the CAD models.

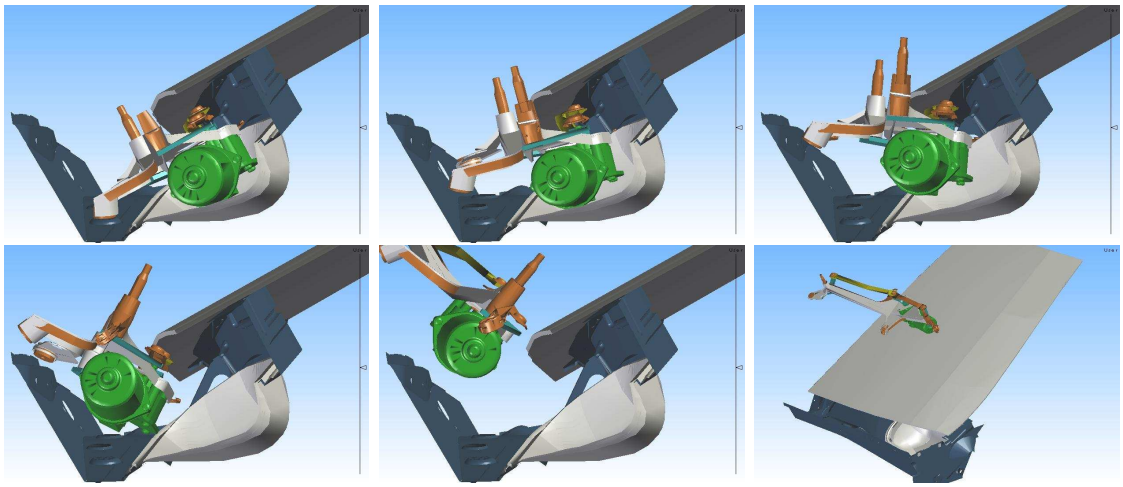


Figure 1.3: An automotive assembly task that involves inserting or removing a windshield wiper motor from a car body cavity. This problem was solved for clients using the motion planning software of Kineo CAM (courtesy of Kineo CAM).

The wiper example is just one of many. The most widespread impact on industry comes from motion planning software developed at Kineo CAM. It has been integrated into Robcad (eM-Workplace) from Tecnomatix, which is a leading tool for designing robotic workcells in numerous factories around the world. Their software has also been applied to assembly problems by Renault, Ford, Airbus, Optivus, and many other major corporations. Other companies and institutions are also heavily involved in developing and delivering motion planning tools for industry (many are secret projects, which unfortunately cannot be described here). One of the first instances of motion planning applied to real assembly problems is documented in [186].

Sealing cracks in automotive assembly Figure 1.4 shows a simulation of robots performing sealing at the Volvo Cars assembly plant in Torslanda, Sweden. Sealing is the process of using robots to spray a sticky substance along the seams of a car body to prevent dirt and water from entering and causing corrosion. The entire robot workcell is designed using CAD tools, which automatically provide the necessary geometric models for motion planning software. The solution shown in Figure 1.4 is one of many problems solved for Volvo Cars and others using motion planning software developed by the Fraunhofer Chalmers Centre (FCC). Using motion planning software, engineers need only specify the high-level task of performing the sealing, and the robot motions are computed automatically. This saves enormous time and expense in the manufacturing process.

Moving furniture Returning to pure entertainment, the problem shown in Figure 1.5 involves moving a grand piano across a room using three mobile robots with manipulation arms mounted on them. The problem is humorously inspired

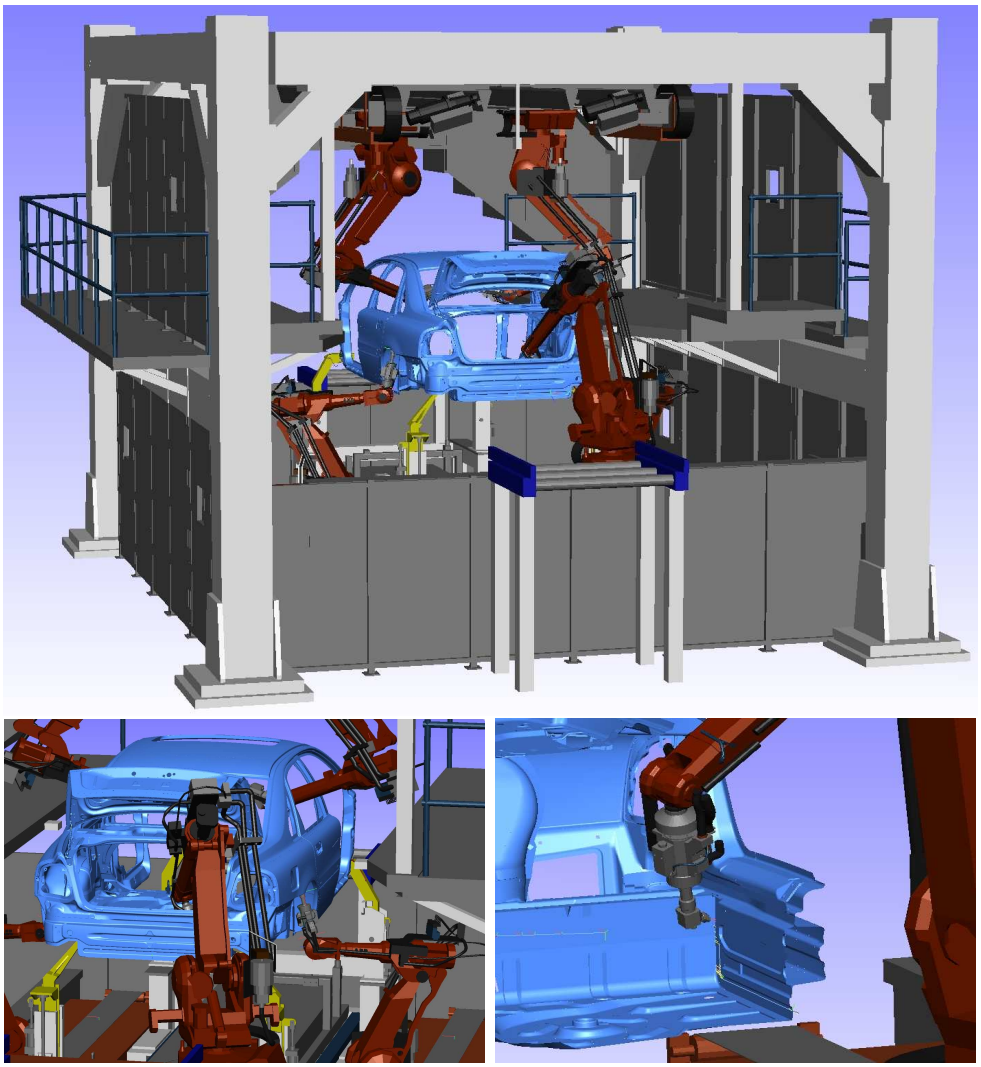


Figure 1.4: An application of motion planning to the sealing process in automotive manufacturing. Planning software developed by the Fraunhofer Chalmers Centre (FCC) is used at the Volvo Cars plant in Sweden (courtesy of Volvo Cars and FCC).

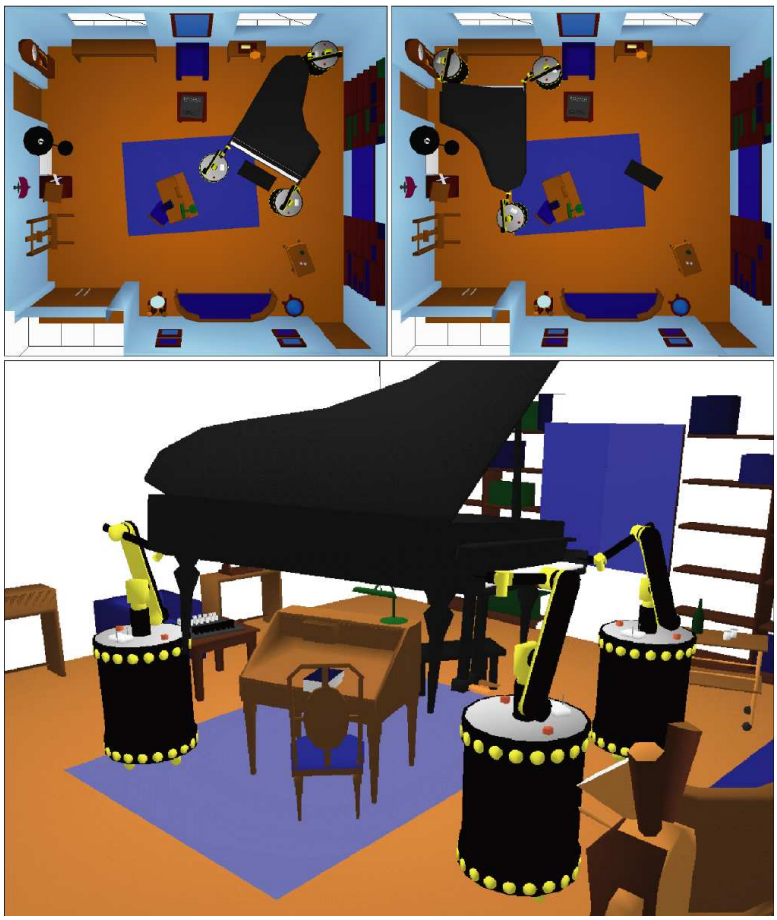
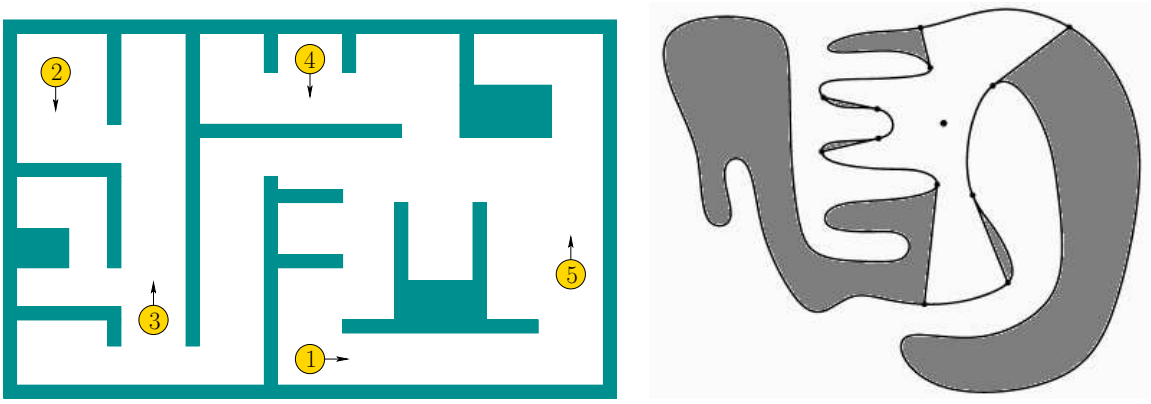


Figure 1.5: Using mobile robots to move a piano [244].

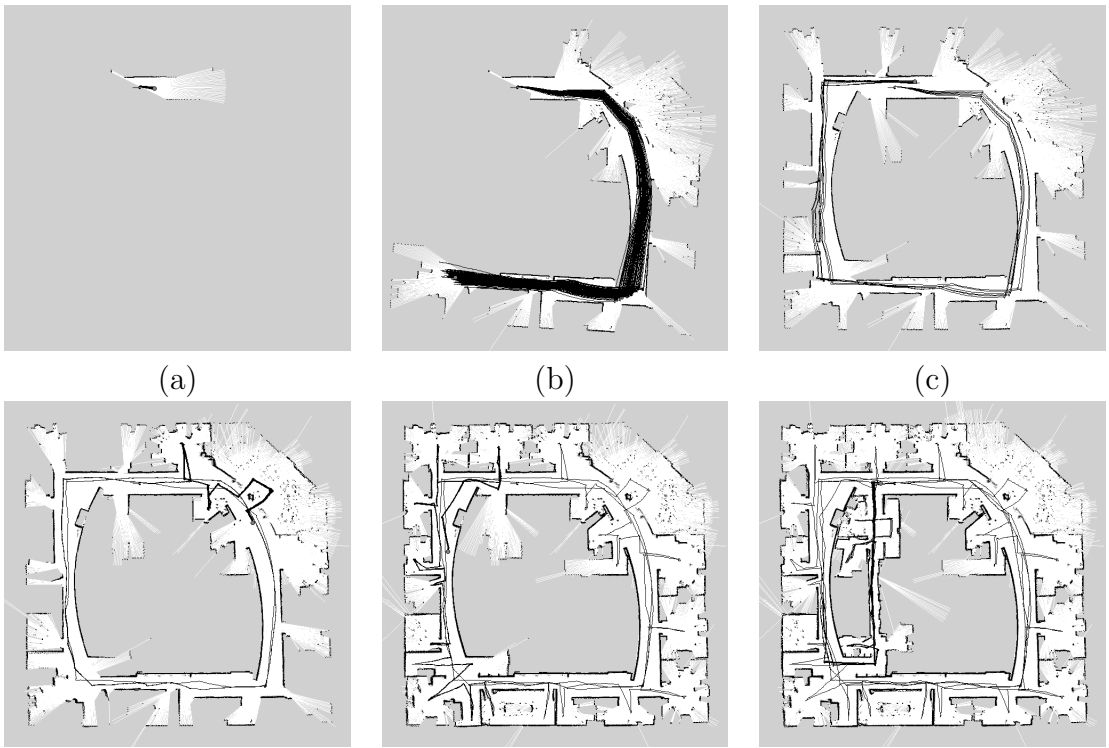
by the phrase *Piano Mover's Problem*. Collisions between robots and with other pieces of furniture must be avoided. The problem is further complicated because the robots, piano, and floor form closed kinematic chains, which are covered in Sections 4.4 and 7.4.

Navigating mobile robots A more common task for mobile robots is to request them to navigate in an indoor environment, as shown in Figure 1.6a. A robot might be asked to perform tasks such as building a map of the environment, determining its precise location within a map, or arriving at a particular place. Acquiring and manipulating information from sensors is quite challenging and is covered in Chapters 11 and 12. Most robots operate in spite of large uncertainties. At one extreme, it may appear that having many sensors is beneficial because it could allow precise estimation of the environment and the robot position and orientation. This is the premise of many existing systems, as shown for the robot system in Figure 1.7, which constructs a map of its environment. It may alternatively be preferable to develop low-cost and reliable robots that achieve specific tasks with little or no sensing. These trade-offs are carefully considered in Chapters 11 and



(a) (b)

Figure 1.6: (a) Several mobile robots attempt to successfully navigate in an indoor environment while avoiding collisions with the walls and each other. (b) Imagine using a lantern to search a cave for missing people.



(a) (b) (c)

(d) (e) (f)

Figure 1.7: A mobile robot can reliably construct a good map of its environment (here, the Intel Research Lab) while simultaneously localizing itself. This is accomplished using laser scanning sensors and performing efficient Bayesian computations on the information space [351].

12. Planning under uncertainty is the focus of Part III.

If there are multiple robots, then many additional issues arise. How can the robots communicate? How can their information be integrated? Should their coordination be centralized or distributed? How can collisions between them be avoided? Do they each achieve independent tasks, or are they required to collaborate in some way? If they are competing in some way, then concepts from game theory may apply. Therefore, some game theory appears in Sections 9.3, 9.4, 10.5, 11.7, and 13.5.

Playing hide and seek One important task for a mobile robot is playing the game of hide and seek. Imagine entering a cave in complete darkness. You are given a lantern and asked to search for any people who might be moving about, as shown in Figure 1.6b. Several questions might come to mind. Does a strategy even exist that guarantees I will find everyone? If not, then how many other searchers are needed before this task can be completed? Where should I move next? Can I keep from exploring the same places multiple times? This scenario arises in many robotics applications. The robots can be embedded in surveillance systems that use mobile robots with various types of sensors (motion, thermal, cameras, etc.). In scenarios that involve multiple robots with little or no communication, the strategy could help one robot locate others. One robot could even try to locate another that is malfunctioning. Outside of robotics, software tools can be developed that assist people in systematically searching or covering complicated environments, for applications such as law enforcement, search and rescue, toxic cleanup, and in the architectural design of secure buildings. The problem is extremely difficult because the status of the pursuit must be carefully computed to avoid unnecessarily allowing the evader to sneak back to places already searched. The information-space concepts of Chapter 11 become critical in solving the problem. For an algorithmic solution to the hide-and-seek game, see Section 12.4.

Making smart video game characters The problem in Figure 1.6b might remind you of a video game. In the arcade classic *Pacman*, the ghosts are programmed to seek the player. Modern video games involve human-like characters that exhibit much more sophisticated behavior. Planning algorithms can enable game developers to program character behaviors at a higher level, with the expectation that the character can determine on its own how to move in an intelligent way.

At present there is a large separation between the planning-algorithm and video-game communities. Some developers of planning algorithms are recently considering more of the particular concerns that are important in video games. Video-game developers have to invest too much energy at present to adapt existing techniques to their problems. For recent books that are geared for game developers, see [152, 371].

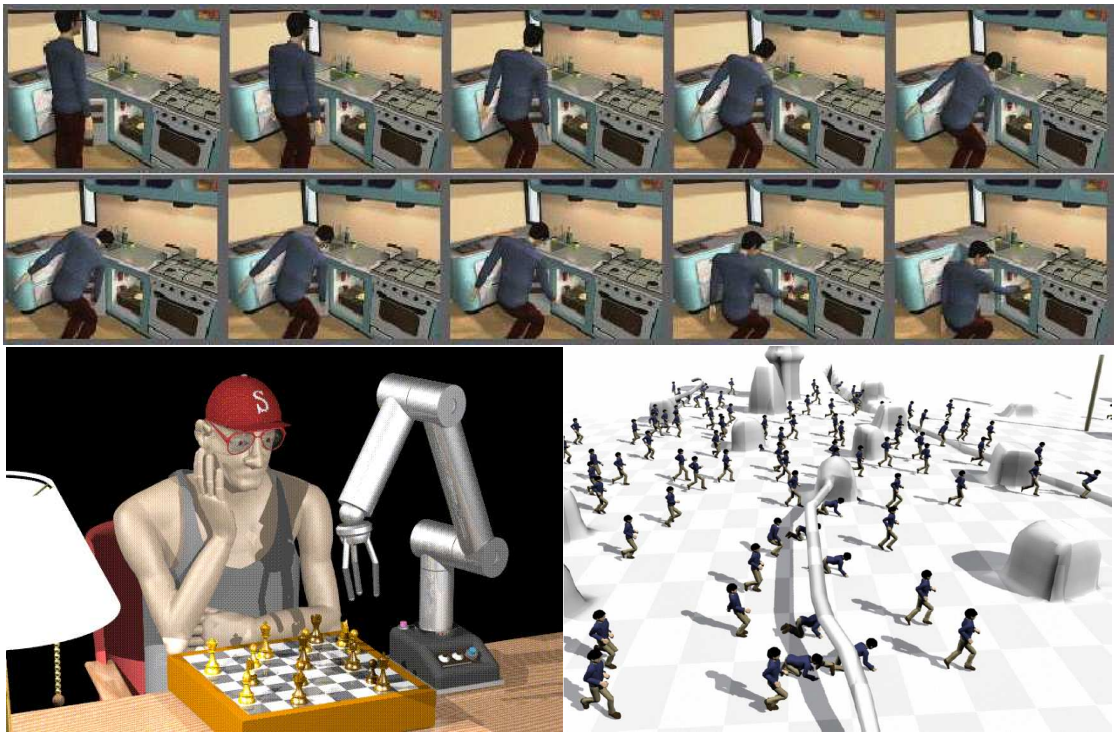
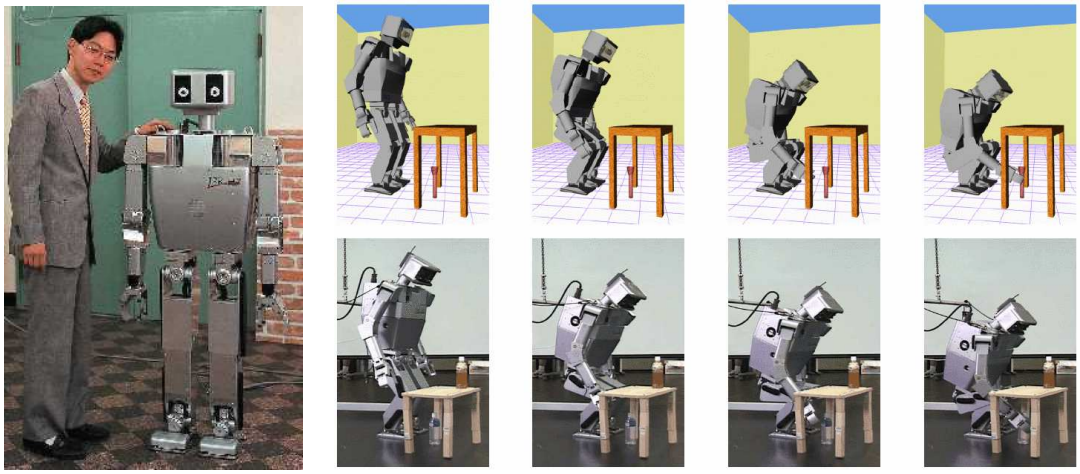


Figure 1.8: Across the top, a motion computed by a planning algorithm, for a digital actor to reach into a refrigerator [498]. In the lower left, a digital actor plays chess with a virtual robot [544]. In the lower right, a planning algorithm computes the motions of 100 digital actors moving across terrain with obstacles [591].

Virtual humans and humanoid robots Beyond video games, there is broader interest in developing virtual humans. See Figure 1.8. In the field of computer graphics, computer-generated animations are a primary focus. Animators would like to develop digital actors that maintain many elusive style characteristics of human actors while at the same time being able to design motions for them from high-level descriptions. It is extremely tedious and time consuming to specify all motions frame-by-frame. The development of planning algorithms in this context is rapidly expanding.

Why stop at *virtual* humans? The Japanese robotics community has inspired the world with its development of advanced humanoid robots. In 1997, Honda shocked the world by unveiling an impressive humanoid that could walk up stairs and recover from lost balance. Since that time, numerous corporations and institutions have improved humanoid designs. Although most of the mechanical issues have been worked out, two principle difficulties that remain are sensing and planning. What good is a humanoid robot if it cannot be programmed to accept high-level commands and execute them autonomously? Figure 1.9 shows work from the University of Tokyo for which a plan computed in simulation for a hu-



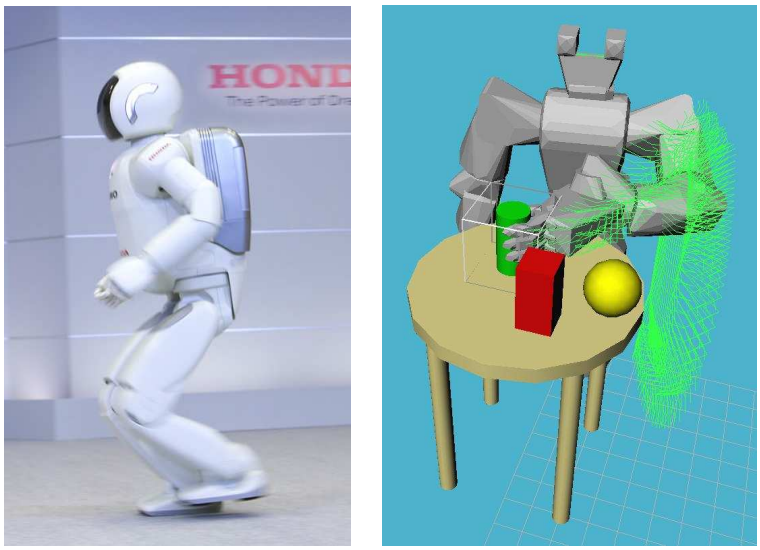
(a) (b)

Figure 1.9: (a) This is a picture of the H7 humanoid robot and one of its developers, S. Kagami. It was developed in the JSK Laboratory at the University of Tokyo. (b) Bringing virtual reality and physical reality together. A planning algorithm computes stable motions for a humanoid to grab an obstructed object on the floor [561].

manoid robot is actually applied on a real humanoid. Figure 1.10 shows humanoid projects from the Japanese automotive industry.

Parking cars and trailers The planning problems discussed so far have not involved differential constraints, which are the main focus in Part IV. Consider the problem of parking slow-moving vehicles, as shown in Figure 1.11. Most people have a little difficulty with parallel parking a car and much greater difficulty parking a truck with a trailer. Imagine the difficulty of parallel parking an airport baggage train! See Chapter 13 for many related examples. What makes these problems so challenging? A car is constrained to move in the direction that the rear wheels are pointing. Maneuvering the car around obstacles therefore becomes challenging. If all four wheels could turn to any orientation, this problem would vanish. The term *nonholonomic planning* encompasses parking problems and many others. Figure 1.12a shows a humorous driving problem. Figure 1.12b shows an extremely complicated vehicle for which nonholonomic planning algorithms were developed and applied in industry.

"Wreckless" driving Now consider driving the car at high speeds. As the speed increases, the car must be treated as a dynamical system due to momentum. The car is no longer able to instantaneously start and stop, which was reasonable for parking problems. Although there exist planning algorithms that address such issues, there are still many unsolved research problems. The impact on industry

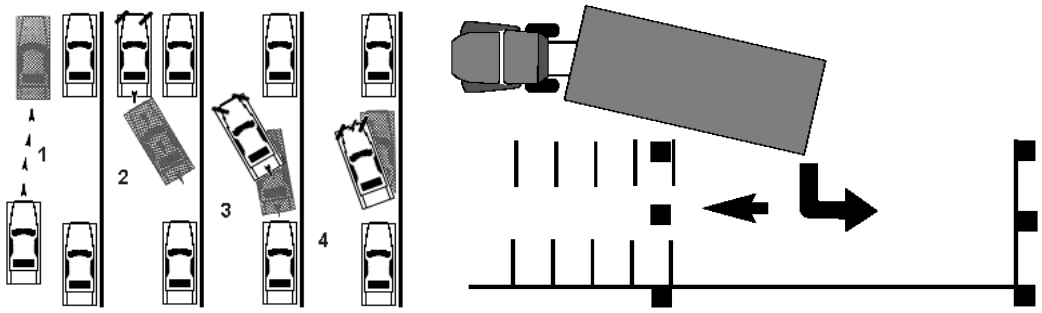


(a) (b)

Figure 1.10: Humanoid robots from the Japanese automotive industry: (a) The latest Asimo robot from Honda can run at 3 km/hr (courtesy of Honda); (b) planning is incorporated with vision in the Toyota humanoid so that it plans to grasp objects [448].

has not yet reached the level achieved by ordinary motion planning, as shown in Figures 1.3 and 1.4. By considering dynamics in the design process, performance and safety evaluations can be performed before constructing the vehicle. Figure 1.13 shows a solution computed by a planning algorithm that determines how to steer a car at high speeds through a town while avoiding collisions with buildings. A planning algorithm could even be used to assess whether a sports utility vehicle tumbles sideways when stopping too quickly. Tremendous time and costs can be spared by determining design flaws early in the development process via simulations and planning. One related problem is *verification*, in which a mechanical system design must be thoroughly tested to make sure that it performs as expected in spite of all possible problems that could go wrong during its use. Planning algorithms can also help in this process. For example, the algorithm can try to violently crash a vehicle, thereby establishing that a better design is needed.

Aside from aiding in the design process, planning algorithms that consider dynamics can be directly embedded into robotic systems. Figure 1.13b shows an application that involves a difficult combination of most of the issues mentioned so far. Driving across rugged, unknown terrain at high speeds involves dynamics, uncertainties, and obstacle avoidance. Numerous unsolved research problems remain in this context.



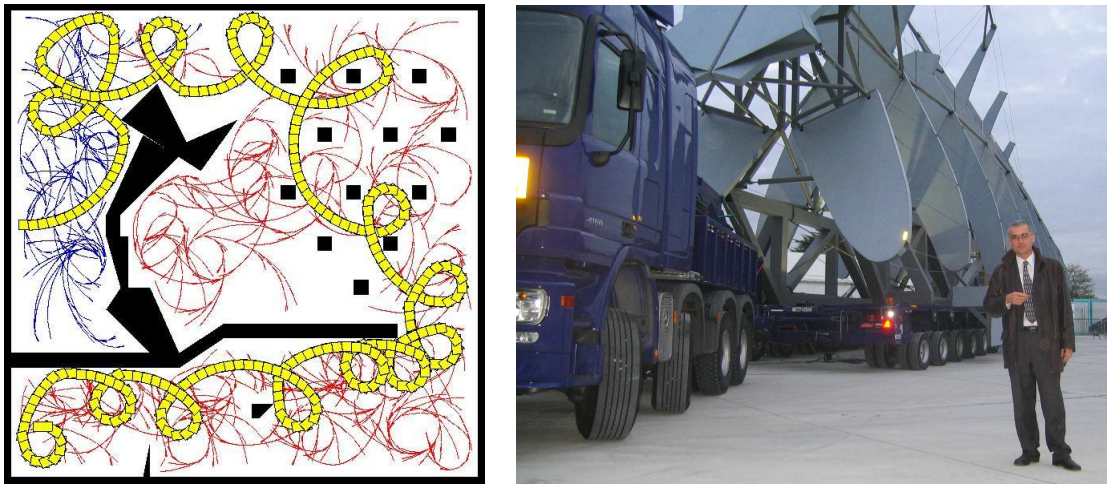
(a) (b)

Figure 1.11: Some parking illustrations from government manuals for driver testing: (a) parking a car (from the 2005 *Missouri Driver Guide*); (b) parking a tractor trailer (published by the Pennsylvania Division of Motor Vehicles). Both humans and planning algorithms can solve these problems.

Flying Through the Air or in Space Driving naturally leads to flying. Planning algorithms can help to navigate autonomous helicopters through obstacles. They can also compute thrusts for a spacecraft so that collisions are avoided around a complicated structure, such as a space station. In Section 14.1.3, the problem of designing entry trajectories for a reusable spacecraft is described. Mission planning for interplanetary spacecraft, including solar sails, can even be performed using planning algorithms [436].

Designing better drugs Planning algorithms are even impacting fields as far away from robotics as computational biology. Two major problems are protein folding and drug design. In both cases, scientists attempt to explain behaviors in organisms by the way large organic molecules interact. Such molecules are generally flexible. Drug molecules are small (see Figure 1.14), and proteins usually have thousands of atoms. The *docking problem* involves determining whether a flexible molecule can insert itself into a protein cavity, as shown in Figure 1.14, while satisfying other constraints, such as maintaining low energy. Once geometric models are applied to molecules, the problem looks very similar to the assembly problem in Figure 1.3 and can be solved by motion planning algorithms. See Section 7.5 and the literature at the end of Chapter 7.

Perspective Planning algorithms have been applied to many more problems than those shown here. In some cases, the work has progressed from modeling, to theoretical algorithms, to practical software that is used in industry. In other cases, substantial research remains to bring planning methods to their full potential. The future holds tremendous excitement for those who participate in the development and application of planning algorithms.



(a) (b)

Figure 1.12: (a) Having a little fun with differential constraints. An obstacle-avoiding path is shown for a car that must move forward and can only turn left. Could you have found such a solution on your own? This is an easy problem for several planning algorithms. (b) This gigantic truck was designed to transport portions of the Airbus A380 across France. Kineo CAM developed nonholonomic planning software that plans routes through villages that avoid obstacles and satisfy differential constraints imposed by 20 steering axles. Jean-Paul Laumond, a pioneer of nonholonomic planning, is also pictured.

(a) (b)

Figure 1.13: Reckless driving: (a) Using a planning algorithm to drive a car quickly through an obstacle course [199]. (b) A contender developed by the Red Team from Carnegie Mellon University in the DARPA Grand Challenge for autonomous vehicles driving at high speeds over rugged terrain (courtesy of the Red Team).

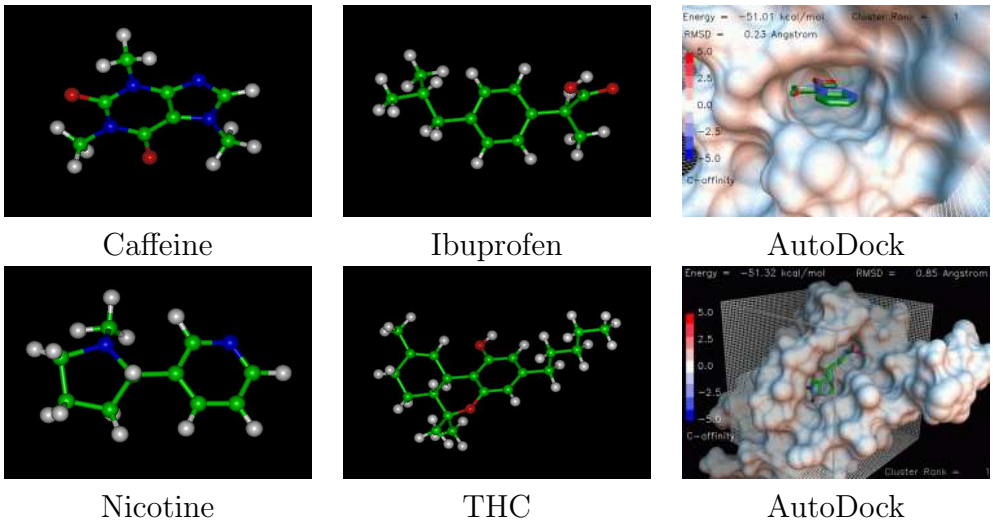


Figure 1.14: On the left, several familiar drugs are pictured as ball-and-stick models (courtesy of the New York University MathMol Library [734]). On the right, 3D models of protein-ligand docking are shown from the AutoDock software package (courtesy of the Scripps Research Institute).

1.3 Basic Ingredients of Planning

Although the subject of this book spans a broad class of models and problems, there are several basic ingredients that arise throughout virtually all of the topics covered as part of planning.

State Planning problems involve a *state space* that captures all possible situations that could arise. The *state* could, for example, represent the position and orientation of a robot, the locations of tiles in a puzzle, or the position and velocity of a helicopter. Both discrete (finite, or countably infinite) and continuous (uncountably infinite) state spaces will be allowed. One recurring theme is that the state space is usually represented *implicitly* by a planning algorithm. In most applications, the size of the state space (in terms of number of states or combinatorial complexity) is much too large to be explicitly represented. Nevertheless, the definition of the state space is an important component in the formulation of a planning problem and in the design and analysis of algorithms that solve it.

Time All planning problems involve a sequence of decisions that must be applied over time. Time might be explicitly modeled, as in a problem such as driving a car as quickly as possible through an obstacle course. Alternatively, time may be implicit, by simply reflecting the fact that actions must follow in succession, as in the case of solving the Rubik's cube. The particular time is unimportant, but the proper sequence must be maintained. Another example of implicit time is a

solution to the Piano Mover's Problem; the solution to moving the piano may be converted into an animation over time, but the particular speed is not specified in the plan. As in the case of state spaces, time may be either discrete or continuous. In the latter case, imagine that a continuum of decisions is being made by a plan.

Actions A plan generates *actions* that manipulate the state. The terms *actions* and *operators* are common in artificial intelligence; in control theory and robotics, the related terms are *inputs* and *controls*. Somewhere in the planning formulation, it must be specified how the state changes when actions are applied. This may be expressed as a state-valued function for the case of discrete time or as an ordinary differential equation for continuous time. For most motion planning problems, explicit reference to time is avoided by directly specifying a path through a continuous state space. Such paths could be obtained as the integral of differential equations, but this is not necessary. For some problems, actions could be chosen by *nature*, which interfere with the outcome and are not under the control of the decision maker. This enables uncertainty in predictability to be introduced into the planning problem; see Chapter 10.

Initial and goal states A planning problem usually involves starting in some initial state and trying to arrive at a specified goal state or any state in a set of goal states. The actions are selected in a way that tries to make this happen.

A criterion This encodes the desired outcome of a plan in terms of the state and actions that are executed. There are generally two different kinds of planning concerns based on the type of criterion:

1. **Feasibility:** Find a plan that causes arrival at a goal state, regardless of its efficiency.

2. **Optimality:** Find a feasible plan that optimizes performance in some carefully specified manner, in addition to arriving in a goal state.

For most of the problems considered in this book, feasibility is already challenging enough; achieving optimality is considerably harder for most problems. Therefore, much of the focus is on finding feasible solutions to problems, as opposed to optimal solutions. The majority of literature in robotics, control theory, and related fields focuses on optimality, but this is not necessarily important for many problems of interest. In many applications, it is difficult to even formulate the right criterion to optimize. Even if a desirable criterion can be formulated, it may be impossible to obtain a practical algorithm that computes optimal plans. In such cases, feasible solutions are certainly preferable to having no solutions at all. Fortunately, for many algorithms the solutions produced are not too far from optimal in practice. This reduces some of the motivation for finding optimal solutions. For problems that involve probabilistic uncertainty, however, optimization arises

more frequently. The probabilities are often utilized to obtain the best performance in terms of expected costs. Feasibility is often associated with performing a worst-case analysis of uncertainties.

A plan In general, a plan imposes a specific strategy or behavior on a decision maker. A plan may simply specify a sequence of actions to be taken; however, it could be more complicated. If it is impossible to predict future states, then the plan can specify actions as a function of state. In this case, regardless of the future states, the appropriate action is determined. Using terminology from other fields, this enables *feedback* or *reactive plans*. It might even be the case that the state cannot be measured. In this case, the appropriate action must be determined from whatever information is available up to the current time. This will generally be referred to as an *information state*, on which the actions of a plan are conditioned.

1.4 Algorithms, Planners, and Plans

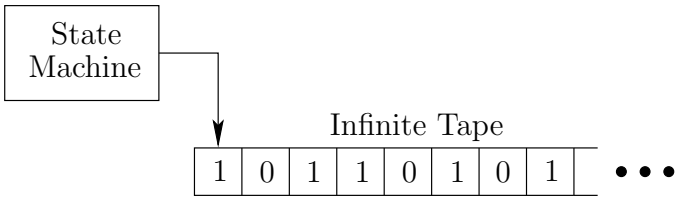


Figure 1.15: According to the Church-Turing thesis, the notion of an algorithm is equivalent to the notion of a Turing machine.

1.4.1 Algorithms

What is a planning algorithm? This is a difficult question, and a precise mathematical definition will not be given in this book. Instead, the general idea will be explained, along with many examples of planning algorithms. A more basic question is, What is an algorithm? One answer is the classical Turing machine model, which is used to define an algorithm in theoretical computer science. A *Turing machine* is a finite state machine with a special head that can read and write along an infinite piece of tape, as depicted in Figure 1.15. The Church-Turing thesis states that an algorithm *is* a Turing machine (see [462, 891] for more details). The *input* to the algorithm is encoded as a string of symbols (usually a binary string) and then is written to the tape. The Turing machine reads the string, performs computations, and then decides whether to *accept* or *reject* the string. This version of the Turing machine only solves *decision problems*; however, there are straightforward extensions that can yield other desired outputs, such as a plan.

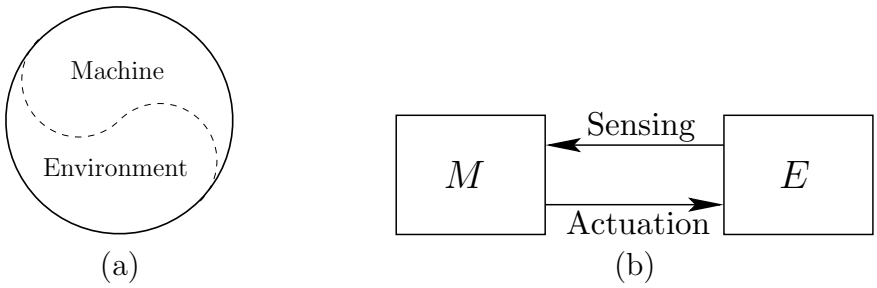


Figure 1.16: (a) The boundary between machine and environment is considered as an arbitrary line that may be drawn in many ways depending on the context. (b) Once the boundary has been drawn, it is assumed that the machine, M, interacts with the environment, E, through sensing and actuation.

The Turing model is reasonable for many of the algorithms in this book; however, others may not exactly fit. The trouble with using the Turing machine in some situations is that plans often interact with the physical world. As indicated in Figure 1.16, the boundary between the machine and the environment is an arbitrary line that varies from problem to problem. Once drawn, *sensors* provide information about the environment; this provides input to the machine during execution. The machine then executes actions, which provides *actuation* to the environment. The actuation may alter the environment in some way that is later measured by sensors. Therefore, the machine and its environment are closely coupled during execution. This is fundamental to robotics and many other fields in which planning is used.

Using the Turing machine as a foundation for algorithms usually implies that the physical world must be first carefully modeled and written on the tape before the algorithm can make decisions. If changes occur in the world during execution of the algorithm, then it is not clear what should happen. For example, a mobile robot could be moving in a cluttered environment in which people are walking around. As another example, a robot might throw an object onto a table without being able to precisely predict how the object will come to rest. It can take measurements of the results with sensors, but it again becomes a difficult task to determine how much information should be explicitly modeled and written on the tape. The *on-line algorithm* model is more appropriate for these kinds of problems [510, 768, 892]; however, it still does not capture a notion of algorithms that is broad enough for all of the topics of this book.

Processes that occur in a physical world are more complicated than the interaction between a state machine and a piece of tape filled with symbols. It is even possible to simulate the tape by imagining a robot that interacts with a long row of switches as depicted in Figure 1.17. The switches serve the same purpose as the tape, and the robot carries a computer that can simulate the finite state machine.[1]

[1]Of course, having infinitely long tape seems impossible in the physical world. Other versions

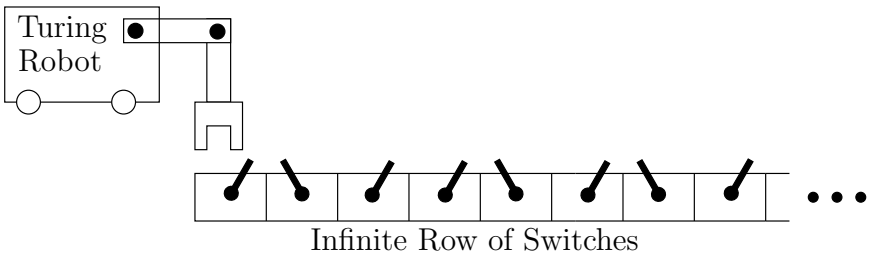


Figure 1.17: A robot and an infinite sequence of switches could be used to simulate a Turing machine. Through manipulation, however, many other kinds of behavior could be obtained that fall outside of the Turing model.

The complicated interaction allowed between a robot and its environment could give rise to many other models of computation.[2] Thus, the term *algorithm* will be used somewhat less formally than in the theory of computation. Both *planners* and *plans* are considered as algorithms in this book.

1.4.2 Planners

A planner simply constructs a plan and may be a machine or a human. If the planner is a machine, it will generally be considered as a planning algorithm. In many circumstances it is an algorithm in the strict Turing sense; however, this is not necessary. In some cases, humans become planners by developing a plan that works in all situations. For example, it is perfectly acceptable for a human to design a state machine that is connected to the environment (see Section 12.3.1). There are no additional inputs in this case because the human fulfills the role of the algorithm. The planning model is given as input to the human, and the human "computes" a plan.

1.4.3 Plans

Once a plan is determined, there are three ways to use it:

1. **Execution:** Execute it either in simulation or in a mechanical device (robot) connected to the physical world.

2. **Refinement:** Refine it into a better plan.

3. **Hierarchical Inclusion:** Package it as an action in a higher level plan.

Each of these will be explained in succession.

---

of Turing machines exist in which the tape is finite but as long as necessary to process the given input. This may be more appropriate for the discussion.

[2]Performing computations with mechanical systems is discussed in [815]. Computation models over the reals are covered in [118].

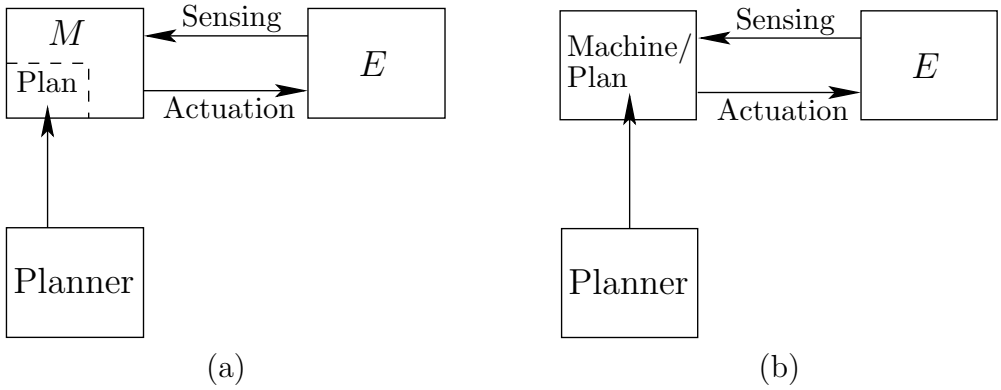


Figure 1.18: (a) A planner produces a plan that may be executed by the machine. The planner may either be a machine itself or even a human. (b) Alternatively, the planner may design the entire machine.

Execution A plan is usually executed by a machine. A human could alternatively execute it; however, the case of machine execution is the primary focus of this book. There are two general types of machine execution. The first is depicted in Figure 1.18a, in which the planner produces a *plan*, which is encoded in some way and given as input to the machine. In this case, the machine is considered *programmable* and can accept possible plans from a planner before execution. It will generally be assumed that once the plan is given, the machine becomes autonomous and can no longer interact with the planner. Of course, this model could be extended to allow machines to be improved over time by receiving better plans; however, we want a strict notion of autonomy for the discussion of planning in this book. This approach does not prohibit the updating of plans in practice; however, this is not preferred because plans should already be designed to take into account new information during execution.

The second type of machine execution of a plan is depicted in Figure 1.18b. In this case, the plan produced by the planner encodes an entire machine. The plan *is* a special-purpose machine that is designed to solve the specific tasks given originally to the planner. Under this interpretation, one may be a *minimalist* and design the simplest machine possible that sufficiently solves the desired tasks. If the plan is encoded as a finite state machine, then it can sometimes be considered as an algorithm in the Turing sense (depending on whether connecting the machine to a tape preserves its operation).

Refinement If a plan is used for refinement, then a planner accepts it as input and determines a new plan that is hopefully an improvement. The new plan may take more problem aspects into account, or it may simply be more efficient. Refinement may be applied repeatedly, to produce a sequence of improved plans, until the final one is executed. Figure 1.19 shows a refinement approach used in robotics. Consider, for example, moving an indoor mobile robot. The first

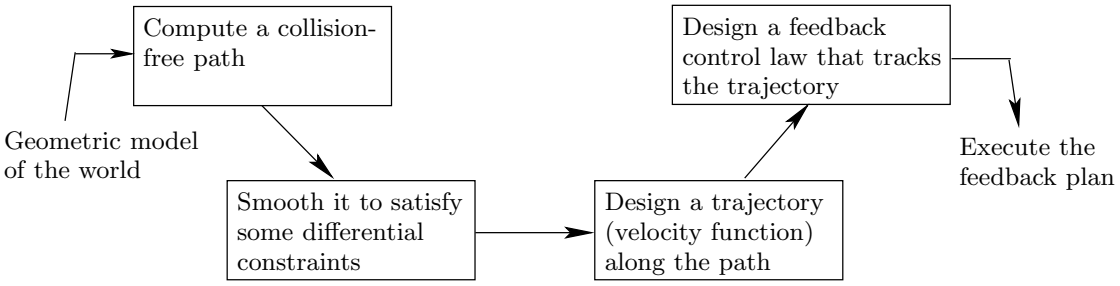


Figure 1.19: A refinement approach that has been used for decades in robotics.

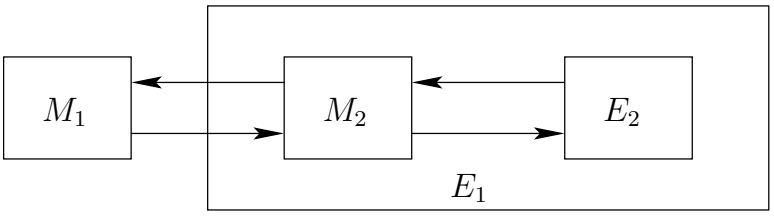


Figure 1.20: In a hierarchical model, the environment of one machine may itself contain a machine.

plan yields a collision-free path through the building. The second plan transforms the route into one that satisfies differential constraints based on wheel motions (recall Figure 1.11). The third plan considers how to move the robot along the path at various speeds while satisfying momentum considerations. The fourth plan incorporates feedback to ensure that the robot stays as close as possible to the planned path in spite of unpredictable behavior. Further elaboration on this approach and its trade-offs appears in Section 14.6.1.

Hierarchical inclusion Under hierarchical inclusion, a plan is incorporated as an action in a larger plan. The original plan can be imagined as a subroutine in the larger plan. For this to succeed, it is important for the original plan to guarantee *termination*, so that the larger plan can execute more actions as needed. Hierarchical inclusion can be performed any number of times, resulting in a rooted *tree* of plans. This leads to a general model of *hierarchical planning*. Each vertex in the tree is a plan. The root vertex represents the *master plan*. The children of any vertex are plans that are incorporated as actions in the plan of the vertex. There is no limit to the tree depth or number of children per vertex. In hierarchical planning, the line between machine and environment is drawn in multiple places. For example, the environment, E_1, with respect to a machine, M_1, might actually include another machine, M_2, that interacts with its environment, E_2, as depicted in Figure 1.20. Examples of hierarchical planning appear in Sections 7.3.2 and 12.5.1.

1.5 Organization of the Book

Here is a brief overview of the book. See also the overviews at the beginning of Parts II–IV.

PART I: Introductory Material

This provides very basic background for the rest of the book.

- **Chapter 1: Introductory Material**
  This chapter offers some general perspective and includes some motivational examples and applications of planning algorithms.

- **Chapter 2: Discrete Planning**
  This chapter covers the simplest form of planning and can be considered as a springboard for entering into the rest of the book. From here, you can continue to Part II, or even head straight to Part III. Sections 2.1 and 2.2 are most important for heading into Part II. For Part III, Section 2.3 is additionally useful.

PART II: Motion Planning

The main source of inspiration for the problems and algorithms covered in this part is robotics. The methods, however, are general enough for use in other applications in other areas, such as computational biology, computer-aided design, and computer graphics. An alternative title that more accurately reflects the kind of planning that occurs is "Planning in Continuous State Spaces."

- **Chapter 3: Geometric Representations and Transformations**
  The chapter gives important background for expressing a motion planning problem. Section 3.1 describes how to construct geometric models, and the remaining sections indicate how to transform them. Sections 3.1 and 3.2 are important for later chapters.

- **Chapter 4: The Configuration Space**
  This chapter introduces concepts from topology and uses them to formulate the *configuration space*, which is the state space that arises in motion planning. Sections 4.1, 4.2, and 4.3.1 are important for understanding most of the material in later chapters. In addition to the previously mentioned sections, all of Section 4.3 provides useful background for the combinatorial methods of Chapter 6.

- **Chapter 5: Sampling-Based Motion Planning**
  This chapter introduces motion planning algorithms that have dominated the literature in recent years and have been applied in fields both in and out of robotics. If you understand the basic idea that the configuration space represents a continuous state space, most of the concepts should be understandable. They even apply to other problems in which continuous state spaces emerge, in addition to motion planning and robotics. Chapter 14 revisits sampling-based planning, but under differential constraints.

- **Chapter 6: Combinatorial Motion Planning**
  The algorithms covered in this section are sometimes called *exact algorithms* because they build discrete representations without losing any information. They are *complete*, which means that they must find a solution if one exists; otherwise, they report failure. The sampling-based algorithms have been more useful in practice, but they only achieve weaker notions of completeness.

- **Chapter 7: Extensions of Basic Motion Planning**
  This chapter introduces many problems and algorithms that are extensions of the methods from Chapters 5 and 6. Most can be followed with basic understanding of the material from these chapters. Section 7.4 covers planning for closed kinematic chains; this requires an understanding of the additional material, from Section 4.4

- **Chapter 8: Feedback Motion Planning**
  This is a transitional chapter that introduces feedback into the motion planning problem but still does not introduce differential constraints, which are deferred until Part IV. The previous chapters of Part II focused on computing *open-loop* plans, which means that any errors that might occur during execution of the plan are ignored, yet the plan will be executed as planned. Using feedback yields a *closed-loop* plan that responds to unpredictable events during execution.

PART III: Decision-Theoretic Planning

An alternative title to Part III is "Planning Under Uncertainty." Most of Part III addresses discrete state spaces, which can be studied immediately following Part I. However, some sections cover extensions to continuous spaces; to understand these parts, it will be helpful to have read some of Part II.

- **Chapter 9: Basic Decision Theory**
  The main idea in this chapter is to design the best decision for a decision maker that is confronted with interference from other decision makers. The others may be true opponents in a game or may be fictitious in order to model uncertainties. The chapter focuses on making a decision in a single step and provides a building block for Part III because planning under uncertainty can be considered as multi-step decision making.

- **Chapter 10: Sequential Decision Theory**
  This chapter takes the concepts from Chapter 9 and extends them by chaining together a sequence of basic decision-making problems. Dynamic programming concepts from Section 2.3 become important here. For all of the problems in this chapter, it is assumed that the current state is always known. All uncertainties that exist are with respect to prediction of future states, as opposed to measuring the current state.

- **Chapter 11: Sensors and Information Spaces**
  The chapter extends the formulations of Chapter 10 into a framework for planning when the current state is unknown during execution. Information regarding the state is obtained from sensor observations and the memory of actions that were previously applied. The information space serves a similar purpose for problems with sensing uncertainty as the configuration space has for motion planning.

- **Chapter 12: Planning Under Sensing Uncertainty**
  This chapter covers several planning problems and algorithms that involve sensing uncertainty. This includes problems such as localization, map building, pursuit-evasion, and manipulation. All of these problems are unified under the idea of planning in information spaces, which follows from Chapter 11.

PART IV: Planning Under Differential Constraints

This can be considered as a continuation of Part II. Here there can be both global (obstacles) and local (differential) constraints on the continuous state spaces that arise in motion planning. Dynamical systems are also considered, which yields state spaces that include both position and velocity information (this coincides with the notion of a *state space* in control theory or a *phase space* in physics and differential equations).

- **Chapter 13: Differential Models**
  This chapter serves as an introduction to Part IV by introducing numerous models that involve differential constraints. This includes constraints that arise from wheels rolling as well as some that arise from the dynamics of mechanical systems.

- **Chapter 14: Sampling-Based Planning Under Differential Constraints**
  Algorithms for solving planning problems under the models of Chapter 13 are presented. Many algorithms are extensions of methods from Chapter 5. All methods are sampling-based because very little can be accomplished with combinatorial techniques in the context of differential constraints.

- **Chapter 15: System Theory and Analytical Techniques**
  This chapter provides an overview of the concepts and tools developed mainly in control theory literature. They are complementary to the algorithms of Chapter 14 and often provide important insights or components in the development of planning algorithms under differential constraints.

Chapter 2

Discrete Planning

This chapter provides introductory concepts that serve as an entry point into other parts of the book. The planning problems considered here are the simplest to describe because the state space will be finite in most cases. When it is not finite, it will at least be countably infinite (i.e., a unique integer may be assigned to every state). Therefore, no geometric models or differential equations will be needed to characterize the discrete planning problems. Furthermore, no forms of uncertainty will be considered, which avoids complications such as probability theory. All models are completely known and predictable.

There are three main parts to this chapter. Sections 2.1 and 2.2 define and present search methods for feasible planning, in which the only concern is to reach a goal state. The search methods will be used throughout the book in numerous other contexts, including motion planning in continuous state spaces. Following feasible planning, Section 2.3 addresses the problem of optimal planning. The *principle of optimality*, or the *dynamic programming principle*, [84] provides a key insight that greatly reduces the computation effort in many planning algorithms. The *value-iteration* method of dynamic programming is the main focus of Section 2.3. The relationship between Dijkstra's algorithm and value iteration is also discussed. Finally, Sections 2.4 and 2.5 describe logic-based representations of planning and methods that exploit these representations to make the problem easier to solve; material from these sections is not needed in later chapters.

Although this chapter addresses a form of planning, it encompasses what is sometimes referred to as *problem solving*. Throughout the history of artificial intelligence research, the distinction between *problem solving* [735] and *planning* has been rather elusive. The widely used textbook by Russell and Norvig [839] provides a representative, modern survey of the field of artificial intelligence. Two of its six main parts are termed "problem-solving" and "planning"; however, their definitions are quite similar. The problem-solving part begins by stating, "Problem solving agents decide what to do by finding sequences of actions that lead to desirable states" ([839], p. 59). The planning part begins with, "The task of coming up with a sequence of actions that will achieve a goal is called planning" ([839], p. 375). Also, the STRIPS system [337] is widely considered as a seminal

planning algorithm, and the "PS" part of its name stands for "Problem Solver." Thus, problem solving and planning appear to be synonymous. Perhaps the term "planning" carries connotations of future time, whereas "problem solving" sounds somewhat more general. A problem-solving task might be to take evidence from a crime scene and piece together the actions taken by suspects. It might seem odd to call this a "plan" because it occurred in the past.

Since it is difficult to make clear distinctions between problem solving and planning, we will simply refer to both as planning. This also helps to keep with the theme of this book. Note, however, that some of the concepts apply to a broader set of problems than what is often meant by planning.

2.1 Introduction to Discrete Feasible Planning

2.1.1 Problem Formulation

The discrete feasible planning model will be defined using state-space models, which will appear repeatedly throughout this book. Most of these will be natural extensions of the model presented in this section. The basic idea is that each distinct situation for the world is called a *state*, denoted by x, and the set of all possible states is called a *state space*, X. For discrete planning, it will be important that this set is countable; in most cases it will be finite. In a given application, the state space should be defined carefully so that irrelevant information is not encoded into a state (e.g., a planning problem that involves moving a robot in France should not encode information about whether certain light bulbs are on in China). The inclusion of irrelevant information can easily convert a problem that is amenable to efficient algorithmic solutions into one that is intractable. On the other hand, it is important that X is large enough to include all information that is relevant to solve the task.

The world may be transformed through the application of *actions* that are chosen by the planner. Each action, u, when applied from the current state, x, produces a new state, x', as specified by a *state transition function*, f. It is convenient to use f to express a *state transition equation*,

$$x' = f(x, u). \tag{2.1}$$

Let $U(x)$ denote the *action space* for each state x, which represents the set of all actions that could be applied from x. For distinct $x, x' \in X$, $U(x)$ and $U(x')$ are not necessarily disjoint; the same action may be applicable in multiple states. Therefore, it is convenient to define the set U of all possible actions over all states:

$$U = \bigcup_{x \in X} U(x). \tag{2.2}$$

As part of the planning problem, a set $X_G \subset X$ of *goal states* is defined. The task of a planning algorithm is to find a finite sequence of actions that when ap-

plied, transforms the initial state x_I to some state in X_G. The model is summarized as:

Formulation 2.1 (Discrete Feasible Planning)

1. A nonempty *state space* X, which is a finite or countably infinite set of *states*.
2. For each state $x \in X$, a finite *action space* $U(x)$.
3. A *state transition function* f that produces a state $f(x, u) \in X$ for every $x \in X$ and $u \in U(x)$. The *state transition equation* is derived from f as $x' = f(x, u)$.
4. An *initial state* $x_I \in X$.
5. A *goal set* $X_G \subset X$.

It is often convenient to express Formulation 2.1 as a directed *state transition graph*. The set of vertices is the state space X. A directed edge from $x \in X$ to $x' \in X$ exists in the graph if and only if there exists an action $u \in U(x)$ such that $x' = f(x, u)$. The initial state and goal set are designated as special vertices in the graph, which completes the representation of Formulation 2.1 in graph form.

2.1.2 Examples of Discrete Planning

Example 2.1 (Moving on a 2D Grid) Suppose that a robot moves on a grid in which each grid point has integer coordinates of the form (i, j). The robot takes discrete steps in one of four directions (up, down, left, right), each of which increments or decrements one coordinate. The motions and corresponding state transition graph are shown in Figure 2.1, which can be imagined as stepping from tile to tile on an infinite tile floor.

This will be expressed using Formulation 2.1. Let X be the set of all integer pairs of the form (i, j), in which $i, j \in \mathbb{Z}$ ($\mathbb{Z}$ denotes the set of all integers). Let $U = \{(0, 1), (0, -1), (1, 0), (-1, 0)\}$. Let $U(x) = U$ for all $x \in X$. The state transition equation is $f(x, u) = x + u$, in which $x \in X$ and $u \in U$ are treated as two-dimensional vectors for the purpose of addition. For example, if $x = (3, 4)$ and $u = (0, 1)$, then $f(x, u) = (3, 5)$. Suppose for convenience that the initial state is $x_I = (0, 0)$. Many interesting goal sets are possible. Suppose, for example, that $X_G = \{(100, 100)\}$. It is easy to find a sequence of actions that transforms the state from $(0, 0)$ to $(100, 100)$.

The problem can be made more interesting by shading in some of the square tiles to represent obstacles that the robot must avoid, as shown in Figure 2.2. In this case, any tile that is shaded has its corresponding vertex and associated edges deleted from the state transition graph. An outer boundary can be made to fence in a bounded region so that X becomes finite. Very complicated labyrinths can be constructed. ■

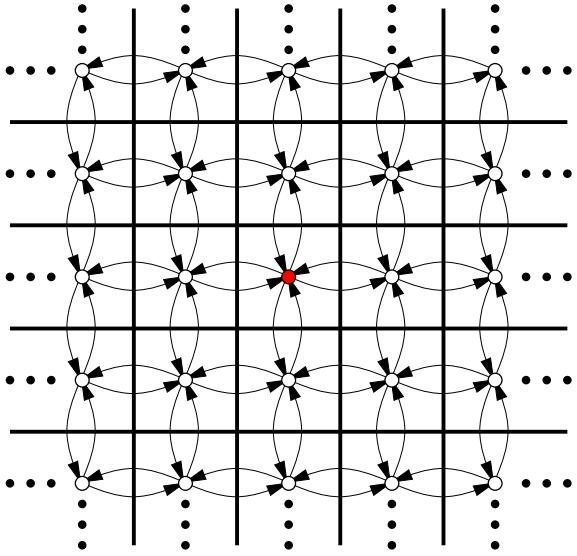


Figure 2.1: The state transition graph for an example problem that involves walking around on an infinite tile floor.

Example 2.2 (Rubik's Cube Puzzle) Many puzzles can be expressed as discrete planning problems. For example, the Rubik's cube is a puzzle that looks like an array of $3 \times 3 \times 3$ little cubes, which together form a larger cube as shown in Figure 1.1a (Section 1.2). Each face of the larger cube is painted one of six colors. An action may be applied to the cube by rotating a 3×3 sheet of cubes by 90 degrees. After applying many actions to the Rubik's cube, each face will generally be a jumble of colors. The state space is the set of configurations for the cube (the orientation of the entire cube is irrelevant). For each state there are 12 possible actions. For some arbitrarily chosen configuration of the Rubik's cube, the planning task is to find a sequence of actions that returns it to the configuration

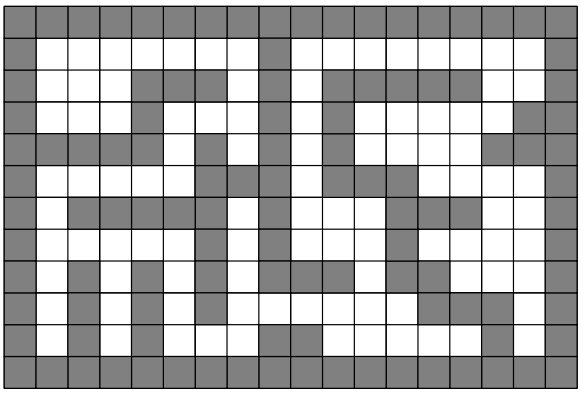


Figure 2.2: Interesting planning problems that involve exploring a labyrinth can be made by shading in tiles.

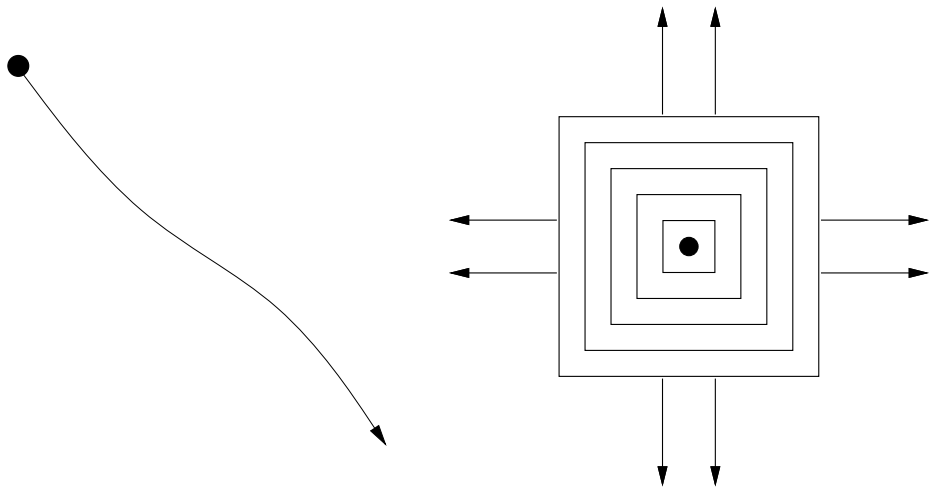
in which each one of its six faces is a single color. ■

It is important to note that a planning problem is usually specified without explicitly representing the entire state transition graph. Instead, it is revealed incrementally in the planning process. In Example 2.1, very little information actually needs to be given to specify a graph that is infinite in size. If a planning problem is given as input to an algorithm, close attention must be paid to the encoding when performing a complexity analysis. For a problem in which X is infinite, the input length must still be finite. For some interesting classes of problems it may be possible to compactly specify a model that is equivalent to Formulation 2.1. Such representation issues have been the basis of much research in artificial intelligence over the past decades as different representation logics have been proposed; see Section 2.4 and [382]. In a sense, these representations can be viewed as input compression schemes.

Readers experienced in computer engineering might recognize that when X is finite, Formulation 2.1 appears almost identical to the definition of a *finite state machine* or *Mealy/Moore machines*. Relating the two models, the actions can be interpreted as *inputs* to the state machine, and the output of the machine simply reports its state. Therefore, the feasible planning problem (if X is finite) may be interpreted as determining whether there exists a sequence of inputs that makes a finite state machine eventually report a desired output. From a planning perspective, it is assumed that the planning algorithm has a complete specification of the machine transitions and is able to read its current state at any time.

Readers experienced with theoretical computer science may observe similar connections to a *deterministic finite automaton* (DFA), which is a special kind of finite state machine that reads an *input string* and makes a decision about whether to *accept* or *reject* the string. The input string is just a finite sequence of inputs, in the same sense as for a finite state machine. A DFA definition includes a set of *accept states*, which in the planning context can be renamed to the *goal set*. This makes the feasible planning problem (if X is finite) equivalent to determining whether there exists an input string that is accepted by a given DFA. Usually, a *language* is associated with a DFA, which is the set of all strings it accepts. DFAs are important in the theory of computation because their languages correspond precisely to regular expressions. The planning problem amounts to determining whether the empty language is associated with the DFA.

Thus, there are several ways to represent and interpret the discrete feasible planning problem that sometimes lead to a very compact, implicit encoding of the problem. This issue will be revisited in Section 2.4. Until then, basic planning algorithms are introduced in Section 2.2, and discrete optimal planning is covered in Section 2.3.



(a) (b)

Figure 2.3: (a) Many search algorithms focus too much on one direction, which may prevent them from being systematic on infinite graphs. (b) If, for example, the search carefully expands in wavefronts, then it becomes systematic. The requirement to be systematic is that, in the limit, as the number of iterations tends to infinity, all reachable vertices are reached.

2.2 Searching for Feasible Plans

The methods presented in this section are just graph search algorithms, but with the understanding that the state transition graph is revealed incrementally through the application of actions, instead of being fully specified in advance. The presentation in this section can therefore be considered as visiting graph search algorithms from a planning perspective. An important requirement for these or any search algorithms is to be *systematic*. If the graph is finite, this means that the algorithm will visit every reachable state, which enables it to correctly declare in finite time whether or not a solution exists. To be systematic, the algorithm should keep track of states already visited; otherwise, the search may run forever by cycling through the same states. Ensuring that no redundant exploration occurs is sufficient to make the search systematic.

If the graph is infinite, then we are willing to tolerate a weaker definition for being systematic. If a solution exists, then the search algorithm still must report it in finite time; however, if a solution does not exist, it is acceptable for the algorithm to search forever. This systematic requirement is achieved by ensuring that, in the limit, as the number of search iterations tends to infinity, every reachable vertex in the graph is explored. Since the number of vertices is assumed to be countable, this must always be possible.

As an example of this requirement, consider Example 2.1 on an infinite tile floor with no obstacles. If the search algorithm explores in only one direction, as

```
FORWARD_SEARCH
1   Q.Insert(x_I) and mark x_I as visited
2   while Q not empty do
3       x ← Q.GetFirst()
4       if x ∈ X_G
5           return SUCCESS
6       forall u ∈ U(x)
7           x' ← f(x, u)
8           if x' not visited
9               Mark x' as visited
10              Q.Insert(x')
11          else
12              Resolve duplicate x'
13  return FAILURE
```

Figure 2.4: A general template for forward search.

depicted in Figure 2.3a, then in the limit most of the space will be left uncovered, even though no states are revisited. If instead the search proceeds outward from the origin in wavefronts, as depicted in Figure 2.3b, then it may be systematic. In practice, each search algorithm has to be carefully analyzed. A search algorithm could expand in multiple directions, or even in wavefronts, but still not be systematic. If the graph is finite, then it is much simpler: Virtually any search algorithm is systematic, provided that it marks visited states to avoid revisiting the same states indefinitely.

2.2.1 General Forward Search

Figure 2.4 gives a general template of search algorithms, expressed using the state-space representation. At any point during the search, there will be three kinds of states:

1. **Unvisited:** States that have not been visited yet. Initially, this is every state except x_I.

2. **Dead:** States that have been visited, and for which every possible next state has also been visited. A *next state* of x is a state x' for which there exists a $u \in U(x)$ such that $x' = f(x, u)$. In a sense, these states are *dead* because there is nothing more that they can contribute to the search; there are no new leads that could help in finding a feasible plan. Section 2.3.3 discusses a variant in which dead states can become alive again in an effort to obtain optimal plans.

3. **Alive:** States that have been encountered, but possibly have unvisited next states. These are considered *alive*. Initially, the only alive state is x_I.

The set of alive states is stored in a priority queue, Q, for which a priority function must be specified. The only significant difference between various search algorithms is the particular function used to sort Q. Many variations will be described later, but for the time being, it might be helpful to pick one. Therefore, assume for now that Q is a common FIFO (First-In First-Out) queue; whichever state has been waiting the longest will be chosen when $Q.GetFirst()$ is called. The rest of the general search algorithm is quite simple. Initially, Q contains the initial state x_I. A **while** loop is then executed, which terminates only when Q is empty. This will only occur when the entire graph has been explored without finding any goal states, which results in a FAILURE (unless the reachable portion of X is infinite, in which case the algorithm should never terminate). In each **while** iteration, the highest ranked element, x, of Q is removed. If x lies in X_G, then it reports SUCCESS and terminates; otherwise, the algorithm tries applying every possible action, $u \in U(x)$. For each next state, $x' = f(x, u)$, it must determine whether x' is being encountered for the first time. If it is unvisited, then it is inserted into Q; otherwise, there is no need to consider it because it must be either dead or already in Q.

The algorithm description in Figure 2.4 omits several details that often become important in practice. For example, how efficient is the test to determine whether $x \in X_G$ in line 4? This depends, of course, on the size of the state space and on the particular representations chosen for x and X_G. At this level, we do not specify a particular method because the representations are not given.

One important detail is that the existing algorithm only indicates whether a solution exists, but does not seem to produce a plan, which is a sequence of actions that achieves the goal. This can be fixed by inserting a line after line 7 that associates with x' its parent, x. If this is performed each time, one can simply trace the pointers from the final state to the initial state to recover the plan. For convenience, one might also store which action was taken, in addition to the pointer from x' to x.

Lines 8 and 9 are conceptually simple, but how can one tell whether x' has been visited? For some problems the state transition graph might actually be a tree, which means that there are no repeated states. Although this does not occur frequently, it is wonderful when it does because there is no need to check whether states have been visited. If the states in X all lie on a grid, one can simply make a lookup table that can be accessed in constant time to determine whether a state has been visited. In general, however, it might be quite difficult because the state x' must be compared with every other state in Q and with all of the dead states. If the representation of each state is long, as is sometimes the case, this will be very costly. A good hashing scheme or another clever data structure can greatly alleviate this cost, but in many applications the computation time will remain high. One alternative is to simply allow repeated states, but this could lead to an increase in computational cost that far outweighs the benefits. Even if the graph is very small, search algorithms could run in time exponential in the size of the state transition graph, or the search may not terminate at all, even if the graph is

finite.

One final detail is that some search algorithms will require a cost to be computed and associated with every state. If the same state is reached multiple times, the cost may have to be updated, which is performed in line 12, if the particular search algorithm requires it. Such costs may be used in some way to sort the priority queue, or they may enable the recovery of the plan on completion of the algorithm. Instead of storing pointers, as mentioned previously, the optimal cost to return to the initial state could be stored with each state. This cost alone is sufficient to determine the action sequence that leads to any visited state. Starting at a visited state, the path back to x_I can be obtained by traversing the state transition graph backward in a way that decreases the cost as quickly as possible in each step. For this to succeed, the costs must have a certain monotonicity property, which is obtained by Dijkstra's algorithm and A^* search, and will be introduced in Section 2.2.2. More generally, the costs must form a *navigation function*, which is considered in Section 8.2.2 as feedback is incorporated into discrete planning.

2.2.2 Particular Forward Search Methods

This section presents several search algorithms, each of which constructs a search tree. Each search algorithm is a special case of the algorithm in Figure 2.4, obtained by defining a different sorting function for Q. Most of these are just classical graph search algorithms [243].

Breadth first The method given in Section 2.2.1 specifies Q as a First-In First-Out (FIFO) queue, which selects states using the first-come, first-serve principle. This causes the search frontier to grow uniformly and is therefore referred to as *breadth-first search*. All plans that have k steps are exhausted before plans with $k + 1$ steps are investigated. Therefore, breadth first guarantees that the first solution found will use the smallest number of steps. On detection that a state has been revisited, there is no work to do in line 12. Since the search progresses in a series of wavefronts, breadth-first search is systematic. In fact, it even remains systematic if it does not keep track of repeated states (however, it will waste time considering irrelevant cycles).

The asymptotic running time of breadth-first search is $O(|V| + |E|)$, in which $|V|$ and $|E|$ are the numbers of vertices and edges, respectively, in the state transition graph (recall, however, that the graph is usually not the input; for example, the input may be the rules of the Rubik's cube). This assumes that all basic operations, such as determining whether a state has been visited, are performed in constant time. In practice, these operations will typically require more time and must be counted as part of the algorithm's complexity. The running time can be expressed in terms of the other representations. Recall that $|V| = |X|$ is the number of states. If the same actions U are available from every state, then $|E| = |U||X|$. If the action sets $U(x_1)$ and $U(x_2)$ are pairwise disjoint for any $x_1, x_2 \in X$, then $|E| = |U|$.

Depth first By making Q a stack (Last-In, First-Out; or LIFO), aggressive exploration of the state transition graph occurs, as opposed to the uniform expansion of breadth-first search. The resulting variant is called *depth-first search* because the search dives quickly into the graph. The preference is toward investigating longer plans very early. Although this aggressive behavior might seem desirable, note that the particular choice of longer plans is arbitrary. Actions are applied in the **forall** loop in whatever order they happen to be defined. Once again, if a state is revisited, there is no work to do in line 12. Depth-first search is systematic for any finite X but not for an infinite X because it could behave like Figure 2.3a. The search could easily focus on one "direction" and completely miss large portions of the search space as the number of iterations tends to infinity. The running time of depth first search is also $O(|V| + |E|)$.

Dijkstra's algorithm Up to this point, there has been no reason to prefer one action over any other in the search. Section 2.3 will formalize optimal discrete planning and will present several algorithms that find optimal plans. Before going into that, we present a systematic search algorithm that finds optimal plans because it is also useful for finding feasible plans. The result is the well-known Dijkstra's algorithm for finding single-source shortest paths in a graph [273], which is a special form of dynamic programming. More general dynamic programming computations appear in Section 2.3 and throughout the book.

Suppose that every edge, $e \in E$, in the graph representation of a discrete planning problem has an associated nonnegative cost $l(e)$, which is the cost to apply the action. The cost $l(e)$ could be written using the state-space representation as $l(x, u)$, indicating that it costs $l(x, u)$ to apply action u from state x. The total cost of a plan is just the sum of the edge costs over the path from the initial state to a goal state.

The priority queue, Q, will be sorted according to a function $C : X \rightarrow [0, \infty]$, called the *cost-to-come*. For each state x, the value $C^*(x)$ is called the *optimal*[1] *cost-to-come* from the initial state x_I. This optimal cost is obtained by summing edge costs, $l(e)$, over all possible paths from x_I to x and using the path that produces the least cumulative cost. If the cost is not known to be optimal, then it is written as $C(x)$.

The cost-to-come is computed incrementally during the execution of the search algorithm in Figure 2.4. Initially, $C^*(x_I) = 0$. Each time the state x' is generated, a cost is computed as $C(x') = C^*(x) + l(e)$, in which e is the edge from x to x' (equivalently, we may write $C(x') = C^*(x) + l(x, u)$). Here, $C(x')$ represents the best cost-to-come that is known so far, but we do not write C^* because it is not yet known whether x' was reached optimally. Due to this, some work is required in line 12. If x' already exists in Q, then it is possible that the newly discovered path to x' is more efficient. If so, then the cost-to-come value $C(x')$ must be lowered for x', and Q must be reordered accordingly.

[1] As in optimization literature, we will use * to mean *optimal*.

When does $C(x)$ finally become $C^*(x)$ for some state x? Once x is removed from Q using $Q.GetFirst()$, the state becomes dead, and it is known that x cannot be reached with a lower cost. This can be argued by induction. For the initial state, $C^*(x_I)$ is known, and this serves as the base case. Now assume that every dead state has its optimal cost-to-come correctly determined. This means that their cost-to-come values can no longer change. For the first element, x, of Q, the value must be optimal because any path that has a lower total cost would have to travel through another state in Q, but these states already have higher costs. All paths that pass only through dead states were already considered in producing $C(x)$. Once all edges leaving x are explored, then x can be declared as dead, and the induction continues. This is not enough detail to constitute a proof of optimality; more arguments appear in Section 2.3.3 and in [243]. The running time is $O(|V| \lg |V| + |E|)$, in which $|V|$ and $|E|$ are the numbers of edges and vertices, respectively, in the graph representation of the discrete planning problem. This assumes that the priority queue is implemented with a Fibonacci heap, and that all other operations, such as determining whether a state has been visited, are performed in constant time. If other data structures are used to implement the priority queue, then higher running times may be obtained.

A-star The A^* (pronounced "ay star") search algorithm is an extension of Dijkstra's algorithm that tries to reduce the total number of states explored by incorporating a heuristic estimate of the cost to get to the goal from a given state. Let $C(x)$ denote the cost-to-come from x_I to x, and let $G(x)$ denote the cost-to-go from x to some state in X_G. It is convenient that $C^*(x)$ can be computed incrementally by dynamic programming; however, there is no way to know the true optimal cost-to-go, G^*, in advance. Fortunately, in many applications it is possible to construct a reasonable underestimate of this cost. As an example of a typical underestimate, consider planning in the labyrinth depicted in Figure 2.2. Suppose that the cost is the total number of steps in the plan. If one state has coordinates (i, j) and another has (i', j'), then $|i' - i| + |j' - j|$ is an underestimate because this is the length of a straightforward plan that ignores obstacles. Once obstacles are included, the cost can only increase as the robot tries to get around them (which may not even be possible). Of course, zero could also serve as an underestimate, but that would not provide any helpful information to the algorithm. The aim is to compute an estimate that is as close as possible to the optimal cost-to-go and is also guaranteed to be no greater. Let $\hat{G}^*(x)$ denote such an estimate.

The A^* search algorithm works in exactly the same way as Dijkstra's algorithm. The only difference is the function used to sort Q. In the A^* algorithm, the sum $C^*(x') + \hat{G}^*(x')$ is used, implying that the priority queue is sorted by estimates of the optimal cost from x_I to X_G. If $\hat{G}^*(x)$ is an underestimate of the true optimal cost-to-go for all $x \in X$, the A^* algorithm is guaranteed to find optimal plans [337, 777]. As $\hat{G}^*$ becomes closer to G^*, fewer vertices tend to be explored in comparison with Dijkstra's algorithm. This would always seem advantageous, but in some problems it is difficult or impossible to find a heuristic that is both efficient

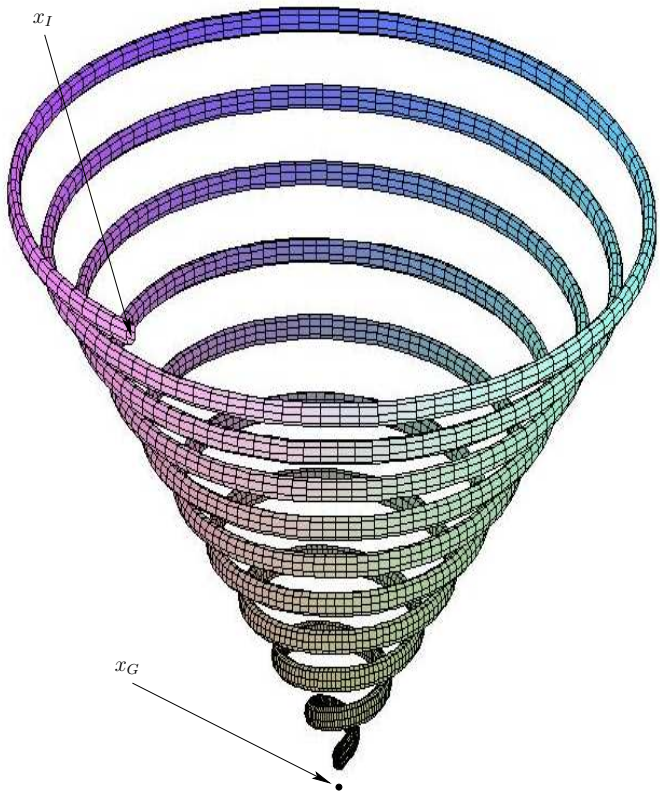


Figure 2.5: Here is a troublesome example for best-first search. Imagine trying to reach a state that is directly below the spiral tube. If the initial state starts inside of the opening at the top of the tube, the search will progress around the spiral instead of leaving the tube and heading straight for the goal.

to evaluate and provides good search guidance. Note that when $\hat{G}^*(x) = 0$ for all $x \in X$, then A^* degenerates to Dijkstra's algorithm. In any case, the search will always be systematic.

Best first For *best-first search*, the priority queue is sorted according to an estimate of the optimal cost-to-go. The solutions obtained in this way are not necessarily optimal; therefore, it does not matter whether the estimate exceeds the true optimal cost-to-go, which was important to maintain optimality for A^* search. Although optimal solutions are not found, in many cases, far fewer vertices are explored, which results in much faster running times. There is no guarantee, however, that this will happen. The worst-case performance of best-first search is worse than that of A^* search and dynamic programming. The algorithm is often too greedy because it prefers states that "look good" very early in the search. Sometimes the price must be paid for being greedy! Figure 2.5 shows a contrived example in which the planning problem involves taking small steps in a 3D world. For any specified number, k, of steps, it is easy to construct a spiral example that wastes at least k steps in comparison to Dijkstra's algorithm. Note that best-first

search is not systematic.

Iterative deepening The *iterative deepening* approach is usually preferable if the search tree has a large branching factor (i.e., there are many more vertices in the next level than in the current level). This could occur if there are many actions per state and only a few states are revisited. The idea is to use depth-first search and find all states that are distance i or less from x_I. If the goal is not found, then the previous work is discarded, and depth first is applied to find all states of distance $i+1$ or less from x_I. This generally iterates from $i = 1$ and proceeds indefinitely until the goal is found. Iterative deepening can be viewed as a way of converting depth-first search into a systematic search method. The motivation for discarding the work of previous iterations is that the number of states reached for $i+1$ is expected to far exceed (e.g., by a factor of 10) the number reached for i. Therefore, once the commitment has been made to reach level $i+1$, the cost of all previous iterations is negligible.

The iterative deepening method has better worst-case performance than breadth-first search for many problems. Furthermore, the space requirements are reduced because the queue in breadth-first search is usually much larger than for depth-first search. If the nearest goal state is i steps from x_I, breadth-first search in the worst case might reach nearly all states of distance $i+1$ before terminating successfully. This occurs each time a state $x \notin X_G$ of distance i from x_I is reached because all new states that can be reached in one step are placed onto Q. The A^* idea can be combined with iterative deepening to yield IDA^*, in which i is replaced by $C^*(x') + \hat{G}^*(x')$. In each iteration of IDA^*, the allowed total cost gradually increases [777].

2.2.3 Other General Search Schemes

This section covers two other general templates for search algorithms. The first one is simply a "backward" version of the tree search algorithm in Figure 2.4. The second one is a bidirectional approach that grows two search trees, one from the initial state and one from a goal state.

Backward search Backward versions of any of the forward search algorithms of Section 2.2.2 can be made. For example, a backward version of Dijkstra's algorithm can be made by starting from x_G. To create backward search algorithms, suppose that there is a single goal state, x_G. For many planning problems, it might be the case that the branching factor is large when starting from x_I. In this case, it might be more efficient to start the search at a goal state and work backward until the initial state is encountered. A general template for this approach is given in Figure 2.6. For forward search, recall that an action $u \in U(x)$ is applied from $x \in X$ to obtain a new state, $x' = f(x,u)$. For backward search, a frequent computation will be to determine for some x', the preceding state $x \in X$, and action $u \in U(x)$ such that $x' = f(x,u)$. The template in Figure 2.6 can be extended to handle a

```
BACKWARD_SEARCH
1   Q.Insert(x_G) and mark x_G as visited
2   while Q not empty do
3       x' ← Q.GetFirst()
4       if x = x_I
5           return SUCCESS
6       forall u^-1 ∈ U^-1(x)
7           x ← f^-1(x', u^-1)
8           if x not visited
9               Mark x as visited
10              Q.Insert(x)
11          else
12              Resolve duplicate x
13  return FAILURE
```

Figure 2.6: A general template for backward search.

goal region, X_G, by inserting all $x_G \in X_G$ into Q in line 1 and marking them as visited.

For most problems, it may be preferable to precompute a representation of the state transition function, f, that is "backward" to be consistent with the search algorithm. Some convenient notation will now be constructed for the backward version of f. Let $U^{-1} = \{(x, u) \in X \times U \mid x \in X, u \in U(x)\}$, which represents the set of all state-action pairs and can also be considered as the domain of f. Imagine from a given state $x' \in X$, the set of all $(x, u) \in U^{-1}$ that map to x' using f. This can be considered as a *backward action space*, defined formally for any $x' \in X$ as

$$U^{-1}(x') = \{(x, u) \in U^{-1} \mid x' = f(x, u)\}. \tag{2.3}$$

For convenience, let u^{-1} denote a state-action pair (x, u) that belongs to some $U^{-1}(x')$. From any $u^{-1} \in U^{-1}(x')$, there is a unique $x \in X$. Thus, let f^{-1} denote a *backward state transition function* that yields x from x' and $u^{-1} \in U^{-1}(x')$. This defines a *backward state transition equation*, $x = f^{-1}(x', u^{-1})$, which looks very similar to the forward version, $x' = f(x, u)$.

The interpretation of f^{-1} is easy to capture in terms of the state transition graph: reverse the direction of every edge. This makes finding a plan in the reversed graph using backward search equivalent to finding one in the original graph using forward search. The backward state transition function is the variant of f that is obtained after reversing all of the edges. Each u^{-1} is a reversed edge. Since there is a perfect symmetry with respect to the forward search of Section 2.2.1, any of the search algorithm variants from Section 2.2.2 can be adapted to the template in Figure 2.6, provided that f^{-1} has been defined.

Bidirectional search Now that forward and backward search have been covered, the next reasonable idea is to conduct a bidirectional search. The general search template given in Figure 2.7 can be considered as a combination of the two in Figures 2.4 and 2.6. One tree is grown from the initial state, and the other is grown from the goal state (assume again that X_G is a singleton, $\{x_G\}$). The search terminates with success when the two trees meet. Failure occurs if either priority queue has been exhausted. For many problems, bidirectional search can dramatically reduce the amount of required exploration. There are Dijkstra and A^* variants of bidirectional search, which lead to optimal solutions. For best-first and other variants, it may be challenging to ensure that the two trees meet quickly. They might come very close to each other and then fail to connect. Additional heuristics may help in some settings to guide the trees into each other. One can even extend this framework to allow any number of search trees. This may be desirable in some applications, but connecting the trees becomes even more complicated and expensive.

2.2.4 A Unified View of the Search Methods

It is convenient to summarize the behavior of all search methods in terms of several basic steps. Variations of these steps will appear later for more complicated planning problems. For example, in Section 5.4, a large family of sampling-based motion planning algorithms can be viewed as an extension of the steps presented here. The extension in this case is made from a discrete state space to a continuous state space (called the configuration space). Each method incrementally constructs a *search graph*, $\mathcal{G}(V, E)$, which is the subgraph of the state transition graph that has been explored so far.

All of the planning methods from this section followed the same basic template:

1. **Initialization:** Let the search graph, $\mathcal{G}(V, E)$, be initialized with E empty and V containing some starting states. For forward search, $V = \{x_I\}$; for backward search, $V = \{x_G\}$. If bidirectional search is used, then $V = \{x_I, x_G\}$. It is possible to grow more than two trees and merge them during the search process. In this case, more states can be initialized in V. The search graph will incrementally grow to reveal more and more of the state transition graph.

2. **Select Vertex:** Choose a vertex $n_{cur} \in V$ for expansion; this is usually accomplished by maintaining a priority queue. Let x_{cur} denote the state associated with n_{cur}.

3. **Apply an Action:** In either a forward or backward direction, a new state, x_{new}, is obtained. This may arise from $x_{new} = f(x, u)$ for some $u \in U(x)$ (forward) or $x = f(x_{new}, u)$ for some $u \in U(x_{new})$ (backward).

4. **Insert a Directed Edge into the Graph:** If certain algorithm-specific tests are passed, then generate an edge from x to x_{new} for the forward case,

```
BIDIRECTIONAL_SEARCH
 1   Q_I.Insert(x_I) and mark x_I as visited
 2   Q_G.Insert(x_G) and mark x_G as visited
 3   while Q_I not empty and Q_G not empty do
 4       if Q_I not empty
 5           x ← Q_I.GetFirst()
 6           if x already visited from x_G
 7               return SUCCESS
 8           forall u ∈ U(x)
 9               x' ← f(x, u)
10               if x' not visited
11                   Mark x' as visited
12                   Q_I.Insert(x')
13               else
14                   Resolve duplicate x'
15       if Q_G not empty
16           x' ← Q_G.GetFirst()
17           if x' already visited from x_I
18               return SUCCESS
19           forall u^-1 ∈ U^-1(x')
20               x ← f^-1(x', u^-1)
21               if x not visited
22                   Mark x as visited
23                   Q_G.Insert(x)
24               else
25                   Resolve duplicate x
26   return FAILURE
```

Figure 2.7: A general template for bidirectional search.

or an edge from x_{new} to x for the backward case. If x_{new} is not yet in V, it will be inserted into V.[2]

5. **Check for Solution:** Determine whether $\mathcal{G}$ encodes a path from x_I to x_G. If there is a single search tree, then this is trivial. If there are two or more search trees, then this step could be expensive.

6. **Return to Step 2:** Iterate unless a solution has been found or an early termination condition is satisfied, in which case the algorithm reports failure.

Note that in this summary, several iterations may have to be made to generate one iteration in the previous formulations. The forward search algorithm in Figure 2.4 tries all actions for the first element of Q. If there are k actions, this corresponds to k iterations in the template above.

2.3 Discrete Optimal Planning

This section extends Formulation 2.1 to allow optimal planning problems to be defined. Rather than being satisfied with any sequence of actions that leads to the goal set, suppose we would like a solution that optimizes some criterion, such as time, distance, or energy consumed. Three important extensions will be made: 1) A stage index will be used to conveniently indicate the current plan step; 2) a cost functional will be introduced, which behaves like a taxi meter by indicating how much cost accumulates during the plan execution; and 3) a termination action will be introduced, which intuitively indicates when it is time to stop the plan and fix the total cost.

The presentation involves three phases. First, the problem of finding optimal paths of a fixed length is covered in Section 2.3.1. The approach, called *value iteration*, involves iteratively computing optimal cost-to-go functions over the state space. Although this case is not very useful by itself, it is much easier to understand than the general case of variable-length plans. Once the concepts from this section are understood, their extension to variable-length plans will be much clearer and is covered in Section 2.3.2. Finally, Section 2.3.3 explains the close relationship between value iteration and Dijkstra's algorithm, which was covered in Section 2.2.1.

With nearly all optimization problems, there is the arbitrary, symmetric choice of whether to define a criterion to *minimize* or *maximize*. If the cost is a kind of energy or expense, then minimization seems sensible, as is typical in robotics and control theory. If the cost is a kind of reward, as in investment planning or in most AI books, then maximization is preferred. Although this issue remains throughout the book, we will choose to minimize everything. If maximization is instead preferred, then multiplying the costs by -1 and swapping minimizations with maximizations should suffice.

[2]In some variations, the vertex could be added without a corresponding edge. This would start another tree in a multiple-tree approach

The fixed-length optimal planning formulation will be given shortly, but first we introduce some new notation. Let π_K denote a *K-step plan*, which is a sequence $(u_1, u_2, \ldots, u_K)$ of K actions. If π_K and x_I are given, then a sequence of states, $(x_1, x_2, \ldots, x_{K+1})$, can be derived using the state transition function, f. Initially, $x_1 = x_I$, and each subsequent state is obtained by $x_{k+1} = f(x_k, u_k)$.

The model is now given; the most important addition with respect to Formulation 2.1 is L, the cost functional.

Formulation 2.2 (Discrete Fixed-Length Optimal Planning)

1. All of the components from Formulation 2.1 are inherited directly: X, $U(x)$, f, x_I, and X_G, except here it is assumed that X is finite (some algorithms may easily extend to the case in which X is countably infinite, but this will not be considered here).

2. A number, K, of *stages*, which is the exact length of a plan (measured as the number of actions, $u_1, u_2, \ldots, u_K$). States may also obtain a stage index. For example, x_{k+1} denotes the state obtained after u_k is applied.

3. Let L denote a stage-additive cost (or loss) functional, which is applied to a K-step plan, π_K. This means that the sequence $(u_1, \ldots, u_K)$ of actions and the sequence $(x_1, \ldots, x_{K+1})$ of states may appear in an expression of L. For convenience, let F denote the *final stage*, $F = K + 1$ (the application of u_K advances the stage to $K + 1$). The *cost functional* is

$$L(\pi_K) = \sum_{k=1}^{K} l(x_k, u_k) + l_F(x_F). \tag{2.4}$$

   The *cost term* $l(x_k, u_k)$ yields a real value for every $x_k \in X$ and $u_k \in U(x_k)$. The *final term* $l_F(x_F)$ is outside of the sum and is defined as $l_F(x_F) = 0$ if $x_F \in X_G$, and $l_F(x_F) = \infty$ otherwise.

An important comment must be made regarding l_F. Including l_F in (2.4) is actually unnecessary if it is agreed in advance that L will only be applied to evaluate plans that reach X_G. It would then be undefined for all other plans. The algorithms to be presented shortly will also function nicely under this assumption; however, the notation and explanation can become more cumbersome because the action space must always be restricted to ensure that successful plans are produced. Instead of this, the domain of L is extended to include all plans, and those that do not reach X_G are penalized with infinite cost so that they are eliminated automatically in any optimization steps. At some point, the role of l_F may become confusing, and it is helpful to remember that it is just a trick to convert feasibility constraints into a straightforward optimization ($L(\pi_K) = \infty$ means *not feasible* and $L(\pi_K) < \infty$ means *feasible with cost* $L(\pi_K)$).

Now the task is to find a plan that minimizes L. To obtain a feasible planning problem like Formulation 2.1 but restricted to K-step plans, let $l(x, u) \equiv 0$. To

obtain a planning problem that requires minimizing the number of stages, let $l(x, u) \equiv 1$. The possibility also exists of having goals that are less "crisp" by letting $l_F(x)$ vary for different $x \in X_G$, as opposed to $l_F(x) = 0$. This is much more general than what was allowed with feasible planning because now states may take on any value, as opposed to being classified as inside or outside of X_G.

2.3.1 Optimal Fixed-Length Plans

Consider computing an optimal plan under Formulation 2.2. One could naively generate all length-K sequences of actions and select the sequence that produces the best cost, but this would require $O(|U|^K)$ running time (imagine K nested loops, one for each stage), which is clearly prohibitive. Luckily, the dynamic programming principle helps. We first say in words what will appear later in equations. The main observation is that portions of optimal plans are themselves optimal. It would be absurd to be able to replace a portion of an optimal plan with a portion that produces lower total cost; this contradicts the optimality of the original plan.

The principle of optimality leads directly to an iterative algorithm, called *value iteration*,[3] that can solve a vast collection of optimal planning problems, including those that involve variable-length plans, stochastic uncertainties, imperfect state measurements, and many other complications. The idea is to iteratively compute optimal cost-to-go (or cost-to-come) functions over the state space. In some cases, the approach can be reduced to Dijkstra's algorithm; however, this only occurs under some special conditions. The *value-iteration* algorithm will be presented next, and Section 2.3.3 discusses its connection to Dijkstra's algorithm.

2.3.1.1 Backward value iteration

As for the search methods, there are both forward and backward versions of the approach. The backward case will be covered first. Even though it may appear superficially to be easier to progress from x_I, it turns out that progressing backward from X_G is notationally simpler. The forward case will then be covered once some additional notation is introduced.

The key to deriving long optimal plans from shorter ones lies in the construction of optimal cost-to-go functions over X. For k from 1 to F, let G_k^* denote the cost that accumulates from stage k to F under the execution of the optimal plan:

$$G_k^*(x_k) = \min_{u_k, \ldots, u_K} \left\{ \sum_{i=k}^{K} l(x_i, u_i) + l_F(x_F) \right\}. \tag{2.5}$$

Inside of the min of (2.5) are the last $F - k$ terms of the cost functional, (2.4). The optimal cost-to-go for the boundary condition of $k = F$ reduces to

$$G_F^*(x_F) = l_F(x_F). \tag{2.6}$$

[3]The "value" here refers to the optimal cost-to-go or cost-to-come. Therefore, an alternative name could be *cost-to-go iteration*.

This makes intuitive sense: Since there are no stages in which an action can be applied, the final stage cost is immediately received.

Now consider an algorithm that makes K passes over X, each time computing G_k^* from G_{k+1}^*, as k ranges from F down to 1. In the first iteration, G_F^* is copied from l_F without significant effort. In the second iteration, G_K^* is computed for each $x_K \in X$ as

$$G_K^*(x_K) = \min_{u_K} \Big\{ l(x_K, u_K) + l_F(x_F) \Big\}. \tag{2.7}$$

Since $l_F = G_F^*$ and $x_F = f(x_K, u_K)$, substitutions can be made into (2.7) to obtain

$$G_K^*(x_K) = \min_{u_K} \Big\{ l(x_K, u_K) + G_F^*(f(x_K, u_K)) \Big\}, \tag{2.8}$$

which is straightforward to compute for each $x_K \in X$. This computes the costs of all optimal one-step plans from stage K to stage $F = K + 1$.

It will be shown next that G_k^* can be computed similarly once G_{k+1}^* is given. Carefully study (2.5) and note that it can be written as

$$G_k^*(x_k) = \min_{u_k} \left\{ \min_{u_{k+1},\ldots,u_K} \left\{ l(x_k, u_k) + \sum_{i=k+1}^{K} l(x_i, u_i) + l_F(x_F) \right\} \right\} \tag{2.9}$$

by pulling the first term out of the sum and by separating the minimization over u_k from the rest, which range from u_{k+1} to u_K. The second min does not affect the $l(x_k, u_k)$ term; thus, $l(x_k, u_k)$ can be pulled outside to obtain

$$G_k^*(x_k) = \min_{u_k} \left\{ l(x_k, u_k) + \min_{u_{k+1},\ldots,u_K} \left\{ \sum_{i=k+1}^{K} l(x_i, u_i) + l_F(x_F) \right\} \right\}. \tag{2.10}$$

The inner min is exactly the definition of the optimal cost-to-go function G_{k+1}^*. Upon substitution, this yields the recurrence

$$G_k^*(x_k) = \min_{u_k} \Big\{ l(x_k, u_k) + G_{k+1}^*(x_{k+1}) \Big\}, \tag{2.11}$$

in which $x_{k+1} = f(x_k, u_k)$. Now that the right side of (2.11) depends only on x_k, u_k, and G_{k+1}^*, the computation of G_k^* easily proceeds in $O(|X||U|)$ time. This computation is called a *value iteration*. Note that in each value iteration, some states receive an infinite value only because they are not reachable; a $(K-k)$-step plan from x_k to X_G does not exist. This means that there are no actions, $u_k \in U(x_k)$, that bring x_k to a state $x_{k+1} \in X$ from which a $(K-k-1)$-step plan exists that terminates in X_G.

Summarizing, the value iterations proceed as follows:

$$G_F^* \;\rightarrow\; G_K^* \;\rightarrow\; G_{K-1}^* \;\cdots\; G_k^* \;\rightarrow\; G_{k-1}^* \;\cdots\; G_2^* \;\rightarrow\; G_1^* \tag{2.12}$$

until finally G_1^* is determined after $O(K|X||U|)$ time. The resulting G_1^* may be applied to yield $G_1^*(x_I)$, the optimal cost to go to the goal from x_I. It also conveniently gives the optimal cost-to-go from any other initial state. This cost is infinity for states from which X_G cannot be reached in K stages.

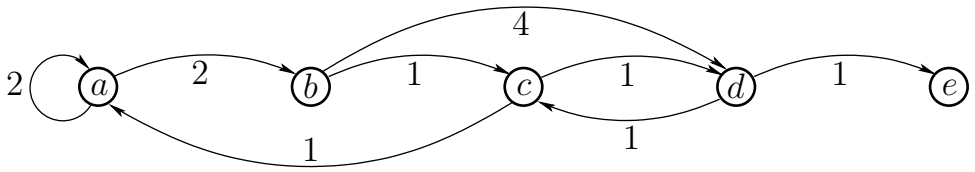


Figure 2.8: A five-state example. Each vertex represents a state, and each edge represents an input that can be applied to the state transition equation to change the state. The weights on the edges represent $l(x_k, u_k)$ (x_k is the originating vertex of the edge).

	a	b	c	d	e
G_5^*	∞	∞	∞	0	∞
G_4^*	∞	4	1	∞	∞
G_3^*	6	2	∞	2	∞
G_2^*	4	6	3	∞	∞
G_1^*	6	4	5	4	∞

Figure 2.9: The optimal cost-to-go functions computed by backward value iteration.

It seems convenient that the cost of the optimal plan can be computed so easily, but how is the actual plan extracted? One possibility is to store the action that satisfied the min in (2.11) from every state, and at every stage. Unfortunately, this requires $O(K|X|)$ storage, but it can be reduced to $O(|X|)$ using the tricks to come in Section 2.3.2 for the more general case of variable-length plans.

Example 2.3 (A Five-State Optimal Planning Problem) Figure 2.8 shows a graph representation of a planning problem in which $X = \{a, c, b, d, e\}$. Suppose that $K = 4$, $x_I = a$, and $X_G = \{d\}$. There will hence be four value iterations, which construct G_4^*, G_3^*, G_2^*, and G_1^*, once the final-stage cost-to-go, G_5^*, is given.

The cost-to-go functions are shown in Figure 2.9. Figures 2.10 and 2.11 il-

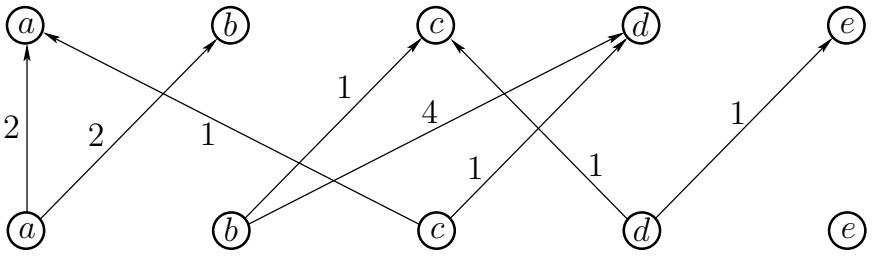


Figure 2.10: The possibilities for advancing forward one stage. This is obtained by making two copies of the states from Figure 2.8, one copy for the current state and one for the potential next state.

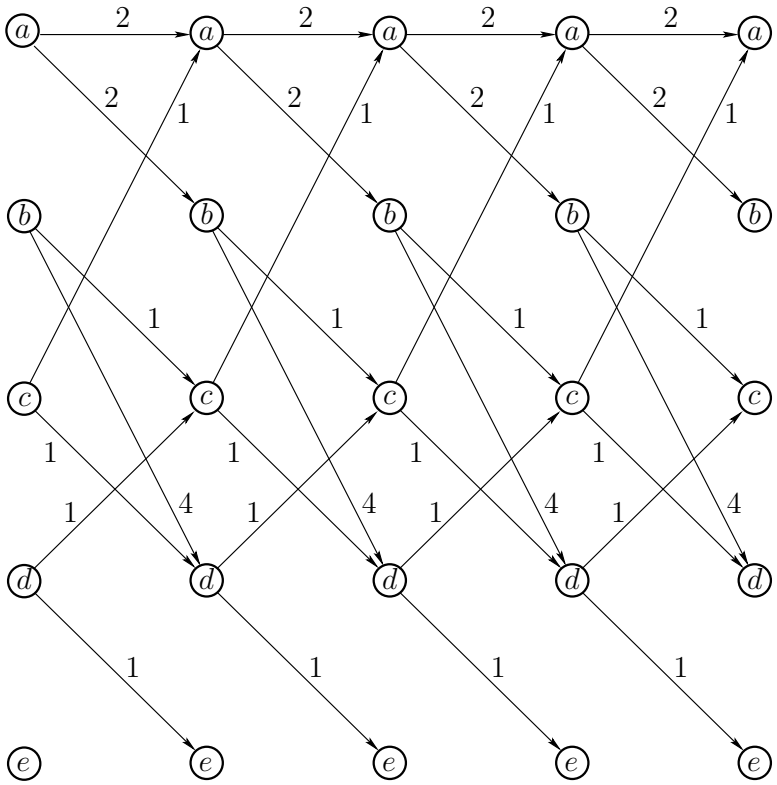


Figure 2.11: By turning Figure 2.10 sideways and copying it K times, a graph can be drawn that easily shows all of the ways to arrive at a final state from an initial state by flowing from left to right. The computations automatically select the optimal route.

lustrate the computations. For computing G_4^*, only b and c receive finite values because only they can reach d in one stage. For computing G_3^*, only the values $G_4^*(b) = 4$ and $G_4^*(c) = 1$ are important. Only paths that reach b or c can possibly lead to d in stage $k = 5$. Note that the minimization in (2.11) always chooses the action that produces the lowest total cost when arriving at a vertex in the next stage. ■

2.3.1.2 Forward value iteration

The ideas from Section 2.3.1.1 may be recycled to yield a symmetrically equivalent method that computes optimal *cost-to-come* functions from the initial stage. Whereas backward value iterations were able to find optimal plans from all initial states simultaneously, forward value iterations can be used to find optimal plans to all states in X. In the backward case, X_G must be fixed, and in the forward case, x_I must be fixed.

The issue of maintaining feasible solutions appears again. In the forward di-

rection, the role of l_F is not important. It may be applied in the last iteration, or it can be dropped altogether for problems that do not have a predetermined X_G. However, one must force all plans considered by forward value iteration to originate from x_I. We again have the choice of either making notation that imposes constraints on the action spaces or simply adding a term that forces infeasible plans to have infinite cost. Once again, the latter will be chosen here.

Let C_k^* denote the *optimal cost-to-come* from stage 1 to stage k, optimized over all $(k-1)$-step plans. To preclude plans that do not start at x_I, the definition of C_1^* is given by

$$C_1^*(x_1) = l_I(x_1), \tag{2.13}$$

in which l_I is a new function that yields $l_I(x_I) = 0$, and $l_I(x) = \infty$ for all $x \neq x_I$. Thus, any plans that try to start from a state other than x_I will immediately receive infinite cost.

For an intermediate stage, $k \in \{2, \ldots, K\}$, the following represents the optimal cost-to-come:

$$C_k^*(x_k) = \min_{u_1,\ldots,u_{k-1}} \left\{ l_I(x_1) + \sum_{i=1}^{k-1} l(x_i, u_i) \right\}. \tag{2.14}$$

Note that the sum refers to a sequence of states, $x_1, \ldots, x_{k-1}$, which is the result of applying the action sequence $(u_1, \ldots, u_{k-2})$. The last state, x_k, is not included because its cost term, $l(x_k, u_k)$, requires the application of an action, u_k, which has not been chosen. If it is possible to write the cost additively, as $l(x_k, u_k) = l_1(x_k) + l_2(u_k)$, then the $l_1(x_k)$ part could be included in the cost-to-come definition, if desired. This detail will not be considered further.

As in (2.5), it is assumed in (2.14) that $u_i \in U(x_i)$ for every $i \in \{1, \ldots, k-1\}$. The resulting x_k, obtained after applying u_{k-1}, must be the same x_k that is named in the argument on the left side of (2.14). It might appear odd that x_1 appears inside of the min above; however, this is not a problem. The state x_1 can be completely determined once $u_1, \ldots, u_{k-1}$ and x_k are given.

The final forward value iteration is the arrival at the final stage, F. The cost-to-come in this case is

$$C_F^*(x_F) = \min_{u_1,\ldots,u_K} \left\{ l_I(x_1) + \sum_{i=1}^{K} l(x_i, u_i) \right\}. \tag{2.15}$$

This equation looks the same as (2.5) after substituting $k = 1$; however, l_I is used here instead of l_F. This has the effect of filtering the plans that are considered to include only those that start at x_I. The forward value iterations find optimal plans to any reachable final state from x_I. This behavior is complementary to that of backward value iteration. In that case, X_G was fixed, and optimal plans from any initial state were found. For forward value iteration, this is reversed.

To express the dynamic-programming recurrence, one further issue remains. Suppose that C_{k-1}^* is known by induction, and we want to compute $C_k^*(x_k)$ for a particular x_k. This means that we must start at some state x_{k-1} and arrive

	a	b	c	d	e
C_1^*	0	∞	∞	∞	∞
C_2^*	2	2	∞	∞	∞
C_3^*	4	4	3	6	∞
C_4^*	4	6	5	4	7
C_5^*	6	6	5	6	5

Figure 2.12: The optimal cost-to-come functions computed by forward value iteration.

in state x_k by applying some action. Once again, the backward state transition equation from Section 2.2.3 is useful. Using the stage indices, it is written here as $x_{k-1} = f^{-1}(x_k, u_k^{-1})$.

The recurrence is

$$C_k^*(x_k) = \min_{u_k^{-1} \in U^{-1}(x_k)} \Big\{ C_{k-1}^*(x_{k-1}) + l(x_{k-1}, u_{k-1}) \Big\}, \tag{2.16}$$

in which $x_{k-1} = f^{-1}(x_k, u_k^{-1})$ and $u_{k-1} \in U(x_{k-1})$ is the input to which $u_k^{-1} \in U^{-1}(x_k)$ corresponds. Using (2.16), the final cost-to-come is iteratively computed in $O(K|X||U|)$ time, as in the case of computing the first-stage cost-to-go in the backward value-iteration method.

Example 2.4 (Forward Value Iteration) Example 2.3 is revisited for the case of forward value iterations with a fixed plan length of $K = 4$. The cost-to-come functions shown in Figure 2.12 are obtained by direct application of (2.16). It will be helpful to refer to Figures 2.10 and 2.11 once again. The first row corresponds to the immediate application of l_I. In the second row, finite values are obtained for a and b, which are reachable in one stage from $x_I = a$. The iterations continue until $k = 5$, at which point that optimal cost-to-come is determined for every state. ■

2.3.2 Optimal Plans of Unspecified Lengths

The value-iteration method for fixed-length plans can be generalized nicely to the case in which plans of different lengths are allowed. There will be no bound on the maximal length of a plan; therefore, the current case is truly a generalization of Formulation 2.1 because arbitrarily long plans may be attempted in efforts to reach X_G. The model for the general case does not require the specification of K but instead introduces a special action, u_T.

Formulation 2.3 (Discrete Optimal Planning)

1. All of the components from Formulation 2.1 are inherited directly: X, $U(x)$, f, x_I, and X_G. Also, the notion of stages from Formulation 2.2 is used.

2. Let L denote a stage-additive cost functional, which may be applied to any K-step plan, π_K, to yield

$$L(\pi_K) = \sum_{k=1}^{K} l(x_k, u_k) + l_F(x_F). \tag{2.17}$$

   In comparison with L from Formulation 2.2, the present expression does not consider K as a predetermined constant. It will now vary, depending on the length of the plan. Thus, the domain of L is much larger.

3. Each $U(x)$ contains the special *termination action*, u_T. If u_T is applied at x_k, then the action is repeatedly applied forever, the state remains unchanged, and no more cost accumulates. Thus, for all $i \geq k$, $u_i = u_T$, $x_i = x_k$, and $l(x_i, u_T) = 0$.

The termination action is the key to allowing plans of different lengths. It will appear throughout this book. Suppose that value iterations are performed up to $K = 5$, and for the problem there exists a two-step solution plan, (u_1, u_2), that arrives in X_G from x_I. This plan is equivalent to the five-step plan $(u_1, u_2, u_T, u_T, u_T)$ because the termination action does not change the state, nor does it accumulate cost. The resulting five-step plan reaches X_G and costs the same as (u_1, u_2). With this simple extension, the forward and backward value iteration methods of Section 2.3.1 may be applied for any fixed K to optimize over all plans of length K or less (instead of fixing K).

The next step is to remove the dependency on K. Consider running backward value iterations indefinitely. At some point, G_1^* will be computed, but there is no reason why the process cannot be continued onward to G_0^*, G_{-1}^*, and so on. Recall that x_I is not utilized in the backward value-iteration method; therefore, there is no concern regarding the starting initial state of the plans. Suppose that backward value iteration was applied for $K = 16$ and was executed down to G_{-8}^*. This considers all plans of length 25 or less. Note that it is harmless to add 9 to all stage indices to shift all of the cost-to-go functions. Instead of running from G_{-8}^* to G_{16}^*, they can run from G_1^* to G_{25}^* without affecting their values. The index shifting is allowed because none of the costs depend on the particular index that is given to the stage. The only important aspect of the value iterations is that they proceed backward and consecutively from stage to stage.

Eventually, enough iterations will have been executed so that an optimal plan is known from every state that can reach X_G. From that stage, say k, onward, the cost-to-go values from one value iteration to the next will be *stationary*, meaning that for all $i \leq k$, $G_{i-1}^*(x) = G_i^*(x)$ for all $x \in X$. Once the stationary condition is reached, the cost-to-go function no longer depends on a particular stage k. In this case, the stage index may be dropped, and the recurrence becomes

$$G^*(x) = \min_u \Big\{ l(x, u) + G^*(f(x, u)) \Big\}. \tag{2.18}$$

Are there any conditions under which backward value iterations could be executed forever, with each iteration producing a cost-to-go function for which some values are different from the previous iteration? If $l(x, u)$ is nonnegative for all $x \in X$ and $u \in U(x)$, then this could never happen. It could certainly be true that, for any fixed K, longer plans will exist, but this cannot be said of *optimal* plans. From every $x \in X$, there either exists a plan that reaches X_G with finite cost or there is no solution. For each state from which there exists a plan that reaches X_G, consider the number of stages in the optimal plan. Consider the maximum number of stages taken from all states that can reach X_G. This serves as an upper bound on the number of value iterations before the cost-to-go becomes stationary. Any further iterations will just consider solutions that are worse than the ones already considered (some may be equivalent due to the termination action and shifting of stages). Some trouble might occur if $l(x, u)$ contains negative values. If the state transition graph contains a cycle for which total cost is negative, then it is preferable to execute a plan that travels around the cycle forever, thereby reducing the total cost to $-\infty$. Therefore, we will assume that the cost functional is defined in a sensible way so that negative cycles do not exist. Otherwise, the optimization model itself appears flawed. Some negative values for $l(x, u)$, however, are allowed as long as there are no negative cycles. (It is straightforward to detect and report negative cycles before running the value iterations.)

Since the particular stage index is unimportant, let $k = 0$ be the index of the final stage, which is the stage at which the backward value iterations begin. Hence, G^*_0 is the final stage cost, which is obtained directly from l_F. Let $-K$ denote the stage index at which the cost-to-go values all become stationary. At this stage, the optimal cost-to-go function, $G^* : X \to \mathbb{R} \cup \{\infty\}$, is expressed by assigning $G^* = G^*_{-K}$. In other words, the particular stage index no longer matters. The value $G^*(x)$ gives the optimal cost to go from state $x \in X$ to the specific goal state x_G.

If the optimal actions are not stored during the value iterations, the optimal cost-to-go, G^*, can be used to efficiently recover them. Consider starting from some $x \in X$. What is the optimal next action? This is given by

$$u^* = \operatorname*{argmin}_{u \in U(x)} \Big\{ l(x, u) + G^*(f(x, u)) \Big\}, \tag{2.19}$$

in which argmin denotes the argument that achieves the minimum value of the expression. The action minimizes an expression that is very similar to (2.11). The only differences between (2.19) and (2.11) are that the stage indices are dropped in (2.19) because the cost-to-go values no longer depend on them, and argmin is used so that u^* is selected. After applying u^*, the state transition equation is used to obtain $x' = f(x, u^*)$, and (2.19) may be applied again on x'. This process continues until a state in X_G is reached. This procedure is based directly on the dynamic programming recurrence; therefore, it recovers the optimal plan. The function G^* serves as a kind of guide that leads the system from any initial state into the goal set optimally. This can be considered as a special case of a *navigation function*, which will be covered in Section 8.2.2.

As in the case of fixed-length plans, the direction of the value iterations can be reversed to obtain a forward value-iteration method for the variable-length planning problem. In this case, the backward state transition equation, f^{-1}, is used once again. Also, the initial cost term l_I is used instead of l_F, as in (2.14). The forward value-iteration method starts at $k = 1$, and then iterates until the cost-to-come becomes stationary. Once again, the termination action, u_T, preserves the cost of plans that arrived at a state in earlier iterations. Note that it is not required to specify X_G. A counterpart to G^* may be obtained, from which optimal actions can be recovered. When the cost-to-come values become stationary, an optimal cost-to-come function, $C^* : X \rightarrow \mathbb{R} \cup \{\infty\}$, may be expressed by assigning $C^* = C^*_F$, in which F is the final stage reached when the algorithm terminates. The value $C^*(x)$ gives the cost of an optimal plan that starts from x_I and reaches x. The optimal action sequence for any specified goal $x_G \in X$ can be obtained using

$$\operatorname*{argmin}_{u^{-1} \in U^{-1}} \Big\{ C^*(f^{-1}(x, u^{-1})) + l(f^{-1}(x, u^{-1}), u') \Big\}, \tag{2.20}$$

which is the forward counterpart of (2.19). The u' is the action in $U(f^{-1}(x, u^{-1}))$ that yields x when the state transition function, f, is applied. The iterations proceed backward from x_G and terminate when x_I is reached.

Example 2.5 (Value Iteration for Variable-Length Plans) Once again, Example 2.3 is revisited; however, this time the plan length is not fixed due to the termination action. Its effect is depicted in Figure 2.13 by the superposition of new edges that have zero cost. It might appear at first that there is no incentive to choose nontermination actions, but remember that any plan that does not terminate in state $x_G = d$ will receive infinite cost.

See Figure 2.14. After a few backward value iterations, the cost-to-go values become stationary. After this point, the termination action is being applied from all reachable states and no further cost accumulates. The final cost-to-go function is defined to be G^*. Since d is not reachable from e, $G^*(e) = \infty$.

As an example of using (2.19) to recover optimal actions, consider starting from state a. The action that leads to b is chosen next because the total cost $2 + G^*(b) = 4$ is better than $2 + G^*(a) = 6$ (the 2 comes from the action cost). From state b, the optimal action leads to c, which produces total cost $1 + G^*(c) = 1$. Similarly, the next action leads to $d \in X_G$, which terminates the plan.

Using forward value iteration, suppose that $x_I = b$. The following cost-to-come functions shown in Figure 2.15 are obtained. For any finite value that remains constant from one iteration to the next, the termination action was applied. Note that the last value iteration is useless in this example. Once C^*_3 is computed, the optimal cost-to-come to every possible state from x_I is determined, and future cost-to-come functions are identical. Therefore, the final cost-to-come is renamed C^*. ■

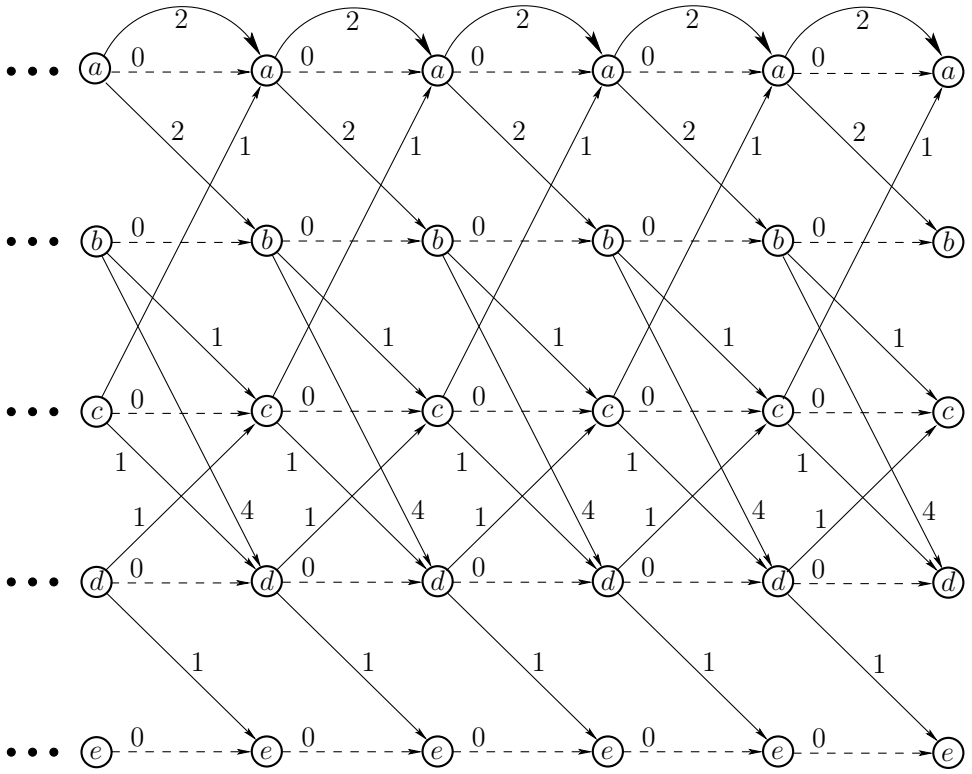


Figure 2.13: Compare this figure to Figure 2.11, for which K was fixed at 4. The effect of the termination action is depicted as dashed-line edges that yield 0 cost when traversed. This enables plans of all finite lengths to be considered. Also, the stages extend indefinitely to the left (for the case of backward value iteration).

	a	b	c	d	e
G^*_0	∞	∞	∞	0	∞
G^*_{-1}	∞	4	1	0	∞
G^*_{-2}	6	2	1	0	∞
G^*_{-3}	4	2	1	0	∞
G^*_{-4}	4	2	1	0	∞
G^*	4	2	1	0	∞

Figure 2.14: The optimal cost-to-go functions computed by backward value iteration applied in the case of variable-length plans.

	a	b	c	d	e
C_1^*	∞	0	∞	∞	∞
C_2^*	∞	0	1	4	∞
C_3^*	2	0	1	2	5
C_4^*	2	0	1	2	3
C^*	2	0	1	2	3

Figure 2.15: The optimal cost-to-come functions computed by forward value iteration applied in the case of variable-length plans.

2.3.3 Dijkstra Revisited

So far two different kinds of dynamic programming have been covered. The value-iteration method of Section 2.3.2 involves repeated computations over the entire state space. Dijkstra's algorithm from Section 2.2.2 flows only once through the state space, but with the additional overhead of maintaining which states are *alive*.

Dijkstra's algorithm can be derived by focusing on the forward value iterations, as in Example 2.5, and identifying exactly where the "interesting" changes occur. Recall that for Dijkstra's algorithm, it was assumed that all costs are nonnegative. For any states that are not reachable, their values remain at infinity. They are precisely the *unvisited* states. States for which the optimal cost-to-come has already become stationary are *dead*. For the remaining states, an initial cost is obtained, but this cost may be lowered one or more times until the optimal cost is obtained. All states for which the cost is finite, but possibly not optimal, are in the queue, Q.

After understanding value iteration, it is easier to understand why Dijkstra's form of dynamic programming correctly computes optimal solutions. It is clear that the unvisited states will remain at infinity in both algorithms because no plan has reached them. It is helpful to consider the forward value iterations in Example 2.5 for comparison. In a sense, Dijkstra's algorithm is very much like the value iteration, except that it efficiently maintains the set of states within which cost-to-go values can change. It correctly inserts any states that are reached for the first time, changing their cost-to-come from infinity to a finite value. The values are changed in the same manner as in the value iterations. At the end of both algorithms, the resulting values correspond to the stationary, optimal cost-to-come, C^*.

If Dijkstra's algorithm seems so clever, then why have we spent time covering the value-iteration method? For some problems it may become too expensive to maintain the sorted queue, and value iteration could provide a more efficient alternative. A more important reason is that value iteration extends easily to a much broader class of problems. Examples include optimal planning over continuous state spaces (Sections 8.5.2 and 14.5), stochastic optimal planning (Section 10.2), and computing dynamic game equilibria (Section 10.5). In some cases, it

```
FORWARD_LABEL_CORRECTING(x_G)
 1  Set C(x) = ∞ for all x ≠ x_I, and set C(x_I) = 0
 2  Q.Insert(x_I)
 3  while Q not empty do
 4      x ← Q.GetFirst()
 5      forall u ∈ U(x)
 6          x' ← f(x, u)
 7          if C(x) + l(x, u) < min{C(x'), C(x_G)} then
 8              C(x') ← C(x) + l(x, u)
 9              if x' ≠ x_G then
10                  Q.Insert(x')
```

Figure 2.16: A generalization of Dijkstra's algorithm, which upon termination produces an optimal plan (if one exists) for any prioritization of Q, as long as X is finite. Compare this to Figure 2.4.

is still possible to obtain a Dijkstra-like algorithm by focusing the computation on the "interesting" region; however, as the model becomes more complicated, it may be inefficient or impossible in practice to maintain this region. Therefore, it is important to have a good understanding of both algorithms to determine which is most appropriate for a given problem.

Dijkstra's algorithm belongs to a broader family of *label-correcting algorithms*, which all produce optimal plans by making small modifications to the general forward-search algorithm in Figure 2.4. Figure 2.16 shows the resulting algorithm. The main difference is to allow states to become alive again if a better cost-to-come is found. This enables other cost-to-come values to be improved accordingly. This is not important for Dijkstra's algorithm and A^* search because they only need to visit each state once. Thus, the algorithms in Figures 2.4 and 2.16 are essentially the same in this case. However, the label-correcting algorithm produces optimal solutions for any sorting of Q, including FIFO (breadth first) and LIFO (depth first), as long as X is finite. If X is not finite, then the issue of systematic search dominates because one must guarantee that states are revisited sufficiently many times to guarantee that optimal solutions will eventually be found.

Another important difference between label-correcting algorithms and the standard forward-search model is that the label-correcting approach uses the cost at the goal state to prune away many candidate paths; this is shown in line 7. Thus, it is only formulated to work for a single goal state; it can be adapted to work for multiple goal states, but performance degrades. The motivation for including $C(x_G)$ in line 7 is that there is no need to worry about improving costs at some state, x', if its new cost-to-come would be higher than $C(x_G)$; there is no way it could be along a path that improves the cost to go to x_G. Similarly, x_G is not inserted in line 10 because there is no need to consider plans that have x_G as an intermediate state. To recover the plan, either pointers can be stored from x to x'

each time an update is made in line 7, or the final, optimal cost-to-come, C^*, can be used to recover the actions using (2.20).

2.4 Using Logic to Formulate Discrete Planning

For many discrete planning problems that we would like a computer to solve, the state space is enormous (e.g., 10^{100} states). Therefore, substantial effort has been invested in constructing *implicit* encodings of problems in hopes that the entire state space does not have to be explored by the algorithm to solve the problem. This will be a recurring theme throughout this book; therefore, it is important to pay close attention to representations. Many planning problems can appear trivial once everything has been explicitly given.

Logic-based representations have been popular for constructing such implicit representations of discrete planning. One historical reason is that such representations were the basis of the majority of artificial intelligence research during the 1950s–1980s. Another reason is that they have been useful for representing certain kinds of planning problems very compactly. It may be helpful to think of these representations as compression schemes. A string such as 010101010101... may compress very nicely, but it is impossible to substantially compress a random string of bits. Similar principles are true for discrete planning. Some problems contain a kind of regularity that enables them to be expressed compactly, whereas for others it may be impossible to find such representations. This is why there has been a variety of representation logics proposed through decades of planning research.

Another reason for using logic-based representations is that many discrete planning algorithms are implemented in large software systems. At some point, when these systems solve a problem, they must provide the complete plan to a user, who may not care about the internals of planning. Logic-based representations have seemed convenient for producing output that logically explains the steps involved to arrive at some goal. Other possibilities may exist, but logic has been a first choice due to its historical popularity.

In spite of these advantages, one shortcoming with logic-based representations is that they are difficult to generalize. It is important in many applications to enable concepts such as continuous spaces, unpredictability, sensing uncertainty, and multiple decision makers to be incorporated into planning. This is the main reason why the state-space representation has been used so far: It will be easy to extend and adapt to the problems covered throughout this book. Nevertheless, it is important to study logic-based representations to understand the relationship between the vast majority of discrete planning research and other problems considered in this book, such as motion planning and planning under differential constraints. There are many recurring themes throughout these different kinds of problems, even though historically they have been investigated by separate research communities. Understanding these connections well provides powerful insights into planning issues across all of these areas.

2.4.1 A STRIPS-Like Representation

STRIPS-like representations have been the most common logic-based representations for discrete planning problems. This refers to the STRIPS system, which is considered one of the first planning algorithms and representations [337]; its name is derived from the STanford Research Institute Problem Solver. The original representation used first-order logic, which had great expressive power but many technical difficulties. Therefore, the representation was later restricted to only propositional logic [743], which is similar to the form introduced in this section. There are many variations of STRIPS-like representations. Here is one formulation:

Formulation 2.4 (STRIPS-Like Planning)

1. A finite, nonempty set I of *instances*.

2. A finite, nonempty set P of *predicates*, which are binary-valued (partial) functions of one of more instances. Each application of a predicate to a specific set of instances is called a *positive literal*. A logically negated positive literal is called a *negative literal*.

3. A finite, nonempty set O of *operators*, each of which has: 1) *preconditions*, which are positive or negative literals that must hold for the operator to apply, and 2) *effects*, which are positive or negative literals that are the result of applying the operator.

4. An *initial set* S which is expressed as a set of *positive literals*. Negative literals are implied. For any positive literal that does not appear in S, its corresponding negative literal is assumed to hold initially.

5. A *goal set* G which is expressed as a set of both *positive* and *negative literals*.

Formulation 2.4.1 provides a definition of discrete feasible planning expressed in a STRIPS-like representation. The three most important components are the sets of *instances* I, *predicates* P, and *operators* O. Informally, the instances characterize the complete set of distinct things that exist in the world. They could, for example, be books, cars, trees, and so on. The predicates correspond to basic properties or statements that can be formed regarding the instances. For example, a predicate called $Under$ might be used to indicate things like $Under(Book, Table)$ (the book is under the table) or $Under(Dirt, Rug)$. A predicate can be interpreted as a kind of function that yields TRUE or FALSE values; however, it is important to note that it is only a partial function because it might not be desirable to allow any instance to be inserted as an argument to the predicate.

If a predicate is evaluated on an instance, for example, $Under(Dirt, Rug)$, the expression is called a *positive literal*. The set of all possible positive literals can be formed by applying all possible instances to the domains over which the predicates are defined. Every positive literal has a corresponding *negative literal*, which is

formed by negating the positive literal. For example, $\neg Under(Dirt, Rug)$ is the negative literal that corresponds to the positive literal $Under(Dirt, Rug)$, and $\neg$ denotes negation. Let a *complementary pair* refer to a positive literal together with its counterpart negative literal. The various components of the planning problem are expressed in terms of positive and negative literals.

The role of an operator is to change the world. To be applicable, a set of *preconditions* must all be satisfied. Each element of this set is a positive or negative literal that must hold TRUE for the operator to be applicable. Any complementary pairs that can be formed from the predicates, but are not mentioned in the preconditions, may assume any value without affecting the applicability of the operator. If the operator is applied, then the world is updated in a manner precisely specified by the set of *effects*, which indicates positive and negative literals that result from the application of the operator. It is assumed that the truth values of all unmentioned complementary pairs are not affected.

Multiple operators are often defined in a single statement by using variables. For example, $Insert(i)$ may allow any instance $i \in I$ to be inserted. In some cases, this dramatically reduces the space required to express the problem.

The planning problem is expressed in terms of an initial set S of positive literals and a goal set G of positive and negative literals. A state can be defined by selecting either the positive or negative literal for every possible complementary pair. The initial set S specifies such a state by giving the positive literals only. For all possible positive literals that do not appear in S, it is assumed that their negative counterparts hold in the initial state. The goal set G actually refers to a set of states because, for any unmentioned complementary pair, the positive or negative literal may be chosen, and the goal is still achieved. The task is to find a sequence of operators that when applied in succession will transform the world from the initial state into one in which all literals of G are TRUE. For each operator, the preconditions must also be satisfied before it can be applied. The following example illustrates Formulation 2.4.

Example 2.6 (Putting Batteries into a Flashlight) Imagine a planning problem that involves putting two batteries into a flashlight, as shown in Figure 2.17. The set of instances are

$$I = \{Battery1, Battery2, Cap, Flashlight\}. \tag{2.21}$$

Two different predicates will be defined, On and In, each of which is a partial function on I. The predicate On may only be applied to evaluate whether the Cap is On the $Flashlight$ and is written as $On(Cap, Flashlight)$. The predicate In may be applied in the following two ways: $In(Battery1, Flashlight)$, $In(Battery2, Flashlight)$, to indicate whether either battery is in the flashlight. Recall that predicates are only partial functions in general. For the predicate In, it is not desirable to apply any instance to any argument. For example, it is meaningless to define $In(Battery1, Battery1)$ and $In(Flashlight, Battery2)$ (they could be included in the model, always retaining a negative value, but it is inefficient).

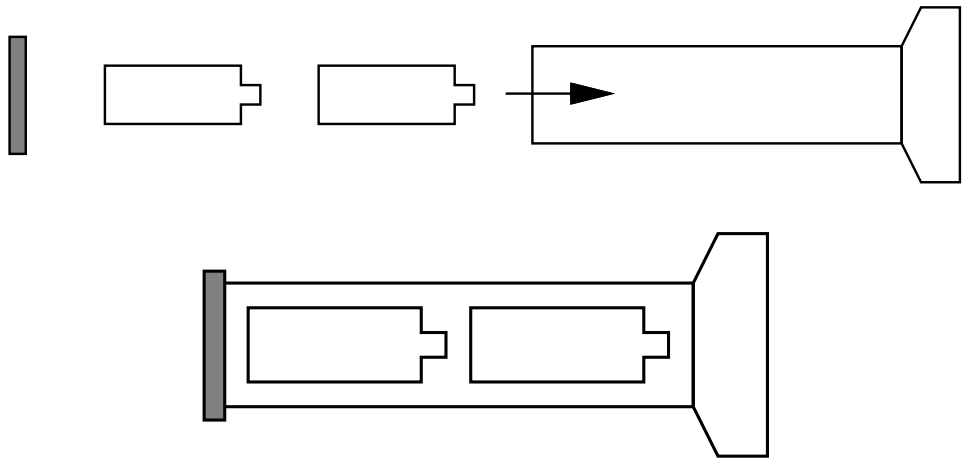


Figure 2.17: An example that involves putting batteries into a flashlight.

Name	Preconditions	Effects
$PlaceCap$	$\{\neg On(Cap, Flashlight)\}$	$\{On(Cap, Flashlight)\}$
$RemoveCap$	$\{On(Cap, Flashlight)\}$	$\{\neg On(Cap, Flashlight)\}$
$Insert(i)$	$\{\neg On(Cap, Flashlight), \neg In(i, Flashlight)\}$	$\{In(i, Flashlight)\}$

Figure 2.18: Three operators for the flashlight problem. Note that an operator can be expressed with variable argument(s) for which different instances could be substituted.

The initial set is

$$S = \{On(Cap, Flashlight)\}. \tag{2.22}$$

Based on S, both $\neg In(Battery1, Flashlight)$ and $\neg In(Battery2, Flashlight)$ are assumed to hold. Thus, S indicates that the cap is on the flashlight, but the batteries are outside.

The goal state is

$$\begin{aligned} G = \{&On(Cap, Flashlight), In(Battery1, Flashlight), \\ &In(Battery2, Flashlight)\}, \end{aligned} \tag{2.23}$$

which means that both batteries must be in the flashlight, and the cap must be on.

The set O consists of the four operators, which are shown in Figure 2.18. Here is a plan that reaches the goal state in the smallest number of steps:

$$(RemoveCap, Insert(Battery1), Insert(Battery2), PlaceCap). \tag{2.24}$$

In words, the plan simply says to take the cap off, put the batteries in, and place the cap back on.

This example appears quite simple, and one would expect a planning algorithm to easily find such a solution. It can be made more challenging by adding many more instances to I, such as more batteries, more flashlights, and a bunch of objects that are irrelevant to achieving the goal. Also, many other predicates and operators can be added so that the different combinations of operators become overwhelming. ■

A large number of complexity results exist for planning expressed using logic. The graph search problem is solved efficiently in polynomial time; however, a state transition graph is not given as the input. An input that is expressed using Formulation 2.4 may describe an enormous state transition graph using very few instances, predicates, and operators. In a sense, the model is highly compressed when using some logic-based formulations. This brings it closer to the *Kolmogorov complexity* [248, 630] of the state transition graph, which is the shortest bit size to which it can possibly be compressed and then fully recovered by a Turing machine. This has the effect of making the planning problem appear more difficult. Concise inputs may encode very challenging planning problems. Most of the known hardness results are surveyed in Chapter 3 of [382]. Under most formulations, logic-based planning is NP-hard. The particular level of hardness (NP, PSPACE, EXPTIME, etc.) depends on the precise problem conditions. For example, the complexity depends on whether the operators are fixed in advance or included in the input. The latter case is much harder. Separate complexities are also obtained based on whether negative literals are allowed in the operator effects and also whether they are allowed in preconditions. The problem is generally harder if both positive and negative literals are allowed in these cases.

2.4.2 Converting to the State-Space Representation

It is useful to characterize the relationship between Formulation 2.4 and the original formulation of discrete feasible planning, Formulation 2.1. One benefit is that it immediately shows how to adapt the search methods of Section 2.2 to work for logic-based representations. It is also helpful to understand the relationships between the algorithmic complexities of the two representations.

Up to now, the notion of "state" has been only vaguely mentioned in the context of the STRIPS-like representation. Now consider making this more concrete. Suppose that every predicate has k arguments, and any instance could appear in each argument. This means that there are $|P|\,|I|^k$ complementary pairs, which corresponds to all of the ways to substitute instances into all arguments of all predicates. To express the state, a positive or negative literal must be selected from every complementary pair. For convenience, this selection can be encoded as a binary string by imposing a linear ordering on the instances and predicates.

Using Example 2.6, the state might be specified in order as

$$(On(Cap, Flashlight), \neg In(Battery1, Flashlight1), In(Battery2, Flashlight)). \tag{2.25}$$

Using a binary string, each element can be "0" to denote a negative literal or "1" to denote positive literal. The encoded state is $x = 101$ for (2.25). If any instance can appear in the argument of any predicate, then the length of the string is $|P|\,|I|^k$. The total number of possible states of the world that could possibly be distinguished corresponds to the set of all possible bit strings. This set has size

$$2^{|P|\,|I|^k}. \tag{2.26}$$

The implication is that with a very small number of instances and predicates, an enormous state space can be generated. Even though the search algorithms of Section 2.2 may appear efficient with respect to the size of the search graph (or the number of states), the algorithms appear horribly inefficient with respect to the sizes of P and I. This has motivated substantial efforts on the development of techniques to help guide the search by exploiting the structure of specific representations. This is the subject of Section 2.5.

The next step in converting to a state-space representation is to encode the initial state x_I as a string. The goal set, X_G, is the set of all strings that are consistent with the positive and negative goal literals. This can be compressed by extending the string alphabet to include a "don't care" symbol, δ. A single string that has a "0" for each negative literal, a "1" for each positive literal, and a "δ" for all others would suffice in representing any X_G that is expressed with positive and negative literals.

Now convert the operators. For each state, $x \in X$, the set $U(x)$ represents the set of operators with preconditions that are satisfied by x. To apply the search techniques of Section 2.2, note that it is not necessary to determine $U(x)$ explicitly in advance for all $x \in X$. Instead, $U(x)$ can be computed whenever each x is encountered for the first time in the search. The effects of the operator are encoded by the state transition equation. From a given $x \in X$, the next state, $f(x, u)$, is obtained by flipping the bits as prescribed by the effects part of the operator.

All of the components of Formulation 2.1 have been derived from the components of Formulation 2.4. Adapting the search techniques of Section 2.2 is straightforward. It is also straightforward to extend Formulation 2.4 to represent optimal planning. A cost can be associated with each operator and set of literals that capture the current state. This would express $l(x, u)$ of the cost functional, L, from Section 2.3. Thus, it is even possible to adapt the value-iteration method to work under the logic-based representation, yielding optimal plans.

2.5 Logic-Based Planning Methods

A huge body of research has been developed over the last few decades for planning using logic-based representations [382, 839]. These methods usually exploit some structure that is particular to the representation. Furthermore, numerous heuristics for accelerating performance have been developed from implementation studies. The main ideas behind some of the most influential approaches are described in this section, but without presenting particular heuristics.

Rather than survey all logic-based planning methods, this section focuses on some of the main approaches that exploit logic-based representations. Keep in mind that the searching methods of Section 2.2 also apply. Once a problem is given using Formulation 2.4, the state transition graph is incrementally revealed during the search. In practice, the search graph may be huge relative to the size of the problem description. One early attempt to reduce the size of this graph was the STRIPS planning algorithm [337, 743]; it dramatically reduced the branching factor but unfortunately was not complete. The methods presented in this section represent other attempts to reduce search complexity in practice while maintaining completeness. For each method, there are some applications in which the method may be more efficient, and others for which performance may be worse. Thus, there is no clear choice of method that is independent of its particular use.

2.5.1 Searching in a Space of Partial Plans

One alternative to searching directly in X is to construct partial plans without reference to particular states. By using the operator representation, partial plans can be incrementally constructed. The idea is to iteratively achieve required subgoals in a partial plan while ensuring that no conflicts arise that could destroy the solution developed so far.

A *partial plan* σ is defined as

1. A set O_σ of operators that need to be applied. If the operators contain variables, these may be filled in by specific values or left as variables. The same operator may appear multiple times in O_σ, possibly with different values for the variables.

2. A partial ordering relation $\prec_\sigma$ on O_σ, which indicates for some pairs $o_1, o_2 \in O_\sigma$ that one must appear before other: $o_1 \prec_\sigma o_2$.

3. A set B_σ of *binding constraints*, in which each indicates that some variables across operators must take on the same value.

4. A set C_σ of *causal links*, in which each is of the form (o_1, l, o_2) and indicates that o_1 achieves the literal l for the purpose of satisfying a precondition of o_2.

Example 2.7 (A Partial Plan) Each partial plan encodes a *set* of possible plans. Recall the model from Example 2.6. Suppose

$$O_\sigma = \{RemoveCap, Insert(Battery1)\}. \tag{2.27}$$

A sensible ordering constraint is that

$$RemoveCap \prec_\sigma Insert(Battery1). \tag{2.28}$$

A causal link,

$$(RemoveCap, \neg On(Cap, Flashlight), Insert(Battery1)), \tag{2.29}$$

indicates that the $RemoveCap$ operator achieves the literal $\neg On(Cap, Flashlight)$, which is a precondition of $Insert(Battery1)$. There are no binding constraints for this example. The partial plan implicitly represents the set of all plans for which $RemoveCap$ appears before $Insert(Battery1)$, under the constraint that the causal link is not violated. ■

Several algorithms have been developed to search in the space of partial plans. To obtain some intuition about the partial-plan approach, a planning algorithm is described in Figure 2.19. A vertex in the partial-plan search graph is a partial plan, and an edge is constructed by extending one partial plan to obtain another partial plan that is closer to completion. Although the general template is simple, the algorithm performance depends critically on the choice of initial plan and the particular flaw that is resolved in each iteration. One straightforward generalization is to develop multiple partial plans and decide which one to refine in each iteration.

In early works, methods based on partial plans seemed to offer substantial benefits; however, they are currently considered to be not "competitive enough" in comparison to methods that search the state space [382]. One problem is that it becomes more difficult to develop application-specific heuristics without explicit references to states. Also, the vertices in the partial-plan search graph are costly to maintain and manipulate in comparison to ordinary states.

2.5.2 Building a Planning Graph

Blum and Furst introduced the notion of a *planning graph*, which is a powerful data structure that encodes information about which states may be reachable [117]. For the logic-based problem expressed in Formulation 2.4, consider performing reachability analysis. Breadth-first search can be used from the initial state to expand the state transition graph. In terms of the input representation, the resulting graph may be of exponential size in the number of stages. This gives precise reachability information and is guaranteed to find the goal state.

The idea of Blum and Furst is to construct a graph that is much smaller than the state transition graph and instead contains only partial information about

PLAN-SPACE PLANNING

1. Start with any initial partial plan, σ.
2. Find a flaw in σ, which may be 1) an operator precondition that has not achieved, or 2) an operator in O_σ that threatens a causal constraint in C_σ.
3. If there is no flaw, then report that σ is a complete solution and compute a linear ordering of O_σ that satisfies all constraints.
4. If the flaw is an unachieved precondition, l, for some operator o_2, then find an operator, o_1, that achieves it and record a new causal constraint, (o_1, l, o_2).
5. If the flaw is a threat on a causal link, then the threat must be removed by updating $\prec_\sigma$ to induce an appropriate operator ordering, or by updating B_σ to bind the operators in a way that resolves the threat.
6. Return to Step 2.

Figure 2.19: Planning in the plan space is achieved by iteratively finding a flaw in the plan and fixing it.

reachability. The resulting *planning graph* is polynomial in size and can be efficiently constructed for some challenging problems. The trade-off is that the planning graph indicates states that can *possibly* be reached. The true reachable set is overapproximated, by eliminating many impossible states from consideration. This enables quick elimination of impossible alternatives in the search process. Planning algorithms have been developed that extract a plan from the planning graph. In the worst case, this may take exponential time, which is not surprising because the problem in Formulation 2.4 is NP-hard in general. Nevertheless, dramatic performance improvements were obtained on some well-known planning benchmarks. Another way to use the planning graph is as a source of information for developing search heuristics for a particular problem.

Planning graph definition A *layered graph* is a graph that has its vertices partitioned into a sequence of *layers*, and its edges are only permitted to connect vertices between successive layers. The *planning graph* is a layered graph in which the layers of vertices form an alternating sequence of literals and operators:

$$(L_1, O_1, L_2, O_2, L_3, O_3, \ldots, L_k, O_k, L_{k+1}). \tag{2.30}$$

The edges are defined as follows. To each operator $o_i \in O_i$, a directed edge is made from each $l_i \in L_i$ that is a precondition of o_i. To each literal $l_i \in L_i$, an edge is made from each operator $o_{i-1} \in O_{i-1}$ that has l_i as an effect.

One important requirement is that no variables are allowed in the operators. Any operator from Formulation 2.4 that contains variables must be converted into

a set that contains a distinct copy of the operator for every possible substitution of values for the variables.

Layer-by-layer construction The planning graph is constructed layer by layer, starting from L_1. In the first stage, L_1 represents the initial state. Every positive literal in S is placed into L_1, along with the negation of every positive literal not in S. Now consider stage i. The set O_i is the set of all operators for which their preconditions are a subset of L_i. The set L_{i+1} is the union of the effects of all operators in O_i. The iterations continue until the planning graph stabilizes, which means that $O_{i+1} = O_i$ and $L_{i+1} = L_i$. This situation is very similar to the stabilization of value iterations in Section 2.3.2. A trick similar to the termination action, u_T, is needed even here so that plans of various lengths are properly handled. In Section 2.3.2, one job of the termination action was to prevent state transitions from occurring. The same idea is needed here. For each possible literal, l, a *trivial operator* is constructed for which l is the only precondition and effect. The introduction of trivial operators ensures that once a literal is reached, it is maintained in the planning graph for every subsequent layer of literals. Thus, each O_i may contain some trivial operators, in addition to operators from the initially given set O. These are required to ensure that the planning graph expansion reaches a steady state, in which the planning graph is identical for all future expansions.

Mutex conditions During the construction of the planning graph, information about the conflict between operators and literals within a layer is maintained. A conflict is called a *mutex condition*, which means that a pair of literals[4] or pair of operators is mutually exclusive. Both cannot be chosen simultaneously without leading to some kind of conflict. A pair in conflict is called *mutex*. For each layer, a *mutex relation* is defined that indicates which pairs satisfy the mutex condition. A pair, $o, o' \in O_i$, of operators is defined to be *mutex* if any of these conditions is met:

1. **Inconsistent effects:** An effect of o is the negated literal of an effect of o'.

2. **Interference:** An effect of o is the negated literal of a precondition of o'.

3. **Competing needs:** A pair of preconditions, one from each of o and o', are mutex in L_i.

The last condition relies on the definition of mutex for literals, which is presented next. Any pair, $l, l' \in L_i$, of literals is defined to be *mutex* if at least one of the two conditions is met:

1. **Negated literals:** l and l' form a complementary pair.

[4]The pair of literals need not be a complementary pair, as defined in Section 2.4.1.

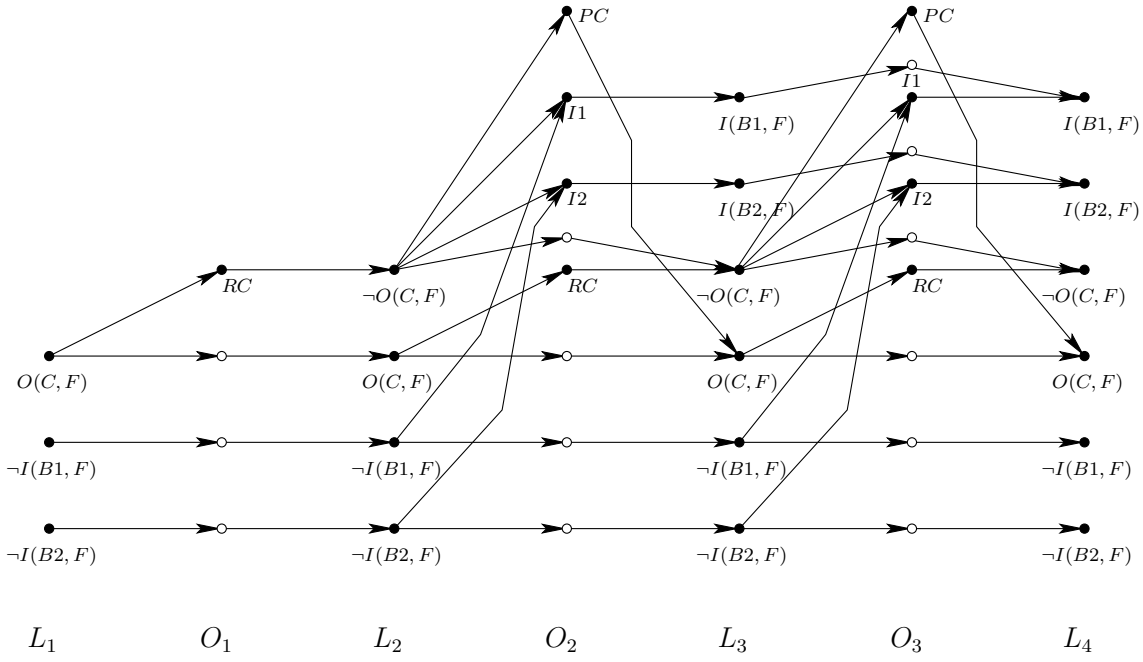


Figure 2.20: The planning graph for the flashlight example. The unlabeled operator vertices correspond to trivial operators. For clarity, the operator and literal names are abbreviated.

2. **Inconsistent support:** Every pair of operators, $o, o' \in O_{i-1}$, that achieve l and l' is mutex. In this case, one operator must achieve l, and the other must achieve l'. If there exists an operator that achieves both, then this condition is false, regardless of the other pairs of operators.

The mutex definition depends on the layers; therefore, it is computed layer by layer during the planning graph construction.

Example 2.8 (The Planning Graph for the Flashlight) Figure 2.20 shows the planning graph for Example 2.6. In the first layer, L_1 expresses the initial state. The only applicable operator is *RemoveCap*. The operator layer O_1 contains *RemoveCap* and three trivial operators, which are needed to maintain the literals from L_1. The appearance of $\neg On(Cap, Flashlight)$ enables the battery-insertion operator to apply. Since variables are not allowed in operator definitions in a planning graph, two different operators (labeled as $I1$ and $I2$) appear, one for each battery. Notice the edges drawn to $I1$ and $I2$ from their preconditions. The cap may also be replaced; hence, *PlaceCap* is included in O_2. At the L_3 layer, all possible literals have been obtained. At O_3, all possible operators, including the trivial ones, are included. Finally, $L_4 = L_3$, and O_4 will be the same as O_3. This implies that the planning graph has stabilized. ■

Plan extraction Suppose that the planning graph has been constructed up to L_i. At this point, the planning graph can be searched for a solution. If no solution is found and the planning graph has stabilized, then no solution exists to the problem in general (this was shown in [117]; see also [382]). If the planning graph has not stabilized, then it can be extended further by adding O_i and L_{i+1}. The extended graph can then be searched for a solution plan. A planning algorithm derived from the planning graph interleaves the graph extensions and the searches for solutions. Either a solution is reported at some point or the algorithm correctly reports that no solution exists after the planning graph stabilizes. The resulting algorithm is complete. One of the key observations in establishing completeness is that the literal and operator layers each increase monotonically as i increases. Furthermore, the sets of pairs that are mutex decrease monotonically, until all possible conflicts are resolved.

Rather than obtaining a fully specified plan, the planning graph yields a *layered plan*, which is a special form of partial plan. All of the necessary operators are included, and the layered plan is specified as

$$(A_1, A_2, \ldots, A_k), \tag{2.31}$$

in which each A_i is a set of operators. Within any A_i, the operators are nonmutex and may be applied in any order without altering the state obtained by the layered plan. The only constraint is that for each i from 1 to k, every operator in A_i must be applied before any operators in A_{i+1} can be applied. For the flashlight example, a layered plan that would be constructed from the planning graph in Figure 2.20 is

$$(\{RemoveCap\}, \{Insert(Battery1), Insert(Battery2)\}, \{PlaceCap\}). \tag{2.32}$$

To obtain a fully specified plan, the layered plan needs to be linearized by specifying a linear ordering for the operators that is consistent with the layer constraints. For (2.32), this results in (2.24). The actual plan execution usually involves more stages than the number in the planning graph. For complicated planning problems, this difference is expected to be huge. With a small number of stages, the planning graph can consider very long plans because it can apply several nonmutex operators in a single layer.

At each level, the search for a plan could be quite costly. The idea is to start from L_i and perform a backward *and/or search*. To even begin the search, the goal literals G must be a subset of L_i, and no pairs are allowed to be mutex; otherwise, immediate failure is declared. From each literal $l \in G$, an "or" part of the search tries possible operators that produce l as an effect. The "and" part of the search must achieve all literals in the precondition of an operator chosen at the previous "or" level. Each of these preconditions must be achieved, which leads to another "or" level in the search. The idea is applied recursively until the initial set L_1 of literals is obtained. During the and/or search, the computed mutex relations provide information that immediately eliminates some

branches. Frequently, triples and higher order tuples are checked for being mutex together, even though they are not pairwise mutex. A hash table is constructed to efficiently retrieve this information as it is considered multiple times in the search. Although the plan extraction is quite costly, superior performance was shown in [117] on several important benchmarks. In the worst case, the search could require exponential time (otherwise, a polynomial-time algorithm would have been found to an NP-hard problem).

2.5.3 Planning as Satisfiability

Another interesting approach is to convert the planning problem into an enormous Boolean satisfiability problem. This means that the planning problem of Formulation 2.4 can be solved by determining whether some assignment of variables is possible for a Boolean expression that leads to a TRUE value. Generic methods for determining satisfiability can be directly applied to the Boolean expression that encodes the planning problem. The *Davis-Putnam procedure* is one of the most widely known algorithms for satisfiability. It performs a depth-first search by iteratively trying assignments for variables and backtracking when assignments fail. During the search, large parts of the expression can be eliminated due to the current assignments. The algorithm is complete and reasonably efficient. Its use in solving planning problems is surveyed in [382]. In practice, stochastic local search methods provide a reasonable alternative to the Davis-Putnam procedure [459].

Suppose a planning problem has been given in terms of Formulation 2.4. All literals and operators will be tagged with a stage index. For example, a literal that appears in two different stages will be considered distinct. This kind of tagging is similar to *situation calculus* [378]; however, in that case, variables are allowed for the tags. To obtain a finite, Boolean expression the total number of stages must be declared. Let K denote the number of stages at which operators can be applied. As usual, the fist stage is $k = 1$ and the final stage is $k = F = K + 1$. Setting a stage limit is a significant drawback of the approach because this is usually not known before the problem is solved. A planning algorithm can assume a small value for F and then gradually increase it each time the resulting Boolean expression is not satisfied. If the problem is not solvable, however, this approach iterates forever.

Let $\vee$ denote logical OR, and let $\wedge$ denote logical AND. The Boolean expression is written as a conjunction[5] of many terms, which arise from five different sources:

1. **Initial state:** A conjunction of all literals in S is formed, along with the negation of all positive literals not in S. These are all tagged with 1, the initial stage index.

2. **Goal state:** A conjunction of all literals in G, tagged with the final stage index, $F = K + 1$.

[5] Conjunction means logical AND.

3. **Operator encodings:** Each operator must be copied over the stages. For each $o \in O$, let o_k denote the operator applied at stage k. A conjunction is formed over all operators at all stages. For each o_k, the expression is

$$\neg o_k \vee (p_1 \wedge p_2 \wedge \cdots \wedge p_m \wedge e_1 \wedge e_2 \wedge \cdots \wedge e_n)\,, \tag{2.33}$$

in which p_1, ..., p_m are the preconditions of o_k, and e_1, ..., e_n are the effects of o_k.

4. **Frame axioms:** The next part is to encode the implicit assumption that every literal that is not an effect of the applied operator remains unchanged in the next stage. This can alternatively be stated as follows: If a literal l becomes negated to $\neg l$, then an operator that includes $\neg l$ as an effect must have been executed. (If l was already a negative literal, then $\neg l$ is a positive literal.) For each stage and literal, an expression is needed. Suppose that l_k and l_{k+1} are the same literal but are tagged for different stages. The expression is

$$(l_k \vee \neg l_{k+1}) \vee (o_{k,1} \vee o_{k,2} \vee \cdots \vee o_{k,j}), \tag{2.34}$$

in which $o_{k,1}$, ..., $o_{k,j}$ are the operators, tagged for stage k, that contain l_{k+1} as an effect. This ensures that if $\neg l_k$ appears, followed by l_{k+1}, then some operator must have caused the change.

5. **Complete exclusion axiom:** This indicates that only one operator applies at every stage. For every stage k, and any pair of stage-tagged operators o_k and o'_k, the expression is

$$\neg o_k \vee \neg o'_k, \tag{2.35}$$

which is logically equivalent to $\neg(o_k \wedge o'_k)$ (meaning, "not both at the same stage").

It is shown in [512] that a solution plan exists if and only if the resulting Boolean expression is satisfiable.

The following example illustrates the construction.

Example 2.9 (The Flashlight Problem as a Boolean Expression) A Boolean expression will be constructed for Example 2.6. Each of the expressions given below is joined into one large expression by connecting them with $\wedge$'s.

The expression for the initial state is

$$O(C, F, 1) \wedge \neg I(B1, F, 1) \wedge \neg I(B2, F, 1), \tag{2.36}$$

which uses the abbreviated names, and the stage tag has been added as an argument to the predicates. The expression for the goal state is

$$O(C, F, 5) \wedge I(B1, F, 5) \wedge I(B2, F, 5), \tag{2.37}$$

which indicates that the goal must be achieved at stage $k = 5$. This value was determined because we already know the solution plan from (2.24). The method

will also work correctly for a larger value of k. The expressions for the operators are

$$\begin{aligned}
&\neg PC_k \vee (\neg O(C,F,k) \wedge O(C,F,k+1)) \\
&\neg RC_k \vee (O(C,F,k) \wedge \neg O(C,F,k+1)) \\
&\neg I1_k \vee (\neg O(C,F,k) \wedge \neg I(B1,F,k) \wedge I(B1,F,k+1)) \\
&\neg I2_k \vee (\neg O(C,F,k) \wedge \neg I(B2,F,k) \wedge I(B2,F,k+1))
\end{aligned} \tag{2.38}$$

for each k from 1 to 4.

The frame axioms yield the expressions

$$\begin{aligned}
&(O(C,F,k) \vee \neg O(C,F,k+1)) \vee (PC_k) \\
&(\neg O(C,F,k) \vee O(C,F,k+1)) \vee (RC_k) \\
&(I(B1,F,k) \vee \neg I(B1,F,k+1)) \vee (I1_k) \\
&(\neg I(B1,F,k) \vee I(B1,F,k+1)) \\
&(I(B2,F,k) \vee \neg I(B2,F,k+1)) \vee (I2_k) \\
&(\neg I(B2,F,k) \vee I(B2,F,k+1)),
\end{aligned} \tag{2.39}$$

for each k from 1 to 4. No operators remove batteries from the flashlight. Hence, two of the expressions list no operators.

Finally, the complete exclusion axiom yields the expressions

$$\begin{aligned}
&\neg RC_k \vee \neg PC_k \qquad \neg RC_k \vee \neg O1_k \qquad \neg RC_k \vee \neg O2_k \\
&\neg PC_k \vee \neg O1_k \qquad \neg PC_k \vee \neg O2_k \qquad \neg O1_k \vee \neg O2_k,
\end{aligned} \tag{2.40}$$

for each k from 1 to 4. The full problem is encoded by combining all of the given expressions into an enormous conjunction. The expression is satisfied by assigning TRUE values to RC_1, $IB1_2$, $IB2_3$, and PC_4. An alternative solution is RC_1, $IB2_2$, $IB1_3$, and PC_4. The stage index tags indicate the order that the actions are applied in the recovered plan. ■

Further Reading

Most of the ideas and methods in this chapter have been known for decades. Most of the search algorithms of Section 2.2 are covered in algorithms literature as graph search [243, 404, 692, 857] and in AI literature as planning or search methods [551, 743, 744, 777, 839, 975]. Many historical references to search in AI appear in [839]. Bidirectional search was introduced in [797, 798] and is closely related to *means-end analysis* [735]; more discussion of bidirectional search appears in [185, 184, 497, 569, 839]. The development of good search heuristics is critical to many applications of discrete planning. For substantial material on this topic, see [382, 550, 777]. For the relationship between planning and scheduling, see [266, 382, 896].

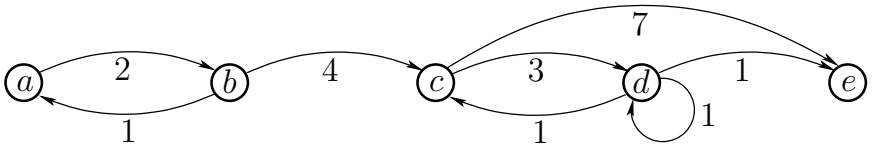


Figure 2.21: Another five-state discrete planning problem.

The dynamic programming principle forms the basis of optimal control theory and many algorithms in computer science. The main ideas follow from Bellman's principle of optimality [84, 85]. These classic works led directly to the value-iteration methods of Section 2.3. For more recent material on this topic, see [95], which includes Dijkstra's algorithm and its generalization to label-correcting algorithms. An important special version of Dijkstra's algorithm is Dial's algorithm [272] (see [946] and Section 8.2.3). Throughout this book, there are close connections between planning methods and control theory. One step in this direction was taken earlier in [267].

The foundations of logic-based planning emerged from early work of Nilsson [337, 743], which contains most of the concepts introduced in Section 2.4. Over the last few decades, an enormous body of literature has been developed. Section 2.5 briefly surveyed some of the highlights; however, several more chapters would be needed to do this subject justice. For a comprehensive, recent treatment of logic-based planning, see [382]; topics beyond those covered here include constraint-satisfaction planning, scheduling, and temporal logic. Other sources for logic-based planning include [378, 839, 963, 984]. A critique of benchmarks used for comparisons of logic-based planning algorithms appears in [464].

Too add uncertainty or multiple decision makers to the problems covered in this chapter, jump ahead to Chapter 10 (this may require some background from Chapter 9). To move from searching in discrete to continuous spaces, try Chapters 5 and 6 (some background from Chapters 3 and 4 is required).

Exercises

1. Consider the planning problem shown in Figure 2.21. Let a be the initial state, and let e be the goal state.

   (a) Use backward value iteration to determine the stationary cost-to-go.

   (b) Do the same but instead use forward value iteration.

2. Try to construct a worst-case example for best-first search that has properties similar to that shown in Figure 2.5, but instead involves moving in a 2D world with obstacles, as introduced in Example 2.1.

3. It turns out that value iteration can be generalized to a cost functional of the form

$$L(\pi_K) = \sum_{k=1}^{K} l(x_k, u_k, x_{k+1}) + l_F(x_F), \tag{2.41}$$

   in which $l(x_k, u_k)$ in (2.4) has been replaced by $l(x_k, u_k, x_{k+1})$.

(a) Show that the dynamic programming principle can be applied in this more general setting to obtain forward and backward value iteration methods that solve the fixed-length optimal planning problem.

(b) Do the same but for the more general problem of variable-length plans, which uses termination conditions.

4. The cost functional can be generalized to being *stage-dependent*, which means that the cost might depend on the particular stage k in addition to the state, x_k and the action u_k. Extend the forward and backward value iteration methods of Section 2.3.1 to work for this case, and show that they give optimal solutions. Each term of the more general cost functional should be denoted as $l(x_k, u_k, k)$.

5. Recall from Section 2.3.2 the method of defining a termination action u_T to make the value iterations work correctly for variable-length planning. Instead of requiring that one remains at the same state, it is also possible to formulate the problem by creating a special state, called the *terminal state*, x_T. Whenever u_T is applied, the state becomes x_T. Describe in detail how to modify the cost functional, state transition equation, and any other necessary components so that the value iterations correctly compute shortest plans.

6. Dijkstra's algorithm was presented as a kind of forward search in Section 2.2.1.

(a) Develop a backward version of Dijkstra's algorithm that starts from the goal. Show that it always yields optimal plans.

(b) Describe the relationship between the algorithm from part (a) and the backward value iterations from Section 2.3.2.

(c) Derive a backward version of the A^* algorithm and show that it yields optimal plans.

7. Reformulate the general forward search algorithm of Section 2.2.1 so that it is expressed in terms of the STRIPS-like representation. Carefully consider what needs to be explicitly constructed by a planning algorithm and what is considered only implicitly.

8. Rather than using bit strings, develop a set-based formulation of the logic-based planning problem. A state in this case can be expressed as a set of positive literals.

9. Extend Formulation 2.4 to allow disjunctive goal sets (there are alternative sets of literals that must be satisfied). How does this affect the binary string representation?

10. Make a *Remove* operator for Example 2.17 that takes a battery away from the flashlight. For this operator to apply, the battery must be in the flashlight and must not be blocked by another battery. Extend the model to allow enough information for the *Remove* operator to function properly.

11. Model the operation of the sliding-tile puzzle in Figure 1.1b using the STRIPS-like representation. You may use variables in the operator definitions.

12. Find the complete set of plans that are implicitly encoded by Example 2.7.

13. Explain why, in Formulation 2.4, G needs to include both positive and negative literals, whereas S only needs positive literals. As an alternative definition, could S have contained only negative literals? Explain.

14. Using Formulation 2.4, model a problem in which a robot checks to determine whether a room is dark, moves to a light switch, and flips on the light. Predicates should indicate whether the robot is at the light switch and whether the light is on. Operators that move the robot and flip the switch are needed.

15. Construct a planning graph for the model developed in Exercise 14.

16. Express the model in Exercise 14 as a Boolean satisfiability problem.

17. In the worst case, how many terms are needed for the Boolean expression for planning as satisfiability? Express your answer in terms of $|I|$, $|P|$, $|O|$, $|S|$, and $|G|$.

Implementations

18. Using A^* search, the performance degrades substantially when there are many alternative solutions that are all optimal, or at least close to optimal. Implement A^* search and evaluate it on various grid-based problems, based on Example 2.1. Compare the performance for two different cases:

    (a) Using $|i' - i| + |j' - j|$ as the heuristic, as suggested in Section 2.2.2.

    (b) Using $\sqrt{(i' - i)^2 + (j' - j)^2}$ as the heuristic.

    Which heuristic seems superior? Explain your answer.

19. Implement A^*, breadth-first, and best-first search for grid-based problems. For each search algorithm, design and demonstrate examples for which one is clearly better than the other two.

20. Experiment with bidirectional search for grid-based planning. Try to understand and explain the trade-off between exploring the state space and the cost of connecting the trees.

21. Try to improve the method used to solve Exercise 18 by detecting when the search might be caught in a local minimum and performing random walks to try to escape. Try using best-first search instead of A^*. There is great flexibility in possible approaches. Can you obtain better performance on average for any particular examples?

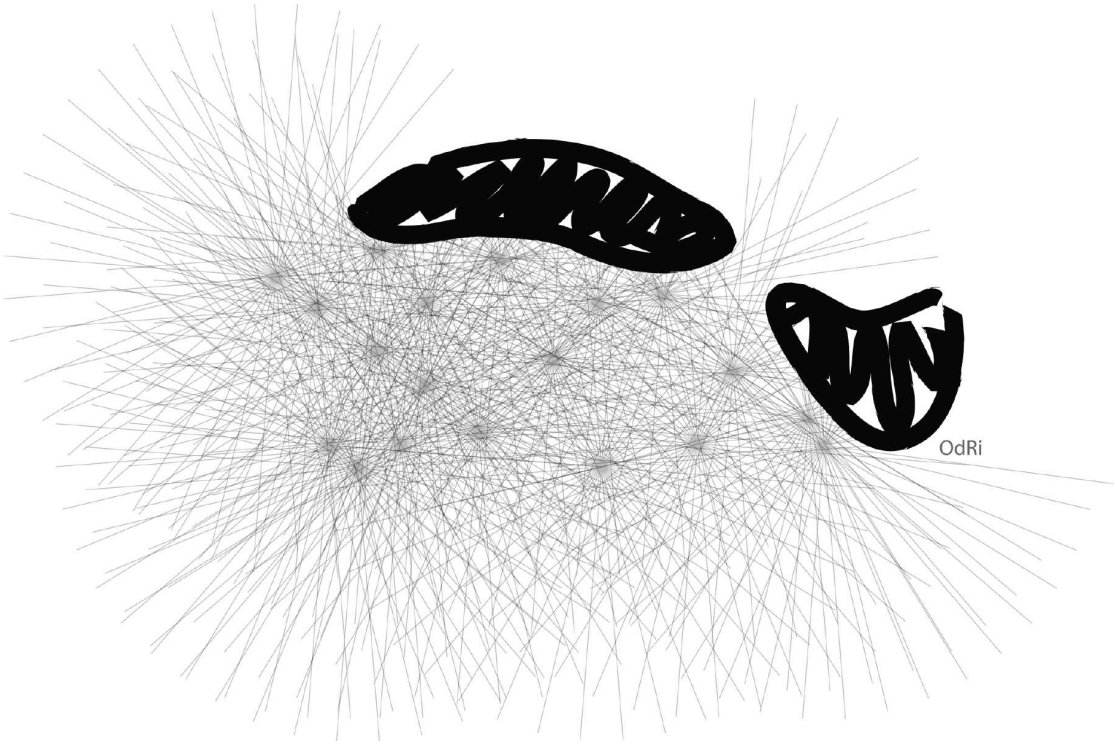
22. Implement backward value iteration and verify its correctness by reconstructing the costs obtained in Example 2.5. Test the implementation on some complicated examples.

23. For a planning problem under Formulation 2.3, implement both Dijkstra's algorithm and forward value iteration. Verify that these find the same plans. Comment on their differences in performance.

24. Consider grid-based problems for which there are mostly large, open rooms. Attempt to develop a multi-resolution search algorithm that first attempts to take larger steps, and only takes smaller steps as larger steps fail. Implement your ideas, conduct experiments on examples, and refine your approach accordingly.

Part II

Motion Planning



Overview of Part II: Motion Planning

Planning in Continuous Spaces

Part II makes the transition from discrete to continuous state spaces. Two alternative titles are appropriate for this part: 1) *motion planning*, or 2) *planning in continuous state spaces*. Chapters 3–8 are based on research from the field of motion planning, which has been building since the 1970s; therefore, the name *motion planning* is widely known to refer to the collection of models and algorithms that will be covered. On the other hand, it is convenient to also think of Part II as *planning in continuous spaces* because this is the primary distinction with respect to most other forms of planning.

In addition, motion planning will frequently refer to motions of a *robot* in a 2D or 3D *world* that contains *obstacles*. The robot could model an actual robot, or any other collection of moving bodies, such as humans or flexible molecules. A *motion plan* involves determining what motions are appropriate for the robot so that it reaches a goal state without colliding into obstacles. Recall the examples from Section 1.2.

Many issues that arose in Chapter 2 appear once again in motion planning. Two themes that may help to see the connection are as follows.

1. Implicit representations

A familiar theme from Chapter 2 is that planning algorithms must deal with *implicit* representations of the state space. In motion planning, this will become even more important because the state space is uncountably infinite. Furthermore, a complicated transformation exists between the world in which the models are defined and the space in which the planning occurs. Chapter 3 covers ways to model motion planning problems, which includes defining 2D and 3D geometric models and transforming them. Chapter 4 introduces the state space that arises for these problems. Following motion planning literature [657, 588], we will refer to this state space as the *configuration space*. The dimension of the configuration space corresponds to the number of degrees of freedom of the robot. Using the configuration space, motion planning will be viewed as a kind of search in a high-dimensional configuration space that contains implicitly represented obstacles. One additional complication is that configuration spaces have unusual topological structure that must be correctly characterized to ensure correct operation of planning algorithms. A motion plan will then be defined as a continuous path in the configuration space.

2. Continuous $\rightarrow$ discrete

A central theme throughout motion planning is to transform the continuous model into a discrete one. Due to this transformation, many algorithms from Chapter 2 are embedded in motion planning algorithms. There are two alternatives to

achieving this transformation, which are covered in Chapters 5 and 6, respectively. Chapter 6 covers *combinatorial motion planning*, which means that from the input model the algorithms build a discrete representation that *exactly* represents the original problem. This leads to *complete* planning approaches, which are guaranteed to find a solution when it exists, or correctly report failure if one does not exist. Chapter 5 covers *sampling-based motion planning*, which refers to algorithms that use collision detection methods to sample the configuration space and conduct discrete searches that utilize these samples. In this case, completeness is sacrificed, but it is often replaced with a weaker notion, such as *resolution completeness* or *probabilistic completeness*. It is important to study both Chapters 5 and 6 because each methodology has its strengths and weaknesses. Combinatorial methods can solve virtually any motion planning problem, and in some restricted cases, very elegant solutions may be efficiently constructed in practice. However, for the majority of "industrial-grade" motion planning problems, the running times and implementation difficulties of these algorithms make them unappealing. Sampling-based algorithms have fulfilled much of this need in recent years by solving challenging problems in several settings, such as automobile assembly, humanoid robot planning, and conformational analysis in drug design. Although the completeness guarantees are weaker, the efficiency and ease of implementation of these methods have bolstered interest in applying motion planning algorithms to a wide variety of applications.

Two additional chapters appear in Part II. Chapter 7 covers several extensions of the basic motion planning problem from the earlier chapters. These extensions include avoiding moving obstacles, multiple robot coordination, manipulation planning, and planning with closed kinematic chains. Algorithms that solve these problems build on the principles of earlier chapters, but each extension involves new challenges.

Chapter 8 is a transitional chapter that involves many elements of motion planning but is additionally concerned with gracefully recovering from unexpected deviations during execution. Although uncertainty in predicting the future is not explicitly modeled until Part III, Chapter 8 redefines the notion of a plan to be a function over state space, as opposed to being a path through it. The function gives the appropriate actions to take during exection, regardless of what configuration is entered. This allows the true configuration to drift away from the commanded configuration. In Part III such uncertainties will be explicitly modeled, but this comes at greater modeling and computational costs. It is worthwhile to develop effective ways to avoid this.

Chapter 3

Geometric Representations and Transformations

This chapter provides important background material that will be needed for Part II. Formulating and solving motion planning problems require defining and manipulating complicated geometric models of a system of bodies in space. Section 3.1 introduces geometric modeling, which focuses mainly on semi-algebraic modeling because it is an important part of Chapter 6. If your interest is mainly in Chapter 5, then understanding semi-algebraic models is not critical. Sections 3.2 and 3.3 describe how to transform a single body and a chain of bodies, respectively. This will enable the robot to "move." These sections are essential for understanding all of Part II and many sections beyond. It is expected that many readers will already have some or all of this background (especially Section 3.2, but it is included for completeness). Section 3.4 extends the framework for transforming chains of bodies to transforming trees of bodies, which allows modeling of complicated systems, such as humanoid robots and flexible organic molecules. Finally, Section 3.5 briefly covers transformations that do not assume each body is rigid.

3.1 Geometric Modeling

A wide variety of approaches and techniques for geometric modeling exist, and the particular choice usually depends on the application and the difficulty of the problem. In most cases, there are generally two alternatives: 1) a *boundary representation*, and 2) a *solid representation*. Suppose we would like to define a model of a planet. Using a boundary representation, we might write the equation of a sphere that roughly coincides with the planet's surface. Using a solid representation, we would describe the set of all points that are contained in the sphere. Both alternatives will be considered in this section.

The first step is to define the *world* $\mathcal{W}$ for which there are two possible choices: 1) a 2D world, in which $\mathcal{W} = \mathbb{R}^2$, and 2) a 3D world, in which $\mathcal{W} = \mathbb{R}^3$. These choices should be sufficient for most problems; however, one might also want to allow more complicated worlds, such as the surface of a sphere or even a higher

dimensional space. Such generalities are avoided in this book because their current applications are limited. Unless otherwise stated, the world generally contains two kinds of entities:

1. **Obstacles:** Portions of the world that are "permanently" occupied, for example, as in the walls of a building.

2. **Robots:** Bodies that are modeled geometrically and are controllable via a motion plan.

Based on the terminology, one obvious application is to model a robot that moves around in a building; however, many other possibilities exist. For example, the robot could be a flexible molecule, and the obstacles could be a folded protein. As another example, the robot could be a virtual human in a graphical simulation that involves obstacles (imagine the family of Doom-like video games).

This section presents a method for systematically constructing representations of obstacles and robots using a collection of primitives. Both obstacles and robots will be considered as (closed) subsets of $\mathcal{W}$. Let the *obstacle region* $\mathcal{O}$ denote the set of all points in $\mathcal{W}$ that lie in one or more obstacles; hence, $\mathcal{O} \subseteq \mathcal{W}$. The next step is to define a systematic way of representing $\mathcal{O}$ that has great expressive power while being computationally efficient. Robots will be defined in a similar way; however, this will be deferred until Section 3.2, where transformations of geometric bodies are defined.

3.1.1 Polygonal and Polyhedral Models

In this and the next subsection, a solid representation of $\mathcal{O}$ will be developed in terms of a combination of *primitives*. Each primitive H_i represents a subset of $\mathcal{W}$ that is easy to represent and manipulate in a computer. A complicated obstacle region will be represented by taking finite, Boolean combinations of primitives. Using set theory, this implies that $\mathcal{O}$ can also be defined in terms of a finite number of unions, intersections, and set differences of primitives.

Convex polygons First consider $\mathcal{O}$ for the case in which the obstacle region is a convex, polygonal subset of a 2D world, $\mathcal{W} = \mathbb{R}^2$. A subset $X \subset \mathbb{R}^n$ is called *convex* if and only if, for any pair of points in X, all points along the line segment that connects them are contained in X. More precisely, this means that for any $x_1, x_2 \in X$ and $\lambda \in [0, 1]$,

$$\lambda x_1 + (1 - \lambda) x_2 \in X. \tag{3.1}$$

Thus, interpolation between x_1 and x_2 always yields points in X. Intuitively, X contains no pockets or indentations. A set that is not convex is called *nonconvex* (as opposed to *concave*, which seems better suited for lenses).

A boundary representation of $\mathcal{O}$ is an m-sided polygon, which can be described using two kinds of *features*: vertices and edges. Every *vertex* corresponds to a "corner" of the polygon, and every *edge* corresponds to a line segment between a

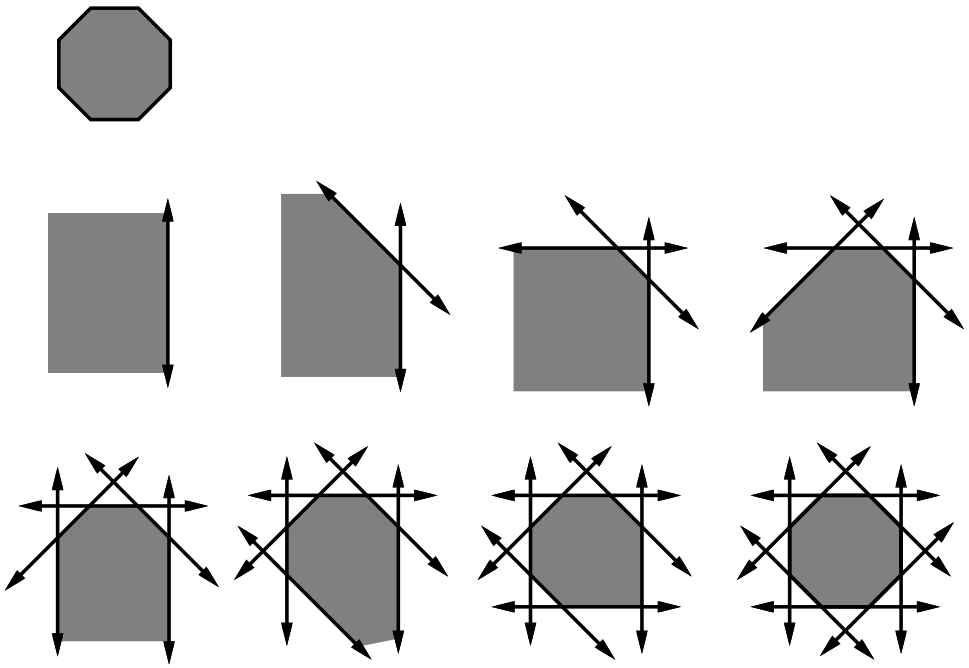


Figure 3.1: A convex polygonal region can be identified by the intersection of half-planes.

pair of vertices. The polygon can be specified by a sequence, (x_1, y_1), (x_2, y_2), ..., (x_m, y_m), of m points in $\mathbb{R}^2$, given in counterclockwise order.

A solid representation of $\mathcal{O}$ can be expressed as the intersection of m half-planes. Each half-plane corresponds to the set of all points that lie to one side of a line that is common to a polygon edge. Figure 3.1 shows an example of an octagon that is represented as the intersection of eight half-planes.

An edge of the polygon is specified by two points, such as (x_1, y_1) and (x_2, y_2). Consider the equation of a line that passes through (x_1, y_1) and (x_2, y_2). An equation can be determined of the form $ax + by + c = 0$, in which $a, b, c \in \mathbb{R}$ are constants that are determined from x_1, y_1, x_2, and y_2. Let $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ be the function given by $f(x, y) = ax + by + c$. Note that $f(x, y) < 0$ on one side of the line, and $f(x, y) > 0$ on the other. (In fact, f may be interpreted as a signed Euclidean distance from (x, y) to the line.) The sign of $f(x, y)$ indicates a half-plane that is bounded by the line, as depicted in Figure 3.2. Without loss of generality, assume that $f(x, y)$ is defined so that $f(x, y) < 0$ for all points to the left of the edge from (x_1, y_1) to (x_2, y_2) (if it is not, then multiply $f(x, y)$ by -1).

Let $f_i(x, y)$ denote the f function derived from the line that corresponds to the edge from (x_i, y_i) to (x_{i+1}, y_{i+1}) for $1 \leq i < m$. Let $f_m(x, y)$ denote the line equation that corresponds to the edge from (x_m, y_m) to (x_1, y_1). Let a *half-plane* H_i for $1 \leq i \leq m$ be defined as a subset of $\mathcal{W}$:

$$H_i = \{(x, y) \in \mathcal{W} \mid f_i(x, y) \leq 0\}. \tag{3.2}$$

Above, H_i is a primitive that describes the set of all points on one side of the

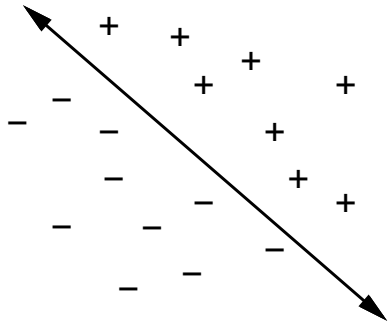


Figure 3.2: The sign of the $f(x, y)$ partitions $\mathbb{R}^2$ into three regions: two half-planes given by $f(x, y) < 0$ and $f(x, y) > 0$, and the line $f(x, y) = 0$.

line $f_i(x, y) = 0$ (including the points on the line). A convex, m-sided, polygonal obstacle region $\mathcal{O}$ is expressed as

$$\mathcal{O} = H_1 \cap H_2 \cap \cdots \cap H_m. \tag{3.3}$$

Nonconvex polygons The assumption that $\mathcal{O}$ is convex is too limited for most applications. Now suppose that $\mathcal{O}$ is a nonconvex, polygonal subset of $\mathcal{W}$. In this case $\mathcal{O}$ can be expressed as

$$\mathcal{O} = \mathcal{O}_1 \cup \mathcal{O}_2 \cup \cdots \cup \mathcal{O}_n, \tag{3.4}$$

in which each $\mathcal{O}_i$ is a convex, polygonal set that is expressed in terms of half-planes using (3.3). Note that $\mathcal{O}_i$ and $\mathcal{O}_j$ for $i \neq j$ need not be disjoint. Using this representation, very complicated obstacle regions in $\mathcal{W}$ can be defined. Although these regions may contain multiple components and holes, if $\mathcal{O}$ is bounded (i.e., $\mathcal{O}$ will fit inside of a big enough rectangular box), its boundary will consist of linear segments.

In general, more complicated representations of $\mathcal{O}$ can be defined in terms of any finite combination of unions, intersections, and set differences of primitives; however, it is always possible to simplify the representation into the form given by (3.3) and (3.4). A set difference can be avoided by redefining the primitive. Suppose the model requires removing a set defined by a primitive H_i that contains[1] $f_i(x, y) < 0$. This is equivalent to keeping all points such that $f_i(x, y) \geq 0$, which is equivalent to $-f_i(x, y) \leq 0$. This can be used to define a new primitive H_i', which when taken in union with other sets, is equivalent to the removal of H_i. Given a complicated combination of primitives, once set differences are removed, the expression can be simplified into a finite union of finite intersections by applying Boolean algebra laws.

[1] In this section, we want the resulting set to include all of the points along the boundary. Therefore, $<$ is used to model a set for removal, as opposed to $\leq$.

Note that the representation of a nonconvex polygon is not unique. There are many ways to decompose $\mathcal{O}$ into convex components. The decomposition should be carefully selected to optimize computational performance in whatever algorithms that model will be used. In most cases, the components may even be allowed to overlap. Ideally, it seems that it would be nice to represent $\mathcal{O}$ with the minimum number of primitives, but automating such a decomposition may lead to an NP-hard problem (see Section 6.5.1 for a brief overview of NP-hardness). One efficient, practical way to decompose $\mathcal{O}$ is to apply the vertical cell decomposition algorithm, which will be presented in Section 6.2.2.

Defining a logical predicate What is the value of the previous representation? As a simple example, we can define a logical predicate that serves as a collision detector. Recall from Section 2.4.1 that a predicate is a Boolean-valued function. Let ϕ be a predicate defined as $\phi : \mathcal{W} \rightarrow \{\text{TRUE}, \text{FALSE}\}$, which returns TRUE for a point in $\mathcal{W}$ that lies in $\mathcal{O}$, and FALSE otherwise. For a line given by $f(x, y) = 0$, let $e(x, y)$ denote a logical predicate that returns TRUE if $f(x, y) \leq 0$, and FALSE otherwise.

A predicate that corresponds to a convex polygonal region is represented by a logical conjunction,

$$\alpha(x, y) = e_1(x, y) \wedge e_2(x, y) \wedge \cdots \wedge e_m(x, y). \tag{3.5}$$

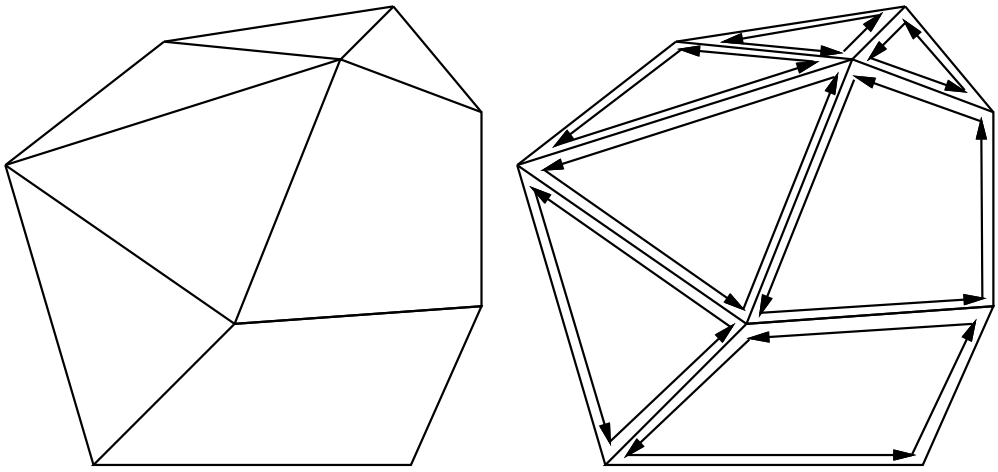
The predicate $\alpha(x, y)$ returns TRUE if the point (x, y) lies in the convex polygonal region, and FALSE otherwise. An obstacle region that consists of n convex polygons is represented by a logical disjunction of conjuncts,

$$\phi(x, y) = \alpha_1(x, y) \vee \alpha_2(x, y) \vee \cdots \vee \alpha_n(x, y). \tag{3.6}$$

Although more efficient methods exist, ϕ can check whether a point (x, y) lies in $\mathcal{O}$ in time $O(n)$, in which n is the number of primitives that appear in the representation of $\mathcal{O}$ (each primitive is evaluated in constant time).

Note the convenient connection between a logical predicate representation and a set-theoretic representation. Using the logical predicate, the unions and intersections of the set-theoretic representation are replaced by logical ORs and ANDs. It is well known from Boolean algebra that any complicated logical sentence can be reduced to a logical disjunction of conjunctions (this is often called "sum of products" in computer engineering). This is equivalent to our previous statement that $\mathcal{O}$ can always be represented as a union of intersections of primitives.

Polyhedral models For a 3D world, $\mathcal{W} = \mathbb{R}^3$, and the previous concepts can be nicely generalized from the 2D case by replacing polygons with polyhedra and replacing half-plane primitives with half-space primitives. A boundary representation can be defined in terms of three features: vertices, edges, and faces. Every face is a "flat" polygon embedded in $\mathbb{R}^3$. Every edge forms a boundary between two faces. Every vertex forms a boundary between three or more edges.



(a) (b)

Figure 3.3: (a) A polyhedron can be described in terms of faces, edges, and vertices. (b) The edges of each face can be stored in a circular list that is traversed in counterclockwise order with respect to the outward normal vector of the face.

Several data structures have been proposed that allow one to conveniently "walk" around the polyhedral features. For example, the *doubly connected edge list* [264] data structure contains three types of records: faces, half-edges, and vertices. Intuitively, a half-edge is a directed edge. Each vertex record holds the point coordinates and a pointer to an arbitrary half-edge that touches the vertex. Each face record contains a pointer to an arbitrary half-edge on its boundary. Each face is bounded by a circular list of half-edges. There is a pair of directed half-edge records for each edge of the polyhedon. Each half-edge is shown as an arrow in Figure 3.3b. Each half-edge record contains pointers to five other records: 1) the vertex from which the half-edge originates; 2) the "twin" half-edge, which bounds the neighboring face, and has the opposite direction; 3) the face that is bounded by the half-edge; 4) the next element in the circular list of edges that bound the face; and 5) the previous element in the circular list of edges that bound the face. Once all of these records have been defined, one can conveniently traverse the structure of the polyhedron.

Now consider a solid representation of a polyhedron. Suppose that $\mathcal{O}$ is a convex polyhedron, as shown in Figure 3.3. A solid representation can be constructed from the vertices. Each face of $\mathcal{O}$ has at least three vertices along its boundary. Assuming these vertices are not collinear, an equation of the plane that passes through them can be determined of the form

$$ax + by + cz + d = 0, \tag{3.7}$$

in which $a, b, c, d \in \mathbb{R}$ are constants.

Once again, f can be constructed, except now $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ and

$$f(x, y, z) = ax + by + cz + d. \tag{3.8}$$

Let m be the number of faces. For each face of $\mathcal{O}$, a *half-space* H_i is defined as a subset of $\mathcal{W}$:

$$H_i = \{(x, y, z) \in \mathcal{W} \mid f_i(x, y, z) \leq 0\}. \tag{3.9}$$

It is important to choose f_i so that it takes on negative values inside of the polyhedron. In the case of a polygonal model, it was possible to consistently define f_i by proceeding in counterclockwise order around the boundary. In the case of a polyhedron, the half-edge data structure can be used to obtain for each face the list of edges that form its boundary in counterclockwise order. Figure 3.3b shows the edge ordering for each face. For every edge, the arrows point in opposite directions, as required by the half-edge data structure. The equation for each face can be consistently determined as follows. Choose three consecutive vertices, p_1, p_2, p_3 (they must not be collinear) in counterclockwise order on the boundary of the face. Let v_{12} denote the vector from p_1 to p_2, and let v_{23} denote the vector from p_2 to p_3. The cross product $v = v_{12} \times v_{23}$ always yields a vector that points out of the polyhedron and is normal to the face. Recall that the vector $[a \;\; b \;\; c]$ is parallel to the normal to the plane. If its components are chosen as $a = v[1]$, $b = v[2]$, and $c = v[3]$, then $f(x, y, z) \leq 0$ for all points in the half-space that contains the polyhedron.

As in the case of a polygonal model, a convex polyhedron can be defined as the intersection of a finite number of half-spaces, one for each face. A nonconvex polyhedron can be defined as the union of a finite number of convex polyhedra. The predicate $\phi(x, y, z)$ can be defined in a similar manner, in this case yielding TRUE if $(x, y, z) \in \mathcal{O}$, and FALSE otherwise.

3.1.2 Semi-Algebraic Models

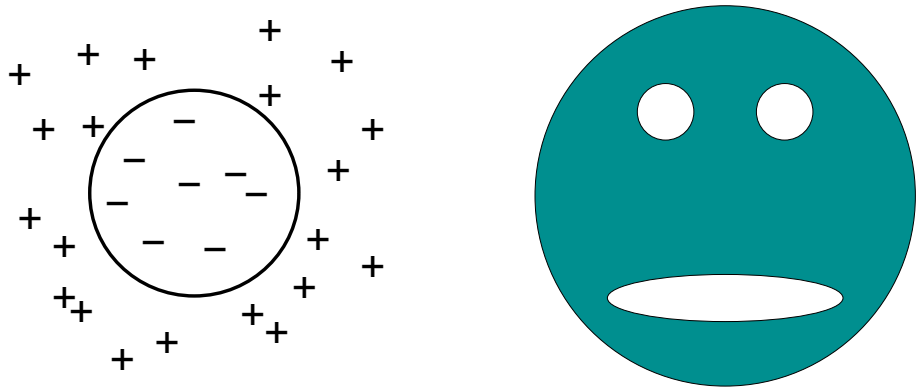
In both the polygonal and polyhedral models, f was a linear function. In the case of a semi-algebraic model for a 2D world, f can be any polynomial with real-valued coefficients and variables x and y. For a 3D world, f is a polynomial with variables x, y, and z. The class of semi-algebraic models includes both polygonal and polyhedral models, which use first-degree polynomials. A point set determined by a single polynomial primitive is called an *algebraic set*; a point set that can be obtained by a finite number of unions and intersections of algebraic sets is called a *semi-algebraic set*.

Consider the case of a 2D world. A solid representation can be defined using *algebraic primitives* of the form

$$H = \{(x, y) \in \mathcal{W} \mid f(x, y) \leq 0\}. \tag{3.10}$$

As an example, let $f = x^2 + y^2 - 4$. In this case, H represents a disc of radius 2 that is centered at the origin. This corresponds to the set of points (x, y) for which $f(x, y) \leq 0$, as depicted in Figure 3.4a.

Example 3.1 (Gingerbread Face) Consider constructing a model of the shaded region shown in Figure 3.4b. Let the center of the outer circle have radius r_1 and



(a) (b)

Figure 3.4: (a) Once again, f is used to partition $\mathbb{R}^2$ into two regions. In this case, the algebraic primitive represents a disc-shaped region. (b) The shaded "face" can be exactly modeled using only four algebraic primitives.

be centered at the origin. Suppose that the "eyes" have radius r_2 and r_3 and are centered at (x_2, y_2) and (x_3, y_3), respectively. Let the "mouth" be an ellipse with major axis a and minor axis b and is centered at $(0, y_4)$. The functions are defined as

$$\begin{aligned} f_1 &= x^2 + y^2 - r_1^2, \\ f_2 &= -\big((x - x_2)^2 + (y - y_2)^2 - r_2^2\big), \\ f_3 &= -\big((x - x_3)^2 + (y - y_3)^2 - r_3^2\big), \\ f_4 &= -\big(x^2/a^2 + (y - y_4)^2/b^2 - 1\big). \end{aligned} \tag{3.11}$$

For f_2, f_3, and f_4, the familiar circle and ellipse equations were multiplied by -1 to yield algebraic primitives for all points outside of the circle or ellipse. The shaded region $\mathcal{O}$ is represented as

$$\mathcal{O} = H_1 \cap H_2 \cap H_3 \cap H_4. \tag{3.12}$$

■

In the case of semi-algebraic models, the intersection of primitives does not necessarily result in a convex subset of $\mathcal{W}$. In general, however, it might be necessary to form $\mathcal{O}$ by taking unions and intersections of algebraic primitives.

A logical predicate, $\phi(x, y)$, can once again be formed, and collision checking is still performed in time that is linear in the number of primitives. Note that it is still very efficient to evaluate every primitive; f is just a polynomial that is evaluated on the point (x, y, z).

The semi-algebraic formulation generalizes easily to the case of a 3D world. This results in algebraic primitives of the form

$$H = \{(x, y, z) \in \mathcal{W} \mid f(x, y, z) \leq 0\}, \tag{3.13}$$

which can be used to define a solid representation of a 3D obstacle $\mathcal{O}$ and a logical predicate ϕ.

Equations (3.10) and (3.13) are sufficient to express any model of interest. One may define many other primitives based on different relations, such as $f(x, y, z) \geq 0$, $f(x, y, z) = 0$, $f(x, y, z) < 0$, $f(x, y, z) = 0$, and $f(x, y, z) \neq 0$; however, most of them do not enhance the set of models that can be expressed. They might, however, be more convenient in certain contexts. To see that some primitives do not allow new models to be expressed, consider the primitive

$$H = \{(x, y, z) \in \mathcal{W} \mid f(x, y, z) \geq 0\}. \tag{3.14}$$

The right part may be alternatively represented as $-f(x, y, z) \leq 0$, and $-f$ may be considered as a new polynomial function of x, y, and z. For an example that involves the $=$ relation, consider the primitive

$$H = \{(x, y, z) \in \mathcal{W} \mid f(x, y, z) = 0\}. \tag{3.15}$$

It can instead be constructed as $H = H_1 \cap H_2$, in which

$$H_1 = \{(x, y, z) \in \mathcal{W} \mid f(x, y, z) \leq 0\} \tag{3.16}$$

and

$$H_2 = \{(x, y, z) \in \mathcal{W} \mid -f(x, y, z) \leq 0\}. \tag{3.17}$$

The relation $<$ does add some expressive power if it is used to construct primitives.[2] It is needed to construct models that do not include the outer boundary (for example, the set of all points *inside* of a sphere, which does not include points *on* the sphere). These are generally called *open sets* and are defined Chapter 4.

3.1.3 Other Models

The choice of a model often depends on the types of operations that will be performed by the planning algorithm. For combinatorial motion planning methods, to be covered in Chapter 6, the particular representation is critical. On the other hand, for sampling-based planning methods, to be covered in Chapter 5, the particular representation is important only to the collision detection algorithm, which is treated as a "black box" as far as planning is concerned. Therefore, the models given in the remainder of this section are more likely to appear in sampling-based approaches and may be invisible to the designer of a planning algorithm (although it is never wise to forget completely about the representation).

[2] An alternative that yields the same expressive power is to still use $\leq$, but allow set complements, in addition to unions and intersections.

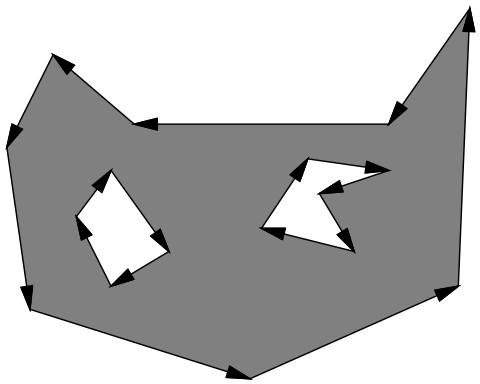


Figure 3.5: A polygon with holes can be expressed by using different orientations: counterclockwise for the outer boundary and clockwise for the hole boundaries. Note that the shaded part is always to the left when following the arrows.

Nonconvex polygons and polyhedra The method in Section 3.1.1 required nonconvex polygons to be represented as a union of convex polygons. Instead, a boundary representation of a nonconvex polygon may be directly encoded by listing vertices in a specific order; assume that counterclockwise order is used. Each polygon of m vertices may be encoded by a list of the form (x_1, y_1), (x_2, y_2), ..., (x_m, y_m). It is assumed that there is an edge between each (x_i, y_i) and (x_{i+1}, y_{i+1}) for each i from 1 to $m-1$, and also an edge between (x_m, y_m) and (x_1, y_1). Ordinarily, the vertices should be chosen in a way that makes the polygon *simple*, meaning that no edges intersect. In this case, there is a well-defined interior of the polygon, which is to the left of every edge, if the vertices are listed in counterclockwise order.

What if a polygon has a hole in it? In this case, the boundary of the hole can be expressed as a polygon, but with its vertices appearing in the clockwise direction. To the left of each edge is the interior of the outer polygon, and to the right is the hole, as shown in Figure 3.5

Although the data structures are a little more complicated for three dimensions, boundary representations of nonconvex polyhedra may be expressed in a similar manner. In this case, instead of an edge list, one must specify faces, edges, and vertices, with pointers that indicate their incidence relations. Consistent orientations must also be chosen, and holes may be modeled once again by selecting opposite orientations.

3D triangles Suppose $\mathcal{W} = \mathbb{R}^3$. One of the most convenient geometric models to express is a set of triangles, each of which is specified by three points, (x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3). This model has been popular in computer graphics because graphics acceleration hardware primarily uses triangle primitives. It is assumed that the interior of the triangle is part of the model. Thus, two triangles are considered as "colliding" if one pokes into the interior of another. This model offers great flexibility because there are no constraints on the way in which triangles must

Figure 3.6: Triangle strips and triangle fans can reduce the number of redundant points.

be expressed; however, this is also one of the drawbacks. There is no coherency that can be exploited to easily declare whether a point is "inside" or "outside" of a 3D obstacle. If there is at least some coherency, then it is sometimes preferable to reduce redundancy in the specification of triangle coordinates (many triangles will share the same corners). Representations that remove this redundancy are called a *triangle strip*, which is a sequence of triangles such that each adjacent pair shares a common edge, and a *triangle fan*, which is a triangle strip in which all triangles share a common vertex. See Figure 3.6.

Nonuniform rational B-splines (NURBS) These are used in many engineering design systems to allow convenient design and adjustment of curved surfaces, in applications such as aircraft or automobile body design. In contrast to semi-algebraic models, which are implicit equations, NURBS and other splines are parametric equations. This makes computations such as rendering easier; however, others, such as collision detection, become more difficult. These models may be defined in any dimension. A brief 2D formulation is given here.

A curve can be expressed as

$$C(u) = \frac{\sum_{i=0}^{n} w_i P_i N_{i,k}(u)}{\sum_{i=0}^{n} w_i N_{i,k}(u)}, \tag{3.18}$$

in which $w_i \in \mathbb{R}$ are *weights* and P_i are control points. The $N_{i,k}$ are normalized basis functions of degree k, which can be expressed recursively as

$$N_{i,k}(u) = \left(\frac{u - t_i}{t_{i+k} - t_i}\right) N_{i,k-1}(u) + \left(\frac{t_{i+k+1} - u}{t_{i+k+1} - t_{i+1}}\right) N_{i+1,k-1}(u). \tag{3.19}$$

The basis of the recursion is $N_{i,0}(u) = 1$ if $t_i \leq u < t_{i+1}$, and $N_{i,0}(u) = 0$ otherwise. A *knot vector* is a nondecreasing sequence of real values, $\{t_0, t_1, \ldots, t_m\}$, that controls the intervals over which certain basic functions take effect.

Bitmaps For either $\mathcal{W} = \mathbb{R}^2$ or $\mathcal{W} = \mathbb{R}^3$, it is possible to discretize a bounded portion of the world into rectangular cells that may or may not be occupied. The resulting model looks very similar to Example 2.1. The resolution of this discretization determines the number of cells per axis and the quality of the approximation. The representation may be considered as a binary image in which

each "1" in the image corresponds to a rectangular region that contains at least one point of $\mathcal{O}$, and "0" represents those that do not contain any of $\mathcal{O}$. Although bitmaps do not have the elegance of the other models, they often arise in applications. One example is a digital map constructed by a mobile robot that explores an environment with its sensors. One generalization of bitmaps is a *gray-scale map* or *occupancy grid*. In this case, a numerical value may be assigned to each cell, indicating quantities such as "the probability that an obstacle exists" or the "expected difficulty of traversing the cell." The latter interpretation is often used in terrain maps for navigating planetary rovers.

Superquadrics Instead of using polynomials to define f_i, many generalizations can be constructed. One popular primitive is a *superquadric*, which generalizes quadric surfaces. One example is a superellipsoid, which is given for $\mathcal{W} = \mathbb{R}^3$ by

$$\left(|x/a|^{n_1} + |y/b|^{n_2}\right)^{n_1/n_2} + |z/c|^{n_1} - 1 \leq 0, \tag{3.20}$$

in which $n_1 \geq 2$ and $n_2 \geq 2$. If $n_1 = n_2 = 2$, an ellipse is generated. As n_1 and n_2 increase, the superellipsoid becomes shaped like a box with rounded corners.

Generalized cylinders A *generalized cylinder* is a generalization of an ordinary cylinder. Instead of being limited to a line, the center axis is a continuous *spine curve*, $(x(s), y(s), z(s))$, for some parameter $s \in [0, 1]$. Instead of a constant radius, a radius function $r(s)$ is defined along the spine. The value $r(s)$ is the radius of the circle obtained as the cross section of the generalized cylinder at the point $(x(s), y(s), z(s))$. The normal to the cross-section plane is the tangent to the spine curve at s.

3.2 Rigid-Body Transformations

Any of the techniques from Section 3.1 can be used to define both the obstacle region and the robot. Let $\mathcal{O}$ refer to the *obstacle region*, which is a subset of $\mathcal{W}$. Let $\mathcal{A}$ refer to the robot, which is a subset of $\mathbb{R}^2$ or $\mathbb{R}^3$, matching the dimension of $\mathcal{W}$. Although $\mathcal{O}$ remains fixed in the world, $\mathcal{W}$, motion planning problems will require "moving" the robot, $\mathcal{A}$.

3.2.1 General Concepts

Before giving specific transformations, it will be helpful to define them in general to avoid confusion in later parts when intuitive notions might fall apart. Suppose that a rigid robot, $\mathcal{A}$, is defined as a subset of $\mathbb{R}^2$ or $\mathbb{R}^3$. A *rigid-body transformation* is a function, $h : \mathcal{A} \to \mathcal{W}$, that maps every point of $\mathcal{A}$ into $\mathcal{W}$ with two requirements: 1) The distance between any pair of points of $\mathcal{A}$ must be preserved, and 2) the orientation of $\mathcal{A}$ must be preserved (no "mirror images").

Using standard function notation, $h(a)$ for some $a \in \mathcal{A}$ refers to the point in $\mathcal{W}$ that is "occupied" by a. Let

$$h(\mathcal{A}) = \{h(a) \in \mathcal{W} \mid a \in \mathcal{A}\}, \tag{3.21}$$

which is the image of h and indicates all points in $\mathcal{W}$ occupied by the transformed robot.

Transforming the robot model Consider transforming a robot model. If $\mathcal{A}$ is expressed by naming specific points in $\mathbb{R}^2$, as in a boundary representation of a polygon, then each point is simply transformed from a to $h(a) \in \mathcal{W}$. In this case, it is straightforward to transform the entire model using h. However, there is a slight complication if the robot model is expressed using primitives, such as

$$H_i = \{a \in \mathbb{R}^2 \mid f_i(a) \leq 0\}. \tag{3.22}$$

This differs slightly from (3.2) because the robot is defined in $\mathbb{R}^2$ (which is not necessarily $\mathcal{W}$), and also a is used to denote a point $(x, y) \in \mathcal{A}$. Under a transformation h, the primitive is transformed as

$$h(H_i) = \{h(a) \in \mathcal{W} \mid f_i(a) \leq 0\}. \tag{3.23}$$

To transform the primitive completely, however, it is better to directly name points in $w \in \mathcal{W}$, as opposed to $h(a) \in \mathcal{W}$. Using the fact that $a = h^{-1}(w)$, this becomes

$$h(H_i) = \{w \in \mathcal{W} \mid f_i(h^{-1}(w)) \leq 0\}, \tag{3.24}$$

in which the inverse of h appears in the right side because the original point $a \in A$ needs to be recovered to evaluate f_i. Therefore, it is important to be careful because either h or h^{-1} may be required to transform the model. This will be observed in more specific contexts in some coming examples.

A parameterized family of transformations It will become important to study families of transformations, in which some parameters are used to select the particular transformation. Therefore, it makes sense to generalize h to accept two variables: a parameter vector, $q \in \mathbb{R}^n$, along with $a \in \mathcal{A}$. The resulting transformed point a is denoted by $h(q, a)$, and the entire robot is transformed to $h(q, \mathcal{A}) \subset \mathcal{W}$.

The coming material will use the following shorthand notation, which requires the specific h to be inferred from the context. Let $h(q, a)$ be shortened to $a(q)$, and let $h(q, \mathcal{A})$ be shortened to $\mathcal{A}(q)$. This notation makes it appear that by adjusting the parameter q, the robot $\mathcal{A}$ travels around in $\mathcal{W}$ as different transformations are selected from the predetermined family. This is slightly abusive notation, but it is convenient. The expression $\mathcal{A}(q)$ can be considered as a set-valued function that yields the set of points in $\mathcal{W}$ that are occupied by $\mathcal{A}$ when it is transformed by q. Most of the time the notation does not cause trouble, but when it does, it is helpful to remember the definitions from this section, especially when trying to determine whether h or h^{-1} is needed.

Defining frames It was assumed so far that $\mathcal{A}$ is defined in $\mathbb{R}^2$ or $\mathbb{R}^3$, but before it is transformed, it is not considered to be a subset of $\mathcal{W}$. The transformation h *places* the robot in $\mathcal{W}$. In the coming material, it will be convenient to indicate this distinction using coordinate frames. The origin and coordinate basis vectors of $\mathcal{W}$ will be referred to as the *world frame*.[3] Thus, any point $w \in \mathcal{W}$ is expressed in terms of the world frame.

The coordinates used to define $\mathcal{A}$ are initially expressed in the *body frame*, which represents the origin and coordinate basis vectors of $\mathbb{R}^2$ or $\mathbb{R}^3$. In the case of $\mathcal{A} \subset \mathbb{R}^2$, it can be imagined that the body frame is painted on the robot. Transforming the robot is equivalent to converting its model from the body frame to the world frame. This has the effect of *placing*[4] $\mathcal{A}$ into $\mathcal{W}$ at some position and orientation. When multiple bodies are covered in Section 3.3, each body will have its own body frame, and transformations require expressing all bodies with respect to the world frame.

3.2.2 2D Transformations

Translation A rigid robot $\mathcal{A} \subset \mathbb{R}^2$ is *translated* by using two parameters, $x_t, y_t \in \mathbb{R}$. Using definitions from Section 3.2.1, $q = (x_t, y_t)$, and h is defined as

$$h(x, y) = (x + x_t, y + y_t). \tag{3.25}$$

A boundary representation of $\mathcal{A}$ can be translated by transforming each vertex in the sequence of polygon vertices using (3.25). Each point, (x_i, y_i), in the sequence is replaced by $(x_i + x_t, y_i + y_t)$.

Now consider a solid representation of $\mathcal{A}$, defined in terms of primitives. Each primitive of the form

$$H_i = \{(x, y) \in \mathbb{R}^2 \mid f(x, y) \leq 0\} \tag{3.26}$$

is transformed to

$$h(H_i) = \{(x, y) \in \mathcal{W} \mid f(x - x_t, y - y_t) \leq 0\}. \tag{3.27}$$

Example 3.2 (Translating a Disc) For example, suppose the robot is a disc of unit radius, centered at the origin. It is modeled by a single primitive,

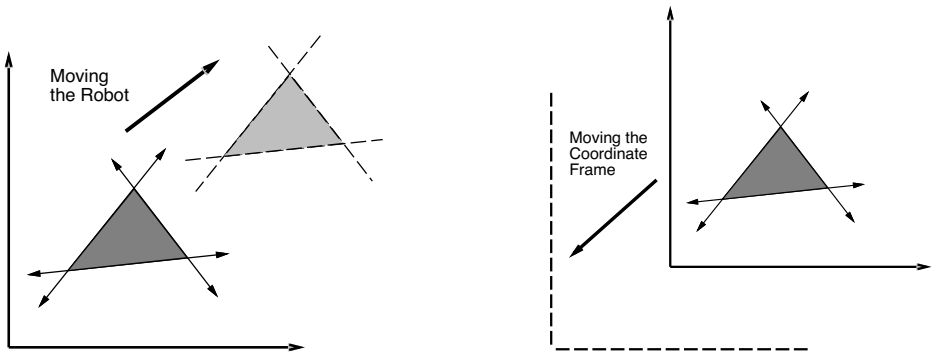
$$H_i = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 - 1 \leq 0\}. \tag{3.28}$$

Suppose $\mathcal{A} = H_i$ is translated x_t units in the x direction and y_t units in the y direction. The transformed primitive is

$$h(H_i) = \{(x, y) \in \mathcal{W} \mid (x - x_t)^2 + (y - y_t)^2 - 1 \leq 0\}, \tag{3.29}$$

[3] The world frame serves the same purpose as an inertial frame in Newtonian mechanics. Intuitively, it is a frame that remains fixed and from which all measurements are taken. See Section 13.3.1.

[4] Technically, this placement is a function called an *orientation-preserving isometric embedding*.



(a) Translation of the robot (b) Translation of the frame

Figure 3.7: Every transformation has two interpretations.

which is the familiar equation for a disc centered at (x_t, y_t). In this example, the inverse, h^{-1} is used, as described in Section 3.2.1. ■

The translated robot is denoted as $\mathcal{A}(x_t, y_t)$. Translation by $(0, 0)$ is the *identity transformation*, which results in $\mathcal{A}(0, 0) = \mathcal{A}$, if it is assumed that $\mathcal{A} \subset \mathcal{W}$ (recall that $\mathcal{A}$ does not necessarily have to be initially embedded in $\mathcal{W}$). It will be convenient to use the term *degrees of freedom* to refer to the maximum number of independent parameters that are needed to completely characterize the transformation applied to the robot. If the set of allowable values for x_t and y_t forms a two-dimensional subset of $\mathbb{R}^2$, then the degrees of freedom is two.

Suppose that $\mathcal{A}$ is defined directly in $\mathcal{W}$ with translation. As shown in Figure 3.7, there are two interpretations of a rigid-body transformation applied to $\mathcal{A}$: 1) The world frame remains fixed and the robot is transformed; 2) the robot remains fixed and the world frame is translated. The first one characterizes the effect of the transformation from a fixed world frame, and the second one indicates how the transformation appears from the robot's perspective. Unless stated otherwise, the first interpretation will be used when we refer to motion planning problems because it often models a robot moving in a physical world. Numerous books cover coordinate transformations under the second interpretation. This has been known to cause confusion because the transformations may sometimes appear "backward" from what is desired in motion planning.

Rotation The robot, $\mathcal{A}$, can be *rotated* counterclockwise by some angle $\theta \in [0, 2\pi)$ by mapping every $(x, y) \in \mathcal{A}$ as

$$(x, y) \mapsto (x \cos\theta - y \sin\theta, \; x \sin\theta + y \cos\theta). \tag{3.30}$$

Using a 2×2 rotation matrix,

$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}, \tag{3.31}$$

the transformation can be written as

$$\begin{pmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{pmatrix} = R(\theta) \begin{pmatrix} x \\ y \end{pmatrix}. \tag{3.32}$$

Using the notation of Section 3.2.1, $R(\theta)$ becomes $h(q)$, for which $q = \theta$. For linear transformations, such as the one defined by (3.32), recall that the column vectors represent the basis vectors of the new coordinate frame. The column vectors of $R(\theta)$ are unit vectors, and their inner product (or dot product) is zero, indicating that they are orthogonal. Suppose that the x and y coordinate axes, which represent the body frame, are "painted" on $\mathcal{A}$. The columns of $R(\theta)$ can be derived by considering the resulting directions of the x- and y-axes, respectively, after performing a counterclockwise rotation by the angle θ. This interpretation generalizes nicely for higher dimensional rotation matrices.

Note that the rotation is performed about the origin. Thus, when defining the model of $\mathcal{A}$, the origin should be placed at the intended axis of rotation. Using the semi-algebraic model, the entire robot model can be rotated by transforming each primitive, yielding $\mathcal{A}(\theta)$. The inverse rotation, $R(-\theta)$, must be applied to each primitive.

Combining translation and rotation Suppose a rotation by θ is performed, followed by a translation by x_t, y_t. This can be used to place the robot in any desired position and orientation. Note that translations and rotations do not commute! If the operations are applied successively, each $(x, y) \in \mathcal{A}$ is transformed to

$$\begin{pmatrix} x\cos\theta - y\sin\theta + x_t \\ x\sin\theta + y\cos\theta + y_t \end{pmatrix}. \tag{3.33}$$

The following matrix multiplication yields the same result for the first two vector components:

$$\begin{pmatrix} \cos\theta & -\sin\theta & x_t \\ \sin\theta & \cos\theta & y_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x\cos\theta - y\sin\theta + x_t \\ x\sin\theta + y\cos\theta + y_t \\ 1 \end{pmatrix}. \tag{3.34}$$

This implies that the 3×3 matrix,

$$T = \begin{pmatrix} \cos\theta & -\sin\theta & x_t \\ \sin\theta & \cos\theta & y_t \\ 0 & 0 & 1 \end{pmatrix}, \tag{3.35}$$

represents a rotation followed by a translation. The matrix T will be referred to as a *homogeneous transformation matrix*. It is important to remember that T

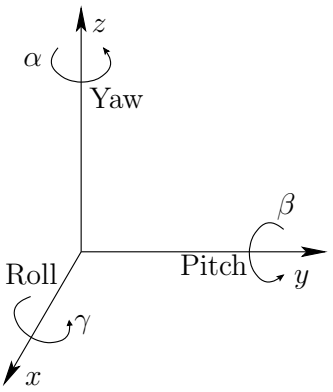


Figure 3.8: Any three-dimensional rotation can be described as a sequence of yaw, pitch, and roll rotations.

represents a rotation *followed by* a translation (not the other way around). Each primitive can be transformed using the inverse of T, resulting in a transformed solid model of the robot. The transformed robot is denoted by $\mathcal{A}(x_t, y_t, \theta)$, and in this case there are three degrees of freedom. The homogeneous transformation matrix is a convenient representation of the combined transformations; therefore, it is frequently used in robotics, mechanics, computer graphics, and elsewhere. It is called homogeneous because over $\mathbb{R}^3$ it is just a linear transformation without any translation. The trick of increasing the dimension by one to absorb the translational part is common in projective geometry [804].

3.2.3 3D Transformations

Rigid-body transformations for the 3D case are conceptually similar to the 2D case; however, the 3D case appears more difficult because rotations are significantly more complicated.

3D translation The robot, $\mathcal{A}$, is *translated* by some $x_t, y_t, z_t \in \mathbb{R}$ using

$$(x, y, z) \mapsto (x + x_t, y + y_t, z + z_t). \tag{3.36}$$

A primitive of the form

$$H_i = \{(x, y, z) \in \mathcal{W} \mid f_i(x, y, z) \leq 0\} \tag{3.37}$$

is transformed to

$$\{(x, y, z) \in \mathcal{W} \mid f_i(x - x_t, y - y_t, z - z_t) \leq 0\}. \tag{3.38}$$

The translated robot is denoted as $\mathcal{A}(x_t, y_t, z_t)$.

Yaw, pitch, and roll rotations A 3D body can be rotated about three orthogonal axes, as shown in Figure 3.8. Borrowing aviation terminology, these rotations will be referred to as yaw, pitch, and roll:

1. A *yaw* is a counterclockwise rotation of α about the z-axis. The rotation matrix is given by

$$R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{3.39}$$

   Note that the upper left entries of $R_z(\alpha)$ form a 2D rotation applied to the x and y coordinates, whereas the z coordinate remains constant.

2. A *pitch* is a counterclockwise rotation of β about the y-axis. The rotation matrix is given by

$$R_y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix}. \tag{3.40}$$

3. A *roll* is a counterclockwise rotation of γ about the x-axis. The rotation matrix is given by

$$R_x(\gamma) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{pmatrix}. \tag{3.41}$$

Each rotation matrix is a simple extension of the 2D rotation matrix, (3.31). For example, the yaw matrix, $R_z(\alpha)$, essentially performs a 2D rotation with respect to the x and y coordinates while leaving the z coordinate unchanged. Thus, the third row and third column of $R_z(\alpha)$ look like part of the identity matrix, while the upper right portion of $R_z(\alpha)$ looks like the 2D rotation matrix.

The yaw, pitch, and roll rotations can be used to place a 3D body in any orientation. A single rotation matrix can be formed by multiplying the yaw, pitch, and roll rotation matrices to obtain

$$R(\alpha,\beta,\gamma) = R_z(\alpha)\, R_y(\beta)\, R_x(\gamma) = \begin{pmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma \end{pmatrix}. \tag{3.42}$$

It is important to note that $R(\alpha, \beta, \gamma)$ performs the roll first, then the pitch, and finally the yaw. If the order of these operations is changed, a different rotation

matrix would result. Be careful when interpreting the rotations. Consider the final rotation, a yaw by α. Imagine sitting inside of a robot $\mathcal{A}$ that looks like an aircraft. If $\beta = \gamma = 0$, then the yaw turns the plane in a way that feels like turning a car to the left. However, for arbitrary values of β and γ, the final rotation axis will not be vertically aligned with the aircraft because the aircraft is left in an unusual orientation before α is applied. The yaw rotation occurs about the z-axis of the world frame, not the body frame of $\mathcal{A}$. Each time a new rotation matrix is introduced from the left, it has no concern for original body frame of $\mathcal{A}$. It simply rotates every point in $\mathbb{R}^3$ in terms of the world frame. Note that 3D rotations depend on three parameters, α, β, and γ, whereas 2D rotations depend only on a single parameter, θ. The primitives of the model can be transformed using $R(\alpha, \beta, \gamma)$, resulting in $\mathcal{A}(\alpha, \beta, \gamma)$.

Determining yaw, pitch, and roll from a rotation matrix It is often convenient to determine the α, β, and γ parameters directly from a given rotation matrix. Suppose an arbitrary rotation matrix

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \tag{3.43}$$

is given. By setting each entry equal to its corresponding entry in (3.42), equations are obtained that must be solved for α, β, and γ. Note that $r_{21}/r_{11} = \tan\alpha$ and $r_{32}/r_{33} = \tan\gamma$. Also, $r_{31} = -\sin\beta$ and $\sqrt{r_{32}^2 + r_{33}^2} = \cos\beta$. Solving for each angle yields

$$\alpha = \tan^{-1}(r_{21}/r_{11}), \tag{3.44}$$

$$\beta = \tan^{-1}\Big(-r_{31}/\sqrt{r_{32}^2 + r_{33}^2}\Big), \tag{3.45}$$

and

$$\gamma = \tan^{-1}(r_{32}/r_{33}). \tag{3.46}$$

There is a choice of four quadrants for the inverse tangent functions. How can the correct quadrant be determined? Each quadrant should be chosen by using the signs of the numerator and denominator of the argument. The numerator sign selects whether the direction will be above or below the x-axis, and the denominator selects whether the direction will be to the left or right of the y-axis. This is the same as the atan2 function in the C programming language, which nicely expands the range of the arctangent to $[0, 2\pi)$. This can be applied to express (3.44), (3.45), and (3.46) as

$$\alpha = \text{atan2}(r_{21}, r_{11}), \tag{3.47}$$

$$\beta = \text{atan2}\Big(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2}\Big), \tag{3.48}$$

and

$$\gamma = \text{atan2}(r_{32}, r_{33}). \tag{3.49}$$

Note that this method assumes $r_{11} \neq 0$ and $r_{33} \neq 0$.

The homogeneous transformation matrix for 3D bodies As in the 2D case, a homogeneous transformation matrix can be defined. For the 3D case, a 4×4 matrix is obtained that performs the rotation given by $R(\alpha, \beta, \gamma)$, followed by a translation given by x_t, y_t, z_t. The result is

$$T = \begin{pmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma & x_t \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma & y_t \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma & z_t \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.50}$$

Once again, the order of operations is critical. The matrix T in (3.50) represents the following sequence of transformations:

1. Roll by γ
2. Pitch by β
3. Yaw by α
4. Translate by (x_t, y_t, z_t).

The robot primitives can be transformed to yield $\mathcal{A}(x_t, y_t, z_t, \alpha, \beta, \gamma)$. A 3D rigid body that is capable of translation and rotation therefore has six degrees of freedom.

3.3 Transforming Kinematic Chains of Bodies

The transformations become more complicated for a chain of attached rigid bodies. For convenience, each rigid body is referred to as a *link*. Let $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m$ denote a set of m links. For each i such that $1 \leq i < m$, link $\mathcal{A}_i$ is "attached" to link $\mathcal{A}_{i+1}$ in a way that allows $\mathcal{A}_{i+1}$ some constrained motion with respect to $\mathcal{A}_i$. The motion constraint must be explicitly given, and will be discussed shortly. As an example, imagine a trailer that is attached to the back of a car by a hitch that allows the trailer to rotate with respect to the car. In general, a set of attached bodies will be referred to as a *linkage*. This section considers bodies that are attached in a single chain. This leads to a particular linkage called a *kinematic chain*.

3.3.1 A 2D Kinematic Chain

Before considering a kinematic chain, suppose $\mathcal{A}_1$ and $\mathcal{A}_2$ are unattached rigid bodies, each of which is capable of translating and rotating in $\mathcal{W} = \mathbb{R}^2$. Since each body has three degrees of freedom, there is a combined total of six degrees of freedom; the independent parameters are x_1, y_1, θ_1, x_2, y_2, and θ_2.

Attaching bodies When bodies are attached in a kinematic chain, degrees of freedom are removed. Figure 3.9 shows two different ways in which a pair of 2D links can be attached. The place at which the links are attached is called a *joint*. For a *revolute joint*, one link is capable only of rotation with respect to the other. For a *prismatic joint* is shown, one link slides along the other. Each type of joint removes two degrees of freedom from the pair of bodies. For example, consider a

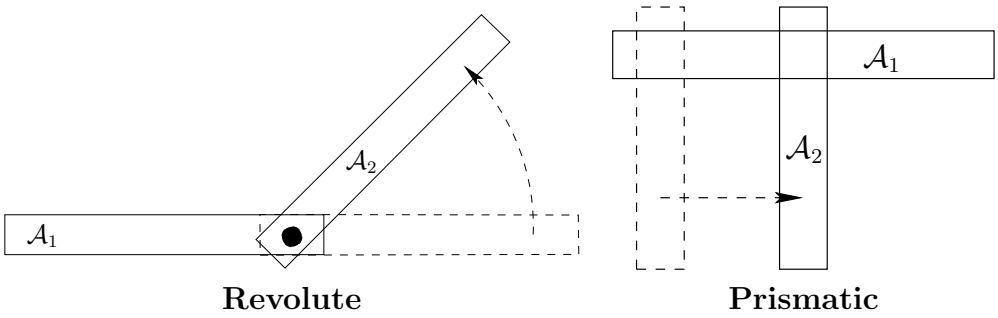


Figure 3.9: Two types of 2D joints: a revolute joint allows one link to rotate with respect to the other, and a prismatic joint allows one link to translate with respect to the other.

revolute joint that connects $\mathcal{A}_1$ to $\mathcal{A}_2$. Assume that the point $(0, 0)$ in the body frame of $\mathcal{A}_2$ is permanently fixed to a point (x_a, y_a) in the body frame of $\mathcal{A}_1$. This implies that the translation of $\mathcal{A}_2$ is completely determined once x_a and y_a are given. Note that x_a and y_a depend on x_1, y_1, and θ_1. This implies that $\mathcal{A}_1$ and $\mathcal{A}_2$ have a total of four degrees of freedom when attached. The independent parameters are x_1, y_1, θ_1, and θ_2. The task in the remainder of this section is to determine exactly how the models of $\mathcal{A}_1$, $\mathcal{A}_2$, ..., $\mathcal{A}_m$ are transformed when they are attached in a chain, and to give the expressions in terms of the independent parameters.

Consider the case of a kinematic chain in which each pair of links is attached by a revolute joint. The first task is to specify the geometric model for each link, $\mathcal{A}_i$. Recall that for a single rigid body, the origin of the body frame determines the axis of rotation. When defining the model for a link in a kinematic chain, excessive complications can be avoided by carefully placing the body frame. Since rotation occurs about a revolute joint, a natural choice for the origin is the joint between $\mathcal{A}_i$ and $\mathcal{A}_{i-1}$ for each $i > 1$. For convenience that will soon become evident, the x_i-axis for the body frame of $\mathcal{A}_i$ is defined as the line through the two joints that lie in $\mathcal{A}_i$, as shown in Figure 3.10. For the last link, $\mathcal{A}_m$, the x_m-axis can be placed arbitrarily, assuming that the origin is placed at the joint that connects $\mathcal{A}_m$ to $\mathcal{A}_{m-1}$. The body frame for the first link, $\mathcal{A}_1$, can be placed using the same considerations as for a single rigid body.

Homogeneous transformation matrices for 2D chains We are now prepared to determine the location of each link. The location in $\mathcal{W}$ of a point in $(x, y) \in \mathcal{A}_1$ is determined by applying the 2D homogeneous transformation matrix (3.35),

$$T_1 = \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & x_t \\ \sin\theta_1 & \cos\theta_1 & y_t \\ 0 & 0 & 1 \end{pmatrix}. \tag{3.51}$$

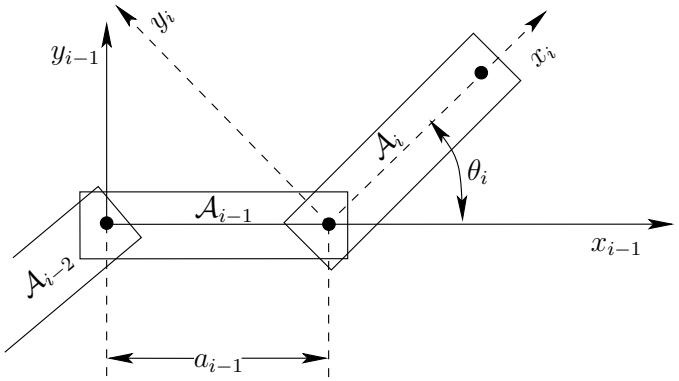


Figure 3.10: The body frame of each $\mathcal{A}_i$, for $1 < i < m$, is based on the joints that connect $\mathcal{A}_i$ to $\mathcal{A}_{i-1}$ and $\mathcal{A}_{i+1}$.

As shown in Figure 3.10, let a_{i-1} be the distance between the joints in $\mathcal{A}_{i-1}$. The orientation difference between $\mathcal{A}_i$ and $\mathcal{A}_{i-1}$ is denoted by the angle θ_i. Let T_i represent a 3×3 homogeneous transformation matrix (3.35), specialized for link $\mathcal{A}_i$ for $1 < i \leq m$,

$$T_i = \begin{pmatrix} \cos\theta_i & -\sin\theta_i & a_{i-1} \\ \sin\theta_i & \cos\theta_i & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{3.52}$$

This generates the following sequence of transformations:

1. Rotate counterclockwise by θ_i.
2. Translate by a_{i-1} along the x-axis.

The transformation T_i expresses the difference between the body frame of $\mathcal{A}_i$ and the body frame of $\mathcal{A}_{i-1}$. The application of T_i moves $\mathcal{A}_i$ from its body frame to the body frame of $\mathcal{A}_{i-1}$. The application of $T_{i-1}T_i$ moves both $\mathcal{A}_i$ and $\mathcal{A}_{i-1}$ to the body frame of $\mathcal{A}_{i-2}$. By following this procedure, the location in $\mathcal{W}$ of any point $(x, y) \in \mathcal{A}_m$ is determined by multiplying the transformation matrices to obtain

$$T_1 T_2 \cdots T_m \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \tag{3.53}$$

Example 3.3 (A 2D Chain of Three Links) To gain an intuitive understanding of these transformations, consider determining the configuration for link $\mathcal{A}_3$, as shown in Figure 3.11. Figure 3.11a shows a three-link chain in which $\mathcal{A}_1$ is at its initial configuration and the other links are each offset by $\pi/4$ from the previous link. Figure 3.11b shows the frame in which the model for $\mathcal{A}_3$ is initially defined. The application of T_3 causes a rotation of θ_3 and a translation by a_2. As shown in Figure 3.11c, this places $\mathcal{A}_3$ in its appropriate configuration. Note that $\mathcal{A}_2$ can be placed in its initial configuration, and it will be attached correctly to $\mathcal{A}_3$. The application of T_2 to the previous result places both $\mathcal{A}_3$ and $\mathcal{A}_2$

in their proper configurations, and $\mathcal{A}_1$ can be placed in its initial configuration. ■

For revolute joints, the a_i parameters are constants, and the θ_i parameters are variables. The transformed mth link is represented as $\mathcal{A}_m(x_t, y_t, \theta_1, \ldots, \theta_m)$. In some cases, the first link might have a fixed location in the world. In this case, the revolute joints account for all degrees of freedom, yielding $\mathcal{A}_m(\theta_1, \ldots, \theta_m)$. For prismatic joints, the a_i parameters are variables, instead of the θ_i parameters. It is straightforward to include both types of joints in the same kinematic chain.

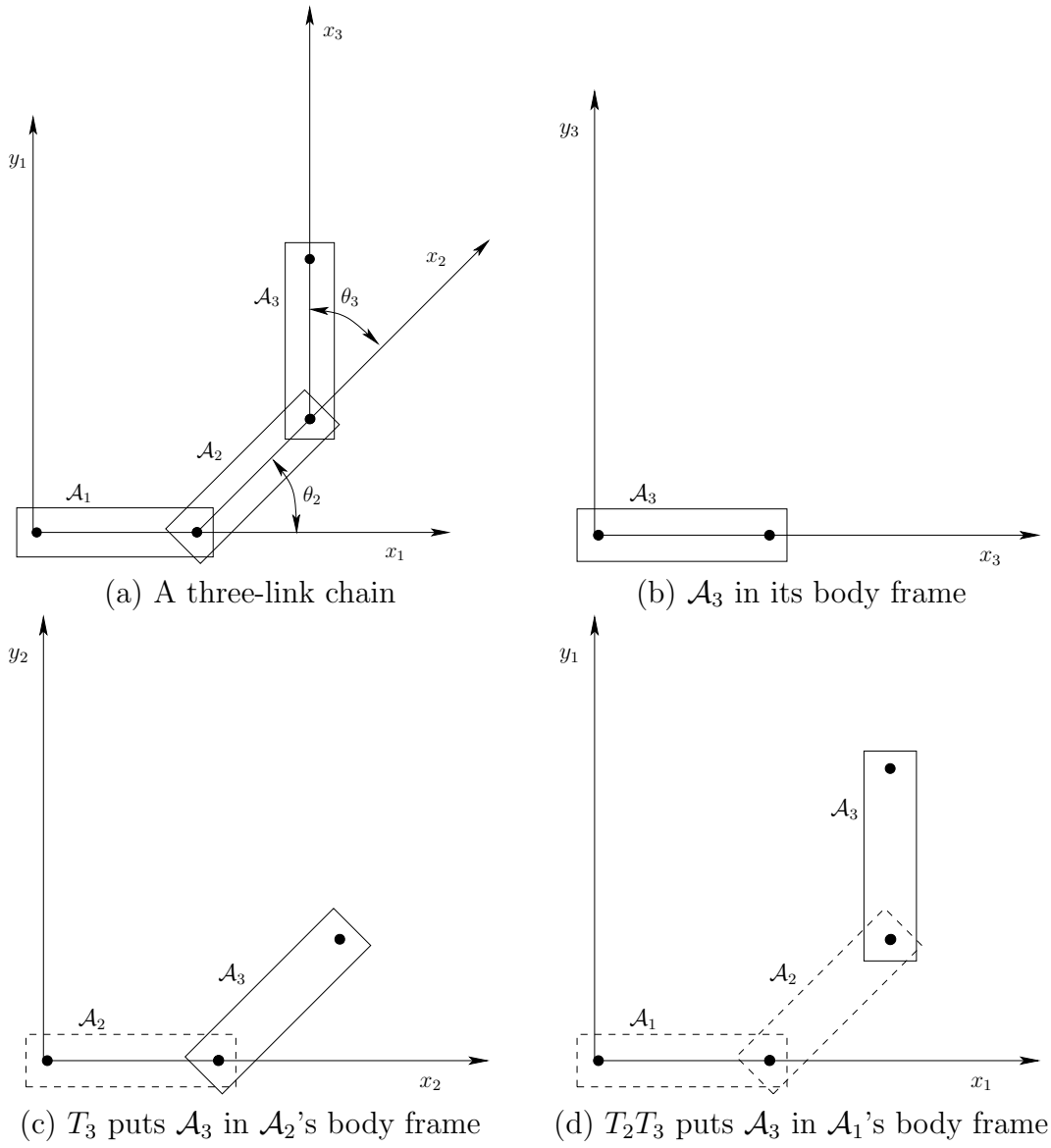
3.3.2 A 3D Kinematic Chain

As for a single rigid body, the 3D case is significantly more complicated than the 2D case due to 3D rotations. Also, several more types of joints are possible, as shown in Figure 3.12. Nevertheless, the main ideas from the transformations of 2D kinematic chains extend to the 3D case. The following steps from Section 3.3.1 will be recycled here:

1. The body frame must be carefully placed for each $\mathcal{A}_i$.
2. Based on joint relationships, several parameters are measured.
3. The parameters define a homogeneous transformation matrix, T_i.
4. The location in $\mathcal{W}$ of any point in $\mathcal{A}_m$ is given by applying the matrix $T_1 T_2 \cdots T_m$.

Consider a kinematic chain of m links in $\mathcal{W} = \mathbb{R}^3$, in which each $\mathcal{A}_i$ for $1 \leq i < m$ is attached to $\mathcal{A}_{i+1}$ by a revolute joint. Each link can be a complicated, rigid body as shown in Figure 3.13. For the 2D problem, the coordinate frames were based on the points of attachment. For the 3D problem, it is convenient to use the axis of rotation of each revolute joint (this is equivalent to the point of attachment for the 2D case). The axes of rotation will generally be skew lines in $\mathbb{R}^3$, as shown in Figure 3.14. Let the z_i-axis be the axis of rotation for the revolute joint that holds $\mathcal{A}_i$ to $\mathcal{A}_{i-1}$. Between each pair of axes in succession, let the x_i-axis join the closest pair of points between the z_i- and z_{i+1}-axes, with the origin on the z_i-axis and the direction pointing towards the nearest point of the z_{i+1}-axis. This axis is uniquely defined if the z_i- and z_{i+1}-axes are not parallel. The recommended body frame for each $\mathcal{A}_i$ will be given with respect to the z_i- and x_i-axes, which are shown in Figure 3.14. Assuming a right-handed coordinate system, the y_i-axis points away from us in Figure 3.14. In the transformations that will appear shortly, the coordinate frame given by x_i, y_i, and z_i will be most convenient for defining the model for $\mathcal{A}_i$. It might not always appear convenient because the origin of the frame may even lie outside of $\mathcal{A}_i$, but the resulting transformation matrices will be easy to understand.

In Section 3.3.1, each T_i was defined in terms of two parameters, a_{i-1} and θ_i. For the 3D case, four parameters will be defined: d_i, θ_i, a_{i-1}, and α_{i-1}. These



(a) A three-link chain

(b) $\mathcal{A}_3$ in its body frame

(c) T_3 puts $\mathcal{A}_3$ in $\mathcal{A}_2$'s body frame

(d) T_2T_3 puts $\mathcal{A}_3$ in $\mathcal{A}_1$'s body frame

Figure 3.11: Applying the transformation T_2T_3 to the model of $\mathcal{A}_3$. If T_1 is the identity matrix, then this yields the location in $\mathcal{W}$ of points in $\mathcal{A}_3$.

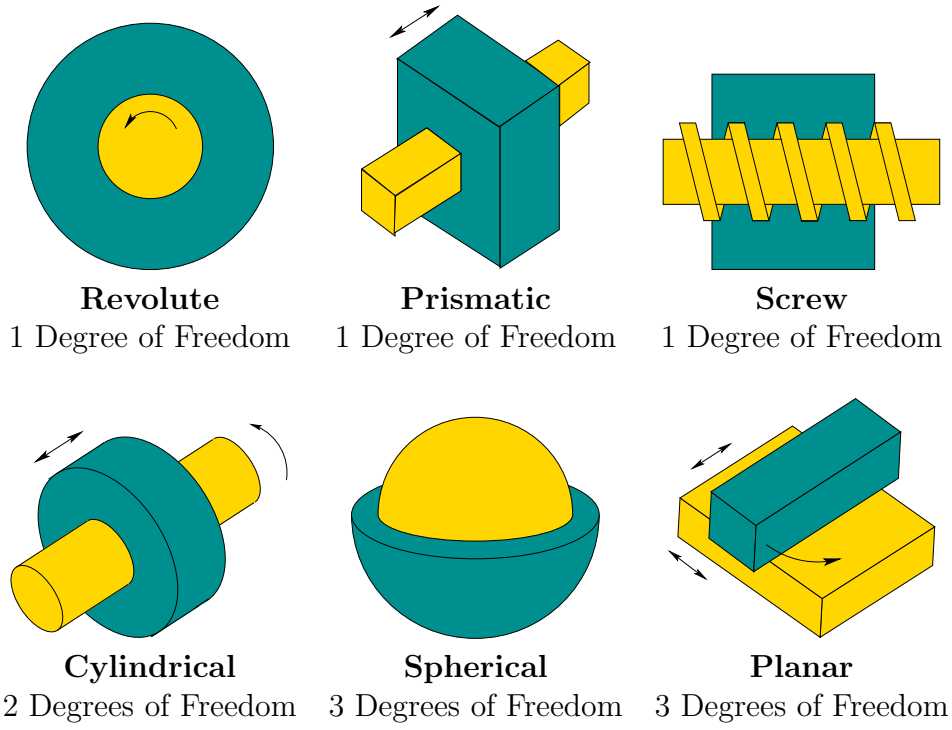


Figure 3.12: Types of 3D joints arising from the 2D surface contact between two bodies.

are referred to as *Denavit-Hartenberg (DH) parameters* [434]. The definition of each parameter is indicated in Figure 3.15. Figure 3.15a shows the definition of d_i. Note that the x_{i-1}- and x_i-axes contact the z_i-axis at two different places. Let d_i denote signed distance between these points of contact. If the x_i-axis is above the x_{i-1}-axis along the z_i-axis, then d_i is positive; otherwise, d_i is negative. The parameter θ_i is the angle between the x_i- and x_{i-1}-axes, which corresponds to the rotation about the z_i-axis that moves the x_{i-1}-axis to coincide with the x_i-axis. The parameter a_i is the distance between the z_i- and z_{i-1}-axes; recall these are generally skew lines in $\mathbb{R}^3$. The parameter α_{i-1} is the angle between the z_i- and z_{i-1}-axes.

Two screws The homogeneous transformation matrix T_i will be constructed by combining two simpler transformations. The transformation

$$R_i = \begin{pmatrix} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.54}$$

causes a rotation of θ_i about the z_i-axis, and a translation of d_i along the z_i-axis. Notice that the rotation by θ_i and translation by d_i commute because both

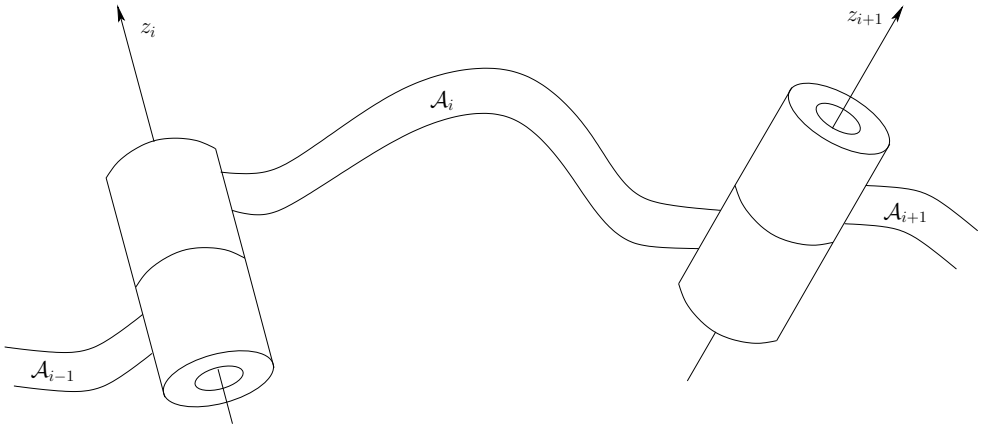


Figure 3.13: The rotation axes for a generic link attached by revolute joints.

operations occur with respect to the same axis, z_i. The combined operation of a translation and rotation with respect to the same axis is referred to as a *screw* (as in the motion of a screw through a nut). The effect of R_i can thus be considered as a screw about the z_i-axis. The second transformation is

$$Q_{i-1} = \begin{pmatrix} 1 & 0 & 0 & a_{i-1} \\ 0 & \cos\alpha_{i-1} & -\sin\alpha_{i-1} & 0 \\ 0 & \sin\alpha_{i-1} & \cos\alpha_{i-1} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.55}$$

which can be considered as a screw about the x_{i-1}-axis. A rotation of α_{i-1} about the x_{i-1}-axis and a translation of a_{i-1} are performed.

The homogeneous transformation matrix The transformation T_i, for each i such that $1 < i \leq m$, is

$$T_i = Q_{i-1}R_i = \begin{pmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_{i-1} \\ \sin\theta_i\cos\alpha_{i-1} & \cos\theta_i\cos\alpha_{i-1} & -\sin\alpha_{i-1} & -\sin\alpha_{i-1}d_i \\ \sin\theta_i\sin\alpha_{i-1} & \cos\theta_i\sin\alpha_{i-1} & \cos\alpha_{i-1} & \cos\alpha_{i-1}d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.56}$$

This can be considered as the 3D counterpart to the 2D transformation matrix, (3.52). The following four operations are performed in succession:

1. Translate by d_i along the z_i-axis.
2. Rotate counterclockwise by θ_i about the z_i-axis.
3. Translate by a_{i-1} along the x_{i-1}-axis.
4. Rotate counterclockwise by α_{i-1} about the x_{i-1}-axis.

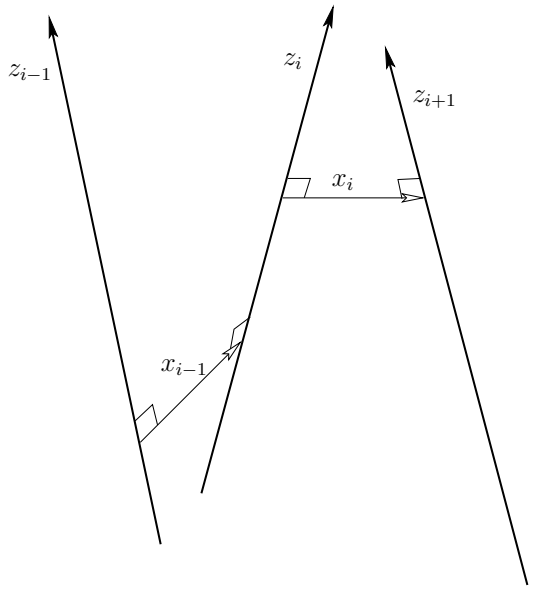


Figure 3.14: The rotation axes of the generic links are skew lines in $\mathbb{R}^3$.

As in the 2D case, the first matrix, T_1, is special. To represent any position and orientation of $\mathcal{A}_1$, it could be defined as a general rigid-body homogeneous transformation matrix, (3.50). If the first body is only capable of rotation via a revolute joint, then a simple convention is usually followed. Let the a_0, α_0 parameters of T_1 be assigned as $a_0 = \alpha_0 = 0$ (there is no z_0-axis). This implies that Q_0 from (3.55) is the identity matrix, which makes $T_1 = R_1$.

The transformation T_i for $i > 1$ gives the relationship between the body frame of $\mathcal{A}_i$ and the body frame of $\mathcal{A}_{i-1}$. The position of a point (x, y, z) on $\mathcal{A}_m$ is given by

$$T_1 T_2 \cdots T_m \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}. \tag{3.57}$$

For each revolute joint, θ_i is treated as the only variable in T_i. Prismatic joints can be modeled by allowing a_i to vary. More complicated joints can be modeled as a sequence of degenerate joints. For example, a spherical joint can be considered as a sequence of three zero-length revolute joints; the joints perform a roll, a pitch, and a yaw. Another option for more complicated joints is to abandon the DH representation and directly develop the homogeneous transformation matrix. This might be needed to preserve topological properties that become important in Chapter 4.

Example 3.4 (Puma 560) This example demonstrates the 3D chain kinematics on a classic robot manipulator, the PUMA 560, shown in Figure 3.16. The current parameterization here is based on [37, 555]. The procedure is to determine

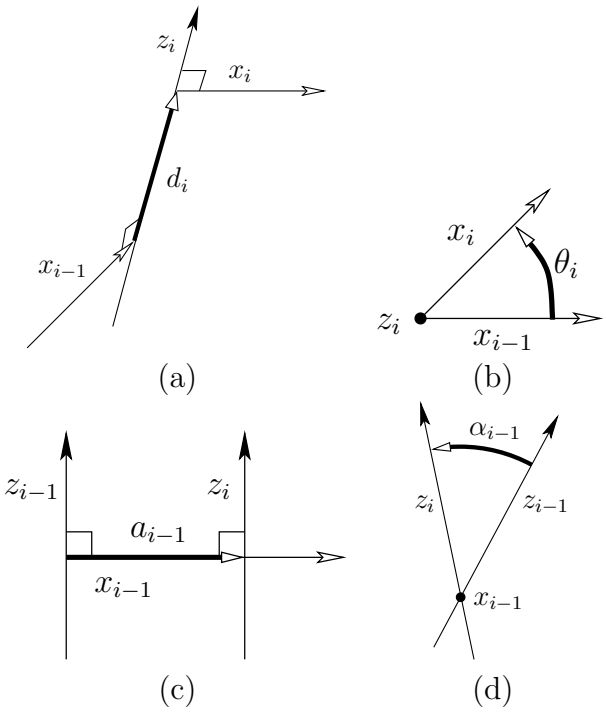


Figure 3.15: Definitions of the four DH parameters: d_i, θ_i, a_{i-1}, α_{i-1}. The z_i- and x_{i-1}-axes in (b) and (d), respectively, are pointing outward. Any parameter may be positive, zero, or negative.

appropriate body frames to represent each of the links. The first three links allow the hand (called an end-effector) to make large movements in $\mathcal{W}$, and the last three enable the hand to achieve a desired orientation. There are six degrees of freedom, each of which arises from a revolute joint. The body frames are shown in Figure 3.16, and the corresponding DH parameters are given in Figure 3.17. Each transformation matrix T_i is a function of θ_i; hence, it is written $T_i(\theta_i)$. The other parameters are fixed for this example. Only θ_1, θ_2, ..., θ_6 are allowed to vary.

The parameters from Figure 3.17 may be substituted into the homogeneous transformation matrices to obtain

$$T_1(\theta_1) = \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.58}$$

$$T_2(\theta_2) = \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & 0 \\ 0 & 0 & 1 & d_2 \\ -\sin\theta_2 & -\cos\theta_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.59}$$

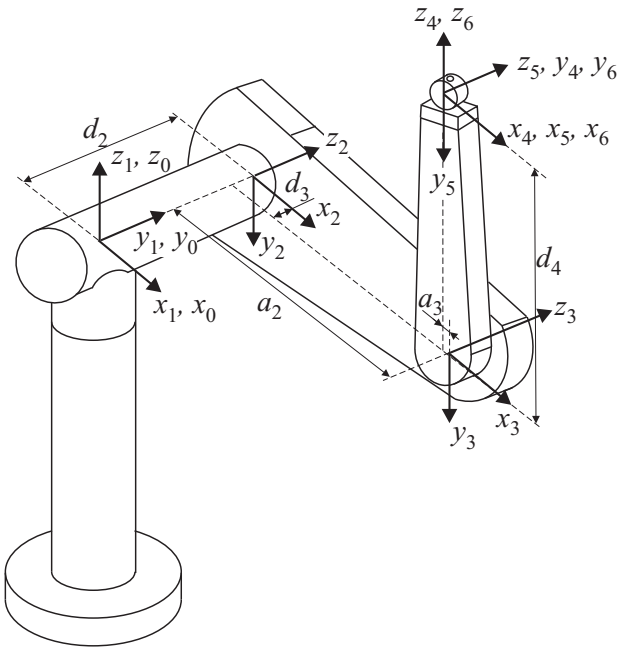


Figure 3.16: The Puma 560 is shown along with the DH parameters and body frames for each link in the chain. This figure is borrowed from [555] by courtesy of the authors.

$$T_3(\theta_3) = \begin{pmatrix} \cos\theta_3 & -\sin\theta_3 & 0 & a_2 \\ \sin\theta_3 & \cos\theta_3 & 0 & 0 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.60}$$

$$T_4(\theta_4) = \begin{pmatrix} \cos\theta_4 & -\sin\theta_4 & 0 & a_3 \\ 0 & 0 & -1 & -d_4 \\ \sin\theta_4 & \cos\theta_4 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.61}$$

$$T_5(\theta_5) = \begin{pmatrix} \cos\theta_5 & -\sin\theta_5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin\theta_5 & -\cos\theta_5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.62}$$

and

$$T_6(\theta_6) = \begin{pmatrix} \cos\theta_6 & -\sin\theta_6 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \sin\theta_6 & \cos\theta_6 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.63}$$

Matrix	α_{i-1}	a_{i-1}	θ_i	d_i
$T_1(\theta_1)$	0	0	θ_1	0
$T_2(\theta_2)$	$-\pi/2$	0	θ_2	d_2
$T_3(\theta_3)$	0	a_2	θ_3	d_3
$T_4(\theta_4)$	$\pi/2$	a_3	θ_4	d_4
$T_5(\theta_5)$	$-\pi/2$	0	θ_5	0
$T_6(\theta_6)$	$\pi/2$	0	θ_6	0

Figure 3.17: The DH parameters are shown for substitution into each homogeneous transformation matrix (3.56). Note that a_3 and d_3 are negative in this example (they are signed displacements, not distances).

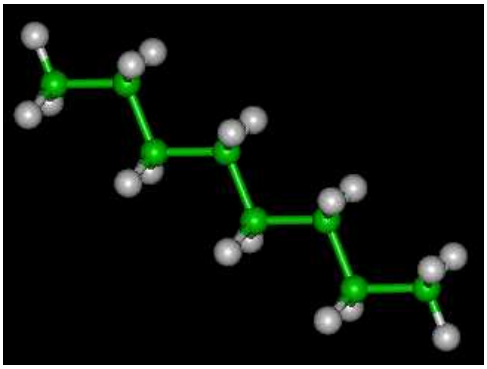


Figure 3.18: A hydrocarbon (octane) molecule with 8 carbon atoms and 18 hydrogen atoms (courtesy of the New York University MathMol Library).

A point (x, y, z) in the body frame of the last link $\mathcal{A}_6$ appears in $\mathcal{W}$ as

$$T_1(\theta_1)T_2(\theta_2)T_3(\theta_3)T_4(\theta_4)T_5(\theta_5)T_6(\theta_6) \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}. \tag{3.64}$$

■

Example 3.5 (Transforming Octane) Figure 3.18 shows a ball-and-stick model of an octane molecule. Each "ball" is an atom, and each "stick" represents a bond between a pair of atoms. There is a linear chain of eight carbon atoms, and a bond exists between each consecutive pair of carbons in the chain. There are also numerous hydrogen atoms, but we will ignore them. Each bond between a pair of carbons is capable of twisting, as shown in Figure 3.19. Studying the configurations (called *conformations*) of molecules is an important part of computational

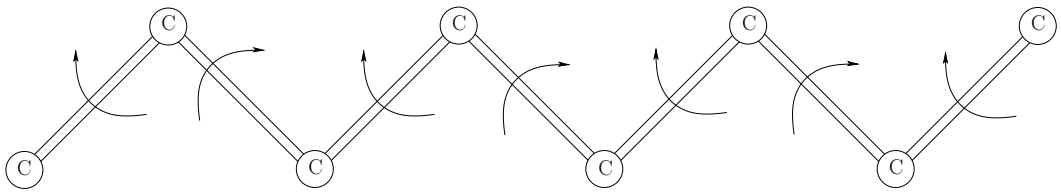


Figure 3.19: Consider transforming the spine of octane by ignoring the hydrogen atoms and allowing the bonds between carbons to rotate. This can be easily constructed with balls and sticks (e.g., Tinkertoys). If the first link is held fixed, then there are six degrees of freedom. The rotation of the last link is ignored.

biology. It is assumed that there are seven degrees of freedom, each of which arises from twisting a bond. The techniques from this section can be applied to represent these transformations.

Note that the bonds correspond exactly to the axes of rotation. This suggests that the z_i axes should be chosen to coincide with the bonds. Since consecutive bonds meet at atoms, there is no distance between them. From Figure 3.15c, observe that this makes $a_i = 0$ for all i. From Figure 3.15a, it can be seen that each d_i corresponds to a *bond length*, the distance between consecutive carbon atoms. See Figure 3.20. This leaves two angular parameters, θ_i and α_i. Since the only possible motion of the links is via rotation of the z_i-axes, the angle between two consecutive axes, as shown in Figure 3.15d, must remain constant. In chemistry, this is referred to as the *bond angle* and is represented in the DH parameterization as α_i. The remaining θ_i parameters are the variables that represent the degrees of freedom. However, looking at Figure 3.15b, observe that the example is degenerate because each x_i-axis has no frame of reference because each $a_i = 0$. This does not, however, cause any problems. For visualization purposes, it may be helpful to replace x_{i-1} and x_i by z_{i-1} and z_{i+1}, respectively. This way it is easy to see that as the bond for the z_i-axis is twisted, the observed angle changes accordingly. Each bond is interpreted as a link, $\mathcal{A}_i$. The origin of each $\mathcal{A}_i$ must be chosen to coincide with the intersection point of the z_i- and z_{i+1}-axes. Thus, most of the points in $\mathcal{A}_i$ will lie in the $-z_i$ direction; see Figure 3.20.

The next task is to write down the matrices. Attach a world frame to the first bond, with the second atom at the origin and the bond aligned with the z-axis, in the negative direction; see Figure 3.20. To define T_1, recall that $T_1 = R_1$ from (3.54) because Q_0 is dropped. The parameter d_1 represents the distance between the intersection points of the x_0- and x_1-axes along the z_1 axis. Since there is no x_0-axis, there is freedom to choose d_1; hence, let $d_1 = 0$ to obtain

$$T_1(\theta_1) = R_1(\theta_1) = \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.65}$$

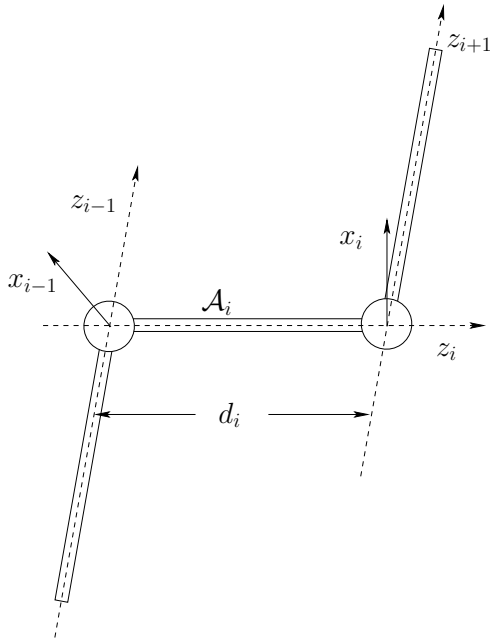


Figure 3.20: Each bond may be interpreted as a "link" of length d_i that is aligned with the z_i-axis. Note that most of $\mathcal{A}_i$ appears in the $-z_i$ direction.

The application of T_1 to points in $\mathcal{A}_1$ causes them to rotate around the z_1-axis, which appears correct.

The matrices for the remaining six bonds are

$$T_i(\theta_i) = \begin{pmatrix} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i \cos\alpha_{i-1} & \cos\theta_i \cos\alpha_{i-1} & -\sin\alpha_{i-1} & -\sin\alpha_{i-1} d_i \\ \sin\theta_i \sin\alpha_{i-1} & \cos\theta_i \sin\alpha_{i-1} & \cos\alpha_{i-1} & \cos\alpha_{i-1} d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.66}$$

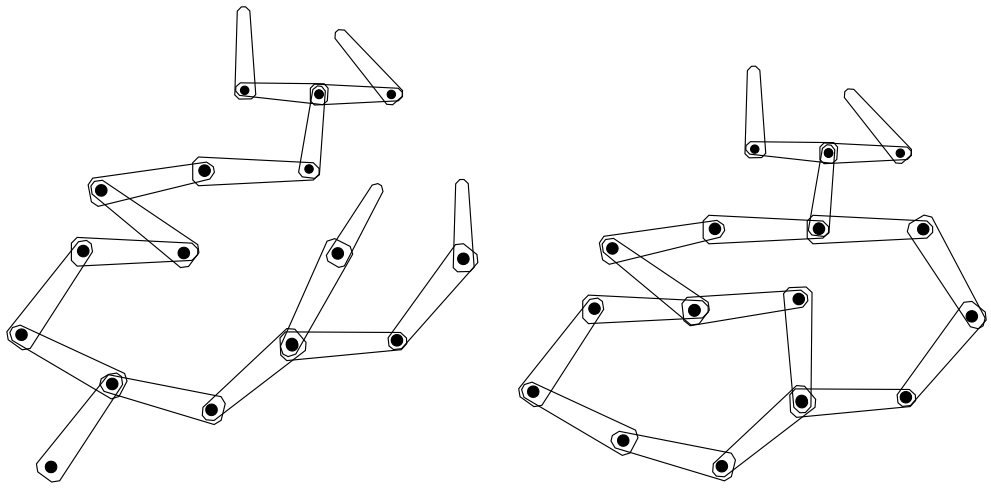
for $i \in \{2, \ldots, 7\}$. The position of any point, $(x, y, z) \in \mathcal{A}_7$, is given by

$$T_1(\theta_1) T_2(\theta_2) T_3(\theta_3) T_4(\theta_4) T_5(\theta_5) T_6(\theta_6) T_7(\theta_7) \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}. \tag{3.67}$$

■

3.4 Transforming Kinematic Trees

Motivation For many interesting problems, the linkage is arranged in a "tree" as shown in Figure 3.21a. Assume here that the links are not attached in ways



(a) (b)

Figure 3.21: General linkages: (a) Instead of a chain of rigid bodies, a "tree" of rigid bodies can be considered. (b) If there are loops, then parameters must be carefully assigned to ensure that the loops are closed.

that form loops (i.e., Figure 3.21b); that case is deferred until Section 4.4, although some comments are also made at the end of this section. The human body, with its joints and limbs attached to the torso, is an example that can be modeled as a tree of rigid links. Joints such as knees and elbows are considered as revolute joints. A shoulder joint is an example of a spherical joint, although it cannot achieve any orientation (without a visit to the emergency room!). As mentioned in Section 1.4, there is widespread interest in animating humans in virtual environments and also in developing humanoid robots. Both of these cases rely on formulations of kinematics that mimic the human body.

Another problem that involves kinematic trees is the conformational analysis of molecules. Example 3.5 involved a single chain; however, most organic molecules are more complicated, as in the familiar drugs shown in Figure 1.14a (Section 1.2). The bonds may twist to give degrees of freedom to the molecule. Moving through the space of conformations requires the formulation of a kinematic tree. Studying these conformations is important because scientists need to determine for some candidate drug whether the molecule can twist the right way so that it docks nicely (i.e., requires low energy) with a protein cavity; this induces a pharmacological effect, which hopefully is the desired one. Another important problem is determining how complicated protein molecules fold into certain configurations. These molecules are orders of magnitude larger (in terms of numbers of atoms and degrees of freedom) than typical drug molecules. For more information, see Section 7.5.

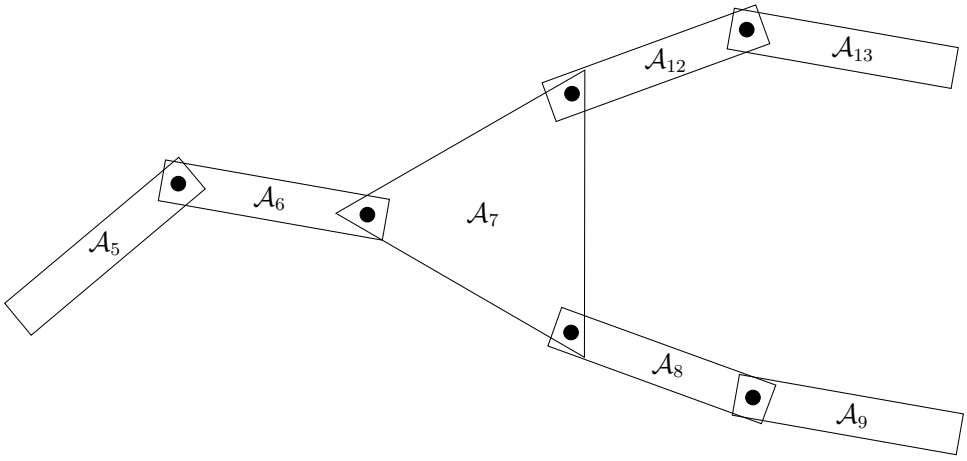


Figure 3.22: Now it is possible for a link to have more than two joints, as in $\mathcal{A}_7$.

Common joints for $\mathcal{W} = \mathbb{R}^2$ First consider the simplest case in which there is a 2D tree of links for which every link has only two points at which revolute joints may be attached. This corresponds to Figure 3.21a. A single link is designated as the *root*, $\mathcal{A}_1$, of the tree. To determine the transformation of a body, $\mathcal{A}_i$, in the tree, the tools from Section 3.3.1 are directly applied to the chain of bodies that connects $\mathcal{A}_i$ to $\mathcal{A}_1$ while ignoring all other bodies. Each link contributes a θ_i to the total degrees of freedom of the tree. This case seems quite straightforward; unfortunately, it is not this easy in general.

Junctions with more than two rotation axes Now consider modeling a more complicated collection of attached links. The main novelty is that one link may have joints attached to it in more than two locations, as in $\mathcal{A}_7$ in Figure 3.22. A link with more than two joints will be referred to as a *junction*.

If there is only one junction, then most of the complications arising from junctions can be avoided by choosing the junction as the root. For example, for a simple humanoid model, the torso would be a junction. It would be sensible to make this the root of the tree, as opposed to the right foot. The legs, arms, and head could all be modeled as independent chains. In each chain, the only concern is that the first link of each chain does not attach to the same point on the torso. This can be solved by inserting a fixed, fictitious link that connects from the origin of the torso to the attachment point of the limb.

The situation is more interesting if there are multiple junctions. Suppose that Figure 3.22 represents part of a 2D system of links for which the root, $\mathcal{A}_1$, is attached via a chain of links to $\mathcal{A}_5$. To transform link $\mathcal{A}_9$, the tools from Section

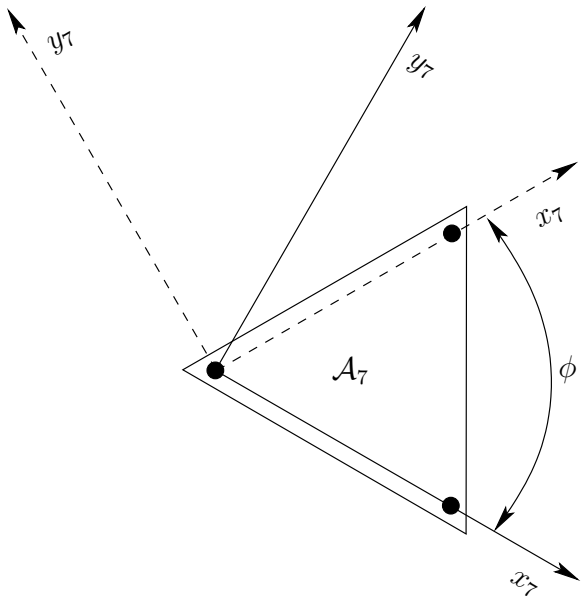


Figure 3.23: The junction is assigned two different frames, depending on which chain was followed. The solid axes were obtained from transforming $\mathcal{A}_9$, and the dashed axes were obtained from transforming $\mathcal{A}_{13}$.

3.3.1 may be directly applied to yield a sequence of transformations,

$$T_1 \cdots T_5 T_6 T_7 T_8 T_9 \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \tag{3.68}$$

for a point $(x, y) \in \mathcal{A}_9$. Likewise, to transform T_{13}, the sequence

$$T_1 \cdots T_5 T_6 T_7 T_{12} T_{13} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{3.69}$$

can be used by ignoring the chain formed by $\mathcal{A}_8$ and $\mathcal{A}_9$. So far everything seems to work well, but take a close look at $\mathcal{A}_7$. As shown in Figure 3.23, its body frame was defined in two different ways, one for each chain. If both are forced to use the same frame, then at least one must abandon the nice conventions of Section 3.3.1 for choosing frames. This situation becomes worse for 3D trees because this would suggest abandoning the DH parameterization. The Khalil-Kleinfinger parameterization is an elegant extension of the DH parameterization and solves these frame assignment issues [524].

Constraining parameters Fortunately, it is fine to use different frames when following different chains; however, one extra piece of information is needed. Imagine transforming the whole tree. The variable θ_7 will appear twice, once from each

of the upper and lower chains. Let θ_{7u} and θ_{7l} denote these θ's. Can θ really be chosen two different ways? This would imply that the tree is instead as pictured in Figure 3.24, in which there are two independently moving links, $\mathcal{A}_{7u}$ and $\mathcal{A}_{7l}$. To fix this problem, a constraint must be imposed. Suppose that θ_{7l} is treated as

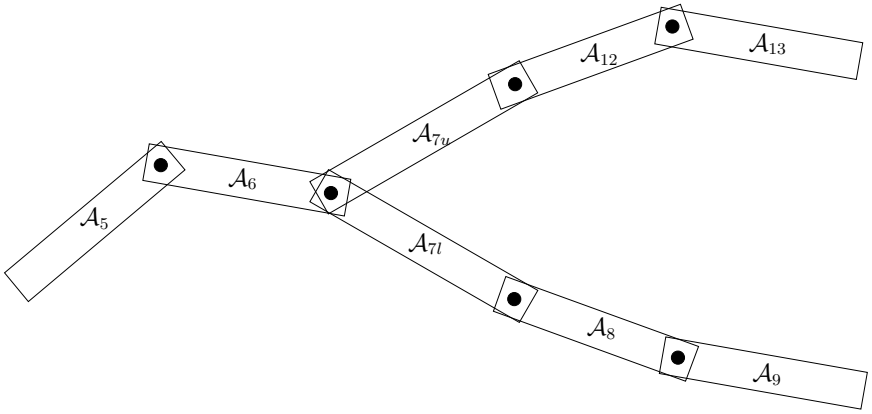


Figure 3.24: Choosing each θ_7 independently would result in a tree that ignores that fact that $\mathcal{A}_7$ is rigid.

an independent variable. The parameter θ_{7u} must then be chosen as $\theta_{7l} + \phi$, in which ϕ is as shown in Figure 3.23.

Example 3.6 (A 2D Tree of Bodies) Figure 3.25 shows a 2D example that involves six links. To transform $(x, y) \in \mathcal{A}_6$, the only relevant links are $\mathcal{A}_5$, $\mathcal{A}_2$, and $\mathcal{A}_1$. The chain of transformations is

$$T_1 T_{2l} T_5 T_6 \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \tag{3.70}$$

in which

$$T_1 = \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & x_t \\ \sin\theta_1 & \cos\theta_1 & y_t \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{3.71}$$

$$T_{2l} = \begin{pmatrix} \cos\theta_{2l} & -\sin\theta_{2l} & a_1 \\ \sin\theta_{2l} & \cos\theta_{2l} & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 1 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{3.72}$$

$$T_5 = \begin{pmatrix} \cos\theta_5 & -\sin\theta_5 & a_2 \\ \sin\theta_5 & \cos\theta_5 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\theta_5 & -\sin\theta_5 & \sqrt{2} \\ \sin\theta_5 & \cos\theta_5 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{3.73}$$

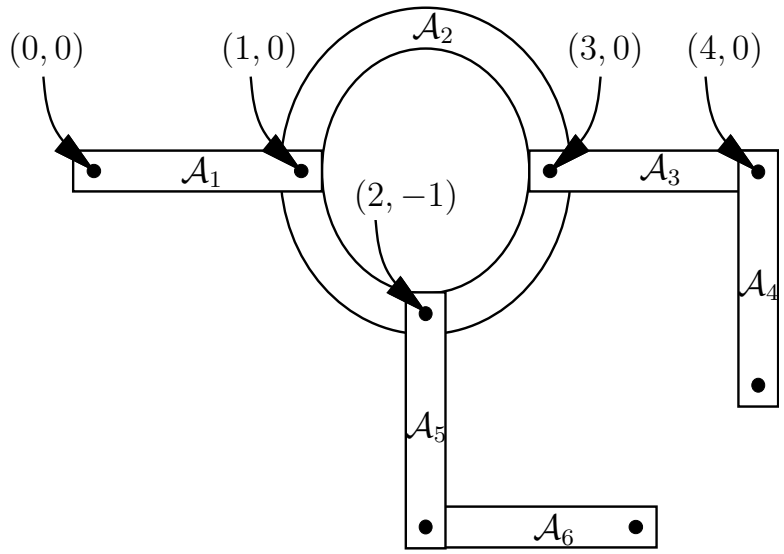


Figure 3.25: A tree of bodies in which the joints are attached in different places.

and

$$T_6 = \begin{pmatrix} \cos\theta_6 & -\sin\theta_6 & a_5 \\ \sin\theta_6 & \cos\theta_6 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\theta_6 & -\sin\theta_6 & 1 \\ \sin\theta_6 & \cos\theta_6 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{3.74}$$

The matrix T_{2l} in (3.72) denotes the fact that the lower chain was followed. The transformation for points in $\mathcal{A}_4$ is

$$T_1 T_{2u} T_4 T_5 \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \tag{3.75}$$

in which T_1 is the same as in (3.71), and

$$T_3 = \begin{pmatrix} \cos\theta_3 & -\sin\theta_3 & a_2 \\ \sin\theta_3 & \cos\theta_3 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\theta_3 & -\sin\theta_3 & \sqrt{2} \\ \sin\theta_3 & \cos\theta_3 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{3.76}$$

and

$$T_4 = \begin{pmatrix} \cos\theta_4 & -\sin\theta_4 & a_4 \\ \sin\theta_4 & \cos\theta_4 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\theta_4 & -\sin\theta_4 & 0 \\ \sin\theta_4 & \cos\theta_4 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{3.77}$$

The interesting case is

$$T_{2u} = \begin{pmatrix} \cos\theta_{2u} & -\sin\theta_{2u} & a_1 \\ \sin\theta_{2u} & \cos\theta_{2u} & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta_{2l} + \pi/4) & -\sin(\theta_{2l} + \pi/4) & a_1 \\ \sin(\theta_{2l} + \pi/4) & \cos(\theta_{2l} + \pi/4) & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{3.78}$$

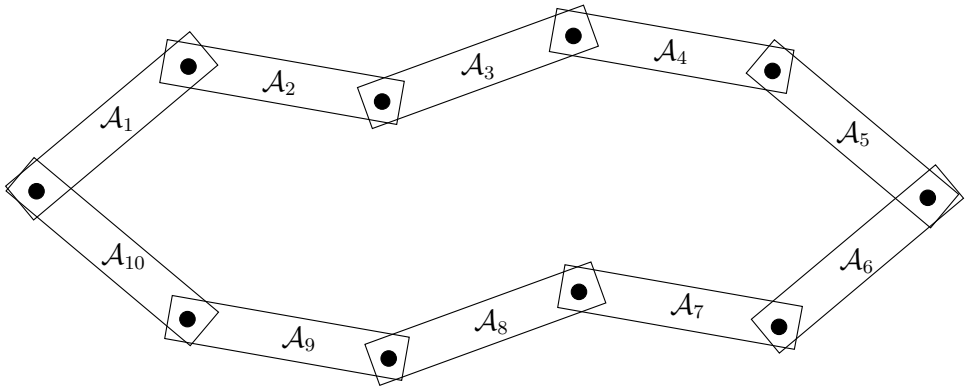


Figure 3.26: There are ten links and ten revolute joints arranged in a loop. This is an example of a closed kinematic chain.

in which the constraint $\theta_{2u} = \theta_{2l} + \pi/4$ is imposed to enforce the fact that $\mathcal{A}_2$ is a junction. ■

For a 3D tree of bodies the same general principles may be followed. In some cases, there will not be any complications that involve special considerations of junctions and constraints. One example of this is the transformation of flexible molecules because all consecutive rotation axes intersect, and junctions occur directly at these points of intersection. In general, however, the DH parameter technique may be applied for each chain, and then the appropriate constraints have to be determined and applied to represent the true degrees of freedom of the tree. The Khalil-Kleinfinger parameterization conveniently captures the resulting solution [524].

What if there are loops? The most general case includes links that are connected in loops, as shown in Figure 3.26. These are generally referred to as *closed kinematic chains*. This arises in many applications. For example, with humanoid robotics or digital actors, a loop is formed when both feet touch the ground. As another example, suppose that two robot manipulators, such as the Puma 560 from Example 3.4, cooperate together to carry an object. If each robot grasps the same object with its hand, then a loop will be formed. A complicated example of this was shown in Figure 1.5, in which mobile robots moved a piano. Outside of robotics, a large fraction of organic molecules have flexible loops. Exploring the space of their conformations requires careful consideration of the difficulties imposed by these loops.

The main difficulty of working with closed kinematic chains is that it is hard to determine which parameter values are within an acceptable range to ensure closure. If these values are given, then the transformations are handled in the

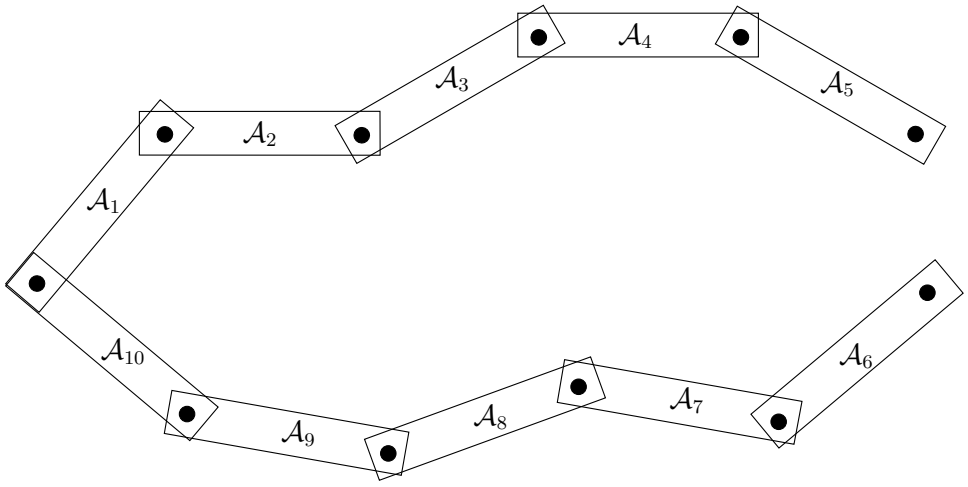


Figure 3.27: Loops may be opened to enable tree-based transformations to be applied; however, a closure constraint must still be satisfied.

same way as the case of trees. For example, the links in Figure 3.26 may be transformed by breaking the loop into two different chains. Suppose we forget that the joint between $\mathcal{A}_5$ and $\mathcal{A}_6$ exists, as shown in Figure 3.27. Consider two different kinematic chains that start at the joint on the extreme left. There is an upper chain from $\mathcal{A}_1$ to $\mathcal{A}_5$ and a lower chain from $\mathcal{A}_{10}$ to $\mathcal{A}_6$. The transformations for any of these bodies can be obtained directly from the techniques of Section 3.3.1. Thus, it is easy to transform the bodies, but how do we choose parameter values that ensure $\mathcal{A}_5$ and $\mathcal{A}_6$ are connected at their common joint? Using the upper chain, the position of this joint may be expressed as

$$T_1(\theta_1)T_2(\theta_2)T_3(\theta_3)T_4(\theta_4)T_5(\theta_5) \begin{pmatrix} a_5 \\ 0 \\ 1 \end{pmatrix}, \tag{3.79}$$

in which $(a_5, 0) \in \mathcal{A}_5$ is the location of the joint of $\mathcal{A}_5$ that is supposed to connect to $\mathcal{A}_6$. The position of this joint may also be expressed using the lower chain as

$$T_{10}(\theta_{10})T_9(\theta_9)T_8(\theta_8)T_7(\theta_7)T_6(\theta_6) \begin{pmatrix} a_6 \\ 0 \\ 1 \end{pmatrix}, \tag{3.80}$$

with $(a_6, 0)$ representing the position of the joint in the body frame of $\mathcal{A}_6$. If the loop does not have to be maintained, then any values for $\theta_1, \ldots, \theta_{10}$ may be selected, resulting in ten degrees of freedom. However, if a loop must maintained,

then (3.79) and (3.80) must be equal,

$$T_1(\theta_1)T_2(\theta_2)T_3(\theta_3)T_4(\theta_4)T_5(\theta_5)\begin{pmatrix} a_5 \\ 0 \\ 1 \end{pmatrix} = T_{10}(\theta_{10})T_9(\theta_9)T_8(\theta_8)T_7(\theta_7)T_6(\theta_6)\begin{pmatrix} a_6 \\ 0 \\ 1 \end{pmatrix}, \tag{3.81}$$

which is quite a mess of nonlinear, trigonometric equations that must be solved. The set of solutions to (3.81) could be very complicated. For the example, the true degrees of freedom is eight because two were removed by making the joint common. Since the common joint allows the links to rotate, exactly two degrees of freedom are lost. If $\mathcal{A}_5$ and $\mathcal{A}_6$ had to be rigidly attached, then the total degrees of freedom would be only seven. For most problems that involve loops, it will not be possible to obtain a nice parameterization of the set of solutions. This a form of the well-known *inverse kinematics problem* [252, 693, 775, 994].

In general, a complicated arrangement of links can be imagined in which there are many loops. Each time a joint along a loop is "ignored," as in the procedure just described, then one less loop exists. This process can be repeated iteratively until there are no more loops in the graph. The resulting arrangement of links will be a tree for which the previous techniques of this section may be applied. However, for each joint that was "ignored" an equation similar to (3.81) must be introduced. All of these equations must be satisfied simultaneously to respect the original loop constraints. Suppose that a set of value parameters is already given. This could happen, for example, using motion capture technology to measure the position and orientation of every part of a human body in contact with the ground. From this the solution parameters could be computed, and all of the transformations are easy to represent. However, as soon as the model moves, it is difficult to ensure that the new transformations respect the closure constraints. The foot of the digital actor may push through the floor, for example. Further information on this problem appears in Section 4.4.

3.5 Nonrigid Transformations

One can easily imagine motion planning for nonrigid bodies. This falls outside of the families of transformations studied so far in this chapter. Several kinds of nonrigid transformations are briefly surveyed here.

Linear transformations A rotation is a special case of a linear transformation, which is generally expressed by an $n \times n$ matrix, M, assuming the transformations are performed over $\mathbb{R}^n$. Consider transforming a point (x, y) in a 2D robot, $\mathcal{A}$, as

$$\begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}. \tag{3.82}$$

If M is a rotation matrix, then the size and shape of $\mathcal{A}$ will remain the same. In some applications, however, it may be desirable to distort these. The robot can

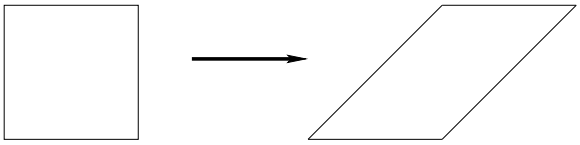


Figure 3.28: Shearing transformations may be performed.

be *scaled* by m_{11} along the x-axis and m_{22} along the y-axis by applying

$$\begin{pmatrix} m_{11} & 0 \\ 0 & m_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}, \tag{3.83}$$

for positive real values m_{11} and m_{22}. If one of them is negated, then a mirror image of $\mathcal{A}$ is obtained. In addition to scaling, $\mathcal{A}$ can be *sheared* by applying

$$\begin{pmatrix} 1 & m_{12} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{3.84}$$

for $m_{12} \neq 0$. The case of $m_{12} = 1$ is shown in Figure 3.28.

The scaling, shearing, and rotation matrices may be multiplied together to yield a general transformation matrix that explicitly parameterizes each effect. It is also possible to extend the M from $n \times n$ to $(n+1) \times (n+1)$ to obtain a homogeneous transformation matrix that includes translation. Also, the concepts extend in a straightforward way to $\mathbb{R}^3$ and beyond. This enables the additional effects of scaling and shearing to be incorporated directly into the concepts from Sections 3.2–3.4.

Flexible materials In some applications there is motivation to move beyond linear transformations. Imagine trying to warp a flexible material, such as a mattress, through a doorway. The mattress could be approximated by a 2D array of links; however, the complexity and degrees of freedom would be too cumbersome. For another example, suppose that a snake-like robot is designed by connecting 100 revolute joints together in a chain. The tools from Section 3.3 may be used to transform it with 100 rotation parameters, $\theta_1, \ldots, \theta_{100}$, but this may become unwieldy for use in a planning algorithm. An alternative is to approximate the snake with a deformable curve or shape.

For problems such as these, it is desirable to use a parameterized family of curves or surfaces. Spline models are often most appropriate because they are designed to provide easy control over the shape of a curve through the adjustment of a small number of parameters. Other possibilities include the generalized-cylinder and superquadric models that were mentioned in Section 3.1.3.

One complication is that complicated constraints may be imposed on the space of allowable parameters. For example, each joint of a snake-like robot could have a

small range of rotation. This would be easy to model using a kinematic chain; however, determining which splines from a spline family satisfy this extra constraint may be difficult. Likewise, for manipulating flexible materials, there are usually complicated constraints based on the elasticity of the material. Even determining its correct shape under the application of some forces requires integration of an elastic energy function over the material [577].

Further Reading

Section 3.1 barely scratches the surface of geometric modeling. Most literature focuses on parametric curves and surfaces [376, 718, 788]. These models are not as popular for motion planning because obtaining efficient collision detection is most important in practice, and processing implicit algebraic surfaces is most important in theoretical methods. A thorough coverage of solid and boundary representations, including semi-algebraic models, can be found in [454]. Theoretical algorithm issues regarding semi-algebraic models are covered in [704, 705]. For a comparison of the doubly connected edge list to its variants, see [522].

The material of Section 3.2 appears in virtually any book on robotics, computer vision, or computer graphics. Consulting linear algebra texts may be helpful to gain more insight into rotations. There are many ways to parameterize the set of all 3D rotation matrices. The yaw-pitch-roll formulation was selected because it is the easiest to understand. There are generally 12 different variants of the yaw-pitch-roll formulation (also called *Euler angles*) based on different rotation orderings and axis selections. This formulation, however, is not well suited for the development of motion planning algorithms. It is easy (and safe) to use for making quick 3D animations of motion planning output, but it incorrectly captures the structure of the state space for planning algorithms. Section 4.2 introduces the quaternion parameterization, which correctly captures this state space; however, it is harder to interpret when constructing examples. Therefore, it is helpful to understand both. In addition to Euler angles and quaternions, there is still motivation for using many other parameterizations of rotations, such as spherical coordinates, Cayley-Rodrigues parameters, and stereographic projection. Chapter 5 of [210] provides extensive coverage of 3D rotations and different parameterizations.

The coverage in Section 3.3 of transformations of chains of bodies was heavily influenced by two classic robotics texts [252, 775]. The DH parameters were introduced in [434] and later extended to trees and loops in [524]. An alternative to DH parameters is exponential coordinates [725], which simplify some computations; however, determining the parameters in the modeling stage may be less intuitive. A fascinating history of mechanisms appears in [435]. Other texts on kinematics include [29, 310, 531, 689]. The standard approach in many robotics books [366, 856, 907, 994] is to introduce the kinematic chain formulations and DH parameters in the first couple of chapters, and then move on to topics that are crucial for controlling robot manipulators, including dynamics modeling, singularities, manipulability, and control. Since this book is concerned instead with planning algorithms, we depart at the point where dynamics would usually be covered and move into a careful study of the configuration space in Chapter 4.

Exercises

1. Define a semi-algebraic model that removes a triangular "nose" from the region shown in Figure 3.4.

2. For distinct values of yaw, pitch, and roll, it is possible to generate the same rotation. In other words, $R(\alpha, \beta, \gamma) = R(\alpha', \beta', \gamma')$ for some cases in which at least $\alpha \neq \alpha$, $\beta \neq \beta'$, or $\gamma \neq \gamma'$. Characterize the sets of angles for which this occurs.

3. Using rotation matrices, prove that 2D rotation is commutative but 3D rotation is not.

4. An alternative to the yaw-pitch-roll formulation from Section 3.2.3 is considered here. Consider the following Euler angle representation of rotation (there are many other variants). The first rotation is $R_z(\gamma)$, which is just (3.39) with α replaced by γ. The next two rotations are identical to the yaw-pitch-roll formulation: $R_y(\beta)$ is applied, followed by $R_z(\alpha)$. This yields $R_{euler}(\alpha, \beta, \gamma) = R_z(\alpha)R_y(\beta)R_z(\gamma)$.

   (a) Determine the matrix R_{euler}.

   (b) Show that $R_{euler}(\alpha, \beta, \gamma) = R_{euler}(\alpha - \pi, -\beta, \gamma - \pi)$.

   (c) Suppose that a rotation matrix is given as shown in (3.43). Show that the Euler angles are

$$\alpha = \text{atan2}(r_{23}, r_{13}), \tag{3.85}$$

$$\beta = \text{atan2}(\sqrt{1 - r_{33}^2}, r_{33}), \tag{3.86}$$

   and

$$\gamma = \text{atan2}(r_{32}, -r_{31}). \tag{3.87}$$

5. There are 12 different variants of yaw-pitch-roll (or Euler angles), depending on which axes are used and the order of these axes. Determine all of the possibilities, using only notation such as $R_z(\alpha)R_y(\beta)R_z(\gamma)$ for each one. Give brief arguments that support why or why not specific combinations of rotations are included in your list of 12.

6. Let $\mathcal{A}$ be a unit disc, centered at the origin, and $\mathcal{W} = \mathbb{R}^2$. Assume that $\mathcal{A}$ is represented by a single, algebraic primitive, $H = \{(x, y) \mid x^2 + y^2 \leq 1\}$. Show that the transformed primitive is unchanged after any rotation is applied.

7. Consider the articulated chain of bodies shown in Figure 3.29. There are three identical rectangular bars in the plane, called $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$. Each bar has width 2 and length 12. The distance between the two points of attachment is 10. The first bar, $\mathcal{A}_1$, is attached to the origin. The second bar, $\mathcal{A}_2$, is attached to $\mathcal{A}_1$, and $\mathcal{A}_3$ is attached to $\mathcal{A}_2$. Each bar is allowed to rotate about its point of attachment. The configuration of the chain can be expressed with three angles, $(\theta_1, \theta_2, \theta_3)$. The first angle, θ_1, represents the angle between the segment drawn between the two points of attachment of $\mathcal{A}_1$ and the x-axis. The second angle, θ_2, represents the angle

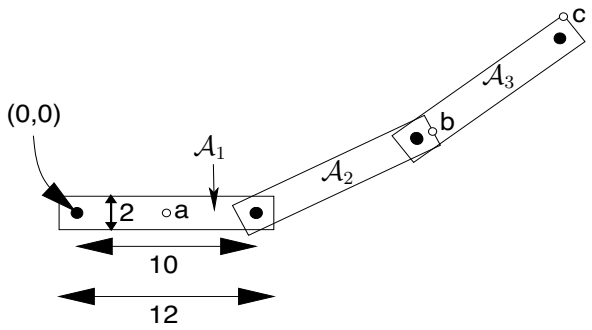


Figure 3.29: A chain of three bodies.

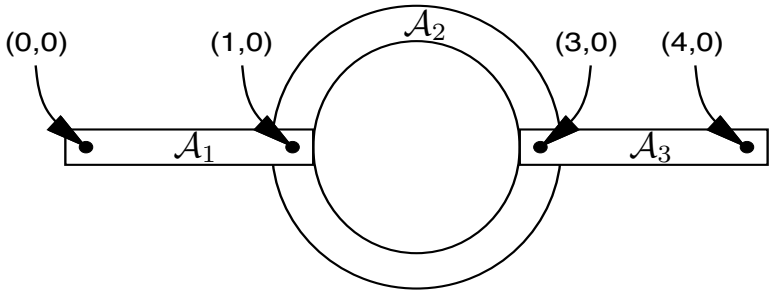


Figure 3.30: Another exercise involving a chain of bodies.

between $\mathcal{A}_2$ and $\mathcal{A}_1$ ($\theta_2 = 0$ when they are parallel). The third angle, θ_3, represents the angle between $\mathcal{A}_3$ and $\mathcal{A}_2$. Suppose the configuration is $(\pi/4, \pi/2, -\pi/4)$.

(a) Use the homogeneous transformation matrices to determine the locations of points a, b, and c.

(b) Characterize the set of all configurations for which the final point of attachment (near the end of $\mathcal{A}_3$) is at $(0, 0)$ (you should be able to figure this out without using the matrices).

8. A three-link chain of bodies that moves in a 2D world is shown Figure 3.30. The first link, $\mathcal{A}_1$, is attached at $(0, 0)$ but can rotate. Each remaining link is attached to another link with a revolute joint. The second link, $\mathcal{A}_2$, is a rigid ring, and the other two links are rectangular bars.

   Assume that the structure is shown in the zero configuration. Suppose that the linkage is moved to the configuration $(\theta_1, \theta_2, \theta_3) = (\frac{\pi}{4}, \frac{\pi}{2}, \frac{\pi}{4})$, in which θ_1 is the angle of $\mathcal{A}_1$, θ_2 is the angle of $\mathcal{A}_2$ with respect to $\mathcal{A}_1$, and θ_3 is the angle of $\mathcal{A}_3$ with respect to $\mathcal{A}_2$. Using homogeneous transformation matrices, compute the position of the point at $(4, 0)$ in Figure 3.30, when the linkage is at configuration $(\frac{\pi}{4}, \frac{\pi}{2}, \frac{\pi}{4})$ (the point is attached to $\mathcal{A}_3$).

9. Approximate a spherical joint as a chain of three short, perpendicular links that are attached by revolute joints and give the sequence of transformation matrices. Show that as the link lengths approach zero, the resulting sequence of transformation matrices converges to exactly representing the freedom of a spherical joint.

Compare this approach to directly using a full rotation matrix, (3.42), to represent the joint in the homogeneous transformation matrix.

10. Figure 3.12 showed six different ways in which 2D surfaces can slide with respect to each other to produce a joint.

    (a) Suppose that two bodies contact each other along a one-dimensional curve. Characterize as many different kinds of "joints" as possible, and indicate the degrees of freedom of each.

    (b) Suppose that the two bodies contact each other at a point. Indicate the types of rolling and sliding that are possible, and their corresponding degrees of freedom.

11. Suppose that two bodies form a screw joint in which the axis of the central axis of the screw aligns with the x-axis of the first body. Determine an appropriate homogeneous transformation matrix to use in place of the DH matrix. Define the matrix with the screw radius, r, and displacement-per-revolution, d, as parameters.

12. Recall Example 3.6. How should the transformations be modified so that the links are in the positions shown in Figure 3.25 at the zero configuration ($\theta_i = 0$ for every revolute joint whose angle can be independently chosen)?

13. Generalize the shearing transformation of (3.84) to enable shearing of 3D models.

Implementations

14. Develop and implement a kinematic model for 2D linkages. Enable the user to display the arrangement of links in the plane.

15. Implement the kinematics of molecules that do not have loops and show them graphically as a "ball and stick" model. The user should be able to input the atomic radii, bond connections, bond lengths, and rotation ranges for each bond.

16. Design and implement a software system in which the user can interactively attach various links to make linkages that resemble those possible from using Tinkertoys (or another popular construction set that allows pieces to move). There are several rods of various lengths, which fit into holes in the center and around the edge of several coin-shaped pieces. Assume that all joints are revolute. The user should be allowed to change parameters and see the resulting positions of all of the links.

17. Construct a model of the human body as a tree of links in a 3D world. For simplicity, the geometric model may be limited to spheres and cylinders. Design and implement a system that displays the virtual human and allows the user to click on joints of the body to enable them to rotate.

18. Develop a simulator with 3D graphics for the Puma 560 model shown in Figure 3.4.

Chapter 4

The Configuration Space

Chapter 3 only covered how to model and transform a collection of bodies; however, for the purposes of planning it is important to define the state space. The state space for motion planning is a set of possible transformations that could be applied to the robot. This will be referred to as the *configuration space*, based on Lagrangian mechanics and the seminal work of Lozano-Pérez [656, 660, 657], who extensively utilized this notion in the context of planning (the idea was also used in early collision avoidance work by Udupa [947]). The motion planning literature was further unified around this concept by Latombe's book [588]. Once the configuration space is clearly understood, many motion planning problems that appear different in terms of geometry and kinematics can be solved by the same planning algorithms. This level of abstraction is therefore very important.

This chapter provides important foundational material that will be very useful in Chapters 5 to 8 and other places where planning over continuous state spaces occurs. Many concepts introduced in this chapter come directly from mathematics, particularly from topology. Therefore, Section 4.1 gives a basic overview of topological concepts. Section 4.2 uses the concepts from Chapter 3 to define the configuration space. After reading this, you should be able to precisely characterize the configuration space of a robot and understand its structure. In Section 4.3, obstacles in the world are transformed into obstacles in the configuration space, but it is important to understand that this transformation may not be explicitly constructed. The implicit representation of the state space is a recurring theme throughout planning. Section 4.4 covers the important case of kinematic chains that have loops, which was mentioned in Section 3.4. This case is so difficult that even the space of transformations usually cannot be explicitly characterized (i.e., parameterized).

4.1 Basic Topological Concepts

This section introduces basic topological concepts that are helpful in understanding configuration spaces. Topology is a challenging subject to understand in depth. The treatment given here provides only a brief overview and is designed to stim-

ulate further study (see the literature overview at the end of the chapter). To advance further in this chapter, it is not necessary to understand all of the material of this section; however, the more you understand, the deeper will be your understanding of motion planning in general.

4.1.1 Topological Spaces

Recall the concepts of open and closed intervals in the set of real numbers $\mathbb{R}$. The open interval $(0, 1)$ includes all real numbers between 0 and 1, *except* 0 and 1. However, for either endpoint, an infinite sequence may be defined that converges to it. For example, the sequence $1/2$, $1/4$, ..., $1/2^i$ converges to 0 as i tends to infinity. This means that we can choose a point in $(0, 1)$ within any small, positive distance from 0 or 1, but we cannot pick one exactly on the boundary of the interval. For a closed interval, such as $[0, 1]$, the boundary points are included.

The notion of an open set lies at the heart of topology. The open set definition that will appear here is a substantial generalization of the concept of an open interval. The concept applies to a very general collection of subsets of some larger space. It is general enough to easily include any kind of configuration space that may be encountered in planning.

A set X is called a *topological space* if there is a collection of subsets of X called *open sets* for which the following axioms hold:

1. The union of any number of open sets is an open set.

2. The intersection of a finite number of open sets is an open set.

3. Both X and $\emptyset$ are open sets.

Note that in the first axiom, the union of an infinite number of open sets may be taken, and the result must remain an open set. Intersecting an infinite number of open sets, however, does not necessarily lead to an open set.

For the special case of $X = \mathbb{R}$, the open sets include open intervals, as expected. Many sets that are not intervals are open sets because taking unions and intersections of open intervals yields other open sets. For example, the set

$$\bigcup_{i=1}^{\infty} \left(\frac{1}{3^i}, \frac{2}{3^i} \right), \tag{4.1}$$

which is an infinite union of pairwise-disjoint intervals, is an open set.

Closed sets Open sets appear directly in the definition of a topological space. It next seems that closed sets are needed. Suppose X is a topological space. A subset $C \subset X$ is defined to be a *closed set* if and only if $X \setminus C$ is an open set. Thus, the complement of any open set is closed, and the complement of any closed set is open. Any closed interval, such as $[0, 1]$, is a closed set because its complement, $(-\infty, 0) \cup (1, \infty)$, is an open set. For another example, $(0, 1)$ is an open set;

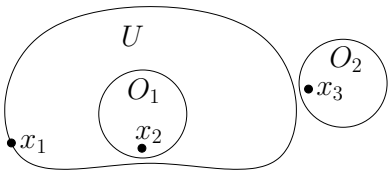


Figure 4.1: An illustration of the boundary definition. Suppose $X = \mathbb{R}^2$, and U is a subset as shown. Three kinds of points appear: 1) x_1 is a boundary point, 2) x_2 is an interior point, and 3) x_3 is an exterior point. Both x_1 and x_2 are limit points of U.

therefore, $\mathbb{R} \setminus (0, 1) = (-\infty, 0] \cup [1, \infty)$ is a closed set. The use of "(" may seem wrong in the last expression, but "[" cannot be used because $-\infty$ and ∞ do not belong to $\mathbb{R}$. Thus, the use of "(" is just a notational quirk.

Are all subsets of X either closed or open? Although it appears that open sets and closed sets are opposites in some sense, the answer is *no*. For $X = \mathbb{R}$, the interval $[0, 2\pi)$ is neither open nor closed (consider its complement: $[2\pi, \infty)$ is closed, and $(-\infty, 0)$ is open). Note that for any topological space, X and $\emptyset$ are both open and closed!

Special points From the definitions and examples so far, it should seem that points on the "edge" or "border" of a set are important. There are several terms that capture where points are relative to the border. Let X be a topological space, and let U be any subset of X. Furthermore, let x be any point in X. The following terms capture the position of point x relative to U (see Figure 4.1):

- If there exists an open set O_1 such that $x \in O_1$ and $O_1 \subseteq U$, then x is called an *interior point* of U. The set of all interior points in U is called the *interior of* U and is denoted by $\text{int}(U)$.

- If there exists an open set O_2 such that $x \in O_2$ and $O_2 \subseteq X \setminus U$, then x is called an *exterior point* with respect to U.

- If x is neither an interior point nor an exterior point, then it is called a *boundary point* of U. The set of all boundary points in X is called the *boundary of* U and is denoted by ∂U.

- All points in $x \in X$ must be one of the three above; however, another term is often used, even though it is redundant given the other three. If x is either an interior point or a boundary point, then it is called a *limit point* (or *accumulation point*) of U. The set of all limit points of U is a closed set called the *closure* of U, and it is denoted by $\text{cl}(U)$. Note that $\text{cl}(U) = \text{int}(U) \cup \partial U$.

For the case of $X = \mathbb{R}$, the boundary points are the endpoints of intervals. For example, 0 and 1 are boundary points of intervals, $(0, 1)$, $[0, 1]$, $[0, 1)$, and $(0, 1]$. Thus, U may or may not include its boundary points. All of the points in $(0, 1)$

are interior points, and all of the points in $[0, 1]$ are limit points. The motivation of the name "limit point" comes from the fact that such a point might be the limit of an infinite sequence of points in U. For example, 0 is the limit point of the sequence generated by $1/2^i$ for each $i \in \mathbb{N}$, the natural numbers.

There are several convenient consequences of the definitions. A closed set C contains the limit point of any sequence that is a subset of C. This implies that it contains all of its boundary points. The closure, cl, always results in a closed set because it adds all of the boundary points to the set. On the other hand, an open set contains none of its boundary points. These interpretations will come in handy when considering obstacles in the configuration space for motion planning.

Some examples The definition of a topological space is so general that an incredible variety of topological spaces can be constructed.

Example 4.1 (The Topology of $\mathbb{R}^n$) We should expect that $X = \mathbb{R}^n$ for any integer n is a topological space. This requires characterizing the open sets. An *open ball* $B(x, \rho)$ is the set of points in the interior of a sphere of radius ρ, centered at x. Thus,

$$B(x, \rho) = \{x' \in \mathbb{R}^n \mid \|x' - x\| < \rho\}, \tag{4.2}$$

in which $\| \cdot \|$ denotes the Euclidean norm (or magnitude) of its argument. The open balls are open sets in $\mathbb{R}^n$. Furthermore, all other open sets can be expressed as a countable union of open balls.[1] For the case of $\mathbb{R}$, this reduces to representing any open set as a union of intervals, which was done so far.

Even though it is possible to express open sets of $\mathbb{R}^n$ as unions of balls, we prefer to use other representations, with the understanding that one could revert to open balls if necessary. The primitives of Section 3.1 can be used to generate many interesting open and closed sets. For example, any algebraic primitive expressed in the form $H = \{x \in \mathbb{R}^n \mid f(x) \leq 0\}$ produces a closed set. Taking finite unions and intersections of these primitives will produce more closed sets. Therefore, all of the models from Sections 3.1.1 and 3.1.2 produce an obstacle region $\mathcal{O}$ that is a closed set. As mentioned in Section 3.1.2, sets constructed only from primitives that use the $<$ relation are open. ■

Example 4.2 (Subspace Topology) A new topological space can easily be constructed from a subset of a topological space. Let X be a topological space, and let $Y \subset X$ be a subset. The *subspace topology* on Y is obtained by defining the open sets to be every subset of Y that can be represented as $U \cap Y$ for some open set $U \subseteq X$. Thus, the open sets for Y are almost the same as for X, except that the points that do not lie in Y are trimmed away. New subspaces can be constructed by intersecting open sets of $\mathbb{R}^n$ with a complicated region defined by semi-algebraic models. This leads to many interesting topological spaces, some of

[1] Such a collection of balls is often referred to as a *basis*.

which will appear later in this chapter. ■

Example 4.3 (The Trivial Topology) For any set X, there is always one trivial example of a topological space that can be constructed from it. Declare that X and $\emptyset$ are the only open sets. Note that all of the axioms are satisfied. ■

Example 4.4 (A Strange Topology) It is important to keep in mind the almost absurd level of generality that is allowed by the definition of a topological space. A topological space can be defined for any set, as long as the declared open sets obey the axioms. Suppose a four-element set is defined as

$$X = \{\text{CAT}, \text{DOG}, \text{TREE}, \text{HOUSE}\}. \tag{4.3}$$

In addition to $\emptyset$ and X, suppose that $\{\text{CAT}\}$ and $\{\text{DOG}\}$ are open sets. Using the axioms, $\{\text{CAT}, \text{DOG}\}$ must also be an open set. Closed sets and boundary points can be derived for this topology once the open sets are defined. ■

After the last example, it seems that topological spaces are so general that not much can be said about them. Most spaces that are considered in topology and analysis satisfy more axioms. For $\mathbb{R}^n$ and any configuration spaces that arise in this book, the following is satisfied:

Hausdorff axiom: For any distinct $x_1, x_2 \in X$, there exist open sets O_1 and O_2 such that $x_1 \in O_1$, $x_2 \in O_2$, and $O_1 \cap O_2 = \emptyset$.

In other words, it is possible to separate x_1 and x_2 into nonoverlapping open sets. Think about how to do this for $\mathbb{R}^n$ by selecting small enough open balls. Any topological space X that satisfies the Hausdorff axiom is referred to as a *Hausdorff space*. Section 4.1.2 will introduce manifolds, which happen to be Hausdorff spaces and are general enough to capture the vast majority of configuration spaces that arise. We will have no need in this book to consider topological spaces that are not Hausdorff spaces.

Continuous functions A very simple definition of continuity exists for topological spaces. It nicely generalizes the definition from standard calculus. Let $f : X \to Y$ denote a function between topological spaces X and Y. For any set $B \subseteq Y$, let the *preimage* of B be denoted and defined by

$$f^{-1}(B) = \{x \in X \mid f(x) \in B\}. \tag{4.4}$$

Note that this definition does not require f to have an inverse.

The function f is called *continuous* if $f^{-1}(O)$ is an open set for every open set $O \subseteq Y$. Analysis is greatly simplified by this definition of continuity. For example, to show that any composition of continuous functions is continuous requires only a one-line argument that the preimage of the preimage of any open set always yields

an open set. Compare this to the cumbersome classical proof that requires a mess of δ's and ϵ's. The notion is also so general that continuous functions can even be defined on the absurd topological space from Example 4.4.

Homeomorphism: Making a donut into a coffee cup You might have heard the expression that to a topologist, a donut and a coffee cup appear the same. In many branches of mathematics, it is important to define when two basic objects are equivalent. In graph theory (and group theory), this equivalence relation is called an *isomorphism*. In topology, the most basic equivalence is a homeomorphism, which allows spaces that appear quite different in most other subjects to be declared equivalent in topology. The surfaces of a donut and a coffee cup (with one handle) are considered equivalent because both have a single hole. This notion needs to be made more precise!

Suppose $f : X \rightarrow Y$ is a bijective (one-to-one and onto) function between topological spaces X and Y. Since f is bijective, the inverse f^{-1} exists. If both f and f^{-1} are continuous, then f is called a *homeomorphism*. Two topological spaces X and Y are said to be *homeomorphic*, denoted by $X \cong Y$, if there exists a homeomorphism between them. This implies an equivalence relation on the set of topological spaces (verify that the reflexive, symmetric, and transitive properties are implied by the homeomorphism).

Example 4.5 (Interval Homeomorphisms) Any open interval of $\mathbb{R}$ is homeomorphic to any other open interval. For example, $(0, 1)$ can be mapped to $(0, 5)$ by the continuous mapping $x \mapsto 5x$. Note that $(0, 1)$ and $(0, 5)$ are each being interpreted here as topological subspaces of $\mathbb{R}$. This kind of homeomorphism can be generalized substantially using linear algebra. If a subset, $X \subset \mathbb{R}^n$, can be mapped to another, $Y \subset \mathbb{R}^n$, via a nonsingular linear transformation, then X and Y are homeomorphic. For example, the rigid-body transformations of the previous chapter were examples of homeomorphisms applied to the robot. Thus, the topology of the robot does not change when it is translated or rotated. (In this example, note that the robot itself is the topological space. This will not be the case for the rest of the chapter.)

Be careful when mixing closed and open sets. The space $[0, 1]$ is not homeomorphic to $(0, 1)$, and neither is homeomorphic to $[0, 1)$. The endpoints cause trouble when trying to make a bijective, continuous function. Surprisingly, a bounded and unbounded set may be homeomorphic. A subset X of $\mathbb{R}^n$ is called *bounded* if there exists a ball $B \subset \mathbb{R}^n$ such that $X \subset B$. The mapping $x \mapsto 1/x$ establishes that $(0, 1)$ and $(1, \infty)$ are homeomorphic. The mapping $x \mapsto 2 \tan^{-1}(x)/\pi$ establishes that $(-1, 1)$ and all of $\mathbb{R}$ are homeomorphic! ■

Example 4.6 (Topological Graphs) Let X be a topological space. The previous example can be extended nicely to make homeomorphisms look like graph

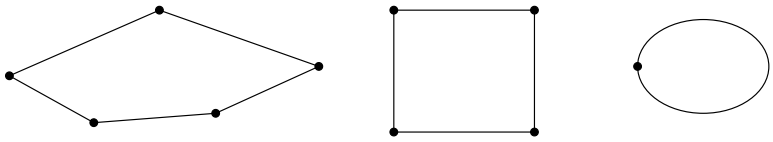


Figure 4.2: Even though the graphs are not isomorphic, the corresponding topological spaces may be homeomorphic due to useless vertices. The example graphs map into $\mathbb{R}^2$, and are all homeomorphic to a circle.

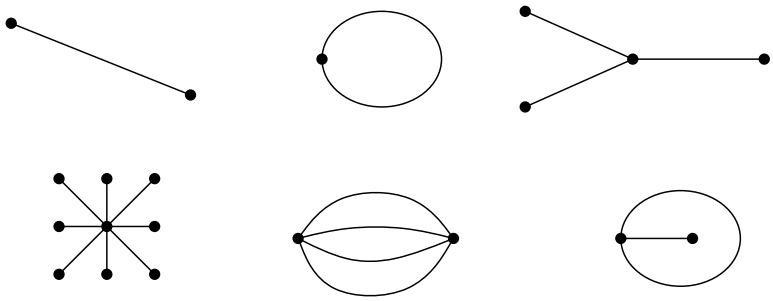


Figure 4.3: These topological graphs map into subsets of $\mathbb{R}^2$ that are not homeomorphic to each other.

isomorphisms. Let a *topological graph*[2] be a graph for which every vertex corresponds to a point in X and every edge corresponds to a continuous, injective (one-to-one) function, $\tau : [0, 1] \to X$. The image of τ connects the points in X that correspond to the endpoints (vertices) of the edge. The images of different edge functions are not allowed to intersect, except at vertices. Recall from graph theory that two graphs, $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, are called *isomorphic* if there exists a bijective mapping, $f : V_1 \to V_2$ such that there is an edge between v_1 and v_1' in G_1, if and only if there exists an edge between $f(v_1)$ and $f(v_1')$ in G_2.

The bijective mapping used in the graph isomorphism can be extended to produce a homeomorphism. Each edge in E_1 is mapped continuously to its corresponding edge in E_2. The mappings nicely coincide at the vertices. Now you should see that two topological graphs are homeomorphic if they are isomorphic under the standard definition from graph theory.[3] What if the graphs are not isomorphic? There is still a chance that the topological graphs may be homeomorphic, as shown in Figure 4.2. The problem is that there appear to be "useless" vertices in the graph. By removing vertices of degree two that can be deleted without affecting the connectivity of the graph, the problem is fixed. In this case,

[2] In topology this is called a 1-complex [439].

[3] Technically, the images of the topological graphs, as subspaces of X, are homeomorphic, not the graphs themselves.

graphs that are not isomorphic produce topological graphs that are not homeomorphic. This allows many distinct, interesting topological spaces to be constructed. A few are shown in Figure 4.3. ■

4.1.2 Manifolds

In motion planning, efforts are made to ensure that the resulting configuration space has nice properties that reflect the true structure of the space of transformations. One important kind of topological space, which is general enough to include most of the configuration spaces considered in Part II, is called a manifold. Intuitively, a manifold can be considered as a "nice" topological space that behaves at every point like our intuitive notion of a surface.

Manifold definition A topological space $M \subseteq \mathbb{R}^m$ is a *manifold*[4] if for every $x \in M$, an open set $O \subset M$ exists such that: 1) $x \in O$, 2) O is homeomorphic to $\mathbb{R}^n$, and 3) n is fixed for all $x \in M$. The fixed n is referred to as the *dimension* of the manifold, M. The second condition is the most important. It states that in the vicinity of any point, $x \in M$, the space behaves just like it would in the vicinity of any point $y \in \mathbb{R}^n$; intuitively, the set of directions that one can move appears the same in either case. Several simple examples that may or may not be manifolds are shown in Figure 4.4.

One natural consequence of the definitions is that $m \geq n$. According to Whitney's embedding theorem [449], $m \leq 2n+1$. In other words, $\mathbb{R}^{2n+1}$ is "big enough" to hold any n-dimensional manifold.[5] Technically, it is said that the n-dimensional manifold M is *embedded* in $\mathbb{R}^m$, which means that an injective mapping exists from M to $\mathbb{R}^m$ (if it is not injective, then the topology of M could change).

As it stands, it is impossible for a manifold to include its boundary points because they are not contained in open sets. A *manifold with boundary* can be defined requiring that the neighborhood of each boundary point of M is homeomorphic to a half-space of dimension n (which was defined for $n = 2$ and $n = 3$ in Section 3.1) and that the interior points must be homeomorphic to $\mathbb{R}^n$.

The presentation now turns to ways of constructing some manifolds that frequently appear in motion planning. It is important to keep in mind that two

[4] Manifolds that are not subsets of $\mathbb{R}^m$ may also be defined. This requires that M is a Hausdorff space and is second countable, which means that there is a countable number of open sets from which any other open set can be constructed by taking a union of some of them. These conditions are automatically satisfied when assuming $M \subseteq \mathbb{R}^m$; thus, it avoids these extra complications and is still general enough for our purposes. Some authors use the term *manifold* to refer to a *smooth manifold*. This requires the definition of a smooth structure, and the homeomorphism is replaced by diffeomorphism. This extra structure is not needed here but will be introduced when it is needed in Section 8.3.

[5] One variant of the theorem is that for smooth manifolds, $\mathbb{R}^{2n}$ is sufficient. This bound is tight because $\mathbb{RP}^n$ (n-dimensional projective space, which will be introduced later in this section), cannot be embedded in $\mathbb{R}^{2n-1}$.

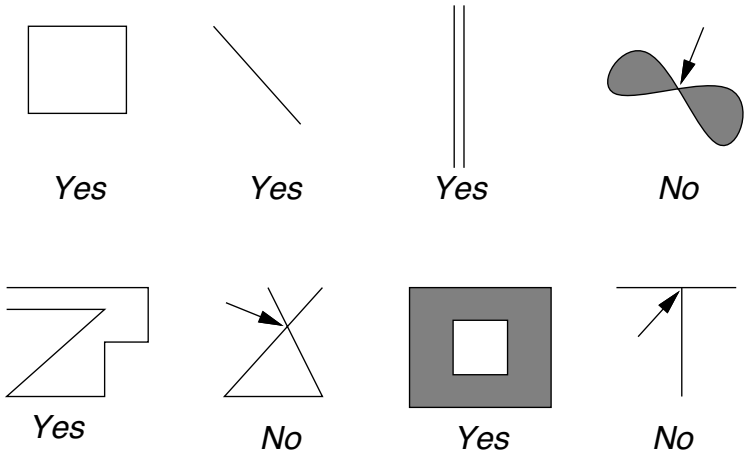


Figure 4.4: Some subsets of $\mathbb{R}^2$ that may or may not be manifolds. For the three that are not, the point that prevents them from being manifolds is indicated.

manifolds will be considered equivalent if they are homeomorphic (recall the donut and coffee cup).

Cartesian products There is a convenient way to construct new topological spaces from existing ones. Suppose that X and Y are topological spaces. The *Cartesian product*, $X \times Y$, defines a new topological space as follows. Every $x \in X$ and $y \in Y$ generates a point (x, y) in $X \times Y$. Each open set in $X \times Y$ is formed by taking the Cartesian product of one open set from X and one from Y. Exactly one open set exists in $X \times Y$ for every pair of open sets that can be formed by taking one from X and one from Y. Furthermore, these new open sets are used as a basis for forming the remaining open sets of $X \times Y$ by allowing any unions and finite intersections of them.

A familiar example of a Cartesian product is $\mathbb{R} \times \mathbb{R}$, which is equivalent to $\mathbb{R}^2$. In general, $\mathbb{R}^n$ is equivalent to $\mathbb{R} \times \mathbb{R}^{n-1}$. The Cartesian product can be taken over many spaces at once. For example, $\mathbb{R} \times \mathbb{R} \times \cdots \times \mathbb{R} = \mathbb{R}^n$. In the coming text, many important manifolds will be constructed via Cartesian products.

1D manifolds The set $\mathbb{R}$ of reals is the most obvious example of a 1D manifold because $\mathbb{R}$ certainly looks like (via homeomorphism) $\mathbb{R}$ in the vicinity of every point. The range can be restricted to the unit interval to yield the manifold $(0, 1)$ because they are homeomorphic (recall Example 4.5).

Another 1D manifold, which is not homeomorphic to $(0, 1)$, is a circle, $\mathbb{S}^1$. In this case $\mathbb{R}^m = \mathbb{R}^2$, and let

$$\mathbb{S}^1 = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}. \tag{4.5}$$

If you are thinking like a topologist, it should appear that this particular circle is not important because there are numerous ways to define manifolds that are

homeomorphic to $\mathbb{S}^1$. For any manifold that is homeomorphic to $\mathbb{S}^1$, we will sometimes say that the manifold *is* $\mathbb{S}^1$, just represented in a different way. Also, $\mathbb{S}^1$ will be called a *circle*, but this is meant only in the topological sense; it only needs to be homeomorphic to the circle that we learned about in high school geometry. Also, when referring to $\mathbb{R}$, we might instead substitute $(0, 1)$ without any trouble. The alternative representations of a manifold can be considered as changing *parameterizations*, which are formally introduced in Section 8.3.2.

Identifications A convenient way to represent $\mathbb{S}^1$ is obtained by *identification*, which is a general method of declaring that some points of a space are identical, even though they originally were distinct.[6] For a topological space X, let $X/\sim$ denote that X has been redefined through some form of identification. The open sets of X become redefined. Using identification, $\mathbb{S}^1$ can be defined as $[0, 1]/\sim$, in which the identification declares that 0 and 1 are equivalent, denoted as $0 \sim 1$. This has the effect of "gluing" the ends of the interval together, forming a closed loop. To see the homeomorphism that makes this possible, use polar coordinates to obtain $\theta \mapsto (\cos 2\pi\theta, \sin 2\pi\theta)$. You should already be familiar with 0 and 2π leading to the same point in polar coordinates; here they are just normalized to 0 and 1. Letting θ run from 0 up to 1, and then "wrapping around" to 0 is a convenient way to represent $\mathbb{S}^1$ because it does not need to be curved as in (4.5).

It might appear that identifications are cheating because the definition of a manifold requires it to be a subset of $\mathbb{R}^m$. This is not a problem because Whitney's theorem, as mentioned previously, states that any n-dimensional manifold can be embedded in $\mathbb{R}^{2n+1}$. The identifications just reduce the number of dimensions needed for visualization. They are also convenient in the implementation of motion planning algorithms.

2D manifolds Many important, 2D manifolds can be defined by applying the Cartesian product to 1D manifolds. The 2D manifold $\mathbb{R}^2$ is formed by $\mathbb{R} \times \mathbb{R}$. The product $\mathbb{R} \times \mathbb{S}^1$ defines a manifold that is equivalent to an infinite cylinder. The product $\mathbb{S}^1 \times \mathbb{S}^1$ is a manifold that is equivalent to a torus (the surface of a donut).

Can any other 2D manifolds be defined? See Figure 4.5. The identification idea can be applied to generate several new manifolds. Start with an open square $M = (0, 1) \times (0, 1)$, which is homeomorphic to $\mathbb{R}^2$. Let (x, y) denote a point in the plane. A *flat cylinder* is obtained by making the identification $(0, y) \sim (1, y)$ for all $y \in (0, 1)$ and adding all of these points to M. The result is depicted in Figure 4.5 by drawing arrows where the identification occurs.

A *Möbius band* can be constructed by taking a strip of paper and connecting the ends after making a 180-degree twist. This result is not homeomorphic to the cylinder. The Möbius band can also be constructed by putting the twist into the identification, as $(0, y) \sim (1, 1 - y)$ for all $y \in (0, 1)$. In this case, the arrows are drawn in opposite directions. The Möbius band has the famous properties that

[6] This is usually defined more formally and called a *quotient topology*.

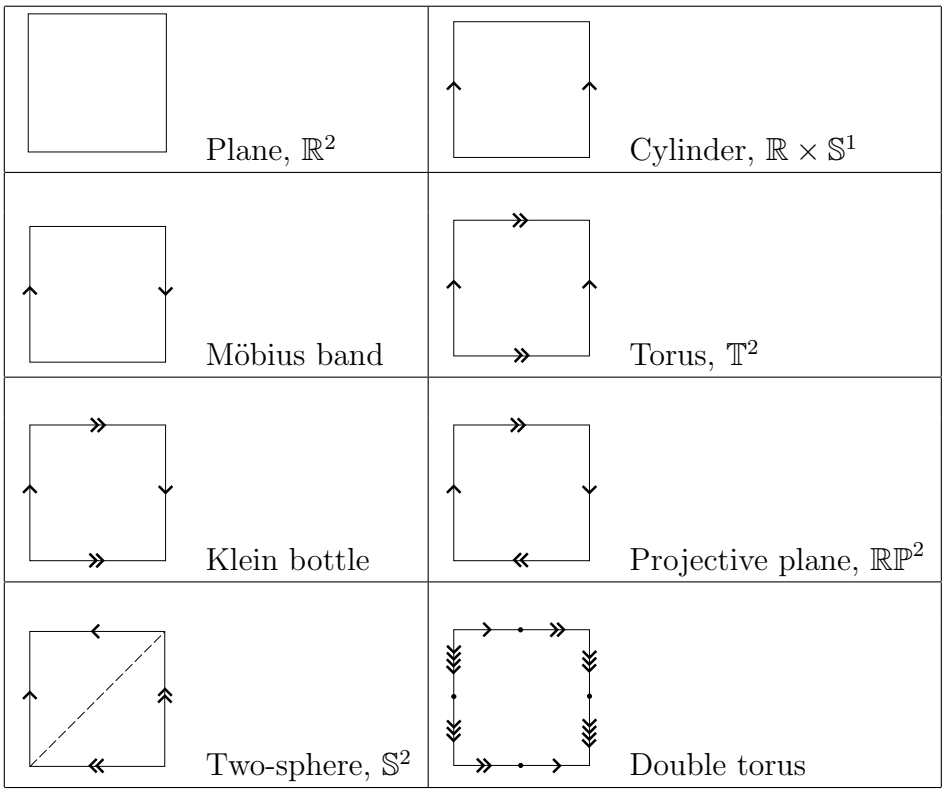


Figure 4.5: Some 2D manifolds that can be obtained by identifying pairs of points along the boundary of a square region.

it has only one side (trace along the paper strip with a pencil, and you will visit both sides of the paper) and is nonorientable (if you try to draw it in the plane, without using identification tricks, it will always have a twist).

For all of the cases so far, there has been a boundary to the set. The next few manifolds will not even have a boundary, even though they may be bounded. If you were to live in one of them, it means that you could walk forever along any trajectory and never encounter the edge of your universe. It might seem like our physical universe is unbounded, but it would only be an illusion. Furthermore, there are several distinct possibilities for the universe that are not homeomorphic to each other. In higher dimensions, such possibilities are the subject of cosmology, which is a branch of astrophysics that uses topology to characterize the structure of our universe.

A *torus* can be constructed by performing identifications of the form $(0, y) \sim (1, y)$, which was done for the cylinder, and also $(x, 0) \sim (x, 1)$, which identifies the top and bottom. Note that the point $(0, 0)$ must be included and is identified with three other points. Double arrows are used in Figure 4.5 to indicate the top and bottom identification. All of the identification points must be added to M. Note that there are no twists. A funny interpretation of the resulting *flat torus* is as the universe appears for a spacecraft in some 1980s-style *Asteroids*-like video games.

The spaceship flies off of the screen in one direction and appears somewhere else, as prescribed by the identification.

Two interesting manifolds can be made by adding twists. Consider performing all of the identifications that were made for the torus, except put a twist in the side identification, as was done for the Möbius band. This yields a fascinating manifold called the *Klein bottle*, which can be embedded in $\mathbb{R}^4$ as a closed 2D surface in which the inside and the outside are the same! (This is in a sense similar to that of the Möbius band.) Now suppose there are twists in both the sides and the top and bottom. This results in the most bizarre manifold yet: the real projective plane, $\mathbb{RP}^2$. This space is equivalent to the set of all lines in $\mathbb{R}^3$ that pass through the origin. The 3D version, $\mathbb{RP}^3$, happens to be one of the most important manifolds for motion planning!

Let $\mathbb{S}^2$ denote the unit sphere, which is defined as

$$\mathbb{S}^2 = \{(x, y, z) \in \mathbb{R}^3 \mid x^2 + y^2 + z^2 = 1\}. \tag{4.6}$$

Another way to represent $\mathbb{S}^2$ is by making the identifications shown in the last row of Figure 4.5. A dashed line is indicated where the equator might appear, if we wanted to make a distorted wall map of the earth. The poles would be at the upper left and lower right corners. The final example shown in Figure 4.5 is a *double torus*, which is the surface of a two-holed donut.

Higher dimensional manifolds The construction techniques used for the 2D manifolds generalize nicely to higher dimensions. Of course, $\mathbb{R}^n$, is an n-dimensional manifold. An n-dimensional torus, $\mathbb{T}^n$, can be made by taking a Cartesian product of n copies of $\mathbb{S}^1$. Note that $\mathbb{S}^1 \times \mathbb{S}^1 \neq \mathbb{S}^2$. Therefore, the notation $\mathbb{T}^n$ is used for $(\mathbb{S}^1)^n$. Different kinds of n-dimensional cylinders can be made by forming a Cartesian product $\mathbb{R}^i \times \mathbb{T}^j$ for positive integers i and j such that $i + j = n$. Higher dimensional spheres are defined as

$$\mathbb{S}^n = \{x \in \mathbb{R}^{n+1} \mid \|x\| = 1\}, \tag{4.7}$$

in which $\|x\|$ denotes the Euclidean norm of x, and n is a positive integer. Many interesting spaces can be made by identifying faces of the cube $(0, 1)^n$ (or even faces of a polyhedron or polytope), especially if different kinds of twists are allowed. An n-dimensional projective space can be defined in this way, for example. *Lens spaces* are a family of manifolds that can be constructed by identification of polyhedral faces [834].

Due to its coming importance in motion planning, more details are given on projective spaces. The standard definition of an *n-dimensional real projective space* $\mathbb{RP}^n$ is the set of all lines in $\mathbb{R}^{n+1}$ that pass through the origin. Each line is considered as a point in $\mathbb{RP}^n$. Using the definition of $\mathbb{S}^n$ in (4.7), note that each of these lines in $\mathbb{R}^{n+1}$ intersects $\mathbb{S}^n \subset \mathbb{R}^{n+1}$ in exactly two places. These intersection points are called *antipodal*, which means that they are as far from each other as possible on $\mathbb{S}^n$. The pair is also unique for each line. If we identify

all pairs of antipodal points of $\mathbb{S}^n$, a homeomorphism can be defined between each line through the origin of $\mathbb{R}^{n+1}$ and each antipodal pair on the sphere. This means that the resulting manifold, $\mathbb{S}^n / \sim$, is homeomorphic to $\mathbb{RP}^n$.

Another way to interpret the identification is that $\mathbb{RP}^n$ is just the upper half of $\mathbb{S}^n$, but with every equatorial point identified with its antipodal point. Thus, if you try to walk into the southern hemisphere, you will find yourself on the other side of the world walking north. It is helpful to visualize the special case of $\mathbb{RP}^2$ and the upper half of $\mathbb{S}^2$. Imagine warping the picture of $\mathbb{RP}^2$ from Figure 4.5 from a square into a circular disc, with opposite points identified. The result still represents $\mathbb{RP}^2$. The center of the disc can now be lifted out of the plane to form the upper half of $\mathbb{S}^2$.

4.1.3 Paths and Connectivity

Central to motion planning is determining whether one part of a space is reachable from another. In Chapter 2, one state was reached from another by applying a sequence of actions. For motion planning, the analog to this is connecting one point in the configuration space to another by a continuous path. Graph connectivity is important in the discrete planning case. An analog to this for topological spaces is presented in this section.

Paths Let X be a topological space, which for our purposes will also be a manifold. A *path* is a continuous function, $\tau : [0, 1] \rightarrow X$. Alternatively, $\mathbb{R}$ may be used for the domain of τ. Keep in mind that a path is a function, not a set of points. Each point along the path is given by $\tau(s)$ for some $s \in [0, 1]$. This makes it appear as a nice generalization to the sequence of states visited when a plan from Chapter 2 is applied. Recall that there, a countable set of stages was defined, and the states visited could be represented as x_1, x_2, ... In the current setting $\tau(s)$ is used, in which s replaces the stage index. To make the connection clearer, we could use x instead of τ to obtain $x(s)$ for each $s \in [0, 1]$.

Connected vs. path connected A topological space X is said to be *connected* if it cannot be represented as the union of two disjoint, nonempty, open sets. While this definition is rather elegant and general, if X is connected, it does not imply that a path exists between any pair of points in X thanks to crazy examples like the *topologist's sine curve*:

$$X = \{(x, y) \in \mathbb{R}^2 \mid x = 0 \text{ or } y = \sin(1/x)\}. \tag{4.8}$$

Consider plotting X. The $\sin(1/x)$ part creates oscillations near the y-axis in which the frequency tends to infinity. After union is taken with the y-axis, this space is connected, but there is no path that reaches the y-axis from the sine curve.

How can we avoid such problems? The standard way to fix this is to use the path definition directly in the definition of connectedness. A topological space X is said to be *path connected* if for all $x_1, x_2 \in X$, there exists a path τ such that

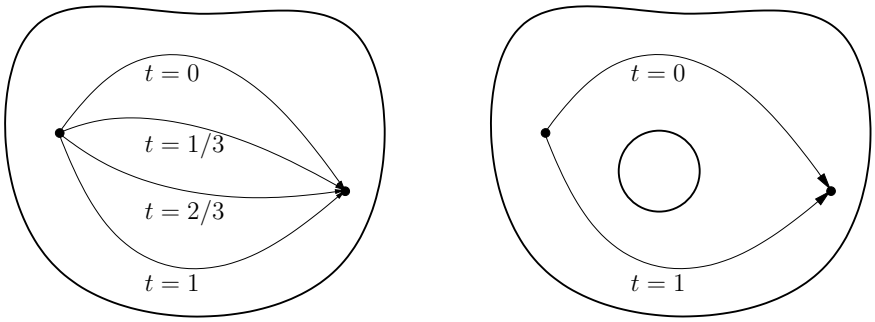


Figure 4.6: (a) Homotopy continuously warps one path into another. (b) The image of the path cannot be continuously warped over a hole in $\mathbb{R}^2$ because it causes a discontinuity. In this case, the two paths are not homotopic.

$\tau(0) = x_1$ and $\tau(1) = x_2$. It can be shown that if X is path connected, then it is also connected in the sense defined previously.

Another way to fix it is to make restrictions on the kinds of topological spaces that will be considered. This approach will be taken here by assuming that all topological spaces are manifolds. In this case, no strange things like (4.8) can happen,[7] and the definitions of connected and path connected coincide [451]. Therefore, we will just say a space is *connected*. However, it is important to remember that this definition of connected is sometimes inadequate, and one should really say that X is *path connected*.

Simply connected Now that the notion of connectedness has been established, the next step is to express different kinds of connectivity. This may be done by using the notion of homotopy, which can intuitively be considered as a way to continuously "warp" or "morph" one path into another, as depicted in Figure 4.6a.

Two paths τ_1 and τ_2 are called *homotopic* (with endpoints fixed) if there exists a continuous function $h : [0,1] \times [0,1] \rightarrow X$ for which the following four conditions are met:

1. (**Start with first path**) $h(s,0) = \tau_1(s)$ for all $s \in [0,1]$.
2. (**End with second path**) $h(s,1) = \tau_2(s)$ for all $s \in [0,1]$.
3. (**Hold starting point fixed**) $h(0,t) = h(0,0)$ for all $t \in [0,1]$.
4. (**Hold ending point fixed**) $h(1,t) = h(1,0)$ for all $t \in [0,1]$.

[7] The topologist's sine curve is not a manifold because all open sets that contain the point $(0,0)$ contain some of the points from the sine curve. These open sets are not homeomorphic to $\mathbb{R}$.

The parameter t can be interpreted as a knob that is turned to gradually deform the path from τ_1 into τ_2. The first two conditions indicate that $t = 0$ yields τ_1 and $t = 1$ yields τ_2, respectively. The remaining two conditions indicate that the path endpoints are held fixed.

During the warping process, the path image cannot make a discontinuous jump. In $\mathbb{R}^2$, this prevents it from moving over holes, such as the one shown in Figure 4.6b. The key to preventing homotopy from jumping over some holes is that h must be continuous. In higher dimensions, however, there are many different kinds of holes. For the case of $\mathbb{R}^3$, for example, suppose the space is like a block of Swiss cheese that contains air bubbles. Homotopy can go around the air bubbles, but it cannot pass through a hole that is drilled through the entire block of cheese. Air bubbles and other kinds of holes that appear in higher dimensions can be characterized by generalizing homotopy to the warping of higher dimensional surfaces, as opposed to paths [439].

It is straightforward to show that homotopy defines an equivalence relation on the set of all paths from some $x_1 \in X$ to some $x_2 \in X$. The resulting notion of "equivalent paths" appears frequently in motion planning, control theory, and many other contexts. Suppose that X is path connected. If all paths fall into the same equivalence class, then X is called *simply connected*; otherwise, X is called *multiply connected*.

Groups The equivalence relation induced by homotopy starts to enter the realm of algebraic topology, which is a branch of mathematics that characterizes the structure of topological spaces in terms of algebraic objects, such as groups. These resulting groups have important implications for motion planning. Therefore, we give a brief overview. First, the notion of a group must be precisely defined. A *group* is a set, G, together with a binary operation, $\circ$, such that the following *group axioms* are satisfied:

1. (**Closure**) For any $a, b \in G$, the product $a \circ b \in G$.
2. (**Associativity**) For all $a, b, c \in G$, $(a \circ b) \circ c = a \circ (b \circ c)$. Hence, parentheses are not needed, and the product may be written as $a \circ b \circ c$.
3. (**Identity**) There is an element $e \in G$, called the *identity*, such that for all $a \in G$, $e \circ a = a$ and $a \circ e = a$.
4. (**Inverse**) For every element $a \in G$, there is an element a^{-1}, called the *inverse* of a, for which $a \circ a^{-1} = e$ and $a^{-1} \circ a = e$.

Here are some examples.

Example 4.7 (Simple Examples of Groups) The set of integers $\mathbb{Z}$ is a group with respect to addition. The identity is 0, and the inverse of each i is $-i$. The set $\mathbb{Q} \setminus 0$ of rational numbers with 0 removed is a group with respect to multiplication. The identity is 1, and the inverse of every element, q, is $1/q$ (0 was removed to

avoid division by zero). ■

An important property, which only some groups possess, is *commutativity*: $a \circ b = b \circ a$ for any $a, b \in G$. The group in this case is called *commutative* or *Abelian*. We will encounter examples of both kinds of groups, both commutative and noncommutative. An example of a commutative group is vector addition over $\mathbb{R}^n$. The set of all 3D rotations is an example of a noncommutative group.

The fundamental group Now an interesting group will be constructed from the space of paths and the equivalence relation obtained by homotopy. The *fundamental group*, $\pi_1(X)$ (or *first homotopy group*), is associated with any topological space, X. Let a (continuous) path for which $f(0) = f(1)$ be called a *loop*. Let some $x_b \in X$ be designated as a *base point*. For some arbitrary but fixed base point, x_b, consider the set of all loops such that $f(0) = f(1) = x_b$. This can be made into a group by defining the following binary operation. Let $\tau_1 : [0,1] \rightarrow X$ and $\tau_2 : [0,1] \rightarrow X$ be two loop paths with the same base point. Their product $\tau = \tau_1 \circ \tau_2$ is defined as

$$\tau(t) = \begin{cases} \tau_1(2t) & \text{if } t \in [0, 1/2) \\ \tau_2(2t-1) & \text{if } t \in [1/2, 1]. \end{cases} \tag{4.9}$$

This results in a continuous loop path because τ_1 terminates at x_b, and τ_2 begins at x_b. In a sense, the two paths are concatenated end-to-end.

Suppose now that the equivalence relation induced by homotopy is applied to the set of all loop paths through a fixed point, x_b. It will no longer be important which particular path was chosen from a class; any representative may be used. The equivalence relation also applies when the set of loops is interpreted as a group. The group operation actually occurs over the set of equivalences of paths.

Consider what happens when two paths from different equivalence classes are concatenated using $\circ$. Is the resulting path homotopic to either of the first two? Is the resulting path homotopic if the original two are from the same homotopy class? The answers in general are *no* and *no*, respectively. The fundamental group describes how the equivalence classes of paths are related and characterizes the connectivity of X. Since fundamental groups are based on paths, there is a nice connection to motion planning.

Example 4.8 (A Simply Connected Space) Suppose that a topological space X is simply connected. In this case, all loop paths from a base point x_b are homotopic, resulting in one equivalence class. The result is $\pi_1(X) = \mathbf{1_G}$, which is the group that consists of only the identity element. ■

Example 4.9 (The Fundamental Group of $\mathbb{S}^1$) Suppose $X = \mathbb{S}^1$. In this case, there is an equivalence class of paths for each $i \in \mathbb{Z}$, the set of integers.

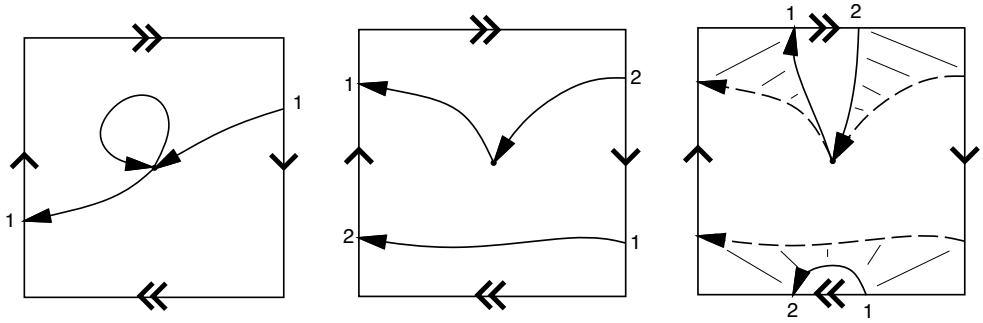


Figure 4.7: An illustration of why $\pi_1(\mathbb{RP}^2) = \mathbb{Z}_2$. The integers 1 and 2 indicate precisely where a path continues when it reaches the boundary. (a) Two paths are shown that are not equivalent. (b) A path that winds around twice is shown. (c) This is homotopic to a loop path that does not wind around at all. Eventually, the part of the path that appears at the bottom is pulled through the top. It finally shrinks into an arbitrarily small loop.

If $i > 0$, then it means that the path winds i times around $\mathbb{S}^1$ in the counterclockwise direction and then returns to x_b. If $i < 0$, then the path winds around i times in the clockwise direction. If $i = 0$, then the path is equivalent to one that remains at x_b. The fundamental group is $\mathbb{Z}$, with respect to the operation of addition. If τ_1 travels i_1 times counterclockwise, and τ_2 travels i_2 times counterclockwise, then $\tau = \tau_1 \circ \tau_2$ belongs to the class of loops that travel around $i_1 + i_2$ times counterclockwise. Consider additive inverses. If a path travels seven times around $\mathbb{S}^1$, and it is combined with a path that travels seven times in the opposite direction, the result is homotopic to a path that remains at x_b. Thus, $\pi_1(\mathbb{S}^1) = \mathbb{Z}$. ■

Example 4.10 (The Fundamental Group of $\mathbb{T}^n$) For the torus, $\pi_1(\mathbb{T}^n) = \mathbb{Z}^n$, in which the ith component of $\mathbb{Z}^n$ corresponds to the number of times a loop path wraps around the ith component of $\mathbb{T}^n$. This makes intuitive sense because $\mathbb{T}^n$ is just the Cartesian product of n circles. The fundamental group $\mathbb{Z}^n$ is obtained by starting with a simply connected subset of the plane and drilling out n disjoint, bounded holes. This situation arises frequently when a mobile robot must avoid collision with n disjoint obstacles in the plane. ■

By now it seems that the fundamental group simply keeps track of how many times a path travels around holes. This next example yields some very bizarre behavior that helps to illustrate some of the interesting structure that arises in algebraic topology.

Example 4.11 (The Fundamental Group of $\mathbb{RP}^2$) Suppose $X = \mathbb{RP}^2$, the projective plane. In this case, there are only two equivalence classes on the space of loop paths. All paths that "wrap around" an even number of times are homotopic. Likewise, all paths that wrap around an odd number of times are homotopic. This strange behavior is illustrated in Figure 4.7. The resulting fundamental group therefore has only two elements: $\pi_1(\mathbb{RP}^2) = \mathbb{Z}_2$, the cyclic group of order 2, which corresponds to addition mod 2. This makes intuitive sense because the group keeps track of whether a sum of integers is odd or even, which in this application corresponds to the total number of traversals over the square representation of $\mathbb{RP}^2$. The fundamental group is the same for $\mathbb{RP}^3$, which arises in Section 4.2.2 because it is homeomorphic to the set of 3D rotations. Thus, there are surprisingly only two path classes for the set of 3D rotations. ■

Unfortunately, two topological spaces may have the same fundamental group even if the spaces are not homeomorphic. For example, $\mathbb{Z}$ is the fundamental group of $\mathbb{S}^1$, the cylinder, $\mathbb{R} \times \mathbb{S}^1$, and the Möbius band. In the last case, the fundamental group does not indicate that there is a "twist" in the space. Another problem is that spaces with interesting connectivity may be declared as simply connected. The fundamental group of the sphere $\mathbb{S}^2$ is just $\mathbf{1_G}$, the same as for $\mathbb{R}^2$. Try envisioning loop paths on the sphere; it can be seen that they all fall into one equivalence class. Hence, $\mathbb{S}^2$ is simply connected. The fundamental group also neglects bubbles in $\mathbb{R}^3$ because the homotopy can warp paths around them. Some of these troubles can be fixed by defining second-order homotopy groups. For example, a continuous function, $[0, 1] \times [0, 1] \rightarrow X$, of two variables can be used instead of a path. The resulting homotopy generates a kind of sheet or surface that can be warped through the space, to yield a homotopy group $\pi_2(X)$ that wraps around bubbles in $\mathbb{R}^3$. This idea can be extended beyond two dimensions to detect many different kinds of holes in higher dimensional spaces. This leads to the *higher order homotopy groups*. A stronger concept than simply connected for a space is that its homotopy groups of all orders are equal to the identity group. This prevents all kinds of holes from occurring and implies that a space, X, is *contractible*, which means a kind of homotopy can be constructed that shrinks X to a point [439]. In the plane, the notions of *contractible* and *simply connected* are equivalent; however, in higher dimensional spaces, such as those arising in motion planning, the term *contractible* should be used to indicate that the space has no interior obstacles (holes).

An alternative to basing groups on homotopy is to derive them using *homology*, which is based on the structure of cell complexes instead of homotopy mappings. This subject is much more complicated to present, but it is more powerful for proving theorems in topology. See the literature overview at the end of the chapter for suggested further reading on algebraic topology.

4.2 Defining the Configuration Space

This section defines the manifolds that arise from the transformations of Chapter 3. If the robot has n degrees of freedom, the set of transformations is usually a manifold of dimension n. This manifold is called the *configuration space* of the robot, and its name is often shortened to *C-space*. In this book, the C-space may be considered as a special state space. To solve a motion planning problem, algorithms must conduct a search in the C-space. The C-space provides a powerful abstraction that converts the complicated models and transformations of Chapter 3 into the general problem of computing a path that traverses a manifold. By developing algorithms directly for this purpose, they apply to a wide variety of different kinds of robots and transformations. In Section 4.3 the problem will be complicated by bringing obstacles into the configuration space, but in Section 4.2 there will be no obstacles.

4.2.1 2D Rigid Bodies: $SE(2)$

Section 3.2.2 expressed how to transform a rigid body in $\mathbb{R}^2$ by a homogeneous transformation matrix, T, given by (3.35). The task in this chapter is to characterize the set of all possible rigid-body transformations. Which manifold will this be? Here is the answer and brief explanation. Since any $x_t, y_t \in \mathbb{R}$ can be selected for translation, this alone yields a manifold $M_1 = \mathbb{R}^2$. Independently, any rotation, $\theta \in [0, 2\pi)$, can be applied. Since 2π yields the same rotation as 0, they can be identified, which makes the set of 2D rotations into a manifold, $M_2 = \mathbb{S}^1$. To obtain the manifold that corresponds to all rigid-body motions, simply take $\mathcal{C} = M_1 \times M_2 = \mathbb{R}^2 \times \mathbb{S}^1$. The answer to the question is that the C-space is a kind of cylinder.

Now we give a more detailed technical argument. The main purpose is that such a simple, intuitive argument will not work for the 3D case. Our approach is to introduce some of the technical machinery here for the 2D case, which is easier to understand, and then extend it to the 3D case in Section 4.2.2.

Matrix groups The first step is to consider the set of transformations as a group, in addition to a topological space.[8] We now derive several important groups from sets of matrices, ultimately leading to $SO(n)$, the group of $n \times n$ rotation matrices, which is very important for motion planning. The set of all nonsingular $n \times n$ real-valued matrices is called the *general linear group*, denoted by $GL(n)$, with respect to matrix multiplication. Each matrix $A \in GL(n)$ has an inverse $A^{-1} \in GL(n)$, which when multiplied yields the identity matrix, $AA^{-1} = I$. The

[8]The groups considered in this section are actually Lie groups because they are smooth manifolds [63]. We will not use that name here, however, because the notion of a smooth structure has not yet been defined. Readers familiar with Lie groups, however, will recognize most of the coming concepts. Some details on Lie groups appear later in Sections 15.4.3 and 15.5.1.

matrices must be nonsingular for the same reason that 0 was removed from $\mathbb{Q}$. The analog of division by zero for matrix algebra is the inability to invert a singular matrix.

Many interesting groups can be formed from one group, G_1, by removing some elements to obtain a *subgroup*, G_2. To be a subgroup, G_2 must be a subset of G_1 and satisfy the group axioms. We will arrive at the set of rotation matrices by constructing subgroups. One important subgroup of $GL(n)$ is the *orthogonal group*, $O(n)$, which is the set of all matrices $A \in GL(n)$ for which $AA^T = I$, in which A^T denotes the matrix *transpose* of A. These matrices have orthogonal columns (the inner product of any pair is zero) and the determinant is always 1 or -1. Thus, note that AA^T takes the inner product of every pair of columns. If the columns are different, the result must be 0; if they are the same, the result is 1 because $AA^T = I$. The *special orthogonal group*, $SO(n)$, is the subgroup of $O(n)$ in which every matrix has determinant 1. Another name for $SO(n)$ is the *group of n-dimensional rotation matrices*.

A chain of groups, $SO(n) \leq O(n) \leq GL(n)$, has been described in which $\leq$ denotes "a subgroup of." Each group can also be considered as a topological space. The set of all $n \times n$ matrices (which is not a group with respect to multiplication) with real-valued entries is homeomorphic to $\mathbb{R}^{n^2}$ because n^2 entries in the matrix can be independently chosen. For $GL(n)$, singular matrices are removed, but an n^2-dimensional manifold is nevertheless obtained. For $O(n)$, the expression $AA^T = I$ corresponds to n^2 algebraic equations that have to be satisfied. This should substantially drop the dimension. Note, however, that many of the equations are redundant (pick your favorite value for n, multiply the matrices, and see what happens). There are only $\binom{n}{2}$ ways (pairwise combinations) to take the inner product of pairs of columns, and there are n equations that require the magnitude of each column to be 1. This yields a total of $n(n+1)/2$ independent equations. Each independent equation drops the manifold dimension by one, and the resulting dimension of $O(n)$ is $n^2 - n(n+1)/2 = n(n-1)/2$, which is easily remembered as $\binom{n}{2}$. To obtain $SO(n)$, the constraint $\det A = 1$ is added, which eliminates exactly half of the elements of $O(n)$ but keeps the dimension the same.

Example 4.12 (Matrix Subgroups) It is helpful to illustrate the concepts for $n = 2$. The set of all 2×2 matrices is

$$\left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \;\middle|\; a, b, c, d \in \mathbb{R} \right\}, \tag{4.10}$$

which is homeomorphic to $\mathbb{R}^4$. The group $GL(2)$ is formed from the set of all nonsingular 2×2 matrices, which introduces the constraint that $ad - bc \neq 0$. The set of singular matrices forms a 3D manifold with boundary in $\mathbb{R}^4$, but all other elements of $\mathbb{R}^4$ are in $GL(2)$; therefore, $GL(2)$ is a 4D manifold.

Next, the constraint $AA^T = I$ is enforced to obtain $O(2)$. This becomes

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} a & c \\ b & d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \tag{4.11}$$

which directly yields four algebraic equations:

$$a^2 + b^2 = 1 \tag{4.12}$$

$$ac + bd = 0 \tag{4.13}$$

$$ca + db = 0 \tag{4.14}$$

$$c^2 + d^2 = 1. \tag{4.15}$$

Note that (4.14) is redundant. There are two kinds of equations. One equation, given by (4.13), forces the inner product of the columns to be 0. There is only one because $\binom{n}{2} = 1$ for $n = 2$. Two other constraints, (4.12) and (4.15), force the rows to be unit vectors. There are two because $n = 2$. The resulting dimension of the manifold is $\binom{n}{2} = 1$ because we started with $\mathbb{R}^4$ and lost three dimensions from (4.12), (4.13), and (4.15). What does this manifold look like? Imagine that there are two different two-dimensional unit vectors, (a, b) and (c, d). Any value can be chosen for (a, b) as long as $a^2 + b^2 = 1$. This looks like $\mathbb{S}^1$, but the inner product of (a, b) and (c, d) must also be 0. Therefore, for each value of (a, b), there are two choices for c and d: 1) $c = b$ and $d = -a$, or 2) $c = -b$ and $d = a$. It appears that there are two circles! The manifold is $\mathbb{S}^1 \sqcup \mathbb{S}^1$, in which $\sqcup$ denotes the union of disjoint sets. Note that this manifold is not connected because no path exists from one circle to the other.

The final step is to require that $\det A = ad - bc = 1$, to obtain $SO(2)$, the set of all 2D rotation matrices. Without this condition, there would be matrices that produce a rotated mirror image of the rigid body. The constraint simply forces the choice for c and d to be $c = -b$ and $a = d$. This throws away one of the circles from $O(2)$, to obtain a single circle for $SO(2)$. We have finally obtained what you already knew: $SO(2)$ is homeomorphic to $\mathbb{S}^1$. The circle can be parameterized using polar coordinates to obtain the standard 2D rotation matrix, (3.31), given in Section 3.2.2. ■

Special Euclidean group Now that the group of rotations, $SO(n)$, is characterized, the next step is to allow both rotations and translations. This corresponds to the set of all $(n + 1) \times (n + 1)$ transformation matrices of the form

$$\left\{ \begin{pmatrix} R & v \\ 0 & 1 \end{pmatrix} \;\middle|\; R \in SO(n) \text{ and } v \in \mathbb{R}^n \right\}. \tag{4.16}$$

This should look like a generalization of (3.52) and (3.56), which were for $n = 2$ and $n = 3$, respectively. The R part of the matrix achieves rotation of an n-dimensional body in $\mathbb{R}^n$, and the v part achieves translation of the same body. The result is a group, $SE(n)$, which is called the *special Euclidean group*. As a topological space, $SE(n)$ is homeomorphic to $\mathbb{R}^n \times SO(n)$, because the rotation matrix and translation vectors may be chosen independently. In the case of $n = 2$, this means $SE(2)$ is homeomorphic to $\mathbb{R}^2 \times \mathbb{S}^1$, which verifies what was stated

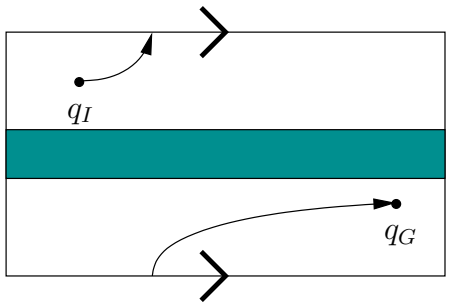


Figure 4.8: A planning algorithm may have to cross the identification boundary to find a solution path.

at the beginning of this section. Thus, the C-space of a 2D rigid body that can translate and rotate in the plane is

$$\mathcal{C} = \mathbb{R}^2 \times \mathbb{S}^1. \tag{4.17}$$

To be more precise, $\cong$ should be used in the place of $=$ to indicate that $\mathcal{C}$ could be any space homeomorphic to $\mathbb{R}^2 \times \mathbb{S}^1$; however, this notation will mostly be avoided.

Interpreting the C-space It is important to consider the topological implications of $\mathcal{C}$. Since $\mathbb{S}^1$ is multiply connected, $\mathbb{R} \times \mathbb{S}^1$ and $\mathbb{R}^2 \times \mathbb{S}^1$ are multiply connected. It is difficult to visualize $\mathcal{C}$ because it is a 3D manifold; however, there is a nice interpretation using identification. Start with the open unit cube, $(0,1)^3 \subset \mathbb{R}^3$. Include the boundary points of the form $(x, y, 0)$ and $(x, y, 1)$, and make the identification $(x, y, 0) \sim (x, y, 1)$ for all $x, y \in (0, 1)$. This means that when traveling in the x and y directions, there is a "frontier" to the C-space; however, traveling in the z direction causes a wraparound.

It is very important for a motion planning algorithm to understand that this wraparound exists. For example, consider $\mathbb{R} \times \mathbb{S}^1$ because it is easier to visualize. Imagine a path planning problem for which $\mathcal{C} = \mathbb{R} \times \mathbb{S}^1$, as depicted in Figure 4.8. Suppose the top and bottom are identified to make a cylinder, and there is an obstacle across the middle. Suppose the task is to find a path from q_I to q_G. If the top and bottom were not identified, then it would not be possible to connect q_I to q_G; however, if the algorithm realizes it was given a cylinder, the task is straightforward. In general, it is very important to understand the topology of $\mathcal{C}$; otherwise, potential solutions will be lost.

The next section addresses $SE(n)$ for $n = 3$. The main difficulty is determining the topology of $SO(3)$. At least we do not have to consider $n > 3$ in this book.

4.2.2 3D Rigid Bodies: $SE(3)$

One might expect that defining $\mathcal{C}$ for a 3D rigid body is an obvious extension of the 2D case; however, 3D rotations are significantly more complicated. The resulting

C-space will be a six-dimensional manifold, $\mathcal{C} = \mathbb{R}^3 \times \mathbb{RP}^3$. Three dimensions come from translation and three more come from rotation.

The main quest in this section is to determine the topology of $SO(3)$. In Section 3.2.3, yaw, pitch, and roll were used to generate rotation matrices. These angles are convenient for visualization, performing transformations in software, and also for deriving the DH parameters. However, these were concerned with applying a single rotation, whereas the current problem is to characterize the set of all rotations. It is possible to use α, β, and γ to parameterize the set of rotations, but it causes serious troubles. There are some cases in which nonzero angles yield the identity rotation matrix, which is equivalent to $\alpha = \beta = \gamma = 0$. There are also cases in which a continuum of values for yaw, pitch, and roll angles yield the same rotation matrix. These problems destroy the topology, which causes both theoretical and practical difficulties in motion planning.

Consider applying the matrix group concepts from Section 4.2.1. The general linear group $GL(3)$ is homeomorphic to $\mathbb{R}^9$. The orthogonal group, $O(3)$, is determined by imposing the constraint $AA^T = I$. There are $\binom{3}{2} = 3$ independent equations that require distinct columns to be orthogonal, and three independent equations that force the magnitude of each column to be 1. This means that $O(3)$ has three dimensions, which matches our intuition since there were three rotation parameters in Section 3.2.3. To obtain $SO(3)$, the last constraint, $\det A = 1$, is added. Recall from Example 4.12 that $SO(2)$ consists of two circles, and the constraint $\det A = 1$ selects one of them. In the case of $O(3)$, there are two three-spheres, $\mathbb{S}^3 \sqcup \mathbb{S}^3$, and $\det A = 1$ selects one of them. However, there is one additional complication: Antipodal points on these spheres generate the same rotation matrix. This will be seen shortly when quaternions are used to parameterize $SO(3)$.

Using complex numbers to represent SO(2) Before introducing quaternions to parameterize 3D rotations, consider using complex numbers to parameterize 2D rotations. Let the term *unit complex number* refer to any complex number, $a + bi$, for which $a^2 + b^2 = 1$.

The set of all unit complex numbers forms a group under multiplication. It will be seen that it is "the same" group as $SO(2)$. This idea needs to be made more precise. Two groups, G and H, are considered "the same" if they are *isomorphic*, which means that there exists a bijective function $f : G \to H$ such that for all $a, b \in G$, $f(a) \circ f(b) = f(a \circ b)$. This means that we can perform some calculations in G, map the result to H, perform more calculations, and map back to G without any trouble. The sets G and H are just two alternative ways to express the same group.

The unit complex numbers and $SO(2)$ are isomorphic. To see this clearly, recall that complex numbers can be represented in polar form as $re^{i\theta}$; a unit complex number is simply $e^{i\theta}$. A bijective mapping can be made between 2D rotation matrices and unit complex numbers by letting $e^{i\theta}$ correspond to the rotation matrix (3.31).

If complex numbers are used to represent rotations, it is important that they behave algebraically in the same way. If two rotations are combined, the matrices are multiplied. The equivalent operation is multiplication of complex numbers. Suppose that a 2D robot is rotated by θ_1, followed by θ_2. In polar form, the complex numbers are multiplied to yield $e^{i\theta_1}e^{i\theta_2} = e^{i(\theta_1+\theta_2)}$, which clearly represents a rotation of $\theta_1 + \theta_2$. If the unit complex number is represented in Cartesian form, then the rotations corresponding to $a_1 + b_1 i$ and $a_2 + b_2 i$ are combined to obtain $(a_1a_2 - b_1b_2) + (a_1b_2 + a_2b_1)i$. Note that here we have not used complex numbers to express the solution to a polynomial equation, which is their more popular use; we simply borrowed their nice algebraic properties. At any time, a complex number $a + bi$ can be converted into the equivalent rotation matrix

$$R(a, b) = \begin{pmatrix} a & -b \\ b & a \end{pmatrix}. \tag{4.18}$$

Recall that only one independent parameter needs to be specified because $a^2 + b^2 = 1$. Hence, it appears that the set of unit complex numbers is the same manifold as $SO(2)$, which is the circle $\mathbb{S}^1$ (recall, that "same" means in the sense of homeomorphism).

Quaternions The manner in which complex numbers were used to represent 2D rotations will now be adapted to using quaternions to represent 3D rotations. Let $\mathbb{H}$ represent the set of *quaternions*, in which each quaternion, $h \in \mathbb{H}$, is represented as $h = a + bi + cj + dk$, and $a, b, c, d \in \mathbb{R}$. A quaternion can be considered as a four-dimensional vector. The symbols i, j, and k are used to denote three "imaginary" components of the quaternion. The following relationships are defined: $i^2 = j^2 = k^2 = ijk = -1$, from which it follows that $ij = k$, $jk = i$, and $ki = j$. Using these, multiplication of two quaternions, $h_1 = a_1 + b_1 i + c_1 j + d_1 k$ and $h_2 = a_2 + b_2 i + c_2 j + d_2 k$, can be derived to obtain $h_1 \cdot h_2 = a_3 + b_3 i + c_3 j + d_3 k$, in which

$$\begin{aligned} a_3 &= a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2 \\ b_3 &= a_1b_2 + a_2b_1 + c_1d_2 - c_2d_1 \\ c_3 &= a_1c_2 + a_2c_1 + b_2d_1 - b_1d_2 \\ d_3 &= a_1d_2 + a_2d_1 + b_1c_2 - b_2c_1. \end{aligned} \tag{4.19}$$

Using this operation, it can be shown that $\mathbb{H}$ is a group with respect to quaternion multiplication. Note, however, that the multiplication is not commutative! This is also true of 3D rotations; there must be a good reason.

For convenience, quaternion multiplication can be expressed in terms of vector multiplications, a dot product, and a cross product. Let $v = [b \;\; c \;\; d]$ be a three-dimensional vector that represents the final three quaternion components. The first component of $h_1 \cdot h_2$ is $a_1a_2 - v_1 \cdot v_2$. The final three components are given by the three-dimensional vector $a_1v_2 + a_2v_1 + v_1 \times v_2$.

In the same way that *unit* complex numbers were needed for $SO(2)$, *unit quaternions* are needed for $SO(3)$, which means that $\mathbb{H}$ is restricted to quaternions for

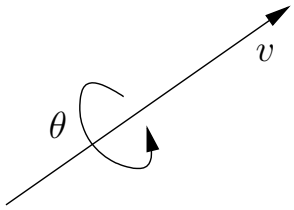


Figure 4.9: Any 3D rotation can be considered as a rotation by an angle θ about the axis given by the unit direction vector $v = [v_1 \;\; v_2 \;\; v_3]$.

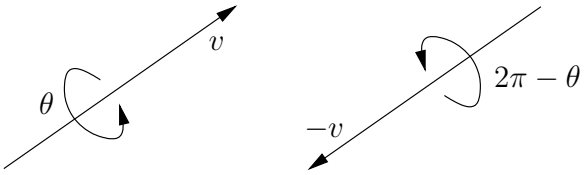


Figure 4.10: There are two ways to encode the same rotation.

which $a^2 + b^2 + c^2 + d^2 = 1$. Note that this forms a subgroup because the multiplication of unit quaternions yields a unit quaternion, and the other group axioms hold.

The next step is to describe a mapping from unit quaternions to $SO(3)$. Let the unit quaternion $h = a + bi + cj + dk$ map to the matrix

$$R(h) = \begin{pmatrix} 2(a^2+b^2)-1 & 2(bc-ad) & 2(bd+ac) \\ 2(bc+ad) & 2(a^2+c^2)-1 & 2(cd-ab) \\ 2(bd-ac) & 2(cd+ab) & 2(a^2+d^2)-1 \end{pmatrix}, \tag{4.20}$$

which can be verified as orthogonal and $\det R(h) = 1$. Therefore, it belongs to $SO(3)$. It is not shown here, but it conveniently turns out that h represents the rotation shown in Figure 4.9, by making the assignment

$$h = \cos\frac{\theta}{2} + \left(v_1 \sin\frac{\theta}{2}\right) i + \left(v_2 \sin\frac{\theta}{2}\right) j + \left(v_3 \sin\frac{\theta}{2}\right) k. \tag{4.21}$$

Unfortunately, this representation is not unique. It can be verified in (4.20) that $R(h) = R(-h)$. A nice geometric interpretation is given in Figure 4.10. The quaternions h and $-h$ represent the same rotation because a rotation of θ about the direction v is equivalent to a rotation of $2\pi - \theta$ about the direction $-v$. Consider the quaternion representation of the second expression of rotation with respect to the first. The real part is

$$\cos\left(\frac{2\pi-\theta}{2}\right) = \cos\left(\pi - \frac{\theta}{2}\right) = -\cos\left(\frac{\theta}{2}\right) = -a. \tag{4.22}$$

The i, j, and k components are

$$-v\sin\left(\frac{2\pi-\theta}{2}\right) = -v\sin\left(\pi - \frac{\theta}{2}\right) = -v\sin\left(\frac{\theta}{2}\right) = [-b \;\; -c \;\; -d]. \tag{4.23}$$

The quaternion $-h$ has been constructed. Thus, h and $-h$ represent the same rotation. Luckily, this is the only problem, and the mapping given by (4.20) is two-to-one from the set of unit quaternions to $SO(3)$.

This can be fixed by the identification trick. Note that the set of unit quaternions is homeomorphic to $\mathbb{S}^3$ because of the constraint $a^2 + b^2 + c^2 + d^2 = 1$. The algebraic properties of quaternions are not relevant at this point. Just imagine each h as an element of $\mathbb{R}^4$, and the constraint $a^2 + b^2 + c^2 + d^2 = 1$ forces the points to lie on $\mathbb{S}^3$. Using identification, declare $h \sim -h$ for all unit quaternions. This means that the antipodal points of $\mathbb{S}^3$ are identified. Recall from the end of Section 4.1.2 that when antipodal points are identified, $\mathbb{RP}^n \cong \mathbb{S}^n / \sim$. Hence, $SO(3) \cong \mathbb{RP}^3$, which can be considered as the set of all lines through the origin of $\mathbb{R}^4$, but this is hard to visualize. The representation of $\mathbb{RP}^2$ in Figure 4.5 can be extended to $\mathbb{RP}^3$. Start with $(0,1)^3 \subset \mathbb{R}^3$, and make three different kinds of identifications, one for each pair of opposite cube faces, and add all of the points to the manifold. For each kind of identification a twist needs to be made (without the twist, $\mathbb{T}^3$ would be obtained). For example, in the z direction, let $(x, y, 0) \sim (1 - x, 1 - y, 1)$ for all $x, y \in [0, 1]$.

One way to force uniqueness of rotations is to require staying in the "upper half" of $\mathbb{S}^3$. For example, require that $a \geq 0$, as long as the boundary case of $a = 0$ is handled properly because of antipodal points at the equator of $\mathbb{S}^3$. If $a = 0$, then require that $b \geq 0$. However, if $a = b = 0$, then require that $c \geq 0$ because points such as $(0, 0, -1, 0)$ and $(0, 0, 1, 0)$ are the same rotation. Finally, if $a = b = c = 0$, then only $d = 1$ is allowed. If such restrictions are made, it is important, however, to remember the connectivity of $\mathbb{RP}^3$. If a path travels across the equator of $\mathbb{S}^3$, it must be mapped to the appropriate place in the "northern hemisphere." At the instant it hits the equator, it must move to the antipodal point. These concepts are much easier to visualize if you remove a dimension and imagine them for $\mathbb{S}^2 \subset \mathbb{R}^3$, as described at the end of Section 4.1.2.

Using quaternion multiplication The representation of rotations boiled down to picking points on $\mathbb{S}^3$ and respecting the fact that antipodal points give the same element of $SO(3)$. In a sense, this has nothing to do with the algebraic properties of quaternions. It merely means that $SO(3)$ can be parameterized by picking points in $\mathbb{S}^3$, just like $SO(2)$ was parameterized by picking points in $\mathbb{S}^1$ (ignoring the antipodal identification problem for $SO(3)$).

However, one important reason why the quaternion arithmetic was introduced is that the group of unit quaternions with h and $-h$ identified is also isomorphic to $SO(3)$. This means that a sequence of rotations can be multiplied together using quaternion multiplication instead of matrix multiplication. This is important because fewer operations are required for quaternion multiplication in comparison to matrix multiplication. At any point, (4.20) can be used to convert the result back into a matrix; however, this is not even necessary. It turns out that a point in the world, $(x, y, z) \in \mathbb{R}^3$, can be transformed by directly using quaternion arithmetic. An analog to the complex conjugate from complex numbers is needed.

For any $h = a + bi + cj + dk \in \mathbb{H}$, let $h^* = a - bi - cj - dk$ be its *conjugate*. For any point $(x, y, z) \in \mathbb{R}^3$, let $p \in \mathbb{H}$ be the quaternion $0 + xi + yj + zk$. It can be shown (with a lot of algebra) that the rotated point (x, y, z) is given by $h \cdot p \cdot h^*$. The i, j, k components of the resulting quaternion are new coordinates for the transformed point. It is equivalent to having transformed (x, y, z) with the matrix $R(h)$.

Finding quaternion parameters from a rotation matrix Recall from Section 3.2.3 that given a rotation matrix (3.43), the yaw, pitch, and roll parameters could be directly determined using the atan2 function. It turns out that the quaternion representation can also be determined directly from the matrix. This is the inverse of the function in (4.20).[9]

For a given rotation matrix (3.43), the quaternion parameters $h = a+bi+cj+dk$ can be computed as follows [210]. The first component is

$$a = \tfrac{1}{2}\sqrt{r_{11} + r_{22} + r_{33} + 1}, \tag{4.24}$$

and if $a \neq 0$, then

$$b = \frac{r_{32} - r_{23}}{4a}, \tag{4.25}$$

$$c = \frac{r_{13} - r_{31}}{4a}, \tag{4.26}$$

and

$$d = \frac{r_{21} - r_{12}}{4a}. \tag{4.27}$$

If $a = 0$, then the previously mentioned equator problem occurs. In this case,

$$b = \frac{r_{13}r_{12}}{\sqrt{r_{12}^2 r_{13}^2 + r_{12}^2 r_{23}^2 + r_{13}^2 r_{23}^2}}, \tag{4.28}$$

$$c = \frac{r_{12}r_{23}}{\sqrt{r_{12}^2 r_{13}^2 + r_{12}^2 r_{23}^2 + r_{13}^2 r_{23}^2}}, \tag{4.29}$$

and

$$d = \frac{r_{13}r_{23}}{\sqrt{r_{12}^2 r_{13}^2 + r_{12}^2 r_{23}^2 + r_{13}^2 r_{23}^2}}. \tag{4.30}$$

This method fails if $r_{12} = r_{23} = 0$ or $r_{13} = r_{23} = 0$ or $r_{12} = r_{23} = 0$. These correspond precisely to the cases in which the rotation matrix is a yaw, (3.39), pitch, (3.40), or roll, (3.41), which can be detected in advance.

[9] Since that function was two-to-one, it is technically not an inverse until the quaternions are restricted to the upper hemisphere, as described previously.

Special Euclidean group Now that the complicated part of representing $SO(3)$ has been handled, the representation of $SE(3)$ is straightforward. The general form of a matrix in $SE(3)$ is given by (4.16), in which $R \in SO(3)$ and $v \in \mathbb{R}^3$. Since $SO(3) \cong \mathbb{RP}^3$, and translations can be chosen independently, the resulting C-space for a rigid body that rotates and translates in $\mathbb{R}^3$ is

$$\mathcal{C} = \mathbb{R}^3 \times \mathbb{RP}^3, \tag{4.31}$$

which is a six-dimensional manifold. As expected, the dimension of $\mathcal{C}$ is exactly the number of degrees of freedom of a free-floating body in space.

4.2.3 Chains and Trees of Bodies

If there are multiple bodies that are allowed to move independently, then their C-spaces can be combined using Cartesian products. Let $\mathcal{C}_i$ denote the C-space of $\mathcal{A}_i$. If there are n free-floating bodies in $\mathcal{W} = \mathbb{R}^2$ or $\mathcal{W} = \mathbb{R}^3$, then

$$\mathcal{C} = \mathcal{C}_1 \times \mathcal{C}_2 \times \cdots \times \mathcal{C}_n. \tag{4.32}$$

If the bodies are attached to form a kinematic chain or kinematic tree, then each C-space must be considered on a case-by-case basis. There is no general rule that simplifies the process. One thing to generally be careful about is that the full range of motion might not be possible for typical joints. For example, a revolute joint might not be able to swing all of the way around to enable any $\theta \in [0, 2\pi)$. If θ cannot wind around $\mathbb{S}^1$, then the C-space for this joint is homeomorphic to $\mathbb{R}$ instead of $\mathbb{S}^1$. A similar situation occurs for a spherical joint. A typical ball joint cannot achieve any orientation in $SO(3)$ due to mechanical obstructions. In this case, the C-space is not $\mathbb{RP}^3$ because part of $SO(3)$ is missing.

Another complication is that the DH parameterization of Section 3.3.2 is designed to facilitate the assignment of coordinate frames and computation of transformations, but it neglects considerations of topology. For example, a common approach to representing a spherical robot wrist is to make three zero-length links that each behave as a revolute joint. If the range of motion is limited, this might not cause problems, but in general the problems would be similar to using yaw, pitch, and roll to represent $SO(3)$. There may be multiple ways to express the same arm configuration.

Several examples are given below to help in determining C-spaces for chains and trees of bodies. Suppose $\mathcal{W} = \mathbb{R}^2$, and there is a chain of n bodies that are attached by revolute joints. Suppose that the first joint is capable of rotation only about a fixed point (e.g., it spins around a nail). If each joint has the full range of motion $\theta_i \in [0, 2\pi)$, the C-space is

$$\mathcal{C} = \mathbb{S}^1 \times \mathbb{S}^1 \times \cdots \times \mathbb{S}^1 = \mathbb{T}^n. \tag{4.33}$$

However, if each joint is restricted to $\theta_i \in (-\pi/2, \pi/2)$, then $\mathcal{C} = \mathbb{R}^n$. If any transformation in $SE(2)$ can be applied to $\mathcal{A}_1$, then an additional $\mathbb{R}^2$ is needed.

In the case of restricted joint motions, this yields $\mathbb{R}^{n+2}$. If the joints can achieve any orientation, then $\mathcal{C} = \mathbb{R}^2 \times \mathbb{T}^n$. If there are prismatic joints, then each joint contributes $\mathbb{R}$ to the C-space.

Recall from Figure 3.12 that for $\mathcal{W} = \mathbb{R}^3$ there are six different kinds of joints. The cases of revolute and prismatic joints behave the same as for $\mathcal{W} = \mathbb{R}^2$. Each screw joint contributes $\mathbb{R}$. A cylindrical joint contributes $\mathbb{R} \times \mathbb{S}^1$, unless its rotational motion is restricted. A planar joint contributes $\mathbb{R}^2 \times \mathbb{S}^1$ because any transformation in $SE(2)$ is possible. If its rotational motions are restricted, then it contributes $\mathbb{R}^3$. Finally, a spherical joint can theoretically contribute $\mathbb{RP}^3$. In practice, however, this rarely occurs. It is more likely to contribute $\mathbb{R}^2 \times \mathbb{S}^1$ or $\mathbb{R}^3$ after restrictions are imposed. Note that if the first joint is a free-floating body, then it contributes $\mathbb{R}^3 \times \mathbb{RP}^3$.

Kinematic trees can be handled in the same way as kinematic chains. One issue that has not been mentioned is that there might be collisions between the links. This has been ignored up to this point, but obviously this imposes very complicated restrictions. The concepts from Section 4.3 can be applied to handle this case and the placement of additional obstacles in $\mathcal{W}$. Reasoning about these kinds of restrictions and the path connectivity of the resulting space is indeed the main point of motion planning.

4.3 Configuration Space Obstacles

Section 4.2 defined $\mathcal{C}$, the manifold of robot transformations in the absence of any collision constraints. The current section removes from $\mathcal{C}$ the configurations that either cause the robot to collide with obstacles or cause some specified links of the robot to collide with each other. The removed part of $\mathcal{C}$ is referred to as the obstacle region. The leftover space is precisely what a solution path must traverse. A motion planning algorithm must find a path in the leftover space from an initial configuration to a goal configuration. Finally, after the models of Chapter 3 and the previous sections of this chapter, the motion planning problem can be precisely described.

4.3.1 Definition of the Basic Motion Planning Problem

Obstacle region for a rigid body Suppose that the world, $\mathcal{W} = \mathbb{R}^2$ or $\mathcal{W} = \mathbb{R}^3$, contains an obstacle region, $\mathcal{O} \subset \mathcal{W}$. Assume here that a rigid robot, $\mathcal{A} \subset \mathcal{W}$, is defined; the case of multiple links will be handled shortly. Assume that both $\mathcal{A}$ and $\mathcal{O}$ are expressed as semi-algebraic models (which includes polygonal and polyhedral models) from Section 3.1. Let $q \in \mathcal{C}$ denote the *configuration* of $\mathcal{A}$, in which $q = (x_t, y_t, \theta)$ for $\mathcal{W} = \mathbb{R}^2$ and $q = (x_t, y_t, z_t, h)$ for $\mathcal{W} = \mathbb{R}^3$ (h represents the unit quaternion).

The *obstacle region*, $\mathcal{C}_{obs} \subseteq \mathcal{C}$, is defined as

$$\mathcal{C}_{obs} = \{q \in \mathcal{C} \mid \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\}, \tag{4.34}$$

which is the set of all configurations, q, at which $\mathcal{A}(q)$, the transformed robot, intersects the obstacle region, $\mathcal{O}$. Since $\mathcal{O}$ and $\mathcal{A}(q)$ are closed sets in $\mathcal{W}$, the obstacle region is a closed set in $\mathcal{C}$.

The leftover configurations are called the *free space*, which is defined and denoted as $\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}$. Since $\mathcal{C}$ is a topological space and $\mathcal{C}_{obs}$ is closed, $\mathcal{C}_{free}$ must be an open set. This implies that the robot can come arbitrarily close to the obstacles while remaining in $\mathcal{C}_{free}$. If $\mathcal{A}$ "touches" $\mathcal{O}$,

$$\text{int}(\mathcal{O}) \cap \text{int}(\mathcal{A}(q)) = \emptyset \text{ and } \mathcal{O} \cap \mathcal{A}(q) \neq \emptyset, \tag{4.35}$$

then $q \in \mathcal{C}_{obs}$ (recall that int means the interior). The condition above indicates that only their boundaries intersect.

The idea of getting arbitrarily close may be nonsense in practical robotics, but it makes a clean formulation of the motion planning problem. Since $\mathcal{C}_{free}$ is open, it becomes impossible to formulate some optimization problems, such as finding the shortest path. In this case, the closure, $\text{cl}(\mathcal{C}_{free})$, should instead be used, as described in Section 7.7.

Obstacle region for multiple bodies If the robot consists of multiple bodies, the situation is more complicated. The definition in (4.34) only implies that the robot does not collide with the obstacles; however, if the robot consists of multiple bodies, then it might also be appropriate to avoid collisions between different links of the robot. Let the robot be modeled as a collection, $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m\}$, of m links, which may or may not be attached together by joints. A single configuration vector q is given for the entire collection of links. We will write $\mathcal{A}_i(q)$ for each link, i, even though some of the parameters of q may be irrelevant for moving link $\mathcal{A}_i$. For example, in a kinematic chain, the configuration of the second body does not depend on the angle between the ninth and tenth bodies.

Let P denote the set of *collision pairs*, in which each collision pair, $(i, j) \in P$, represents a pair of link indices $i, j \in \{1, 2, \ldots, m\}$, such that $i \neq j$. If (i, j) appears in P, it means that A_i and A_j are not allowed to be in a configuration, q, for which $\mathcal{A}_i(q) \cap \mathcal{A}_j(q) \neq \emptyset$. Usually, P does not represent all pairs because consecutive links are in contact all of the time due to the joint that connects them. One common definition for P is that each link must avoid collisions with any links to which it is not attached by a joint. For m bodies, P is generally of size $O(m^2)$; however, in practice it is often possible to eliminate many pairs by some geometric analysis of the linkage. Collisions between some pairs of links may be impossible over all of $\mathcal{C}$, in which case they do not need to appear in P.

Using P, the consideration of robot self-collisions is added to the definition of $\mathcal{C}_{obs}$ to obtain

$$\mathcal{C}_{obs} = \left(\bigcup_{i=1}^{m} \{q \in \mathcal{C} \mid \mathcal{A}_i(q) \cap \mathcal{O} \neq \emptyset\} \right) \bigcup \left(\bigcup_{[i,j] \in P} \{q \in \mathcal{C} \mid \mathcal{A}_i(q) \cap \mathcal{A}_j(q) \neq \emptyset\} \right). \tag{4.36}$$

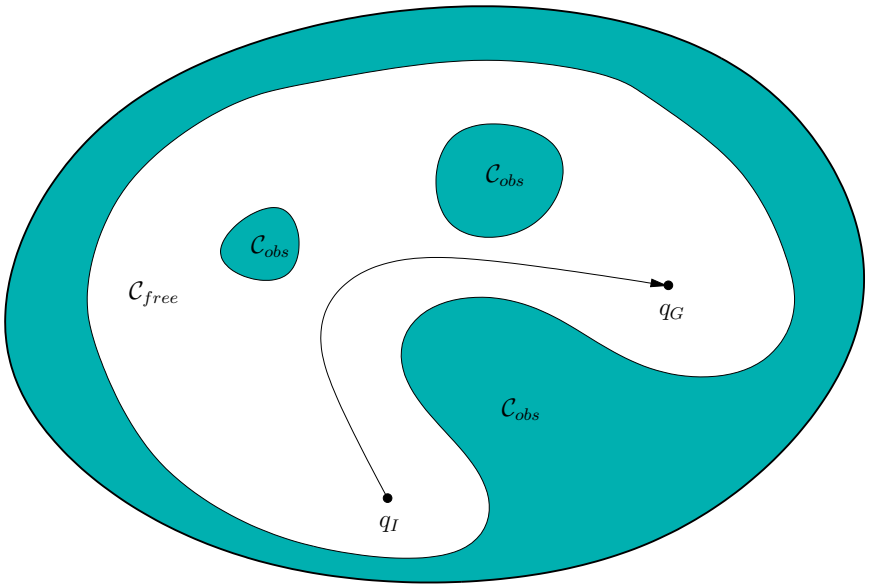


Figure 4.11: The basic motion planning problem is conceptually very simple using C-space ideas. The task is to find a path from q_I to q_G in $\mathcal{C}_{free}$. The entire blob represents $\mathcal{C} = \mathcal{C}_{free} \cup \mathcal{C}_{obs}$.

Thus, a configuration $q \in \mathcal{C}$ is in $\mathcal{C}_{obs}$ if at least one link collides with $\mathcal{O}$ or a pair of links indicated by P collide with each other.

Definition of basic motion planning Finally, enough tools have been introduced to precisely define the motion planning problem. The problem is conceptually illustrated in Figure 4.11. The main difficulty is that it is neither straightforward nor efficient to construct an explicit boundary or solid representation of either $\mathcal{C}_{free}$ or $\mathcal{C}_{obs}$. The components are as follows:

Formulation 4.1 (The Piano Mover's Problem)

1. A *world* $\mathcal{W}$ in which either $\mathcal{W} = \mathbb{R}^2$ or $\mathcal{W} = \mathbb{R}^3$.

2. A semi-algebraic *obstacle region* $\mathcal{O} \subset \mathcal{W}$ in the world.

3. A semi-algebraic *robot* is defined in $\mathcal{W}$. It may be a rigid robot $\mathcal{A}$ or a collection of m links, $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m$.

4. The *configuration space* $\mathcal{C}$ determined by specifying the set of all possible transformations that may be applied to the robot. From this, $\mathcal{C}_{obs}$ and $\mathcal{C}_{free}$ are derived.

5. A configuration, $q_I \in \mathcal{C}_{free}$ designated as the *initial configuration*.

6. A configuration $q_G \in \mathcal{C}_{free}$ designated as the *goal configuration*. The initial and goal configurations together are often called a *query pair* (or *query*) and designated as (q_I, q_G).

7. A complete algorithm must compute a (continuous) *path*, $\tau : [0,1] \rightarrow \mathcal{C}_{free}$, such that $\tau(0) = q_I$ and $\tau(1) = q_G$, or correctly report that such a path does not exist.

It was shown by Reif [817] that this problem is PSPACE-hard, which implies NP-hard. The main problem is that the dimension of $\mathcal{C}$ is unbounded.

4.3.2 Explicitly Modeling $\mathcal{C}_{obs}$: The Translational Case

It is important to understand how to construct a representation of $\mathcal{C}_{obs}$. In some algorithms, especially the combinatorial methods of Chapter 6, this represents an important first step to solving the problem. In other algorithms, especially the sampling-based planning algorithms of Chapter 5, it helps to understand why such constructions are avoided due to their complexity.

The simplest case for characterizing $\mathcal{C}_{obs}$ is when $\mathcal{C} = \mathbb{R}^n$ for $n = 1$, 2, and 3, and the robot is a rigid body that is restricted to translation only. Under these conditions, $\mathcal{C}_{obs}$ can be expressed as a type of convolution. For any two sets $X, Y \subset \mathbb{R}^n$, let their *Minkowski difference*[10] be defined as

$$X \ominus Y = \{x - y \in \mathbb{R}^n \mid x \in X \text{ and } y \in Y\}, \tag{4.37}$$

in which $x - y$ is just vector subtraction on $\mathbb{R}^n$. The Minkowski difference between X and Y can also be considered as the Minkowski sum of X and $-Y$. The *Minkowski sum* $\oplus$ is obtained by simply adding elements of X and Y in (4.37), as opposed to subtracting them. The set $-Y$ is obtained by replacing each $y \in Y$ by $-y$.

In terms of the Minkowski difference, $\mathcal{C}_{obs} = \mathcal{O} \ominus \mathcal{A}(0)$. To see this, it is helpful to consider a one-dimensional example.

Example 4.13 (One-Dimensional C-Space Obstacle) In Figure 4.12, both the robot $\mathcal{A} = [-1, 2]$ and obstacle region $\mathcal{O} = [0, 4]$ are intervals in a one-dimensional world, $\mathcal{W} = \mathbb{R}$. The negation, $-\mathcal{A}$, of the robot is shown as the interval $[-2, 1]$. Finally, by applying the Minkowski sum to $\mathcal{O}$ and $-\mathcal{A}$, the C-space obstacle, $\mathcal{C}_{obs} = [-2, 5]$, is obtained. ■

The Minkowski difference is often considered as a *convolution*. It can even be defined to appear the same as studied in differential equations and system theory.

[10] In some contexts, which include mathematics and image processing, the Minkowski difference or *Minkowski subtraction* is defined differently (instead, it is a kind of "erosion"). For this reason, some authors prefer to define all operations in terms of the Minkowski sum, $\oplus$, which is consistently defined in all contexts. Following this convention, we would define $X \oplus (-Y)$, which is equivalent to $X \ominus Y$.

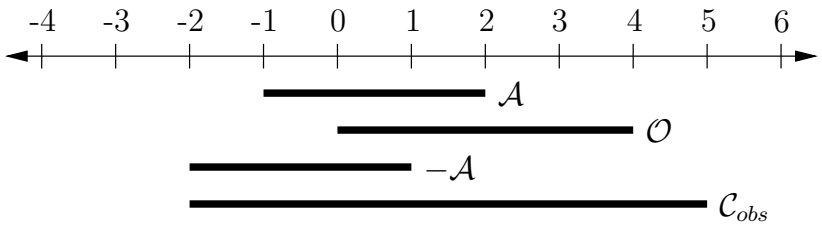


Figure 4.12: A one-dimensional C-space obstacle.

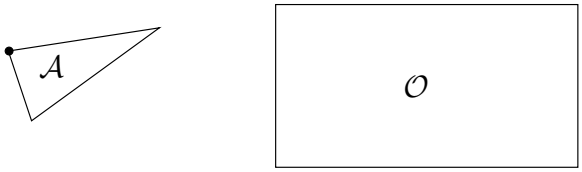


Figure 4.13: A triangular robot and a rectangular obstacle.

For a one-dimensional example, let $f : \mathbb{R} \rightarrow \{0, 1\}$ be a function such that $f(x) = 1$ if and only if $x \in \mathcal{O}$. Similarly, let $g : \mathbb{R} \rightarrow \{0, 1\}$ be a function such that $g(x) = 1$ if and only if $x \in \mathcal{A}$. The convolution

$$h(x) = \int_{-\infty}^{\infty} f(\tau) g(x - \tau) d\tau, \tag{4.38}$$

yields a function h, for which $h(x) > 0$ if $x \in \text{int}(\mathcal{C}_{obs})$, and $h(x) = 0$ otherwise.

A polygonal C-space obstacle A simple algorithm for computing $\mathcal{C}_{obs}$ exists in the case of a 2D world that contains a convex polygonal obstacle $\mathcal{O}$ and a convex polygonal robot $\mathcal{A}$ [657]. This is often called the *star algorithm*. For this problem, $\mathcal{C}_{obs}$ is also a convex polygon. Recall that nonconvex obstacles and robots can be modeled as the union of convex parts. The concepts discussed below can also be applied in the nonconvex case by considering $\mathcal{C}_{obs}$ as the union of convex components, each of which corresponds to a convex component of $\mathcal{A}$ colliding with a convex component of $\mathcal{O}$.

The method is based on sorting normals to the edges of the polygons on the basis of angles. The key observation is that every edge of $\mathcal{C}_{obs}$ is a translated edge from either $\mathcal{A}$ or $\mathcal{O}$. In fact, every edge from $\mathcal{O}$ and $\mathcal{A}$ is used exactly once in the construction of $\mathcal{C}_{obs}$. The only problem is to determine the ordering of these edges of $\mathcal{C}_{obs}$. Let $\alpha_1, \alpha_2, \ldots, \alpha_n$ denote the angles of the inward edge normals in counterclockwise order around $\mathcal{A}$. Let $\beta_1, \beta_2, \ldots, \beta_n$ denote the outward edge normals to $\mathcal{O}$. After sorting both sets of angles in circular order around $\mathbb{S}^1$, $\mathcal{C}_{obs}$ can be constructed incrementally by using the edges that correspond to the sorted normals, in the order in which they are encountered.

Example 4.14 (A Triangular Robot and Rectangular Obstacle) To gain an understanding of the method, consider the case of a triangular robot and a rectangular obstacle, as shown in Figure 4.13. The black dot on $\mathcal{A}$ denotes the origin

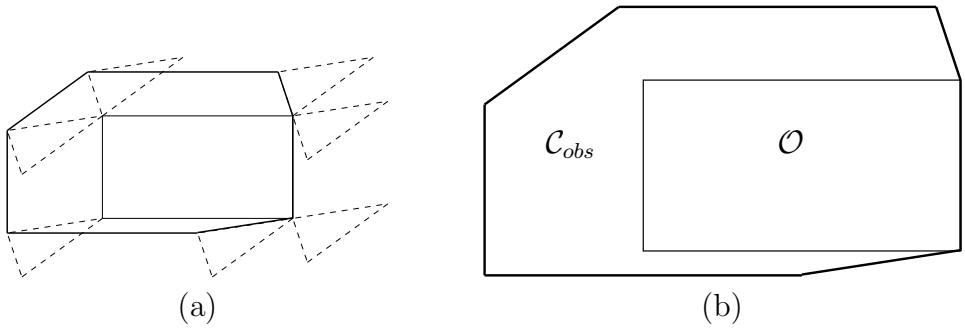


Figure 4.14: (a) Slide the robot around the obstacle while keeping them both in contact. (b) The edges traced out by the origin of $\mathcal{A}$ form $\mathcal{C}_{obs}$.

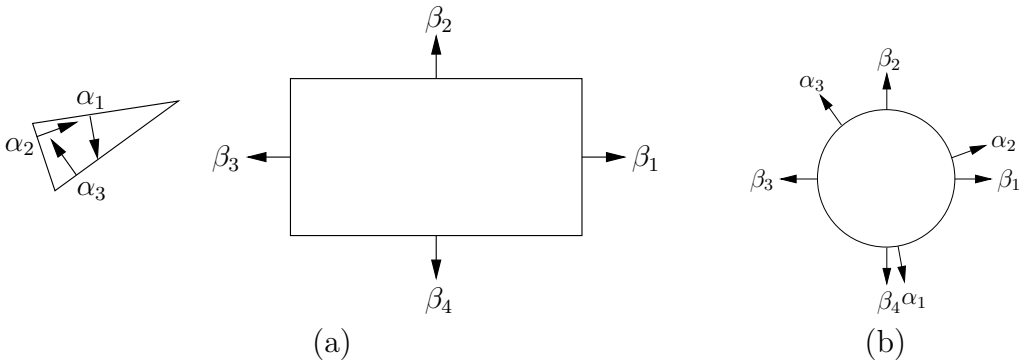


Figure 4.15: (a) Take the inward edge normals of $\mathcal{A}$ and the outward edge normals of $\mathcal{O}$. (b) Sort the edge normals around $\mathbb{S}^1$. This gives the order of edges in $\mathcal{C}_{obs}$.

of its body frame. Consider sliding the robot around the obstacle in such a way that they are always in contact, as shown in Figure 4.14a. This corresponds to the traversal of all of the configurations in $\partial\mathcal{C}_{obs}$ (the boundary of $\mathcal{C}_{obs}$). The origin of $\mathcal{A}$ traces out the edges of $\mathcal{C}_{obs}$, as shown in Figure 4.14b. There are seven edges, and each edge corresponds to either an edge of $\mathcal{A}$ or an edge of $\mathcal{O}$. The directions of the normals are defined as shown in Figure 4.15a. When sorted as shown in Figure 4.15b, the edges of $\mathcal{C}_{obs}$ can be incrementally constructed. ■

The running time of the algorithm is $O(n + m)$, in which n is the number of edges defining $\mathcal{A}$, and m is the number of edges defining $\mathcal{O}$. Note that the angles can be sorted in linear time because they already appear in counterclockwise order around $\mathcal{A}$ and $\mathcal{O}$; they only need to be merged. If two edges are collinear, then they can be placed end-to-end as a single edge of $\mathcal{C}_{obs}$.

Computing the boundary of $\mathcal{C}_{obs}$ So far, the method quickly identifies each edge that contributes to $\mathcal{C}_{obs}$. It can also construct a solid representation of $\mathcal{C}_{obs}$ in terms of half-planes. This requires defining $n + m$ linear equations (assuming there are no collinear edges).

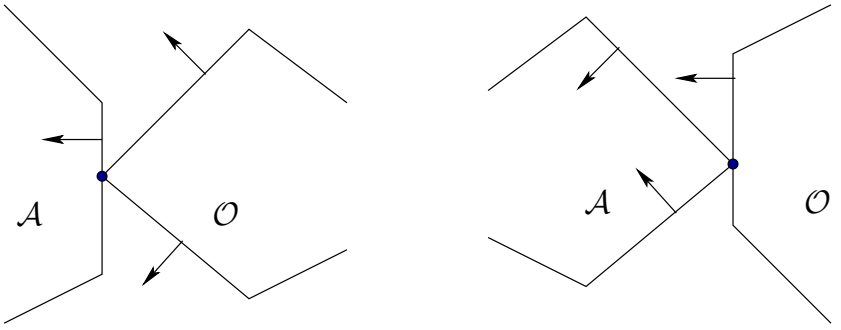


Figure 4.16: Two different types of contact, each of which generates a different kind of $\mathcal{C}_{obs}$ edge [280, 657].

There are two different ways in which an edge of $\mathcal{C}_{obs}$ is generated, as shown in Figure 4.16 [282, 657]. *Type EV* contact refers to the case in which an edge of $\mathcal{A}$ is in contact with a vertex of $\mathcal{O}$. Type EV contacts contribute to n edges of $\mathcal{C}_{obs}$, once for each edge of $\mathcal{A}$. *Type VE* contact refers to the case in which a vertex of $\mathcal{A}$ is in contact with an edge of $\mathcal{O}$. This contributes to m edges of $\mathcal{C}_{obs}$. The relationships between the edge normals are also shown in Figure 4.16. For Type EV, the inward edge normal points between the outward edge normals of the obstacle edges that share the contact vertex. Likewise for Type VE, the outward edge normal of $\mathcal{O}$ points between the inward edge normals of $\mathcal{A}$.

Using the ordering shown in Figure 4.15b, Type EV contacts occur precisely when an edge normal of $\mathcal{A}$ is encountered, and Type VE contacts occur when an

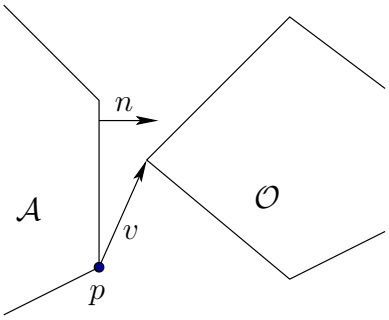


Figure 4.17: Contact occurs when n and v are perpendicular.

edge normal of $\mathcal{O}$ is encountered. The task is to determine the line equation for each occurrence. Consider the case of a Type EV contact; the Type VE contact can be handled in a similar manner. In addition to the constraint on the directions of the edge normals, the contact vertex of $\mathcal{O}$ must lie on the contact edge of $\mathcal{A}$. Recall that convex obstacles were constructed by the intersection of half-planes. Each edge of $\mathcal{C}_{obs}$ can be defined in terms of a supporting half-plane; hence, it is only necessary to determine whether the vertex of $\mathcal{O}$ lies on the line through the contact edge of $\mathcal{A}$. This condition occurs precisely as n and v are perpendicular, as shown in Figure 4.17, and yields the constraint $n \cdot v = 0$.

Note that the normal vector n does not depend on the configuration of $\mathcal{A}$ because the robot cannot rotate. The vector v, however, depends on the translation $q = (x_t, y_t)$ of the point p. Therefore, it is more appropriate to write the condition as $n \cdot v(x_t, y_t) = 0$. The transformation equations are linear for translation; hence, $n \cdot v(x_t, y_t) = 0$ is the equation of a line in $\mathcal{C}$. For example, if the coordinates of p are $(1, 2)$ for $\mathcal{A}(0, 0)$, then the expression for p at configuration (x_t, y_t) is $(1 + x_t, 2 + y_t)$. Let $f(x_t, y_t) = n \cdot v(x_t, y_t)$. Let $H = \{(x_t, y_t) \in \mathcal{C} \mid f(x_t, y_t) \leq 0\}$. Observe that any configurations not in H must lie in $\mathcal{C}_{free}$. The half-plane H is used to define one edge of $\mathcal{C}_{obs}$. The obstacle region $\mathcal{C}_{obs}$ can be completely characterized by intersecting the resulting half-planes for each of the Type EV and Type VE contacts. This yields a convex polygon in $\mathcal{C}$ that has $n + m$ sides, as expected.

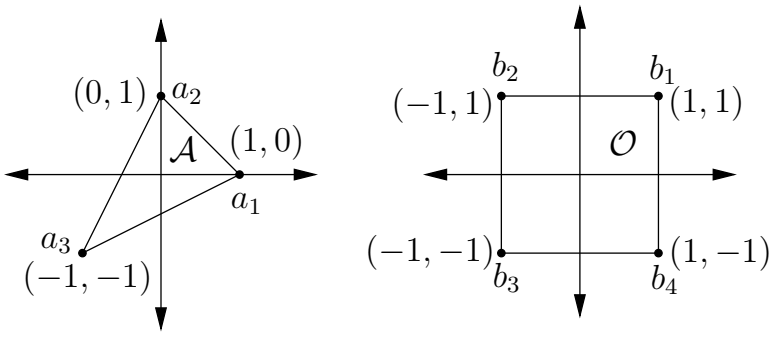


Figure 4.18: Consider constructing the obstacle region for this example.

Type	Vtx.	Edge	n	v	Half-Plane
VE	a_3	b_4-b_1	$[1, 0]$	$[x_t - 2, y_t]$	$\{q \in \mathcal{C} \mid x_t - 2 \leq 0\}$
VE	a_3	b_1-b_2	$[0, 1]$	$[x_t - 2, y_t - 2]$	$\{q \in \mathcal{C} \mid y_t - 2 \leq 0\}$
EV	b_2	a_3-a_1	[1,-2]	$[-x_t, 2 - y_t]$	$\{q \in \mathcal{C} \mid -x_t + 2y_t - 4 \leq 0\}$
VE	a_1	b_2-b_3	$[-1, 0]$	$[2 + x_t, y_t - 1]$	$\{q \in \mathcal{C} \mid -x_t - 2 \leq 0\}$
EV	b_3	a_1-a_2	$[1, 1]$	$[-1 - x_t, -y_t]$	$\{q \in \mathcal{C} \mid -x_t - y_t - 1 \leq 0\}$
VE	a_2	b_3-b_4	$[0, -1]$	$[x_t + 1, y_t + 2]$	$\{q \in \mathcal{C} \mid -y_t - 2 \leq 0\}$
EV	b_4	a_2-a_3	$[-2, 1]$	$[2 - x_t, -y_t]$	$\{q \in \mathcal{C} \mid 2x_t - y_t - 4 \leq 0\}$

Figure 4.19: The various contact conditions are shown in the order as the edge normals appear around $\mathbb{S}^1$ (using inward normals for $\mathcal{A}$ and outward normals for $\mathcal{O}$).

Example 4.15 (The Boundary of $\mathcal{C}_{obs}$) Consider building a geometric model of $\mathcal{C}_{obs}$ for the robot and obstacle shown in Figure 4.18. Suppose that the orientation of $\mathcal{A}$ is fixed as shown, and $\mathcal{C} = \mathbb{R}^2$. In this case, $\mathcal{C}_{obs}$ will be a convex polygon with seven sides. The contact conditions that occur are shown in Figure 4.19. The ordering as the normals appear around $\mathbb{S}^1$ (using inward edge normals for $\mathcal{A}$ and outward edge normals for $\mathcal{O}$). The $\mathcal{C}_{obs}$ edges and their corresponding contact types are shown in Figure 4.19. ■

A polyhedral C-space obstacle Most of the previous ideas generalize nicely for the case of a polyhedral robot that is capable of translation only in a 3D world that contains polyhedral obstacles. If $\mathcal{A}$ and $\mathcal{O}$ are convex polyhedra, the resulting $\mathcal{C}_{obs}$ is a convex polyhedron.

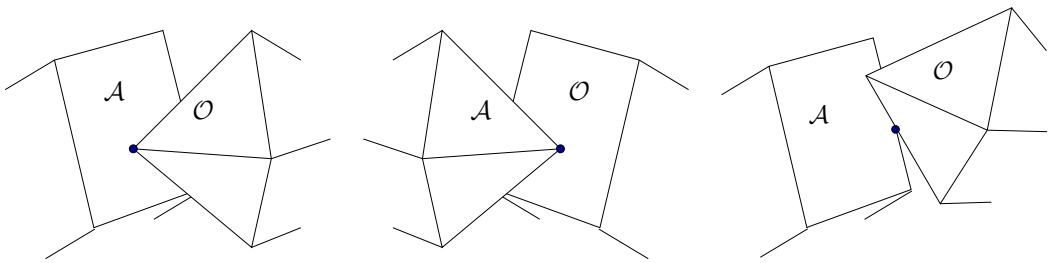


Figure 4.20: Three different types of contact, each of which generates a different kind of $\mathcal{C}_{obs}$ face.

There are three different kinds of contacts that each lead to half-spaces in $\mathcal{C}$:

1. **Type FV:** A face of $\mathcal{A}$ and a vertex of $\mathcal{O}$

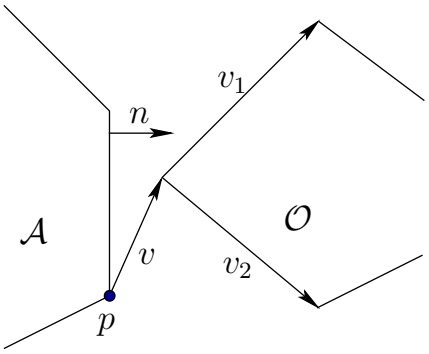


Figure 4.21: An illustration to help in constructing $\mathcal{C}_{obs}$ when rotation is allowed.

2. **Type VF:** A vertex of $\mathcal{A}$ and a face of $\mathcal{O}$

3. **Type EE:** An edge of $\mathcal{A}$ and an edge of $\mathcal{O}$.

These are shown in Figure 4.20. Each half-space defines a face of the polyhedron, $\mathcal{C}_{obs}$. The representation of $\mathcal{C}_{obs}$ can be constructed in $O(n+m+k)$ time, in which n is the number of faces of $\mathcal{A}$, m is the number of faces of $\mathcal{O}$, and k is the number of faces of $\mathcal{C}_{obs}$, which is at most nm [411].

4.3.3 Explicitly Modeling $\mathcal{C}_{obs}$: The General Case

Unfortunately, the cases in which $\mathcal{C}_{obs}$ is polygonal or polyhedral are quite limited. Most problems yield extremely complicated C-space obstacles. One good point is that $\mathcal{C}_{obs}$ can be expressed using semi-algebraic models, for any robots and obstacles defined using semi-algebraic models, even after applying any of the transformations from Sections 3.2 to 3.4. It might not be true, however, for other kinds of transformations, such as warping a flexible material [32, 577].

Consider the case of a convex polygonal robot and a convex polygonal obstacle in a 2D world. Assume that any transformation in $SE(2)$ may be applied to $\mathcal{A}$; thus, $\mathcal{C} = \mathbb{R}^2 \times \mathbb{S}^1$ and $q = (x_t, y_t, \theta)$. The task is to define a set of algebraic primitives that can be combined to define $\mathcal{C}_{obs}$. Once again, it is important to distinguish between Type EV and Type VE contacts. Consider how to construct the algebraic primitives for the Type EV contacts; Type VE can be handled in a similar manner.

For the translation-only case, we were able to determine all of the Type EV contacts by sorting the edge normals. With rotation, the ordering of edge normals depends on θ. This implies that the applicability of a Type EV contact depends on θ, the robot orientation. Recall the constraint that the inward normal of $\mathcal{A}$ must point between the outward normals of the edges of $\mathcal{O}$ that contain the vertex of contact, as shown in Figure 4.21. This constraint can be expressed in terms of inner products using the vectors v_1 and v_2. The statement regarding the directions of the normals can equivalently be formulated as the statement that the angle between n and v_1, and between n and v_2, must each be less than $\pi/2$. Using inner products,

this implies that $n \cdot v_1 \geq 0$ and $n \cdot v_2 \geq 0$. As in the translation case, the condition $n \cdot v = 0$ is required for contact. Observe that n now depends on θ. For any $q \in \mathcal{C}$, if $n(\theta) \cdot v_1 \geq 0$, $n(\theta) \cdot v_2 \geq 0$, and $n(\theta) \cdot v(q) > 0$, then $q \in \mathcal{C}_{free}$. Let H_f denote the set of configurations that satisfy these conditions. These conditions imply that a point is in $\mathcal{C}_{free}$. Furthermore, any other Type EV and Type VE contacts could imply that more points are in $\mathcal{C}_{free}$. Ordinarily, $H_f \subset \mathcal{C}_{free}$, which implies that the complement, $\mathcal{C} \setminus H_f$, is a superset of $\mathcal{C}_{obs}$ (thus, $\mathcal{C}_{obs} \subset \mathcal{C} \setminus H_f$). Let $H_A = \mathcal{C} \setminus H_f$. Using the primitives

$$H_1 = \{q \in \mathcal{C} \mid n(\theta) \cdot v_1 \leq 0\}, \tag{4.39}$$

$$H_2 = \{q \in \mathcal{C} \mid n(\theta) \cdot v_2 \leq 0\}, \tag{4.40}$$

and

$$H_3 = \{q \in \mathcal{C} \mid n(\theta) \cdot v(q) \leq 0\}, \tag{4.41}$$

let $H_A = H_1 \cup H_2 \cup H_3$.

It is known that $\mathcal{C}_{obs} \subseteq H_A$, but H_A may contain points in $\mathcal{C}_{free}$. The situation is similar to what was explained in Section 3.1.1 for building a model of a convex polygon from half-planes. In the current setting, it is only known that any configuration outside of H_A must be in $\mathcal{C}_{free}$. If H_A is intersected with all other corresponding sets for each possible Type EV and Type VE contact, then the result is $\mathcal{C}_{obs}$. Each contact has the opportunity to remove a portion of $\mathcal{C}_{free}$ from consideration. Eventually, enough pieces of $\mathcal{C}_{free}$ are removed so that the only configurations remaining must lie in $\mathcal{C}_{obs}$. For any Type EV contact, $(H_1 \cup H_2) \setminus H_3 \subseteq \mathcal{C}_{free}$. A similar statement can be made for Type VE contacts. A logical predicate, similar to that defined in Section 3.1.1, can be constructed to determine whether $q \in \mathcal{C}_{obs}$ in time that is linear in the number of $\mathcal{C}_{obs}$ primitives.

One important issue remains. The expression $n(\theta)$ is not a polynomial because of the $\cos\theta$ and $\sin\theta$ terms in the rotation matrix of $SO(2)$. If polynomials could be substituted for these expressions, then everything would be fixed because the expression of the normal vector (not a unit normal) and the inner product are both linear functions, thereby transforming polynomials into polynomials. Such a substitution can be made using stereographic projection (see [588]); however, a simpler approach is to use complex numbers to represent rotation. Recall that when $a + bi$ is used to represent rotation, each rotation matrix in $SO(2)$ is represented as (4.18), and the 3×3 homogeneous transformation matrix becomes

$$T(a, b, x_t, y_t) = \begin{pmatrix} a & -b & x_t \\ b & a & y_t \\ 0 & 0 & 1 \end{pmatrix}. \tag{4.42}$$

Using this matrix to transform a point $[x \;\; y \;\; 1]$ results in the point coordinates $(ax - by + x_t, bx + ay + y_t)$. Thus, any transformed point on $\mathcal{A}$ is a linear function of a, b, x_t, and y_t.

This was a simple trick to make a nice, linear function, but what was the cost? The dependency is now on a and b instead of θ. This appears to increase the

dimension of $\mathcal{C}$ from 3 to 4, and $\mathcal{C} = \mathbb{R}^4$. However, an algebraic primitive must be added that constrains a and b to lie on the unit circle.

By using complex numbers, primitives in $\mathbb{R}^4$ are obtained for each Type EV and Type VE contact. By defining $\mathcal{C} = \mathbb{R}^4$, the following algebraic primitives are obtained for a Type EV contact:

$$H_1 = \{(x_t, y_t, a, b) \in \mathcal{C} \mid n(x_t, y_t, a, b) \cdot v_1 \leq 0\}, \tag{4.43}$$

$$H_2 = \{(x_t, y_t, a, b) \in \mathcal{C} \mid n(x_t, y_t, a, b) \cdot v_2 \leq 0\}, \tag{4.44}$$

and

$$H_3 = \{(x_t, y_t, a, b) \in \mathcal{C} \mid n(x_t, y_t, a, b) \cdot v(x_t, y_t, a, b) \leq 0\}. \tag{4.45}$$

This yields $H_A = H_1 \cup H_2 \cup H_3$. To preserve the correct $\mathbb{R}^2 \times \mathbb{S}^1$ topology of $\mathcal{C}$, the set

$$H_s = \{(x_t, y_t, a, b) \in \mathcal{C} \mid a^2 + b^2 - 1 = 0\} \tag{4.46}$$

is intersected with H_A. The set H_s remains fixed over all Type EV and Type VE contacts; therefore, it only needs to be considered once.

Example 4.16 (A Nonlinear Boundary for $\mathcal{C}_{obs}$) Consider adding rotation to the model described in Example 4.15. In this case, all possible contacts between pairs of edges must be considered. For this example, there are 12 Type EV contacts and 12 Type VE contacts. Each contact produces 3 algebraic primitives. With the inclusion of H_s, this simple example produces 73 primitives! Rather than construct all of these, we derive the primitives for a single contact. Consider the Type VE contact between a_3 and b_4-b_1. The outward edge normal n remains fixed at $n = [1, 0]$. The vectors v_1 and v_2 are derived from the edges adjacent to a_3, which are a_3-a_2 and a_3-a_1. Note that each of a_1, a_2, and a_3 depend on the configuration. Using the 2D homogeneous transformation (3.35), a_1 at configuration (x_t, y_t, θ) is $(\cos\theta + x_t, \sin\theta + y_t)$. Using $a + bi$ to represent rotation, the expression of a_1 becomes $(a + x_t, b + y_t)$. The expressions of a_2 and a_3 are $(-b + x_t, a + y_t)$ and $(-a + b + x_t, -b - a + y_t)$, respectively. It follows that $v_1 = a_2 - a_3 = [a - 2b, 2a + b]$ and $v_2 = a_1 - a_3 = [2a - b, a + 2b]$. Note that v_1 and v_2 depend only on the orientation of $\mathcal{A}$, as expected. Assume that v is drawn from b_4 to a_3. This yields $v = a_3 - b_4 = [-a + b + x_t - 1, -a - b + y_t + 1]$. The inner products $v_1 \cdot n$, $v_2 \cdot n$, and $v \cdot n$ can easily be computed to form H_1, H_2, and H_3 as algebraic primitives.

One interesting observation can be made here. The only nonlinear primitive is $a^2 + b^2 = 1$. Therefore, $\mathcal{C}_{obs}$ can be considered as a linear polytope (like a polyhedron, but one dimension higher) in $\mathbb{R}^4$ that is intersected with a cylinder. ■

3D rigid bodies For the case of a 3D rigid body to which any transformation in $SE(3)$ may be applied, the same general principles apply. The quaternion parameterization once again becomes the right way to represent $SO(3)$ because using (4.20) avoids all trigonometric functions in the same way that (4.18) avoided them for $SO(2)$. Unfortunately, (4.20) is not linear in the configuration variables,

as it was for (4.18), but it is at least polynomial. This enables semi-algebraic models to be formed for $\mathcal{C}_{obs}$. Type FV, VF, and EE contacts arise for the $SE(3)$ case. From all of the contact conditions, polynomials that correspond to each patch of $\mathcal{C}_{obs}$ can be made. These patches are polynomials in seven variables: x_t, y_t, z_t, a, b, c, and d. Once again, a special primitive must be intersected with all others; here, it enforces the constraint that unit quaternions are used. This reduces the dimension from 7 back down to 6. Also, constraints should be added to throw away half of $\mathbb{S}^3$, which is redundant because of the identification of antipodal points on $\mathbb{S}^3$.

Chains and trees of bodies For chains and trees of bodies, the ideas are conceptually the same, but the algebra becomes more cumbersome. Recall that the transformation for each link is obtained by a product of homogeneous transformation matrices, as given in (3.53) and (3.57) for the 2D and 3D cases, respectively. If the rotation part is parameterized using complex numbers for $SO(2)$ or quaternions for $SO(3)$, then each matrix consists of polynomial entries. After the matrix product is formed, polynomial expressions in terms of the configuration variables are obtained. Therefore, a semi-algebraic model can be constructed. For each link, all of the contact types need to be considered. Extrapolating from Examples 4.15 and 4.16, you can imagine that no human would ever want to do all of that by hand, but it can at least be automated. The ability to construct this representation automatically is also very important for the existence of theoretical algorithms that solve the motion planning problem combinatorially; see Section 6.4.

If the kinematic chains were formulated for $\mathcal{W} = \mathbb{R}^3$ using the DH parameterization, it may be inconvenient to convert to the quaternion representation. One way to avoid this is to use complex numbers to represent each of the θ_i and α_i variables that appear as configuration variables. This can be accomplished because only cos and sin functions appear in the transformation matrices. They can be replaced by the real and imaginary parts, respectively, of a complex number. The dimension will be increased, but this will be appropriately reduced after imposing the constraints that all complex numbers must have unit magnitude.

4.4 Closed Kinematic Chains

This section continues the discussion from Section 3.4. Suppose that a collection of links is arranged in a way that forms loops. In this case, the C-space becomes much more complicated because the joint angles must be chosen to ensure that the loops remain closed. This leads to constraints such as that shown in (3.80) and Figure 3.26, in which some links must maintain specified positions relative to each other. Consider the set of all configurations that satisfy such constraints. Is this a manifold? It turns out, unfortunately, that the answer is generally *no*. However, the C-space belongs to a nice family of spaces from algebraic geometry

called *varieties*. Algebraic geometry deals with characterizing the solution sets of polynomials. As seen so far in this chapter, all of the kinematics can be expressed as polynomials. Therefore, it may not be surprising that the resulting constraints are a system of polynomials whose solution set represents the C-space for closed kinematic linkages. Although the algebraic varieties considered here need not be manifolds, they can be decomposed into a finite collection of manifolds that fit together nicely.[11]

Unfortunately, a parameterization of the variety that arises from closed chains is available in only a few simple cases. Even the topology of the variety is extremely difficult to characterize. To make matters worse, it was proved in [489] that for every closed, bounded real algebraic variety that can be embedded in $\mathbb{R}^n$, there exists a linkage whose C-space is homeomorphic to it. These troubles imply that most of the time, motion planning algorithms need to work directly with implicit polynomials. For the algebraic methods of Section 6.4.2, this does not pose any conceptual difficulty because the methods already work directly with polynomials. Sampling-based methods usually rely on the ability to efficiently sample configurations, which cannot be easily adapted to a variety without a parameterization. Section 7.4 covers recent methods that extend sampling-based planning algorithms to work for varieties that arise from closed chains.

4.4.1 Mathematical Concepts

To understand varieties, it will be helpful to have definitions of polynomials and their solutions that are more formal than the presentation in Chapter 3.

Fields Polynomials are usually defined over a *field*, which is another object from algebra. A field is similar to a group, but it has more operations and axioms. The definition is given below, and while reading it, keep in mind several familiar examples of fields: the rationals, $\mathbb{Q}$; the reals, $\mathbb{R}$; and the complex plane, $\mathbb{C}$. You may verify that these fields satisfy the following six axioms.

A *field* is a set $\mathbb{F}$ that has two binary operations, $\cdot : \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{F}$ (called *multiplication*) and $+ : \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{F}$ (called *addition*), for which the following axioms are satisfied:

1. (**Associativity**) For all $a, b, c \in \mathbb{F}$, $(a+b)+c = a+(b+c)$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
2. (**Commutativity**) For all $a, b \in \mathbb{F}$, $a + b = b + a$ and $a \cdot b = b \cdot a$.
3. (**Distributivity**) For all $a, b, c \in \mathbb{F}$, $a \cdot (b + c) = a \cdot b + a \cdot c$.
4. (**Identities**) There exist $0, 1 \in \mathbb{F}$, such that $a + 0 = a \cdot 1 = a$ for all $a \in \mathbb{F}$.
5. (**Additive Inverses**) For every $a \in \mathbb{F}$, there exists some $b \in \mathbb{F}$ such that $a + b = 0$.

[11] This is called a Whitney stratification [173, 968].

6. (**Multiplicative Inverses**) For every $a \in F$, except $a = 0$, there exists some $c \in \mathbb{F}$ such that $a \cdot c = 1$.

Compare these axioms to the group definition from Section 4.2.1. Note that a field can be considered as two different kinds of groups, one with respect to multiplication and the other with respect to addition. Fields additionally require commutativity; hence, we cannot, for example, build a field from quaternions. The distributivity axiom appears because there is now an interaction between two different operations, which was not possible with groups.

Polynomials Suppose there are n variables, $x_1, x_2, \ldots, x_n$. A *monomial* over a field $\mathbb{F}$ is a product of the form

$$x_1^{d_1} \cdot x_2^{d_2} \cdot \cdots \cdot x_n^{d_n}, \tag{4.47}$$

in which all of the exponents d_1, d_2, ..., d_n are positive integers. The *total degree* of the monomial is $d_1 + \cdots + d_n$.

A *polynomial* f in variables $x_1, \ldots, x_n$ with coefficients in $\mathbb{F}$ is a finite linear combination of monomials that have coefficients in $\mathbb{F}$. A polynomial can be expressed as

$$\sum_{i=1}^{m} c_i m_i, \tag{4.48}$$

in which m_i is a monomial as shown in (4.47), and $c_i \in \mathbb{F}$ is a *coefficient*. If $c_i \neq 0$, then each $c_i m_i$ is called a *term*. Note that the exponents d_i may be different for every term of f. The *total degree of* f is the maximum total degree among the monomials of the terms of f. The set of all polynomials in $x_1, \ldots, x_n$ with coefficients in $\mathbb{F}$ is denoted by $\mathbb{F}[x_1, \ldots, x_n]$.

Example 4.17 (Polynomials) The definitions correspond exactly to our intuitive notion of a polynomial. For example, suppose $\mathbb{F} = \mathbb{Q}$. An example of a polynomial in $\mathbb{Q}[x_1, x_2, x_3]$ is

$$x_1^4 - \tfrac{1}{2} x_1 x_2 x_3^3 + x_1^2 x_2^2 + 4. \tag{4.49}$$

Note that 1 is a valid monomial; hence, any element of $\mathbb{F}$ may appear alone as a term, such as the $4 \in \mathbb{Q}$ in the polynomial above. The total degree of (4.49) is 5 due to the second term. An equivalent polynomial may be written using nicer variables. Using x, y, and z as variables yields

$$x^4 - \tfrac{1}{2} x y z^3 + x^2 y^2 + 4, \tag{4.50}$$

which belongs to $\mathbb{Q}[x, y, z]$. ■

The set $\mathbb{F}[x_1, \ldots, x_n]$ of polynomials is actually a group with respect to addition; however, it is not a field. Even though polynomials can be multiplied, some polynomials do not have a multiplicative inverse. Therefore, the set $\mathbb{F}[x_1, \ldots, x_n]$ is often referred to as a *commutative ring* of polynomials. A commutative ring is a set with two operations for which every axiom for fields is satisfied except the last one, which would require a multiplicative inverse.

Varieties For a given field $\mathbb{F}$ and positive integer n, the n-dimensional *affine space* over $\mathbb{F}$ is the set

$$\mathbb{F}^n = \{(c_1, \ldots, c_n) \mid c_1, \ldots, c_n \in \mathbb{F}\}. \tag{4.51}$$

For our purposes in this section, an affine space can be considered as a vector space (for an exact definition, see [438]). Thus, $\mathbb{F}^n$ is like a vector version of the scalar field $\mathbb{F}$. Familiar examples of this are $\mathbb{Q}^n$, $\mathbb{R}^n$, and $\mathbb{C}^n$.

A polynomial in $f \in \mathbb{F}[x_1, \ldots, x_n]$ can be converted into a function,

$$f : \mathbb{F}^n \to \mathbb{F}, \tag{4.52}$$

by substituting elements of $\mathbb{F}$ for each variable and evaluating the expression using the field operations. This can be written as $f(a_1, \ldots, a_n) \in \mathbb{F}$, in which each a_i denotes an element of $\mathbb{F}$ that is substituted for the variable x_i.

We now arrive at an interesting question. For a given f, what are the elements of $\mathbb{F}^n$ such that $f(a_1, \ldots, a_n) = 0$? We could also ask the question for some nonzero element, but notice that this is not necessary because the polynomial may be redefined to formulate the question using 0. For example, what are the elements of $\mathbb{R}^2$ such that $x^2 + y^2 = 1$? This familiar equation for $\mathbb{S}^1$ can be reformulated to yield: What are the elements of $\mathbb{R}^2$ such that $x^2 + y^2 - 1 = 0$?

Let $\mathbb{F}$ be a field and let $\{f_1, \ldots, f_k\}$ be a set of polynomials in $\mathbb{F}[x_1, \ldots, x_n]$. The set

$$V(f_1, \ldots, f_k) = \{(a_1, \ldots, a_n) \in \mathbb{F} \mid f_i(a_1, \ldots, a_n) = 0 \text{ for all } 1 \leq i \leq k\} \tag{4.53}$$

is called the *(affine) variety* defined by $f_1, \ldots, f_k$. One interesting fact is that unions and intersections of varieties are varieties. Therefore, they behave like the semi-algebraic sets from Section 3.1.2, but for varieties only equality constraints are allowed. Consider the varieties $V(f_1, \ldots, f_k)$ and $V(g_1, \ldots, g_l)$. Their intersection is given by

$$V(f_1, \ldots, f_k) \cap V(g_1, \ldots, g_l) = V(f_1, \ldots, f_k, g_1, \ldots, g_l), \tag{4.54}$$

because each element of $\mathbb{F}^n$ must produce a 0 value for each of the polynomials in $\{f_1, \ldots, f_k, g_1, \ldots, g_l\}$.

To obtain unions, the polynomials simply need to be multiplied. For example, consider the varieties $V_1, V_2 \subset \mathbb{F}$ defined as

$$V_1 = \{(a_1, \ldots, a_n) \in \mathbb{F} \mid f_1(a_1, \ldots, a_n) = 0\} \tag{4.55}$$

and

$$V_2 = \{(a_1, \ldots, a_n) \in \mathbb{F} \mid f_2(a_1, \ldots, a_n) = 0\}. \tag{4.56}$$

The set $V_1 \cup V_2 \subset \mathbb{F}$ is obtained by forming the polynomial $f = f_1 f_2$. Note that $f(a_1, \ldots, a_n) = 0$ if either $f_1(a_1, \ldots, a_n) = 0$ or $f_2(a_1, \ldots, a_n) = 0$. Therefore, $V_1 \cup V_2$ is a variety. The varieties V_1 and V_2 were defined using a single polynomial, but the same idea applies to any variety. All pairs of the form $f_i g_j$ must appear in the argument of $V(\cdot)$ if there are multiple polynomials.

4.4.2 Kinematic Chains in $\mathbb{R}^2$

To illustrate the concepts it will be helpful to study a simple case in detail. Let $\mathcal{W} = \mathbb{R}^2$, and suppose there is a chain of links, $\mathcal{A}_1$, ..., $\mathcal{A}_n$, as considered in Example 3.3 for $n = 3$. Suppose that the first link is attached at the origin of $\mathcal{W}$ by a revolute joint, and every other link, $\mathcal{A}_i$ is attached to $\mathcal{A}_{i-1}$ by a revolute joint. This yields the C-space

$$\mathcal{C} = \mathbb{S}^1 \times \mathbb{S}^1 \times \cdots \times \mathbb{S}^1 = \mathbb{T}^n, \tag{4.57}$$

which is the n-dimensional torus.

Two links If there are two links, $\mathcal{A}_1$ and $\mathcal{A}_2$, then the C-space can be nicely visualized as a square with opposite faces identified. Each coordinate, θ_1 and θ_2, ranges from 0 to 2π, for which $0 \sim 2\pi$. Suppose that each link has length 1. This yields $a_1 = 1$. A point $(x, y) \in \mathcal{A}_2$ is transformed as

$$\begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 1 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \tag{4.58}$$

To obtain polynomials, the technique from Section 4.2.2 is applied to replace the trigonometric functions using $a_i = \cos\theta_i$ and $b_i = \sin\theta_i$, subject to the constraint $a_i^2 + b_i^2 = 1$. This results in

$$\begin{pmatrix} a_1 & -b_1 & 0 \\ b_1 & a_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_2 & -b_2 & 1 \\ b_2 & a_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \tag{4.59}$$

for which the constraints $a_i^2 + b_i^2 = 1$ for $i = 1, 2$ must be satisfied. This preserves the torus topology of $\mathcal{C}$, but now the C-space is embedded in $\mathbb{R}^4$. The coordinates of each point are $(a_1, b_1, a_2, b_2) \in \mathbb{R}^4$; however, there are only two degrees of freedom because each a_i, b_i pair must lie on a unit circle.

Multiplying the matrices in (4.59) yields the polynomials, $f_1, f_2 \in \mathbb{R}[a_1, b_1, a_2, b_2]$,

$$f_1 = x a_1 a_2 - y a_1 b_2 - x b_1 b_2 + y a_2 b_1 + a_1 \tag{4.60}$$

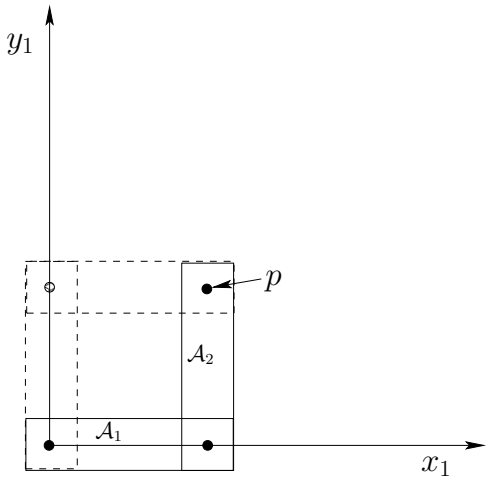


Figure 4.22: Two configurations hold the point p at $(1, 1)$.

and

$$f_2 = -ya_1a_2 + xa_1b_2 + xa_2b_1 - yb_1b_2 + b_1, \tag{4.61}$$

for the x and y coordinates, respectively. Note that the polynomial variables are configuration parameters; x and y are not polynomial variables. For a given point $(x, y) \in \mathcal{A}_2$, all coefficients are determined.

A zero-dimensional variety Now a kinematic closure constraint will be imposed. Fix the point $(1, 0)$ in the body frame of $\mathcal{A}_2$ at $(1, 1)$ in $\mathcal{W}$. This yields the constraints

$$f_1 = a_1a_2 - b_1b_2 + a_1 = 1 \tag{4.62}$$

and

$$f_2 = a_1b_2 + a_2b_1 + b_1 = 1, \tag{4.63}$$

by substituting $x = 1$ and $y = 0$ into (4.60) and (4.61). This yields the variety

$$V(a_1a_2 - b_1b_2 + a_1 - 1, a_1b_2 + a_2b_1 + b_1 - 1, a_1^2 + b_1^2 - 1, a_2^2 + b_2^2 - 1), \tag{4.64}$$

which is a subset of $\mathbb{R}^4$. The polynomials are slightly modified because each constraint must be written in the form $f = 0$.

Although (4.64) represents the constrained configuration space for the chain of two links, it is not very explicit. Without an explicit characterization (i.e., a parameterization), it complicates motion planning. From Figure 4.22 it can be seen that there are only two solutions. These occur for $\theta_1 = 0$, $\theta_2 = \pi/2$ and $\theta_1 = \pi/2$, $\theta_2 = -\pi/2$. In terms of the polynomial variables, (a_1, b_1, a_2, b_2), the two solutions are $(1, 0, 0, 1)$ and $(0, 1, 0, -1)$. These may be substituted into each polynomial in (4.64) to verify that 0 is obtained. Thus, the variety represents two points in $\mathbb{R}^4$. This can also be interpreted as two points on the torus, $\mathbb{S}^1 \times \mathbb{S}^1$.

It might not be surprising that the set of solutions has dimension zero because there are four independent constraints, shown in (4.64), and four variables. Depending on the choices, the variety may be empty. For example, it is physically impossible to bring the point $(1, 0) \in \mathcal{A}_2$ to $(1000, 0) \in \mathcal{W}$.

A one-dimensional variety The most interesting and complicated situations occur when there is a continuum of solutions. For example, if one of the constraints is removed, then a one-dimensional set of solutions can be obtained. Suppose only one variable is constrained for the example in Figure 4.22. Intuitively, this should yield a one-dimensional variety. Set the x coordinate to 0, which yields

$$a_1 a_2 - b_1 b_2 + a_1 = 0, \tag{4.65}$$

and allow any possible value for y. As shown in Figure 4.23a, the point p must follow the y-axis. (This is equivalent to a three-bar linkage that can be constructed by making a third joint that is prismatic and forced to stay along the y-axis.) Figure 4.23b shows the resulting variety $V(a_1 a_2 - b_1 b_2 + a_1)$ but plotted in $\theta_1 - \theta_2$ coordinates to reduce the dimension from 4 to 2 for visualization purposes. To correctly interpret the figures in Figure 4.23, recall that the topology is $\mathbb{S}^1 \times \mathbb{S}^1$, which means that the top and bottom are identified, and also the sides are identified. The center of Figure 4.23b, which corresponds to $(\theta_1, \theta_2) = (\pi, \pi)$, prevents the variety from being a manifold. The resulting space is actually homeomorphic to two circles that touch at a point. Thus, even with such a simple example, the nice manifold structure may disappear. Observe that at (π, π) the links are completely overlapped, and the point p of $\mathcal{A}_2$ is placed at $(0, 0)$ in $\mathcal{W}$. The horizontal line in Figure 4.23b corresponds to keeping the two links overlapping and swinging them around together by varying θ_1. The diagonal lines correspond to moving along configurations such as the one shown in Figure 4.23a. Note that the links and the y-axis always form an isosceles triangle, which can be used to show that the solution set is any pair of angles, θ_1, θ_2 for which $\theta_2 = \pi - \theta_1$. This is the reason why the diagonal curves in Figure 4.23b are linear. Figures 4.23c and 4.23d show the varieties for the constraints

$$a_1 a_2 - b_1 b_2 + a_1 = \tfrac{1}{8}, \tag{4.66}$$

and

$$a_1 a_2 - b_1 b_2 + a_1 = 1, \tag{4.67}$$

respectively. In these cases, the point $(0, 1)$ in $\mathcal{A}_2$ must follow the $x = 1/8$ and $x = 1$ axes, respectively. The varieties are manifolds, which are homeomorphic to $\mathbb{S}^1$. The sequence from Figure 4.23b to 4.23d can be imagined as part of an animation in which the variety shrinks into a small circle. Eventually, it shrinks to a point for the case $a_1 a_2 - b_1 b_2 + a_1 = 2$, because the only solution is when $\theta_1 = \theta_2 = 0$. Beyond this, the variety is the empty set because there are no solutions. Thus, by allowing one constraint to vary, four different topologies are obtained: 1) two circles joined at a point, 2) a circle, 3) a point, and 4) the empty set.

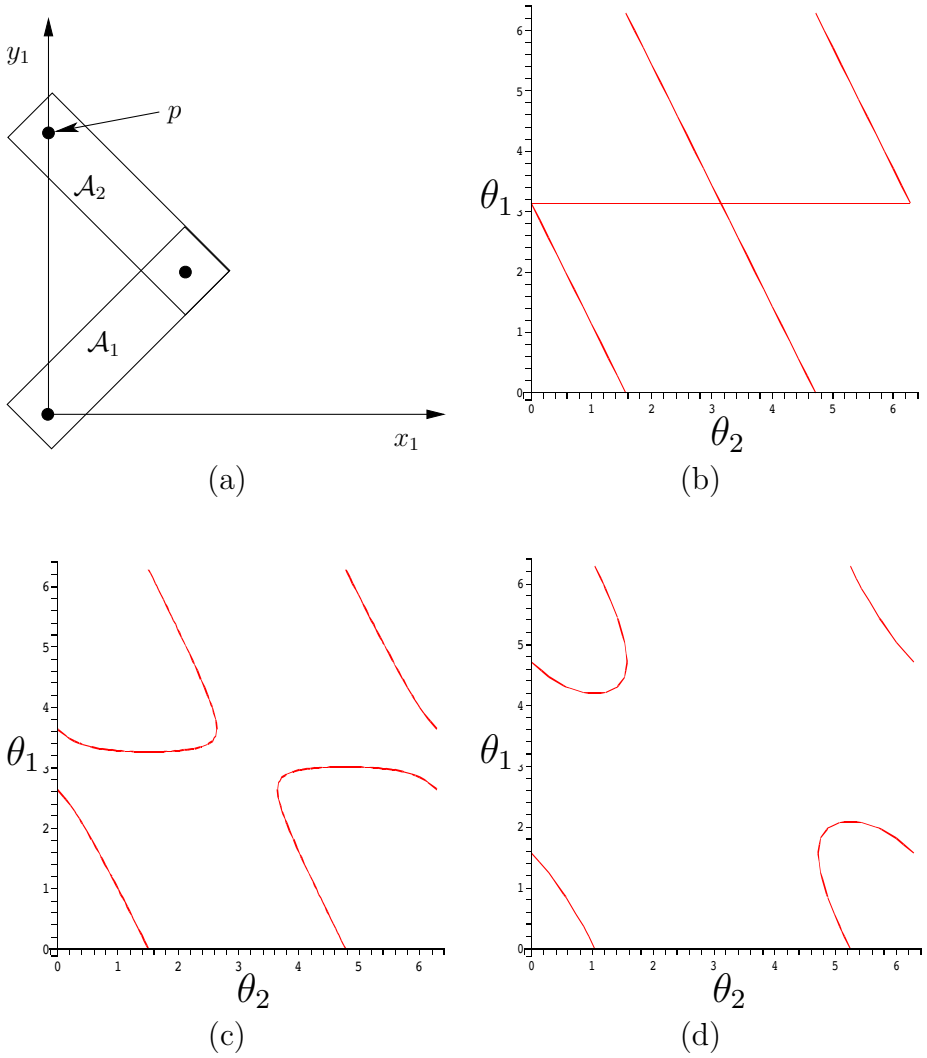


Figure 4.23: A single constraint was added to the point p on $\mathcal{A}_2$, as shown in (a). The curves in (b), (c), and (d) depict the variety for the cases of $f_1 = 0$, $f_1 = 1/8$, and $f_1 = 1$, respectively.

Three links Since visualization is still possible with one more dimension, suppose there are three links, $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$. The C-space can be visualized as a 3D cube with opposite faces identified. Each coordinate θ_i ranges from 0 to 2π, for which $0 \sim 2\pi$. Suppose that each link has length 1 to obtain $a_1 = a_2 = 1$. A point $(x, y) \in \mathcal{A}_3$ is transformed as

$$\begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 10 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_3 & -\sin\theta_3 & 10 \\ \sin\theta_3 & \cos\theta_3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \tag{4.68}$$

To obtain polynomials, let $a_i = \cos\theta_i$ and $b_i = \sin\theta_i$, which results in

$$\begin{pmatrix} a_1 & -b_1 & 0 \\ b_1 & a_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_2 & -b_2 & 1 \\ b_2 & a_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_3 & -b_3 & 1 \\ b_3 & a_3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \tag{4.69}$$

for which the constraints $a_i^2 + b_i^2 = 1$ for $i = 1, 2, 3$ must also be satisfied. This preserves the torus topology of $\mathcal{C}$, but now it is embedded in $\mathbb{R}^6$. Multiplying the matrices yields the polynomials $f_1, f_2 \in \mathbb{R}[a_1, b_1, a_2, b_2, a_3, b_3]$, defined as

$$f_1 = 2a_1a_2a_3 - a_1b_2b_3 + a_1a_2 - 2b_1b_2a_3 - b_1a_2b_3 + a_1, \tag{4.70}$$

and

$$f_2 = 2b_1a_2a_3 - b_1b_2b_3 + b_1a_2 + 2a_1b_2a_3 + a_1a_2b_3, \tag{4.71}$$

for the x and y coordinates, respectively.

Again, consider imposing a single constraint,

$$2a_1a_2a_3 - a_1b_2b_3 + a_1a_2 - 2b_1b_2a_3 - b_1a_2b_3 + a_1 = 0, \tag{4.72}$$

which constrains the point $(1, 0) \in \mathcal{A}_3$ to traverse the y-axis. The resulting variety is an interesting manifold, depicted in Figure 4.24 (remember that the sides of the cube are identified).

Increasing the required f_1 value for the constraint on the final point causes the variety to shrink. Snapshots for $f_1 = 7/8$ and $f_1 = 2$ are shown in Figure 4.25. At $f_1 = 1$, the variety is not a manifold, but it then changes to $\mathbb{S}^2$. Eventually, this sphere is reduced to a point at $f_1 = 3$, and then for $f_1 > 3$ the variety is empty.

Instead of the constraint $f_1 = 0$, we could instead constrain the y coordinate of p to obtain $f_2 = 0$. This yields another 2D variety. If both constraints are enforced simultaneously, then the result is the intersection of the two original varieties. For example, suppose $f_1 = 1$ and $f_2 = 0$. This is equivalent to a kind of *four-bar mechanism* [310], in which the fourth link, $\mathcal{A}_4$, is fixed along the x-axis from 0 to 1. The resulting variety,

$$\begin{aligned} V(&2a_1a_2a_3 - a_1b_2b_3 + a_1a_2 - 2b_1b_2a_3 - b_1a_2b_3 + a_1 - 1, \\ &2b_1a_2a_3 - b_1b_2b_3 + b_1a_2 + 2a_1b_2a_3 + a_1a_2b_3), \end{aligned} \tag{4.73}$$

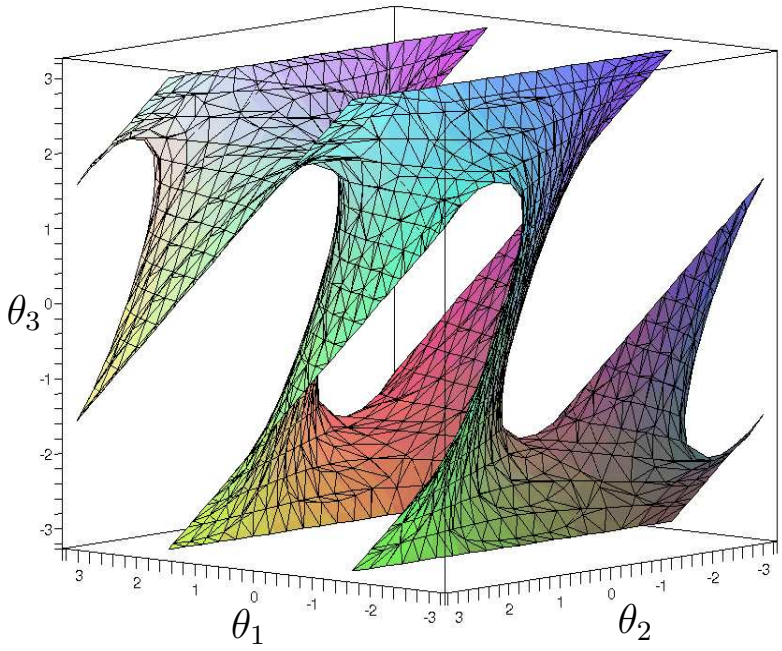


Figure 4.24: The variety for the three-link chain with $f_1 = 0$ is a 2D manifold.

is depicted in Figure 4.26. Using the $\theta_1, \theta_2, \theta_3$ coordinates, the solution may be easily parameterized as a collection of line segments. For all $t \in [0, \pi]$, there exist solution points at $(0, 2t, \pi)$, $(t, 2\pi - t, \pi + t)$, $(2\pi - t, t, \pi - t)$, $(2\pi - t, \pi, \pi + t)$, and $(t, \pi, \pi - t)$. Note that once again the variety is not a manifold. A family of interesting varieties can be generated for the four-bar mechanism by selecting different lengths for the links. The topologies of these mechanisms have been determined for 2D and a 3D extension that uses spherical joints (see [698]).

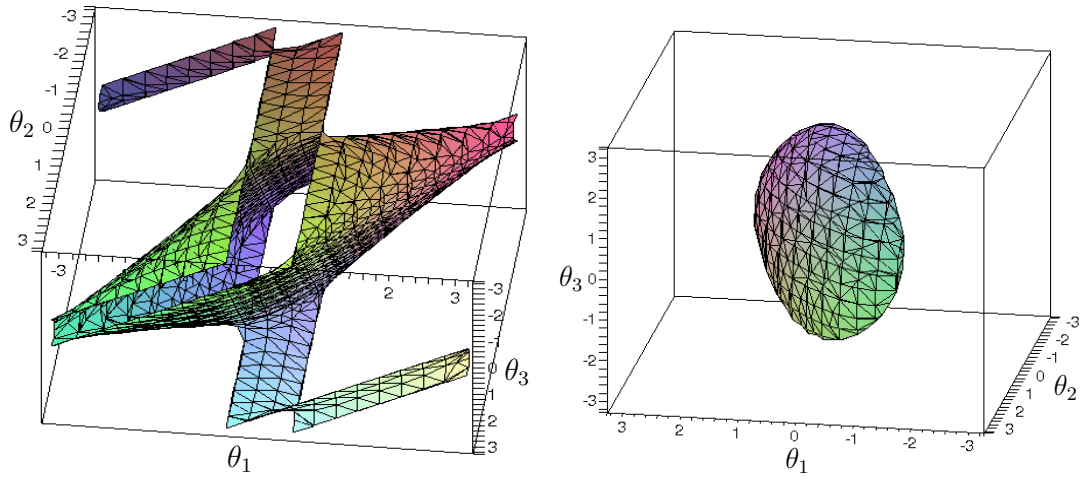
4.4.3 Defining the Variety for General Linkages

We now describe a general methodology for defining the variety. Keeping the previous examples in mind will help in understanding the formulation. In the general case, each constraint can be thought of as a statement of the form:

> *The* ith *coordinate of a point* $p \in \mathcal{A}_j$ *needs to be held at the value* x *in the body frame of* $\mathcal{A}_k$.

For the variety in Figure 4.23b, the first coordinate of a point $p \in \mathcal{A}_2$ was held at the value 0 in $\mathcal{W}$ in the body frame of $\mathcal{A}_1$. The general form must also allow a point to be fixed with respect to the body frames of links other than $\mathcal{A}_1$; this did not occur for the example in Section 4.4.2.

Suppose that n links, $\mathcal{A}_1, \ldots, \mathcal{A}_n$, move in $\mathcal{W} = \mathbb{R}^2$ or $\mathcal{W} = \mathbb{R}^3$. One link, $\mathcal{A}_1$ for convenience, is designated as the root as defined in Section 3.4. Some links



$f_1 = 7/8$ $f_1 = 2$

Figure 4.25: If $f_1 > 0$, then the variety shrinks. If $1 < p < 3$, the variety is a sphere. At $f_1 = 0$ it is a point, and for $f_1 > 3$ it completely vanishes.

are attached in pairs to form joints. A *linkage graph*, $\mathcal{G}(V, E)$, is constructed from the links and joints. Each vertex of $\mathcal{G}$ represents a link in L. Each edge in $\mathcal{G}$ represents a joint. This definition may seem somewhat backward, especially in the plane because links often look like edges and joints look like vertices. This alternative assignment is also possible, but it is not easy to generalize to the case of a single link that has more than two joints. If more than two links are attached at the same point, each generates an edge of $\mathcal{G}$.

The steps to determine the polynomial constraints that express the variety are as follows:

1. Define the linkage graph $\mathcal{G}$ with one vertex per link and one edge per joint. If a joint connects more than two bodies, then one body must be designated as a junction. See Figures 4.27 and 4.28a. In Figure 4.28, links 4, 13, and 23 are designated as junctions in this way.

2. Designate one link as the root, $\mathcal{A}_1$. This link may either be fixed in $\mathcal{W}$, or transformations may be applied. In the latter case, the set of transformations could be $SE(2)$ or $SE(3)$, depending on the dimension of $\mathcal{W}$. This enables the entire linkage to move independently of its internal motions.

3. Eliminate the loops by constructing a spanning tree T of the linkage graph, $\mathcal{G}$. This implies that every vertex (or link) is reachable by a path from the root). Any spanning tree may be used. Figure 4.28b shows a resulting spanning tree after deleting the edges shown with dashed lines.

4. Apply the techniques of Section 3.4 to assign body frames and transformations to the resulting tree of links.

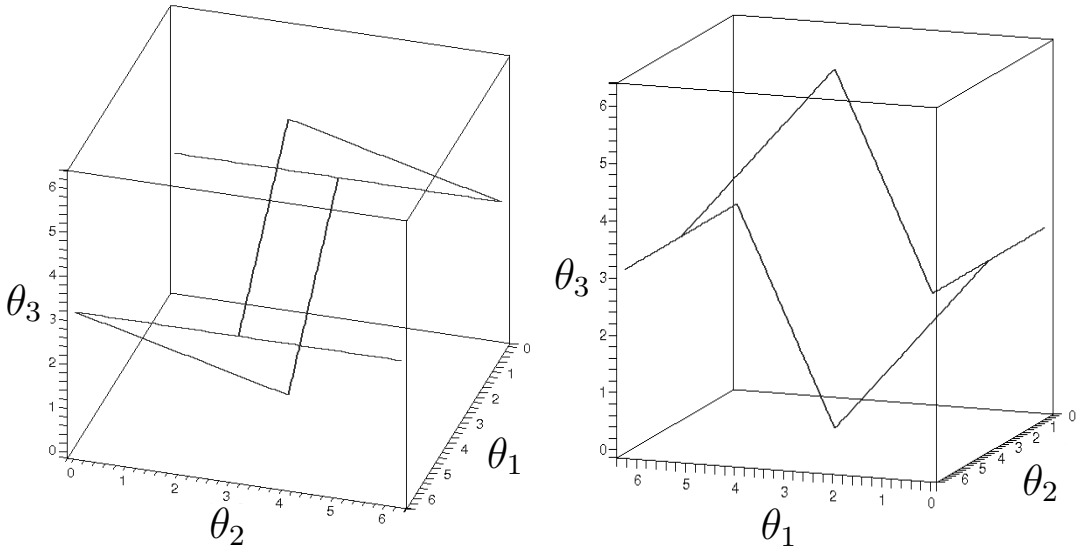


Figure 4.26: If two constraints, $f_1 = 1$ and $f_2 = 0$, are imposed, then the varieties are intersected to obtain a 1D set of solutions. The example is equivalent to a well-studied four-bar mechanism.

5. For each edge of $\mathcal{G}$ that does not appear in T, write a set of constraints between the two corresponding links. In Figure 4.28b, it can be seen that constraints are needed between four pairs of links: 14–15, 21–22, 23–24, and 19–23.

   This is perhaps the trickiest part. For examples like the one shown in Figure 3.27, the constraint may be formulated as in (3.81). This is equivalent to what was done to obtain the example in Figure 4.26, which means that there are actually two constraints, one for each of the x and y coordinates. This will also work for the example shown in Figure 4.27 if all joints are revolute. Suppose instead that two bodies, $\mathcal{A}_j$ and $\mathcal{A}_k$, must be rigidly attached. This requires adding one more constraint that prevents mutual rotation. This could be achieved by selecting another point on $\mathcal{A}_j$ and ensuring that one of its coordinates is in the correct position in the body frame of $\mathcal{A}_k$. If four equations are added, two from each point, then one of them would be redundant because there are only three degrees of freedom possible for $\mathcal{A}_j$ relative to $\mathcal{A}_k$ (which comes from the dimension of $SE(2)$).

   A similar but more complicated situation occurs for $\mathcal{W} = \mathbb{R}^3$. Holding a single point fixed produces three constraints. If a single point is held fixed, then $\mathcal{A}_j$ may achieve any rotation in $SO(3)$ with respect to $\mathcal{A}_k$. This implies that $\mathcal{A}_j$ and $\mathcal{A}_k$ are attached by a spherical joint. If they are attached by a revolute joint, then two more constraints are needed, which can be chosen from the coordinates of a second point. If $\mathcal{A}_j$ and $\mathcal{A}_k$ are rigidly attached, then one constraint from a third point is needed. In total, however, there can

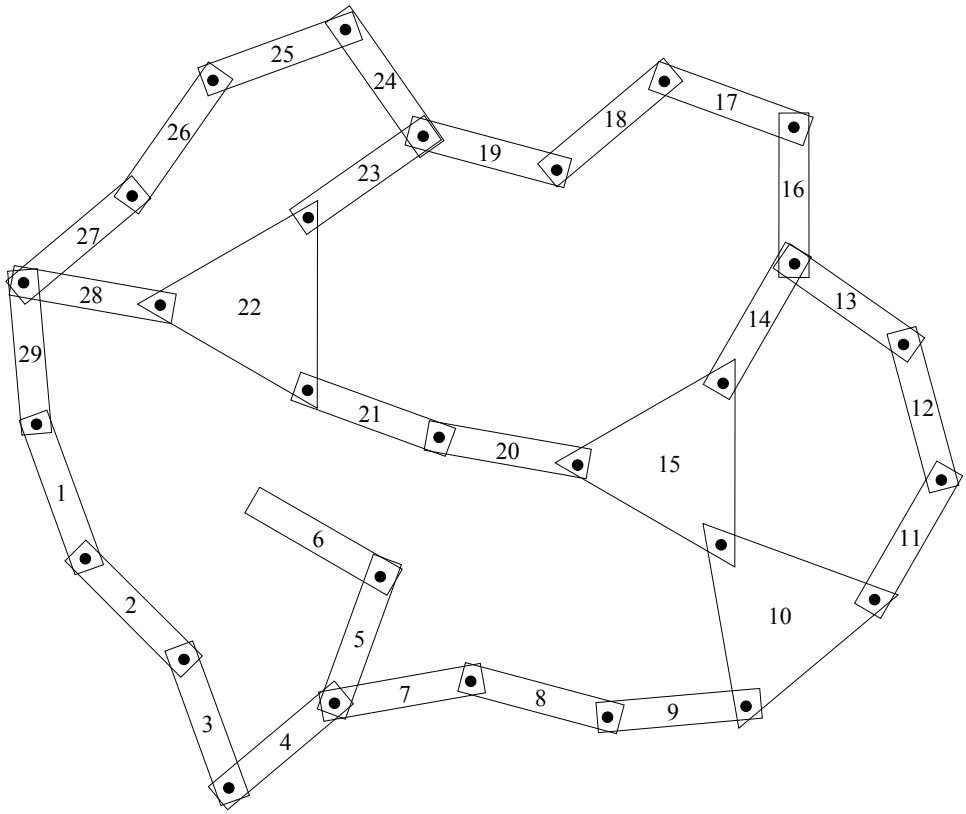


Figure 4.27: A complicated linkage that has 29 links, several loops, links with more than two bodies, and bodies with more than two links. Each integer i indicates link $\mathcal{A}_i$.

be no more than six independent constraints because this is the dimension of $SE(3)$.

6. Convert the trigonometric functions to polynomials. For any 2D transformation, the familiar substitution of complex numbers may be made. If the DH parameterization is used for the 3D case, then each of the $\cos\theta_i$, $\sin\theta_i$ expressions can be parameterized with one complex number, and each of the $\cos\alpha_i$, $\sin\alpha_i$ expressions can be parameterized with another. If the rotation matrix for $SO(3)$ is directly used in the parameterization, then the quaternion parameterization should be used. In all of these cases, polynomial expressions are obtained.

7. List the constraints as polynomial equations of the form $f = 0$. To write the description of the variety, all of the polynomials must be set equal to zero, as was done for the examples in Section 4.4.2.

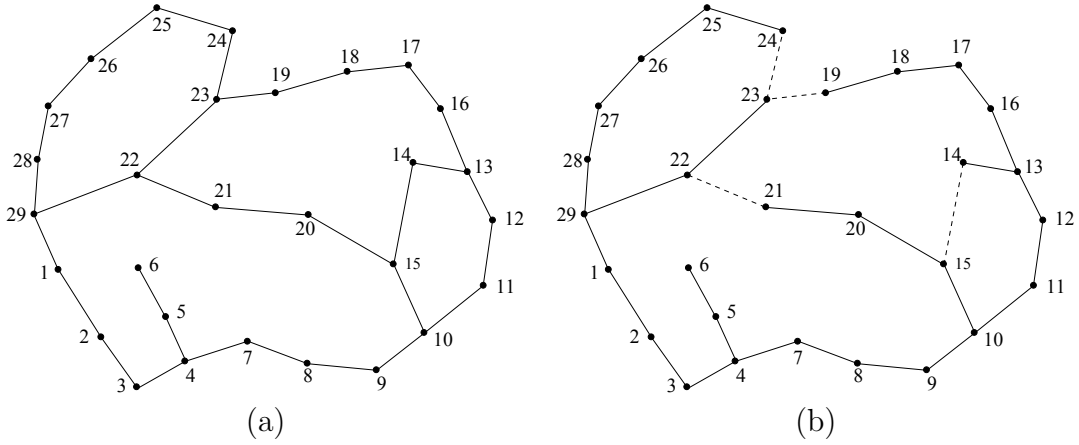


Figure 4.28: (a) One way to make the linkage graph that corresponds to the linkage in Figure 4.27. (b) A spanning tree is indicated by showing the removed edges with dashed lines.

Is it possible to determine the dimension of the variety from the number of independent constraints? The answer is generally *no*, which can be easily seen from the chains of links in Section 4.4.2; they produced varieties of various dimensions, depending on the particular equations. Techniques for computing the dimension exist but require much more machinery than is presented here (see the literature overview at the end of the chapter). However, there is a way to provide a simple upper bound on the number of degrees of freedom. Suppose the total degrees of freedom of the linkage in spanning tree form is m. Each independent constraint can remove at most one degree of freedom. Thus, if there are l independent constraints, then the variety can have no more than $m - l$ dimensions. One expression of this for a general class of mechanisms is the Kutzbach criterion; the planar version of this is called Grübler's formula [310].

One final concern is the obstacle region, $\mathcal{C}_{obs}$. Once the variety has been identified, the obstacle region and motion planning definitions in (4.34) and Formulation 4.1 do not need to be changed. The configuration space $\mathcal{C}$ must be redefined, however, to be the set of configurations that satisfy the closure constraints.

Further Reading

Section 4.1 introduced the basic definitions and concepts of topology. Further study of this fascinating subject can provide a much deeper understanding of configuration spaces. There are many books on topology, some of which may be intimidating, depending on your level of math training. For a heavily illustrated, gentle introduction to topology, see [535]. Another gentle introduction appears in [496]. An excellent text at the graduate level is available on-line: [439]. Other sources include [38, 451]. To understand the motivation for many technical definitions in topology, [911] is helpful. The manifold coverage in Section 4.1.2 was simpler than that found in most sources

because most sources introduce *smooth manifolds*, which are complicated by differentiability requirements (these were not needed in this chapter); see Section 8.3.2 for smooth manifolds. For the configuration spaces of points moving on a topological graph, see [5].

Section 4.2 provided basic C-space definitions. For further reading on matrix groups and their topological properties, see [63], which provides a transition into more advanced material on Lie group theory. For more about quaternions in engineering, see [210, 563]. The remainder of Section 4.2 and most of Section 4.3 were inspired by the coverage in [588]. C-spaces are also covered in [220]. For further reading on computing representations of $\mathcal{C}_{obs}$, see [513, 736] for bitmaps, and Chapter 6 and [865] for combinatorial approaches.

Much of the presentation in Section 4.4 was inspired by the nice introduction to algebraic varieties in [250], which even includes robotics examples; methods for determining the dimension of a variety are also covered. More algorithmic coverage appears in [704]. See [693] for detailed coverage of robots that are designed with closed kinematic chains.

Exercises

1. Consider the set $X = \{1, 2, 3, 4, 5\}$. Let X, $\emptyset$, $\{1, 3\}$, $\{1, 2\}$, $\{2, 3\}$, $\{1\}$, $\{2\}$, and $\{3\}$ be the collection of all subsets of X that are designated as *open sets*.

   (a) Is X a topological space?

   (b) Is it a topological space if $\{1, 2, 3\}$ is added to the collection of open sets? Explain.

   (c) What are the closed sets (assuming $\{1, 2, 3\}$ is included as an open set)?

   (d) Are any subsets of X neither open nor closed?

2. Continuous functions for the strange topology:

   (a) Give an example of a continuous function, $f : X \rightarrow X$, for the strange topology in Example 4.4.

   (b) Characterize the set of all possible continuous functions.

3. For the letters of the Russian alphabet, А, Б, В, Г, Д, Е, Ё, Ж, З, И, Й, К, Л, М, Н, О, П, Р, С, Т, У, Ф, Х, Ц, Ч, Ш, Щ, Ъ, Ы, Ь, Э, Ю, Я, determine which pairs are homeomorphic. Imagine each as a 1D subset of $\mathbb{R}^2$ and draw them accordingly before solving the problem.

4. Prove that homeomorphisms yield an equivalence relation on the collection of all topological spaces.

5. What is the dimension of the C-space for a cylindrical rod that can translate and rotate in $\mathbb{R}^3$? If the rod is rotated about its central axis, it is assumed that the rod's position and orientation are not changed in any detectable way. Express the C-space of the rod in terms of a Cartesian product of simpler spaces (such as $\mathbb{S}^1$, $\mathbb{S}^2$, $\mathbb{R}^n$, P^2, etc.). What is your reasoning?

6. Let $\tau_1 : [0,1] \rightarrow \mathbb{R}^2$ be a loop path that traverses the unit circle in the plane, defined as $\tau_1(s) = (\cos(2\pi s), \sin(2\pi s))$. Let $\tau_2 : [0,1] \rightarrow \mathbb{R}^2$ be another loop path: $\tau_1(s) = (-2 + 3\cos(2\pi s), \frac{1}{2}\sin(2\pi s))$. This path traverses an ellipse that is centered at $(-2, 0)$. Show that τ_1 and τ_2 are homotopic (by constructing a continuous function with an additional parameter that "morphs" τ_1 into τ_2).

7. Prove that homotopy yields an equivalence relation on the set of all paths from some $x_1 \in X$ to some $x_2 \in X$, in which x_1 and x_2 may be chosen arbitrarily.

8. Determine the C-space for a spacecraft that can translate and rotate in a 2D *Asteroids*-style video game. The sides of the screen are identified. The top and bottom are also identified. There are no "twists" in the identifications.

9. Repeat the derivation of H_A from Section 4.3.3, but instead consider Type VE contacts.

10. Determine the C-space for a car that drives around on a huge sphere (such as the earth with no mountains or oceans). Assume the sphere is big enough so that its curvature may be neglected (e.g., the car rests flatly on the earth without wobbling). [Hint: It is not $\mathbb{S}^2 \times \mathbb{S}^1$.]

11. Suppose that $\mathcal{A}$ and $\mathcal{O}$ are each defined as equilateral triangles, with coordinates $(0,0)$, $(2,0)$, and $(1, \sqrt{3})$. Determine the C-space obstacle. Specify the coordinates of all of its vertices and indicate the corresponding contact type for each edge.

12. Show that (4.20) is a valid rotation matrix for all unit quaternions.

13. Show that $\mathbb{F}[x_1, \ldots, x_n]$, the set of polynomials over a field $\mathbb{F}$ with variables $x_1, \ldots, x_n$, is a group with respect to addition.

14. Quaternions:

    (a) Define a unit quaternion h_1 that expresses a rotation of $-\frac{\pi}{2}$ around the axis given by the vector $[\frac{1}{\sqrt{3}} \;\; \frac{1}{\sqrt{3}} \;\; \frac{1}{\sqrt{3}}]$.

    (b) Define a unit quaternion h_2 that expresses a rotation of π around the axis given by the vector $[0 \;\; 1 \;\; 0]$.

    (c) Suppose the rotation represented by h_1 is performed, followed by the rotation represented by h_2. This combination of rotations can be represented as a single rotation around an axis given by a vector. Find this axis and the angle of rotation about this axis.

15. What topological space is contributed to the C-space by a spherical joint that achieves any orientation except the identity?

16. Suppose five polyhedral bodies float freely in a 3D world. They are each capable of rotating and translating. If these are treated as "one" composite robot, what is the topology of the resulting C-space (assume that the bodies are *not* attached to each other)? What is its dimension?

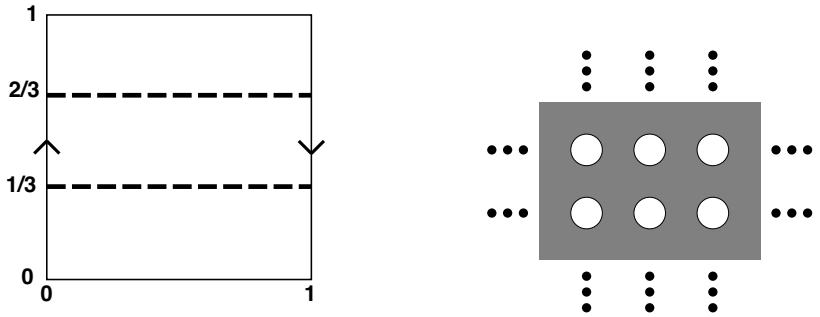


Figure 4.29: (a) What topological space is obtained after slicing the Möbius band? (b) Is a manifold obtained after tearing holes out of the plane?

17. Suppose a goal region $G \subseteq \mathcal{W}$ is defined in the C-space by requiring that the *entire* robot is contained in G. For example, a car may have to be parked entirely within a space in a parking lot.

    (a) Give a definition of $\mathcal{C}_{goal}$ that is similar to (4.34) but pertains to containment of $\mathcal{A}$ inside of G.

    (b) For the case in which $\mathcal{A}$ and G are convex and polygonal, develop an algorithm for efficiently computing $\mathcal{C}_{goal}$.

18. Figure 4.29a shows the Möbius band defined by identification of sides of the unit square. Imagine that scissors are used to cut the band along the two dashed lines. Describe the resulting topological space. Is it a manifold? Explain.

19. Consider Figure 4.29b, which shows the set of points in $\mathbb{R}^2$ that are remaining after a closed disc of radius $1/4$ with center (x, y) is removed for every value of (x, y) such that x and y are both integers.

    (a) Is the remaining set of points a manifold? Explain.

    (b) Now remove discs of radius $1/2$ instead of $1/4$. Is a manifold obtained?

    (c) Finally, remove disks of radius $2/3$. Is a manifold obtained?

20. Show that the solution curves shown in Figure 4.26 correctly illustrate the variety given in (4.73).

21. Find the number of faces of $\mathcal{C}_{obs}$ for a cube and regular tetrahedron, assuming $\mathcal{C}$ is $SE(3)$. How many faces of each contact type are obtained?

22. Following the analysis matrix subgroups from Section 4.2, determine the dimension of $SO(4)$, the group of 4×4 rotation matrices. Can you characterize this topological space?

23. Suppose that a kinematic chain of spherical joints is given. Show how to use (4.20) as the rotation part in each homogeneous transformation matrix, as opposed to

using the DH parameterization. Explain why using (4.20) would be preferable for motion planning applications.

24. Suppose that the constraint that c is held to position $(10, 10)$ is imposed on the mechanism shown in Figure 3.29. Using complex numbers to represent rotation, express this constraint using polynomial equations.

25. The Tangle toy is made of 18 pieces of macaroni-shaped joints that are attached together to form a loop. Each attachment between joints forms a revolute joint. Each link is a curved tube that extends around 1/4 of a circle. What is the dimension of the variety that results from maintaining the loop? What is its configuration space (accounting for internal degrees of freedom), assuming the toy can be placed anywhere in $\mathbb{R}^3$?

Implementations

26. Computing C-space obstacles:

    (a) Implement the algorithm from Section 4.3.2 to construct a convex, polygonal C-space obstacle.

    (b) Now allow the robot to rotate in the plane. For any convex robot and obstacle, compute the orientations at which the C-space obstacle fundamentally changes due to different Type EV and Type VE contacts becoming active.

    (c) Animate the changing C-space obstacle by using the robot orientation as the time axis in the animation.

27. Consider "straight-line" paths that start at the origin (lower left corner) of the manifolds shown in Figure 4.5 and leave at a particular angle, which is input to the program. The lines must respect identifications; thus, as the line hits the edge of the square, it may continue onward. Study the conditions under which the lines fill the entire space versus forming a finite pattern (i.e., a segment, stripes, or a tiling).

Chapter 5

Sampling-Based Motion Planning

There are two main philosophies for addressing the motion planning problem, in Formulation 4.1 from Section 4.3.1. This chapter presents one of the philosophies, *sampling-based motion planning*, which is outlined in Figure 5.1. The main idea is to avoid the explicit construction of $\mathcal{C}_{obs}$, as described in Section 4.3, and instead conduct a search that probes the C-space with a sampling scheme. This probing is enabled by a collision detection module, which the motion planning algorithm considers as a "black box." This enables the development of planning algorithms that are independent of the particular geometric models. The collision detection module handles concerns such as whether the models are semi-algebraic sets, 3D triangles, nonconvex polyhedra, and so on. This general philosophy has been very successful in recent years for solving problems from robotics, manufacturing, and biological applications that involve thousands and even millions of geometric primitives. Such problems would be practically impossible to solve using techniques that explicitly represent $\mathcal{C}_{obs}$.

Notions of completeness It is useful to define several notions of completeness for sampling-based algorithms. These algorithms have the drawback that they result in weaker guarantees that the problem will be solved. An algorithm is considered *complete* if for any input it correctly reports whether there is a so-

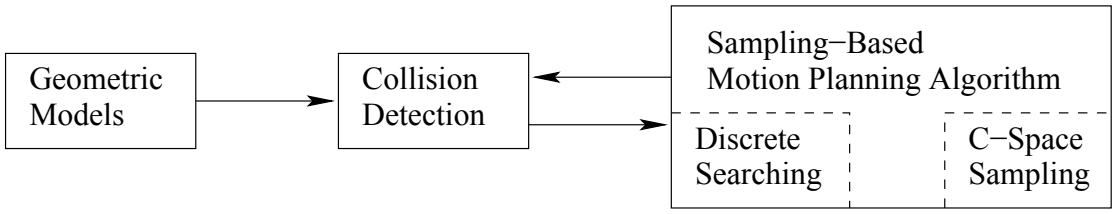


Figure 5.1: The sampling-based planning philosophy uses collision detection as a "black box" that separates the motion planning from the particular geometric and kinematic models. C-space sampling and discrete planning (i.e., searching) are performed.

lution in a finite amount of time. If a solution exists, it must return one in finite time. The combinatorial motion planning methods of Chapter 6 will achieve this. Unfortunately, such completeness is not achieved with sampling-based planning. Instead, weaker notions of completeness are tolerated. The notion of denseness becomes important, which means that the samples come arbitrarily close to any configuration as the number of iterations tends to infinity. A deterministic approach that samples densely will be called *resolution complete*. This means that if a solution exists, the algorithm will find it in finite time; however, if a solution does not exist, the algorithm may run forever. Many sampling-based approaches are based on random sampling, which is dense with probability one. This leads to algorithms that are *probabilistically complete*, which means that with enough points, the probability that it finds an existing solution converges to one. The most relevant information, however, is the rate of convergence, which is usually very difficult to establish.

Section 5.1 presents metric and measure space concepts, which are fundamental to nearly all sampling-based planning algorithms. Section 5.2 presents general sampling concepts and quality criteria that are effective for analyzing the performance of sampling-based algorithms. Section 5.3 gives a brief overview of collision detection algorithms, to gain an understanding of the information available to a planning algorithm and the computation price that must be paid to obtain it. Section 5.4 presents a framework that defines algorithms which solve motion planning problems by integrating sampling and discrete planning (i.e., searching) techniques. These approaches can be considered *single query* in the sense that a single pair, (q_I, q_G), is given, and the algorithm must search until it finds a solution (or it may report early failure). Section 5.5 focuses on *rapidly exploring random trees* (RRTs) and *rapidly exploring dense trees* (RDTs), which are used to develop efficient single-query planning algorithms. Section 5.6 covers *multiple-query* algorithms, which invest substantial preprocessing effort to build a data structure that is later used to obtain efficient solutions for many initial-goal pairs. In this case, it is assumed that the obstacle region $\mathcal{O}$ remains the same for every query.

5.1 Distance and Volume in C-Space

Virtually all sampling-based planning algorithms require a function that measures the distance between two points in $\mathcal{C}$. In most cases, this results in a *metric space*, which is introduced in Section 5.1.1. Useful examples for motion planning are given in Section 5.1.2. It will also be important in many of these algorithms to define the volume of a subset of $\mathcal{C}$. This requires a *measure space*, which is introduced in Section 5.1.3. Section 5.1.4 introduces invariant measures, which should be used whenever possible.

5.1.1 Metric Spaces

It is straightforward to define Euclidean distance in $\mathbb{R}^n$. To define a distance function over any $\mathcal{C}$, however, certain axioms will have to be satisfied so that it coincides with our expectations based on Euclidean distance.

The following definition and axioms are used to create a function that converts a topological space into a metric space.[1] A *metric space* (X, ρ) is a topological space X equipped with a function $\rho : X \times X \rightarrow \mathbb{R}$ such that for any $a, b, c \in X$:

1. **Nonnegativity:** $\rho(a, b) \geq 0$.
2. **Reflexivity:** $\rho(a, b) = 0$ if and only if $a = b$.
3. **Symmetry:** $\rho(a, b) = \rho(b, a)$.
4. **Triangle inequality:** $\rho(a, b) + \rho(b, c) \geq \rho(a, c)$.

The function ρ defines distances between points in the metric space, and each of the four conditions on ρ agrees with our intuitions about distance. The final condition implies that ρ is optimal in the sense that the distance from a to c will always be less than or equal to the total distance obtained by traveling through an intermediate point b on the way from a to c.

L_p metrics The most important family of metrics over $\mathbb{R}^n$ is given for any $p \geq 1$ as

$$\rho(x, x') = \left(\sum_{i=1}^{n} |x_i - x'_i|^p \right)^{1/p}. \tag{5.1}$$

For each value of p, (5.1) is called an L_p *metric* (pronounced "el pee"). The three most common cases are

1. L_2: The *Euclidean metric*, which is the familiar Euclidean distance in $\mathbb{R}^n$.
2. L_1: The *Manhattan metric*, which is often nicknamed this way because in $\mathbb{R}^2$ it corresponds to the length of a path that is obtained by moving along an axis-aligned grid. For example, the distance from $(0, 0)$ to $(2, 5)$ is 7 by traveling "east two blocks" and then "north five blocks".
3. L_∞: The L_∞ *metric* must actually be defined by taking the limit of (5.1) as p tends to infinity. The result is

$$L_\infty(x, x') = \max_{1 \leq i \leq n} \{|x_i - x'_i|\}, \tag{5.2}$$

which seems correct because the larger the value of p, the more the largest term of the sum in (5.1) dominates.

[1] Some topological spaces are not *metrizable*, which means that no function exists that satisfies the axioms. Many metrization theorems give sufficient conditions for a topological space to be metrizable [451], and virtually any space that has arisen in motion planning will be metrizable.

An L_p metric can be derived from a norm on a vector space. An L_p *norm* over $\mathbb{R}^n$ is defined as

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}. \tag{5.3}$$

The case of $p = 2$ is the familiar definition of the magnitude of a vector, which is called the *Euclidean norm*. For example, assume the vector space is $\mathbb{R}^n$, and let $\| \cdot \|$ be the standard Euclidean norm. The L_2 metric is $\rho(x, y) = \|x - y\|$. Any L_p metric can be written in terms of a vector subtraction, which is notationally convenient.

Metric subspaces By verifying the axioms, it can be shown that any subspace $Y \subset X$ of a metric space (X, ρ) itself becomes a metric space by restricting the domain of ρ to $Y \times Y$. This conveniently provides metrics on any of the manifolds and varieties from Chapter 4 by simply using any L_p metric on $\mathbb{R}^m$, the space in which the manifold or variety is embedded.

Cartesian products of metric spaces Metrics extend nicely across Cartesian products, which is very convenient because C-spaces are often constructed from Cartesian products, especially in the case of multiple bodies. Let (X, ρ_x) and (Y, ρ_y) be two metric spaces. A metric space (Z, ρ_z) can be constructed for the Cartesian product $Z = X \times Y$ by defining the metric ρ_z as

$$\rho_z(z, z') = \rho_z(x, y, x', y') = c_1 \rho_x(x, x') + c_2 \rho_y(y, y'), \tag{5.4}$$

in which $c_1 > 0$ and $c_2 > 0$ are any positive real constants, and $x, x' \in X$ and $y, y' \in Y$. Each $z \in Z$ is represented as $z = (x, y)$.

Other combinations lead to a metric for Z; for example,

$$\rho_z(z, z') = \Big(c_1 \big[\rho_x(x, x')\big]^p + c_2 \big[\rho_y(y, y')\big]^p \Big)^{1/p}, \tag{5.5}$$

is a metric for any positive integer p. Once again, two positive constants must be chosen. It is important to understand that many choices are possible, and there may not necessarily be a "correct" one.

5.1.2 Important Metric Spaces for Motion Planning

This section presents some metric spaces that arise frequently in motion planning.

Example 5.1 ($SO(2)$ Metric Using Complex Numbers) If $SO(2)$ is represented by unit complex numbers, recall that the C-space is the subset of $\mathbb{R}^2$ given by $\{(a, b) \in \mathbb{R}^2 \mid a^2 + b^2 = 1\}$. Any L_p metric from $\mathbb{R}^2$ may be applied. Using the Euclidean metric,

$$\rho(a_1, b_1, a_2, b_2) = \sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2}, \tag{5.6}$$

for any pair of points (a_1, b_1) and (a_2, b_2). ■

Example 5.2 ($SO(2)$ Metric by Comparing Angles) You might have noticed that the previous metric for $SO(2)$ does not give the distance traveling along the circle. It instead takes a shortcut by computing the length of the line segment in $\mathbb{R}^2$ that connects the two points. This distortion may be undesirable. An alternative metric is obtained by directly comparing angles, θ_1 and θ_2. However, in this case special care has to be given to the identification, because there are two ways to reach θ_2 from θ_1 by traveling along the circle. This causes a min to appear in the metric definition:

$$\rho(\theta_1, \theta_2) = \min\big\{|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|\big\}, \tag{5.7}$$

for which $\theta_1, \theta_2 \in [0, 2\pi]/\sim$. This may alternatively be expressed using the complex number representation $a + bi$ as an angle between two vectors:

$$\rho(a_1, b_1, a_2, b_2) = \cos^{-1}(a_1 a_2 + b_1 b_2), \tag{5.8}$$

for two points (a_1, b_1) and (a_2, b_2). ■

Example 5.3 (An $SE(2)$ Metric) Again by using the subspace principle, a metric can easily be obtained for $SE(2)$. Using the complex number representation of $SO(2)$, each element of $SE(2)$ is a point $(x_t, y_t, a, b) \in \mathbb{R}^4$. The Euclidean metric, or any other L_p metric on $\mathbb{R}^4$, can be immediately applied to obtain a metric. ■

Example 5.4 ($SO(3)$ Metrics Using Quaternions) As usual, the situation becomes more complicated for $SO(3)$. The unit quaternions form a subset $\mathbb{S}^3$ of $\mathbb{R}^4$. Therefore, any L_p metric may be used to define a metric on $\mathbb{S}^3$, but this will not be a metric for $SO(3)$ because antipodal points need to be identified. Let $h_1, h_2 \in \mathbb{R}^4$ represent two unit quaternions (which are being interpreted here as elements of $\mathbb{R}^4$ by ignoring the quaternion algebra). Taking the identifications into account, the metric is

$$\rho(h_1, h_2) = \min\big\{\|h_1 - h_2\|, \|h_1 + h_2\|\big\}, \tag{5.9}$$

in which the two arguments of the min correspond to the distances from h_1 to h_2 and $-h_2$, respectively. The $h_1 + h_2$ appears because h_2 was negated to yield its antipodal point, $-h_2$.

As in the case of $SO(2)$, the metric in (5.9) may seem distorted because it measures the length of line segments that cut through the interior of $\mathbb{S}^3$, as opposed to traveling along the surface. This problem can be fixed to give a very natural metric for $SO(3)$, which is based on *spherical linear interpolation*. This takes the line segment that connects the points and pushes it outward onto $\mathbb{S}^3$. It is easier to visualize this by dropping a dimension. Imagine computing the distance between two points on $\mathbb{S}^2$. If these points lie on the equator, then spherical linear interpolation yields a distance proportional to that obtained by traveling along

the equator, as opposed to cutting through the interior of $\mathbb{S}^2$ (for points not on the equator, use the *great circle* through the points).

It turns out that this metric can easily be defined in terms of the inner product between the two quaternions. Recall that for unit vectors v_1 and v_2 in $\mathbb{R}^n$, $v_1 \cdot v_2 = \cos\theta$, in which θ is the angle between the vectors. This angle is precisely what is needed to give the proper distance along $\mathbb{S}^3$. The resulting metric is a surprisingly simple extension of (5.8). The distance along $\mathbb{S}^3$ between two quaternions is

$$\rho_s(h_1, h_2) = \cos^{-1}(a_1a_2 + b_1b_2 + c_1c_2 + d_1d_2), \tag{5.10}$$

in which each $h_i = (a_i, b_i, c_i, d_i)$. Taking identification into account yields the metric

$$\rho(h_1, h_2) = \min\big\{\rho_s(h_1, h_2), \rho_s(h_1, -h_2)\big\}. \tag{5.11}$$

■

Example 5.5 (Another $SE(2)$ Metric) For many C-spaces, the problem of relating different kinds of quantities arises. For example, any metric defined on $SE(2)$ must compare both distance in the plane and an angular quantity. For example, even if $c_1 = c_2 = 1$, the range for $\mathbb{S}^1$ is $[0, 2\pi)$ using radians but $[0, 360)$ using degrees. If the same constant c_2 is used in either case, two very different metrics are obtained. The units applied to $\mathbb{R}^2$ and $\mathbb{S}^1$ are completely incompatible. ■

Example 5.6 (Robot Displacement Metric) Sometimes this incompatibility problem can be fixed by considering the robot displacement. For any two configurations $q_1, q_2 \in \mathcal{C}$, a robot displacement metric can be defined as

$$\rho(q_1, q_2) = \max_{a \in \mathcal{A}} \big\{\|a(q_1) - a(q_2)\|\big\}, \tag{5.12}$$

in which $a(q_i)$ is the position of the point a in the world when the robot $\mathcal{A}$ is at configuration q_i. Intuitively, the robot displacement metric yields the maximum amount in $\mathcal{W}$ that any part of the robot is displaced when moving from configuration q_1 to q_2. The difficulty and efficiency with which this metric can be computed depend strongly on the particular robot geometric model and kinematics. For a convex polyhedral robot that can translate and rotate, it is sufficient to check only vertices. The metric may appear to be ideal, but efficient algorithms are not known for most situations. ■

Example 5.7 ($\mathbb{T}^n$ Metrics) Next consider making a metric over a torus $\mathbb{T}^n$. The Cartesian product rules such as (5.4) and (5.5) can be extended over every copy of $\mathbb{S}^1$ (one for each parameter θ_i). This leads to n arbitrary coefficients c_1, c_2, ..., c_n.

Robot displacement could be used to determine the coefficients. For example, if the robot is a chain of links, it might make sense to weight changes in the first link more heavily because the entire chain moves in this case. When the last parameter is changed, only the last link moves; in this case, it might make sense to give it less weight. ■

Example 5.8 ($SE(3)$ Metrics) Metrics for $SE(3)$ can be formed by applying the Cartesian product rules to a metric for $\mathbb{R}^3$ and a metric for $SO(3)$, such as that given in (5.11). Again, this unfortunately leaves coefficients to be specified. These issues will arise again in Section 5.3.4, where more details appear on robot displacement. ■

Pseudometrics Many planning algorithms use functions that behave somewhat like a distance function but may fail to satisfy all of the metric axioms. If such distance functions are used, they will be referred to as *pseudometrics*. One general principle that can be used to derive pseudometrics is to define the distance to be the optimal cost-to-go for some criterion (recall discrete cost-to-go functions from Section 2.3). This will become more important when differential constraints are considered in Chapter 14.

In the continuous setting, the cost could correspond to the distance traveled by a robot or even the amount of energy consumed. Sometimes, the resulting pseudometric is not symmetric. For example, it requires less energy for a car to travel downhill as opposed to uphill. Alternatively, suppose that a car is only capable of driving forward. It might travel a short distance to go forward from q_1 to some q_2, but it might have to travel a longer distance to reach q_1 from q_2 because it cannot drive in reverse. These issues arise for the Dubins car, which is covered in Sections 13.1.2 and 15.3.1.

An important example of a pseudometric from robotics is a *potential function*, which is an important part of the randomized potential field method, which is discussed in Section 5.4.3. The idea is to make a scalar function that estimates the distance to the goal; however, there may be additional terms that attempt to repel the robot away from obstacles. This generally causes local minima to appear in the distance function, which may cause potential functions to violate the triangle inequality.

5.1.3 Basic Measure Theory Definitions

This section briefly indicates how to define volume in a metric space. This provides a basis for defining concepts such as integrals or probability densities. Measure theory is an advanced mathematical topic that is well beyond the scope of this

book; however, it is worthwhile to briefly introduce some of the basic definitions because they sometimes arise in sampling-based planning.

Measure can be considered as a function that produces real values for subsets of a metric space, (X, ρ). Ideally, we would like to produce a nonnegative value, $\mu(A) \in [0, \infty]$, for any subset $A \subseteq X$. Unfortunately, due to the Banach-Tarski paradox, if $X = \mathbb{R}^n$, there are some subsets for which trying to assign volume leads to a contradiction. If X is finite, this cannot happen. Therefore, it is hard to visualize the problem; see [836] for a construction of the bizarre nonmeasurable sets. Due to this problem, a workaround was developed by defining a collection of subsets that avoids the paradoxical sets. A collection $\mathcal{B}$ of subsets of X is called a *sigma algebra* if the following axioms are satisfied:

1. The empty set is in $\mathcal{B}$.
2. If $B \in \mathcal{B}$, then $X \setminus B \in \mathcal{B}$.
3. For any collection of a countable number of sets in $\mathcal{B}$, their union must also be in $\mathcal{B}$.

Note that the last two conditions together imply that the intersection of a countable number of sets in $\mathcal{B}$ is also in $\mathcal{B}$. The sets in $\mathcal{B}$ are called the *measurable sets*.

A nice sigma algebra, called the *Borel sets*, can be formed from any metric space (X, ρ) as follows. Start with the set of all open balls in X. These are the sets of the form

$$B(x, r) = \{x' \in X \mid \rho(x, x') < r\} \tag{5.13}$$

for any $x \in X$ and any $r \in (0, \infty)$. From the open balls, the *Borel sets* $\mathcal{B}$ are the sets that can be constructed from these open balls by using the sigma algebra axioms. For example, an open square in $\mathbb{R}^2$ is in $\mathcal{B}$ because it can be constructed as the union of a countable number of balls (infinitely many are needed because the curved balls must converge to covering the straight square edges). By using Borel sets, the nastiness of nonmeasurable sets is safely avoided.

Example 5.9 (Borel Sets) A simple example of $\mathcal{B}$ can be constructed for $\mathbb{R}$. The open balls are just the set of all open intervals, $(x_1, x_2) \subset \mathbb{R}$, for any $x_1, x_2 \in \mathbb{R}$ such that $x_1 < x_2$. ■

Using $\mathcal{B}$, a *measure* μ is now defined as a function $\mu : \mathcal{B} \rightarrow [0, \infty]$ such that the *measure axioms* are satisfied:

1. For the empty set, $\mu(\emptyset) = 0$.
2. For any collection, $E_1, E_2, E_3, \ldots$, of a countable (possibly finite) number of pairwise disjoint, measurable sets, let E denote their union. The measure μ must satisfy

$$\mu(E) = \sum_i \mu(E_i), \tag{5.14}$$

in which i counts over the whole collection.

Example 5.10 (Lebesgue Measure) The most common and important measure is the *Lebesgue measure*, which becomes the standard notions of length in $\mathbb{R}$, area in $\mathbb{R}^2$, and volume in $\mathbb{R}^n$ for $n \geq 3$. One important concept with Lebesgue measure is the existence of sets of *measure zero*. For any countable set A, the Lebesgue measure yields $\mu(A) = 0$. For example, what is the total length of the point $\{1\} \subset \mathbb{R}$? The length of any single point must be zero. To satisfy the measure axioms, sets such as $\{1, 3, 4, 5\}$ must also have measure zero. Even infinite subsets such as $\mathbb{Z}$ and $\mathbb{Q}$ have measure zero in $\mathbb{R}$. If the dimension of a set $A \subseteq \mathbb{R}^n$ is m for some integer $m < n$, then $\mu(A) = 0$, according to the Lebesgue measure on $\mathbb{R}^n$. For example, the set $\mathbb{S}^2 \subset \mathbb{R}^3$ has measure zero because the sphere has no volume. However, if the measure space is restricted to $\mathbb{S}^2$ and then the surface area is defined, then nonzero measure is obtained. ■

Example 5.11 (The Counting Measure) If (X, ρ) is finite, then the *counting measure* can be defined. In this case, the measure can be defined over the entire power set of X. For any $A \subset X$, the counting measure yields $\mu(A) = |A|$, the number of elements in A. Verify that this satisfies the measure axioms. ■

Example 5.12 (Probability Measure) Measure theory even unifies discrete and continuous probability theory. The measure μ can be defined to yield probability mass. The probability axioms (see Section 9.1.2) are consistent with the measure axioms, which therefore yield a measure space. The integrals and sums needed to define expectations of random variables for continuous and discrete cases, respectively, unify into a single measure-theoretic integral. ■

Measure theory can be used to define very general notions of integration that are much more powerful than the Riemann integral that is learned in classical calculus. One of the most important concepts is the *Lebesgue integral*. Instead of being limited to partitioning the domain of integration into intervals, virtually any partition into measurable sets can be used. Its definition requires the notion of a *measurable function* to ensure that the function domain is partitioned into measurable sets. For further study, see [346, 546, 836].

5.1.4 Using the Correct Measure

Since many metrics and measures are possible, it may sometimes seem that there is no "correct" choice. This can be frustrating because the performance of sampling-based planning algorithms can depend strongly on these. Conveniently, there is a

natural measure, called the Haar measure, for some transformation groups, including $SO(N)$. Good metrics also follow from the Haar measure, but unfortunately, there are still arbitrary alternatives.

The basic requirement is that the measure does not vary when the sets are transformed using the group elements. More formally, let G represent a matrix group with real-valued entries, and let μ denote a measure on G. If for any measurable subset $A \subseteq G$, and any element $g \in G$, $\mu(A) = \mu(gA) = \mu(Ag)$, then μ is called the *Haar measure*[2] for G. The notation gA represents the set of all matrices obtained by the product ga, for any $a \in A$. Similarly, Ag represents all products of the form ag.

Example 5.13 (Haar Measure for $SO(2)$) The Haar measure for $SO(2)$ can be obtained by parameterizing the rotations as $[0,1]/\sim$ with 0 and 1 identified, and letting μ be the Lebesgue measure on the unit interval. To see the invariance property, consider the interval $[1/4, 1/2]$, which produces a set $A \subset SO(2)$ of rotation matrices. This corresponds to the set of all rotations from $\theta = \pi/2$ to $\theta = \pi$. The measure yields $\mu(A) = 1/4$. Now consider multiplying every matrix $a \in A$ by a rotation matrix, $g \in SO(2)$, to yield Ag. Suppose g is the rotation matrix for $\theta = \pi$. The set Ag is the set of all rotation matrices from $\theta = 3\pi/2$ up to $\theta = 2\pi = 0$. The measure $\mu(Ag) = 1/4$ remains unchanged. Invariance for gA may be checked similarly. The transformation g translates the intervals in $[0,1]/\sim$. Since the measure is based on interval lengths, it is invariant with respect to translation. Note that μ can be multiplied by a fixed constant (such as 2π) without affecting the invariance property.

An invariant metric can be defined from the Haar measure on $SO(2)$. For any points $x_1, x_2 \in [0,1]$, let $\rho = \mu([x_1, x_2])$, in which $[x_1, x_2]$ is the shortest length (smallest measure) interval that contains x_1 and x_2 as endpoints. This metric was already given in Example 5.2.

To obtain examples that are not the Haar measure, let μ represent probability mass over $[0,1]$ and define any nonuniform probability density function (the uniform density yields the Haar measure). Any shifting of intervals will change the probability mass, resulting in a different measure.

Failing to use the Haar measure weights some parts of $SO(2)$ more heavily than others. Sometimes imposing a bias may be desirable, but it is at least as important to know how to eliminate bias. These ideas may appear obvious, but in the case of $SO(3)$ and many other groups it is more challenging to eliminate this bias and obtain the Haar measure. ■

Example 5.14 (Haar Measure for $SO(3)$) For $SO(3)$ it turns out once again that quaternions come to the rescue. If unit quaternions are used, recall that $SO(3)$ becomes parameterized in terms of $\mathbb{S}^3$, but opposite points are identified.

[2] Such a measure is unique up to scale and exists for any locally compact topological group [346, 836].

It can be shown that the surface area on $\mathbb{S}^3$ is the Haar measure. (Since $\mathbb{S}^3$ is a 3D manifold, it may more appropriately be considered as a surface "volume.") It will be seen in Section 5.2.2 that uniform random sampling over $SO(3)$ must be done with a uniform probability density over $\mathbb{S}^3$. This corresponds exactly to the Haar measure. If instead $SO(3)$ is parameterized with Euler angles, the Haar measure will not be obtained. An unintentional bias will be introduced; some rotations in $SO(3)$ will have more weight than others for no particularly good reason. ■

5.2 Sampling Theory

5.2.1 Motivation and Basic Concepts

The state space for motion planning, $\mathcal{C}$, is uncountably infinite, yet a sampling-based planning algorithm can consider at most a countable number of samples. If the algorithm runs forever, this may be countably infinite, but in practice we expect it to terminate early after only considering a finite number of samples. This mismatch between the cardinality of $\mathcal{C}$ and the set that can be probed by an algorithm motivates careful consideration of sampling techniques. Once the sampling component has been defined, discrete planning methods from Chapter 2 may be adapted to the current setting. Their performance, however, hinges on the way the C-space is sampled.

Since sampling-based planning algorithms are often terminated early, the particular order in which samples are chosen becomes critical. Therefore, a distinction is made between a sample *set* and a sample *sequence*. A unique sample set can always be constructed from a sample sequence, but many alternative sequences can be constructed from one sample set.

Denseness Consider constructing an infinite sample sequence over $\mathcal{C}$. What would be some desirable properties for this sequence? It would be nice if the sequence eventually reached every point in $\mathcal{C}$, but this is impossible because $\mathcal{C}$ is uncountably infinite. Strangely, it is still possible for a sequence to get arbitrarily close to every element of $\mathcal{C}$ (assuming $\mathcal{C} \subseteq \mathbb{R}^m$). In topology, this is the notion of denseness. Let U and V be any subsets of a topological space. The set U is said to be *dense* in V if $\text{cl}(U) = V$ (recall the closure of a set from Section 4.1.1). This means adding the boundary points to U produces V. A simple example is that $(0, 1) \subset \mathbb{R}$ is dense in $[0, 1] \subset \mathbb{R}$. A more interesting example is that the set $\mathbb{Q}$ of rational numbers is both countable and dense in $\mathbb{R}$. Think about why. For any real number, such as $\pi \in \mathbb{R}$, there exists a sequence of fractions that converges to it. This sequence of fractions must be a subset of $\mathbb{Q}$. A sequence (as opposed to a set) is called *dense* if its underlying set is dense. The bare minimum for sampling methods is that they produce a dense sequence. Stronger requirements, such as uniformity and regularity, will be explained shortly.

A random sequence is probably dense Suppose that $\mathcal{C} = [0,1]$. One of the simplest ways conceptually to obtain a dense sequence is to pick points at random. Suppose $I \subset [0,1]$ is an interval of length e. If k samples are chosen independently at random,[3] the probability that none of them falls into I is $(1-e)^k$. As k approaches infinity, this probability converges to zero. This means that the probability that any nonzero-length interval in $[0,1]$ contains no points converges to zero. One small technicality exists. The infinite sequence of independently, randomly chosen points is only dense *with probability one*, which is not the same as being guaranteed. This is one of the strange outcomes of dealing with uncountably infinite sets in probability theory. For example, if a number between $[0,1]$ is chosen at random, the probably that $\pi/4$ is chosen is zero; however, it is still possible. (The probability is just the Lebesgue measure, which is zero for a set of measure zero.) For motion planning purposes, this technicality has no practical implications; however, if k is not very large, then it might be frustrating to obtain only probabilistic assurances, as opposed to absolute guarantees of coverage. The next sequence is guaranteed to be dense because it is deterministic.

The van der Corput sequence A beautiful yet underutilized sequence was published in 1935 by van der Corput, a Dutch mathematician [952]. It exhibits many ideal qualities for applications. At the same time, it is based on a simple idea. Unfortunately, it is only defined for the unit interval. The quest to extend many of its qualities to higher dimensional spaces motivates the formal quality measures and sampling techniques in the remainder of this section.

To explain the van der Corput sequence, let $\mathcal{C} = [0,1]/\sim$, in which $0 \sim 1$, which can be interpreted as $SO(2)$. Suppose that we want to place 16 samples in $\mathcal{C}$. An ideal choice is the set $S = \{i/16 \mid 0 \leq i < 16\}$, which evenly spaces the points at intervals of length $1/16$. This means that no point in $\mathcal{C}$ is further than $1/32$ from the nearest sample. What if we want to make S into a sequence? What is the best ordering? What if we are not even sure that 16 points are sufficient? Maybe 16 is too few or even too many.

The first two columns of Figure 5.2 show a naive attempt at making S into a sequence by sorting the points by increasing value. The problem is that after $i = 8$, half of $\mathcal{C}$ has been neglected. It would be preferable to have a nice covering of $\mathcal{C}$ for any i. Van der Corput's clever idea was to reverse the order of the bits, when the sequence is represented with binary decimals. In the original sequence, the most significant bit toggles only once, whereas the least significant bit toggles in every step. By reversing the bits, the most significant bit toggles in every step, which means that the sequence alternates between the lower and upper halves of $\mathcal{C}$. The third and fourth columns of Figure 5.2 show the original and reversed-order binary representations. The resulting sequence dances around $[0,1]/\sim$ in a nice way, as shown in the last two columns of Figure 5.2. Let $\nu(i)$ denote the ith point of the van der Corput sequence.

[3] See Section 9.1.2 for a review of probability theory.

i	Naive Sequence	Binary	Reverse Binary	Van der Corput	Points in $[0,1]/\sim$
1	0	.0000	.0000	0	
2	1/16	.0001	.1000	1/2	
3	1/8	.0010	.0100	1/4	
4	3/16	.0011	.1100	3/4	
5	1/4	.0100	.0010	1/8	
6	5/16	.0101	.1010	5/8	
7	3/8	.0110	.0110	3/8	
8	7/16	.0111	.1110	7/8	
9	1/2	.1000	.0001	1/16	
10	9/16	.1001	.1001	9/16	
11	5/8	.1010	.0101	5/16	
12	11/16	.1011	.1101	13/16	
13	3/4	.1100	.0011	3/16	
14	13/16	.1101	.1011	11/16	
15	7/8	.1110	.0111	7/16	
16	15/16	.1111	.1111	15/16	

Figure 5.2: The van der Corput sequence is obtained by reversing the bits in the binary decimal representation of the naive sequence.

In contrast to the naive sequence, each $\nu(i)$ lies far away from $\nu(i+1)$. Furthermore, the first i points of the sequence, for any i, provide reasonably uniform coverage of $\mathcal{C}$. When i is a power of 2, the points are perfectly spaced. For other i, the coverage is still good in the sense that the number of points that appear in any interval of length l is roughly il. For example, when $i = 10$, every interval of length $1/2$ contains roughly 5 points.

The length, 16, of the naive sequence is actually not important. If instead 8 is used, the same $\nu(1)$, ..., $\nu(8)$ are obtained. Observe in the reverse binary column of Figure 5.2 that this amounts to removing the last zero from each binary decimal representation, which does not alter their values. If 32 is used for the naive sequence, then the same $\nu(1)$, ..., $\nu(16)$ are obtained, and the sequence continues nicely from $\nu(17)$ to $\nu(32)$. To obtain the van der Corput sequence from $\nu(33)$ to $\nu(64)$, six-bit sequences are reversed (corresponding to the case in which the naive sequence has 64 points). The process repeats to produce an infinite sequence that does not require a fixed number of points to be specified a priori. In addition to the nice uniformity properties for every i, the infinite van der Corput sequence is also dense in $[0,1]/\sim$. This implies that every open subset must contain at least one sample.

You have now seen ways to generate nice samples in a unit interval both randomly and deterministically. Sections 5.2.2–5.2.4 explain how to generate dense samples with nice properties in the complicated spaces that arise in motion plan-

ning.

5.2.2 Random Sampling

Now imagine moving beyond $[0,1]$ and generating a dense sample sequence for any bounded C-space, $\mathcal{C} \subseteq \mathbb{R}^m$. In this section the goal is to generate *uniform random* samples. This means that the probability density function $p(q)$ over $\mathcal{C}$ is uniform. Wherever relevant, it also will mean that the probability density is also consistent with the Haar measure. We will not allow any artificial bias to be introduced by selecting a poor parameterization. For example, picking uniform random Euler angles does *not* lead to uniform random samples over $SO(3)$. However, picking uniform random unit quaternions works perfectly because quaternions use the same parameterization as the Haar measure; both choose points on $\mathbb{S}^3$.

Random sampling is the easiest of all sampling methods to apply to C-spaces. One of the main reasons is that C-spaces are formed from Cartesian products, and independent random samples extend easily across these products. If $X = X_1 \times X_2$, and uniform random samples x_1 and x_2 are taken from X_1 and X_2, respectively, then (x_1, x_2) is a uniform random sample for X. This is very convenient in implementations. For example, suppose the motion planning problem involves 15 robots that each translate for any $(x_t, y_t) \in [0,1]^2$; this yields $\mathcal{C} = [0,1]^{30}$. In this case, 30 points can be chosen uniformly at random from $[0,1]$ and combined into a 30-dimensional vector. Samples generated this way are uniformly randomly distributed over $\mathcal{C}$. Combining samples over Cartesian products is much more difficult for nonrandom (deterministic) methods, which are presented in Sections 5.2.3 and 5.2.4.

Generating a random element of $SO(3)$ One has to be very careful about sampling uniformly over the space of rotations. The probability density must correspond to the Haar measure, which means that a random rotation should be obtained by picking a point at random on $\mathbb{S}^3$ and forming the unit quaternion. An extremely clever way to sample $SO(3)$ uniformly at random is given in [883] and is reproduced here. Choose three points $u_1, u_2, u_3 \in [0,1]$ uniformly at random. A uniform, random quaternion is given by the simple expression

$$h = (\sqrt{1-u_1} \sin 2\pi u_2,\ \sqrt{1-u_1} \cos 2\pi u_2,\ \sqrt{u_1} \sin 2\pi u_3,\ \sqrt{u_1} \cos 2\pi u_3). \quad (5.15)$$

A full explanation of the method is given in [883], and a brief intuition is given here. First drop down a dimension and pick $u_1, u_2 \in [0,1]$ to generate points on $\mathbb{S}^2$. Let u_1 represent the value for the third coordinate, $(0,0,u_1) \in \mathbb{R}^3$. The slice of points on $\mathbb{S}^2$ for which u_1 is fixed for $0 < u_1 < 1$ yields a circle on $\mathbb{S}^2$ that corresponds to some line of latitude on $\mathbb{S}^2$. The second parameter selects the longitude, $2\pi u_2$. Fortunately, the points are uniformly distributed over $\mathbb{S}^2$. Why? Imagine $\mathbb{S}^2$ as the crust on a spherical loaf of bread that is run through a bread slicer. The slices are cut in a direction parallel to the equator and are of equal thickness. The crusts of each slice have equal area; therefore, the points are uniformly distributed. The

method proceeds by using that fact that $\mathbb{S}^3$ can be partitioned into a spherical arrangement of circles (known as the Hopf fibration); there is an $\mathbb{S}^1$ copy for each point in $\mathbb{S}^2$. The method above is used to provide a random point on $\mathbb{S}^2$ using u_2 and u_3, and u_1 produces a random point on $\mathbb{S}^1$; they are carefully combined in (5.15) to yield a random rotation. To respect the antipodal identification for rotations, any quaternion h found in the lower hemisphere (i.e., $a < 0$) can be negated to yield $-h$. This does not distort the uniform random distribution of the samples.

Generating random directions Some sampling-based algorithms require choosing motion directions at random.[4] From a configuration q, the possible directions of motion can be imagined as being distributed around a sphere. In an $(n+1)$-dimensional C-space, this corresponds to sampling on $\mathbb{S}^n$. For example, choosing a direction in $\mathbb{R}^2$ amounts to picking an element of $\mathbb{S}^1$; this can be parameterized as $\theta \in [0, 2\pi]/\sim$. If $n = 4$, then the previously mentioned trick for $SO(3)$ should be used. If $n = 3$ or $n > 4$, then samples can be generated using a slightly more expensive method that exploits spherical symmetries of the multidimensional Gaussian density function [341]. The method is explained for $\mathbb{R}^{n+1}$; boundaries and identifications must be taken into account for other spaces. For each of the $n+1$ coordinates, generate a sample u_i from a zero-mean Gaussian distribution with the same variance for each coordinate. Following from the Central Limit Theorem, u_i can be approximately obtained by generating k samples at random over $[-1, 1]$ and adding them ($k \geq 12$ is usually sufficient in practice). The vector $(u_1, u_2, \ldots, u_{n+1})$ gives a random direction in $\mathbb{R}^{n+1}$ because each u_i was obtained independently, and the level sets of the resulting probability density function are spheres. We did not use uniform random samples for each u_i because this would bias the directions toward the corners of a cube; instead, the Gaussian yields spherical symmetry. The final step is to normalize the vector by taking $u_i/\|u\|$ for each coordinate.

Pseudorandom number generation Although there are advantages to uniform random sampling, there are also several disadvantages. This motivates the consideration of deterministic alternatives. Since there are trade-offs, it is important to understand how to use both kinds of sampling in motion planning. One of the first issues is that computer-generated numbers are not random.[5] A *pseudorandom number generator* is usually employed, which is a deterministic method that simulates the behavior of randomness. Since the samples are not truly random, the advantage of extending the samples over Cartesian products does not necessarily hold. Sometimes problems are caused by unforeseen deterministic dependencies. One of the best pseudorandom number generators for avoiding such

[4] The directions will be formalized in Section 8.3.2 when smooth manifolds are introduced. In that case, the directions correspond to the set of possible velocities that have unit magnitude. Presently, the notion of a direction is only given informally.

[5] There are exceptions, which use physical phenomena as a random source [808].

troubles is the Mersenne twister [684], for which implementations can be found on the Internet.

To help see the general difficulties, the classical *linear congruential* pseudorandom number generator is briefly explained [619, 738]. The method uses three integer parameters, M, a, and c, which are chosen by the user. The first two, M and a, must be relatively prime, meaning that $\gcd(M, a) = 1$. The third parameter, c, must be chosen to satisfy $0 \leq c < M$. Using modular arithmetic, a sequence can be generated as

$$y_{i+1} = ay_i + c \mod M, \tag{5.16}$$

by starting with some arbitrary seed $1 \leq y_0 \leq M$. Pseudorandom numbers in $[0, 1]$ are generated by the sequence

$$x_i = y_i/M. \tag{5.17}$$

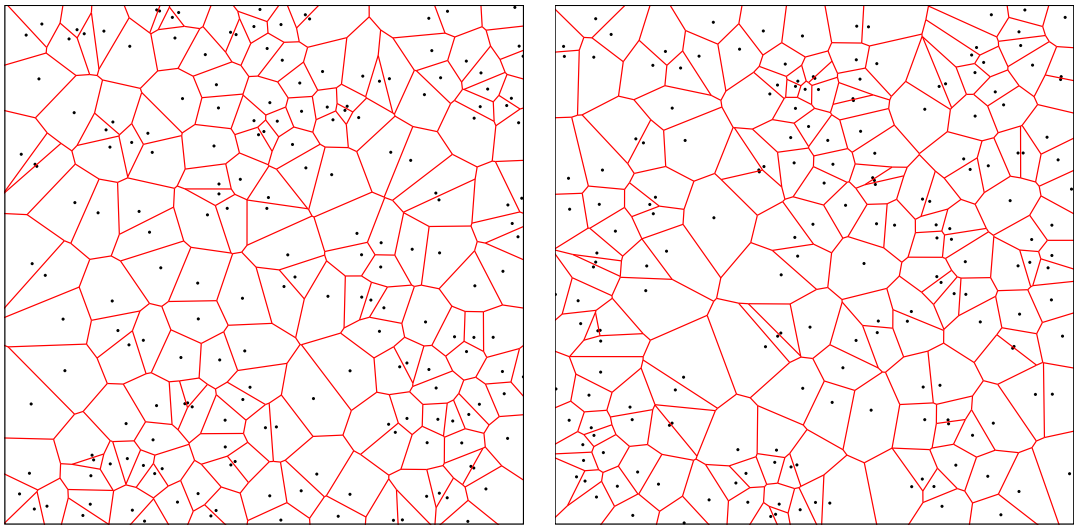
The sequence is periodic; therefore, M is typically very large (e.g., $M = 2^{31} - 1$). Due to periodicity, there are potential problems of regularity appearing in the samples, especially when applied across a Cartesian product to generate points in $\mathbb{R}^n$. Particular values must be chosen for the parameters, and statistical tests are used to evaluate the samples either experimentally or theoretically [738].

Testing for randomness Thus, it is important to realize that even the "random" samples are deterministic. They are designed to optimize performance on statistical tests. Many sophisticated statistical tests of uniform randomness are used. One of the simplest, the *chi-square test*, is described here. This test measures how far computed statistics are from their expected value. As a simple example, suppose $\mathcal{C} = [0, 1]^2$ and is partitioned into a 10 by 10 array of 100 square boxes. If a set P of k samples is chosen at random, then intuitively each box should receive roughly $k/100$ of the samples. An error function can be defined to measure how far from true this intuition is:

$$e(P) = \sum_{i=1}^{100} (b_i - k/100)^2, \tag{5.18}$$

in which b_i is the number of samples that fall into box i. It is shown [521] that $e(P)$ follows a *chi-squared* distribution. A surprising fact is that the goal is not to minimize $e(P)$. If the error is too small, we would declare that the samples are too uniform to be random! Imagine $k = 1,000,000$ and exactly $10,000$ samples appear in each of the 100 boxes. This yields $e(P) = 0$, but how likely is this to ever occur? The error must generally be larger (it appears in many statistical tables) to account for the irregularity due to randomness.

This irregularity can be observed in terms of *Voronoi diagrams*, as shown in Figure 5.3. The Voronoi diagram partitions $\mathbb{R}^2$ into regions based on the samples. Each sample x has an associated *Voronoi region* $\mathrm{Vor}(x)$. For any point $y \in \mathrm{Vor}(x)$, x is the closest sample to y using Euclidean distance. The different sizes and shapes



(a) 196 pseudorandom samples (b) 196 pseudorandom samples

Figure 5.3: Irregularity in a collection of (pseudo)random samples can be nicely observed with Voronoi diagrams.

of these regions give some indication of the required irregularity of random sampling. This irregularity may be undesirable for sampling-based motion planning and is somewhat repaired by the deterministic sampling methods of Sections 5.2.3 and 5.2.4 (however, these methods also have drawbacks).

5.2.3 Low-Dispersion Sampling

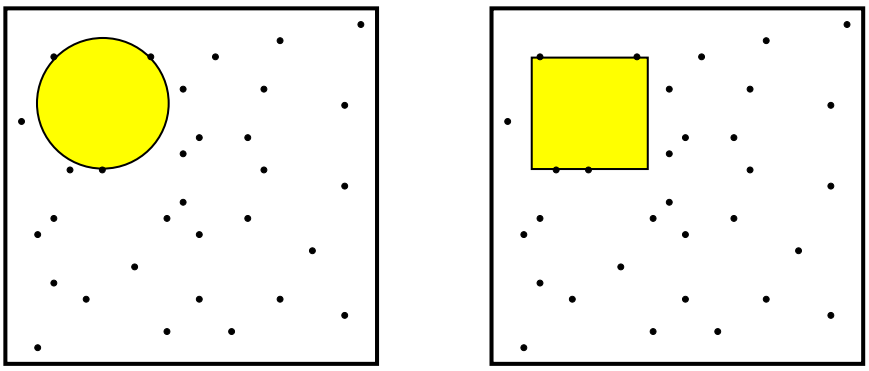
This section describes an alternative to random sampling. Here, the goal is to optimize a criterion called *dispersion* [738]. Intuitively, the idea is to place samples in a way that makes the largest uncovered area be as small as possible. This generalizes of the idea of *grid resolution*. For a grid, the *resolution* may be selected by defining the step size for each axis. As the step size is decreased, the resolution increases. If a grid-based motion planning algorithm can increase the resolution arbitrarily, it becomes resolution complete. Using the concepts in this section, it may instead reduce its dispersion arbitrarily to obtain a resolution complete algorithm. Thus, dispersion can be considered as a powerful generalization of the notion of "resolution."

Dispersion definition The *dispersion*[6] of a finite set P of samples in a metric space (X, ρ) is[7]

$$\delta(P) = \sup_{x \in X} \left\{ \min_{p \in P} \left\{ \rho(x, p) \right\} \right\}. \tag{5.19}$$

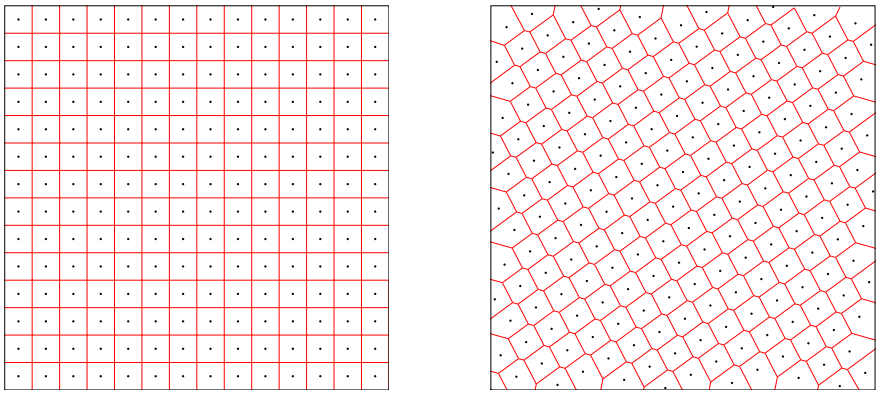
[6]The definition is unfortunately backward from intuition. Lower dispersion means that the points are nicely dispersed. Thus, more dispersion is bad, which is counterintuitive.

[7]The sup represents the *supremum*, which is the least upper bound. If X is closed, then the sup becomes a max. See Section 9.1.1 for more details.



(a) L_2 dispersion (b) L_∞ dispersion

Figure 5.4: Reducing the dispersion means reducing the radius of the largest empty ball.



(a) 196-point Sukharev grid (b) 196 lattice points

Figure 5.5: The Sukharev grid and a nongrid lattice.

Figure 5.4 gives an interpretation of the definition for two different metrics. An alternative way to consider dispersion is as the radius of the largest empty ball (for the L_∞ metric, the balls are actually cubes). Note that at the boundary of X (if it exists), the empty ball becomes truncated because it cannot exceed the boundary. There is also a nice interpretation in terms of Voronoi diagrams. Figure 5.3 can be used to help explain L_2 dispersion in $\mathbb{R}^2$. The *Voronoi vertices* are the points at which three or more Voronoi regions meet. These are points in $\mathcal{C}$ for which the nearest sample is far. An open, empty disc can be placed at any Voronoi vertex, with a radius equal to the distance to the three (or more) closest samples. The radius of the largest disc among those placed at all Voronoi vertices is the dispersion. This interpretation also extends nicely to higher dimensions.

Making good grids Optimizing dispersion forces the points to be distributed more uniformly over $\mathcal{C}$. This causes them to fail statistical tests, but the point distribution is often better for motion planning purposes. Consider the best way to reduce dispersion if ρ is the L_∞ metric and $X = [0,1]^n$. Suppose that the number of samples, k, is given. Optimal dispersion is obtained by partitioning $[0,1]$ into a grid of cubes and placing a point at the center of each cube, as shown for $n = 2$ and $k = 196$ in Figure 5.5a. The number of cubes per axis must be $\lfloor k^{\frac{1}{n}} \rfloor$, in which $\lfloor \cdot \rfloor$ denotes the *floor*. If $k^{\frac{1}{n}}$ is not an integer, then there are leftover points that may be placed anywhere without affecting the dispersion. Notice that $k^{\frac{1}{n}}$ just gives the number of points per axis for a grid of k points in n dimensions. The resulting grid will be referred to as a *Sukharev grid* [922].

The dispersion obtained by the Sukharev grid is the best possible. Therefore, a useful lower bound can be given for *any* set P of k samples [922]:

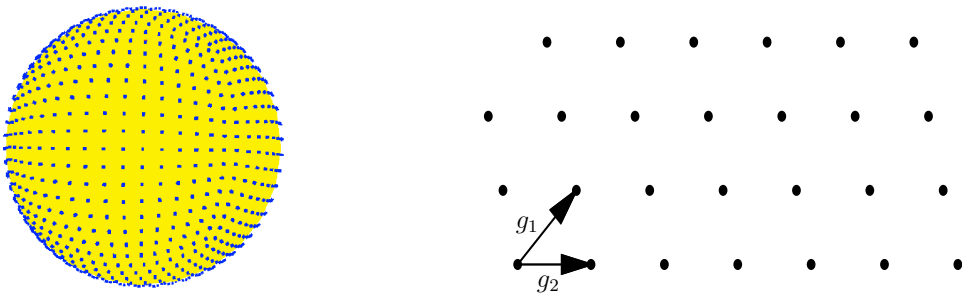
$$\delta(P) \geq \frac{1}{2\lfloor k^{\frac{1}{d}} \rfloor}. \tag{5.20}$$

This implies that keeping the dispersion fixed *requires* exponentially many points in the dimension, d.

At this point you might wonder why L_∞ was used instead of L_2, which seems more natural. This is because the L_2 case is extremely difficult to optimize (except in $\mathbb{R}^2$, where a tiling of equilateral triangles can be made, with a point in the center of each one). Even the simple problem of determining the best way to distribute a fixed number of points in $[0,1]^3$ is unsolved for most values of k. See [241] for extensive treatment of this problem.

Suppose now that other topologies are considered instead of $[0,1]^n$. Let $X = [0,1]/\sim$, in which the identification produces a torus. The situation is quite different because X no longer has a boundary. The Sukharev grid still produces optimal dispersion, but it can also be shifted without increasing the dispersion. In this case, a *standard grid* may also be used, which has the same number of points as the Sukharev grid but is translated to the origin. Thus, the first grid point is $(0,0)$, which is actually the same as $2^n - 1$ other points by identification. If X represents a cylinder and the number of points, k, is given, then it is best to just use the Sukharev grid. It is possible, however, to shift each coordinate that behaves like $\mathbb{S}^1$. If X is rectangular but not a square, a good grid can still be made by tiling the space with cubes. In some cases this will produce optimal dispersion. For complicated spaces such as $SO(3)$, no grid exists in the sense defined so far. It is possible, however, to generate grids on the faces of an inscribed Platonic solid [251] and lift the samples to $\mathbb{S}^n$ with relatively little distortion [987]. For example, to sample $\mathbb{S}^2$, Sukharev grids can be placed on each face of a cube. These are lifted to obtain the warped grid shown in Figure 5.6a.

Example 5.15 (Sukharev Grid) Suppose that $n = 2$ and $k = 9$. If $X = [0,1]^2$, then the Sukharev grid yields points for the nine cases in which either coordinate may be $1/6$, $1/2$, or $5/6$. The L_∞ dispersion is $1/6$. The spacing between the points



(a) (b)

Figure 5.6: (a) A distorted grid can even be placed over spheres and $SO(3)$ by putting grids on the faces of an inscribed cube and lifting them to the surface [987]. (b) A lattice can be considered as a grid in which the generators are not necessarily orthogonal.

along each axis is $1/3$, which is twice the dispersion. If instead $X = [0,1]^2/\sim$, which represents a torus, then the nine points may be shifted to yield the standard grid. In this case each coordinate may be 0, $1/3$, or $2/3$. The dispersion and spacing between the points remain unchanged. ■

One nice property of grids is that they have a lattice structure. This means that neighboring points can be obtained very easily be adding or subtracting vectors. Let g_j be an n-dimensional vector called a *generator*. A point on a lattice can be expressed as

$$x = \sum_{j=1}^{n} k_j g_j \tag{5.21}$$

for n independent generators, as depicted in Figure 5.6b. In a 2D grid, the generators represent "up" and "right." If $X = [0,100]^2$ and a standard grid with integer spacing is used, then the neighbors of the point $(50,50)$ are obtained by adding $(0,1)$, $(0,-1)$, $(-1,0)$, or $(1,0)$. In a general lattice, the generators need not be orthogonal. An example is shown in Figure 5.5b. In Section 5.4.2, lattice structure will become important and convenient for defining the search graph.

Infinite grid sequences Now suppose that the number, k, of samples is not given. The task is to define an infinite sequence that has the nice properties of the van der Corput sequence but works for any dimension. This will become the notion of a *multi-resolution grid*. The resolution can be iteratively doubled. For a multi-resolution standard grid in $\mathbb{R}^n$, the sequence will first place one point at the origin. After 2^n points have been placed, there will be a grid with two points per axis. After 4^n points, there will be four points per axis. Thus, after 2^{ni} points for any positive integer i, a grid with 2^i points per axis will be represented. If only complete grids are allowed, then it becomes clear why they appear inappropriate

for high-dimensional problems. For example, if $n = 10$, then full grids appear after 1, 2^{10}, 2^{20}, 2^{30}, and so on, samples. Each doubling in resolution multiplies the number of points by 2^n. Thus, to use grids in high dimensions, one must be willing to accept *partial grids* and define an infinite sequence that places points in a nice way.

The van der Corput sequence can be extended in a straightforward way as follows. Suppose $X = \mathbb{T}^2 = [0,1]^2/\sim$. The original van der Corput sequence started by counting in binary. The least significant bit was used to select which half of $[0,1]$ was sampled. In the current setting, the two least significant bits can be used to select the quadrant of $[0,1]^2$. The next two bits can be used to select the quadrant within the quadrant. This procedure continues recursively to obtain a complete grid after $k = 2^{2i}$ points, for any positive integer i. For any k, however, there is only a partial grid. The points are distributed with optimal L_∞ dispersion. This same idea can be applied in dimension n by using n bits at a time from the binary sequence to select the orthant (n-dimensional quadrant). Many other orderings produce L_∞-optimal dispersion. Selecting orderings that additionally optimize other criteria, such as discrepancy or L_2 dispersion, are covered in [639, 644]. Unfortunately, it is more difficult to make a multi-resolution Sukharev grid. The base becomes 3 instead of 2; after every 3^{ni} points a complete grid is obtained. For example, in one dimension, the first point appears at $1/2$. The next two points appear at $1/6$ and $5/6$. The next complete one-dimensional grid appears after there are 9 points.

Dispersion bounds Since the sample sequence is infinite, it is interesting to consider asymptotic bounds on dispersion. It is known that for $X = [0,1]^n$ and any L_p metric, the best possible asymptotic dispersion is $O(k^{-1/n})$ for k points and n dimensions [738]. In this expression, k is the variable in the limit and n is treated as a constant. Therefore, any function of n may appear as a constant (i.e., $O(f(n)k^{-1/n}) = O(k^{-1/n})$ for any positive $f(n)$). An important practical consideration is the size of $f(n)$ in the asymptotic analysis. For example, for the van der Corput sequence from Section 5.2.1, the dispersion is bounded by $1/k$, which means that $f(n) = 1$. This does not seem good because for values of k that are powers of two, the dispersion is $1/2k$. Using a multi-resolution Sukharev grid, the constant becomes $3/2$ because it takes a longer time before a full grid is obtained. Nongrid, low-dispersion infinite sequences exist that have $f(n) = 1/\ln 4$ [738]; these are not even uniformly distributed, which is rather surprising.

5.2.4 Low-Discrepancy Sampling

In some applications, selecting points that align with the coordinate axis may be undesirable. Therefore, extensive sampling theory has been developed to determine methods that avoid alignments while distributing the points uniformly. In sampling-based motion planning, grids sometimes yield unexpected behavior because a row of points may align nicely with a corridor in $\mathcal{C}_{free}$. In some cases, a

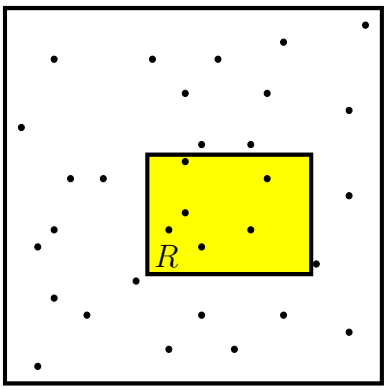


Figure 5.7: Discrepancy measures whether the right number of points fall into boxes. It is related to the chi-square test but optimizes over all possible boxes.

solution is obtained with surprisingly few samples, and in others, too many samples are necessary. These alignment problems, when they exist, generally drive the variance higher in computation times because it is difficult to predict when they will help or hurt. This provides motivation for developing sampling techniques that try to reduce this sensitivity.

Discrepancy theory and its corresponding sampling methods were developed to avoid these problems for numerical integration [738]. Let X be a measure space, such as $[0,1]^n$. Let $\mathcal{R}$ be a collection of subsets of X that is called a *range space*. In most cases, $\mathcal{R}$ is chosen as the set of all axis-aligned rectangular subsets; hence, this will be assumed from this point onward. With respect to a particular point set, P, and range space, $\mathcal{R}$, the *discrepancy* [965] for k samples is defined as (see Figure 5.7)

$$D(P,\mathcal{R}) = \sup_{R \in \mathcal{R}} \left\{ \left| \frac{|P \cap R|}{k} - \frac{\mu(R)}{\mu(X)} \right| \right\}, \tag{5.22}$$

in which $|P \cap R|$ denotes the number of points in $P \cap R$. Each term in the supremum considers how well P can be used to estimate the volume of R. For example, if $\mu(R)$ is $1/5$, then we would hope that about $1/5$ of the points in P fall into R. The discrepancy measures the largest volume estimation error that can be obtained over all sets in $\mathcal{R}$.

Asymptotic bounds There are many different asymptotic bounds for discrepancy, depending on the particular range space and measure space [682]. The most widely referenced bounds are based on the standard range space of axis-aligned rectangular boxes in $[0,1]^n$. There are two different bounds, depending on whether the number of points, k, is given. The best possible asymptotic discrepancy for a single sequence is $O(k^{-1} \log^n k)$. This implies that k is not specified. If, however, for every k a new set of points can be chosen, then the best possible discrepancy is $O(k^{-1} \log^{n-1} k)$. This bound is lower because it considers the best that can be achieved by a sequence of points sets, in which every point set may be completely

different. In a single sequence, the next set must be extended from the current set by adding a single sample.

Relating dispersion and discrepancy Since balls have positive volume, there is a close relationship between discrepancy, which is measure-based, and dispersion, which is metric-based. For example, for any $P \subset [0,1]^n$,

$$\delta(P, L_\infty) \leq D(P, \mathcal{R})^{1/d}, \tag{5.23}$$

which means low-discrepancy implies low-dispersion. Note that the converse is not true. An axis-aligned grid yields high discrepancy because of alignments with the boundaries of sets in $\mathcal{R}$, but the dispersion is very low. Even though low-discrepancy implies low-dispersion, lower dispersion can usually be obtained by ignoring discrepancy (this is one less constraint to worry about). Thus, a trade-off must be carefully considered in applications.

Low-discrepancy sampling methods Due to the fundamental importance of numerical integration and the intricate link between discrepancy and integration error, most sampling literature has led to low-discrepancy sequences and point sets [738, 893, 937]. Although motion planning is quite different from integration, it is worth evaluating these carefully constructed and well-analyzed samples. Their potential use in motion planning is no less reasonable than using pseudorandom sequences, which were also designed with a different intention in mind (satisfying statistical tests of randomness).

Low-discrepancy sampling methods can be divided into three categories: 1) Halton/Hammersley sampling; 2) (t,s)-sequences and (t,m,s)-nets; and 3) lattices. The first category represents one of the earliest methods, and is based on extending the van der Corput sequence. The *Halton sequence* is an n-dimensional generalization of the van der Corput sequence, but instead of using binary representations, a different basis is used for each coordinate [430]. The result is a reasonable deterministic replacement for random samples in many applications. The resulting discrepancy (and dispersion) is lower than that for random samples (with high probability). Figure 5.8a shows the first 196 Halton points in $\mathbb{R}^2$.

Choose n relatively prime integers $p_1, p_2, \ldots, p_n$ (usually the first n primes, $p_1 = 2$, $p_2 = 3$, ..., are chosen). To construct the ith sample, consider the base-p representation for i, which takes the form $i = a_0 + pa_1 + p^2a_2 + p^3a_3 + \ldots$. The following point in $[0,1]$ is obtained by reversing the order of the bits and moving the decimal point (as was done in Figure 5.2):

$$r(i,p) = \frac{a_0}{p} + \frac{a_1}{p^2} + \frac{a_2}{p^3} + \frac{a_3}{p^4} + \cdots . \tag{5.24}$$

For $p = 2$, this yields the ith point in the van der Corput sequence. Starting from $i = 0$, the ith sample in the Halton sequence is

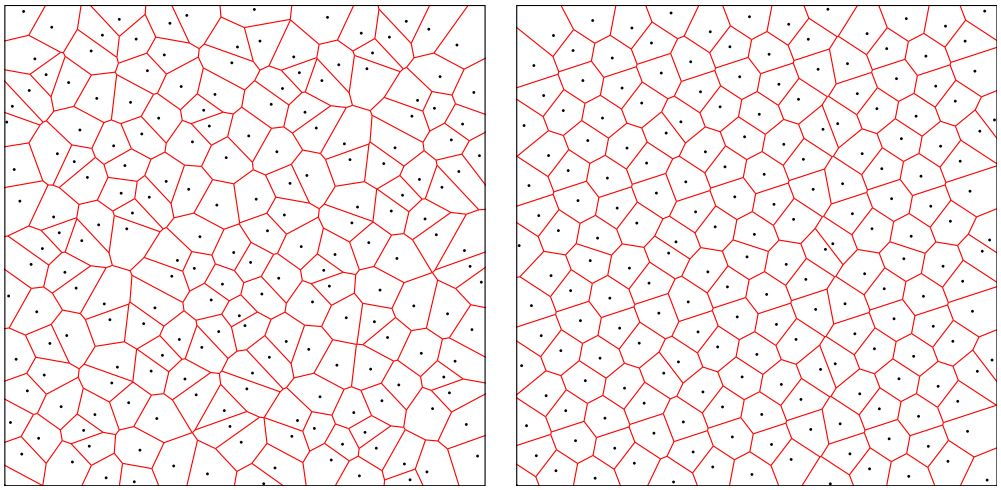
$$\big(r(i,p_1), r(i,p_2), \ldots, r(i,p_n)\big). \tag{5.25}$$

Suppose instead that k, the required number of points, is known. In this case, a better distribution of samples can be obtained. The *Hammersley point set* [431] is an adaptation of the Halton sequence. Using only $d-1$ distinct primes and starting at $i=0$, the ith sample in a Hammersley point set with k elements is

$$\big(i/k, r(i,p_1), \ldots, r(i,p_{n-1})\big). \tag{5.26}$$

Figure 5.8b shows the Hammersley set for $n=2$ and $k=196$.

The construction of Halton/Hammersley samples is simple and efficient, which has led to widespread application. They both achieve asymptotically optimal discrepancy; however, the constant in their asymptotic analysis increases more than exponentially with dimension [738]. The constant for the dispersion also increases exponentially, which is much worse than for the methods of Section 5.2.3.



(a) 196 Halton points (b) 196 Hammersley points

Figure 5.8: The Halton and Hammersley points are easy to construct and provide a nice alternative to random sampling that achieves more regularity. Compare the Voronoi regions to those in Figure 5.3. Beware that although these sequences produce asymptotically optimal discrepancy, their performance degrades substantially in higher dimensions (e.g., beyond 10).

Improved constants are obtained for sequences and finite points by using (t,s)-sequences, and (t,m,s)-nets, respectively [738]. The key idea is to enforce zero discrepancy over particular subsets of $\mathcal{R}$ known as *canonical rectangles*, and all remaining ranges in $\mathcal{R}$ will contribute small amounts to discrepancy. The most famous and widely used (t,s)-sequences are Sobol' and Faure (see [738]). The Niederreiter-Xing (t,s)-sequence has the best-known asymptotic constant, $(a/n)^n$, in which a is a small positive constant [739].

The third category is *lattices*, which can be considered as a generalization of grids that allows nonorthogonal axes [682, 893, 959]. As an example, consider

Figure 5.5b, which shows 196 lattice points generated by the following technique. Let α be a positive irrational number. For a fixed k, generate the ith point according to $(i/k, \{i\alpha\})$, in which $\{\cdot\}$ denotes the fractional part of the real value (modulo-one arithmetic). In Figure 5.5b, $\alpha = (\sqrt{5}+1)/2$, the *golden ratio*. This procedure can be generalized to n dimensions by picking $n-1$ distinct irrational numbers. A technique for choosing the α parameters by using the roots of irreducible polynomials is discussed in [682]. The ith sample in the lattice is

$$\left(\frac{i}{k}, \{i\alpha_1\}, \ldots, \{i\alpha_{n-1}\} \right). \tag{5.27}$$

Recent analysis shows that some lattice sets achieve asymptotic discrepancy that is very close to that of the best-known nonlattice sample sets [445, 938]. Thus, restricting the points to lie on a lattice seems to entail little or no loss in performance, but has the added benefit of a regular neighborhood structure that is useful for path planning. Historically, lattices have required the specification of k in advance; however, there has been increasing interest in extensible lattices, which are infinite sequences [446, 938].

5.3 Collision Detection

Once it has been decided where the samples will be placed, the next problem is to determine whether the configuration is in collision. Thus, collision detection is a critical component of sampling-based planning. Even though it is often treated as a black box, it is important to study its inner workings to understand the information it provides and its associated computational cost. In most motion planning applications, the majority of computation time is spent on collision checking.

A variety of collision detection algorithms exist, ranging from theoretical algorithms that have excellent computational complexity to heuristic, practical algorithms whose performance is tailored to a particular application. The techniques from Section 4.3 can be used to develop a collision detection algorithm by defining a logical predicate using the geometric model of $\mathcal{C}_{obs}$. For the case of a 2D world with a convex robot and obstacle, this leads to an linear-time collision detection algorithm. In general, however, it can be determined whether a configuration is in collision more efficiently by avoiding the full construction of $\mathcal{C}_{obs}$.

5.3.1 Basic Concepts

As in Section 3.1.1, collision detection may be viewed as a logical predicate. In the current setting it appears as $\phi : \mathcal{C} \rightarrow \{\text{TRUE}, \text{FALSE}\}$, in which the domain is $\mathcal{C}$ instead of $\mathcal{W}$. If $q \in \mathcal{C}_{obs}$, then $\phi(q) = \text{TRUE}$; otherwise, $\phi(q) = \text{FALSE}$.

Distance between two sets For the Boolean-valued function ϕ, there is no information about how far the robot is from hitting the obstacles. Such information is very important in planning algorithms. A *distance function* provides this

information and is defined as $d : \mathcal{C} \rightarrow [0, \infty)$, in which the real value in the range of f indicates the distance in the world, $\mathcal{W}$, between the closest pair of points over all pairs from $\mathcal{A}(q)$ and $\mathcal{O}$. In general, for two closed, bounded subsets, E and F, of $\mathbb{R}^n$, the *distance* is defined as

$$\rho(E, F) = \min_{e \in E} \left\{ \min_{f \in F} \left\{ \| e - f \| \right\} \right\}, \tag{5.28}$$

in which $\| \cdot \|$ is the Euclidean norm. Clearly, if $E \cap F \neq \emptyset$, then $\rho(E, F) = 0$. The methods described in this section may be used to either compute distance or only determine whether $q \in \mathcal{C}_{obs}$. In the latter case, the computation is often much faster because less information is required.

Two-phase collision detection Suppose that the robot is a collection of m attached links, $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m$, and that $\mathcal{O}$ has k connected components. For this complicated situation, collision detection can be viewed as a two-phase process.

1. **Broad Phase:** In the *broad phase*, the task is to avoid performing expensive computations for bodies that are far away from each other. Simple bounding boxes can be placed around each of the bodies, and simple tests can be performed to avoid costly collision checking unless the boxes overlap. Hashing schemes can be employed in some cases to greatly reduce the number of pairs of boxes that have to be tested for overlap [703]. For a robot that consists of multiple bodies, the pairs of bodies that should be considered for collision must be specified in advance, as described in Section 4.3.1.

2. **Narrow Phase:** In the *narrow phase*, individual pairs of bodies are each checked carefully for collision. Approaches to this phase are described in Sections 5.3.2 and 5.3.3.

5.3.2 Hierarchical Methods

In this section, suppose that two complicated, nonconvex bodies, E and F, are to be checked for collision. Each body could be part of either the robot or the obstacle region. They are subsets of $\mathbb{R}^2$ or $\mathbb{R}^3$, defined using any kind of geometric primitives, such as triangles in $\mathbb{R}^3$. *Hierarchical methods* generally decompose each body into a tree. Each vertex in the tree represents a *bounding region* that contains some subset of the body. The bounding region of the root vertex contains the whole body.

There are generally two opposing criteria that guide the selection of the type of bounding region:

1. The region should fit the intended body points as tightly as possible.

2. The intersection test for two regions should be as efficient as possible.

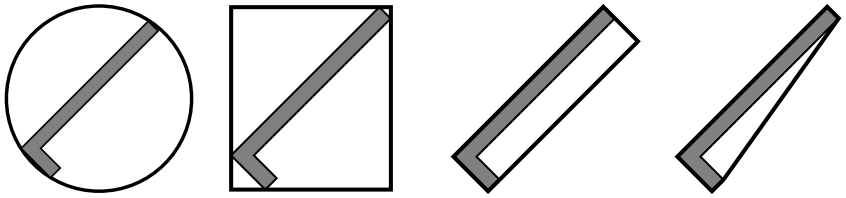


Figure 5.9: Four different kinds of bounding regions: (a) sphere, (b) axis-aligned bounding box (AABB), (c) oriented bounding box (OBB), and (d) convex hull. Each usually provides a tighter approximation than the previous one but is more expensive to test for overlapping pairs.

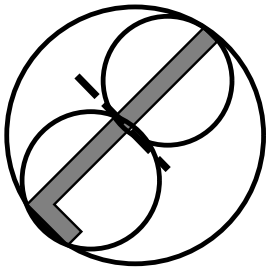


Figure 5.10: The large circle shows the bounding region for a vertex that covers an L-shaped body. After performing a split along the dashed line, two smaller circles are used to cover the two halves of the body. Each circle corresponds to a child vertex.

Several popular choices are shown in Figure 5.9 for an L-shaped body.

The tree is constructed for a body, E (or alternatively, F) recursively as follows. For each vertex, consider the set X of all points in E that are contained in the bounding region. Two child vertices are constructed by defining two smaller bounding regions whose union covers X. The split is made so that the portion covered by each child is of similar size. If the geometric model consists of primitives such as triangles, then a split could be made to separate the triangles into two sets of roughly the same number of triangles. A bounding region is then computed for each of the children. Figure 5.10 shows an example of a split for the case of an L-shaped body. Children are generated recursively by making splits until very simple sets are obtained. For example, in the case of triangles in space, a split is made unless the vertex represents a single triangle. In this case, it is easy to test for the intersection of two triangles.

Consider the problem of determining whether bodies E and F are in collision. Suppose that the trees T_e and T_f have been constructed for E and F, respectively. If the bounding regions of the root vertices of T_e and T_f do not intersect, then it is known that T_e and T_f are not in collision without performing any additional computation. If the bounding regions do intersect, then the bounding regions of

the children of T_e are compared to the bounding region of T_f. If either of these intersect, then the bounding region of T_f is replaced with the bounding regions of its children, and the process continues recursively. As long as the bounding regions overlap, lower levels of the trees are traversed, until eventually the leaves are reached. If triangle primitives are used for the geometric models, then at the leaves the algorithm tests the individual triangles for collision, instead of bounding regions. Note that as the trees are traversed, if a bounding region from the vertex v_1 of T_e does not intersect the bounding region from a vertex, v_2, of T_f, then no children of v_1 have to be compared to children of v_1. Usually, this dramatically reduces the number of comparisons, relative in a naive approach that tests all pairs of triangles for intersection.

It is possible to extend the hierarchical collision detection scheme to also compute distance. The closest pair of points found so far serves as an upper bound that prunes aways some future pairs from consideration. If a pair of bounding regions has a distance greater than the smallest distance computed so far, then their children do not have to be considered [638]. In this case, an additional requirement usually must be imposed. Every bounding region must be a proper subset of its parent bounding region [807]. If distance information is not needed, then this requirement can be dropped.

5.3.3 Incremental Methods

This section focuses on a particular approach called *incremental distance computation*, which assumes that between successive calls to the collision detection algorithm, the bodies move only a small amount. Under this assumption the algorithm achieves "almost constant time" performance for the case of convex polyhedral bodies [636, 702]. Nonconvex bodies can be decomposed into convex components.

These collision detection algorithms seem to offer wonderful performance, but this comes at a price. The models must be *coherent*, which means that all of the primitives must fit together nicely. For example, if a 2D model uses line segments, all of the line segments must fit together perfectly to form polygons. There can be no isolated segments or chains of segments. In three dimensions, polyhedral models are required to have all faces come together perfectly to form the boundaries of 3D shapes. The model cannot be an arbitrary collection of 3D triangles.

The method will be explained for the case of 2D convex polygons, which are interpreted as convex subsets of $\mathbb{R}^2$. Voronoi regions for a convex polygon will be defined in terms of features. The *feature set* is the set of all vertices and edges of a convex polygon. Thus, a polygon with n edges has $2n$ features. Any point outside of the polygon has a closest feature in terms of Euclidean distance. For a given feature, F, the set of all points in $\mathbb{R}^2$ from which F is the closest feature is called the Voronoi region of F and is denoted $\mathrm{Vor}(F)$. Figure 5.11 shows all ten Voronoi regions for a pentagon. Each feature is considered as a point set in the discussion below.

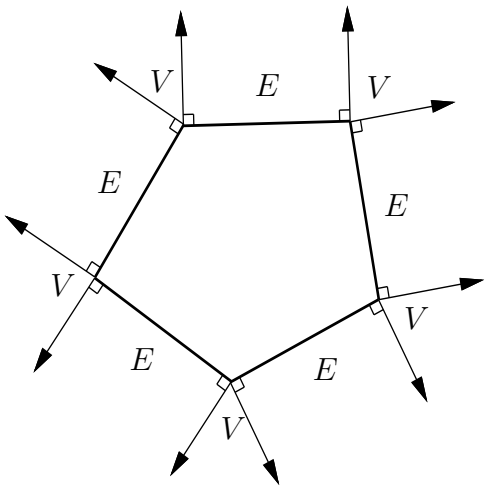


Figure 5.11: The Voronoi regions alternate between being edge-based and vertex-based. The Voronoi regions of vertices are labeled with a "V" and the Voronoi regions of edges are labeled with an "E."

For any two convex polygons that do not intersect, the closest point is determined by a pair of points, one on each polygon (the points are unique, except in the case of parallel edges). Consider the feature for each point in the closest pair. There are only three possible combinations:

- **Vertex-Vertex** Each point of the closest pair is a vertex of a polygon.
- **Edge-Vertex** One point of the closest pair lies on an edge, and the other lies on a vertex.
- **Edge-Edge** Each point of the closest pair lies on an edge. In this case, the edges must be parallel.

Let P_1 and P_2 be two convex polygons, and let F_1 and F_2 represent any feature pair, one from each polygon. Let $(x_1, y_1) \in F_1$ and $(x_2, y_2) \in F_2$ denote the closest pair of points, among all pairs of points in F_1 and F_2, respectively. The following condition implies that the distance between (x_1, y_1) and (x_2, y_2) is the distance between P_1 and P_2:

$$(x_1, y_1) \in \text{Vor}(F_2) \text{ and } (x_2, y_2) \in \text{Vor}(F_1). \tag{5.29}$$

If (5.29) is satisfied for a given feature pair, then the distance between P_1 and P_2 equals the distance between F_1 and F_2. This implies that the distance between P_1 and P_2 can be determined in constant time. The assumption that P_1 moves only a small amount relative to P_2 is made to increase the likelihood that the closest feature pair remains the same. This is why the phrase "almost constant time" is used to describe the performance of the algorithm. Of course, it is possible that the closest feature pair will change. In this case, neighboring features are tested

using the condition above until the new closest pair of features is found. In this worst case, this search could be costly, but this violates the assumption that the bodies do not move far between successive collision detection calls.

The 2D ideas extend to 3D convex polyhedra [247, 636, 702]. The primary difference is that three kinds of features are considered: faces, edges, and vertices. The cases become more complicated, but the idea is the same. Once again, the condition regarding mutual Voronoi regions holds, and the resulting incremental collision detection algorithm has "almost constant time" performance.

5.3.4 Checking a Path Segment

Collision detection algorithms determine whether a configuration lies in $\mathcal{C}_{free}$, but motion planning algorithms require that an entire path maps into $\mathcal{C}_{free}$. The interface between the planner and collision detection usually involves validation of a path segment (i.e., a path, but often a short one). This cannot be checked point-by-point because it would require an uncountably infinite number of calls to the collision detection algorithm.

Suppose that a path, $\tau : [0, 1] \rightarrow \mathcal{C}$, needs to be checked to determine whether $\tau([0, 1]) \subset \mathcal{C}_{free}$. A common approach is to sample the interval $[0, 1]$ and call the collision checker only on the samples. What resolution of sampling is required? How can one ever guarantee that the places where the path is not sampled are collision-free? There are both practical and theoretical answers to these questions. In practice, a fixed $\Delta q > 0$ is often chosen as the C-space step size. Points $t_1, t_2 \in [0, 1]$ are then chosen close enough together to ensure that $\rho(\tau(t_1), \tau(t_2)) \leq \Delta q$, in which ρ is the metric on $\mathcal{C}$. The value of Δq is often determined experimentally. If Δq is too small, then considerable time is wasted on collision checking. If Δq is too large, then there is a chance that the robot could jump through a thin obstacle.

Setting Δq empirically might not seem satisfying. Fortunately, there are sound algorithmic ways to verify that a path is collision-free. In some applications the methods are still not used because they are trickier to implement and they often yield worse performance. Therefore, both methods are presented here, and you can decide which is appropriate, depending on the context and your personal tastes.

Ensuring that $\tau([0, 1]) \subset \mathcal{C}_{free}$ requires the use of both distance information and bounds on the distance that points on $\mathcal{A}$ can travel in $\mathbb{R}$. Such bounds can be obtained by using the robot displacement metric from Example 5.6. Before expressing the general case, first we will explain the concept in terms of a rigid robot that translates and rotates in $\mathcal{W} = \mathbb{R}^2$. Let $x_t, y_t \in \mathbb{R}^2$ and $\theta \in [0, 2\pi]/\sim$. Suppose that a collision detection algorithm indicates that $\mathcal{A}(q)$ is at least d units away from collision with obstacles in $\mathcal{W}$. This information can be used to determine a region in $\mathcal{C}_{free}$ that contains q. Suppose that the next candidate configuration to be checked along τ is q'. If no point on $\mathcal{A}$ travels more than distance d when moving from q to q' along τ, then q' and all configurations between q and q' must be collision-free. This assumes that on the path from q to q', every visited configuration must lie between q_i and q'_i for the ith coordinate and any i from 1 to n.

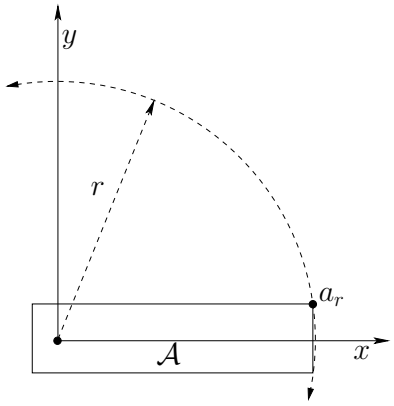


Figure 5.12: The furthest point on $\mathcal{A}$ from the origin travels the fastest when $\mathcal{A}$ is rotated. At most it can be displaced by $2\pi r$, if x_t and y_t are fixed.

If the robot can instead take any path between q and q', then no such guarantee can be made).

When $\mathcal{A}$ undergoes a translation, all points move the same distance. For rotation, however, the distance traveled depends on how far the point on $\mathcal{A}$ is from the rotation center, $(0, 0)$. Let $a_r = (x_r, y_r)$ denote the point on $\mathcal{A}$ that has the largest magnitude, $r = \sqrt{x_r^2 + y_r^2}$. Figure 5.12 shows an example. A transformed point $a \in \mathcal{A}$ may be denoted by $a(x_t, y_t, \theta)$. The following bound is obtained for any $a \in \mathcal{A}$, if the robot is rotated from orientation θ to θ':

$$\|a(x_t, y_t, \theta) - a(x_t, y_t, \theta')\| \leq \|a_r(x_t, y_t, \theta) - a_r(x_t, y_t, \theta')\| < r|\theta - \theta'|, \qquad (5.30)$$

assuming that a path in $\mathcal{C}$ is followed that interpolates between θ and θ' (using the shortest path in $\mathbb{S}^1$ between θ and θ'). Thus, if $\mathcal{A}(q)$ is at least d away from the obstacles, then the orientation may be changed without causing collision as long as $r|\theta - \theta'| < d$. Note that this is a loose upper bound because a_r travels along a circular arc and can be displaced by no more than $2\pi r$.

Similarly, x_t and y_t may individually vary up to d, yielding $|x_t - x_t'| < d$ and $|y_t - y_t'| < d$. If all three parameters vary simultaneously, then a region in $\mathcal{C}_{free}$ can be defined as

$$\{(x_t', y_t', \theta') \in \mathcal{C} \mid |x_t - x_t'| + |y_t - y_t'| + r|\theta - \theta'| < d\}. \qquad (5.31)$$

Such bounds can generally be used to set a step size, Δq, for collision checking that guarantees the intermediate points lie in $\mathcal{C}_{free}$. The particular value used may vary depending on d and the direction[8] of the path.

For the case of $SO(3)$, once again the displacement of the point on $\mathcal{A}$ that has the largest magnitude can be bounded. It is best in this case to express the bounds in terms of quaternion differences, $\|h - h'\|$. Euler angles may also be used

[8]To formally talk about directions, it would be better to define a differentiable structure on $\mathcal{C}$. This will be deferred to Section 8.3, where it seems unavoidable.

to obtain a straightforward generalization of (5.31) that has six terms, three for translation and three for rotation. For each of the three rotation parts, a point with the largest magnitude in the plane perpendicular to the rotation axis must be chosen.

If there are multiple links, it becomes much more complicated to determine the step size. Each point $a \in \mathcal{A}_i$ is transformed by some nonlinear function based on the kinematic expressions from Sections 3.3 and 3.4. Let $a : \mathcal{C} \rightarrow \mathcal{W}$ denote this transformation. In some cases, it might be possible to derive a *Lipschitz condition* of the form

$$\|a(q) - a(q')\| < c\|q - q'\|, \tag{5.32}$$

in which $c \in (0, \infty)$ is a fixed constant, a is any point on $\mathcal{A}_i$, and the expression holds for any $q, q' \in \mathcal{C}$. The goal is to make the *Lipschitz constant*, c, as small as possible; this enables larger variations in q.

A better method is to individually bound the link displacement with respect to each parameter,

$$\|a(q_1, \ldots, q_{i-1}, q_i, q_{i+1}, \ldots, q_n) - a(q_1, \ldots, q_{i-1}, q'_i, q_{i+1}, \ldots, q_n)\| < c_i|q_i - q'_i|, \tag{5.33}$$

to obtain the Lipschitz constants c_1, ..., c_n. The bound on robot displacement becomes

$$\|a(q) - a(q')\| < \sum_{i=1}^{n} c_i|q_i - q'_i|. \tag{5.34}$$

The benefit of using individual parameter bounds can be seen by considering a long chain. Consider a 50-link chain of line segments in $\mathbb{R}^2$, and each link has length 10. The C-space is $\mathbb{T}^{50}$, which can be parameterized as $[0, 2\pi]^{50}/\sim$. Suppose that the chain is in a straight-line configuration ($\theta_i = 0$ for all $1 \leq i \leq 50$), which means that the last point is at $(500, 0) \in \mathcal{W}$. Changes in θ_1, the orientation of the first link, dramatically move $\mathcal{A}_{50}$. However, changes in θ_{50} move $\mathcal{A}_{50}$ a smaller amount. Therefore, it is advantageous to pick a different Δq_i for each $1 \leq i \leq 50$. In this example, a smaller value should be used for $\Delta\theta_1$ in comparison to $\Delta\theta_{50}$.

Unfortunately, there are more complications. Suppose the 50-link chain is in a configuration that folds all of the links on top of each other ($\theta_i = \pi$ for each $1 \leq i \leq n$). In this case, $\mathcal{A}_{50}$ does not move as fast when θ_1 is perturbed, in comparison to the straight-line configuration. A larger step size for θ_1 could be used for this configuration, relative to other parts of $\mathcal{C}$. The implication is that, although Lipschitz constants can be made to hold over all of $\mathcal{C}$, it might be preferable to determine a better bound in a local region around $q \in \mathcal{C}$. A linear method could be obtained by analyzing the Jacobian of the transformations, such as (3.53) and (3.57).

Another important concern when checking a path is the order in which the samples are checked. For simplicity, suppose that Δq is constant and that the path is a constant-speed parameterization. Should the collision checker step along from 0 up to 1? Experimental evidence indicates that it is best to use a recursive binary strategy [379]. This makes no difference if the path is collision-free, but

it often saves time if the path is in collision. This is a kind of sampling problem over $[0, 1]$, which is addressed nicely by the van der Corput sequence, ν. The last column in Figure 5.2 indicates precisely where to check along the path in each step. Initially, $\tau(1)$ is checked. Following this, points from the van der Corput sequence are checked in order: $\tau(0)$, $\tau(1/2)$, $\tau(1/4)$, $\tau(3/4)$, $\tau(1/8)$, ... The process terminates if a collision is found or when the dispersion falls below Δq. If Δq is not constant, then it is possible to skip over some points of ν in regions where the allowable variation in q is larger.

5.4 Incremental Sampling and Searching

5.4.1 The General Framework

The algorithms of Sections 5.4 and 5.5 follow the *single-query model*, which means (q_I, q_G) is given only once per robot and obstacle set. This means that there are no advantages to precomputation, and the sampling-based motion planning problem can be considered as a kind of search. The *multiple-query model*, which favors precomputation, is covered in Section 5.6.

The sampling-based planning algorithms presented in the present section are strikingly similar to the family of search algorithms summarized in Section 2.2.4. The main difference lies in step 3 below, in which applying an action, u, is replaced by generating a path segment, τ_s. Another difference is that the search graph, $\mathcal{G}$, is undirected, with edges that represent paths, as opposed to a directed graph in which edges represent actions. It is possible to make these look similar by defining an action space for motion planning that consists of a collection of paths, but this is avoided here. In the case of motion planning with differential constraints, this will actually be required; see Chapter 14.

Most single-query, sampling-based planning algorithms follow this template:

1. **Initialization:** Let $\mathcal{G}(V, E)$ represent an undirected *search graph*, for which V contains at least one vertex and E contains no edges. Typically, V contains q_I, q_G, or both. In general, other points in $\mathcal{C}_{free}$ may be included.

2. **Vertex Selection Method (VSM):** Choose a vertex $q_{cur} \in V$ for expansion.

3. **Local Planning Method (LPM):** For some $q_{new} \in \mathcal{C}_{free}$ that may or may not be represented by a vertex in V, attempt to construct a path $\tau_s : [0, 1] \rightarrow \mathcal{C}_{free}$ such that $\tau(0) = q_{cur}$ and $\tau(1) = q_{new}$. Using the methods of Section 5.3.4, τ_s must be checked to ensure that it does not cause a collision. If this step fails to produce a collision-free path segment, then go to step 2.

4. **Insert an Edge in the Graph:** Insert τ_s into E, as an edge from q_{cur} to q_{new}. If q_{new} is not already in V, then it is inserted.

5. **Check for a Solution:** Determine whether $\mathcal{G}$ encodes a solution path. As in the discrete case, if there is a single search tree, then this is trivial; otherwise, it can become complicated and expensive.

6. **Return to Step 2:** Iterate unless a solution has been found or some termination condition is satisfied, in which case the algorithm reports failure.

In the present context, $\mathcal{G}$ is a topological graph, as defined in Example 4.6. Each vertex is a configuration and each edge is a path that connects two configurations. In this chapter, it will be simply referred to as a graph when there is no chance of confusion. Some authors refer to such a graph as a *roadmap*; however, we reserve the term roadmap for a graph that contains enough paths to make any motion planning query easily solvable. This case is covered in Section 5.6 and throughout Chapter 6.

A large family of sampling-based algorithms can be described by varying the implementations of steps 2 and 3. Implementations of the other steps may also vary, but this is less important and will be described where appropriate. For convenience, step 2 will be called the vertex selection method (VSM) and step 3 will be called the *local planning method* (LPM). The role of the VSM is similar to that of the priority queue, Q, in Section 2.2.1. The role of the LPM is to compute a collision-free path segment that can be added to the graph. It is called *local* because the path segment is usually simple (e.g., the shortest path) and travels a short distance. It is not *global* in the sense that the LPM does not try to solve the entire planning problem; it is expected that the LPM may often fail to construct path segments.

It will be formalized shortly, but imagine for the time being that any of the search algorithms from Section 2.2 may be applied to motion planning by approximating $\mathcal{C}$ with a high-resolution grid. The resulting problem looks like a multi-dimensional extension of Example 2.1 (the "labyrinth" walls are formed by $\mathcal{C}_{obs}$). For a high-resolution grid in a high-dimensional space, most classical discrete searching algorithms have trouble getting trapped in a local minimum. There could be an astronomical number of configurations that fall within a concavity in $\mathcal{C}_{obs}$ that must be escaped to solve the problem, as shown in Figure 5.13a. Imagine a problem in which the C-space obstacle is a giant "bowl" that can trap the configuration. This figure is drawn in two dimensions, but imagine that the $\mathcal{C}$ has many dimensions, such as six for $SE(3)$ or perhaps dozens for a linkage. If the discrete planning algorithms from Section 2.2 are applied to a high-resolution grid approximation of $\mathcal{C}$, then they will all waste their time filling up the bowl before being able to escape to q_G. The number of grid points in this bowl would typically be on the order of 100^n for an n-dimensional C-space. Therefore, sampling-based motion planning algorithms combine sampling and searching in a way that attempts to overcome this difficulty.

As in the case of discrete search algorithms, there are several classes of algorithms based on the number of search trees.

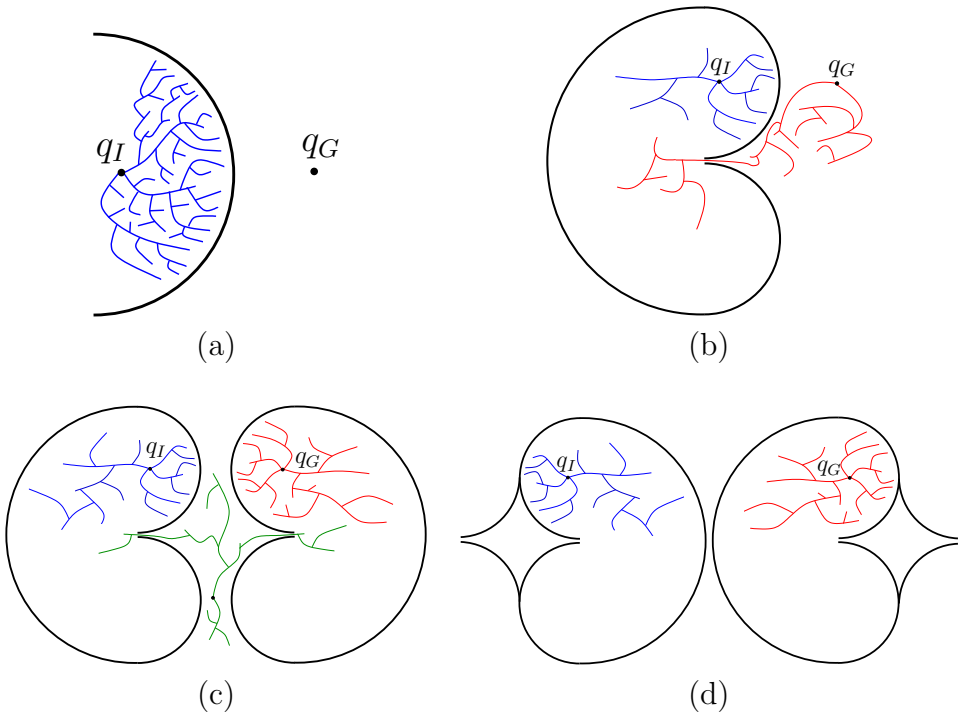


Figure 5.13: All of these depict high-dimensional obstacle regions in C-space. (a) The search must involve some sort of multi-resolution aspect, otherwise, that algorithm may explore too many points within a cavity. (b) Sometimes the problem is like a bug trap, in which case bidirectional search can help. (c) For a double bug trap, multi-directional search may be needed. (d) This example is hard to solve even for multi-directional search.

Unidirectional (single-tree) methods: In this case, the planning appears very similar to discrete forward search, which was given in Figure 2.4. The main difference between algorithms in this category is how they implement the VSM and LPM. Figure 5.13b shows a *bug trap*[9] example for which forward-search algorithms would have great trouble; however, the problem might not be difficult for backward search, if the planner incorporates some kind of greedy, best-first behavior. This example, again in high dimensions, can be considered as a kind of "bug trap." To leave the trap, a path must be found from q_I into the narrow opening. Imagine a fly buzzing around through the high-dimensional trap. The escape opening might not look too difficult in two dimensions, but if it has a small range with respect to each configuration parameter, it is nearly impossible to find the opening. The tip of the "volcano" would be astronomically small compared to the rest of the bug trap. Examples such as this provide some motivation for bidirectional

[9]This principle is actually used in real life to trap flying bugs. This analogy was suggested by James O'Brien in a discussion with James Kuffner.

algorithms. It might be easier for a search tree that starts in q_G to arrive in the bug trap.

Bidirectional (two-tree) methods: Since it is not known whether q_I or q_G might lie in a bug trap (or another challenging region), a bidirectional approach is often preferable. This follows from an intuition that two propagating wavefronts, one centered on q_I and the other on q_G, will meet after covering less area in comparison to a single wavefront centered at q_I that must arrive at q_G. A bidirectional search is achieved by defining the VSM to alternate between trees when selecting vertices. The LPM sometimes generates paths that explore new parts of $\mathcal{C}_{free}$, and at other times it tries to generate a path that connects the two trees.

Multi-directional (more than two trees) methods: If the problem is so bad that a double bug trap exists, as shown in Figure 5.13c, then it might make sense to grow trees from other places in the hopes that there are better chances to enter the traps in the other direction. This complicates the problem of connecting trees, however. Which pairs of trees should be selected in each iteration for possible connection? How often should the same pair be selected? Which vertex pairs should be selected? Many heuristic parameters may arise in practice to answer these questions.

Of course, one can play the devil's advocate and construct the example in Figure 5.13d, for which virtually all sampling-based planning algorithms are doomed. Even harder versions can be made in which a sequence of several narrow corridors must be located and traversed. We must accept the fact that some problems are hopeless to solve using sampling-based planning methods, unless there is some problem-specific structure that can be additionally exploited.

5.4.2 Adapting Discrete Search Algorithms

One of the most convenient and straightforward ways to make sampling-based planning algorithms is to define a grid over $\mathcal{C}$ and conduct a discrete search using the algorithms of Section 2.2. The resulting planning problem actually looks very similar to Example 2.1. Each edge now corresponds to a path in $\mathcal{C}_{free}$. Some edges may not exist because of collisions, but this will have to be revealed incrementally during the search because an explicit representation of $\mathcal{C}_{obs}$ is too expensive to construct (recall Section 4.3).

Assume that an n-dimensional C-space is represented as a unit cube, $\mathcal{C} = [0,1]^n/\sim$, in which $\sim$ indicates that identifications of the sides of the cube are made to reflect the C-space topology. Representing $\mathcal{C}$ as a unit cube usually requires a reparameterization. For example, an angle $\theta \in [0, 2\pi)$ would be replaced with $\theta/2\pi$ to make the range lie within $[0,1]$. If quaternions are used for $SO(3)$, then the upper half of $\mathbb{S}^3$ must be deformed into $[0,1]^3/\sim$.

Discretization Assume that $\mathcal{C}$ is *discretized* by using the *resolutions* k_1, k_2,..., and k_n, in which each k_i is a positive integer. This allows the resolution to be different for each C-space coordinate. Either a standard grid or a Sukharev grid can be used. Let

$$\Delta q_i = [0 \;\; \cdots \;\; 0 \;\; 1/k_i \;\; 0 \;\; \cdots \;\; 0], \tag{5.35}$$

in which the first $i-1$ components and the last $n-i$ components are 0. A *grid point* is a configuration $q \in \mathcal{C}$ that can be expressed in the form[10]

$$\sum_{i=1}^{n} j_i \Delta q_i, \tag{5.36}$$

in which each $j_i \in \{0, 1, \ldots, k_i\}$. The integers j_1, ..., j_n can be imagined as array indices for the grid. Let the term *boundary grid point* refer to a grid point for which $j_i = 0$ or $j_i = k_i$ for some i. Due to identifications, boundary grid points might have more than one representation using (5.36).

Neighborhoods For each grid point q we need to define the set of nearby grid points for which an edge may be constructed. Special care must be given to defining the neighborhood of a boundary grid point to ensure that identifications and the C-space boundary (if it exists) are respected. If q is not a boundary grid point, then the *1-neighborhood* is defined as

$$N_1(q) = \{q + \Delta q_1, \ldots, q + \Delta q_n, q - \Delta q_1, \ldots, q - \Delta q_n\}. \tag{5.37}$$

For an n-dimensional C-space there at most $2n$ 1-neighbors. In two dimensions, this yields at most four 1-neighbors, which can be thought of as "up," "down," "left," and "right." There are *at most* four because some directions may be blocked by the obstacle region.

A *2-neighborhood* is defined as

$$N_2(q) = \{q \pm \Delta q_i \pm \Delta q_j \;\; | \;\; 1 \leq i, j \leq n, \; i \neq j\} \cup N_1(q). \tag{5.38}$$

Similarly, a *k-neighborhood* can be defined for any positive integer $k \leq n$. For an n-neighborhood, there are at most $3^n - 1$ neighbors; there may be fewer due to boundaries or collisions. The definitions can be easily extended to handle the boundary points.

Obtaining a discrete planning problem Once the grid and neighborhoods have been defined, a discrete planning problem is obtained. Figure 5.14 depicts the process for a problem in which there are nine Sukharev grid points in $[0, 1]^2$. Using 1-neighborhoods, the potential edges in the search graph, $\mathcal{G}(V, E)$, appear in Figure 5.14a. Note that $\mathcal{G}$ is a topological graph, as defined in Example 4.6, because each vertex is a configuration and each edge is a path. If q_I and q_G do not

[10] Alternatively, the general lattice definition in (5.21) could be used.

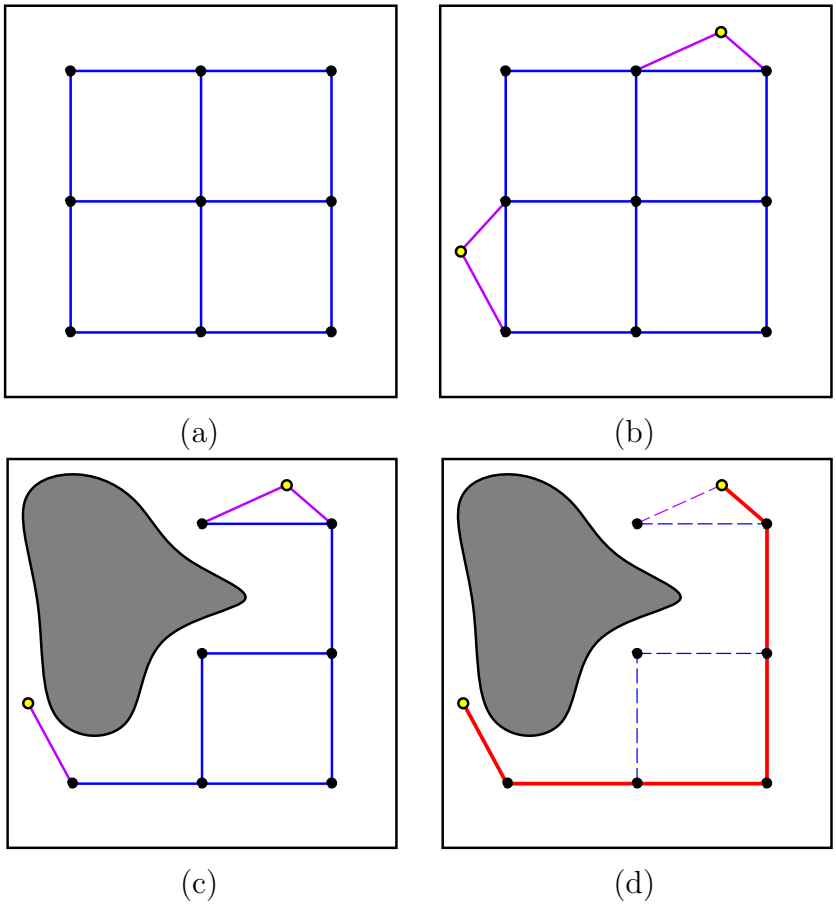


Figure 5.14: A topological graph can be constructed during the search and can successfully solve a motion planning problem using very few samples.

coincide with grid points, they need to be connected to some nearby grid points, as shown in Figure 5.14b. What grid points should q_I and q_G be connected to? As a general rule, if k-neighbors are used, then one should try connecting q_I and q_G to any grid points that are at least as close as the furthest k-neighbor from a typical grid point.

Usually, all of the vertices and edges shown in Figure 5.14b do not appear in $\mathcal{G}$ because some intersect with $\mathcal{C}_{obs}$. Figure 5.14c shows a more typical situation, in which some of the potential vertices and edges are removed because of collisions. This representation could be computed in advance by checking all potential vertices and edges for collision. This would lead to a roadmap, which is suited for multiple queries and is covered in Section 5.6. In this section, it is assumed that $\mathcal{G}$ is revealed "on the fly" during the search. This is the same situation that occurs for the discrete planning methods from Section 2.2. In the current setting, the potential edges of $\mathcal{G}$ are validated during the search. The candidate edges to evaluate are given by the definition of the k-neighborhoods. During the search,

any edge or vertex that has been checked for collision explicitly appears in a data structure so that it does not need to be checked again. At the end of the search, a path is found, as depicted in Figure 5.14d.

Grid resolution issues The method explained so far will nicely find the solution to many problems when provided with the correct resolution. If the number of points per axis is too high, then the search may be too slow. This motivates selecting fewer points per axis, but then solutions might be missed. This trade-off is fundamental to sampling-based motion planning. In a more general setting, if other forms of sampling and neighborhoods are used, then enough samples have to be generated to yield a sufficiently small dispersion.

There are two general ways to avoid having to select this resolution (or more generally, dispersion):

1. Iteratively refine the resolution until a solution is found. In this case, sampling and searching become interleaved. One important variable is how frequently to alternate between the two processes. This will be presented shortly.

2. An alternative is to abandon the adaptation of discrete search algorithms and develop algorithms directly for the continuous problem. This forms the basis of the methods in Sections 5.4.3, 5.4.4, and 5.5.

The most straightforward approach is to iteratively improve the grid resolution. Suppose that initially a standard grid with 2^n points total and 2 points per axis is searched using one of the discrete search algorithms, such as best-first or A^*. If the search fails, what should be done? One possibility is to double the resolution, which yields a grid with 4^n points. Many of the edges can be reused from the first grid; however, the savings diminish rapidly in higher dimensions. Once the resolution is doubled, the search can be applied again. If it fails again, then the resolution can be doubled again to yield 8^n points. In general, there would be a full grid for 2^{ni} points, for each i. The problem is that if n is large, then the rate of growth is too large. For example, if $n = 10$, then there would initially be 1024 points; however, when this fails, the search is not performed again until there are over one million points! If this also fails, then it might take a very long time to reach the next level of resolution, which has 2^{30} points.

A method similar to iterative deepening from Section 2.2.2 would be preferable. Simply discard the efforts of the previous resolution and make grids that have i^n points per axis for each iteration i. This yields grids of sizes 2^n, 3^n, 4^n, and so on, which is much better. The amount of effort involved in searching a larger grid is insignificant compared to the time wasted on lower resolution grids. Therefore, it seems harmless to discard previous work.

A better solution is not to require that a complete grid exists before it can be searched. For example, the resolution can be increased for one axis at a time before attempting to search again. Even better yet may be to tightly interleave

searching and sampling. For example, imagine that the samples appear as an infinite, dense sequence α. The graph can be searched after every 100 points are added, assuming that neighborhoods can be defined or constructed even though the grid is only partially completed. If the search is performed too frequently, then searching would dominate the running time. An easy way make this efficient is to apply the *union-find algorithm* [243, 823] to iteratively keep track of connected components in $\mathcal{G}$ instead of performing explicit searching. If q_I and q_G become part of the same connected component, then a solution path has been found. Every time a new point in the sequence α is added, the "search" is performed in nearly[11] constant time by the union-find algorithm. This is the tightest interleaving of the sampling and searching, and results in a nice sampling-based algorithm that requires no resolution parameter. It is perhaps best to select a sequence α that contains some lattice structure to facilitate the determination of neighborhoods in each iteration.

What if we simply declare the resolution to be outrageously high at the outset? Imagine there are 100^n points in the grid. This places all of the burden on the search algorithm. If the search algorithm itself is good at avoiding local minima and has built-in multi-resolution qualities, then it may perform well without the iterative refinement of the sampling. The method of Section 5.4.3 is based on this idea by performing best-first search on a high-resolution grid, combined with random walks to avoid local minima. The algorithms of Section 5.5 go one step further and search in a multi-resolution way without requiring resolutions and neighborhoods to be explicitly determined. This can be considered as the limiting case as the number of points per axis approaches infinity.

Although this section has focused on grids, it is also possible to use other forms of sampling from Section 5.2. This requires defining the neighborhoods in a suitable way that generalizes the k-neighborhoods of this section. In every case, an infinite, dense sample sequence must be defined to obtain resolution completeness by reducing the dispersion to zero in the limit. Methods for obtaining neighborhoods for irregular sample sets have been developed in the context of sampling-based roadmaps; see Section 5.6. The notion of improving resolution becomes generalized to adding samples that improve dispersion (or even discrepancy).

5.4.3 Randomized Potential Fields

Adapting the discrete algorithms from Section 2.2 works well if the problem can be solved with a small number of points. The number of points per axis must be small or the dimension must be low, to ensure that the number of points, k^n, for k points per axis and n dimensions is small enough so that every vertex in g can be reached in a reasonable amount of time. If, for example, the problem requires 50 points per axis and the dimension is 10, then it is impossible to search all of

[11] It is not constant because the running time is proportional to the inverse Ackerman function, which grows very, very slowly. For all practical purposes, the algorithm operates in constant time. See Section 6.5.2.

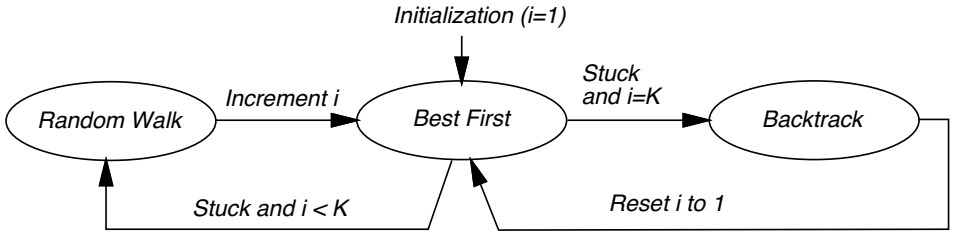


Figure 5.15: The randomized potential field method can be modeled as a three-state machine.

the 50^{10} samples. Planners that exploit best-first heuristics might find the answer without searching most of them; however, for a simple problem such as that shown in Figure 5.13a, the planner will take too long exploring the vertices in the bowl.[12]

The *randomized potential field* [70, 72, 588] approach uses random walks to attempt to escape local minima when best-first search becomes stuck. It was one of the first sampling-based planners that developed specialized techniques beyond classical discrete search, in an attempt to better solve challenging motion planning problems. In many cases, remarkable results were obtained. In its time, the approach was able to solve problems up to 31 degrees of freedom, which was well beyond what had been previously possible. The main drawback, however, was that the method involved many heuristic parameters that had to be adjusted for each problem. This frustration eventually led to the development of better approaches, which are covered in Sections 5.4.4, 5.5, and 5.6. Nevertheless, it is worthwhile to study the clever heuristics involved in this earlier method because they illustrate many interesting issues, and the method was very influential in the development of other sampling-based planning algorithms.[13]

The most complicated part of the algorithm is the definition of a *potential function*, which can be considered as a pseudometric that tries to estimate the distance from any configuration to the goal. In most formulations, there is an *attractive* term, which is a metric on $\mathcal{C}$ that yields the distance to the goal, and a *repulsive* term, which penalizes configurations that come too close to obstacles. The construction of potential functions involves many heuristics and is covered in great detail in [588]. One of the most effective methods involves constructing cost-to-go functions over $\mathcal{W}$ and lifting them to $\mathcal{C}$ [71]. In this section, it will be sufficient to assume that some potential function, $g(q)$, is defined, which is the same notation (and notion) as a cost-to-go function in Section 2.2.2. In this case, however, there is no requirement that $g(q)$ is optimal or even an underestimate of the true cost to go.

When the search becomes stuck and a random walk is needed, it is executed for some number of iterations. Using the discretization procedures of Section 5.4.2, a

[12] Of course, that problem does not appear to need so many points per axis; fewer may be used instead, if the algorithm can adapt the sampling resolution or dispersion.

[13] The exciting results obtained by the method even helped inspire me many years ago to work on motion planning.

high-resolution grid (e.g., 50 points per axis) is initially defined. In each iteration, the current configuration is modified as follows. Each coordinate, q_i, is increased or decreased by Δq_i (the grid step size) based on the outcome of a fair coin toss. Topological identifications must be respected, of course. After each iteration, the new configuration is checked for collision, or whether it exceeds the boundary of $\mathcal{C}$ (if it has a boundary). If so, then it is discarded, and another attempt is made from the previous configuration. The failures can repeat indefinitely until a new configuration in $\mathcal{C}_{free}$ is obtained.

The resulting planner can be described in terms of a three-state machine, which is shown in Figure 5.15. Each state is called a *mode* to avoid confusion with earlier state-space concepts. The VSM and LPM are defined in terms of the mode. Initially, the planner is in the BEST FIRST mode and uses q_I to start a gradient descent. While in the BEST FIRST mode, the VSM selects the newest vertex, $v \in V$. In the first iteration, this is q_I. The LPM creates a new vertex, v_n, in a neighborhood of v, in a direction that minimizes g. The direction sampling may be performed using randomly selected or deterministic samples. Using random samples, the sphere sampling method from Section 5.2.2 can be applied. After some number of tries (another parameter), if the LPM is unsuccessful at reducing g, then the mode is changed to RANDOM WALK because the best-first search is stuck in a local minimum of g.

In the RANDOM WALK mode, a random walk is executed from the newest vertex. The random walk terminates if either g is lowered or a specified limit of iterations is reached. The limit is actually sampled from a predetermined random variable (which contains parameters that also must be selected). When the RANDOM WALK mode terminates, the mode is changed back to BEST FIRST. A counter is incremented to keep track of the number of times that the random walk was attempted. A parameter K determines the maximum number of attempted random walks (a reasonable value is $K = 20$ [71]). If BEST FIRST fails after K random walks have been attempted, then the BACKTRACK mode is entered. The BACKTRACK mode selects a vertex at random from among the vertices in V that were obtained during a random walk. Following this, the counter is reset, and the mode is changed back to BEST FIRST.

Due to the random walks, the resulting paths are often too complicated to be useful in applications. Fortunately, it is straightforward to transform a computed path into a simpler one that is still collision-free. A common approach is to iteratively pick pairs of points at random along the domain of the path and attempt to replace the path segment with a straight-line path (in general, the shortest path in $\mathcal{C}$). For example, suppose $t_1, t_2 \in [0,1]$ are chosen at random, and $\tau : [0,1] \rightarrow \mathcal{C}_{free}$ is the computed solution path. This path is transformed into a new path,

$$\tau'(t) = \begin{cases} \tau(t) & \text{if } 0 \leq t \leq t_1 \\ a\tau(t_1) + (1-a)\tau(t_2) & \text{if } t_1 \leq t \leq t_2 \\ \tau(t) & \text{if } t_2 \leq t \leq 1, \end{cases} \tag{5.39}$$

in which $a \in [0,1]$ represents the fraction of the way from t_1 to t_2. Explicitly,

$a = (t_2 - t)/(t_2 - t_1)$. The new path must be checked for collision. If it passes, then it replaces the old path; otherwise, it is discarded and a new pair t_1, t_2, is chosen.

The randomized potential field approach can escape high-dimensional local minima, which allow interesting solutions to be found for many challenging high-dimensional problems. Unfortunately, the heavy amount of parameter tuning caused most people to abandon the method in recent times, in favor of newer methods.

5.4.4 Other Methods

Several influential sampling-based methods are given here. Each of them appears to offer advantages over the randomized potential field method. All of them use randomization, which was perhaps inspired by the potential field method.

Ariadne's Clew algorithm This approach grows a search tree that is biased to explore as much new territory as possible in each iteration [688, 687]. There are two modes, SEARCH and EXPLORE, which alternate over successive iterations. In the EXPLORE mode, the VSM selects a vertex, v_e, at random, and the LPM finds a new configuration that can be easily connected to v_e and is as far as possible from the other vertices in $\mathcal{G}$. A global optimization function that aggregates the distances to other vertices is optimized using a genetic algorithm. In the SEARCH mode, an attempt is made to extend the vertex added in the EXPLORE mode to the goal configuration. The key idea from this approach, which influenced both the next approach and the methods in Section 5.5, is that some of the time must be spent exploring the space, as opposed to focusing on finding the solution. The greedy behavior of the randomized potential field led to some efficiency but was also its downfall for some problems because it was all based on escaping local minima with respect to the goal instead of investing time on global exploration. One disadvantage of Ariadne's Clew algorithm is that it is very difficult to solve the optimization problem for placing a new vertex in the EXPLORE mode. Genetic algorithms were used in [687], which are generally avoided for motion planning because of the required problem-specific parameter tuning.

Expansive-space planner This method [467, 844] generates samples in a way that attempts to explore new parts of the space. In this sense, it is similar to the explore mode of the Ariadne's Clew algorithm. Furthermore, the planner is made more efficient by borrowing the bidirectional search idea from discrete search algorithms, as covered in Section 2.2.3. The VSM selects a vertex, v_e, from $\mathcal{G}$ with a probability that is inversely proportional to the number of other vertices of $\mathcal{G}$ that lie within a predetermined neighborhood of v_e. Thus, "isolated" vertices are more likely to be chosen. The LPM generates a new vertex v_n at random within a predetermined neighborhood of v_e. It will decide to insert v_n into $\mathcal{G}$ with a probability that is inversely proportional to the number of other vertices

of $\mathcal{G}$ that lie within a predetermined neighborhood of v_n. For a fixed number of iterations, the VSM repeatedly chooses the same vertex, until moving on to another vertex. The resulting planner is able to solve many interesting problems by using a surprisingly simple criterion for the placement of points. The main drawbacks are that the planner requires substantial parameter tuning, which is problem-specific (or at least specific to a similar family of problems), and the performance tends to degrade if the query requires systematically searching a long labyrinth. Choosing the radius of the predetermined neighborhoods essentially amounts to determining the appropriate resolution.

Random-walk planner A surprisingly simple and efficient algorithm can be made entirely from random walks [179]. To avoid parameter tuning, the algorithm adjusts its distribution of directions and magnitude in each iteration, based on the success of the past k iterations (perhaps k is the only parameter). In each iteration, the VSM just selects the vertex that was most recently added to $\mathcal{G}$. The LPM generates a direction and magnitude by generating samples from a multivariate Gaussian distribution whose covariance parameters are adaptively tuned. The main drawback of the method is similar to that of the previous method. Both have difficulty traveling through long, winding corridors. It is possible to combine adaptive random walks with other search algorithms, such as the potential field planner [178].

5.5 Rapidly Exploring Dense Trees

This section introduces an incremental sampling and searching approach that yields good performance in practice without any parameter tuning.[14] The idea is to incrementally construct a search tree that gradually improves the resolution but does not need to explicitly set any resolution parameters. In the limit, the tree densely covers the space. Thus, it has properties similar to space filling curves [842], but instead of one long path, there are shorter paths that are organized into a tree. A dense sequence of samples is used as a guide in the incremental construction of the tree. If this sequence is random, the resulting tree is called a *rapidly exploring random tree (RRT)*. In general, this family of trees, whether the sequence is random or deterministic, will be referred to as *rapidly exploring dense trees (RDTs)* to indicate that a dense covering of the space is obtained. This method was originally developed for motion planning under differential constraints [608, 611]; that case is covered in Section 14.4.3.

[14] The original RRT [598] was introduced with a step size parameter, but this is eliminated in the current presentation. For implementation purposes, one might still want to revert to this older way of formulating the algorithm because the implementation is a little easier. This will be discussed shortly.

```
SIMPLE_RDT(q_0)
1  G.init(q_0);
2  for i = 1 to k do
3      G.add_vertex(α(i));
4      q_n ← NEAREST(S(G), α(i));
5      G.add_edge(q_n, α(i));
```

Figure 5.16: The basic algorithm for constructing RDTs (which includes RRTs as a special case) when there are no obstacles. It requires the availability of a dense sequence, α, and iteratively connects from $\alpha(i)$ to the nearest point among all those reached by $\mathcal{G}$.

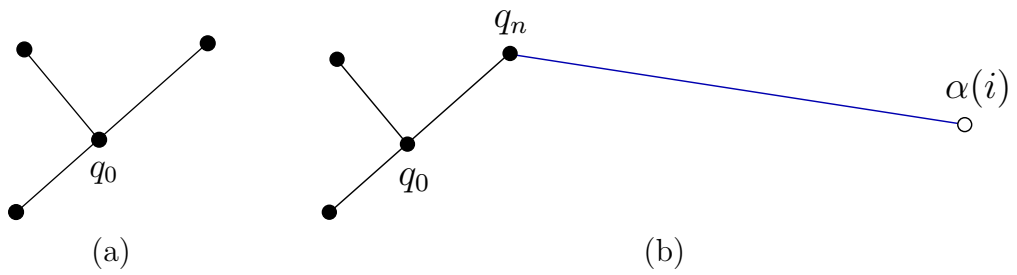


Figure 5.17: (a) Suppose inductively that this tree has been constructed so far using the algorithm in Figure 5.16. (b) A new edge is added that connects from the sample $\alpha(i)$ to the nearest point in S, which is the vertex q_n.

5.5.1 The Exploration Algorithm

Before explaining how to use these trees to solve a planning query, imagine that the goal is to get as close as possible to every configuration, starting from an initial configuration. The method works for any dense sequence. Once again, let α denote an infinite, dense sequence of samples in $\mathcal{C}$. The ith sample is denoted by $\alpha(i)$. This may possibly include a uniform, random sequence, which is only dense with probability one. Random sequences that induce a nonuniform bias are also acceptable, as long as they are dense with probability one.

An RDT is a topological graph, $\mathcal{G}(V, E)$. Let $S \subset \mathcal{C}_{free}$ indicate the set of all points reached by $\mathcal{G}$. Since each $e \in E$ is a path, this can be expressed as the *swath*, S, of the graph, which is defined as

$$S = \bigcup_{e \in E} e([0,1]). \tag{5.40}$$

In (5.40), $e([0,1]) \subseteq \mathcal{C}_{free}$ is the image of the path e.

The exploration algorithm is first explained in Figure 5.16 without any obstacles or boundary obstructions. It is assumed that $\mathcal{C}$ is a metric space. Initially, a vertex is made at q_0. For k iterations, a tree is iteratively grown by connecting

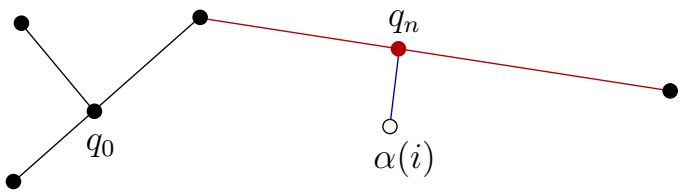
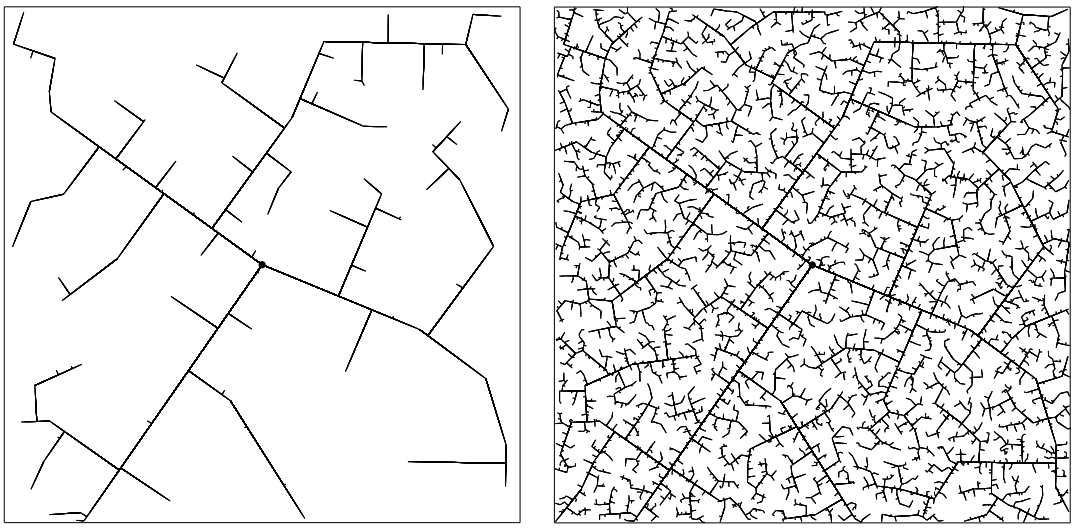


Figure 5.18: If the nearest point in S lies in an edge, then the edge is split into two, and a new vertex is inserted into $\mathcal{G}$.



45 iterations 2345 iterations

Figure 5.19: In the early iterations, the RRT quickly reaches the unexplored parts. However, the RRT is dense in the limit (with probability one), which means that it gets arbitrarily close to any point in the space.

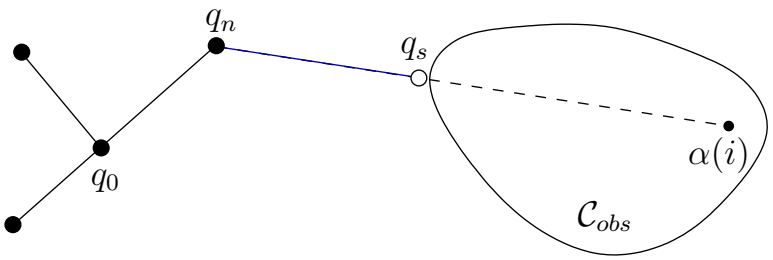


Figure 5.20: If there is an obstacle, the edge travels up to the obstacle boundary, as far as allowed by the collision detection algorithm.

$\alpha(i)$ to its nearest point in the swath, S. The connection is usually made along the shortest possible path. In every iteration, $\alpha(i)$ becomes a vertex. Therefore, the resulting tree is dense. Figures 5.17–5.18 illustrate an iteration graphically. Suppose the tree has three edges and four vertices, as shown in Figure 5.17a. If the nearest point, $q_n \in S$, to $\alpha(i)$ is a vertex, as shown in Figure 5.17b, then an edge is made from q_n to $\alpha(i)$. However, if the nearest point lies in the interior of an edge, as shown in Figure 5.18, then the existing edge is split so that q_n appears as a new vertex, and an edge is made from q_n to $\alpha(i)$. The edge splitting, if required, is assumed to be handled in line 4 by the method that adds edges. Note that the total number of edges may increase by one or two in each iteration.

The method as described here does not fit precisely under the general framework from Section 5.4.1; however, with the modifications suggested in Section 5.5.2, it can be adapted to fit. In the RDT formulation, the NEAREST function serves the purpose of the VSM, but in the RDT, a point may be selected from anywhere in the swath of the graph. The VSM can be generalized to a *swath-point selection method*, SSM. This generalization will be used in Section 14.3.4. The LPM tries to connect $\alpha(i)$ to q_n along the shortest path possible in $\mathcal{C}$.

Figure 5.19 shows an execution of the algorithm in Figure 5.16 for the case in which $\mathcal{C} = [0,1]^2$ and $q_0 = (1/2, 1/2)$. It exhibits a kind of fractal behavior.[15] Several main branches are first constructed as it rapidly reaches the far corners of the space. Gradually, more and more area is filled in by smaller branches. From the pictures, it is clear that in the limit, the tree densely fills the space. Thus, it can be seen that the tree gradually improves the resolution (or dispersion) as the iterations continue. This behavior turns out to be ideal for sampling-based motion planning.

Recall that in sampling-based motion planning, the obstacle region $\mathcal{C}_{obs}$ is not explicitly represented. Therefore, it must be taken into account in the construction of the tree. Figure 5.20 indicates how to modify the algorithm in Figure 5.16 so that collision checking is taken into account. The modified algorithm appears in Figure 5.21. The procedure STOPPING-CONFIGURATION yields the nearest configuration possible to the boundary of $\mathcal{C}_{free}$, along the direction toward $\alpha(i)$. The nearest

[15] If α is uniform, random, then a *stochastic fractal* [586] is obtained. Deterministic fractals can be constructed using sequences that have appropriate symmetries.

```
RDT(q_0)
1  G.init(q_0);
2  for i = 1 to k do
3      q_n ← NEAREST(S, α(i));
4      q_s ← STOPPING-CONFIGURATION(q_n, α(i));
5      if q_s ≠ q_n then
6          G.add_vertex(q_s);
7          G.add_edge(q_n, q_s);
```

Figure 5.21: The RDT with obstacles.

point $q_n \in S$ is defined to be same (obstacles are ignored); however, the new edge might not reach to $\alpha(i)$. In this case, an edge is made from q_n to q_s, the last point possible before hitting the obstacle. How close can the edge come to the obstacle boundary? This depends on the method used to check for collision, as explained in Section 5.3.4. It is sometimes possible that q_n is already as close as possible to the boundary of $\mathcal{C}_{free}$ in the direction of $\alpha(i)$. In this case, no new edge or vertex is added that for that iteration.

5.5.2 Efficiently Finding Nearest Points

There are several interesting alternatives for implementing the NEAREST function in line 3 of the algorithm in Figure 5.16. There are generally two families of methods: *exact* or *approximate*. First consider the exact case.

Exact solutions Suppose that all edges in $\mathcal{G}$ are line segments in $\mathbb{R}^m$ for some dimension $m \geq n$. An edge that is generated early in the construction process will be split many times in later iterations. For the purposes of finding the nearest point in S, however, it is best to handle this as a single segment. For example, see the three large branches that extend from the root in Figure 5.19. As the number of points increases, the benefit of agglomerating the segments increases. Let each of these agglomerated segments be referred to as a *supersegment*. To implement NEAREST, a primitive is needed that computes the distance between a point and a line segment. This can be performed in constant time with simple vector calculus. Using this primitive, NEAREST is implemented by iterating over all supersegments and taking the point with minimum distance among all of them. It may be possible to improve performance by building hierarchical data structures that can eliminate large sets of supersegments, but this remains to be seen experimentally.

In some cases, the edges of $\mathcal{G}$ may not be line segments. For example, the shortest paths between two points in $SO(3)$ are actually circular arcs along $\mathbb{S}^3$. One possible solution is to maintain a separate parameterization of $\mathcal{C}$ for the purposes of computing the NEAREST function. For example, $SO(3)$ can be represented as $[0,1]^3/\sim$, by making the appropriate identifications to obtain $\mathbb{RP}^3$. Straight-

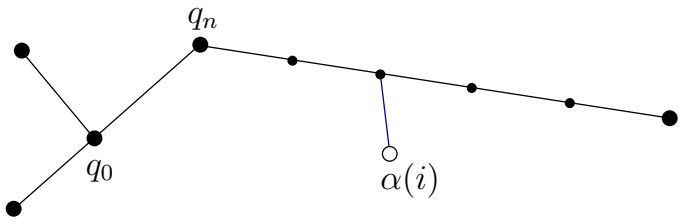


Figure 5.22: For implementation ease, intermediate vertices can be inserted to avoid checking for closest points along line segments. The trade-off is that the number of vertices is increased dramatically.

line segments can then be used. The problem is that the resulting metric is not consistent with the Haar measure, which means that an accidental bias would result. Another option is to tightly enclose $\mathbb{S}^3$ in a 4D cube. Every point on $\mathbb{S}^3$ can be mapped outward onto a cube face. Due to antipodal identification, only four of the eight cube faces need to be used to obtain a bijection between the set of all rotation and the cube surface. Linear interpolation can be used along the cube faces, as long as both points remain on the same face. If the points are on different faces, then two line segments can be used by bending the shortest path around the corner between the two faces. This scheme will result in less distortion than mapping $SO(3)$ to $[0,1]^3/\sim$; however, some distortion will still exist.

Another approach is to avoid distortion altogether and implement primitives that can compute the distance between a point and a curve. In the case of $SO(3)$, a primitive is needed that can find the distance between a circular arc in $\mathbb{R}^m$ and a point in $\mathbb{R}^m$. This might not be too difficult, but if the curves are more complicated, then an exact implementation of the NEAREST function may be too expensive computationally.

Approximate solutions Approximate solutions are much easier to construct, however, a resolution parameter is introduced. Each path segment can be approximated by inserting intermediate vertices along long segments, as shown in Figure 5.22. The intermediate vertices should be added each time a new sample, $\alpha(i)$, is inserted into $\mathcal{G}$. A parameter Δq can be defined, and intermediate samples are inserted to ensure that no two consecutive vertices in $\mathcal{G}$ are ever further than Δq from each other. Using intermediate vertices, the interiors of the edges in $\mathcal{G}$ are ignored when finding the nearest point in S. The approximate computation of NEAREST is performed by finding the closest vertex to $\alpha(i)$ in $\mathcal{G}$. This approach is by far the simplest to implement. It also fits precisely under the incremental sampling and searching framework from Section 5.4.1.

When using intermediate vertices, the trade-offs are clear. The computation time for each evaluation of NEAREST is linear in the number of vertices. Increasing the number of vertices improves the quality of the approximation, but it also dramatically increases the running time. One way to recover some of this cost is to insert the vertices into an efficient data structure for nearest-neighbor searching.

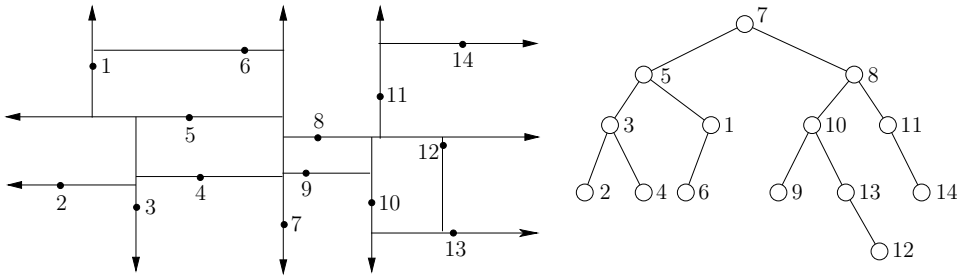


Figure 5.23: A Kd-tree can be used for efficient nearest-neighbor computations.

One of the most practical and widely used data structures is the *Kd-tree* [264, 365, 758]. A depiction is shown in Figure 5.23 for 14 points in $\mathbb{R}^2$. The Kd-tree can be considered as a multi-dimensional generalization of a binary search tree. The Kd-tree is constructed for points, P, in $\mathbb{R}^2$ as follows. Initially, sort the points with respect to the x coordinate. Take the median point, $p \in P$, and divide P into two sets, depending on which side of a vertical line through p the other points fall. For each of the two sides, sort the points by the y coordinate and find their medians. Points are divided at this level based on whether they are above or below horizontal lines. At the next level of recursion, vertical lines are used again, followed by horizontal again, and so on. The same idea can be applied in $\mathbb{R}^n$ by cycling through the n coordinates, instead of alternating between x and y, to form the divisions. In [52], the Kd-tree is extended to topological spaces that arise in motion planning and is shown to yield good performance for RRTs and sampling-based roadmaps. A Kd-tree of k points can be constructed in $O(nk \lg k)$ time. Topological identifications must be carefully considered when traversing the tree. To find the nearest point in the tree to some given point, the query algorithm descends to a leaf vertex whose associated region contains the query point, finds all distances from the data points in this leaf to the query point, and picks the closest one. Next, it recursively visits those surrounding leaf vertices that are further from the query point than the closest point found so far [47, 52]. The nearest point can be found in time logarithmic in k.

Unfortunately, these bounds hide a constant that increases exponentially with the dimension, n. In practice, the Kd-tree is useful in motion planning for problems of up to about 20 dimensions. After this, the performance usually degrades too much. As an empirical rule, if there are more than 2^n points, then the Kd-tree should be more efficient than naive nearest neighbors. In general, the trade-offs must be carefully considered in a particular application to determine whether exact solutions, approximate solutions with naive nearest-neighbor computations, or approximate solutions with Kd-trees will be more efficient. There is also the issue of implementation complexity, which probably has caused most people to prefer the approximate solution with naive nearest-neighbor computations.

5.5.3 Using the Trees for Planning

So far, the discussion has focused on exploring $\mathcal{C}_{free}$, but this does not solve a planning query by itself. RRTs and RDTs can be used in many ways in planning algorithms. For example, they could be used to escape local minima in the randomized potential field planner of Section 5.4.3.

Single-tree search A reasonably efficient planner can be made by directly using the algorithm in Figure 5.21 to grow a tree from q_I and periodically check whether it is possible to connect the RDT to q_G. An easy way to achieve this is to start with a dense sequence α and periodically insert q_G at regularly spaced intervals. For example, every 100th sample could be q_G. Each time this sample is reached, an attempt is made to reach q_G from the closest vertex in the RDT. If the sample sequence is random, which generates an RRT, then the following modification works well. In each iteration, toss a biased coin that has probability 99/100 of being HEADS and 1/100 of being TAILS. If the result is HEADS, then set $\alpha(i)$, to be the next element of the pseudorandom sequence; otherwise, set $\alpha(i) = q_G$. This forces the RRT to occasionally attempt to make a connection to the goal, q_G. Of course, 1/100 is arbitrary, but it is in a range that works well experimentally. If the bias is too strong, then the RRT becomes too greedy like the randomized potential field. If the bias is not strong enough, then there is no incentive to connect the tree to q_G. An alternative is to consider other dense, but not necessarily nonuniform sequences in $\mathcal{C}$. For example, in the case of random sampling, the probability density function could contain a gentle bias towards the goal. Choosing such a bias is a difficult heuristic problem; therefore, such a technique should be used with caution (or avoided altogether).

Balanced, bidirectional search Much better performance can usually be obtained by growing two RDTs, one from q_I and the other from q_G. This is particularly valuable for escaping one of the bug traps, as mentioned in Section 5.4.1. For a grid search, it is straightforward to implement a bidirectional search that ensures that the two trees meet. For the RDT, special considerations must be made to ensure that the two trees will connect while retaining their "rapidly exploring" property. One additional idea is to make sure that the bidirectional search is balanced [560], which ensures that both trees are the same size.

Figure 5.24 gives an outline of the algorithm. The graph $\mathcal{G}$ is decomposed into two trees, denoted by T_a and T_b. Initially, these trees start from q_I and q_G, respectively. After some iterations, T_a and T_b are swapped; therefore, keep in mind that T_a is not always the tree that contains q_I. In each iteration, T_a is grown exactly the same way as in one iteration of the algorithm in Figure 5.16. If a new vertex, q_s, is added to T_a, then an attempt is made in lines 10–12 to extend T_b. Rather than using $\alpha(i)$ to extend T_b, the new vertex q_s of T_a is used. This causes T_b to try to grow toward T_a. If the two connect, which is tested in line 13, then a solution has been found.

```
RDT_BALANCED_BIDIRECTIONAL(q_I, q_G)
 1   T_a.init(q_I); T_b.init(q_G);
 2   for i = 1 to K do
 3       q_n ← NEAREST(S_a, α(i));
 4       q_s ← STOPPING-CONFIGURATION(q_n, α(i));
 5       if q_s ≠ q_n then
 6           T_a.add_vertex(q_s);
 7           T_a.add_edge(q_n, q_s);
 8           q'_n ← NEAREST(S_b, q_s);
 9           q'_s ← STOPPING-CONFIGURATION(q'_n, q_s);
10           if q'_s ≠ q'_n then
11               T_b.add_vertex(q'_s);
12               T_b.add_edge(q'_n, q'_s);
13           if q'_s = q_s then return SOLUTION;
14       if |T_b| > |T_a| then SWAP(T_a, T_b);
15   return FAILURE
```

Figure 5.24: A bidirectional RDT-based planner.

Line 14 represents an important step that balances the search. This is particularly important for a problem such as the bug trap shown in Figure 5.13b or the puzzle shown in Figure 1.2. If one of the trees is having trouble exploring, then it makes sense to focus more energy on it. Therefore, new exploration is always performed for the smaller tree. How is "smaller" defined? A simple criterion is to use the total number of vertices. Another reasonable criterion is to use the total length of all segments in the tree.

An unbalanced bidirectional search can instead be made by forcing the trees to be swapped in every iteration. Once the trees are swapped, then the roles are reversed. For example, after the first swap, T_b is extended in the same way as an integration in Figure 5.16, and if a new vertex q_s is added then an attempt is made to connect T_a to q_s.

One important concern exists when α is deterministic. It might be possible that even though α is dense, when the samples are divided among the trees, each may not receive a dense set. If each uses its own deterministic sequence, then this problem can be avoided. In the case of making a bidirectional RRT planner, the same (pseudo)random sequence can be used for each tree without encountering such troubles.

More than two trees If a dual-tree approach offers advantages over a single tree, then it is natural to ask whether growing three or more RDTs might be even better. This is particularly helpful for problems like the double bug trap in Figure 5.13c. New trees can be grown from parts of $\mathcal{C}$ that are difficult to reach. Controlling the number of trees and determining when to attempt connections

between them is difficult. Some interesting recent work has been done in this direction [82, 918, 919].

These additional trees could be started at arbitrary (possibly random) configurations. As more trees are considered, a complicated decision problem arises. The computation time must be divided between attempting to explore the space and attempting to connect trees to each other. It is also not clear which connections should be attempted. Many research issues remain in the development of this and other RRT-based planners. A limiting case would be to start a new tree from every sample in $\alpha(i)$ and to try to connect nearby trees whenever possible. This approach results in a graph that covers the space in a nice way that is independent of the query. This leads to the main topic of the next section.

5.6 Roadmap Methods for Multiple Queries

Previously, it was assumed that a single initial-goal pair was given to the planning algorithm. Suppose now that numerous initial-goal queries will be given to the algorithm, while keeping the robot model and obstacles fixed. This leads to a *multiple-query* version of the motion planning problem. In this case, it makes sense to invest substantial time to preprocess the models so that future queries can be answered efficiently. The goal is to construct a topological graph called a *roadmap*, which efficiently solves multiple initial-goal queries. Intuitively, the paths on the roadmap should be easy to reach from each of q_I and q_G, and the graph can be quickly searched for a solution. The general framework presented here was mainly introduced in [516] under the name *probabilistic roadmaps (PRMs)*. The probabilistic aspect, however, is not important to the method. Therefore, we call this family of methods *sampling-based roadmaps*. This distinguishes them from *combinatorial roadmaps*, which will appear in Chapter 6.

5.6.1 The Basic Method

Once again, let $\mathcal{G}(V, E)$ represent a topological graph in which V is a set of vertices and E is the set of paths that map into $\mathcal{C}_{free}$. Under the multiple-query philosophy, motion planning is divided into two phases of computation:

Preprocessing Phase: During the preprocessing phase, substantial effort is invested to build $\mathcal{G}$ in a way that is useful for quickly answering future queries. For this reason, it is called a *roadmap*, which in some sense should be accessible from every part of $\mathcal{C}_{free}$.

Query Phase: During the query phase, a pair, q_I and q_G, is given. Each configuration must be connected easily to $\mathcal{G}$ using a local planner. Following this, a discrete search is performed using any of the algorithms in Section 2.2 to obtain a sequence of edges that forms a path from q_I to q_G.

```
BUILD_ROADMAP
1  G.init(); i ← 0;
2  while i < N
3      if α(i) ∈ C_free then
4          G.add_vertex(α(i)); i ← i + 1;
5          for each q ∈ NEIGHBORHOOD(α(i),G)
6              if ((not G.same_component(α(i), q)) and CONNECT(α(i), q)) then
7                  G.add_edge(α(i), q);
```

Figure 5.25: The basic construction algorithm for sampling-based roadmaps. Note that i is not incremented if $\alpha(i)$ is in collision. This forces i to correctly count the number of vertices in the roadmap.

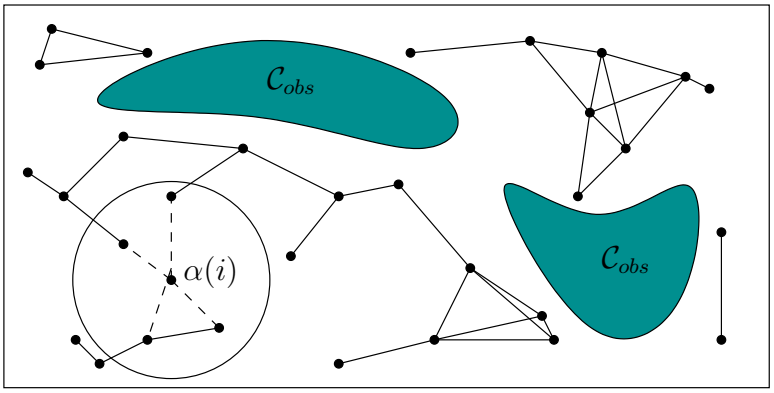


Figure 5.26: The sampling-based roadmap is constructed incrementally by attempting to connect each new sample, $\alpha(i)$, to nearby vertices in the roadmap.

Generic preprocessing phase Figure 5.25 presents an outline of the basic preprocessing phase, and Figure 5.26 illustrates the algorithm. As seen throughout this chapter, the algorithm utilizes a uniform, dense sequence α. In each iteration, the algorithm must check whether $\alpha(i) \in \mathcal{C}_{free}$. If $\alpha(i) \in \mathcal{C}_{obs}$, then it must continue to iterate until a collision-free sample is obtained. Once $\alpha(i) \in \mathcal{C}_{free}$, then in line 4 it is inserted as a vertex of $\mathcal{G}$. The next step is to try to connect $\alpha(i)$ to some nearby vertices, q, of $\mathcal{G}$. Each connection is attempted by the CONNECT function, which is a typical LPM (local planning method) from Section 5.4.1. In most implementations, this simply tests the shortest path between $\alpha(i)$ and q. Experimentally, it seems most efficient to use the multi-resolution, van der Corput–based method described at the end of Section 5.3.4 [379]. Instead of the shortest path, it is possible to use more sophisticated connection methods, such as the bidirectional algorithm in Figure 5.24. If the path is collision-free, then CONNECT returns TRUE.

The same_component condition in line 6 checks to make sure $\alpha(i)$ and q are in different components of $\mathcal{G}$ before wasting time on collision checking. This ensures that every time a connection is made, the number of connected components

of $\mathcal{G}$ is decreased. This can be implemented very efficiently (near constant time) using the previously mentioned *union-find algorithm* [243, 823]. In some implementations this step may be ignored, especially if it is important to generate multiple, alternative solutions. For example, it may be desirable to generate solution paths from different homotopy classes. In this case the condition (**not** $\mathcal{G}$.same_component$(\alpha(i), q)$) is replaced with $\mathcal{G}$.vertex_degree$(q) < K$, for some fixed K (e.g., K = 15).

Selecting neighboring samples Several possible implementations of line 5 can be made. In all of these, it seems best to sort the vertices that will be considered for connection in order of increasing distance from $\alpha(i)$. This makes sense because shorter paths are usually less costly to check for collision, and they also have a higher likelihood of being collision-free. If a connection is made, this avoids costly collision checking of longer paths to configurations that would eventually belong to the same connected component.

Several useful implementations of NEIGHBORHOOD are

1. **Nearest K:** The K closest points to $\alpha(i)$ are considered. This requires setting the parameter K (a typical value is 15). If you are unsure which implementation to use, try this one.

2. **Component K:** Try to obtain up to K nearest samples from each connected component of $\mathcal{G}$. A reasonable value is $K = 1$; otherwise, too many connections would be tried.

3. **Radius:** Take all points within a ball of radius r centered at $\alpha(i)$. An upper limit, K, may be set to prevent too many connections from being attempted. Typically, $K = 20$. A radius can be determined adaptively by shrinking the ball as the number of points increases. This reduction can be based on dispersion or discrepancy, if either of these is available for α. Note that if the samples are highly regular (e.g., a grid), then choosing the nearest K and taking points within a ball become essentially equivalent. If the point set is highly irregular, as in the case of random samples, then taking the nearest K seems preferable.

4. **Visibility:** In Section 5.6.2, a variant will be described for which it is worthwhile to try connecting α to all vertices in $\mathcal{G}$.

Note that all of these require $\mathcal{C}$ to be a metric space. One variation that has not yet been given much attention is to ensure that the directions of the NEIGHBORHOOD points relative to $\alpha(i)$ are distributed uniformly. For example, if the 20 closest points are all clumped together in the same direction, then it may be preferable to try connecting to a further point because it is in the opposite direction.

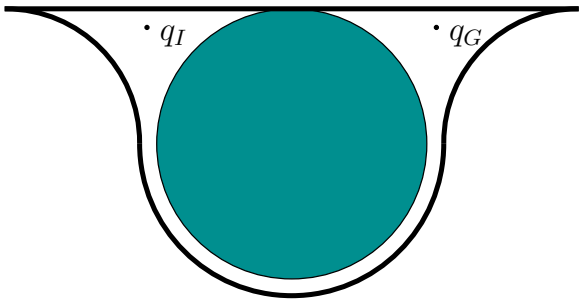


Figure 5.27: An example such as this is difficult for sampling-based roadmaps (in higher dimensional C-spaces) because some samples must fall along many points in the curved tube. Other methods, however, may be able to easily solve it.

Query phase In the query phase, it is assumed that $\mathcal{G}$ is sufficiently complete to answer many queries, each of which gives an initial configuration, q_I, and a goal configuration, q_G. First, the query phase pretends as if q_I and q_G were chosen from α for connection to $\mathcal{G}$. This requires running two more iterations of the algorithm in Figure 5.25. If q_I and q_G are successfully connected to other vertices in $\mathcal{G}$, then a search is performed for a path that connects the vertex q_I to the vertex q_G. The path in the graph corresponds directly to a path in $\mathcal{C}_{free}$, which is a solution to the query. Unfortunately, if this method fails, it cannot be determined conclusively whether a solution exists. If the dispersion is known for a sample sequence, α, then it is at least possible to conclude that no solution exists for the resolution of the planner. In other words, if a solution does exist, it would require the path to travel through a corridor no wider than the radius of the largest empty ball [600].

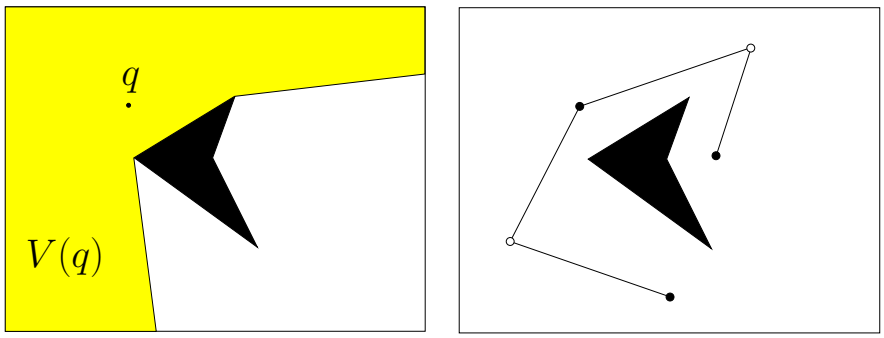
Some analysis There have been many works that analyze the performance of sampling-based roadmaps. The basic idea from one of them [69] is briefly presented here. Consider problems such as the one in Figure 5.27, in which the CONNECT method will mostly likely fail in the thin tube, even though a connection exists. The higher dimensional versions of these problems are even more difficult. Many planning problems involve moving a robot through an area with tight clearance. This generally causes narrow channels to form in $\mathcal{C}_{free}$, which leads to a challenging planning problem for the sampling-based roadmap algorithm. Finding the escape of a bug trap is also challenging, but for the roadmap methods, even traveling through a single corridor is hard (unless more sophisticated LPMs are used [479]).

Let $V(q)$ denote the set of all configurations that can be connected to q using the CONNECT method. Intuitively, this is considered as the set of all configurations that can be "seen" using line-of-sight visibility, as shown in Figure 5.28a

The ϵ-goodness of $\mathcal{C}_{free}$ is defined as

$$\epsilon(\mathcal{C}_{free}) = \min_{q \in \mathcal{C}_{free}} \left\{ \frac{\mu(V(q))}{\mu(\mathcal{C}_{free})} \right\}, \tag{5.41}$$

in which μ represents the measure. Intuitively, $\epsilon(\mathcal{C}_{free})$ represents the small frac-



(a) Visibility definition (b) Visibility roadmap

Figure 5.28: (a) $V(q)$ is the set of points reachable by the LPM from q. (b) A visibility roadmap has two kinds of vertices: guards, which are shown in black, and connectors, shown in white. Guards are not allowed to see other guards. Connectors must see at least two guards.

tion of $\mathcal{C}_{free}$ that is visible from any point. In terms of ϵ and the number of vertices in $\mathcal{G}$, bounds can be established that yield the probability that a solution will be found [69]. The main difficulties are that the ϵ-goodness concept is very conservative (it uses worst-case analysis over all configurations), and ϵ-goodness is defined in terms of the structure of $\mathcal{C}_{free}$, which cannot be computed efficiently. This result and other related results help to gain a better understanding of sampling-based planning, but such bounds are difficult to apply to particular problems to determine whether an algorithm will perform well.

5.6.2 Visibility Roadmap

One of the most useful variations of sampling-based roadmaps is the *visibility roadmap* [885]. The approach works very hard to ensure that the roadmap representation is small yet covers $\mathcal{C}_{free}$ well. The running time is often greater than the basic algorithm in Figure 5.25, but the extra expense is usually worthwhile if the multiple-query philosophy is followed to its fullest extent.

The idea is to define two different kinds of vertices in $\mathcal{G}$:

Guards: To become a *guard*, a vertex, q must not be able to see other guards. Thus, the visibility region, $V(q)$, must be empty of other guards.

Connectors: To become a *connector*, a vertex, q, must see at least two guards. Thus, there exist guards q_1 and q_2, such that $q \in V(q_1) \cap V(q_2)$.

The roadmap construction phase proceeds similarly to the algorithm in Figure 5.25. The *neighborhood function* returns all vertices in $\mathcal{G}$. Therefore, for each new sample $\alpha(i)$, an attempt is made to connect it to every other vertex in $\mathcal{G}$.

The main novelty of the visibility roadmap is using a strong criterion to determine whether to keep $\alpha(i)$ and its associated edges in $\mathcal{G}$. There are three possible cases for each $\alpha(i)$:

1. The new sample, $\alpha(i)$, is not able to connect to any guards. In this case, $\alpha(i)$ earns the privilege of becoming a guard itself and is inserted into $\mathcal{G}$.

2. The new sample can connect to guards from at least two different connected components of $\mathcal{G}$. In this case, it becomes a connector that is inserted into $\mathcal{G}$ along with its associated edges, which connect it to these guards from different components.

3. Neither of the previous two conditions were satisfied. This means that the sample could only connect to guards in the same connected component. In this case, $\alpha(i)$ is discarded.

The final condition causes a dramatic reduction in the number of roadmap vertices.

One problem with this method is that it does not allow guards to be deleted in favor of better guards that might appear later. The placement of guards depends strongly on the order in which samples appear in α. The method may perform poorly if guards are not positioned well early in the sequence. It would be better to have an adaptive scheme in which guards could be reassigned in later iterations as better positions become available. Accomplishing this efficiently remains an open problem. Note the algorithm is still probabilistically complete using random sampling or resolution complete if α is dense, even though many samples are rejected.

5.6.3 Heuristics for Improving Roadmaps

The quest to design a good roadmap through sampling has spawned many heuristic approaches to sampling and making connections in roadmaps. Most of these exploit properties that are specific to the shape of the C-space and/or the particular geometry and kinematics of the robot and obstacles. The emphasis is usually on finding ways to dramatically reduce the number or required samples. Several of these methods are briefly described here.

Vertex enhancement [516] This heuristic strategy focuses effort on vertices that were difficult to connect to other vertices in the roadmap construction algorithm in Figure 5.25. A probability distribution, $P(v)$, is defined over the vertices $v \in V$. A number of iterations are then performed in which a vertex is sampled from V according to $P(v)$, and then some random motions are performed from v to try to reach new configurations. These new configurations are added as vertices, and attempts are made to connect them to other vertices, as selected by the NEIGHBORHOOD function in an ordinary iteration of the algorithm in Figure 5.25. A recommended heuristic [516] for defining $P(v)$ is to define a statistic for each v as $n_f/(n_t + 1)$, in which n_t is the total number of connections attempted for

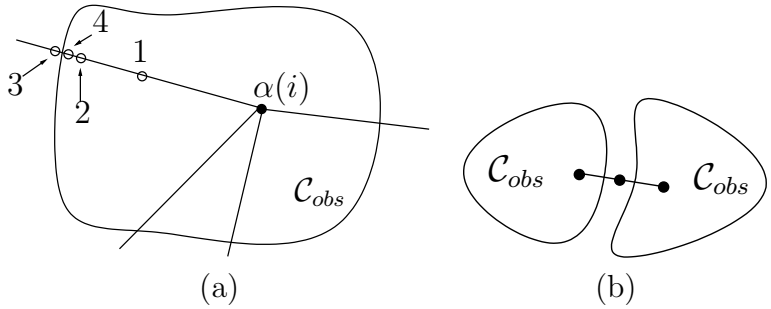


Figure 5.29: (a) To obtain samples along the boundary, binary search is used along random directions from a sample in $\mathcal{C}_{obs}$. (b) The bridge test finds narrow corridors by examining a triple of nearby samples along a line.

v, and n_f is the number of times these attempts failed. The probability $P(v)$ is assigned as $n_f/(n_t+1)m$, in which m is the sum of the statistics over all $v \in V$ (this normalizes the statistics to obtain a valid probability distribution).

Sampling on the $\mathcal{C}_{free}$ boundary [22, 26] This scheme is based on the intuition that it is sometimes better to sample along the boundary, $\partial\mathcal{C}_{free}$, rather than waste samples on large areas of $\mathcal{C}_{free}$ that might be free of obstacles. Figure 5.29a shows one way in which this can be implemented. For each sample of $\alpha(i)$ that falls into $\mathcal{C}_{obs}$, a number of random directions are chosen in $\mathcal{C}$; these directions can be sampled using the $\mathbb{S}^n$ sampling method from Section 5.2.2. For each direction, a binary search is performed to get a sample in $\mathcal{C}_{free}$ that is as close as possible to $\mathcal{C}_{obs}$. The order of point evaluation in the binary search is shown in Figure 5.29a. Let $\tau : [0,1]$ denote the path for which $\tau(0) \in \mathcal{C}_{obs}$ and $\tau(1) \in \mathcal{C}_{free}$. In the first step, test the midpoint, $\tau(1/2)$. If $\tau(1/2) \in \mathcal{C}_{free}$, this means that $\partial\mathcal{C}_{free}$ lies between $\tau(0)$ and $\tau(1/2)$; otherwise, it lies between $\tau(1/2)$ and $\tau(1)$. The next iteration selects the midpoint of the path segment that contains $\partial\mathcal{C}_{free}$. This will be either $\tau(1/4)$ or $\tau(3/4)$. The process continues recursively until the desired resolution is obtained.

Gaussian sampling [132] The Gaussian sampling strategy follows some of the same motivation for sampling on the boundary. In this case, the goal is to obtain points near $\partial\mathcal{C}_{free}$ by using a Gaussian distribution that biases the samples to be closer to $\partial\mathcal{C}_{free}$, but the bias is gentler, as prescribed by the variance parameter of the Gaussian. The samples are generated as follows. Generate one sample, $q_1 \in \mathcal{C}$, uniformly at random. Following this, generate another sample, $q_2 \in \mathcal{C}$, according to a Gaussian with mean q_1; the distribution must be adapted for any topological identifications and/or boundaries of $\mathcal{C}$. If one of q_1 or q_2 lies in $\mathcal{C}_{free}$ and the other lies in $\mathcal{C}_{obs}$, then the one that lies in $\mathcal{C}_{free}$ is kept as a vertex in the roadmap. For some examples, this dramatically prunes the number of required vertices.

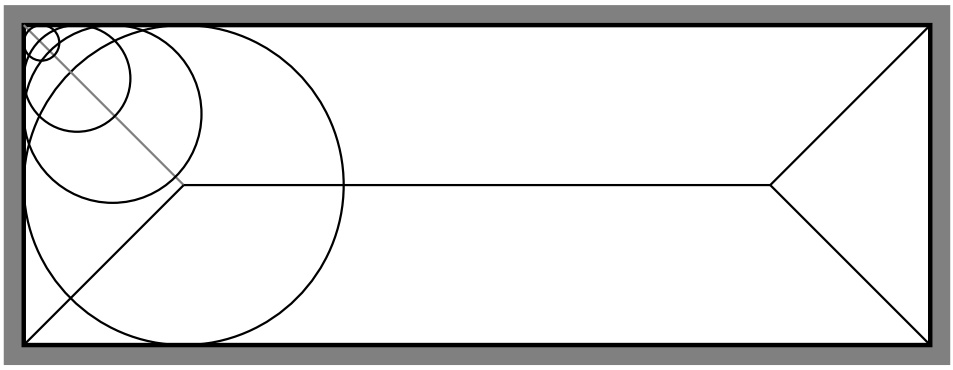


Figure 5.30: The medial axis is traced out by the centers of the largest inscribed balls. The five line segments inside of the rectangle correspond to the medial axis.

Bridge-test sampling [465] The Gaussian sampling strategy decides to keep a point based in part on testing a pair of samples. This idea can be carried one step further to obtain a *bridge test*, which uses three samples along a line segment. If the samples are arranged as shown in Figure 5.29b, then the middle sample becomes a roadmap vertex. This is based on the intuition that narrow corridors are thin in at least one direction. The bridge test indicates that a point lies in a thin corridor, which is often an important place to locate a vertex.

Medial-axis sampling [455, 635, 971] Rather than trying to sample close to the boundary, another strategy is to force the samples to be as far from the boundary as possible. Let (X, ρ) be a metric space. Let a *maximal ball* be a ball $B(x, r) \subseteq X$ such that no other ball can be a proper subset. The centers of all maximal balls trace out a one-dimensional set of points referred to as the *medial axis*. A simple example of a medial axis is shown for a rectangular subset of $\mathbb{R}^2$ in Figure 5.30. The medial axis in $\mathcal{C}_{free}$ is based on the largest balls that can be inscribed in $\mathrm{cl}(\mathcal{C}_{free})$. Sampling on the medial axis is generally difficult, especially because the representation of $\mathcal{C}_{free}$ is implicit. Distance information from collision checking can be used to start with a sample, $\alpha(i)$, and iteratively perturb it to increase its distance from $\partial\mathcal{C}_{free}$ [635, 971]. Sampling on the medial axis of $W \setminus \mathcal{O}$ has also been proposed [455]. In this case, the medial axis in $\mathcal{W} \setminus \mathcal{O}$ is easier to compute, and it can be used to heuristically guide the placement of good roadmap vertices in $\mathcal{C}_{free}$.

Further Reading

Unlike the last two chapters, the material of Chapter 5 is a synthesis of very recent research results. Some aspects of sampling-based motion planning are still evolving. Early approaches include [70, 144, 193, 280, 282, 329, 330, 658, 760]. The Gilbert-Johnson-Keerthi algorithm [388] is an early collision detection approach that helped inspire sampling-based motion planning; see [472] and [588] for many early references.

In much of the early work, randomization appeared to be the main selling point; however, more recently it has been understood that deterministic sampling can work at least as well while obtaining resolution completeness. For a more recent survey of sampling-based motion planning, see [640].

Section 5.1 is based on material from basic mathematics books. For a summary of basic theorems and numerous examples of metric spaces, see [696]. More material appears in basic point-set topology books (e.g., [451, 496]) and analysis books (e.g., [346]). Metric issues in the context of sampling-based motion planning are discussed in [21, 609]. Measure theory is most often introduced in the context of real analysis [346, 425, 546, 836, 837]. More material on Haar measure appears in [425].

Section 5.2 is mainly inspired by literature on Monte Carlo and quasi–Monte Carlo methods for numerical integration and optimization. An excellent source of material is [738]. Other important references for further reading include [191, 540, 682, 937, 938]. Sampling issues in the context of motion planning are considered in [380, 559, 600, 639, 987]. Comprehensive introductions to pure Monte Carlo algorithms appear in [341, 502]. The original source for the Monte Carlo method is [695]. For a survey on algorithms that compute Voronoi diagrams, see [54].

For further reading on collision detection (beyond Section 5.3), see the surveys in [488, 637, 638, 703]. Hierarchical collision detection is covered in [406, 638, 702]. The incremental collision detection ideas in Section 5.3.3 were inspired by the algorithm [636] and V-Clip [247, 702]. Distance computation is covered in [167, 306, 387, 406, 413, 702, 807]. A method suited for detecting self-collisions of linkages appears in [653]. A combinatorial approach to collision detection for motion planning appears in [855]. Numerous collision detection packages are available for use in motion planning research. One of the most widely used is PQP because it works well for any mess of 3D triangles [948].

The incremental sampling and searching framework was synthesized by unifying ideas from many planning methods. Some of these include grid-based search [71, 548, 620] and probabilistic roadmaps (PRMs) [516]. Although the PRM was developed for multiple queries, the single-query version developed in [125] helped shed light on the connection to earlier planning methods. This even led to grid-based variants of PRMs [123, 600]. Another single-query variant is presented in [845].

RDTs were developed in the literature mainly as RRTs, and were introduced in [598, 610]. RRTs have been used in several applications, and many variants have been developed [82, 138, 150, 200, 224, 244, 265, 362, 393, 495, 499, 498, 528, 631, 641, 642, 918, 919, 949, 979, 986]. Originally, they were developed for planning under differential constraints, but most of their applications to date have been for ordinary motion planning. For more information on efficient nearest-neighbor searching, see the recent survey [475], and [46, 47, 48, 52, 99, 230, 365, 476, 538, 758, 908, 989].

Section 5.6 is based mainly on the PRM framework [516]. The "probabilistic" part is not critical to the method; thus, it was referred to here as a *sampling-based roadmap*. A related precursor to the PRM was proposed in [390, 391]. The PRM has been widely used in practice, and many variants have been proposed [1, 23, 61, 62, 125, 161, 181, 244, 479, 544, 600, 627, 628, 740, 784, 792, 885, 900, 950, 971, 979, 995]. An experimental comparison of many of these variants appears in [380]. Some analysis of PRMs appears in [69, 467, 573]. In some works, the term PRM has been applied to virtually any

sampling-based planning algorithm (e.g., [467]); however, in recent years the term has been used more consistently with its original meaning in [516].

Many other methods and issues fall outside of the scope of this chapter. Several interesting methods based on *approximate cell decomposition* [144, 328, 646, 658] can be considered as a form of sampling-based motion planning. A sampling-based method of developing global potential functions appears in [124]. Other sampling-based planning algorithms appear in [194, 348, 417, 418, 463]. The algorithms of this chapter are generally unable to guarantee that a solution does not exist for a motion planning problem. It is possible, however, to use sampling-based techniques to establish in finite time that no solution exists [75]. Such a result is called a *disconnection proof*. Parallelization issues have also been investigated in the context of sampling-based motion planning [82, 177, 183, 257, 795].

Exercises

1. Prove that the Cartesian product of a metric space is a metric space by taking a linear combination as in (5.4).

2. Prove or disprove: If ρ is a metric, then ρ^2 is a metric.

3. Determine whether the following function is a metric on any topological space: X: $\rho(x, x') = 1$ is $x \neq x'$; otherwise, $\rho(x, x') = 0$.

4. State and prove whether or not (5.28) yields a metric space on $\mathcal{C} = SE(3)$, assuming that the two sets are rigid bodies.

5. The dispersion definition given in (5.19) is based on the worst case. Consider defining the *average dispersion*:

$$\bar{\delta}(P) = \frac{1}{\mu(X)} \int_X \min_{p \in P} \{\rho(x, p)\} dx. \tag{5.42}$$

   Describe a Monte Carlo (randomized) method to approximately evaluate (5.42).

6. Determine the average dispersion (as a function of i) for the van der Corput sequence (base 2) on $[0, 1]/\sim$.

7. Show that using the Lebesgue measure on $\mathbb{S}^3$ (spreading mass around uniformly on $\mathbb{S}^3$) yields the Haar measure for $SO(3)$.

8. Is the Haar measure useful in selecting an appropriate C-space metric? Explain.

9. Determine an expression for the (worst-case) dispersion of the ith sample in the base-p (Figure 5.2 shows base-2) van der Corput sequence in $[0, 1]/\sim$, in which 0 and 1 are identified.

10. Determine the dispersion of the following sequence on $[0, 1]$. The first point is $\alpha(1) = 1$. For each $i > 1$, let $c_i = \ln(2i - 3)/\ln 4$ and $\alpha(i) = c_i - \lfloor c_i \rfloor$. It turns out that this sequence achieves the best asymptotic dispersion possible, even in terms of the preceding constant. Also, the points are not uniformly distributed. Can you explain why this happens? [It may be helpful to plot the points in the sequence.]

11. Prove that (5.20) holds.

12. Prove that (5.23) holds.

13. Show that for any given set of points in $[0,1]^n$, a range space $\mathcal{R}$ can be designed so that the discrepancy is as close as desired to 1.

14. Suppose $\mathcal{A}$ is a rigid body in $\mathbb{R}^3$ with a fixed orientation specified by a quaternion, h. Suppose that h is perturbed a small amount to obtain another quaternion, h' (no translation occurs). Construct a good upper bound on distance traveled by points on $\mathcal{A}$, expressed in terms of the change in the quaternion.

15. Design combinations of robots and obstacles in $\mathcal{W}$ that lead to C-space obstacles resembling bug traps.

16. How many k-neighbors can there be at most in an n-dimensional grid with $1 \leq k \leq n$?

17. In a high-dimensional grid, it becomes too costly to consider all $3^n - 1$ n-neighbors. It might not be enough to consider only $2n$ 1-neighbors. Determine a scheme for selecting neighbors that are spatially distributed in a good way, but without requiring too many. For example, what is a good way to select 50 neighbors for a grid in $\mathbb{R}^{10}$?

18. Explain the difference between searching an implicit, high-resolution grid and growing search trees directly on the C-space without a grid.

19. Improve the bound in (5.31) by considering the fact that rotating points trace out a circle, instead of a straight line.

20. (Open problem) Prove there are $n+1$ main branches for an RRT starting from the center of an "infinite" n-dimensional ball in $\mathbb{R}^n$. The directions of the branches align with the vertices of a regular simplex centered at the initial configuration.

Implementations

21. Implement 2D incremental collision checking for convex polygons to obtain "near constant time" performance.

22. Implement the sampling-based roadmap approach. Select an appropriate family of motion planning problems: 2D rigid bodies, 2D chains of bodies, 3D rigid bodies, etc.

    (a) Compare the roadmaps obtained using visibility-based sampling to those obtained for the ordinary sampling method.

    (b) Study the sensitivity of the method with respect to the particular NEIGHBORHOOD method.

    (c) Compare random and deterministic sampling methods.

(d) Use the bridge test to attempt to produce better samples.

23. Implement the balanced, bidirectional RRT planning algorithm.

    (a) Study the effect of varying the amount of intermediate vertices created along edges.

    (b) Try connecting to the random sample using more powerful descent functions.

    (c) Explore the performance gains from using Kd-trees to select nearest neighbors.

24. Make an RRT-based planning algorithm that uses more than two trees. Carefully resolve issues such as the maximum number of allowable trees, when to start a tree, and when to attempt connections between trees.

25. Implement both the expansive-space planner and the RRT, and conduct comparative experiments on planning problems. For the full set of problems, keep the algorithm parameters fixed.

26. Implement a sampling-based algorithm that computes collision-free paths for a rigid robot that can translate or rotate on any of the flat 2D manifolds shown in Figure 4.5.

Chapter 6

Combinatorial Motion Planning

Combinatorial approaches to motion planning find paths through the continuous configuration space without resorting to approximations. Due to this property, they are alternatively referred to as *exact* algorithms. This is in contrast to the sampling-based motion planning algorithms from Chapter 5.

6.1 Introduction

All of the algorithms presented in this chapter are *complete*, which means that for any problem instance (over the space of problems for which the algorithm is designed), the algorithm will either find a solution or will correctly report that no solution exists. By contrast, in the case of sampling-based planning algorithms, weaker notions of completeness were tolerated: resolution completeness and probabilistic completeness.

Representation is important When studying combinatorial motion planning algorithms, it is important to carefully consider the definition of the input. What is the representation used for the robot and obstacles? What set of transformations may be applied to the robot? What is the dimension of the world? Are the robot and obstacles convex? Are they piecewise linear? The specification of possible inputs defines a set of problem instances on which the algorithm will operate. If the instances have certain convenient properties (e.g., low dimensionality, convex models), then a combinatorial algorithm may provide an elegant, practical solution. If the set of instances is too broad, then a requirement of both completeness and practical solutions may be unreasonable. Many general formulations of general motion planning problems are PSPACE-hard[1]; therefore, such a hope appears unattainable. Nevertheless, there exist general, complete motion planning algorithms. Note that focusing on the representation is the opposite philosophy from sampling-based planning, which hides these issues in the collision detection module.

[1]This implies NP-hard. An overview of such complexity statements appears in Section 6.5.1.

Reasons to study combinatorial methods There are generally two good reasons to study combinatorial approaches to motion planning:

1. In many applications, one may only be interested in a special class of planning problems. For example, the world might be 2D, and the robot might only be capable of translation. For many special classes, elegant and efficient algorithms can be developed. These algorithms are complete, do not depend on approximation, and can offer much better performance than sampling-based planning methods, such as those in Chapter 5.

2. It is both interesting and satisfying to know that there are complete algorithms for an extremely broad class of motion planning problems. Thus, even if the class of interest does not have some special limiting assumptions, there still exist general-purpose tools and algorithms that can solve it. These algorithms also provide theoretical upper bounds on the time needed to solve motion planning problems.

Warning: Some methods are impractical Be careful not to make the wrong assumptions when studying the algorithms of this chapter. A few of them are efficient and easy to implement, but many might be neither. Even if an algorithm has an amazing asymptotic running time, it might be close to impossible to implement. For example, one of the most famous algorithms from computational geometry can split a simple[2] polygon into triangles in $O(n)$ time for a polygon with n edges [190]. This is so amazing that it was covered in the *New York Times*, but the algorithm is so complicated that it is doubtful that anyone will ever implement it. Sometimes it is preferable to use an algorithm that has worse theoretical running time but is much easier to understand and implement. In general, though, it is valuable to understand both kinds of methods and decide on the trade-offs for yourself. It is also an interesting intellectual pursuit to try to determine how efficiently a problem can be solved, even if the result is mainly of theoretical interest. This might motivate others to look for simpler algorithms that have the same or similar asymptotic running times.

Roadmaps Virtually all combinatorial motion planning approaches construct a *roadmap* along the way to solving queries. This notion was introduced in Section 5.6, but in this chapter stricter requirements are imposed in the roadmap definition because any algorithm that constructs one needs to be complete. Some of the algorithms in this chapter first construct a cell decomposition of $\mathcal{C}_{free}$ from which the roadmap is consequently derived. Other methods directly construct a roadmap without the consideration of cells.

Let $\mathcal{G}$ be a topological graph (defined in Example 4.6) that maps into $\mathcal{C}_{free}$. Furthermore, let $S \subset \mathcal{C}_{free}$ be the swath, which is set of all points reached by $\mathcal{G}$, as defined in (5.40). The graph $\mathcal{G}$ is called a *roadmap* if it satisfies two important conditions:

[2] A polygonal region that has no holes.

1. **Accessibility:** From any $q \in \mathcal{C}_{free}$, it is simple and efficient to compute a path $\tau : [0,1] \rightarrow \mathcal{C}_{free}$ such that $\tau(0) = q$ and $\tau(1) = s$, in which s may be any point in S. Usually, s is the closest point to q, assuming $\mathcal{C}$ is a metric space.

2. **Connectivity-preserving:** Using the first condition, it is always possible to connect some q_I and q_G to some s_1 and s_2, respectively, in S. The second condition requires that if there exists a path $\tau : [0,1] \rightarrow \mathcal{C}_{free}$ such that $\tau(0) = q_I$ and $\tau(1) = q_G$, then there also exists a path $\tau' : [0,1] \rightarrow S$, such that $\tau'(0) = s_1$ and $\tau'(1) = s_2$. Thus, solutions are not missed because $\mathcal{G}$ fails to capture the connectivity of $\mathcal{C}_{free}$. This ensures that complete algorithms are developed.

By satisfying these properties, a roadmap provides a discrete representation of the continuous motion planning problem without losing any of the original connectivity information needed to solve it. A query, (q_I, q_G), is solved by connecting each query point to the roadmap and then performing a discrete graph search on $\mathcal{G}$. To maintain completeness, the first condition ensures that any query can be connected to $\mathcal{G}$, and the second condition ensures that the search always succeeds if a solution exists.

6.2 Polygonal Obstacle Regions

Rather than diving into the most general forms of combinatorial motion planning, it is helpful to first see several methods explained for a case that is easy to visualize. Several elegant, straightforward algorithms exist for the case in which $\mathcal{C} = \mathbb{R}^2$ and $\mathcal{C}_{obs}$ is polygonal. Most of these cannot be directly extended to higher dimensions; however, some of the general principles remain the same. Therefore, it is very instructive to see how combinatorial motion planning approaches work in two dimensions. There are also applications where these algorithms may directly apply. One example is planning for a small mobile robot that may be modeled as a point moving in a building that can be modeled with a 2D polygonal floor plan.

After covering representations in Section 6.2.1, Sections 6.2.2–6.2.4 present three different algorithms to solve the same problem. The one in Section 6.2.2 first performs *cell decomposition* on the way to building the roadmap, and the ones in Sections 6.2.3 and 6.2.4 directly produce a roadmap. The algorithm in Section 6.2.3 computes maximum clearance paths, and the one in Section 6.2.4 computes shortest paths (which consequently have no clearance).

6.2.1 Representation

Assume that $\mathcal{W} = \mathbb{R}^2$; the obstacles, $\mathcal{O}$, are polygonal; and the robot, $\mathcal{A}$, is a polygonal body that is only capable of translation. Under these assumptions, $\mathcal{C}_{obs}$ will be polygonal. For the special case in which $\mathcal{A}$ is a point in $\mathcal{W}$, $\mathcal{O}$ maps directly

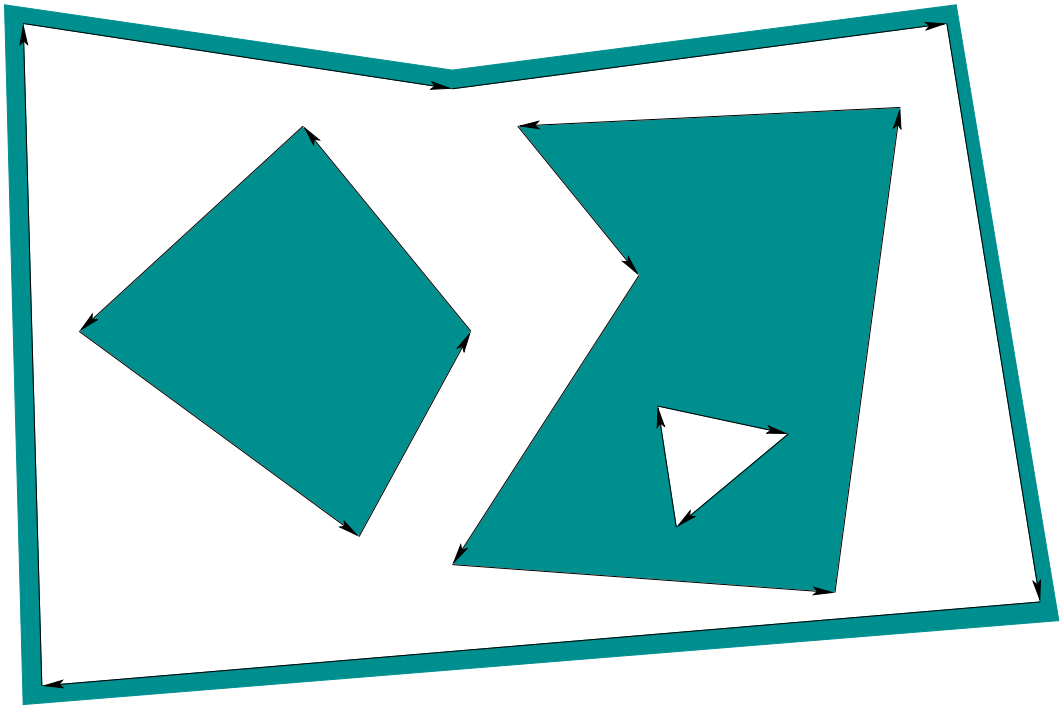


Figure 6.1: A polygonal model specified by four oriented simple polygons.

to $\mathcal{C}_{obs}$ without any distortion. Thus, the problems considered in this section may also be considered as planning for a *point robot*. If $\mathcal{A}$ is not a point robot, then the Minkowski difference, (4.37), of $\mathcal{O}$ and $\mathcal{A}$ must be computed. For the case in which both $\mathcal{A}$ and each component of $\mathcal{O}$ are convex, the algorithm in Section 4.3.2 can be applied to compute each component of $\mathcal{C}_{obs}$. In general, both $\mathcal{A}$ and $\mathcal{O}$ may be nonconvex. They may even contain holes, which results in a $\mathcal{C}_{obs}$ model such as that shown in Figure 6.1. In this case, $\mathcal{A}$ and $\mathcal{O}$ may be decomposed into convex components, and the Minkowski difference can be computed for each pair of components. The decompositions into convex components can actually be performed by adapting the cell decomposition algorithm that will be presented in Section 6.2.2. Once the Minkowski differences have been computed, they need to be merged to obtain a representation that can be specified in terms of simple polygons, such as those in Figure 6.1. An efficient algorithm to perform this merging is given in Section 2.4 of [264]. It can also be based on many of the same principles as the planning algorithm in Section 6.2.2.

To implement the algorithms described in this section, it will be helpful to have a data structure that allows convenient access to the information contained in a model such as Figure 6.1. How is the outer boundary represented? How are holes inside of obstacles represented? How do we know which holes are inside of which obstacles? These questions can be efficiently answered by using the doubly connected edge list data structure, which was described in Section 3.1.3 for consistent labeling of polyhedral faces. We will need to represent models, such

as the one in Figure 6.1, and any other information that planning algorithms need to maintain during execution. There are three different records:

Vertices: Every vertex v contains a pointer to a point $(x, y) \in \mathcal{C} = \mathbb{R}^2$ and a pointer to some half-edge that has v as its origin.

Faces: Every face has one pointer to a half-edge on the boundary that surrounds the face; the pointer value is NIL if the face is the outermost boundary. The face also contains a list of pointers for each connected component (i.e., hole) that is contained inside of that face. Each pointer in the list points to a half-edge of the component's boundary.

Half-edges: Each half-edge is directed so that the obstacle portion is always to its left. It contains five different pointers. There is a pointer to its *origin vertex*. There is a *twin* half-edge pointer, which may point to a half-edge that runs in the opposite direction (see Section 3.1.3). If the half-edge borders an obstacle, then this pointer is NIL. Half-edges are always arranged in circular chains to form the boundary of a face. Such chains are oriented so that the obstacle portion (or a twin half-edge) is always to its left. Each half-edge stores a pointer to its internal face. It also contains pointers to the next and previous half-edges in the circular chain of half-edges.

For the example in Figure 6.1, there are four circular chains of half-edges that each bound a different face. The face record of the small triangular hole points to the obstacle face that contains the hole. Each obstacle contains a pointer to the face represented by the outermost boundary. By consistently assigning orientations to the half-edges, circular chains that bound an obstacle always run counterclockwise, and chains that bound holes run clockwise. There are no twin half-edges because all half-edges bound part of $\mathcal{C}_{obs}$. The doubly connected edge list data structure is general enough to allow extra edges to be inserted that slice through $\mathcal{C}_{free}$. These edges will not be on the border of $\mathcal{C}_{obs}$, but they can be managed using twin half-edge pointers. This will be useful for the algorithm in Section 6.2.2.

6.2.2 Vertical Cell Decomposition

Cell decompositions will be defined formally in Section 6.3, but here we use the notion informally. Combinatorial methods must construct a finite data structure that exactly encodes the planning problem. Cell decomposition algorithms achieve this partitioning of $\mathcal{C}_{free}$ into a finite set of regions called *cells*. The term *k-cell* refers to a k-dimensional cell. The cell decomposition should satisfy three properties:

1. Computing a path from one point to another inside of a cell must be trivially easy. For example, if every cell is convex, then any pair of points in a cell can be connected by a line segment.

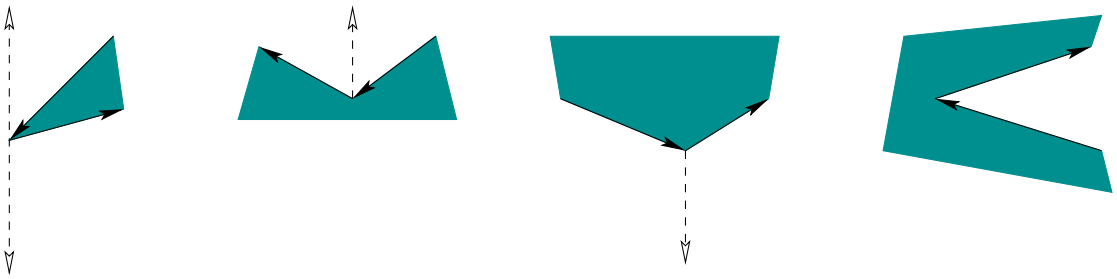


Figure 6.2: There are four general cases: 1) extending upward and downward, 2) upward only, 3) downward only, and 4) no possible extension.

2. Adjacency information for the cells can be easily extracted to build the roadmap.

3. For a given q_I and q_G, it should be efficient to determine which cells contain them.

If a cell decomposition satisfies these properties, then the motion planning problem is reduced to a graph search problem. Once again the algorithms of Section 2.2 may be applied; however, in the current setting, the entire graph, $\mathcal{G}$, is usually known in advance.[3] This was not assumed for discrete planning problems.

Defining the vertical decomposition We next present an algorithm that constructs a *vertical cell decomposition* [189], which partitions $\mathcal{C}_{free}$ into a finite collection of 2-cells and 1-cells. Each 2-cell is either a trapezoid that has vertical sides or a triangle (which is a degenerate trapezoid). For this reason, the method is sometimes called *trapezoidal decomposition*. The decomposition is defined as follows. Let P denote the set of vertices used to define $\mathcal{C}_{obs}$. At every $p \in P$, try to extend rays upward and downward through $\mathcal{C}_{free}$, until $\mathcal{C}_{obs}$ is hit. There are four possible cases, as shown in Figure 6.2, depending on whether or not it is possible to extend in each of the two directions. If $\mathcal{C}_{free}$ is partitioned according to these rays, then a vertical decomposition results. Extending these rays for the example in Figure 6.3a leads to the decomposition of $\mathcal{C}_{free}$ shown in Figure 6.3b. Note that only trapezoids and triangles are obtained for the 2-cells in $\mathcal{C}_{free}$.

Every 1-cell is a vertical segment that serves as the border between two 2-cells. We must ensure that the topology of $\mathcal{C}_{free}$ is correctly represented. Recall that $\mathcal{C}_{free}$ was defined to be an open set. Every 2-cell is actually defined to be an open set in $\mathbb{R}^2$; thus, it is the interior of a trapezoid or triangle. The 1-cells are the interiors of segments. It is tempting to make 0-cells, which correspond to the endpoints of segments, but these are not allowed because they lie in $\mathcal{C}_{obs}$.

[3] Exceptions to this are some algorithms mentioned in Section 6.5.3, which obtain greater efficiency by only maintaining one connected component of $\mathcal{C}_{obs}$.

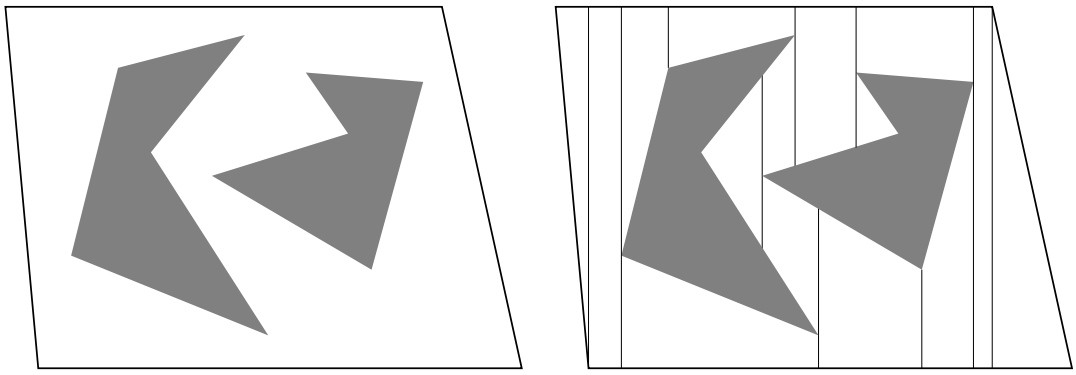


Figure 6.3: The vertical cell decomposition method uses the cells to construct a roadmap, which is searched to yield a solution to a query.

General position issues What if two points along $\mathcal{C}_{obs}$ lie on a vertical line that slices through $\mathcal{C}_{free}$? What happens when one of the edges of $\mathcal{C}_{obs}$ is vertical? These are special cases that have been ignored so far. Throughout much of combinatorial motion planning it is common to ignore such special cases and assume $\mathcal{C}_{obs}$ is in *general position*. This usually means that if all of the data points are perturbed by a small amount in some random direction, the probability that the special case remains is zero. Since a vertical edge is no longer vertical after being slightly perturbed, it is not in general position. The general position assumption is usually made because it greatly simplifies the presentation of an algorithm (and, in some cases, its asymptotic running time is even lower). In practice, however, this assumption can be very frustrating. Most of the implementation time is often devoted to correctly handling such special cases. Performing random perturbations may avoid this problem, but it tends to unnecessarily complicate the solutions. For the vertical decomposition, the problems are not too difficult to handle without resorting to perturbations; however, in general, it is important to be aware of this difficulty, which is not as easy to fix in most other settings.

Defining the roadmap To handle motion planning queries, a roadmap is constructed from the vertical cell decomposition. For each cell C_i, let q_i denote a designated *sample point* such that $q_i \in C_i$. The sample points can be selected as the cell centroids, but the particular choice is not too important. Let $\mathcal{G}(V, E)$ be a topological graph defined as follows. For every cell, C_i, define a vertex $q_i \in V$. There is a vertex for every 1-cell and every 2-cell. For each 2-cell, define an edge from its sample point to the sample point of every 1-cell that lies along its boundary. Each edge is a line-segment path between the sample points of the cells. The resulting graph is a roadmap, as depicted in Figure 6.4. The accessibility condition is satisfied because every sample point can be reached by a straight-line path thanks to the convexity of every cell. The connectivity condition is also satisfied

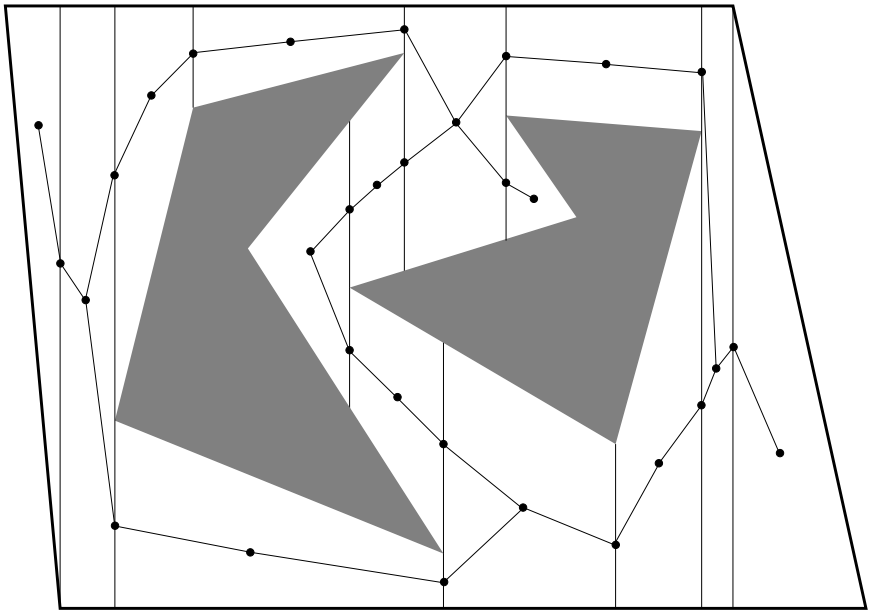


Figure 6.4: The roadmap derived from the vertical cell decomposition.

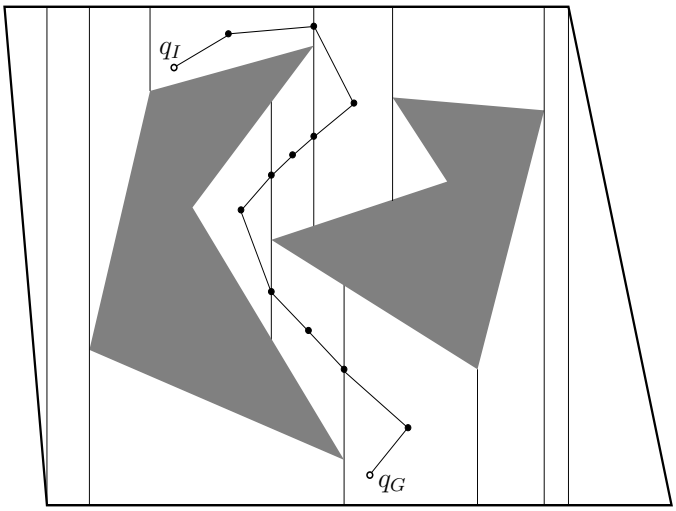


Figure 6.5: An example solution path.

because $\mathcal{G}$ is derived directly from the cell decomposition, which also preserves the connectivity of $\mathcal{C}_{free}$. Once the roadmap is constructed, the cell information is no longer needed for answering planning queries.

Solving a query Once the roadmap is obtained, it is straightforward to solve a motion planning query, (q_I, q_G). Let C_0 and C_k denote the cells that contain q_I and q_G, respectively. In the graph $\mathcal{G}$, search for a path that connects the sample point of C_0 to the sample point of C_k. If no such path exists, then the planning algorithm correctly declares that no solution exists. If one does exist, then let C_1, C_2, ..., C_{k-1} denote the sequence of 1-cells and 2-cells visited along the computed path in $\mathcal{G}$ from C_0 to C_k.

A solution path can be formed by simply "connecting the dots." Let q_0, q_1, q_2, ..., q_{k-1}, q_k, denote the sample points along the path in $\mathcal{G}$. There is one sample point for every cell that is crossed. The solution path, $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$, is formed by setting $\tau(0) = q_I$, $\tau(1) = q_G$, and visiting each of the points in the sequence from q_0 to q_k by traveling along the shortest path. For the example, this leads to the solution shown in Figure 6.5. In selecting the sample points, it was important to ensure that each path segment from the sample point of one cell to the sample point of its neighboring cell is collision-free.[4]

Computing the decomposition The problem of efficiently computing the decomposition has not yet been considered. Without concern for efficiency, the problem appears simple enough that all of the required steps can be computed by brute-force computations. If $\mathcal{C}_{obs}$ has n vertices, then this approach would take at least $O(n^2)$ time because intersection tests have to be made between each vertical ray and each segment. This even ignores the data structure issues involved in finding the cells that contain the query points and in building the roadmap that holds the connectivity information. By careful organization of the computation, it turns out that all of this can be nicely handled, and the resulting running time is only $O(n \lg n)$.

Plane-sweep principle The algorithm is based on the *plane-sweep* (or *line-sweep*) principle from computational geometry [129, 264, 302], which forms the basis of many combinatorial motion planning algorithms and many other algorithms in general. Much of computational geometry can be considered as the development of data structures and algorithms that generalize the sorting problem to multiple dimensions. In other words, the algorithms carefully "sort" geometric information.

The word "sweep" is used to refer to these algorithms because it can be imagined that a line (or plane, etc.) sweeps across the space, only to stop where some

[4]This is the reason why the approach is defined differently from Chapter 1 of [588]. In that case, sample points were not placed in the interiors of the 2-cells, and collision could result for some queries.

critical change occurs in the information. This gives the intuition, but the sweeping line is not explicitly represented by the algorithm. To construct the vertical decomposition, imagine that a vertical line sweeps from $x = -\infty$ to $x = \infty$, using (x, y) to denote a point in $\mathcal{C} = \mathbb{R}^2$.

From Section 6.2.1, note that the set P of $\mathcal{C}_{obs}$ vertices are the only data in $\mathbb{R}^2$ that appear in the problem input. It therefore seems reasonable that interesting things can only occur at these points. Sort the points in P in increasing order by their X coordinate. Assuming general position, no two points have the same X coordinate. The points in P will now be visited in order of increasing x value. Each visit to a point will be referred to as an *event*. Before, after, and in between every event, a list, L, of some $\mathcal{C}_{obs}$ edges will be maintained. This list must be maintained at all times in the order that the edges appear when stabbed by the vertical sweep line. The ordering is maintained from lower to higher.

Algorithm execution Figures 6.6 and 6.7 show how the algorithm proceeds. Initially, L is empty, and a doubly connected edge list is used to represent $\mathcal{C}_{free}$. Each connected component of $\mathcal{C}_{free}$ yields a single face in the data structure. Suppose inductively that after several events occur, L is correctly maintained. For each event, one of the four cases in Figure 6.2 occurs. By maintaining L in a balanced binary search tree [243], the edges above and below p can be determined in $O(\lg n)$ time. This is much better than $O(n)$ time, which would arise from checking every segment. Depending on which of the four cases from Figure 6.2 occurs, different updates to L are made. If the first case occurs, then two different edges are inserted, and the face of which p is on the border is split two times by vertical line segments. For each of the two vertical line segments, two half-edges are added, and all faces and half-edges must be updated correctly (this operation is local in that only records adjacent to where the change occurs need to be updated). The next two cases in Figure 6.2 are simpler; only a single face split is made. For the final case, no splitting occurs.

Once the face splitting operations have been performed, L needs to be updated. When the sweep line crosses p, two edges are always affected. For example, in the first and last cases of Figure 6.2, two edges must be inserted into L (the mirror images of these cases cause two edges to be deleted from L). If the middle two cases occur, then one edge is replaced by another in L. These insertion and deletion operations can be performed in $O(\lg n)$ time. Since there are n events, the running time for the construction algorithm is $O(n \lg n)$.

The roadmap $\mathcal{G}$ can be computed from the face pointers of the doubly connected edge list. A more elegant approach is to incrementally build $\mathcal{G}$ at each event. In fact, all of the pointer maintenance required to obtain a consistent doubly connected edge list can be ignored if desired, as long as $\mathcal{G}$ is correctly built and the sample point is obtained for each cell along the way. We can even go one step further, by forgetting about the cell decomposition and directly building a topological graph of line-segment paths between all sample points of adjacent cells.

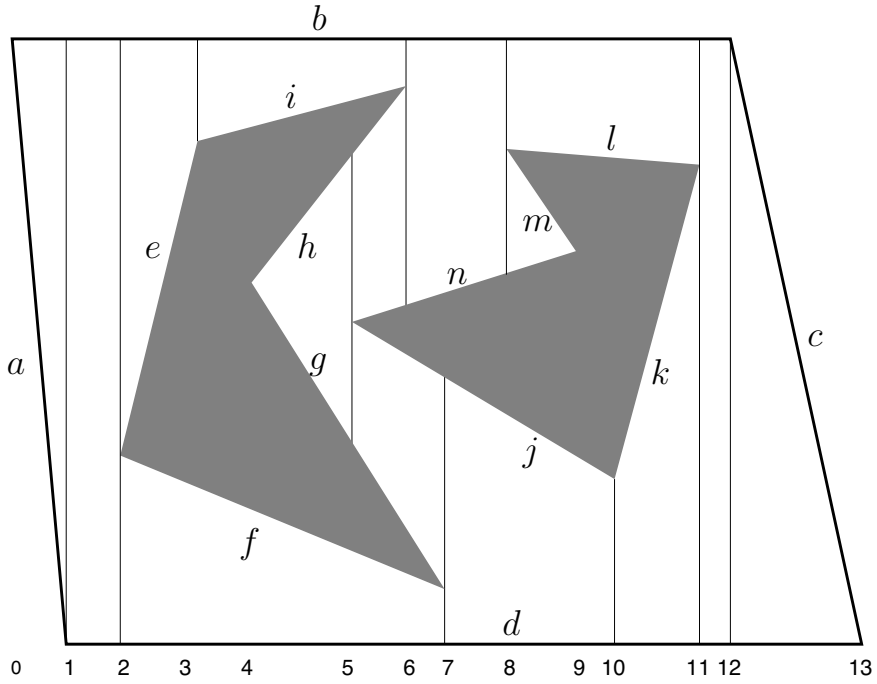


Figure 6.6: There are 14 events in this example.

Event	**Sorted Edges in L**	**Event**	**Sorted Edges in L**
0	$\{a, b\}$	7	$\{d, j, n, b\}$
1	$\{d, b\}$	8	$\{d, j, n, m, l, b\}$
2	$\{d, f, e, b\}$	9	$\{d, j, l, b\}$
3	$\{d, f, i, b\}$	10	$\{d, k, l, b\}$
4	$\{d, f, g, h, i, b\}$	11	$\{d, b\}$
5	$\{d, f, g, j, n, h, i, b\}$	12	$\{d, c\}$
6	$\{d, f, g, j, n, b\}$	13	$\{\}$

Figure 6.7: The status of L is shown after each of 14 events occurs. Before the first event, L is empty.

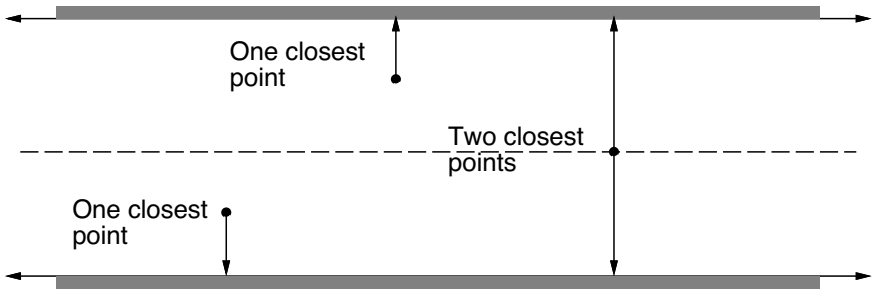


Figure 6.8: The maximum clearance roadmap keeps as far away from the $\mathcal{C}_{obs}$ as possible. This involves traveling along points that are equidistant from two or more points on the boundary of $\mathcal{C}_{obs}$.

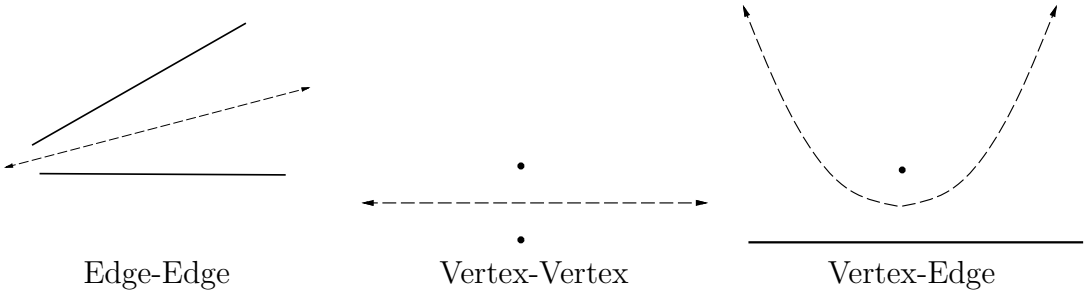


Figure 6.9: Voronoi roadmap pieces are generated in one of three possible cases. The third case leads to a quadratic curve.

6.2.3 Maximum-Clearance Roadmaps

A *maximum-clearance roadmap* tries to keep as far as possible from $\mathcal{C}_{obs}$, as shown for the corridor in Figure 6.8. The resulting solution paths are sometimes preferred in mobile robotics applications because it is difficult to measure and control the precise position of a mobile robot. Traveling along the maximum-clearance roadmap reduces the chances of collisions due to these uncertainties. Other names for this roadmap are *generalized Voronoi diagram* and *retraction method* [749]. It is considered as a generalization of the Voronoi diagram (recall from Section 5.2.2) from the case of points to the case of polygons. Each point along a roadmap edge is equidistant from two points on the boundary of $\mathcal{C}_{obs}$. Each roadmap vertex corresponds to the intersection of two or more roadmap edges and is therefore equidistant from three or more points along the boundary of $\mathcal{C}_{obs}$.

The retraction term comes from topology and provides a nice intuition about the method. A subspace S is a *deformation retract* of a topological space X if the following continuous homotopy, $h : X \times [0, 1] \rightarrow X$, can be defined as follows [451]:

1. $h(x, 0) = x$ for all $x \in X$.

2. $h(x, 1)$ is a continuous function that maps every element of X to some element of S.

3. For all $t \in [0,1]$, $h(s,t) = s$ for any $s \in S$.

The intuition is that $\mathcal{C}_{free}$ is gradually thinned through the homotopy process, until a skeleton, S, is obtained. An approximation to this shrinking process can be imagined by shaving off a thin layer around the whole boundary of $\mathcal{C}_{free}$. If this is repeated iteratively, the maximum-clearance roadmap is the only part that remains (assuming that the shaving always stops when thin "slivers" are obtained).

To construct the maximum-clearance roadmap, the concept of *features* from Section 5.3.3 is used again. Let the *feature set* refer to the set of all edges and vertices of $\mathcal{C}_{obs}$. Candidate paths for the roadmap are produced by every pair of features. This leads to a naive $O(n^4)$ time algorithm as follows. For every edge-edge feature pair, generate a line as shown in Figure 6.9a. For every vertex-vertex pair, generate a line as shown in Figure 6.9b. The maximum-clearance path between a point and a line is a parabola. Thus, for every edge-point pair, generate a parabolic curve as shown in Figure 6.9c. The portions of the paths that actually lie on the maximum-clearance roadmap are determined by intersecting the curves. Several algorithms exist that provide better asymptotic running times [616, 626], but they are considerably more difficult to implement. The best-known algorithm runs in $O(n \lg n)$ time in which n is the number of roadmap curves [865].

6.2.4 Shortest-Path Roadmaps

Instead of generating paths that maximize clearance, suppose that the goal is to find shortest paths. This leads to the *shortest-path roadmap*, which is also called the *reduced visibility graph* in [588]. The idea was first introduced in [742] and may perhaps be the first example of a motion planning algorithm. The shortest-path roadmap is in direct conflict with maximum clearance because shortest paths tend to graze the corners of $\mathcal{C}_{obs}$. In fact, the problem is ill posed because $\mathcal{C}_{free}$ is an open set. For any path $\tau : [0,1] \rightarrow \mathcal{C}_{free}$, it is always possible to find a shorter one. For this reason, we must consider the problem of determining shortest paths in $\mathrm{cl}(\mathcal{C}_{free})$, the closure of $\mathcal{C}_{free}$. This means that the robot is allowed to "touch" or "graze" the obstacles, but it is not allowed to penetrate them. To actually use the computed paths as solutions to a motion planning problem, they need to be slightly adjusted so that they come very close to $\mathcal{C}_{obs}$ but do not make contact. This slightly increases the path length, but the additional cost can be made arbitrarily small as the path gets arbitrarily close to $\mathcal{C}_{obs}$.

The *shortest-path roadmap*, $\mathcal{G}$, is constructed as follows. Let a *reflex vertex* be a polygon vertex for which the interior angle (in $\mathcal{C}_{free}$) is greater than π. All vertices of a convex polygon (assuming that no three consecutive vertices are collinear) are reflex vertices. The vertices of $\mathcal{G}$ are the reflex vertices. Edges of $\mathcal{G}$ are formed from two different sources:

Consecutive reflex vertices: If two reflex vertices are the endpoints of an edge of $\mathcal{C}_{obs}$, then an edge between them is made in $\mathcal{G}$.

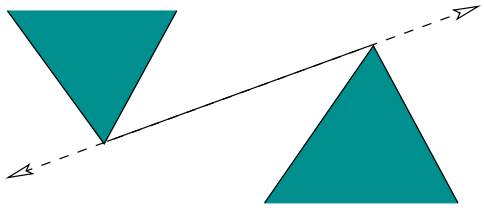


Figure 6.10: A bitangent edge must touch two reflex vertices that are mutually visible from each other, and the line must extend outward past each of them without poking into $\mathcal{C}_{obs}$.

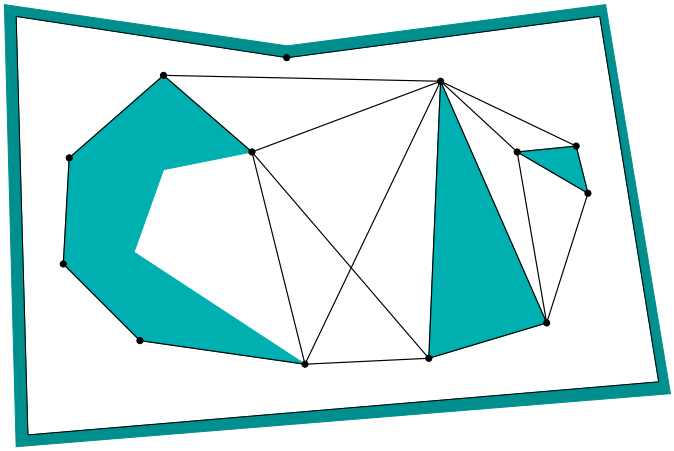


Figure 6.11: The shortest-path roadmap includes edges between consecutive reflex vertices on $\mathcal{C}_{obs}$ and also bitangent edges.

> **Bitangent edges:** If a *bitangent line* can be drawn through a pair of reflex vertices, then a corresponding edge is made in $\mathcal{G}$. A bitangent line, depicted in Figure 6.10, is a line that is incident to two reflex vertices and does not poke into the interior of $\mathcal{C}_{obs}$ at any of these vertices. Furthermore, these vertices must be mutually visible from each other.

An example of the resulting roadmap is shown in Figure 6.11. Note that the roadmap may have isolated vertices, such as the one at the top of the figure. To solve a query, q_I and q_G are connected to all roadmap vertices that are visible; this is shown in Figure 6.12. This makes an extended roadmap that is searched for a solution. If Dijkstra's algorithm is used, and if each edge is given a cost that corresponds to its path length, then the resulting solution path is the shortest path between q_I and q_G. The shortest path for the example in Figure 6.12 is shown in Figure 6.13.

If the bitangent tests are performed naively, then the resulting algorithm requires $O(n^3)$ time, in which n is the number of vertices of $\mathcal{C}_{obs}$. There are $O(n^2)$ pairs of reflex vertices that need to be checked, and each check requires $O(n)$ time to make certain that no other edges prevent their mutual visibility. The plane-sweep principle from Section 6.2.2 can be adapted to obtain a better algorithm,

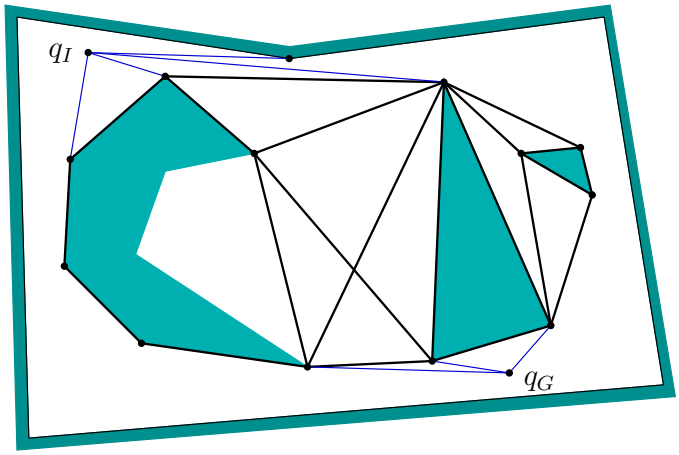


Figure 6.12: To solve a query, q_I and q_G are connected to all visible roadmap vertices, and graph search is performed.

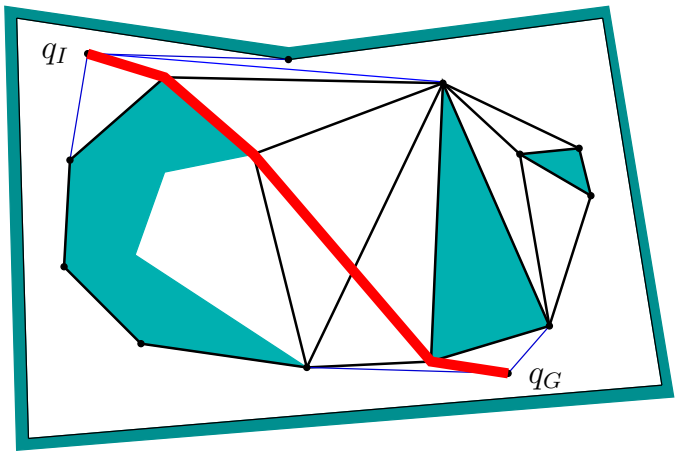


Figure 6.13: The shortest path in the extended roadmap is the shortest path between q_I and q_G.

which takes only $O(n^2 \lg n)$ time. The idea is to perform a *radial sweep* from each reflex vertex, v. A ray is started at $\theta = 0$, and events occur when the ray touches vertices. A set of bitangents through v can be computed in this way in $O(n \lg n)$ time. Since there are $O(n)$ reflex vertices, the total running time is $O(n^2 \lg n)$. See Chapter 15 of [264] for more details. There exists an algorithm that can compute the shortest-path roadmap in time $O(n \lg n + m)$, in which m is the total number of edges in the roadmap [384]. If the obstacle region is described by a simple polygon, the time complexity can be reduced to $O(n)$; see [709] for many shortest-path variations and references.

To improve numerical robustness, the shortest-path roadmap can be implemented without the use of trigonometric functions. For a sequence of three points, p_1, p_2, p_3, define the *left-turn predicate*, $f_l : \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \{\text{TRUE}, \text{FALSE}\}$, as

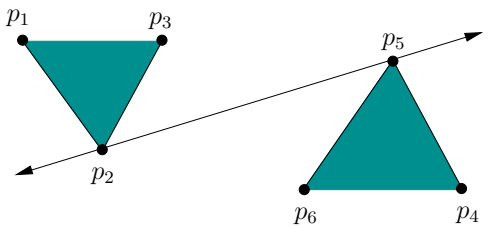


Figure 6.14: Potential bitangents can be identified by checking for left turns, which avoids the use of trigonometric functions and their associated numerical problems.

$f_l(p_1, p_2, p_3) =$ TRUE if and only if p_3 is to the left of the ray that starts at p_1 and pierces p_2. A point p_2 is a reflex vertex if and only if $f_l(p_1, p_2, p_3) =$ TRUE, in which p_1 and p_3 are the points before and after, respectively, along the boundary of $\mathcal{C}_{obs}$. The bitangent test can be performed by assigning points as shown in Figure 6.14. Assume that no three points are collinear and the segment that connects p_2 and p_5 is not in collision. The pair, p_2, p_5, of vertices should receive a bitangent edge if the following sentence is FALSE:

$$\big(f_l(p_1, p_2, p_5) \oplus f_l(p_3, p_2, p_5)\big) \vee \big(f_l(p_4, p_5, p_2) \oplus f_l(p_6, p_5, p_2)\big), \tag{6.1}$$

in which $\oplus$ denotes logical "exclusive or." The f_l predicate can be implemented without trigonometric functions by defining

$$M(p_1, p_2, p_3) = \begin{pmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{pmatrix}, \tag{6.2}$$

in which $p_i = (x_i, y_i)$. If $\det(M) > 0$, then $f_l(p_1, p_2, p_3) =$ TRUE; otherwise, $f_l(p_1, p_2, p_3) =$ FALSE.

6.3 Cell Decompositions

Section 6.2.2 introduced the vertical cell decomposition to solve the motion planning problem when $\mathcal{C}_{obs}$ is polygonal. It is important to understand, however, that this is just one choice among many for the decomposition. Some of these choices may not be preferable in 2D; however, they might generalize better to higher dimensions. Therefore, other cell decompositions are covered in this section, to provide a smoother transition from vertical cell decomposition to cylindrical algebraic decomposition in Section 6.4, which solves the motion planning problem in any dimension for any semi-algebraic model. Along the way, a cylindrical decomposition will appear in Section 6.3.4 for the special case of a line-segment robot in $\mathcal{W} = \mathbb{R}^2$.

6.3.1 General Definitions

In this section, the term *complex* refers to a collection of cells together with their boundaries. A partition into cells can be derived from a complex, but the complex contains additional information that describes how the cells must fit together. The term *cell decomposition* still refers to the partition of the space into cells, which is derived from a *complex*.

It is tempting to define complexes and cell decompositions in a very general manner. Imagine that any partition of $\mathcal{C}_{free}$ could be called a cell decomposition. A cell could be so complicated that the notion would be useless. Even $\mathcal{C}_{free}$ itself could be declared as one big cell. It is more useful to build decompositions out of simpler cells, such as ones that contain no holes. Formally, this requires that every k-dimensional cell is homeomorphic to $B^k \subset \mathbb{R}^k$, an open k-dimensional unit ball. From a motion planning perspective, this still yields cells that are quite complicated, and it will be up to the particular cell decomposition method to enforce further constraints to yield a complete planning algorithm.

Two different complexes will be introduced. The *simplicial complex* is explained because it is one of the easiest to understand. Although it is useful in many applications, it is not powerful enough to represent all of the complexes that arise in motion planning. Therefore, the *singular complex* is also introduced. Although it is more complicated to define, it encompasses all of the cell complexes that are of interest in this book. It also provides an elegant way to represent topological spaces. Another important cell complex, which is not covered here, is the *CW-complex* [439].

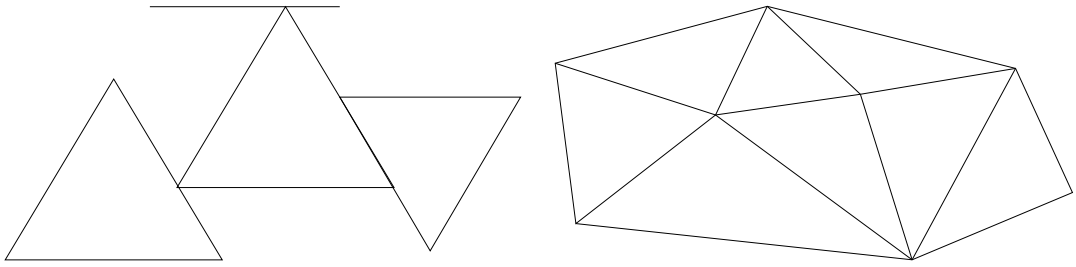
Simplicial Complex For this definition, it is assumed that $X = \mathbb{R}^n$. Let p_1, p_2, …, p_{k+1}, be $k+1$ linearly independent[5] points in $\mathbb{R}^n$. A k-simplex, $[p_1, \ldots, p_{k+1}]$, is formed from these points as

$$[p_1, \ldots, p_{k+1}] = \left\{ \sum_{i=1}^{k+1} \alpha_i p_i \in \mathbb{R}^n \;\middle|\; \alpha_i \geq 0 \text{ for all } i \text{ and } \sum_{i=1}^{k+1} \alpha_i = 1 \right\}, \tag{6.3}$$

in which $\alpha_i p_i$ is the scalar multiplication of α_i by each of the point coordinates. Another way to view (6.3) is as the convex hull of the $k+1$ points (i.e., all ways to linearly interpolate between them). If $k = 2$, a triangular region is obtained. For $k = 3$, a tetrahedron is produced.

For any k-simplex and any i such that $1 \leq i \leq k+1$, let $\alpha_i = 0$. This yields a $(k-1)$-dimensional simplex that is called a *face* of the original simplex. A 2-simplex has three faces, each of which is a 1-simplex that may be called an edge. Each 1-simplex (or edge) has two faces, which are 0-simplexes called *vertices*.

[5] Form k vectors by subtracting p_1 from the other k points for some positive integer k such that $k \leq n$. Arrange the vectors into a $k \times n$ matrix. For linear independence, there must be at least one $k \times k$ cofactor with a nonzero determinant. For example, if $k = 2$, then the three points cannot be collinear.



Not a simplicial complex A simplicial complex

Figure 6.15: To become a simplicial complex, the simplex faces must fit together nicely.

To form a complex, the simplexes must fit together in a nice way. This yields a high-dimensional notion of a *triangulation*, which in $\mathbb{R}^2$ is a tiling composed of triangular regions. A *simplicial complex*, $\mathcal{K}$, is a finite set of simplexes that satisfies the following:

1. Any face of a simplex in $\mathcal{K}$ is also in $\mathcal{K}$.

2. The intersection of any two simplexes in $\mathcal{K}$ is either a common face of both of them or the intersection is empty.

Figure 6.15 illustrates these requirements. For $k > 0$, a k-cell of $\mathcal{K}$ is defined to be interior, $\text{int}([p_1, \ldots, p_{k+1}])$, of any k-simplex. For $k = 0$, every 0-simplex is a 0-cell. The union of all of the cells forms a partition of the point set covered by $\mathcal{K}$. This therefore provides a *cell decomposition* in a sense that is consistent with Section 6.2.2.

Singular complex Simplicial complexes are useful in applications such as geometric modeling and computer graphics for computing the topology of models. Due to the complicated topological spaces, implicit, nonlinear models, and decomposition algorithms that arise in motion planning, they are insufficient for the most general problems. A *singular complex* is a generalization of the *simplicial complex*. Instead of being limited to $\mathbb{R}^n$, a singular complex can be defined on any manifold, X (it can even be defined on any Hausdorff topological space). The main difference is that, for a simplicial complex, each simplex is a subset of $\mathbb{R}^n$; however, for a singular complex, each *singular simplex* is actually a homeomorphism from a (simplicial) simplex in $\mathbb{R}^n$ to a subset of X.

To help understand the idea, first consider a 1D singular complex, which happens to be a topological graph (as introduced in Example 4.6). The interval $[0, 1]$ is a 1-simplex, and a continuous path $\tau : [0, 1] \rightarrow X$ is a *singular* 1*-simplex* because it is a homeomorphism of $[0, 1]$ to the image of τ in X. Suppose $\mathcal{G}(V, E)$ is a topological graph. The cells are subsets of X that are defined as follows. Each point $v \in V$ is a 0-cell in X. To follow the formalism, each is considered as the

image of a function $f : \{0\} \rightarrow X$, which makes it a *singular* 0*-simplex*, because $\{0\}$ is a 0-simplex. For each path $\tau \in E$, the corresponding 1-cell is

$$\{x \in X \mid \tau(s) = x \text{ for some } s \in (0,1)\}. \tag{6.4}$$

Expressed differently, it is $\tau((0,1))$, the image of the path τ, except that the endpoints are removed because they are already covered by the 0-cells (the cells must form a partition).

These principles will now be generalized to higher dimensions. Since all balls and simplexes of the same dimension are homeomorphic, balls can be used instead of a simplex in the definition of a singular simplex. Let $B^k \subset \mathbb{R}^k$ denote a closed, k-dimensional unit ball,

$$D^k = \{x \in \mathbb{R}^n \mid \|x\| \leq 1\}, \tag{6.5}$$

in which $\|\cdot\|$ is the Euclidean norm. A *singular* k*-simplex* is a continuous mapping $\sigma : D^k \rightarrow X$. Let $\text{int}(D^k)$ refer to the interior of D^k. For $k \geq 1$, the k*-cell*, C, corresponding to a singular k-simplex, σ, is the image $C = \sigma(\text{int}(D^k)) \subseteq X$. The 0-cells are obtained directly as the images of the 0 singular simplexes. Each singular 0-simplex maps to the 0-cell in X. If σ is restricted to $\text{int}(D^k)$, then it actually defines a homeomorphism between D^k and C. Note that both of these are open sets if $k > 0$.

A simplicial complex requires that the simplexes fit together nicely. The same concept is applied here, but topological concepts are used instead because they are more general. Let $\mathcal{K}$ be a set of singular simplexes of varying dimensions. Let S_k denote the union of the images of all singular i-simplexes for all $i \leq k$.

A collection of singular simplexes that map into a topological space X is called a *singular complex* if:

1. For each dimension k, the set $S_k \subseteq X$ must be closed. This means that the cells must all fit together nicely.

2. Each k-cell is an open set in the topological subspace S_k. Note that 0-cells are open in S_0, even though they are usually closed in X.

Example 6.1 (Vertical Decomposition) The vertical decomposition of Section 6.2.2 is a nice example of a singular complex that is not a simplicial complex because it contains trapezoids. The interior of each trapezoid and triangle forms a 2-cell, which is an open set. For every pair of adjacent 2-cells, there is a 1-cell on their common boundary. There are no 0-cells because the vertices lie in $\mathcal{C}_{obs}$, not in $\mathcal{C}_{free}$. The subspace S_2 is formed by taking the union of all 2-cells and 1-cells to yield $S_2 = \mathcal{C}_{free}$. This satisfies the closure requirement because the complex is built in $\mathcal{C}_{free}$ only; hence, the topological space is $\mathcal{C}_{free}$. The set $S_2 = \mathcal{C}_{free}$ is both open and closed. The set S_1 is the union of all 1-cells. This is also closed because the 1-cell endpoints all lie in $\mathcal{C}_{obs}$. Each 1-cell is also an open set.

One way to avoid some of these strange conclusions from the topology restricted to $\mathcal{C}_{free}$ is to build the vertical decomposition in $\text{cl}(\mathcal{C}_{free})$, the closure of $\mathcal{C}_{free}$. This

can be obtained by starting with the previously defined vertical decomposition and adding a new 1-cell for every edge of $\mathcal{C}_{obs}$ and a 0-cell for every vertex of $\mathcal{C}_{obs}$. Now $S_3 = \text{cl}(\mathcal{C}_{free})$, which is closed in $\mathbb{R}^2$. Likewise, S_2, S_1, and S_0, are closed in the usual way. Each of the individual k-dimensional cells, however, is open in the topological space S_k. The only strange case is that the 0-cells are considered open, but this is true in the discrete topological space S_0. ■

6.3.2 2D Decompositions

The vertical decomposition method of Section 6.2.2 is just one choice of many cell decomposition methods for solving the problem when $\mathcal{C}_{obs}$ is polygonal. It provides a nice balance between the number of cells, computational efficiency, and implementation ease. It is usually possible to decompose $\mathcal{C}_{obs}$ into far fewer convex cells. This would be preferable for multiple-query applications because the roadmap would be smaller. It is unfortunately quite difficult to optimize the number of cells. Determining the decomposition of a polygonal $\mathcal{C}_{obs}$ with holes that uses the smallest number of convex cells is NP-hard [519, 645]. Therefore, we are willing to tolerate nonoptimal decompositions.

Triangulation One alternative to the vertical decomposition is to perform a *triangulation*, which yields a simplicial complex over $\mathcal{C}_{free}$. Figure 6.16 shows an example. Since $\mathcal{C}_{free}$ is an open set, there are no 0-cells. Each 2-simplex (triangle) has either one, two, or three faces, depending on how much of its boundary is shared with $\mathcal{C}_{obs}$. A roadmap can be made by connecting the samples for 1-cells and 2-cells as shown in Figure 6.17. Note that there are many ways to triangulate $\mathcal{C}_{free}$ for a given problem. Finding good triangulations, which for example means trying to avoid thin triangles, is given considerable attention in computational geometry [129, 264, 302].

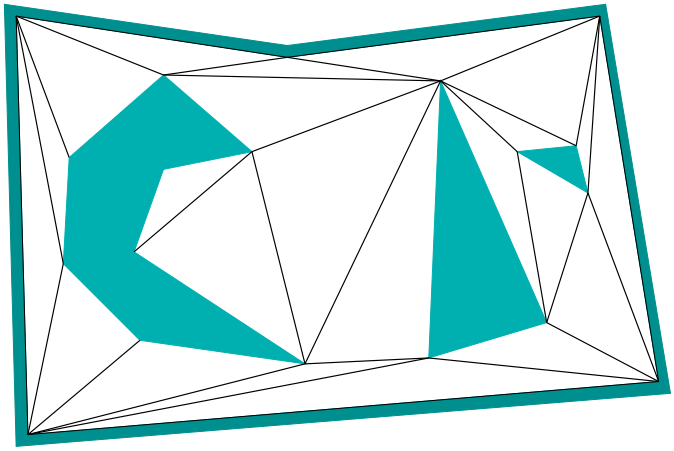


Figure 6.16: A triangulation of $\mathcal{C}_{free}$.

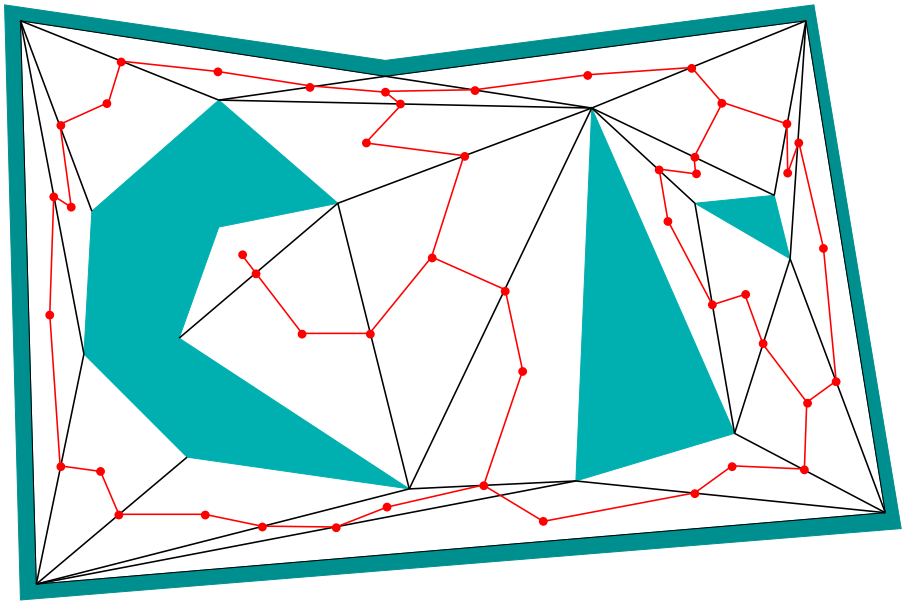


Figure 6.17: A roadmap obtained from the triangulation.

How can the triangulation be computed? It might seem tempting to run the vertical decomposition algorithm of Section 6.2.2 and split each trapezoid into two triangles. Even though this leads to triangular cells, it does not produce a simplicial complex (two triangles could abut the same side of a triangle edge). A naive approach is to incrementally split faces by attempting to connect two vertices of a face by a line segment. If this segment does not intersect other segments, then the split can be made. This process can be iteratively performed over all vertices of faces that have more than three vertices, until a triangulation is eventually obtained. Unfortunately, this results in an $O(n^3)$ time algorithm because $O(n^2)$ pairs must be checked in the worst case, and each check requires $O(n)$ time to determine whether an intersection occurs with other segments. This can be easily reduced to $O(n^2 \lg n)$ by performing radial sweeping. Chapter 3 of [264] presents an algorithm that runs in $O(n \lg n)$ time by first partitioning $\mathcal{C}_{free}$ into *monotone polygons*, and then efficiently triangulating each monotone polygon. If $\mathcal{C}_{free}$ is simply connected, then, surprisingly, a triangulation can be computed in linear time [190]. Unfortunately, this algorithm is too complicated to use in practice (there are, however, simpler algorithms for which the complexity is close to $O(n)$; see [90] and the end of Chapter 3 of [264] for surveys).

Cylindrical decomposition The *cylindrical decomposition* is very similar to the vertical decomposition, except that when any of the cases in Figure 6.2 occurs, then a vertical line slices through all faces, all the way from $y = -\infty$ to $y = \infty$. The result is shown in Figure 6.18, which may be considered as a singular complex. This may appear very inefficient in comparison to the vertical decomposition; however, it is presented here because it generalizes nicely to any dimension, any C-space topology, and any semi-algebraic model. Therefore, it is presented here to ease

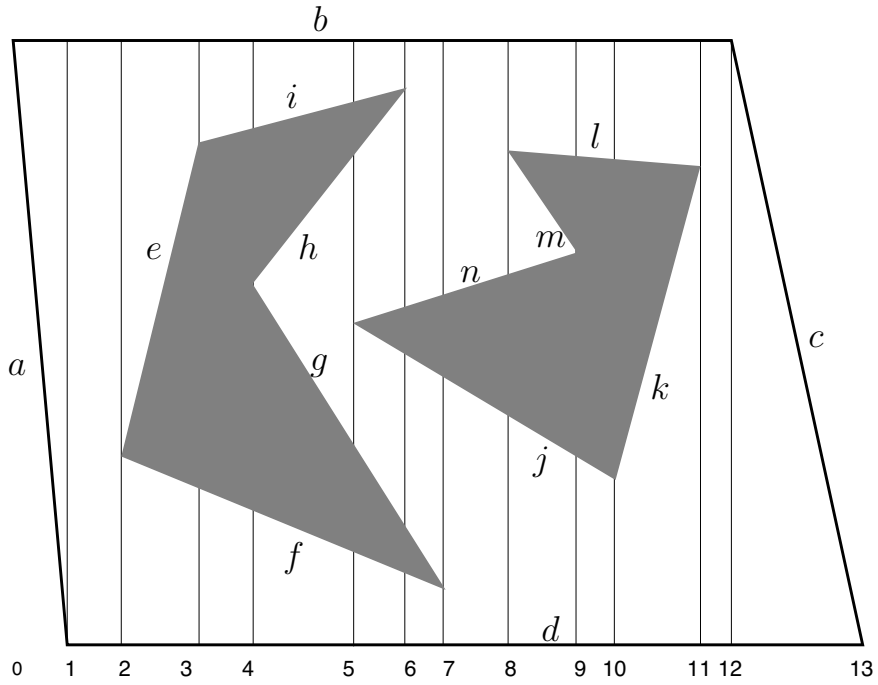


Figure 6.18: The cylindrical decomposition differs from the vertical decomposition in that the rays continue forever instead of stopping at the nearest edge. Compare this figure to Figure 6.6.

the transition to more general decompositions. The most important property of the cylindrical decomposition is shown in Figure 6.19. Consider each vertical strip between two events. When traversing a strip from $y = -\infty$ to $y = \infty$, the points alternate between being $\mathcal{C}_{obs}$ and $\mathcal{C}_{free}$. For example, between events 4 and 5, the points below edge f are in $\mathcal{C}_{free}$. Points between f and g lie in $\mathcal{C}_{obs}$. Points between g and h lie in $\mathcal{C}_{free}$, and so forth. The cell decomposition can be defined so that 2D cells are also created in $\mathcal{C}_{obs}$. Let $S(x, y)$ denote the logical predicate (3.6) from Section 3.1.1. When traversing a strip, the value of $S(x, y)$ also alternates. This behavior is the main reason to construct a cylindrical decomposition, which will become clearer in Section 6.4.2. Each vertical strip is actually considered to be a *cylinder*, hence, the name cylindrical decomposition (i.e., there are not necessarily any cylinders in the 3D geometric sense).

6.3.3 3D Vertical Decomposition

It turns out that the vertical decomposition method of Section 6.2.2 can be extended to any dimension by recursively applying the sweeping idea. The method requires, however, that $\mathcal{C}_{obs}$ is piecewise linear. In other words, $\mathcal{C}_{obs}$ is represented as a semi-algebraic model for which all primitives are linear. Unfortunately, most of the general motion planning problems involve nonlinear algebraic primitives because of the nonlinear transformations that arise from rotations. Recall the

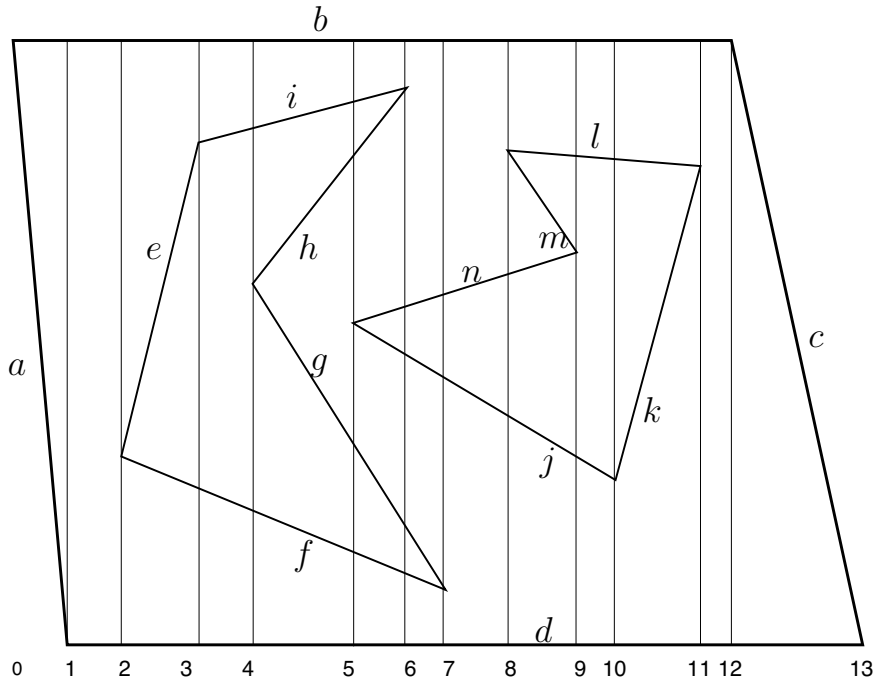


Figure 6.19: The cylindrical decomposition produces vertical strips. Inside of a strip, there is a stack of collision-free cells, separated by $\mathcal{C}_{obs}$.

complicated algebraic $\mathcal{C}_{obs}$ model constructed in Section 4.3.3. To handle generic algebraic models, powerful techniques from computational algebraic geometry are needed. This will be covered in Section 6.4.

One problem for which $\mathcal{C}_{obs}$ is piecewise linear is a polyhedral robot that can translate in $\mathbb{R}^3$, and the obstacles in $\mathcal{W}$ are polyhedra. Since the transformation equations are linear in this case, $\mathcal{C}_{obs} \subset \mathbb{R}^3$ is polyhedral. The polygonal faces of $\mathcal{C}_{obs}$ are obtained by forming geometric primitives for each of the Type FV, Type VF, and Type EE cases of contact between $\mathcal{A}$ and $\mathcal{O}$, as mentioned in Section 4.3.2.

Figure 6.20 illustrates the algorithm that constructs the 3D vertical decomposition. Compare this to the algorithm in Section 6.2.2. Let (x, y, z) denote a point in $\mathcal{C} = \mathbb{R}^3$. The vertical decomposition yields convex 3-cells, 2-cells, and 1-cells. Neglecting degeneracies, a generic 3-cell is bounded by six planes. The cross section of a 3-cell for some fixed x value yields a trapezoid or triangle, exactly as in the 2D case, but in a plane parallel to the yz plane. Two sides of a generic 3-cell are parallel to the yz plane, and two other sides are parallel to the xz plane. The 3-cell is bounded above and below by two polygonal faces of $\mathcal{C}_{obs}$.

Initially, sort the $\mathcal{C}_{obs}$ vertices by their x coordinate to obtain the events. Now consider sweeping a plane perpendicular to the x-axis. The plane for a fixed value of x produces a 2D polygonal slice of $\mathcal{C}_{obs}$. Three such slices are shown at the bottom of Figure 6.20. Each slice is parallel to the yz plane and appears to look exactly like a problem that can be solved by the 2D vertical decomposition method.

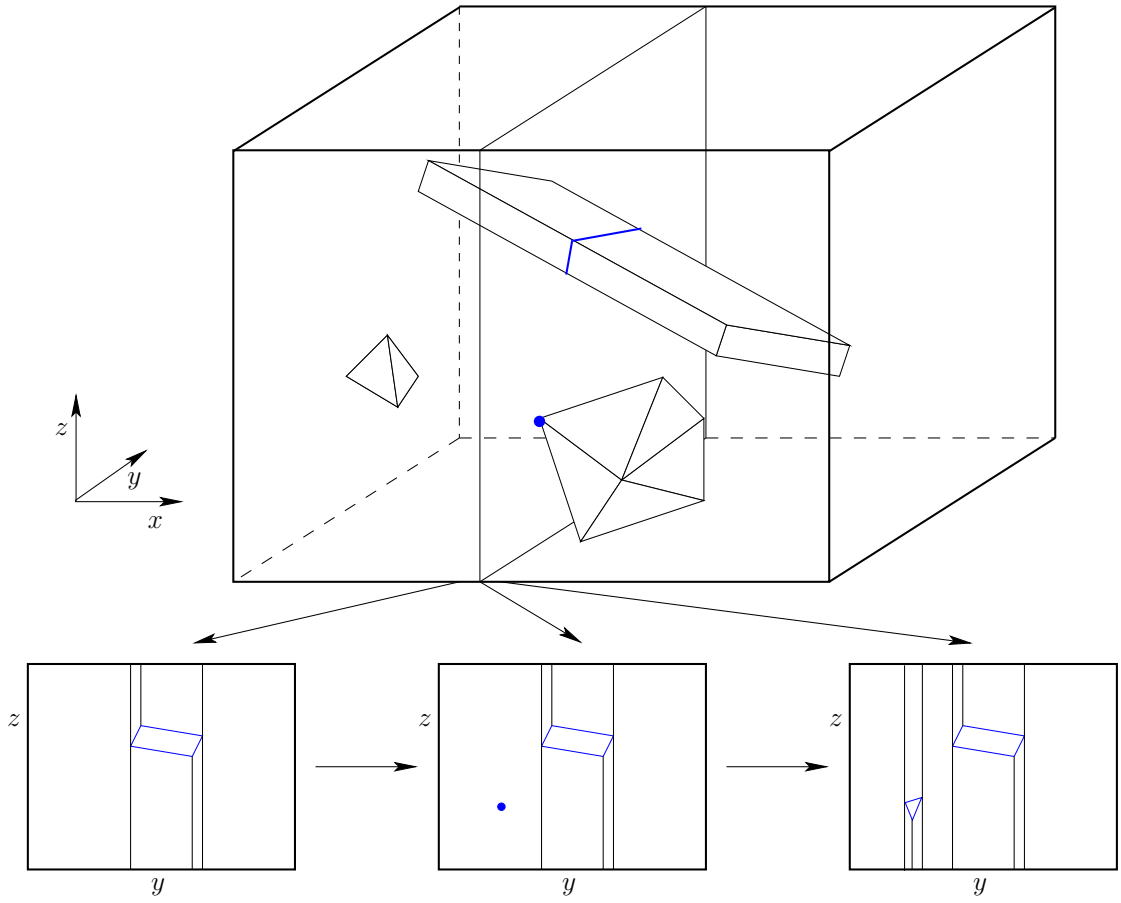


Figure 6.20: In higher dimensions, the sweeping idea can be applied recursively.

The 2-cells in a slice are actually slices of 3-cells in the 3D decomposition. The only places in which these 3-cells can critically change is when the sweeping plane stops at some x value. The center slice in Figure 6.20 corresponds to the case in which a vertex of a convex polyhedron is encountered, and all of the polyhedron lies to right of the sweep plane (i.e., the rest of the polyhedron has not been encountered yet). This corresponds to a place where a critical change must occur in the slices. These are 3D versions of the cases in Figure 6.2, which indicate how the vertical decomposition needs to be updated. The algorithm proceeds by first building the 2D vertical decomposition at the first x event. At each event, the 2D vertical decomposition must be updated to take into account the critical changes. During this process, the 3D cell decomposition and roadmap can be incrementally constructed, as in the 2D case.

The roadmap is constructed by placing a sample point in the center of each 3-cell and 2-cell. The vertices are the sample points, and edges are added to the roadmap by connecting the sample points for each case in which a 3-cell is adjacent to a 2-cell.

This same principle can be extended to any dimension, but the applications to

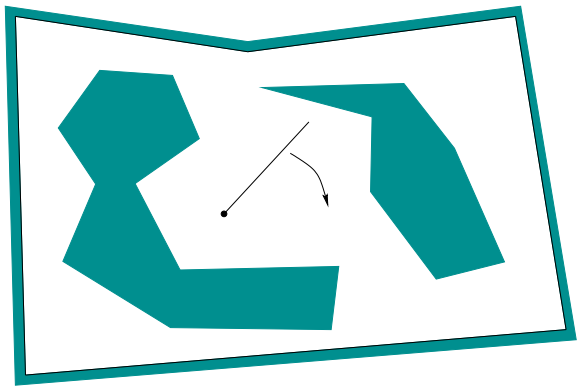


Figure 6.21: Motion planning for a line segment that can translate and rotate in a 2D world.

motion planning are limited because the method requires linear models (or at least it is very challenging to adapt to nonlinear models; in some special cases, this can be done). See [426] for a summary of the complexity of vertical decompositions for various geometric primitives and dimensions.

6.3.4 A Decomposition for a Line-Segment Robot

This section presents one of the simplest cell decompositions that involves nonlinear models, yet it is already fairly complicated. This will help to give an appreciation of the difficulty of combinatorial planning in general. Consider the planning problem shown in Figure 6.21. The robot, $\mathcal{A}$, is a single line segment that can translate or rotate in $\mathcal{W} = \mathbb{R}^2$. The dot on one end of $\mathcal{A}$ is used to illustrate its origin and is not part of the model. The C-space, $\mathcal{C}$, is homeomorphic to $\mathbb{R}^2 \times \mathbb{S}^1$. Assume that the parameterization $\mathbb{R}^2 \times [0, 2\pi]/\sim$ is used in which the identification equates $\theta = 0$ and $\theta = 2\pi$. A point in $\mathcal{C}$ is represented as (x, y, θ).

An approximate solution First consider making a cell decomposition for the case in which the segment can only translate. The method from Section 4.3.2 can be used to compute $\mathcal{C}_{obs}$ by treating the robot-obstacle interaction with Type EV and Type VE contacts. When the interior of $\mathcal{A}$ touches an obstacle vertex, then Type EV is obtained. An endpoint of $\mathcal{A}$ touching an object interior yields Type VE. Each case produces an edge of $\mathcal{C}_{obs}$, which is polygonal. Once this is represented, the vertical decomposition can be used to solve the problem. This inspires a reasonable numerical approach to the rotational case, which is to discretize θ into K values, $i\Delta\theta$, for $0 \leq i \leq K$, and $\Delta\theta = 2\pi/K$ [20]. The obstacle region, $\mathcal{C}_{obs}$, is polygonal for each case, and we can imagine having a stack of K polygonal regions. A roadmap can be formed by connecting sampling points inside of a slice in the usual way, and also by connecting samples between corresponding cells in neighboring slices. If K is large enough, this strategy works well, but the method is not complete because a sufficient value for K cannot be determined in

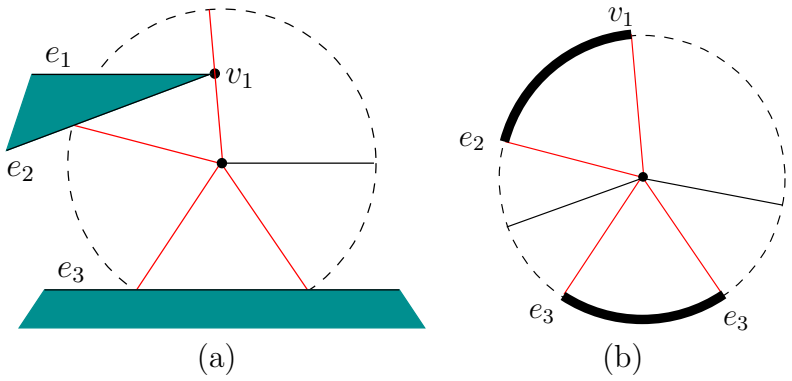


Figure 6.22: Fix (x, y) and swing the segment around for all values of $\theta \in [0, 2\pi]/\sim$. (a) Note the vertex and edge features that are hit by the segment. (b) Record orientation intervals over which the robot is not in collision.

advance. The method is actually an interesting hybrid between combinatorial and sampling-based motion planning. A resolution-complete version can be imagined.

In the limiting case, as K tends to infinity, the surfaces of $\mathcal{C}_{obs}$ become curved along the θ direction. The conditions in Section 4.3.3 must be applied to generate the actual obstacle regions. This is possible, but it yields a semi-algebraic representation of $\mathcal{C}_{obs}$ in terms of implicit polynomial primitives. It is no easy task to determine an explicit representation in terms of simple cells that can be used for motion planning. The method of Section 6.3.3 cannot be used because $\mathcal{C}_{obs}$ is not polyhedral. Therefore, special analysis is warranted to produce a cell decomposition.

The general idea is to construct a cell decomposition in $\mathbb{R}^2$ by considering only the translation part, (x, y). Each cell in $\mathbb{R}^2$ is then lifted into $\mathcal{C}$ by considering θ as a third axis that is "above" the xy plane. A cylindrical decomposition results in which each cell in the xy plane produces a cylindrical stack of cells for different θ values. Recall the cylinders in Figures 6.18 and 6.19. The vertical axis corresponds to θ in the current setting, and the horizontal axis is replaced by two axes, x and y.

To construct the decomposition in $\mathbb{R}^2$, consider the various robot-obstacle contacts shown in Figure 6.22. In Figure 6.22a, the segment swings around from a fixed (x, y). Two different kinds of contacts arise. For some orientation (value of θ), the segment contacts v_1, forming a Type EV contact. For three other orientations, the segment contacts an edge, forming Type VE contacts. Once again using the *feature* concept, there are four orientations at which the segment contacts a feature. Each feature may be either a vertex or an edge. Between the two contacts with e_2 and e_3, the robot is not in collision. These configurations lie in $\mathcal{C}_{free}$. Also, configurations for which the robot is between contacts e_3 (the rightmost contact) and v_1 are also in $\mathcal{C}_{free}$. All other orientations produce configurations in $\mathcal{C}_{obs}$. Note that the line segment cannot get from being between e_2 and e_3 to being between

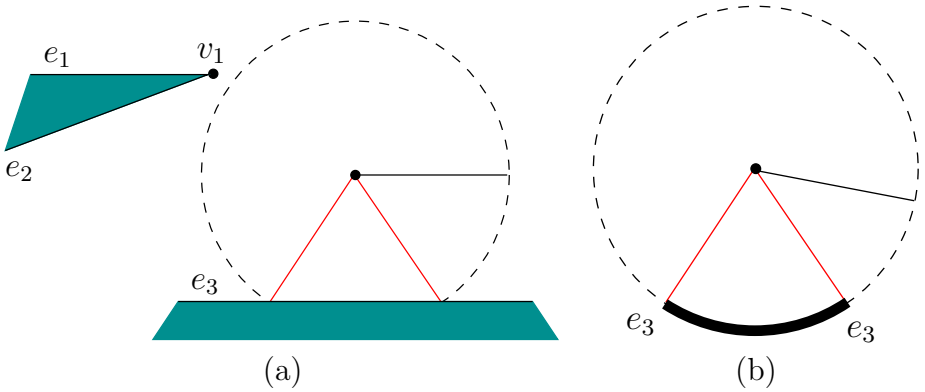


Figure 6.23: If x is increased enough, a critical change occurs in the radar map because v_1 can no longer be reached by the robot.

e_3 and v_1, unless the (x, y) position is changed. It therefore seems sensible that these must correspond to different cells in whatever decomposition is made.

Radar maps Figure 6.22b illustrates which values of θ produce collision. We will refer to this representation as a *radar map*. The four contact orientations are indicated by the contact feature. The notation $[e_3, v_1]$ and $[e_2, e_3]$ identifies the two intervals for which $(x, y, \theta) \in \mathcal{C}_{free}$. Now imagine changing (x, y) by a small amount, to obtain (x', y'). How would the radar map change? The precise angles at which the contacts occur would change, but the notation $[e_3, v_1]$ and $[e_2, e_3]$, for configurations that lie in $\mathcal{C}_{free}$, remains unchanged. Even though the angles change, there is no interesting change in terms of the contacts; therefore, it makes sense to declare (x, y, θ) and (x, y, θ') to lie in the same cell in $\mathcal{C}_{free}$ because θ and θ' both place the segment between the same contacts. Imagine a column of two 3-cells above a small area around (x, y). One 3-cell is for orientations in $[e_3, v_1]$, and the other is for orientations in $[e_2, e_3]$. These appear to be 3D regions in $\mathcal{C}_{free}$ because each of x, y, and θ can be perturbed a small amount without leaving the cell.

Of course, if (x, y) is changed enough, then eventually we expect a dramatic change to occur in the radar map. For example, imagine e_3 is infinitely long, and the x value is gradually increased in Figure 6.22a. The black band between v_1 and e_2 in Figure 6.22b shrinks in length. Eventually, when the distance from (x', y') to v_1 is greater than the length of $\mathcal{A}$, the black band disappears. This situation is shown in Figure 6.23. The change is very important to notice because after that region vanishes, any orientation θ' between e_3 and e_3, traveling the long way around the circle, produces a configuration $(x', y', \theta') \in \mathcal{C}_{free}$. This seems very important because it tells us that we can travel between the original two cells by moving the robot further way from v_1, rotating the robot, and then moving back. Now move from the position shown in Figure 6.23 into the positive y direction. The remaining black band begins to shrink and finally disappears when the distance

to e_3 is further than the robot length. This represents another critical change.

The radar map can be characterized by specifying a circular ordering

$$([f_1, f_2], [f_3, f_4], [f_5, f_6], \ldots, [f_{2k-1}, f_{2k}]), \tag{6.6}$$

when there are k orientation intervals over which the configurations lie in $\mathcal{C}_{free}$. For the radar map in Figure 6.22b, this representation yields $([e_3, v_1], [e_2, e_3])$. Each f_i is a feature, which may be an edge or a vertex. Some of the f_i may be identical; the representation for Figure 6.23b is $([e_3, e_3])$. The intervals are specified in counterclockwise order around the radar map. Since the ordering is circular, it does not matter which interval is specified first. There are two degenerate cases. If $(x, y, \theta) \in \mathcal{C}_{free}$ for all $\theta \in [0, 2\pi)$, then we write () for the ordering. On the other hand, if $(x, y, \theta) \in \mathcal{C}_{obs}$ for all $\theta \in [0, 2\pi)$, then we write $\emptyset$.

Critical changes in cells Now we are prepared to explain the cell decomposition in more detail. Imagine traveling along a path in $\mathbb{R}^2$ and producing an animated version of the radar map in Figure 6.22b. We say that a *critical change* occurs each time the circular ordering representation of (6.6) changes. Changes occur when intervals: 1) appear, 2) disappear, 3) split apart, 4) merge into one, or 5) when the feature of an interval changes. The first task is to partition $\mathbb{R}^2$ into maximal 2-cells over which no critical changes occur. Each one of these 2-cells, R, represents the projection of a strip of 3-cells in $\mathcal{C}_{free}$. Each 3-cell is defined as follows. Let $\{R, [f_i, f_{i+1}]\}$ denote the 3D region in $\mathcal{C}_{free}$ for which $(x, y) \in R$ and θ places the segment between contacts f_i and f_{i+1}. The *cylinder* of cells above R is given by $\{R, [f_i, f_{i+1}]\}$ for each interval in the circular ordering representation, (6.6). If any orientation is possible because $\mathcal{A}$ never contacts an obstacle while in R, then we write $\{R\}$.

What are the positions in $\mathbb{R}^2$ that cause critical changes to occur? It turns out that there are five different cases to consider, each of which produces a set of *critical curves* in $\mathbb{R}^2$. When one of these curves is crossed, a critical change occurs. If none of these curves is crossed, then no critical change can occur. Therefore, these curves precisely define the boundaries of the desired 2-cells in $\mathbb{R}^2$. Let L denote the length of the robot (which is the line segment).

Consider how the five cases mentioned above may occur. Two of the five cases have already been observed in Figures 6.22 and 6.23. These appear in Figures 6.24a and Figures 6.24b, and occur if (x, y) is within L of an edge or a vertex. The third and fourth cases are shown in Figures 6.24c and 6.24d, respectively. The third case occurs because crossing the curve causes $\mathcal{A}$ to change between being able to touch e and being able to touch v. This must be extended from any edge at an endpoint that is a reflex vertex (interior angle is greater than π). The fourth case is actually a return of the bitangent case from Figure 6.10, which arose for the shortest path graph. If the vertices are within L of each other, then a linear critical curve is generated because $\mathcal{A}$ is no longer able to touch v_2 when crossing it from right to left. Bitangents always produce curves in pairs; the curve above v_2 is not shown. The final case, shown in Figure 6.25, is the most complicated. It is a

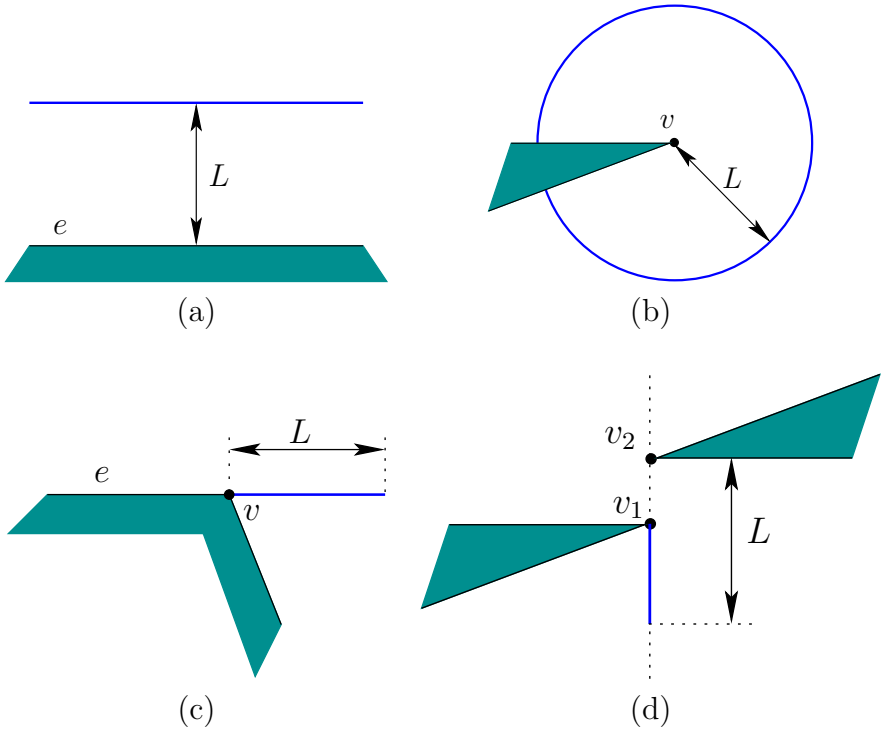


Figure 6.24: Four of the five cases that produce critical curves in $\mathbb{R}^2$.

fourth-degree algebraic curve called the Conchoid of Nicomedes, which arises from $\mathcal{A}$ being in simultaneous contact between v and e. Inside of the teardrop-shaped curve, $\mathcal{A}$ can contact e but not v. Just outside of the curve, it can touch v. If the xy coordinate frame is placed so that v is at $(0, 0)$, then the equation of the curve is

$$(x^2 - y^2)(y + d)^2 - y^2 L^2 = 0, \tag{6.7}$$

in which d is the distance from v to e.

Putting all of the curves together generates a cell decomposition of $\mathbb{R}^2$. There are *noncritical regions*, over which there is no change in (6.6); these form the 2-cells. The boundaries between adjacent 2-cells are sections of the critical curves

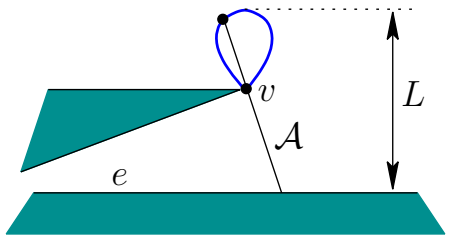


Figure 6.25: The fifth case is the most complicated. It results in a fourth-degree algebraic curve called the Conchoid of Nicomedes.

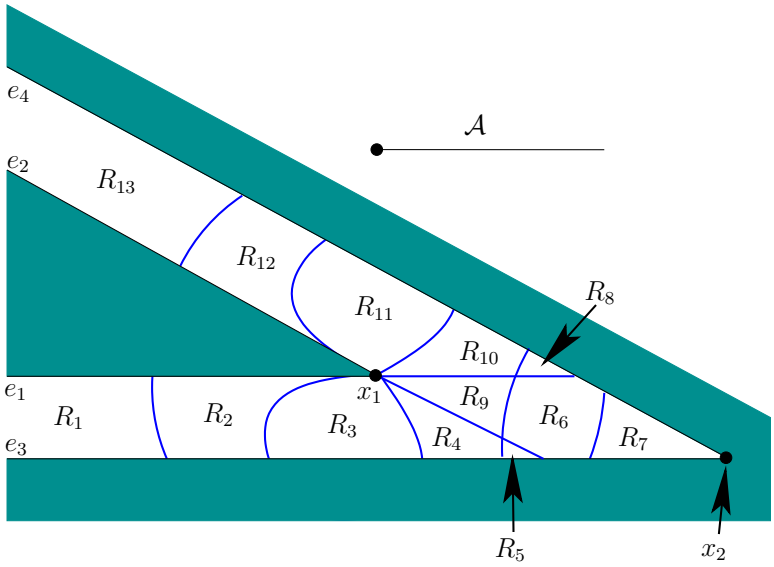


Figure 6.26: The critical curves form the boundaries of the noncritical regions in $\mathbb{R}^2$.

and form 1-cells. There are also 0-cells at places where critical curves intersect. Figure 6.26 shows an example adapted from [588]. Note that critical curves are not drawn if their corresponding configurations are all in $\mathcal{C}_{obs}$. The method still works correctly if they are included, but unnecessary cell boundaries are made. Just for fun, they could be used to form a nice cell decomposition of $\mathcal{C}_{obs}$, in addition to $\mathcal{C}_{free}$. Since $\mathcal{C}_{obs}$ is avoided, is seems best to avoid wasting time on decomposing it. These unnecessary cases can be detected by imagining that $\mathcal{A}$ is a laser with range L. As the laser sweeps around, only features that are contacted by the laser are relevant. Any features that are hidden from view of the laser correspond to unnecessary boundaries.

After the cell decomposition has been constructed in $\mathbb{R}^2$, it needs to be lifted into $\mathbb{R}^2 \times [0, 2\pi]/\sim$. This generates a cylinder of 3-cells above each 2D noncritical region, R. The roadmap could easily be defined to have a vertex for every 3-cell and 2-cell, which would be consistent with previous cell decompositions; however, vertices at 2-cells are not generated here to make the coming example easier to understand. Each 3-cell, $\{R, [f_i, f_{i+1}]\}$, corresponds to the vertex in a roadmap. The roadmap edges connect neighboring 3-cells that have a 2-cell as part of their common boundary. This means that in $\mathbb{R}^2$ they share a one-dimensional portion of a critical curve.

Constructing the roadmap The problem is to determine which 3-cells are actually adjacent. Figure 6.27 depicts the cases in which connections need to be made. The xy plane is represented as one axis (imagine looking in a direction parallel to it). Consider two neighboring 2-cells (noncritical regions), R and R', in the plane. It is assumed that a 1-cell (critical curve) in $\mathbb{R}^2$ separates them. The

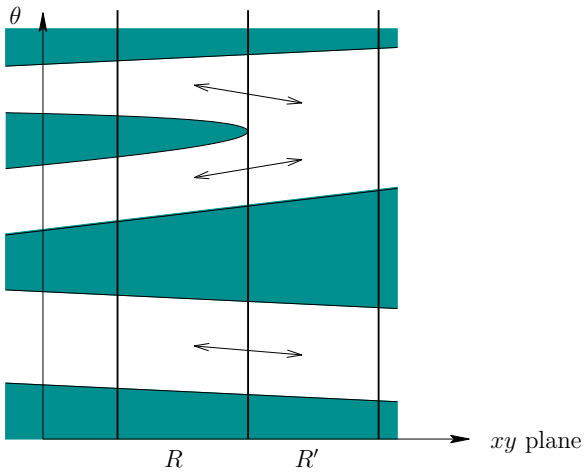


Figure 6.27: Connections are made between neighboring 3-cells that lie above neighboring noncritical regions.

task is to connect together 3-cells in the cylinders above R and R'. If neighboring cells share the same feature pair, then they are connected. This means that $\{R, [f_i, f_{i+1}]\}$ and $\{R', [f_i, f_{i+1}]\}$ must be connected. In some cases, one feature may change, while the interval of orientations remains unchanged. This may happen, for example, when the robot changes from contacting an edge to contacting a vertex of the edge. In these cases, a connection must also be made. One case illustrated in Figure 6.27 is when a splitting or merging of orientation intervals occurs. Traveling from R to R', the figure shows two regions merging into one. In this case, connections must be made from each of the original two 3-cells to the merged 3-cell. When constructing the roadmap edges, sample points of both the 3-cells and 2-cells should be used to ensure collision-free paths are obtained, as in the case of the vertical decomposition in Section 6.2.2. Figure 6.28 depicts the cells for the example in Figure 6.26. Each noncritical region has between one and three cells above it. Each of the various cells is indicated by a shortened robot that points in the general direction of the cell. The connections between the cells are also shown. Using the noncritical region and feature names from Figure 6.26, the resulting roadmap is depicted abstractly in Figure 6.29. Each vertex represents a 3-cell in $\mathcal{C}_{free}$, and each edge represents the crossing of a 2-cell between adjacent 3-cells. To make the roadmap consistent with previous roadmaps, we could insert a vertex into every edge and force the path to travel through the sample point of the corresponding 2-cell.

Once the roadmap has been constructed, it can be used in the same way as other roadmaps in this chapter to solve a query. Many implementation details have been neglected here. Due to the fifth case, some of the region boundaries in $\mathbb{R}^2$ are fourth-degree algebraic curves. Ways to prevent the explicit characterization of every noncritical region boundary, and other implementation details, are covered in [56]. Some of these details are also summarized in [588].

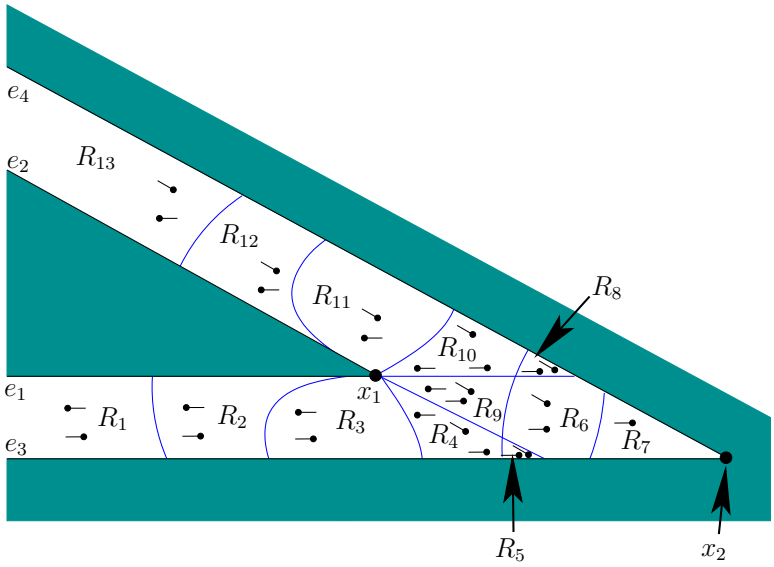


Figure 6.28: A depiction of the 3-cells above the noncritical regions. Sample rod orientations are shown for each cell (however, the rod length is shortened for clarity). Edges between cells are shown in Figure 6.29.

Complexity How many cells can there possibly be in the worst case? First count the number of noncritical regions in $\mathbb{R}^2$. There are $O(n)$ different ways to generate critical curves of the first three types because each corresponds to a single feature. Unfortunately, there are $O(n^2)$ different ways to generate bitangents and the Conchoid of Nicomedes because these are based on pairs of features. Assuming no self-intersections, a collection of $O(n^2)$ curves in $\mathbb{R}^2$, may intersect to generate at most $O(n^4)$ regions. Above each noncritical region in $\mathbb{R}^2$, there could be a cylinder of $O(n)$ 3-cells. Therefore, the size of the cell decomposition is $O(n^5)$ in the worst case. In practice, however, it is highly unlikely that all of these intersections will occur, and the number of cells is expected to be reasonable. In [851], an $O(n^5)$-time algorithm is given to construct the cell decomposition. Algorithms that have much better running time are mentioned in Section 6.5.3, but they are more complicated to understand and implement.

6.4 Computational Algebraic Geometry

This section presents algorithms that are so general that they solve any problem of Formulation 4.1 and even the closed-chain problems of Section 4.4. It is amazing that such algorithms exist; however, it is also unfortunate that they are both extremely challenging to implement and not efficient enough for most applications. The concepts and tools of this section were mostly developed in the context of computational real algebraic geometry [77, 250]. They are powerful enough to conquer numerous problems in robotics, computer vision, geometric modeling, computer-

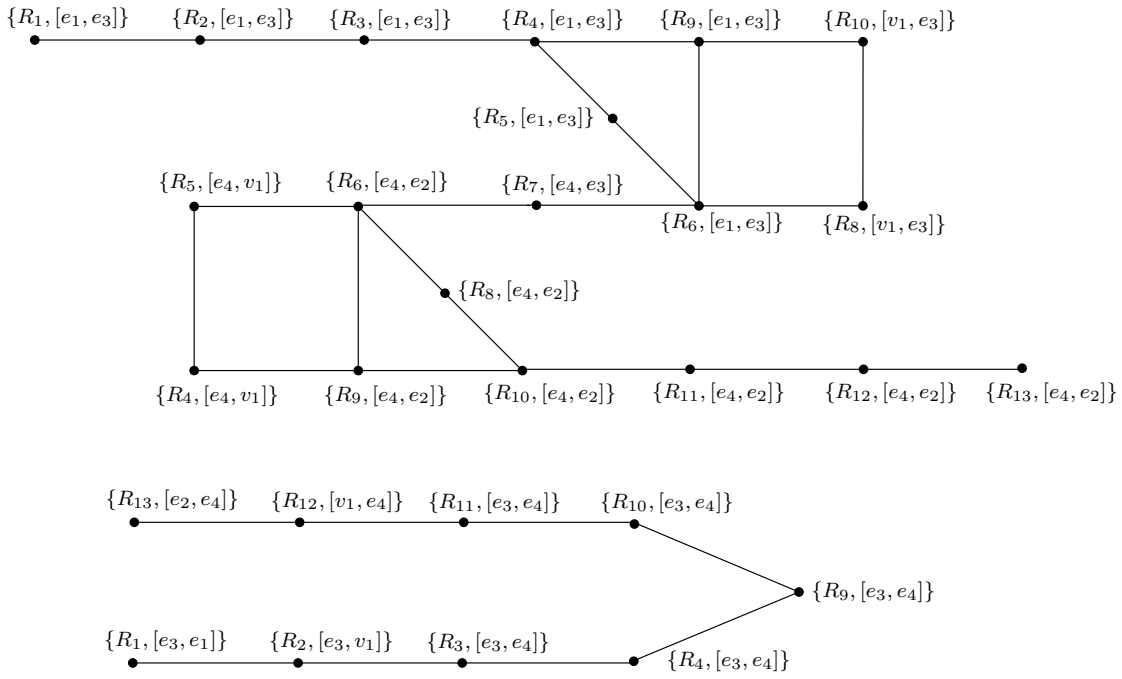


Figure 6.29: The roadmap corresponding to the example in Figure 6.26.

aided design, and geometric theorem proving. One of these problems happens to be motion planning, for which the connection to computational algebraic geometry was first recognized in [852].

6.4.1 Basic Definitions and Concepts

This section builds on the semi-algebraic model definitions from Section 3.1 and the polynomial definitions from Section 4.4.1. It will be assumed that $\mathcal{C} \subseteq \mathbb{R}^n$, which could for example arise by representing each copy of $SO(2)$ or $SO(3)$ in its 2×2 or 3×3 matrix form. For example, in the case of a 3D rigid body, we know that $\mathcal{C} = \mathbb{R}^3 \times \mathbb{RP}^3$, which is a six-dimensional manifold, but it can be embedded in $\mathbb{R}^{12}$, which is obtained from the Cartesian product of $\mathbb{R}^3$ and the set of all 3×3 matrices. The constraints that force the matrices to lie in $SO(2)$ or $SO(3)$ are polynomials, and they can therefore be added to the semi-algebraic models of $\mathcal{C}_{obs}$ and $\mathcal{C}_{free}$. If the dimension of $\mathcal{C}$ is less than n, then the algorithm presented below is sufficient, but there are some representation and complexity issues that motivate using a special parameterization of $\mathcal{C}$ to make both dimensions the same while altering the topology of $\mathcal{C}$ to become homeomorphic to $\mathbb{R}^n$. This is discussed briefly in Section 6.4.2.

Suppose that the models in $\mathbb{R}^n$ are all expressed using polynomials from $\mathbb{Q}[x_1, \ldots, x_n]$, the set of polynomials[6] over the field of rational numbers $\mathbb{Q}$. Let $f \in \mathbb{Q}[x_1, \ldots, x_n]$ denote a polynomial.

[6]It will be explained shortly why $\mathbb{Q}[x_1, \ldots, x_n]$ is preferred over $\mathbb{R}[x_1, \ldots, x_n]$.

Tarski sentences Recall the logical predicates that were formed in Section 3.1. They will be used again here, but now they are defined with a little more flexibility. For any $f \in \mathbb{Q}[x_1, \ldots, x_n]$, an *atom* is an expression of the form $f \bowtie 0$, in which $\bowtie$ may be any relation in the set $\{=, \neq, <, >, \leq, \geq\}$. In Section 3.1, such expressions were used to define logical predicates. Here, we assume that relations other than $\leq$ can be used and that the vector of polynomial variables lies in $\mathbb{R}^n$.

A *quantifier-free formula*, $\phi(x_1, \ldots, x_n)$, is a logical predicate composed of atoms and logical connectives, "and," "or," and "not," which are denoted by $\wedge$, $\vee$, and $\neg$, respectively. Each atom itself is considered as a logical predicate that yields TRUE if and only if the relation is satisfied when the polynomial is evaluated at the point $(x_1, \ldots, x_n) \in \mathbb{R}^n$.

Example 6.2 (An Example Predicate) Let ϕ be a predicate over $\mathbb{R}^3$, defined as

$$\phi(x_1, x_2, x_3) = (x_1^2 x_3 - x_2^4 < 0) \vee \big(\neg(3x^2x^3 \neq 0) \wedge (2x_3^2 - x_1x_2x_3 + 2 \geq 0)\big). \quad (6.8)$$

The precedence order of the connectives follows the laws of Boolean algebra. ■

Let a *quantifier* $\mathcal{Q}$ be either of the symbols, $\forall$, which means "for all," or $\exists$, which means "there exists." A *Tarski sentence* Φ is a logical predicate that may additionally involve quantifiers on some or all of the variables. In general, a Tarski sentence takes the form

$$\Phi(x_1, \ldots, x_{n-k}) = (\mathcal{Q}z_1)(\mathcal{Q}z_2) \ldots (\mathcal{Q}z_k)\ \phi(z_1, \ldots, z_k, x_1, \ldots, x_{n-k}), \quad (6.9)$$

in which the z_i are the *quantified variables*, the x_i are the *free variables*, and ϕ is a quantifier-free formula. The quantifiers do not necessarily have to appear at the left to be a valid Tarski sentence; however, any expression can be manipulated into an equivalent expression that has all quantifiers in front, as shown in (6.9). The procedure for moving quantifiers to the front is as follows [705]: 1) Eliminate any redundant quantifiers; 2) rename some of the variables to ensure that the same variable does not appear both free and bound; 3) move negation symbols as far inward as possible; and 4) push the quantifiers to the left.

Example 6.3 (Several Tarski Sentences) Tarski sentences that have no free variables are either TRUE or FALSE in general because there are no arguments on which the results depend. The sentence

$$\Phi = \forall x \exists y\ (x^2 - y < 0), \quad (6.10)$$

is TRUE because for any $x \in \mathbb{R}$, some $y \in \mathbb{R}$ can always be chosen so that $y > x^2$. In the general notation of (6.9), this example becomes $\mathcal{Q}z_1 = \forall x$, $\mathcal{Q}z_2 = \exists y$, and $\phi(z_1, z_2) = (x^2 - y < 0)$.

Swapping the order of the quantifiers yields the Tarski sentence

$$\Phi = \exists y \forall x\ (x^2 - y < 0), \quad (6.11)$$

which is FALSE because for any y, there is always an x such that $x^2 > y$.

Now consider a Tarski sentence that has a free variable:

$$\Phi(z) = \exists y \forall x \; (x^2 - zx^2 - y < 0). \tag{6.12}$$

This yields a function $\Phi : \mathbb{R} \rightarrow \{\text{TRUE}, \text{FALSE}\}$, in which

$$\Phi(z) = \begin{cases} \text{TRUE} & \text{if } z \geq 1 \\ \text{FALSE} & \text{if } z < 1. \end{cases} \tag{6.13}$$

An equivalent quantifier-free formula ϕ can be defined as $\phi(z) = (z > 1)$, which takes on the same truth values as the Tarski sentence in (6.12). This might make you wonder whether it is always possible to make a simplification that eliminates the quantifiers. This is called the *quantifier-elimination problem*, which will be explained shortly. ■

The decision problem The sentences in (6.10) and (6.11) lead to an interesting problem. Consider the set of all Tarski sentences that have no free variables. The subset of these that are TRUE comprise the *first-order theory of the reals*. Can an algorithm be developed to determine whether such a sentence is true? This is called the *decision problem* for the first-order theory of the reals. At first it may appear hopeless because $\mathbb{R}^n$ is uncountably infinite, and an algorithm must work with a finite set. This is a familiar issue faced throughout motion planning. The sampling-based approaches in Chapter 5 provided one kind of solution. This idea could be applied to the decision problem, but the resulting lack of completeness would be similar. It is not possible to check all possible points in $\mathbb{R}^n$ by sampling. Instead, the decision problem can be solved by constructing a combinatorial representation that exactly represents the decision problem by partitioning $\mathbb{R}^n$ into a finite collection of regions. Inside of each region, only one point needs to be checked. This should already seem related to cell decompositions in motion planning; it turns out that methods developed to solve the decision problem can also conquer motion planning.

The quantifier-elimination problem Another important problem was exemplified in (6.12). Consider the set of all Tarski sentences of the form (6.9), which may or may not have free variables. Can an algorithm be developed that takes a Tarski sentence Φ and produces an equivalent quantifier-free formula ϕ? Let $x_1, \ldots, x_n$ denote the free variables. To be equivalent, both must take on the same true values over $\mathbb{R}^n$, which is the set of all assignments $(x_1, \ldots, x_n)$ for the free variables.

Given a Tarski sentence, (6.9), the *quantifier-elimination problem* is to find a quantifier-free formula ϕ such that

$$\Phi(x_1, \ldots, x_n) = \phi(x_1, \ldots, x_n) \tag{6.14}$$

for all $(x_1, \ldots, x_n) \in \mathbb{R}^n$. This is equivalent to constructing a semi-algebraic model because ϕ can always be expressed in the form

$$\phi(x_1, \ldots, x_n) = \bigvee_{i=1}^{k} \bigwedge_{j=1}^{m_i} (f_{i,j}(x_1, \ldots, x_n) \bowtie 0) \,, \tag{6.15}$$

in which $\bowtie$ may be either $<$, $=$, or $>$. This appears to be the same (3.6), except that (6.15) uses the relations $<$, $=$, and $>$ to allow open and closed semi-algebraic sets, whereas (3.6) only used $\leq$ to construct closed semi-algebraic sets for $\mathcal{O}$ and $\mathcal{A}$.

Once again, the problem is defined on $\mathbb{R}^n$, which is uncountably infinite, but an algorithm must work with a finite representation. This will be achieved by the cell decomposition technique presented in Section 6.4.2.

Semi-algebraic decomposition As stated in Section 6.3.1, motion planning inside of each cell in a complex should be trivial. To solve the decision and quantifier-elimination problems, a cell decomposition was developed for which these problems become trivial in each cell. The decomposition is designed so that only a single point in each cell needs to be checked to solve the decision problem.

The semi-algebraic set $Y \subseteq \mathbb{R}^n$ that is expressed with (6.15) is

$$Y = \bigcup_{i=1}^{k} \bigcap_{j=1}^{m_i} \{(x_1, \ldots, x_n) \in \mathbb{R}^n \mid \mathrm{sgn}(f_{i,j}(x_1, \ldots, x_n)) = s_{i,j}\} \,, \tag{6.16}$$

in which sgn is the sign function, and each $s_{i,j} \in \{-1, 0, 1\}$, which is the range of sgn. Once again the nice relationship between set-theory and logic, which was described in Section 3.1, appears here. We convert from a set-theoretic description to a logical predicate by changing $\cup$ and $\cap$ to $\vee$ and $\wedge$, respectively.

Let $\mathcal{F}$ denote the set of $m = \sum_{i=1}^{k} m_i$ polynomials that appear in (6.16). A *sign assignment* with respect to $\mathcal{F}$ is a vector-valued function, $\mathrm{sgn}_{\mathcal{F}} : \mathbb{R}^n \rightarrow \{-1, 0, 1\}^m$. Each $f \in \mathcal{F}$ has a corresponding position in the sign assignment vector. At this position, the sign, $\mathrm{sgn}(f(x_1, \ldots, x_n)) \in \{-1, 0, 1\}$, appears. A *semi-algebraic decomposition* is a partition of $\mathbb{R}^n$ into a finite set of connected regions that are each *sign invariant*. This means that inside of each region, $\mathrm{sgn}_{\mathcal{F}}$ must remain constant. The regions will not be called *cells* because a semi-algebraic decomposition is not necessarily a singular complex as defined in Section 6.3.1; the regions here may contain holes.

Example 6.4 (Sign assignment) Recall Example 3.1 and Figure 3.4 from Section 3.1.2. Figure 3.4a shows a sign assignment for a case in which there is only one polynomial, $\mathcal{F} = \{x^2 + y^2 - 4\}$. The sign assignment is defined as

$$\mathrm{sgn}_{\mathcal{F}}(x, y) = \begin{cases} -1 & \text{if } x^2 + y^2 - 4 < 0 \\ 0 & \text{if } x^2 + y^2 - 4 = 0 \\ 1 & \text{if } x^2 + y^2 - 4 > 0. \end{cases} \tag{6.17}$$