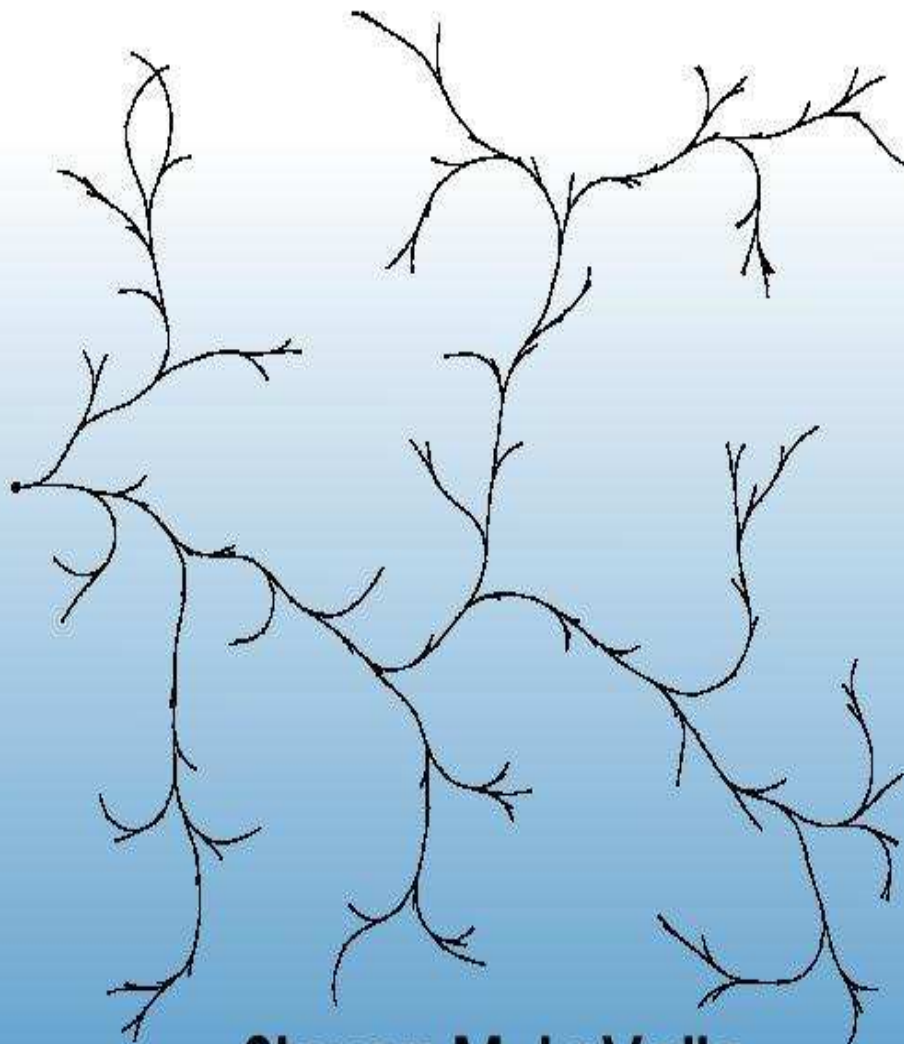


PLANNING ALGORITHMS



Steven M. LaValle

PLANNING ALGORITHMS

Steven M. LaValle

University of Illinois

Copyright 1999-2004

Contents

| | |
|--|-----------|
| Preface | xi |
| 0.1 What is meant by “Planning Algorithms”? | xi |
| 0.2 Who is the Intended Audience? | xii |
| 0.3 Suggested Use | xii |
| 0.4 Acknowledgments | xii |
| 0.5 Help! | xiii |
| | |
| I Introductory Material | 1 |
| | |
| 1 Introduction | 3 |
| 1.1 Planning to Plan | 3 |
| 1.2 Illustrative Problems | 5 |
| 1.3 Basic Ingredients of Planning | 5 |
| 1.4 What is a Planning Algorithm? | 10 |
| 1.5 Organization of the Book | 14 |
| | |
| 2 Discrete Planning | 19 |
| 2.1 Introduction | 19 |
| 2.2 Definition of Discrete Feasible Planning | 20 |
| 2.3 Searching for Feasible Plans | 24 |
| 2.3.1 General Forward Search | 25 |
| 2.3.2 Particular Forward Search Methods | 28 |
| 2.3.3 Other General Search Schemes | 32 |
| 2.3.4 A Unified View of the Search Methods | 34 |
| 2.4 Discrete Optimal Planning | 36 |
| 2.4.1 Optimal Fixed-Length Plans | 38 |
| 2.4.2 The General Case | 43 |
| 2.4.3 Dijkstra Revisited | 47 |
| 2.5 Logic-Based Representations of Planning | 49 |
| 2.5.1 A STRIPS-Like Representation | 50 |
| 2.5.2 Converting to the State Space Representation | 53 |
| 2.5.3 Logic-Based Planning | 54 |

| | | |
|-----------|--|------------|
| II | Motion Planning | 57 |
| 3 | Geometric Representations and Transformations | 61 |
| 3.1 | Geometric Modeling | 61 |
| 3.1.1 | Polygonal and Polyhedral Models | 62 |
| 3.1.2 | Semi-Algebraic Models | 68 |
| 3.1.3 | Other Models | 70 |
| 3.2 | Rigid Body Transformations | 73 |
| 3.2.1 | General Concepts | 73 |
| 3.2.2 | 2D Transformations | 74 |
| 3.2.3 | 3D Transformations | 77 |
| 3.3 | Transformations of Kinematic Chains of Bodies | 80 |
| 3.3.1 | A Kinematic Chain in \mathbb{R}^2 | 80 |
| 3.3.2 | A Kinematic Chain in \mathbb{R}^3 | 84 |
| 3.4 | Transformations of Kinematic Trees | 95 |
| 3.5 | Nonrigid Transformations | 103 |
| 4 | The Configuration Space | 111 |
| 4.1 | Basic Topological Concepts | 112 |
| 4.1.1 | Topological Spaces | 112 |
| 4.1.2 | Manifolds | 118 |
| 4.1.3 | Paths and Connectivity | 123 |
| 4.2 | Defining the Configuration Space | 128 |
| 4.2.1 | 2D Rigid Bodies: $SE(2)$ | 129 |
| 4.2.2 | 3D Rigid Bodies: $SE(3)$ | 132 |
| 4.2.3 | Chains and Trees of Bodies | 138 |
| 4.3 | Configuration Space Obstacles | 140 |
| 4.3.1 | Definition of the Basic Motion Planning Problem | 140 |
| 4.3.2 | Explicitly Modeling \mathcal{C}_{obs} : The Translational Case | 142 |
| 4.3.3 | Explicitly Modeling \mathcal{C}_{obs} : The General Case | 148 |
| 4.4 | Kinematic Closure and Varieties | 152 |
| 4.4.1 | Mathematical Concepts | 153 |
| 4.4.2 | Kinematic Chains in \mathbb{R}^2 | 156 |
| 4.4.3 | Defining the Variety for General Problems | 161 |
| 5 | Sampling-Based Motion Planning | 171 |
| 5.1 | Distance and Volume in C-Space | 172 |
| 5.1.1 | Metric Spaces | 172 |
| 5.1.2 | Important Metric Spaces for Motion Planning | 174 |
| 5.1.3 | Basic Measure Theory Definitions | 177 |
| 5.1.4 | Using the Correct Measure | 179 |
| 5.2 | Sampling Theory | 180 |
| 5.2.1 | Motivation and Basic Concepts | 180 |
| 5.2.2 | Random Sampling | 183 |

| | | |
|----------|--|------------|
| 5.2.3 | Low-Dispersion Sampling | 187 |
| 5.2.4 | Low-Discrepancy Sampling | 192 |
| 5.3 | Collision Detection | 195 |
| 5.3.1 | Basic Concepts | 196 |
| 5.3.2 | Hierarchical Methods | 197 |
| 5.3.3 | Incremental Methods | 198 |
| 5.3.4 | Checking a Path Segment | 200 |
| 5.4 | Incremental Sampling and Searching | 203 |
| 5.4.1 | The General Framework | 203 |
| 5.4.2 | Adapting Classical Search Algorithms | 206 |
| 5.4.3 | Randomized Potential Fields | 213 |
| 5.4.4 | Other Methods | 216 |
| 5.5 | Rapidly-Exploring Dense Trees | 217 |
| 5.5.1 | The Exploration Algorithm | 218 |
| 5.5.2 | Efficiently Finding Nearest Points | 222 |
| 5.5.3 | Using the Trees for Planning | 225 |
| 5.6 | Roadmap Methods for Multiple Queries | 227 |
| 5.6.1 | The Basic Method | 228 |
| 5.6.2 | Visibility Roadmap | 232 |
| 5.6.3 | Heuristics for Improving Roadmaps | 233 |
| 6 | Combinatorial Motion Planning | 245 |
| 6.1 | Introduction | 245 |
| 6.2 | Polygonal Obstacle Regions | 247 |
| 6.2.1 | Representation | 248 |
| 6.2.2 | Vertical Cell Decomposition | 250 |
| 6.2.3 | Maximum Clearance Roadmaps | 256 |
| 6.2.4 | Shortest Path Roadmaps | 257 |
| 6.3 | Cell Decompositions | 260 |
| 6.3.1 | General Definitions | 261 |
| 6.3.2 | 2D Decompositions | 265 |
| 6.3.3 | 3D Vertical Decomposition | 267 |
| 6.3.4 | A Decomposition for a Line-Segment Robot | 270 |
| 6.4 | Computational Algebraic Geometry | 278 |
| 6.4.1 | Basic Definitions and Concepts | 278 |
| 6.4.2 | Cylindrical Algebraic Decomposition | 283 |
| 6.4.3 | Canny's Roadmap Algorithm | 291 |
| 6.5 | Complexity of Motion Planning | 296 |
| 6.5.1 | Lower Bounds | 296 |
| 6.5.2 | Davenport-Schinzel Sequences | 300 |
| 6.5.3 | Upper Bounds | 302 |

| | | |
|------------|--|------------|
| 7 | Extensions of Basic Motion Planning | 307 |
| 7.1 | Time-Varying Problems | 308 |
| 7.1.1 | Problem Formulation | 308 |
| 7.1.2 | Direct Solutions | 311 |
| 7.1.3 | The Velocity Tuning Method | 313 |
| 7.2 | Multiple Robots | 315 |
| 7.2.1 | Problem Formulation | 316 |
| 7.2.2 | Decoupled Planning | 318 |
| 7.3 | Hybrid Systems: Discrete and Continuous | 325 |
| 7.3.1 | General Framework | 325 |
| 7.3.2 | Manipulation Planning | 329 |
| 7.4 | Planning for Closed Kinematic Chains | 336 |
| 7.4.1 | Adaptation of Motion Planning Algorithms | 337 |
| 7.4.2 | Active-Passive Link Decompositions | 342 |
| 7.5 | Folding Problems in Robotics and Biology | 346 |
| 7.6 | Coverage Planning | 354 |
| 7.7 | Optimal Motion Planning | 356 |
| 7.7.1 | Optimality for One Robot | 357 |
| 7.7.2 | Multiple-Robot Optimality | 365 |
| 8 | Feedback Motion Strategies | 369 |
| 8.1 | Feedback in Discrete Planning | 369 |
| 8.2 | Vector Fields on Manifolds | 369 |
| 8.3 | Feedback Strategies in Motion Planning | 369 |
| 8.3.1 | Navigation Functions | 370 |
| 8.4 | Combinatorial Algorithms | 371 |
| 8.4.1 | Harmonic Functions | 371 |
| 8.4.2 | Feedback Strategies over Complexes | 371 |
| 8.5 | Sampling-Based Algorithms | 371 |
| 8.5.1 | Sampling-Based Neighborhood Graph | 371 |
| 8.6 | Computing Optimal Feedback Strategies | 375 |
| III | Decision-Theoretic Planning | 377 |
| 9 | Basic Decision Theory | 381 |
| 9.1 | Basic Definitions | 381 |
| 9.2 | A Game Against Nature | 383 |
| 9.2.1 | Having a single observation | 384 |
| 9.3 | Applications of Optimal Decision Making | 386 |
| 9.3.1 | Classification | 386 |
| 9.3.2 | Parameter Estimation | 387 |
| 9.4 | Utility Theory | 388 |
| 9.4.1 | Choosing a Good Reward | 388 |

| | | |
|-----------|--|------------|
| 9.4.2 | Axioms of Rationality | 389 |
| 9.5 | Criticisms of Decision Theory | 390 |
| 9.5.1 | Nondeterministic decision making | 390 |
| 9.5.2 | Bayesian decision making | 391 |
| 9.6 | Multiobjective Optimality | 392 |
| 9.6.1 | Scalarizing L | 393 |
| 9.7 | Two-Player Zero Sum Games | 393 |
| 9.7.1 | Overview of game theory | 393 |
| 9.7.2 | Regret | 396 |
| 9.7.3 | Saddle Points | 397 |
| 9.7.4 | Mixed Strategies | 397 |
| 9.7.5 | Computation of Equilibria | 399 |
| 9.8 | Nonzero Sum Games | 402 |
| 9.8.1 | Nash Equilibria | 402 |
| 9.8.2 | Admissibility | 403 |
| 9.8.3 | The Prisoner's Dilemma | 404 |
| 9.8.4 | Nash Equilibrium for mixed strategies | 404 |
| 10 | Sequential Decision Theory | 407 |
| 10.1 | Basic Definitions | 407 |
| 10.1.1 | Non-Deterministic Forward Projection | 407 |
| 10.1.2 | Probabilistic Forward Projection | 408 |
| 10.1.3 | Strategies | 408 |
| 10.2 | Dynamic Programming over Discrete Spaces | 409 |
| 10.3 | Infinite Horizon Problems | 411 |
| 10.3.1 | Average Loss-Per-Stage | 412 |
| 10.3.2 | Discounted Loss | 412 |
| 10.3.3 | Optimization in the Discounted Loss Model | 412 |
| 10.3.4 | Forward Dynamic Programming | 413 |
| 10.3.5 | Backwards Dynamic Programming | 413 |
| 10.3.6 | Policy Iteration | 415 |
| 10.4 | Dynamic Programming over Continuous Spaces | 416 |
| 10.4.1 | Reformulating Motion Planning | 416 |
| 10.4.2 | The Algorithm | 419 |
| 10.5 | Reinforcement Learning | 424 |
| 10.5.1 | Stochastic Iterative Algorithms | 425 |
| 10.5.2 | Finding an Optimal Strategy: Q-learning | 425 |
| 10.6 | Sequential Game Theory | 426 |
| 10.6.1 | Dynamic Programming over Sequential Games | 427 |
| 10.6.2 | Algorithms for Special Games | 427 |

| | |
|--|------------|
| 11 The Information Space | 429 |
| 11.1 Discrete State Spaces | 431 |
| 11.1.1 Sensors | 431 |
| 11.1.2 Defining the Information Space | 435 |
| 11.1.3 Defining a Planning Problem | 437 |
| 11.2 Alternative Representations of Information Spaces | 440 |
| 11.2.1 Nondeterministic Derived Information States | 440 |
| 11.2.2 Probabilistic Derived Information States | 443 |
| 11.2.3 Collapsing the Information Space | 446 |
| 11.2.4 Limited Memory Models | 448 |
| 11.3 Examples for Discrete State Spaces | 448 |
| 11.3.1 Basic Nondeterministic Examples | 448 |
| 11.3.2 Nondeterministic Finite Automata | 452 |
| 11.3.3 Probabilistic Examples | 457 |
| 11.4 Continuous State Spaces | 457 |
| 11.4.1 Discrete-Stage Information Spaces | 457 |
| 11.4.2 Continuous-Time Information Spaces | 458 |
| 11.4.3 Alternative Representations | 460 |
| 11.4.4 Approximating Information States | 461 |
| 11.5 Sensors for Continuous Spaces | 461 |
| 11.6 Examples for Continuous State Spaces | 462 |
| 11.6.1 Projection Sensors | 462 |
| 11.6.2 Sensorless Manipulation | 464 |
| 11.6.3 Environment Spaces | 465 |
| 11.7 State Estimation | 465 |
| 11.7.1 Mapping Histories to States | 465 |
| 11.7.2 Kalman Filtering | 465 |
| 11.8 Multiple Decision Makers | 466 |
| 11.8.1 Information Spaces for Everyone | 466 |
| 11.8.2 Extended Form Games | 467 |
| 11.8.3 Examples | 467 |
| 12 Planning in the Information Space | 471 |
| 12.1 Information Spaces over Sets of Environments | 471 |
| 12.1.1 Maze Searching | 471 |
| 12.1.2 Bug Algorithms | 472 |
| 12.1.3 Gap Navigation Trees | 473 |
| 12.2 Localization and Map Building | 473 |
| 12.3 Manipulation with Minimal Information | 473 |
| 12.4 Visibility-Based Pursuit-Evasion | 473 |
| 12.5 Preimage Planning | 476 |
| 12.6 Algorithms for Solving POMDPs | 476 |
| 12.7 Dynamic Programming on Information Spaces | 476 |

| | | |
|-----------|--|------------|
| IV | Planning Under Differential Constraints | 479 |
| 13 | Differential Models | 481 |
| 13.1 | Motivation | 481 |
| 13.2 | Representing Differential Constraints | 482 |
| 13.3 | Kinematics for Wheeled Systems | 485 |
| 13.3.1 | A Simple Car | 485 |
| 13.3.2 | A Continuous-Steering Car | 486 |
| 13.3.3 | A Car Pulling Trailers | 487 |
| 13.3.4 | A Differential Drive | 487 |
| 13.4 | Rigid-Body Dynamics | 488 |
| 13.5 | Multiple-Body Dynamics | 490 |
| 13.6 | More Examples | 490 |
| 14 | Nonholonomic System Theory | 491 |
| 14.1 | Vector Fields and Distributions | 491 |
| 14.2 | The Lie Bracket | 493 |
| 14.3 | Integrability and Controllability | 495 |
| 15 | Planning Under Differential Constraints | 499 |
| 15.1 | Problem formulations | 499 |
| 15.2 | Steering Methods | 499 |
| 15.2.1 | Geodesic curve families | 499 |
| 15.2.2 | Series Methods | 501 |
| 15.3 | Sampling-Based Planning Methods | 501 |
| 15.3.1 | An Incremental Search Framework | 502 |
| 15.3.2 | Tree-Based Dynamic Programming | 506 |
| 15.3.3 | RDT-Based Methods | 507 |
| 15.3.4 | Other Sampling-Based Methods | 508 |
| 15.4 | Gradient-Based Optimization Techniques | 508 |
| 15.5 | Optimal Feedback Strategies | 508 |
| 15.5.1 | Problem Definition | 508 |
| 15.5.2 | Exact Solutions for Linear Systems | 508 |
| 15.5.3 | Functional Dynamic Programming | 508 |

Preface

0.1 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 of 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 is focused primarily 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 compute appropriate actions. Within control theory, the focus of the presentation is on algorithms that numerically compute feasible trajectories or even optimal feedback control laws. This means that analytical techniques, which account for the majority of control theory literature, are not addressed 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, we make no such distinctions. Ignoring old connotations that come with the terms, “planning” or “control” could be used interchangeably.

Both can refer to some kind of decision making in this text, with no associated notion of “high” or “low” level. A hierarchical planning (or control!) strategy could be developed in any case.

0.2 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 and artificial intelligence. 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, and computer graphics.

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 at the end of some chapters.

0.3 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.

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 Chapters 3-7.

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.

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

0.4 Acknowledgments

I am very grateful to many students and colleagues who have given me extensive feedback and advice in developing this text.

Many thanks go to Stefano Carpin, Sanjit Jhala, Stephane Redon, Sanketh Shetty, Mohan Sirchabesan, and Zbynek Winkler for pointing out mistakes in the on-line manuscript.

I also appreciate the efforts of graduate students in my courses who scribed class notes which served as an early draft for some parts. These include students at Iowa State: Brian George, ..., and students at the University of Illinois: Shamsi Tamara Iqbal, Rishi Talreja, Sherwin Tam, Benjamin Tovar ...

I am also thankful for the supportive environments provided both by Iowa State University and the University of Illinois. In both universities, I have been able to develop courses for which the material presented here has been developed and refined.

I sincerely thank Krzysztof Kozłowski and his staff at the Politechnika Poznańska for their help during my sabbatical in Poland.

0.5 Help!

Since the text appears on the web, it is easy for me to incorporate feedback from readers. This will be very helpful as I complete this project. If you find mistakes, have requests for coverage of topics, find some explanations difficult to follow, have suggested exercises, etc., please let me know by sending mail to lavalle@cs.uiuc.edu. Note that this book is current a work in progress. Please be patient as I update parts over the coming year or two.

Part I

Introductory Material

Chapter 1

Introduction

Chapter Status



What does this mean? Check
<http://msl.cs.uiuc.edu/planning/status.html>
for information on the latest version.

1.1 Planning to Plan

Planning is a term that means different things to different groups of people. A fundamental need in robotics is to have algorithms that can automatically tell robots how to move when they are given high-level commands. 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 3D 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 uncertainties, 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 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, but the problem is already challenging enough without these.

In artificial intelligence, the term *AI planning* takes 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. 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 here in detail.

Given the broad range of problems to which the term planning has been applied in the artificial intelligence, control theory, and robotics communities, one 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 will not be limited to combinatorics and asymptotic complexity analysis, which is the main focus in pure algorithms. The focus here includes numerous modeling considerations and concepts that are not necessarily algorithmic, but aid in solving or analyzing planning problems.

The obvious goal of virtually any planning algorithm is to produce a *plan*. Natural questions are: What is a plan? How is a plan represented? What is it supposed to achieve? How will its quality be evaluated? Who or what is going to use it? Regarding the user of the plan, it obviously depends on the application.

In most applications, an algorithm will execute the plan; however, sometimes the user may be a human. Imagine, for example, that the planning algorithm provides you with an investment strategy. A generic term that will be used frequently here to refer to the user is *decision maker*. In robotics, the decision maker is simply referred to as a *robot*. In artificial intelligence and related areas, it has become popular in recent years to use the term *agent*, possibly with adjectives to make *intelligent agent* or *software agent*. Control theory usually refers to the decision maker as a *system* or *plant*. 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, it is important to remember that terms like “robot”, “agent”, and “system” are interchangeable.

1.2 Illustrative Problems

This section only has a couple of pasted examples. It still needs to be written, to include other examples from discrete planning, information spaces, game theory, etc. More examples will be added gradually as other parts of the book are written.

Suppose that we have a tiny mobile robot that can move along the floor in a building. The task is to determine a path that it should follow from a starting location to a goal location, while avoiding collisions. A reasonable model can be formulated by assuming that the robot is a moving point in a two-dimensional environment that contains obstacles.

Let $\mathcal{W} = \mathbb{R}^2$ denote a two-dimensional *world* which contains a point *robot*, denoted by \mathcal{A} . A subset, \mathcal{O} , of the world is called the *obstacle region*. Let the remaining portion of the world, $\mathcal{W} \setminus \mathcal{O}$ be referred to as the *free space*. The task is to design an algorithm that accepts an obstacle region defined by a set of polygons, an initial position, and a goal position. The algorithm must return a path that will bring the robot from the initial position to the goal position, while only traversing the free space. Algorithms that find exact solutions to this problem are given in Section 6.2.

Figures 1.2 and 1.3 show considerably more challenging problems.

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 will involve a *state space* that captures all possible situations that could exist. The *state* could, for example, represent the

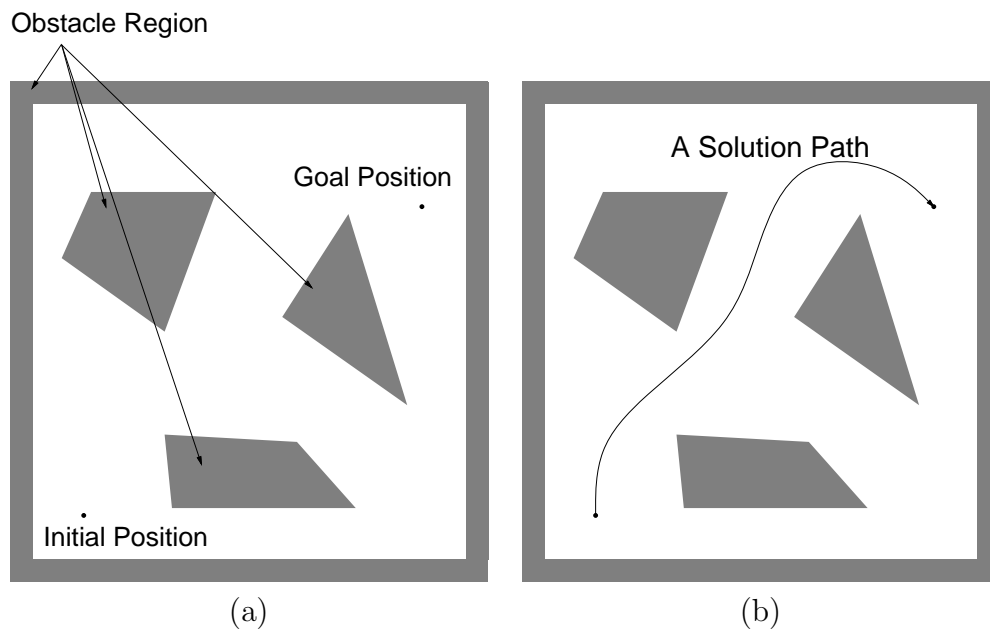


Figure 1.1: A simple illustration of the two dimensional path planning problem: a) The obstacles, initial position, and goal positions are specified as input; b) A path planning algorithm will compute a collision free path from the initial position to the goal position.

configuration 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 through most of planning is that the state space will usually be 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 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 of motions is not specified in the planning problem. Just as in the case of state, time may be either discrete or continuous. In the latter case,

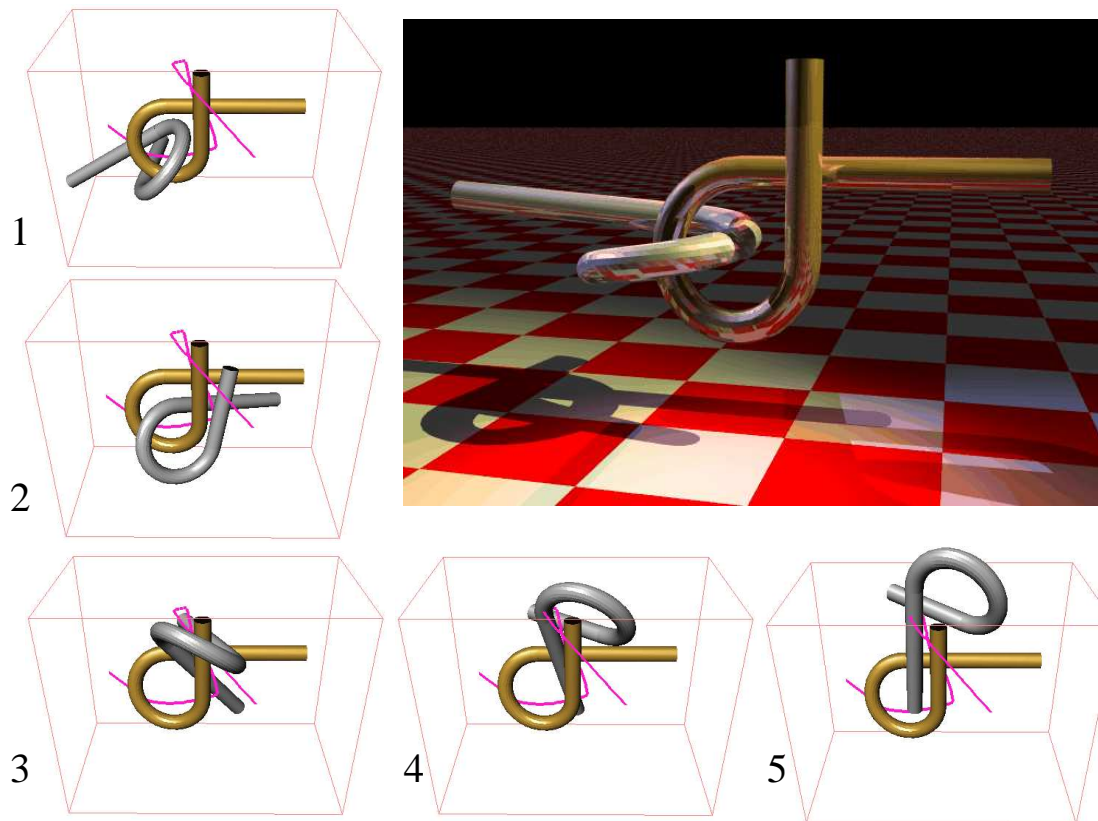


Figure 1.2: Remember puzzles like this? Imagine trying to solve one with an algorithm. The goal is to pull the two bars apart. This example is called the Alpha 1.0 Puzzle. It was created by Boris Yamrom, GE Corporate Research & Development Center, and posted as a research benchmark by Nancy Amato at Texas A&M University. This animation was made by James Kuffner, of Carnegie Mellon University. The solution was computed by the balanced, bidirectional RRT algorithm, developed by James Kuffner and Steve LaValle, and covered in Section 5.5



Figure 1.3: Using robots to move a piano [177]. This solution was computed using planning techniques developed by Juan Cortés, Thierry Simeon, and Jean-Paul Laumond, and are covered in Section 7.4.

we can imagine that a continuum of decisions are 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 equivalent terms are *inputs* or *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 designing paths through a continuous state space. Such paths may be expressed as the integral of differential equations, but it is an unnecessary complication in this case. For some problems uncontrollable actions could be chosen by *nature*, which interfere with the outcome, but are not specified as part of the plan. This enables various forms of uncertainty to be introduced into the planning problem.

Initial and goal states: Planning generally involves starting in some initial state and trying to arrive at a specified goal state. The actions are selected in a way that causes this to happen.

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

1. **Feasibility:** In this case, the only concern is whether the plan results in arriving at a goal state.
2. **Optimality:** Find feasible plans that optimize 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, a substantial amount of 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, such as those developed in motion planning, the solutions produced are usually not too far from optimal in practice. This reduces the amount of 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 per-

formance in terms of expected costs. Feasibility is usually associated with performing worst-case analysis of uncertainties.

A plan: In general, a plan will impose a specific strategy or behavior on decision makers. A plan might simply specify a sequence of actions to be taken; however, it may be more complicated. If it is impossible to predict future states, the plan may provide actions as a function of state. In this case, regardless of 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 action must be chosen based on whatever information is available up to the current time. This will generally be referred to as an *information state*, on which a plan will be conditioned.

1.4 What is a Planning Algorithm?

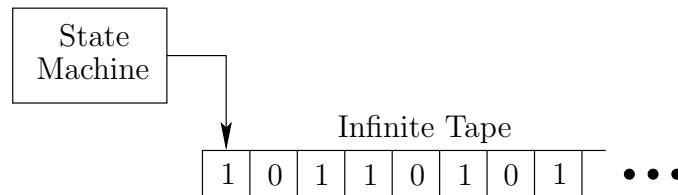


Figure 1.4: According to the Church-Turing thesis, the notion of an algorithm is equivalent to the notion of a Turing machine.

What is a planning algorithm? This is a difficult question, which is difficult to completely answer in this section without formally introducing the planning concepts that appear in later chapters. One point needs to be made clear at this point: the classical Turing machine model used to define an algorithm in theoretical computer science is insufficient to encompass planning algorithms. 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 1.4. The Church-Turing thesis states that an algorithm *is* a Turing machine (see [339, 711] 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 extends that can yield other desired outputs, such as a plan.

The trouble with using the Turing machine as a model for planning algorithms is that plans will be generally assumed to interact with a physical world, as depicted in Figure 1.5. This is fundamental to robotics and many other fields in which planning is used. Using the Turing machine as a foundation for algorithms

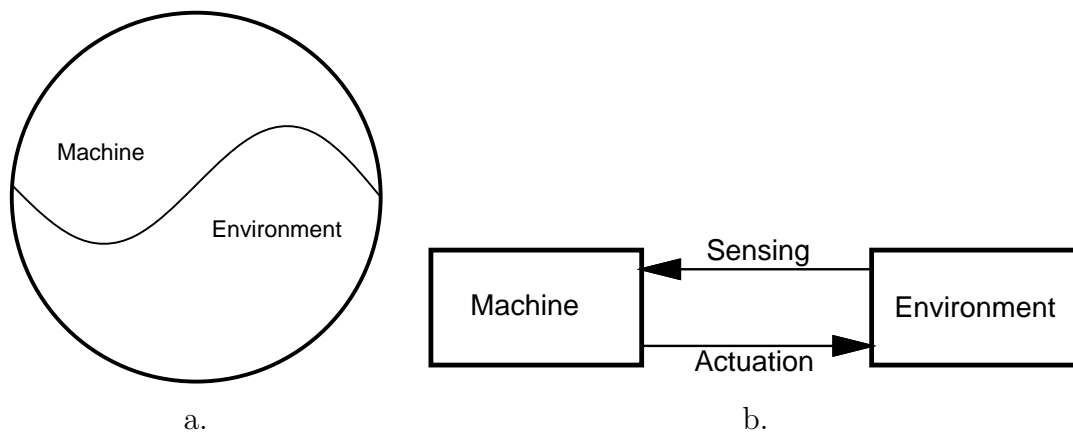


Figure 1.5: a) The boundary between machine and environment is considered as an arbitrary line that may be drawn in many ways depending on the context. b) Once the boundary has been drawn, it is assumed that the machine interacts with the environment through sensing and actuation.

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. The 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 should be explicitly modeled and written on the tape. The *on-line algorithm* model is more appropriate for these kind of problems []; however, it still does not capture a notion of planning algorithms that is sufficiently broad for the topics of this book.

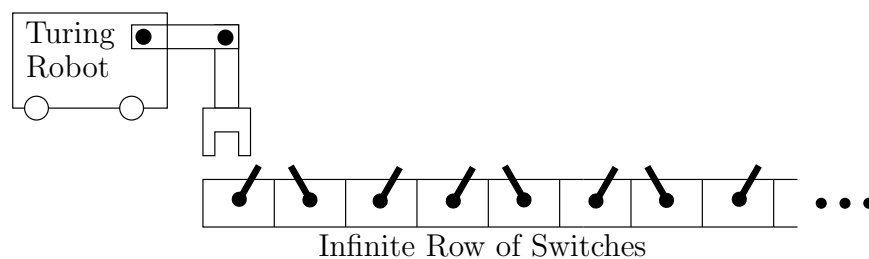


Figure 1.6: A robot could be used to simulate a Turing machine. Through manipulation, many other kinds of behavior could be obtained that fall outside of the Turing model.

The processes that can 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

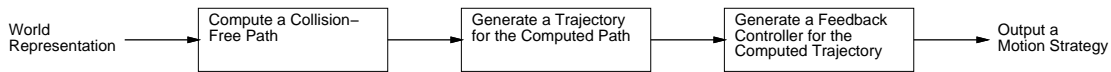


Figure 1.7: A classical model that has been used for decades in robotics.

a long row of switches as depicted in Figure 1.6. The switches serve the same purpose as the tape, and the robot carries a computer that can simulate the finite state machine.¹ The complicated interaction allowed between a robot and its environment could give rise to many other models of computation. A discussion of performing computations with mechanical systems is given in [?].

In general, the physical world will be referred to as the *environment*. The device that implements a plan will be referred to as the *machine*. Practical examples of the machine include a robot, a piece of software, or even specialized hardware which may be digital or analog. As indicated in Figure 1.5, 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 which serves as 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, there is close coupling between the machine and its environment during execution.

It is even possible to draw the line between machine and environment in multiple places, which results in a *hierarchical* approach. The environment with respect to a machine, M_1 , might actually include another machine M_2 that interacts with its environment, as depicted in Figure ???. Figure ??? shows a typical hierarchy used for years in robotics. In general, any number of planning layers may be defined. For the design of planning algorithms, reference will usually only be made to a single layer. If the models are formulated correctly for each layer, and if each designed plan functions correctly, then the global hierarchy should solve tasks as desired. There are many interesting issues involving the construction of such hierarchies, but these will not be addressed in this book because they depend heavily on the particular context in which planning is used. Determining the appropriate places to draw boundaries and modularize a complicated problem is mostly the burden of the expert who applies planning techniques in a particular context.

Once the boundary has been drawn between the machine and its environment, a third component can be introduced: the *planner*. The task of the planner is to produce a *plan* based on a description of possible environments. There are two general models for plans constructed by the planner. The first case is depicted in Figure 1.8, 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 planner before execution.

¹Of course, simulating infinitely-long tape seems impossible in the physical world. Other versions of Turing machines exist in which the tape is finite, but unbounded. This may be more appropriate for the current discussion.

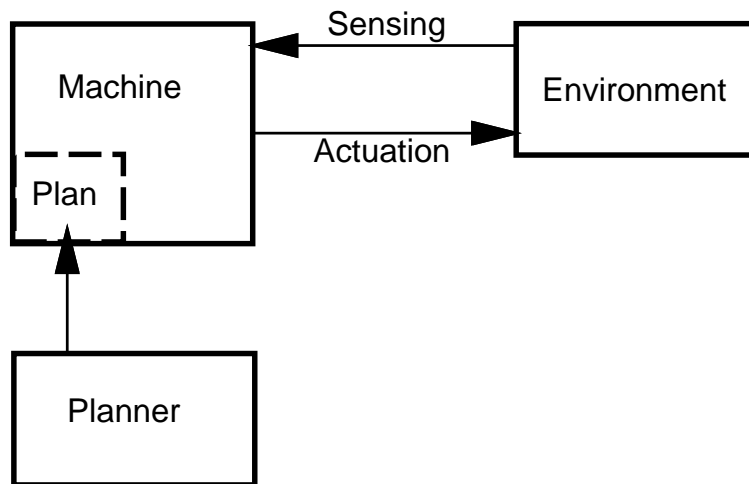


Figure 1.8: A planner produces a plan that may be executed by the machine. The planner may either be a machine itself or even a human.

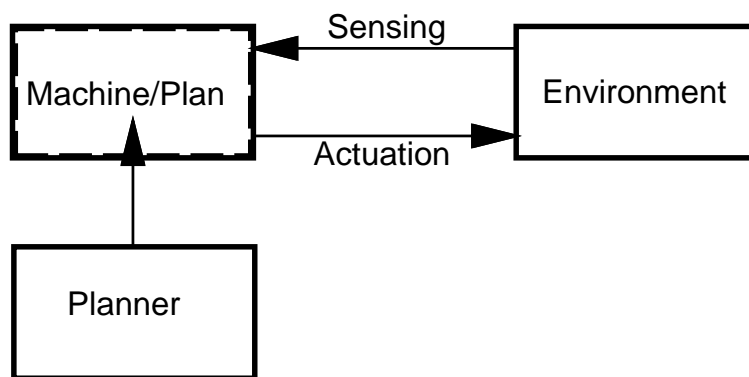


Figure 1.9: Alternatively, the planner may design the entire machine.

It will generally be assumed that once the plan is given, the machine becomes autonomous and can no longer interact with the planner.²

The second general model for plans is depicted in Figure 1.9. In this case, the plan produced by the planner encodes an entire machine. The plan can be considered as a special-purpose machine that is designed to solve the specific tasks given originally to the planner. Under this interpretation, it may be preferable to be *minimalist* and design the simplest machine possible that is sufficiently solves the desired tasks.

There are two possible ways to implement the planner. The planner may either be an algorithm in the Turing machine sense (or some related variant), or the planner may even be a human. For example, it is perfectly acceptable for a human to design a state machine that is connected to the environment. There are no additional inputs in this case because the human fulfilled the role of the traditional algorithm. The environment model is given as input to the human, and the human “computes” a plan. An example in which the planner is a traditional algorithm is given in robotics by classical motion planning. An algorithm receives a description of the environment in terms of geometric models and then computes a plan, which is a collision-free path to be followed by the robot. Whether the planner is a human or is a machine itself, the process of developing plans will be generally referred to as *planning algorithms*.

To summarize, there are three general components:

1. The *environment*, which models the physical world with which a plan must interact.
2. The *machine*, which interacts with the environment through sensing and actuation. The machine may be *programmable*, which means a plan can be downloaded, or the machine may simply be the plan itself.
3. The *planner*, which takes one of a set of environment descriptions and produces a plan. In some cases, the human designs the planner, and in others, the human *is* the planner. In both cases, these will be referred to as *planning algorithms*.

1.5 Organization of the Book

PART I: Introductory Material

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

- **Chapter 1: Introductory Material**

This includes some examples and provides a high-level overview of planning philosophy.

²Of course this model can 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 model does not prohibit the updating of plans in practice.

- **Chapter 2: Discrete Planning**

This chapter 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 2.2 and 2.3 are most important for heading into Part II. For Part III, Section 2.4 is additionally useful.

PART II: Motion Planning

The main source of inspiration for the problems and algorithms covered in this part comes from robotics. The methods, however, are general enough to apply to applications in other areas, such as computational biology, computer-aided design, and computer graphics. An alternative title that more appropriately 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 most 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 most critical 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 many applications 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 6: Combinatorial Motion Planning**

The algorithms covered in this section are sometimes called *exact algorithms*. They provide complete (i.e., they find a solution if one exists, or report failure, otherwise) solutions to motion planning problems. The sampling-based algorithms have been more useful in practice, but these are not complete in the same sense.

- **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, which is covered in Section 4.4

- **Chapter 8: Feedback Motion Strategies**

This is a transitional chapter that introduces feedback into the motion planning problem, but still does not introduce differential constraints, which is 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. The plan will be executed as planned.

PART III: Decision-Theoretic Planning

An alternative title is “Planning under Uncertainty”. Most of the part 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 concepts and concepts developed here involve making decisions in a single step, but in the face of uncertainty. Therefore, the problems generally are not considered planning, and there is no talk of state spaces. This chapter provides important background for Part III, however, because planning under uncertainty can be considered as multi-step decision making. Chapter 9 covers a single step, which is used as a building block for later planning concepts.

- **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 2.4 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: The Information Space**

The chapter defines a framework for planning when the current state is not known. Information regarding the state is obtained from sensor observations and memory of actions that were previously applied. The information space serves a similar purpose for problems with sensing uncertainty as the configuration space did for motion planning.

- **Chapter 12: Planning in the Information Space**

This chapter covers several planning problems and algorithms that involve sensing uncertainty.

PART IV: Planning under Differential Constants

This can be considered as a continuation of Part II. Now there can be both global (obstacles) and local (differential) constants 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 giving examples of differential constraints that arise in practice and explaining how to model them in the context of planning.

- **Chapter 14: Nonholonomic System Theory**


This section provides an overview of important theory developed for the control of nonlinear systems. The basic characteristic is that the dimension of the action space is less than that of the state space, which locally constraints the possible motions. This can sometimes be overcome by constructing the Control Lie Algebra (CLA) of the system.

- **Chapter 15: Planning Under Differential Constraints**

This covers both sampling-based and exact methods for planning under differential constraints. If obstacles are involved, sampling-based methods are usually required because the problems are so difficult. Nevertheless, many useful and important methods exist for planning under differential constants alone.

Chapter 2

Discrete Planning

| Chapter Status |
|--|
|  <p>What does this mean? Check http://msl.cs.uiuc.edu/planning/status.html for information on the latest version.</p> |



What does this mean? Check

<http://msl.cs.uiuc.edu/planning/status.html>

for information on the latest version.

2.1 Introduction

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 2.2 and 2.3 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 2.4 addresses the more general problem of optimal planning. The *principle of optimality* or *dynamic programming (DP) principle* [63] provides a key insight that greatly reduces the computation effort in many planning algorithms. Therefore it forms that basis of the algorithms in Section 2.4 and throughout this book. The relationship between Dijkstra's algorithm, which is widely known, and more general dynamic programming iterations is discussed. Finally, Section 2.5 briefly overviews logic-based representations of planning and methods that exploit these representations to construct plans.

Although this chapter addresses a form of planning, it may also be sometimes referred to as problem solving. Throughout the history of artificial intelligence research, the distinction between *problem solving* and *planning* has been rather elusive. For example, in a current leading textbook [665], two of the eight major parts are termed “Problem-solving” and “Planning”. The problem solving part begins by stating, “Problem solving agents decide what to do by finding sequences of actions that lead to desirable states.” ([665], 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.” ([665], p. 375). The STRIPS system is considered one of the first planning algorithms and representations [247], and its name means STanford Research Institute Problem Solver. Perhaps the term “planning” carries connotations of future time, where as “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.

Given that there are no clear distinctions between problem solving and planning, we will simply refer to both as planning. This also helps to keep with the theme of the book. Note, however, that some of the concepts apply to a broader set of problems than what is often meant by planning.

2.2 Definition of Discrete Feasible Planning

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 or not 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.

Refer to the model from Chapter 1: The planner is an algorithm that computes a sequence of actions. There is no feedback from the environment. The actions are sequenced by the machine.

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 . 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 will

be convenient to define U as the set of all possible actions over all states:

$$U = \bigcup_{x \in X} U(x). \quad (2.1)$$

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 determine whether a finite sequence of actions, when applied, transforms the world from an initial state x_I to some state in X_G . The model is summarized below:

Formulation 2.2.1 (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 , which produces a state, $f(x, u) \in X$, for every $x \in X$ and $u \in U(x)$.
4. An *initial state*, $x_I \in X$.
5. A *goal set*, $X_G \subset X$.

It is often convenient to view Formulation 2.2.1 as a directed graph $G(V, E)$, in which V and E denote the sets of *vertices* and *edges*, respectively. The set of vertices is the state space, $V = X$.¹ Let $e(x, x')$ denote a directed edge from $x \in X$ to x' . Such an edge exists in E only if there exists some $u \in U(x)$ such that $x' = f(x, u)$.

Example 2.2.1 (Moving on a 2D Grid) Suppose that a robot moves around on a grid in which each grid point has coordinates of the form (i, j) , in which i and j are both integers. The robot takes discrete steps in one of four directions (e.g., up, down, left, right), which can increment or decrement one coordinate. The motions and corresponding graph are shown in Figure 2.1, which can be imagined as stepping from tile to tile, on an infinite tile floor.

Let X be the set of all integer pairs of the form (i, j) , in which $i, j \in \mathbb{Z}$. Let $U = \{(0, 1), (0, -1), (1, 0), (0, -1)\}$. 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 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 should be easy for the reader to find a sequence of inputs that transforms the world from $(0, 0)$ to $(100, 100)$.

¹Instead, one may want to make a technical distinction between V and X and define a bijection between them because each contains a different kind of entities.

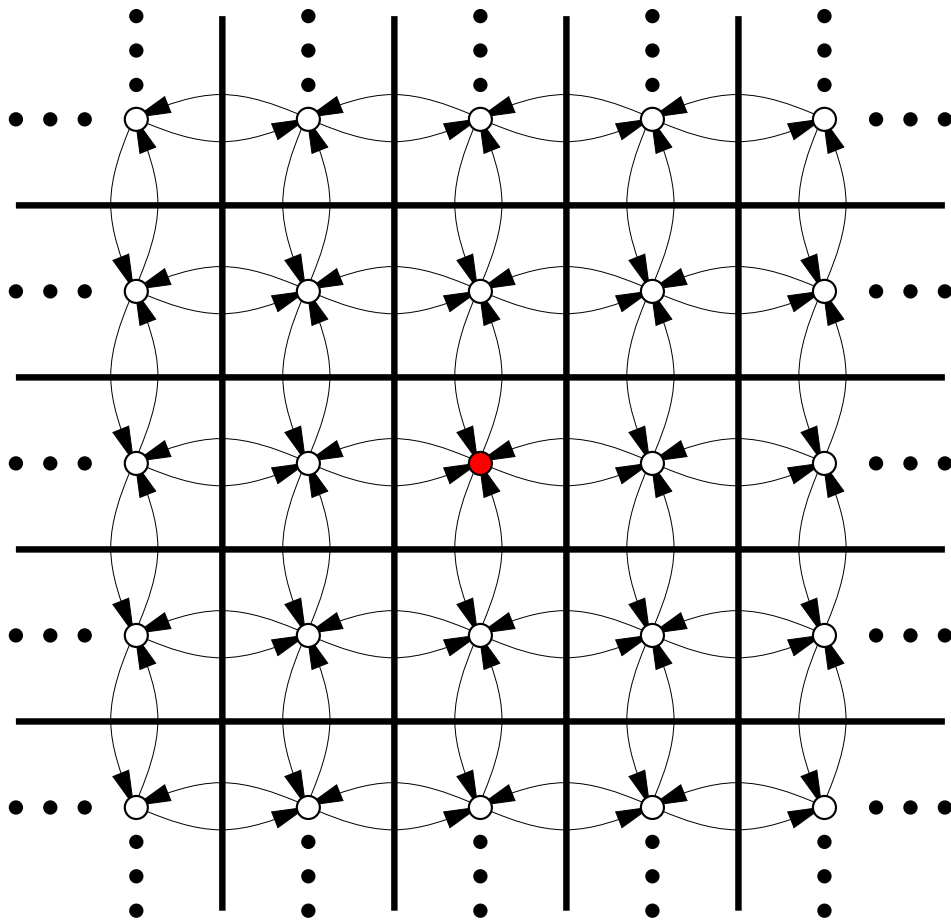


Figure 2.1: An example problem that involves walking around on an infinite tile floor.

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

Example 2.2.2 (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 a stack of 3 by 3 by 3 little cubes, which together form a larger cube as shown in Figure 2.3. Each face of the larger cube is painted one of six colors. An action may be applied to the cube by rotating a 3x3 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 (rotation of the entire cube is irrelevant). For each state there are 12 possible actions. For some

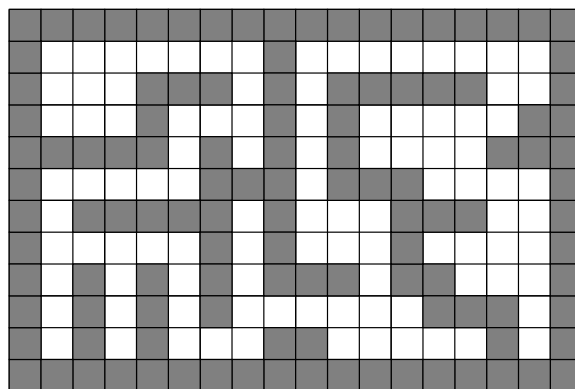


Figure 2.2: Interesting planning problems that involve exploring a labyrinth can be made by shading in tiles.

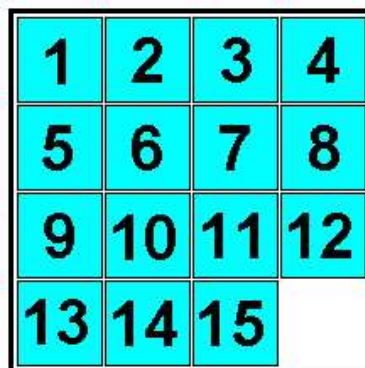
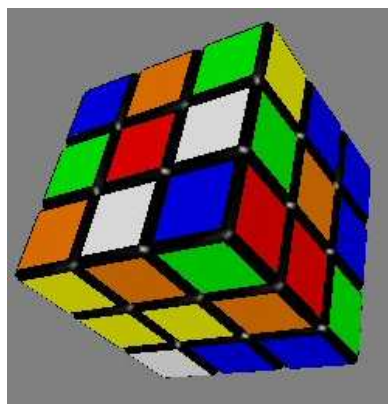


Figure 2.3: The Rubik's cube and other puzzles make nice examples of discrete planning problems.

arbitrarily chosen configuration of the Rubik's cube, the planning task is to find a sequence of actions that returns it to the configuration 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 graph G . Instead, it is revealed incrementally in the planning process. In Example 2.2.1, 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 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 2.2.1. 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 2.5. 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 2.2.1 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 representation of the machine 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 or not the associated language is empty. In terms of Unix-like constructions, this means determining whether there is some match to a regular expression.

Thus, there are several ways to represent and interpret the discrete feasible planning problem. Other important representation issues will be discussed in Section 2.5, which often to a very compact, implicit encoding of the problem. Before reaching these issues, basic planning algorithms are introduced in Section 2.3, and discrete optimal planning is covered in Section 2.4.

2.3 Searching for Feasible Plans

The methods presented in this section are just graph search algorithms, but with the understanding that the graph is revealed incrementally through the application of actions. The presentation in this section can be considered as 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

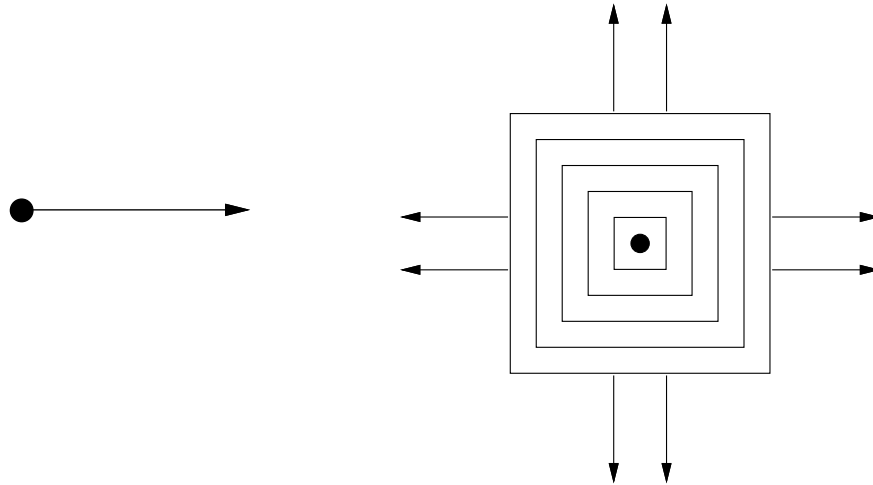


Figure 2.4: a) Many search algorithms focus too much on one direction, which may prevent them from being systematic on infinite graphs. b) If, for example, the search carefully expands in wavefronts, then it becomes systematic. The requirement to be systematic is that in the limit as the number of iterations tends to infinity, all reachable vertices are reached.

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 fine 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 2.2.1 on an infinite tile floor with no obstacles. If the search algorithm explores in only one direction, as depicted in Figure 2.4.a, 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 2.4.b, then it may be systematic. In general, 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 parts indefinitely.

2.3.1 General Forward Search

Figure 2.5 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:

```

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

```

Figure 2.5: A general template for forward search.

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 2.4.3 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 and may have next states that have not been visited. 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 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 2.5 omits several details that often become important in practice. For example, how efficient is the test 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 or not 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 simply adding another line after Line 7 which stores 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 entire plan. For convenience, one might also store which action was taken, in addition to the pointer.

Lines 8 and 9 are conceptually simple, but how can one tell whether x' has been visited? For some problems the G 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 graph, or they may not even terminate at all, even if G 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 upon 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 state visited state. Starting at x_I , simply choose the action $u \in U(x)$ that produces the lowest-cost next state, and continue the process iteratively until G is reached. The costs must have a certain monotonicity property, which is obtained by Dijkstra's algorithm

and A^* search, which will be introduced in Section 2.3.2.

2.3.2 Particular Forward Search Methods

This section presents several single-tree search algorithms, each of which is a special case of the algorithm in Figure 2.5, obtained by defining a different sorting function for Q . Most of these are just classical graph search algorithms.

Breadth First

The method given in Section 2.3.1 specifies Q as a 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. Upon 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 running time breadth first search is $O(|V| + |E|)$, in which $|V|$ and $|E|$ are the numbers of vertices and edges, respectively, in the graph representation of the planning problem. This assumes that all 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 counting as part of the algorithm complexity. The running time 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 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), aggressive exploration is the 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 finite X , but not for an infinite X because it could behave like Figure 2.4.a. 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 any action over any other in the search. Section 2.4 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 [], which is a special form of dynamic programming. More-general dynamic programming computations appear in Section 2.4 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, $L^* : X \rightarrow [0, \infty]$, called the *optimal cost-to-come* or just *cost-to-come* if it is clearly optimal from the context. For each state, x , the value $C^*(x)$ will represent the optimal² cost to reach x 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.

The cost-to-come is computed incrementally during the execution of the search algorithm in Figure 2.5. 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 $C(x') = L^*(x) + l(x, u)$). Here, $C(x')$ represents 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. Because of 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.

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 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 all dead states have their 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 lower total cost would have to travel through another state in Q , but these states already have higher cost. 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; much

²As in optimization literature, we will use $*$ to mean *optimal*.

more detailed arguments appear in Section 2.4.3 and [176]. 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 different running times will be obtained.

A-Star

The A^* (pronounced “ay star”) search algorithm is a variant of dynamic programming 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 . Although $C^*(x)$ can be computed incrementally by dynamic programming, there is no way to know the true optimal cost-to-go, G^* , in advance. However, 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 planning steps. 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 will 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 [247, 622]. As \hat{G}^* becomes closer to G^* , fewer nodes tend to be explored in comparison with dynamic programming. This would always seem advantageous, but in some problems it is not possible to find a good heuristic. 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 or not the estimate exceeds the true optimal cost-to-go, which was important for A^* . Although optimal solutions

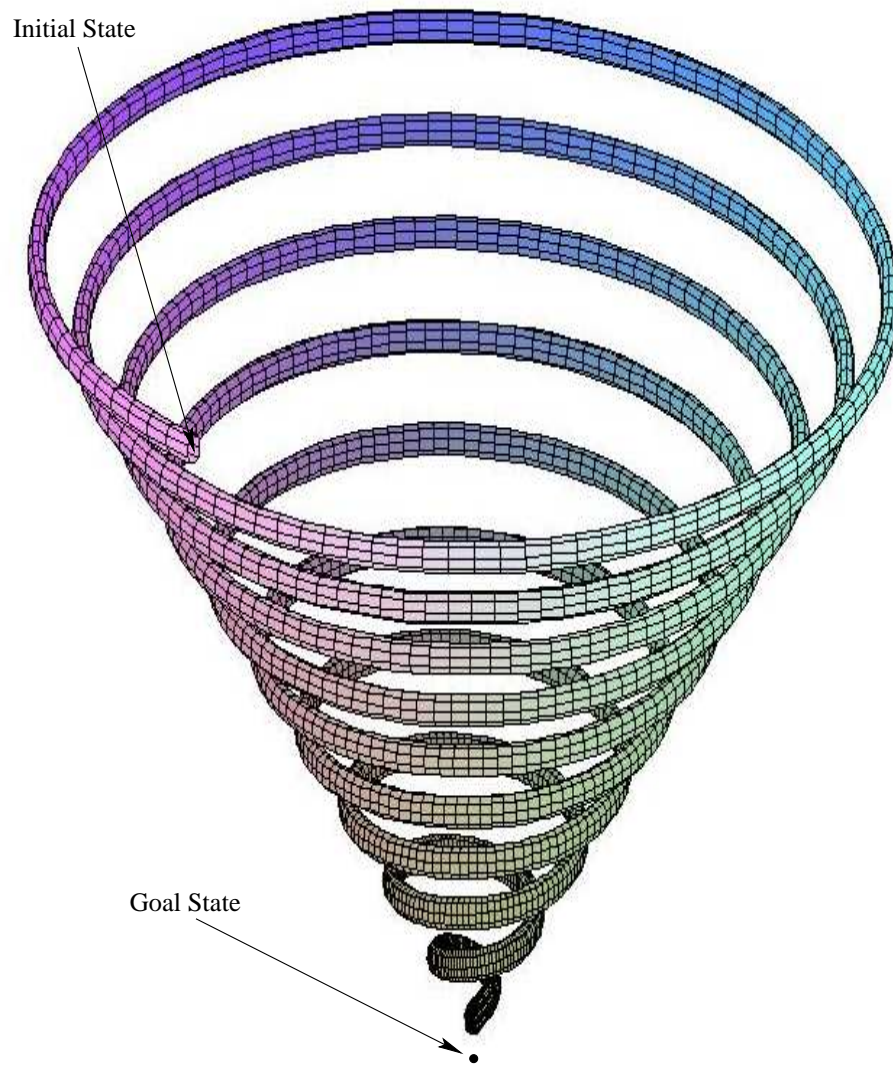


Figure 2.6: Here is bad example for best-first search. Imagine trying to reach a state that is directly below the spiral tube. If the initial state starts inside of the opening at the top of the tube, the search will progress around the spiral instead of leaving the tube and heading straight for the goal.

are not found, in many cases, far fewer nodes 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^* 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 2.6 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 when there is a large branching factor. This could occur if there are many actions per state and 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 search graph 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. 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 ten) the number reached for i . Therefore, there once the commitment has been made to reach level $i + 1$, all of the previous efforts to low relative cost. The iterative deepening method has better worst case performance than breadth-first search for many problems. If the nearest goal state is i steps from x_I , breadth-first in the worst case might reach nearly all states of distance $i + 1$. 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^* , larger and larger values of total cost are allowed [622].

2.3.3 Other General Search Schemes

This section covers two other general templates for search algorithms. The first one is simply a “backwards” version of the tree search algorithm in Figure 2.5. The second one is a bidirectional approach that grows two search trees, one from the initial state, and one from a goal state.

Backwards Search

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 backwards until the initial state is encountered. A general template for this approach is given in Figure 2.7. an action $u \in U(x)$ is applied from $x \in X$, to obtain a new state,

$x' = f(x, u)$. For backwards search a frequent computation will be to determine for some x' , what could be the preceding state, $x \in X$ and action $u \in U(x)$ such that $x' = f(x, u)$?

For most problems, it may be preferable to precompute a representation of the state transition equation, f , that is “backwards” to be consistent with the search algorithm. Some convenient notation will now be constructed for the backwards version of f . Let $U^{-1} = \{(x, u) \mid x \in X, u \in U(x)\}$, 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 *backwards action space*, defined formally for any $x' \in X$ as:

$$U^{-1}(x') = \{(x, u) \in U^{-1} \mid x' = f(x, u)\}. \quad (2.2)$$

For convenience, let u^{-1} denote a state-action pair (x, u) which 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 *backwards state transition equation* that yields x from x' and $u^{-1} \in U^{-1}(x')$. Hence, we can write $x = f^{-1}(x', u^{-1})$, 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 graph representation. Imagine reversing the direction of every edge. This will make finding a plan in the reversed graph using backwards search equivalent to finding one in the original graph using forward search. The backwards state transition equation is just the version of f that is obtained after reversing all of the edges. Each u^{-1} is just a reversed edge. Since there is a perfect symmetry with respect to the forward search of Section 2.3.1, any of the search algorithm variants from Section 2.3.2 could be adapted work under the template in Figure 2.7 once f^{-1} has been defined.

Bidirectional Search

Now that forward and backwards search have been covered, the next reasonable idea is to conduct a bidirectional search. The general search template given in Figure 2.8 can be considered as a combination of the two in Figures 2.5 and 2.7. One tree is grown from the initial state, and the other is grown from the goal state. The search terminates with success when the two trees meet. Failure occurs if both priority queues have been exhausted. For many problems bidirectional search can dramatically reduce the amount of exploration required to solve the problem. The dynamic programming and A^* variants of bidirectional search will 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 help 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.

```

BACKWARDS_SEARCH
1   $Q.Insert(x_G)$ 
2  while  $Q$  not empty do
3       $x \leftarrow Q.GetFirst()$ 
4      if  $x = x_I$ 
5          return SUCCESS
6      forall  $u \in U^{-1}(x)$ 
7           $x' \leftarrow \phi(x, u)$ 
8          if  $x'$  not visited
9              Mark  $x'$  as visited
10              $Q.Insert(x')$ 
11         else
12             Resolve duplicate  $x'$ 
13 return FAILURE

```

Figure 2.7: A general template for backwards search.

2.3.4 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 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 (the configuration space).

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

1. **Initialization:** Let the search graph, $G(V, E)$, be initialized with E empty and V containing x_I and possibly some other states. If bidirectional search is used, then initially, $V = \{x_I, x_G\}$. 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 .
2. **Select Node:** Choose a node $n_{cur} \in V$ for expansion. Let x_{cur} denote its associated state.
3. **Apply an Action:** In either a forward or backwards 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 $u \in U(x_{new})$ (backwards).
4. **Insert A Directed Edge in the Graph:** If certain algorithm-specific tests are passed, then generate an edge from x to x_{new} for the forward case, or an edge from x_{new} to x for the backwards case. If x_{new} is not yet in V , it

```

BIDIRECTIONAL_SEARCH
1   $Q_I.Insert(x_I)$ 
2   $Q_G.Insert(x_G)$ 
3  while  $Q_I$  not empty or  $Q_G$  not empty do
4      if  $Q_I$  not empty
5           $x \leftarrow Q_I.GetFirst()$ 
6          if  $x \in x_G$  or  $x \in Q_G$ 
7              return SUCCESS
8          forall  $u \in U(x)$ 
9               $x' \leftarrow f(x, u)$ 
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          $x' \leftarrow Q_G.GetFirst()$ 
17         if  $x' = x_I$  or  $x' \in Q_I$ 
18             return SUCCESS
19         forall  $u^{-1} \in U^{-1}(x')$ 
20              $x \leftarrow \phi(x', u^{-1})$ 
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 2.8: A general template for bidirectional search.

will be inserted into V .³

5. **Check for Solution:** Determine whether 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 can become 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.

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 2.5 iterates tries all actions for the first element of Q . If there are k actions, this corresponds to k iterations in the algorithm above.

2.4 Discrete Optimal Planning

This section extends Formulation 2.2.1 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 added for convenience to indicate the current plan step; 2) a cost functional will be introduced, which serves as a kind of taxi meter to determine how much cost will accumulate; 3) a termination action, 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 Section 2.4.1. The approach involves performing dynamic programming iterations 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 2.4.2. Finally, Section 2.4.3 explains the close relationship between the general DP iterations of Section 2.4 and the special case of Dijkstra's algorithm, which was covered in Section 2.3.1 as a particular search algorithm.

With nearly all optimization problems, there is the arbitrary, symmetric issue of defining the task in way that requires minimization or maximization. If the cost is a kind of energy or expense, then minimization seems sensible, as is typical in control theory. If the cost is a kind of reward, as in investing or typical AI research, then maximization is preferred. Although this issue remains throughout the book, we will choose to minimize everything. If maximization is preferred, then multiplying the costs by -1 , and maximizing wherever it says to minimize (also minimizing where it says to maximize in some later chapters), should suffice.

³In 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 model will be given shortly, but first some new notation is introduced. Let π_K denote a K -step plan, which is a sequence (u_1, u_2, \dots, u_K) of K actions. Note that if π_K and x_I are given, then a sequence of states, x_1, x_2, \dots, x_{K+1} , can be derived using the state transition equation, f . Initially, $x_1 = x_I$, and each following 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 2.2.1 is L , the cost functional.

Formulation 2.4.1 (Discrete Fixed-Length Optimal Planning)

1. All of the components from Formulation 2.2.1 are inherited directly: X , $U(x)$, f , x_I , and X_G , except here it is assumed that X is finite.
2. A number, K , of *stages*, which is the exact length of a plan (measured as the number of actions, u_1, u_2, \dots, u_K). States will also obtain a stage index: x_{k+1} denotes the state obtained after u_k is applied.
3. Let L denote a real-valued, additive cost (or loss) functional, which is applied to a K -step plan, π_K . This means that the sequence, (u_1, \dots, u_K) , of actions and the sequence, (x_1, \dots, x_{K+1}) , of states may appear in an expression of L . For convenience, let $F \equiv K + 1$, to denote the *final state* (note that the application of u_K advances the stage to $K + 1$). The *cost functional* is

$$L(\pi_K) = \sum_{k=1}^K l(x_k, u_k) + l_F(x_F). \quad (2.3)$$

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 (7.26) is actually unnecessary if it is agreed in advance that L will only be applied to evaluate plans that reach X_G . It would 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 is helpful to remember that it is just a trick to convert feasibility constraints into a straightforward optimization ($L = \infty$ means *not feasible* and $L < \infty$ means *feasible with cost L*).

Now the task is to find a plan that minimizes L . To obtain a feasible planning problem like Formulation 2.2.1, but restricted to K -step plans, let $l(x, u) \equiv 0$. To obtain a planning problem that requires minimizing the number of stages, then 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 .

2.4.1 Optimal Fixed-Length Plans

Consider computing an optimal plan under Formulation 1. 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, dynamic programming (DP) principle will help. We first say in words what will appear later in equations. The DP idea 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 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. In some cases, the approach can be adapted to the well-known Dijkstra's algorithm; however, it is important to realize that this is only a special case which applies to a narrower set of problems. The following text will describe the *general DP iterations*, and Section 2.4.3 discusses their connection to Dijkstra's algorithm.

Backwards dynamic programming

Just as for the search methods, there will be both a forward and backwards version of the approach. The backwards case will be covered first. Even though it does not appear as straightforward on the surface to progress backwards from the goal, it turns out that this case 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 $1 \leq k \leq F$, let G_k^* denote the cost that accumulates from stage k to F under the execution of the optimal plan:

$$G_k^*(x_k) = \min_{u_k, \dots, u_K} \left\{ \sum_{i=k}^K l(x_i, u_i) + l_F(x_F) \right\} \quad (2.4)$$

Inside of the min of (2.4) are the last $K - k + 1$ terms of the cost functional, (7.26). The optimal cost-to-go for the boundary condition of $k = F$ reduces to

$$G_F^*(x_F) = l_F(x_F). \quad (2.5)$$

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 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

$$G_K^*(x_K) = \min_{u_K} \{l(x_K, u_K) + l_F(x_F)\}. \quad (2.6)$$

Because $l_F = G_F^*$ and $x_F = f(x_K, u_K)$, substitutions can be made into (2.6) to obtain

$$G_K^*(x_K) = \min_{u_K} \{l(x_K, u_K) + G_F^*(f(x_K, u_K))\}, \quad (2.7)$$

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 next be shown that G_k^* can be computed similarly once G_{k+1}^* is given. Carefully study (2.4) and note that it can be written as

$$G_k^*(x_k) = \min_{u_k} \min_{u_{k+1}, \dots, u_K} \left\{ l(x_k, u_k) + \sum_{i=k+1}^K l(x_i, u_i) + l_F(x_F) \right\} \quad (2.8)$$

by pulling the first term out of the sum, and by separating the minimization over u_k from 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

$$G_k^*(x_k) = \min_{u_k} \left[l(x_k, u_k) + \min_{u_{k+1}, \dots, u_K} \left\{ \sum_{i=k+1}^K l(x_i, u_i) + l(x_F) \right\} \right]. \quad (2.9)$$

The inner min is exactly the definition of the cost-to-go function G_{k+1}^* , which yields the following recurrence:

$$G_k^*(x_k) = \min_{u_k} \{l(x_k, u_k) + G_{k+1}^*(x_{k+1})\}, \quad (2.10)$$

in which $x_{k+1} = f(x_k, u_k)$. Now that the right side of (2.10) depends only on x_k , u_k , and G_{k+1}^* , the computation of G_k^* easily proceeds in $O(|X||U|)$ time. Note that in each pass over X , some states receive an infinite value only because they are not reachable: a k -step plan from x_k to X_G does not exist. In terms of DP, this means that an action $u_k \in X(x_k)$ does not exist that brings x_k to some state $x_{k+1} \in X$ from which a $(k-1)$ -step plan exists that terminates in X_G .

Summarizing, the computations of cost-to-go functions proceeds as follows:

$$G_F^* \rightarrow G_K^* \rightarrow G_{K-1}^* \cdots G_k^* \rightarrow G_{k-1}^* \cdots G_2^* \rightarrow G_1^*, \quad (2.11)$$

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 get to the goal from x_I . It will also conveniently give the optimal cost to go for any other initial state, which may be infinity for those from which the X_G cannot be completed.

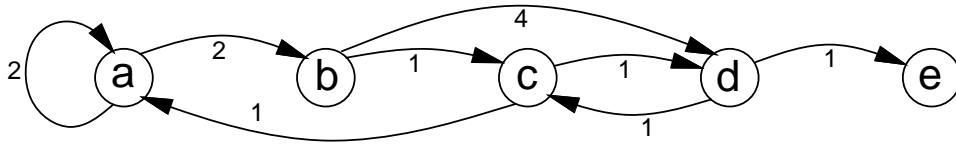


Figure 2.9: A five-state example is shown. Each vertex represents a state, and each edge represents an input that can be applied to the state transition equation to change the state. The weights on the edges represent $l(x_k, u_k)$ (x_k is the originating vertex of the edge).

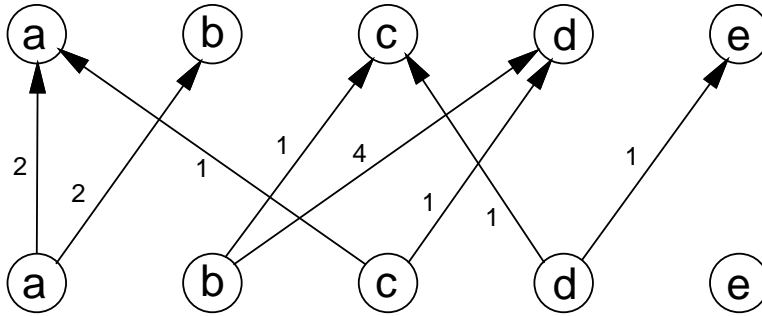


Figure 2.10: The possibilities are shown for advancing forward one stage. This is obtained by making two copies of the states from Figure 2.9, one copy for the current state, and one for the potential next state.

It seems nice that the cost of the optimal plan can be computed so easily, but how is such a plan extracted? One possibility is to store the action that satisfied the min 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 in Section 2.4.2 for the more general case of optimizing over variable-length plans.

Example 2.4.1 (A five-state optimal planning problem)

Figure 2.9 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 DP 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:

| State | 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 | ∞ |

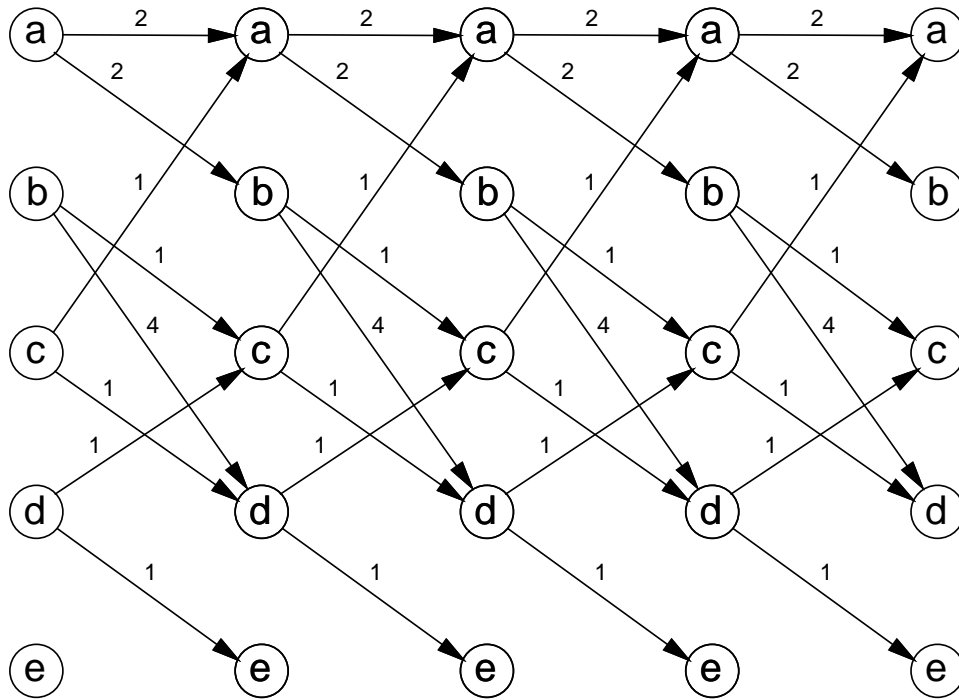


Figure 2.11: By turning Figure 2.10 sideways and copying it K times, a graph can be drawn that easily shows all ways to arrive at a final state from an initial state by flowing from left to right. The DP computations select automatically the optimal route.

Figures 2.10 and 2.11 help illustrate 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 could possibly lead to d in stage $k = 5$. Note that the minimization in (2.10) always chooses the action that produces the best total cost when arriving at a vertex in the next stage. ■

Forward dynamic programming

The ideas from Section 2.4.1 may be recycled to yield a symmetrically equivalent method that computes *cost-to-come* functions from the initial stage. Whereas backwards DP was able to find optimal plans from all initial states simultaneously, forward DP can be used to find optimal plans too all states in X . In the backwards 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 direction, 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 DP to originate

from x_I . There is the familiar choice of making notation that imposes constraints on the action spaces, or simply adding a term that forces infeasible plans to have infinite cost. Once again, we chose the latter.

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

$$C_1^*(x_1) = l_I(x_1), \quad (2.12)$$

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

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

$$C_k^*(x_k) = \min_{u_1, \dots, u_{k-1}} \left\{ l_I(x_1) + \sum_{i=1}^{k-1} l(x_i, u_i) \right\}. \quad (2.13)$$

Note that the sum refers to a sequence of states, x_1, \dots, x_{k-1} , which is the result of applying the action sequence (u_1, \dots, u_{k-1}) . 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 (2.4) it is assumed in (2.13) that $u_i \in U(x_i)$ for every $i \in \{1, \dots, k-1\}$. The resulting x_k , obtained after applying u_{k-1} must be the same x_k that is named in the argument on the right side of (2.13). 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, \dots, u_{k-1} and x_k are given.

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

$$C_K^*(x_F) = \min_{u_1, \dots, u_K} \left\{ l_I(x_1) + \sum_{i=1}^K l(x_i, u_i) \right\}. \quad (2.14)$$

This equation looks the same as (2.7), but l_I is used instead of l_F . This has the effect of filtering the plans that are considered to only those that start at x_I . The forward DP iterations will find optimal plans to any reachable final state from x_I . This behavior is complementary to that of backwards DP. In that case, X_G was fixed, and optimal plans from any initial state were found. For forward DP, this is reversed.

To express the DP 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 in state x_k by applying some action. Once again, the backwards state transition equation from Section 2.3.3 is useful. Using the stage indices, it is written here as $x_{k-1} = f^{-1}(x_k, u_k^{-1})$.

Using f^{-1} , the DP equation is:

$$C_k^*(x_k) = \min_{u^{-1} \in U^{-1}(x_k)} \{C_{k-1}^*(x_{k-1}) + l(x_{k-1}, u_{k-1})\}, \quad (2.15)$$

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 (2.15), the final cost-to-come may be iteratively computed in $O(K|X||U|)$ time, just as in the case of computing the first-stage cost-to-go in backwards dynamic programming.

Example 2.4.2 (Forward DP for the five-state problem)

Example 2.4.1 will now be revisited for the case of forward DP with fixed plan length for $K = 4$. The following cost-to-come functions are obtained by direct application of (2.15):

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

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. ■

2.4.2 The General Case

The dynamic programming techniques for fixed-length plans can be generalized nicely to the more interesting case in which plans of varying 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 2.2.1 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 and also introduces a special action, u_T :

Formulation 2.4.2 (Discrete Optimal Planning)

1. All of the components from Formulation 2.2.1 are inherited directly: X , $U(x)$, f , x_I , and X_G . Also, the notion of stages from Formulation will be used.

2. Let L denote a real-valued, additive cost (or loss) functional, which may be applied to any K -step plan, π_K , to yield

$$L(\pi_K) = \sum_{k=1}^K l(x_k, u_k) + l_F(x_{K+1}). \quad (2.16)$$

In comparison with L from Formulation 1, 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 a special *termination action*, u_T . If u_T is applied to x_k , at stage k , then the action is repeatedly applied forever, the state remains in x_k forever, 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 we would like to perform the DP iterations for $K = 5$, and there is a two-step plan, (u_1, u_2) , that 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 will reach X_G and cost the same as (u_1, u_2) . With this simple extension, the forward and backwards DP methods of Section 2.4.1 may be applied for any fixed K to optimize over all plans of length K or less (instead of fixed K).

The next step is to remove the dependency on K . Consider running backwards DP 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}^* , etc. Recall that x_I is not utilized in the backwards DP; therefore, there is no concern regarding the starting state of the plans. Suppose that backwards dynamic programming was used for $K = 16$ and was executed down to G_{-8}^* . This considers all plans of length 25 or less. Note that for convenience, 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}^* . The shifting of indices is allowed because none of the costs depend on the particular index that is given to the stage. The only important aspect of the DP computations is that they proceed backwards, and sequentially from state to stage.

Eventually, enough iterations will have 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 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 no longer depends on a particular stage k .

Are there any conditions under which backwards DP could be run forever, with each iteration producing a cost-to-go function that in 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. For every $x \in X$, there either exists a plan that reaches X_G or there does not. For each state from which there exists a plan that reaches X_G , consider the number of steps in the optimal plan. Take the maximum number of steps over such optimal plans, one from each state that can reach X_G . This serves as a limit on the number of DP iterations that are needed. 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 in the corresponding graph representation there is a cycle whose total cost is negative that it will be preferable to execute a plan that travels around the cycle forever, reducing the total cost to $-\infty$. We will assume that the cost functional is defined in a sensible way so that such 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 cycles.

Let $-K$ denote the iteration at which the cost-to-go values become stationary. At this point, a real-valued, optimal cost-to-go function, $G^* : X \rightarrow \mathbb{R}$, may be 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 . The optimal cost-to-go, G^* , can be used to recover the optimal actions, if they were not explicitly stored by the algorithm. Consider starting from some $x \in X$. What is the optimal next action? This is given by

$$\arg \min_u \{l(x, u) + G^*(f(x, u))\}, \quad (2.17)$$

which is the action, u , that minizes an expression that is very similar to (2.10). The only difference is that the stage indices are dropped because the cost-to-go values no longer depend on them. After applying u , the state transition equation is used to obtain $x' = f(x, u)$, and (2.17) may be applied again on x' . This process continues until a state in X_G is reached. This procedure is based directly on the DP equations; 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 Chapter 8.

Just as in the case of fixed-length plans, the direction of the DP iterations may be reversed to obtain a forward DP algorithm that solves the variable-length planning problem. In this case, the backwards state transition equation, f^{-1} , is used once again. Also, the initial cost term l_I instead of l_F , just as in (2.13). The forward DP algorithm can start at $k = 1$, and then it iterates until the cost-to-come become 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 for these forward DP iterations. 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, $C^* : X \rightarrow \mathbb{R}$, may

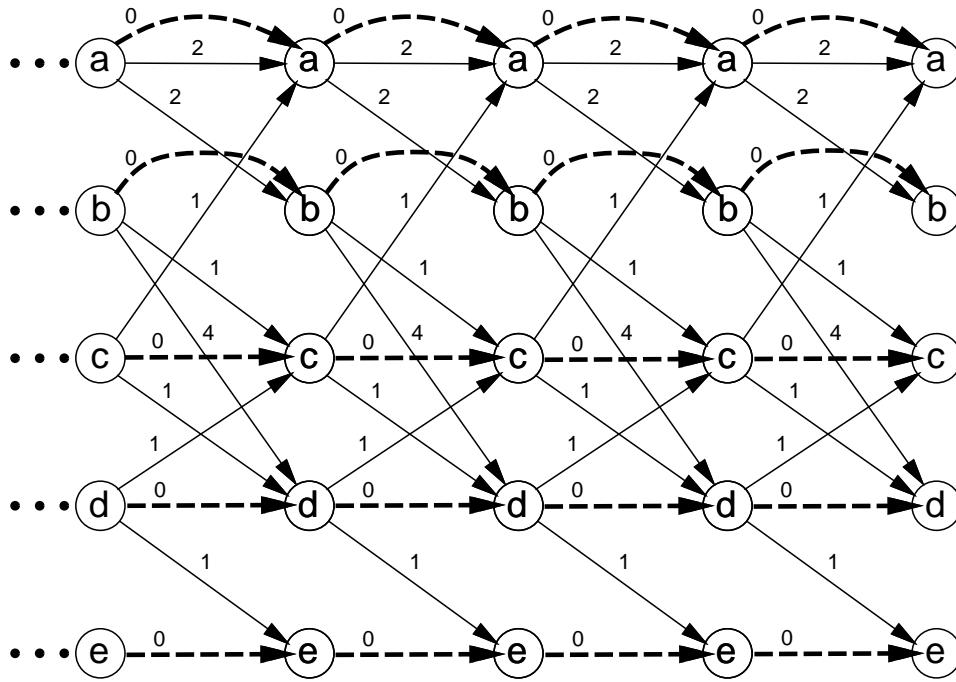


Figure 2.12: Compare this figure to Figure 2.11, for which K was fixed at 4. The effect of the termination action is depicted as dashed-line edges that yield 0 cost when traversed. This enables plans of all finite lengths to be considered. Also, the stages extend indefinitely to the left (for the case of backwards DP).

be expressed by assigning $C^* = G_F^*$, 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

$$\arg \min_{u^{-1} \in U^{-1}} \{C^*(f^{-1}(x, u^{-1})) + l(f^{-1}(x, u^{-1}), u')\}, \quad (2.18)$$

which is the forward DP counterpart of (2.17). The u' is the action in $U(f^{-1}(x, u^{-1}))$ that yields x when the state transition equation, f , is applied. The iterations proceed backwards from x_G , and terminate when x_I is reached.

Example 2.4.3 (DP iterations for variable-length plans)

Once again, Example 2.4.1 is revisited; however, this time the plan length is not fixed thanks to the termination action. Its effect is depicted in Figure 2.12 by the superposition of new edges that have zero cost. It might appear at first there is no incentive to choose other actions, but remember that any plan that does not terminate in state $x_G = d$ will receive infinite cost.

| State | 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 | ∞ |

After a few backwards DP iterations, the cost-to-go values become stationary. After this point, the termination action is being applied from all reachable states and no further loss 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 (2.17) 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 DP, suppose that $x_I = b$. The following cost-to-come functions are obtained:

| State | 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 |

For any finite value that remains content from one iteration to the next, the termination action was applied. Note that the last DP iteration is useless in this example. Once $L_{1,3}^*$ is computed, the optimal cost-to-come to every possible state from x_I is determined, and future cost-to-come functions will look identical. Therefore, the final cost-to-come is renamed to C^* . ■

2.4.3 Dijkstra Revisited

So far two different kinds of dynamic programming have been covered. The methods of Section 2.4.2 involve repeated computations over the entire state space. Dijkstra's algorithm from Section 2.3.2 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 dynamic programming computations, as in Example 2.4.3, 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 been finalized are *dead*. For the remaining states, an initial cost is obtained, but this cost may be lowered multiple 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 the general DP iterations of this section, 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 backwards DP iterations in Example 2.4.3 for comparison. In a sense, Dijkstra’s algorithm is very much like the general DP iterations, except that it efficiently maintains the set of states within which cost-to-go values 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 DP iterations. At the end of both algorithms, the resulting values should correspond to the stationary, optimal cost-to-come, C^* .

At the end of both algorithms, the resulting values should correspond to the stationary, optimal cost-to-come, C^* .

If Dijkstra’s algorithm seems so clever, then why have we spent time covering the general DP algorithm? For some problems it may become too expensive to maintain the sorted queue, and the DP iterations could provide a more efficient alternative. A more important reason is that the general DP iterations apply to a much broader class of problems by simple extensions of the method. Examples to which that apply include optimal planning over continuous state spaces (Section ??), stochastic optimal planning (Section ??), and computing dynamic game equilibria (Section ??). In some cases, it 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 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 2.5. Figure 2.13 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^* because they only need to visit each state once. Thus, the algorithms in Figures 2.5 and 2.13 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.

```

FORWARD_LABEL_CORRECTING( $x_G$ )
1  Set  $G(x) = \infty$  for all  $x \neq x_I$ , and set  $G(x_I) = 0$ 
2   $Q.Insert(x_I)$ 
3  while  $Q$  not empty do
4       $x \leftarrow Q.GetFirst()$ 
5      forall  $u \in U(x)$ 
6           $x' \leftarrow f(x, u)$ 
7          if  $G(x) + l(x, u) < \min\{G(x'), G(x_G)\}$  then
8               $G(x') \leftarrow G(x) + l(x, u)$ 
9              if  $x' \neq x_G$  then
10                  $Q.Insert(x')$ 

```

Figure 2.13: 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 2.5.

Another important difference is that the algorithm uses the cost at the goal state to prune away many candidate paths, which 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 $C(x_G)$ because 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 (2.18).

2.5 Logic-Based Representations of Planning

For many discrete planning problems that we would hope a computer can 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 the planning algorithms covered in 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 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 “0101010101...” may compress very nicely, while 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, while 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 or 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 to enable concepts such as modeling uncertainty, unpredictability, sensing errors, and game theory 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, or planning with 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 will give you a powerful understanding of planning issues across all of these areas.

2.5.1 A STRIPS-Like Representation

STRIPS-like representations have been the most common logic-based representation for discrete planning problems. This refers to the STRIPS system, which is considered one of the first planning algorithms and representations [247]; its name means 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 use only propositional logic [583], which is similar to the form introduced in this section. There are many variations of STRIPS-like representations, one of which is presented here.

The following model is given, followed by a detailed explanation.

Formulation 2.5.1 (STRIPS-Like Planning)

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

2. A nonempty set, P , of *predicates*, which are binary-valued (partial) functions of one or more instances. Each application of a predicate to a specific set of instances is called a *positive literal* if the predicate is TRUE or a *negative literal* if it is FALSE .
3. A nonempty set, O , of *operators*, each of which has: 1) *preconditions*, which is a set of positive and negative literals that must hold for the operator to apply, and 2) *effects*, which is a set of positive and negative literals that are the result of applying the operator.
4. An *initial set*, S , which is expressed as a set of *positive literals*. All literals not appearing in S are assumed to be negative.
5. A *goal set*, G , which is expressed as a set of both *positive* and *negative literals*.

Formulation 2.5.1 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, etc. 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 $Under(Dirt, Rug)$. When a predicate is shown with instances, such as $Under(Dirt, Rug)$, then it is called a *literal*, which must either have the value TRUE or FALSE . If it is TRUE , it is called a *positive literal*; otherwise, it is called a *negative literal*. 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.

The role of an operator is to change the world. To be applicable, a set of *preconditions* that must all be satisfied. Each element of this set is a literal along with required a TRUE or FALSE value for the operator to be applicable. Any literals that can be formed from the predicates, but are not mentioned in the preconditions, may assume any value for applicability of the operator. If the operator is applied, then the world is updated in a manner precisely specified by the set of *effects*. This set of literals indicates positive and negative literals that will result from the application of the operator. All other literals that could be constructed will retain their values if they do not appear in the effects.

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. 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 satisfied. For each operator, the preconditions must also be satisfied before it can be applied.

The following example illustrates Formulation 2.5.1.

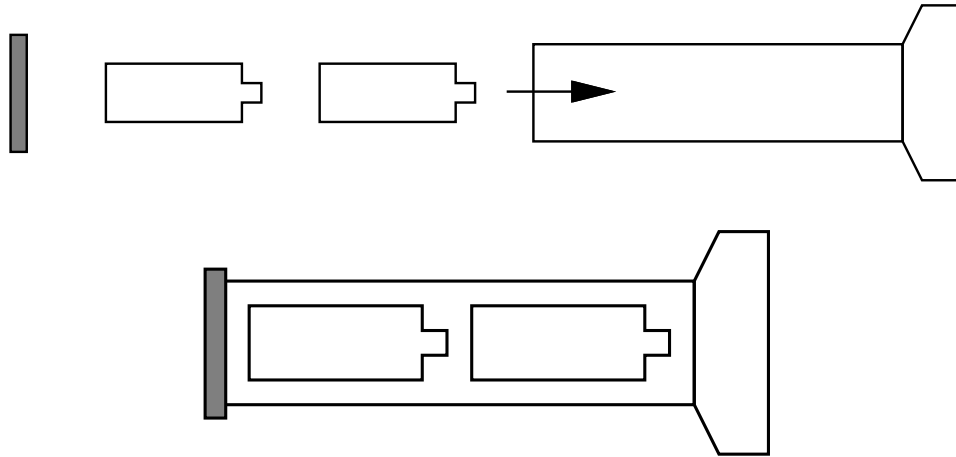


Figure 2.14: An example that involves putting batteries into a flashlight.

Example 2.5.1 Imagine a planning problem that involves putting two batteries into a flashlight, as shown in Figure 2.14. The set of instances are

$$I = \{Battery1, Battery2, Cap, Flashlight\}. \quad (2.19)$$

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 or not either battery is in the flashlight. Recall that predicates are only partial functions in general. For predicate *In* it is not desirable to apply any instance to any argument. For example, $In(Battery1, Battery1)$, and $In(Flashlight, Battery2)$ are senseless to maintain (they could be included in the model, always retaining a negative value, but it is inefficient).

The initial set is

$$S = \{On(Cap, Flashlight), \neg In(Battery1, Flashlight), \neg In(Battery2, Flashlight)\}, \quad (2.20)$$

which means that the first literal is positive, and the remaining two are negative, as indicated by the preceding \neg symbol (the cap is on the flashlight, but the batteries are outside). The goal state is

$$G = \{On(Cap, Flashlight), In(Battery1, Flashlight), In(Battery2, Flashlight)\}. \quad (2.21)$$

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

| Name | Preconditions | Effects |
|------------------|--|--------------------------------|
| <i>PlaceCap</i> | $\{\neg On(Cap, Flashlight)\}$ | $\{On(Cap, Flashlight)\}$ |
| <i>RemoveCap</i> | $\{On(Cap, Flashlight)\}$ | $\{\neg On(Cap, Flashlight)\}$ |
| <i>Insert(i)</i> | $\{\neg On(Cap, Flashlight), \neg In(i, Flashlight)\}$ | $\{In(i, Flashlight)\}$ |
| <i>Remove(i)</i> | $\{\neg On(Cap, Flashlight), In(i, Flashlight)\}$ | $\{\neg In(i, Flashlight)\}$ |

Table 2.1: Four operators for the flashlight problem. Note that an operator can be expressed with variable argument(s) for which different instances could be substituted.

The set O consists of the four operators, which are shown in Figure 2.1. Here is a plan that reaches the goal state in the smallest number of steps:

$$(RemoveCap, Insert(Battery1), Insert(Battery2), PlaceCap) \quad (2.22)$$

In plain english, it 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 becomes overwhelming. ■

2.5.2 Converting to the State Space Representation

It is useful to characterize the relationship between Model 2.5.1 and the original formulation discrete feasible planning, Formulation 2.2.1. One benefit is that it will immediately indicate how the search methods of Section 2.3 can be adapted 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 only been vaguely mentioned in the context of the STRIPS-like representation. Now consider making this more concrete. Suppose that every predicate has k arguments, and in each argument any instance could appear. This means that there are $|P| |I|^k$ different literals at any given time, which corresponds to all ways to substitute instances into all arguments of all predicates. Each literal may be either TRUE or FALSE. The complete set of literals may be encoded as a binary string by imposing a linear ordering on the instances and predicates. The state of the world is then specified in order. Using Example 2.5.1, this might appear like:

$$(On(Cap1, Flashlight1), \neg On(Cap2, Flashlight1), \dots, In(Battery7, Flashlight41), \dots). \quad (2.23)$$

Using the binary string, each element can be “0” to denote FALSE, or “1” to denote TRUE. The resulting state would be $x = 10 \cdots 1 \cdots$, for the example above. The length of the string is thus $|P||I|^k$. The total number of possible states of the world that could possibly be distinguished corresponds to the set of all possible bit strings, which is of size

$$2^{|P||I|^k}. \quad (2.24)$$

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 2.3 may appear efficient with respect to 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 heuristics to help guide the search more efficiently by exploiting the structure of specific representations.

The next step in converging 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 goal 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.

The next step is to convert the operators. For each state, $x \in X$, the set $U(x)$ will represent the set of operators with preconditions that are satisfied by x . To apply the search techniques of Section 2.3, note that it is not necessary to determine $U(x)$ explicitly in advance for all $x \in X$. Instead, it can be computed whenever each x is encountered for the first time in the search. The effect of the operator is 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 2.2.1 have been derived from the components of Formulation 2.5.1. Adapting the search techniques of Section 2.3 is straightforward. It is also straightforward to extend Formulation 2.5.1 to represent optimal planning. A cost can be associated with each operator and set of literals that capture the current state. This will express $l(x, u)$ of the cost functional, L , from Section 2.4. Thus, it is also possible to adapt the DP iterations to work under the logic-based representation, yielding optimal plans.

2.5.3 Logic-Based Planning

Need to give a brief survey of heuristic planning methods that work directly with the logic-based representation.

Literature

(This will get filled in a little more later. Here are some references for now.)

- Introduction of DP [63, 64]
- Graph search algorithms [176]
- Logic representations [247, 583]
- AI search [409, 622, 634]
- Discrete-time optimal control [19, 70, 67]
- Recent survey on AI planning (which they rename to *automated planning*, which expands considerably the subject of Section 2.5. This is an excellent source of material which is also planning, but is complementary to this book in many ways. [274]
- More coverage of labeling algorithms [67]

Exercises

(Exercises in italics are not yet fully specified)

1. *A simple example to simulate the algorithms. Verify that forward DP iterations and Dijkstra get the same result.*
2. *Try implementing and experimenting with some search variants.*
3. 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 labyrinth problems, based on Example 2.2.1. Compare the performance for two different cases:
 - (a) Using $|i' - i| + |j' - j|$ as the heuristic, as suggested in Section 2.3.2.
 - (b) Using $\sqrt{|i' - i|^2 + |j' - j|^2}$ as the heuristic.

Which heuristic seems superior? Explain your answer.

4. *Design some kind of multiresolution expanding search algorithm for the infinite tile floor.*
5. *Play with randomization on the grid problem.*
6. Try to construct a worst-case example for best-first search that has properties similar to that shown in Figure 2.6, but instead involves moving in a 2D world with obstacles, as introduced in Example 2.2.1.

7. It turns out that the general DP iterations can be generalized to a loss functional of the form

$$L = \sum_{k=1}^K l(x_k, u_k, x_{k+1}) + l_F(x_F), \quad (2.25)$$

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

- (a) Show that the dynamic programming principle can be applied in this more general settings to obtain forward and backwards DP iterations 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.
8. The cost functional can be generalized to become *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 DP algorithms of Section 2.4.1 to work for this case, and show that they give optimal solutions. Each term of the more-general cost-functional should be denoted as $l(x_k, u_k, k)$.
9. Recall from Section 2.4.2 the method of defining a termination action, u_T to make the DP 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 DP iterations will correctly compute shortest plans.
10. Dijkstra's algorithm was presented as a kind of forward search in Section 2.3.1.
- (a) Derive a backwards 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 backwards DP iterations from 2.4.2.
 - (a) Derive a backwards version of the A^* algorithm and show that it yields optimal plans.
11. Reformulate the general forward search algorithm of Section 2.3.1 so that it is expressed in terms of the STRIPS-like representation. Carefully consider what needs to be explicitly constructed by the algorithm and what is considered only implicitly.
12. *Experiment with the original STRIPS heuristic.*

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 may be considered for this part: 1) *motion planning*, and 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 may 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 with obstacles. An earlier name for motion planning is the *Piano Movers' Problem*, which brings to mind the image of trying to move a grand piano through narrow passages in a house. Have you ever been involved in an argument about how to move a sofa up some stairs? Motion planning tries to resolve such debates.

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

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 [504, 437], 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 geometric model. Using the configuration space, motion planning will be viewed as a kind of search in an implicitly-represented, high-dimensional state space. 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.

Continuous \rightarrow discrete


A central theme throughout motion planning is to transform the continuous model into a discrete one. Because of this transformation, many algorithms from Chapter 2 are embedded in motion planning algorithms. There are two alternatives to achieving this, which are covered in Chapters 6 and 5, 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 is often replaced with a weaker notion, such as *resolution completeness* or *probabilistic completeness*. It is important to study both Chapters 6 and 5 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 time and implementation difficulty of these algorithms make them prohibitive. 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 has 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 execution, regardless of what configuration is entered. This allows the true configuration to drift away from the commanded configuration. In later chapters, 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

| Chapter Status |
|---|
|  <p>What does this mean? Check http://msl.cs.uiuc.edu/planning/status.html for information on the latest version.</p> |



This chapter provides important background material that will be needed for Part II. Formulating and solving motion planning problems requires 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 only in Chapter 6, 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 the bodies are 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 task 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*: Geometric bodies that 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. Another example, the robot could be a virtual human in a graphical simulation that involves obstacles (imagine the family of Doom-like adventure games).

This section presents a method of 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 will have great expressive power and be 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 Sections 3.1.1 and 3.1.2, 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. 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 $x_1, x_2 \in X$, all points that can be expressed in the form $\lambda x_1 + (1 - \lambda)x_2$ (linear interpolation), for some scalar $\lambda \in (0, 1)$, must also lie 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 pair of vertices. The polygon can be specified by a sequence, $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$, 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 such 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} :

$$H_i = \{(x, y) \in \mathcal{W} \mid f_i(x, y) \leq 0\}. \quad (3.1)$$

Above, H_i is a primitive that describes the set of all points on one side of the line $f_i(x, y) = 0$ (including the points on the line).

A convex, m -sided, polygonal obstacle region, \mathcal{O} , is expressed as

$$\mathcal{O} = H_1 \cap H_2 \cap \dots \cap H_m. \quad (3.2)$$

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

$$\mathcal{O} = \mathcal{O}_1 \cup \mathcal{O}_2 \cup \dots \cup \mathcal{O}_n, \quad (3.3)$$

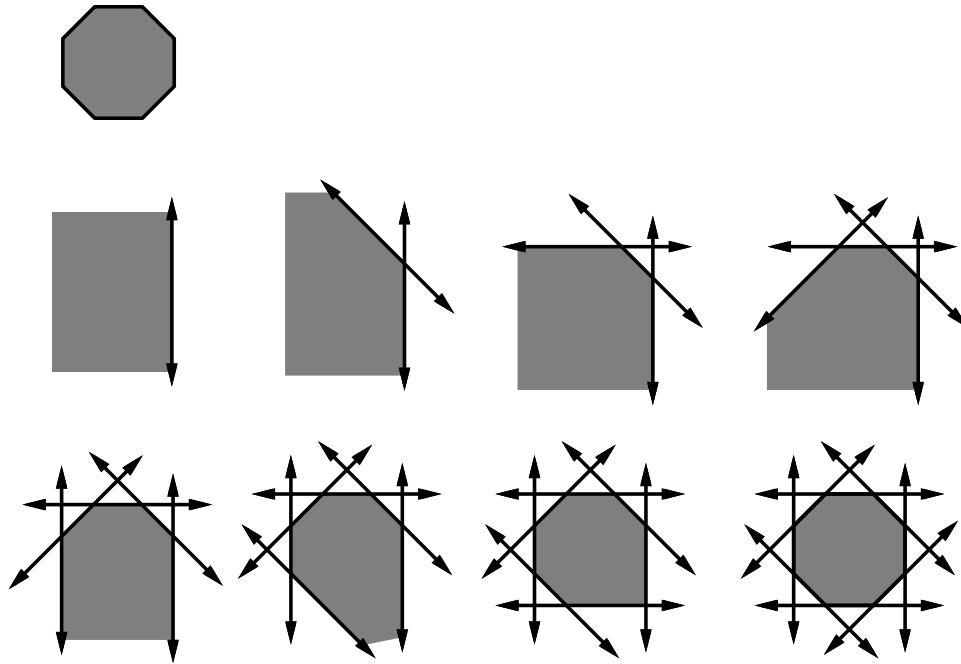


Figure 3.1: A convex polygonal region can be identified by the intersection of half-planes.

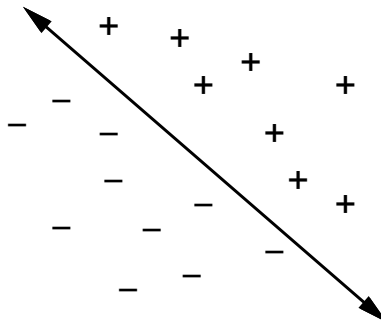


Figure 3.2: The sign of the $f(x, y)$ partitions \mathbb{R}^2 into three regions: two half planes given by $f(x, y) < 0$ and $f(x, y) > 0$, and the line $f(x, y) = 0$.

in which each \mathcal{O}_i is a convex, polygonal set that is expressed in terms of half spaces using (3.2). 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 (3.2) and (3.3). A set difference can be avoided by redefining the primitive. Suppose the model requires removing a set defined by a primitive H_i , that contains¹ $f_i(x, y) < 0$. 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.

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 the literature remarks at the end of this chapter. 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 2.5.1 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 can be represented by a logical conjunction,

$$\alpha(x, y) = e_1(x, y) \wedge e_2(x, y) \wedge \cdots \wedge e_m(x, y). \quad (3.4)$$

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

¹In 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 .

polygons can be represented by a logical disjunction of conjuncts:

$$\phi(x, y) = \alpha_1(x, y) \vee \alpha_2(x, y) \vee \cdots \vee \alpha_n(x, y) \quad (3.5)$$

Although more efficient methods exist, the predicate $\phi(x, y)$ can be used to check whether a point (x_t, y_t) lies inside of \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 OR's and AND's. 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.

Several data structures have been proposed that allow one to conveniently “walk” around the polyhedral features. For example, the *doubly-connected edge list* [189] data structure contains three types of records: faces, half edges, and vertices. 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 polyhedron. Each half-edge is shown as an arrow in Figure 3.3.b. 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, 5) the previous element in the circular list of edges that bound the face. One all of these records have been defined, one can conveniently traverse the structure of the polyhedron.

Next 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 $ax + by + cz + d = 0$, in which $a, b, c, d \in \mathbb{R}$ are constants.

Once again, the function, f can be constructed, except this time $f : \mathbb{R}^3 \rightarrow \mathbb{R}$, and $f(x, y, z) = ax + by + cz + d$. Let a *half space*, H_i , for $1 \leq i \leq m$, for all m

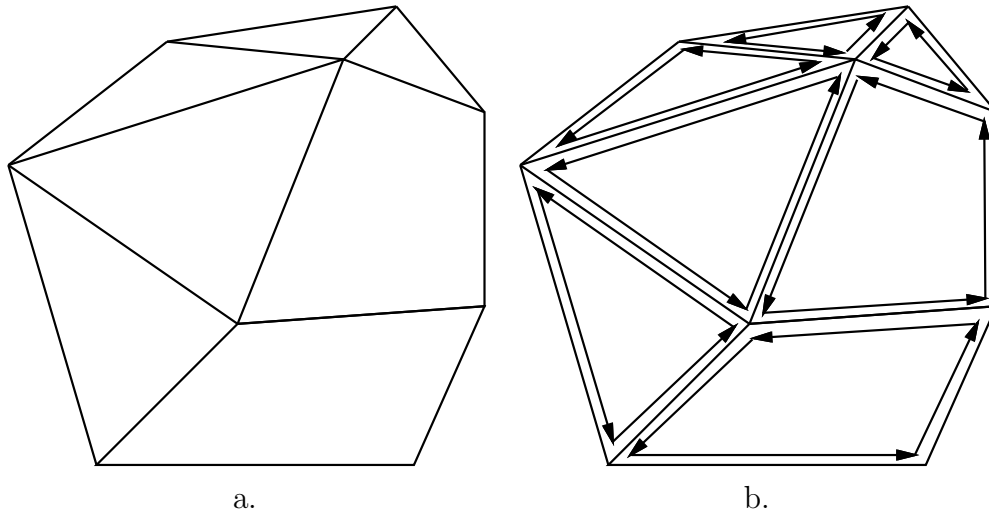


Figure 3.3: a) A polyhedron can be described in terms of faces, edges, and vertices. b) The edges of each face can be stored in a circular list that is traversed in counterclockwise order with respect to the outward normal vector of the face.

faces of \mathcal{O} , be defined as a subset of \mathcal{W} :

$$H_i = \{(x, y, z) \in \mathcal{W} \mid f_i(x, y, z) \leq 0\}. \quad (3.6)$$

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 3.3.b shows the edge ordering for each face. Note that the boundary of each face can be traversed in counterclockwise order. 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 $v = v_{12} \times v_{23}$ will always yield 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 these 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.

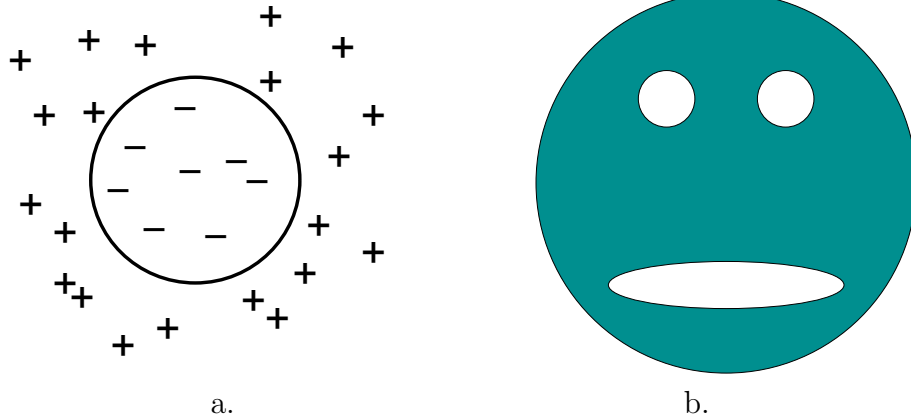


Figure 3.4: a) Once again, f is used to partition \mathbb{R}^2 into two regions. In this case, the algebraic primitive represents a disc-shaped region. b) The shaded “face” can be exactly modeled using only four algebraic primitives.

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 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

$$H = \{(x, y) \in \mathcal{W} \mid f(x, y) \leq 0\}. \quad (3.7)$$

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 3.4.a.

Example 3.1.1 (Gingerbread face) Consider constructing a model of the shaded region shown in Figure 3.4.b. Let the center of the outer circle have radius r_1 and 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 $f_1 = x^2 + y^2 - r_1^2$, $f_2 = -[(x - x_2)^2 + (y - y_2)^2 - r_2^2]$, $f_3 = -[(x - x_3)^2 + (y - y_3)^2 - r_3^2]$, and $f_4 = -[x^2/a^2 + (y - y_4)^2/b^2 - 1]$. 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} , can be represented as

$$\mathcal{O} = H_1 \cap H_2 \cap H_3 \cap H_4. \quad (3.8)$$

■

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.

For semi-algebraic models, 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 because it does not depend on the particular 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 ideas generalize easily for the case of a 3D world, obtaining algebraic primitives of the form

$$H = \{(x, y, z) \in \mathcal{W} \mid f(x, y, z) \leq 0\}, \quad (3.9)$$

which be used to define a solid representation of a 3D obstacle, \mathcal{O} , and also may be used to construct the predicate $\phi(x, y, z)$.

Equations 3.7 and 3.9 are sufficient to express any model of interest. One may define many other primitives based on different relations, such as $f(x, y) \geq 0$, $f(x, y) = 0$, $f(x, y) < 0$, $f(x, y) = 0$, and $f(x, y) \neq 0$; 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 following primitive

$$H = \{(x, y, z) \in \mathcal{W} \mid f(x, y, z) \geq 0\}. \quad (3.10)$$

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

$$H = \{(x, y, z) \in \mathcal{W} \mid f(x, y, z) = 0\}. \quad (3.11)$$

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

$$H_1 = \{(x, y, z) \in \mathcal{W} \mid f(x, y, z) \leq 0\} \quad (3.12)$$

and

$$H_2 = \{(x, y, z) \in \mathcal{W} \mid -f(x, y, z) \leq 0\}. \quad (3.13)$$

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.

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

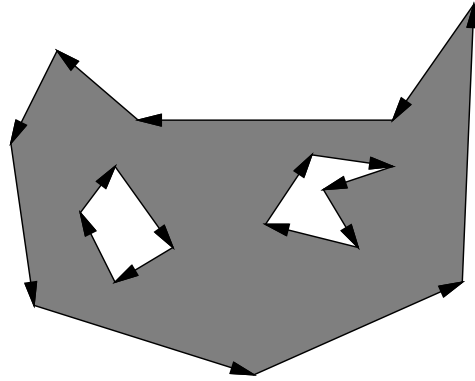


Figure 3.5: A polygon with holes can be expressed by using different orientations: counterclockwise for the outer boundary and clockwise for the hole boundaries. Note that the shaded part is always to the left when following the arrows.

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 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 the problem of 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 about the representation).

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 counterclockwise. Each polygon of m vertices may be encoded by a list of the form $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$. It is assumed that there is an edge between each (x_i, y_i) and (x_{i+1}, y_{i+1}) , and also 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 expressed in the clockwise direction. To the left of each edge will be the interior of the outer polygon, and the 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



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

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 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 in hardware has mainly been developed in terms of 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 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 *triangle strips*, which is a sequence of triangles such that each adjacent pair share a common edge, and *triangle fans*, which is 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 two-dimensional formulation is given here.

A curve can be expressed as

$$C(u) = \frac{\sum_{i=0}^n w_i P_i N_{i,k}(u)}{\sum_{i=0}^n w_i N_{i,k}(u)}, \quad (3.14)$$

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

$$N_{i,k}(u) = \frac{u - t_i}{t_{i+k} - t_i} N_{i,k-1}(u) + \frac{t_{i+k+1} - u}{t_{i+k+1} - t_{i+1}} N_{i+1,k-1}(u). \quad (3.15)$$

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, $\{t_0, t_1, \dots, t_m\}$, that controls 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 will look very similar to Example 2.2.1. 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 some part 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 in environment with its sensors. One generalization of bitmaps is a *grey-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 case 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 type of model is a *superquadric*, which generalizes quadric surfaces. One example is a superellipsoid, given for $\mathcal{W} = \mathbb{R}^3$ by

$$\left\{ \left| \frac{x}{a} \right|^{n_1} + \left| \frac{y}{b} \right|^{n_2} \right\}^{\frac{n_1}{n_2}} + \left| \frac{z}{c} \right|^{n_1} - 1 \leq 0, \quad (3.16)$$

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 $r(s)$ 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 the robot, \mathcal{A} , is defined as a subset of \mathbb{R}^2 or \mathbb{R}^3 . A *rigid body transformation* is a function, $h : \mathcal{A} \rightarrow \mathcal{W}$, 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

$$h(\mathcal{A}) = \{h(a) \in \mathbb{R}^2 \mid a \in \mathcal{A}\}, \quad (3.17)$$

which is the image of h , indicating all points in \mathcal{W} occupied by the transformed robot.

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 $h(a) \in \mathcal{W}$, and the entire model has easily transformed. However, be careful when the model is expressed with primitives, such as

$$H_i = \{a \in \mathbb{R}^2 \mid f_i(a) \leq 0\}, \quad (3.18)$$

which differs slightly from (3.1) because the robot is not directly defined in \mathcal{W} , and also a is used to denote a point $(x, y) \in \mathcal{A}$. Under a transformation h , the half plane in \mathcal{W} may be represented as

$$h(H_i) = \{h(a) \in \mathcal{W} \mid f_i(a) \leq 0\}. \quad (3.19)$$

To transform the primitive completely, however, it is better to directly name points in $w \in \mathcal{W}$, as opposed to $h(a) \in \mathcal{W}$. This becomes

$$h(H_i) = \{w \in \mathcal{W} \mid f_i(h^{-1}(w)) \leq 0\}, \quad (3.20)$$

in which the inverse of h appears in the right side because the original point $a \in \mathcal{A}$ needs to be recovered to evaluate f_i .

Thus, sometimes the forward transformation is needed, and at other times the inverse is needed. Be careful! Specific samples will be given shortly that clearly illustrate this.

The coming sections will introduce 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 new parameter q , 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 shorted 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 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 forward or inverse versions of the transformations need to be used.

One final comment before starting: note that \mathcal{A} , before it is transformed, is also a subset of \mathcal{W} . It was written only as a subset of \mathbb{R}^2 or \mathbb{R}^3 to avoid confusion in the discussion above. Another way to make the distinction clear is to borrow from mechanics [], and give the robot a separate coordinate *frame* from the world. Thus, the robot is defined in an *object frame*, and the world is defined in a *reference frame*. A transformation indicates where the object frame appears with respect to the reference frame. When multiple bodies are covered in Section 3.3, each body will have its own object frame, and all bodies will be expressed with respect to the reference frame.

3.2.2 2D Transformations

Translation The robot \mathcal{A} will be *translated* by using two parameters, $x_t, y_t \in \mathbb{R}$. From Section 3.2.1, this means that $q = (x_t, y_t)$. The function h is defined as $h(x, y) = (x + x_t, y + y_t)$. A boundary representation of \mathcal{A} can be translated by transforming each vertex in the sequence of polygon vertices. Each point (x_i, y_i) in the sequence is simply 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

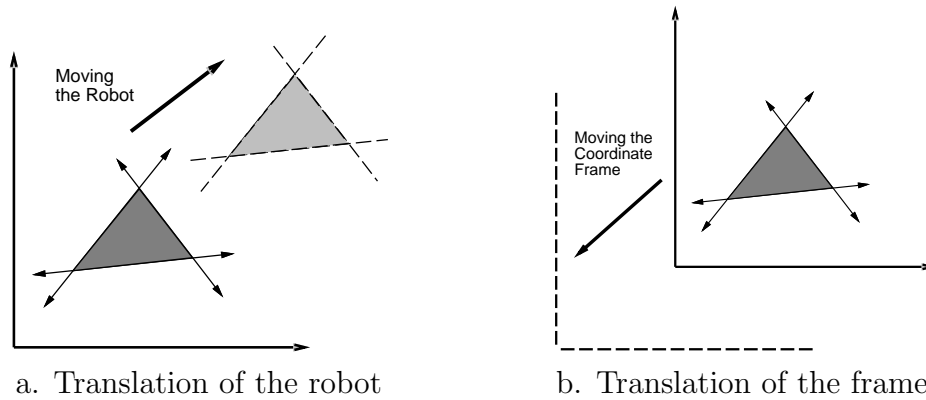
$$H_i = \{(x, y) \in \mathbb{R}^2 \mid f(x, y) \leq 0\} \quad (3.21)$$

is transformed to

$$h(H_i) = \{(x, y) \in \mathcal{W} \mid f(x - x_t, y - y_t) \leq 0\}. \quad (3.22)$$

For example, suppose the robot is a disc of unit radius, centered at the origin. It is modeled by a single primitive,

$$\mathcal{A} = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 - 1 \leq 0\}. \quad (3.23)$$



a. Translation of the robot

b. Translation of the frame

Figure 3.7: For every transformation there are two interpretations.

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

$$h(\mathcal{A}) = \{(x, y) \in \mathcal{W} \mid (x - x_t)^2 + (y - y_t)^2 - 1 \leq 0\}, \quad (3.24)$$

which is the familiar equation for a disc centered at (x_t, y_t) . In this example, the inverse, h^{-1} was 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 can be selected to completely characterize the robot in the world. If the set of allowable values for x_t and y_t form a two-dimensional subset of \mathbb{R}^2 , then the degrees of freedom is two.

As shown in Figure 3.7, there are two interpretations of the transformation of \mathcal{A} : 1) the coordinate system remains fixed, and the \mathcal{A} is translated; 2) \mathcal{A} remains fixed and the coordinate system is translated in the opposite direction. The first one indicates how the transformation appears while standing at the origin, 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. Note that numerous books cover coordinate transformations under the second interpretation. This has been known to cause confusion since the transformations may sometimes appear “backwards” from what is desired.

Rotation The robot, \mathcal{A} , can be *rotated* counterclockwise by some angle $\theta \in [0, 2\pi)$ by mapping every $(x, y) \in \mathcal{A}$ to $(x \cos \theta - y \sin \theta, x \sin \theta + y \cos \theta)$. Using a 2×2 rotation matrix,

$$R(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}, \quad (3.25)$$

the transformation can be written as

$$\begin{pmatrix} x \cos \theta - y \sin \theta \\ x \sin \theta + y \cos \theta \end{pmatrix} = R(\theta) \begin{pmatrix} x \\ y \end{pmatrix}. \quad (3.26)$$

Using the notation of Section 3.2.1, $R(\theta)$ would be $h(q)$, for which $q = \theta$. For linear transformations, such as the one defined above, 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 they are orthogonal. Suppose that the X and Y coordinate axes 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 rotation matrices of any dimension.

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.

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 in \mathcal{W} . Note these two transformations do not commute! If the operations are applied successively, each $(x, y) \in \mathcal{A}$ is transformed to

$$\begin{pmatrix} x \cos \theta - y \sin \theta + x_t \\ x \sin \theta + y \cos \theta + y_t \end{pmatrix}. \quad (3.27)$$

Notice that the following matrix multiplication will yield the same result for the first two vector components

$$\begin{pmatrix} \cos \theta & -\sin \theta & x_t \\ \sin \theta & \cos \theta & y_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x \cos \theta - y \sin \theta + x_t \\ x \sin \theta + y \cos \theta + y_t \\ 1 \end{pmatrix}. \quad (3.28)$$

This implies that the 3×3 matrix,

$$T = \begin{pmatrix} \cos \theta & -\sin \theta & x_t \\ \sin \theta & \cos \theta & y_t \\ 0 & 0 & 1 \end{pmatrix}, \quad (3.29)$$

may be used to represent a rotation followed by a translation:

$$T = \begin{pmatrix} \cos \theta & -\sin \theta & x_t \\ \sin \theta & \cos \theta & y_t \\ 0 & 0 & 1 \end{pmatrix}. \quad (3.30)$$

The matrix T will be referred to as a *homogeneous transformation*. It is important to remember that T represents a rotation *followed by* a translation (not the other

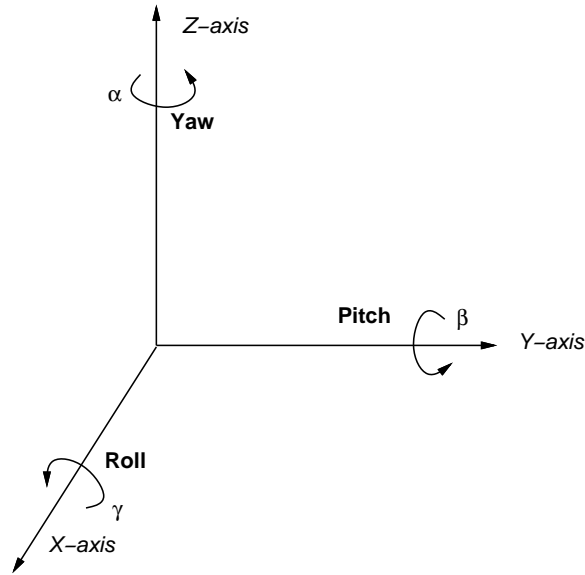


Figure 3.8: Any rotation in 3D can be described as a sequence of yaw, pitch, and roll rotations.

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 borrowed from projective geometry, where it plays an important role.

3.2.3 3D Transformations

The rigid body transformations for the 3D case are conceptually similar the 2D case; however, the 3D case appears more difficult because 3D rotations are significantly more complicated than 2D rotations.

One *translates* \mathcal{A} by some $x_t, y_t, z_t \in \mathbb{R}$ by mapping every $(x, y, z) \in \mathcal{A}$ to $(x + x_t, y + y_t, z + z_t)$. Primitives of the form $H_i = \{(x, y, z) \in \mathcal{W} \mid f_i(x, y, z) \leq 0\}$, are transformed to $\{(x, y, z) \in \mathcal{W} \mid f_i(x - x_t, y - y_t, z - z_t) \leq 0\}$. The translated robot is denoted as $\mathcal{A}(x_t, y_t, z_t)$.

Note that a 3D body can be independently 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

$$R_Z(\alpha) = \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}. \quad (3.31)$$

Note that the upper left entries of $R_Z(\alpha)$ form a 2D rotation applied to the XY coordinates, while the Z coordinate remains constant.

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

$$R_Y(\beta) = \begin{pmatrix} \cos \beta & 0 & \sin \beta \\ 0 & 1 & 0 \\ -\sin \beta & 0 & \cos \beta \end{pmatrix}. \quad (3.32)$$

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

$$R_X(\gamma) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \gamma & -\sin \gamma \\ 0 & \sin \gamma & \cos \gamma \end{pmatrix}. \quad (3.33)$$

Each rotation matrix is a simple extension of the 2D rotation matrix, (3.25). For example, the yaw matrix, $R_z(\alpha)$ essentially performs a 2D rotation with respect to the XY 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 $R(\alpha, \beta, \gamma) = R_Z(\alpha) R_Y(\beta) R_X(\gamma) =$

$$\begin{pmatrix} \cos \alpha \cos \beta & \cos \alpha \sin \beta \sin \gamma - \sin \alpha \cos \gamma & \cos \alpha \sin \beta \cos \gamma + \sin \alpha \sin \gamma \\ \sin \alpha \cos \beta & \sin \alpha \sin \beta \sin \gamma + \cos \alpha \cos \gamma & \sin \alpha \sin \beta \cos \gamma - \cos \alpha \sin \gamma \\ -\sin \beta & \cos \beta \sin \gamma & \cos \beta \cos \gamma \end{pmatrix}. \quad (3.34)$$

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, 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 (or reference) frame, not the frame in which \mathcal{A} is defined. Each

time a new rotation matrix is introduced from the left, it has no concern for the orientations of any axes that were used for defining \mathcal{A} . It simply rotates every point in \mathbb{R}^3 in terms of the global reference 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 $\mathcal{A}(\alpha, \beta, \gamma)$.

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

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}, \quad (3.35)$$

is given. By setting each entry equal to its corresponding entry in (3.34), 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

$$\alpha = \tan^{-1}(r_{11}/r_{21}), \quad (3.36)$$

$$\beta = \tan^{-1}(\sqrt{r_{32}^2 + r_{33}^2}/-r_{31}), \quad (3.37)$$

and

$$\gamma = \tan^{-1}(r_{32}/r_{33}). \quad (3.38)$$

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 to the left or right of the Y axis, and the denominator selects whether the direction will be above or below the X axis. This is the same as the *atan2* function in C, which nicely expands the range of the arctangent to $[0, 2\pi)$. This can be applied to express (3.36), (3.37) and (3.38) as

$$\alpha = \text{atan2}(r_{11}, r_{21}), \quad (3.39)$$

$$\beta = \text{atan2}(\sqrt{r_{32}^2 + r_{33}^2}, -r_{31}), \quad (3.40)$$

and

$$\gamma = \text{atan2}(r_{32}, r_{33}). \quad (3.41)$$

Note that this method assumes $r_{21} \neq 0$ and $r_{33} \neq 0$.

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 $T =$

$$\begin{pmatrix} \cos \alpha \cos \beta & \cos \alpha \sin \beta \sin \gamma - \sin \alpha \cos \gamma & \cos \alpha \sin \beta \cos \gamma + \sin \alpha \sin \gamma & x_t \\ \sin \alpha \cos \beta & \sin \alpha \sin \beta \sin \gamma + \cos \alpha \cos \gamma & \sin \alpha \sin \beta \cos \gamma - \cos \alpha \sin \gamma & y_t \\ -\sin \beta & \cos \beta \sin \gamma & \cos \beta \cos \gamma & z_t \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3.42)$$

Once again, the order of operations is critical. The matrix T in (3.42) represents the following sequence of transformations:

1. Roll by γ .
2. Pitch by β .
3. Yaw by α .
4. Translation 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 Transformations of 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 $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_m$ 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 Kinematic Chain in \mathbb{R}^2

Before considering a chain, suppose \mathcal{A}_1 and \mathcal{A}_2 are two 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, in which the independent parameters are $x_1, y_1, \theta_1, x_2, y_2$, and θ_2 . 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*. In Figure 3.9.a, a *revolute joint* is shown, in which one link is capable only of rotation with respect to the other. In Figure 3.9.b, a *prismatic joint* is shown, in which one link translates along the other. Each type of joint removes two degrees of freedom from the pair of bodies. For example, consider a revolute joint that connects \mathcal{A}_1 to \mathcal{A}_2 . Assume that the point $(0, 0)$ in the model for \mathcal{A}_2 is permanently fixed to a point (x_a, y_a) on \mathcal{A}_1 . This implies that the translation of \mathcal{A}_2 will be completely determined once

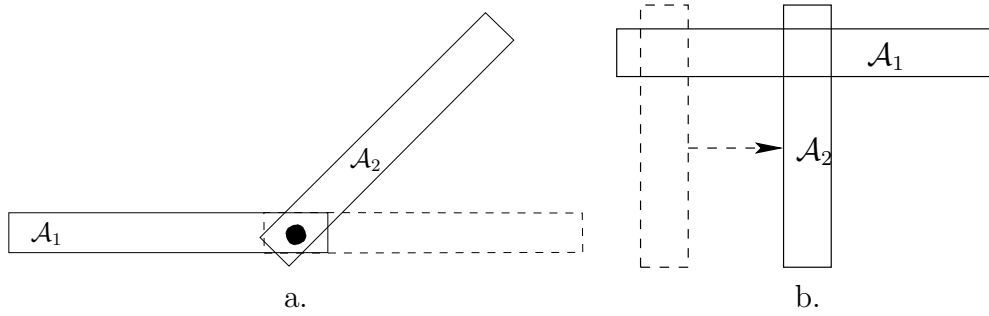


Figure 3.9: Two types of 2D joints: a) a revolute joint allows one link to rotate with respect to the other, b) a prismatic joint allows one link to translate with respect to the other.

x_a and y_a are given. Note that x_a and y_a are functions of 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 , x_2 , 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 , \dots , \mathcal{A}_m are transformed, and give the expressions in terms of these 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 coordinate 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 coordinate 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 -axis is defined as the line through both joints that lie in \mathcal{A}_i , as shown in Figure 3.9. For the last link, \mathcal{A}_m , the X -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 coordinate frame for the first link, \mathcal{A}_1 , can be placed using the same considerations as for a single rigid body.

We are now prepared to determine the location of each link. The position and orientation of link \mathcal{A}_1 in \mathcal{W} is determined by applying the 2D homogeneous transform matrix (3.30),

$$T_1 = \begin{pmatrix} \cos \theta_1 & -\sin \theta_1 & x_t \\ \sin \theta_1 & \cos \theta_1 & y_t \\ 0 & 0 & 1 \end{pmatrix}. \quad (3.43)$$

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 transform matrix (3.30), specialized for link \mathcal{A}_i

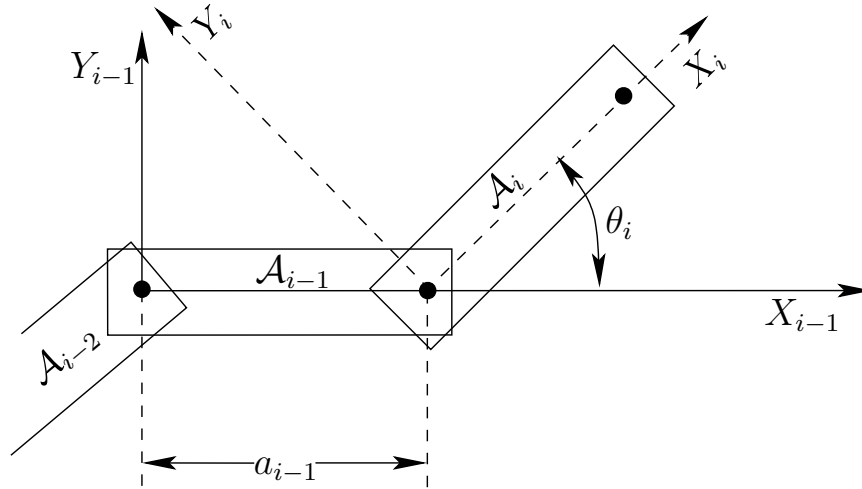


Figure 3.10: The coordinate frame that is used to define the geometric model for each \mathcal{A}_i , for $1 < i < m$, is based on the joints that connect \mathcal{A}_i to \mathcal{A}_{i-1} and \mathcal{A}_{i+1} .

for $1 < i \leq m$,

$$T_i = \begin{pmatrix} \cos \theta_i & -\sin \theta_i & a_{i-1} \\ \sin \theta_i & \cos \theta_i & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad (3.44)$$

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

$$T_1 T_2 \cdots T_m \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \quad (3.45)$$

Example 3.3.1 To gain an intuitive understanding of these transformations, consider determining the configuration for link \mathcal{A}_3 , as shown in Figure 3.11. Figure 3.11.a shows a three-link chain in which \mathcal{A}_1 is at its initial configuration, and the other links are each offset by $\frac{\pi}{4}$ from the previous link. Figure 3.11.b 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 3.11.c, 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

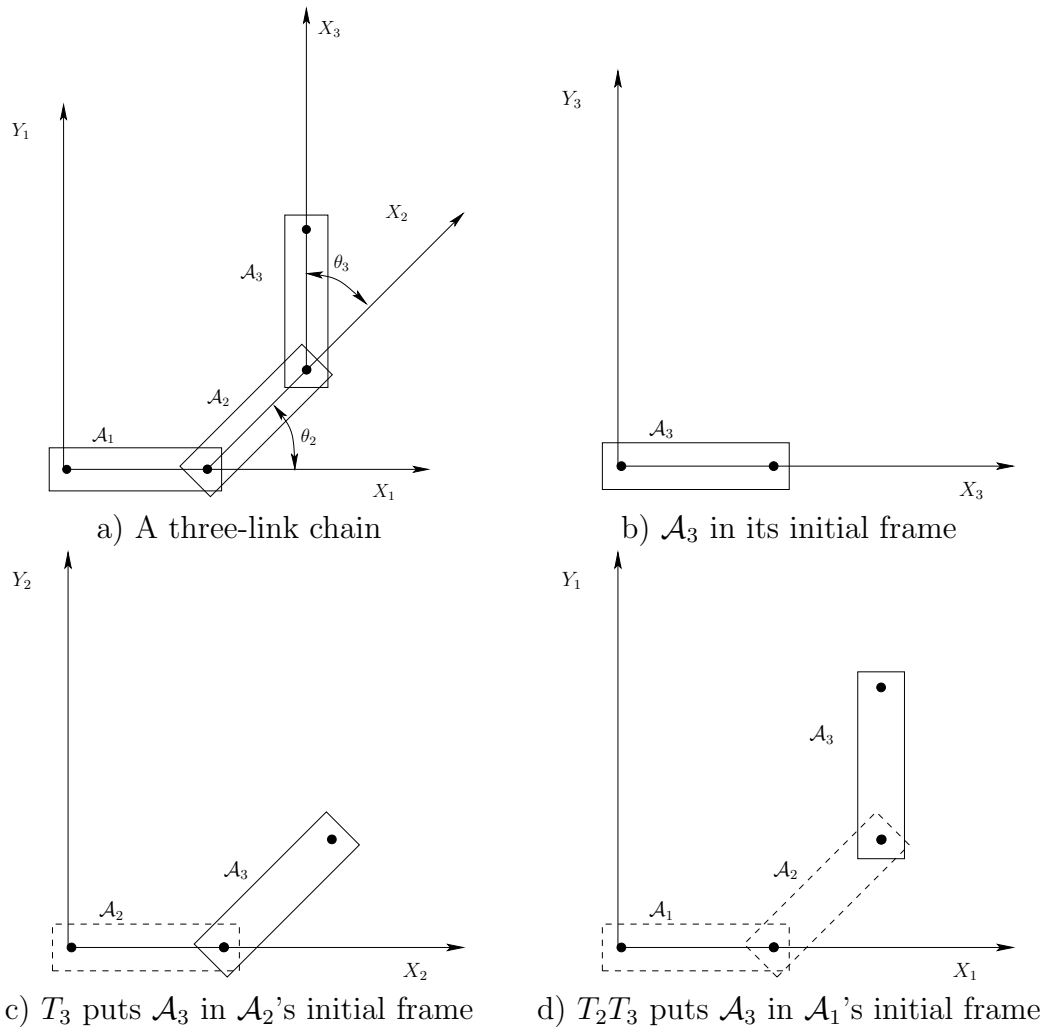


Figure 3.11: Applying the transformation T_2T_3 to the model of \mathcal{A}_3 . In this case, T_1 is the identity matrix.

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 parameters a_i are treated as constants, and the θ_i are variables. The transformed m^{th} link is represented as $\mathcal{A}_m(x_t, y_t, \theta_1, \dots, \theta_m)$. 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 $\mathcal{A}_m(\theta_1, \dots, \theta_m)$. For prismatic joints, the a_i are treated as variables, as opposed to the θ_i . Of course, it is possible to include both types of joints in a single kinematic chain.

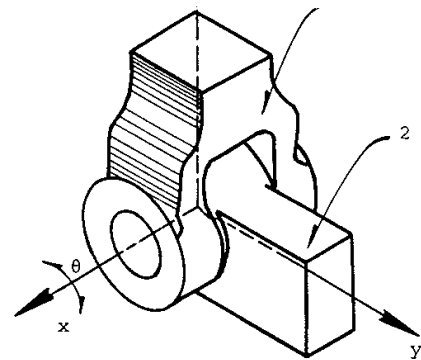
3.3.2 A Kinematic Chain in \mathbb{R}^3

As for a single rigid body, the 3D case is significantly more complicated than 2D 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:

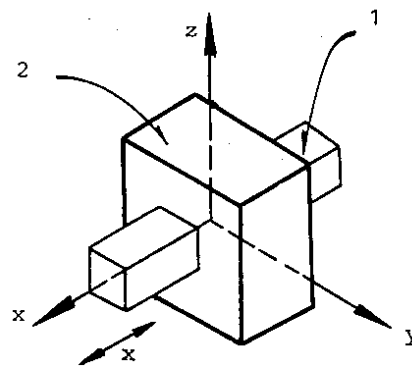
1. The coordinate frame must be carefully placed to define the model for each \mathcal{A}_i .
2. Based on joint relationships, several parameters are measured.
3. The parameters are used to define a homogeneous transformation matrix, T_i .
4. The transformation of any point on link \mathcal{A}_m is given 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 Z_i refer to 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 X_i join the closest pair of points between Z_i and Z_{i+1} , with the origin on Z_i and the direction pointing towards the nearest point of Z_{i+1} . This axis is uniquely defined if the Z_i and Z_{i+1} are not parallel. The recommended coordinate frame for defining the geometric model for each \mathcal{A}_i will be given with respect to Z_i and X_i , which are given 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.

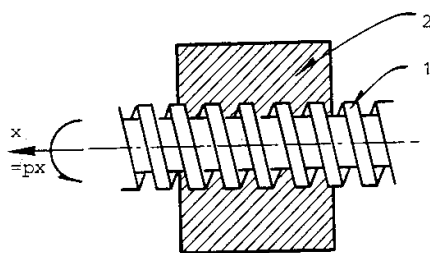
In Section 3.3.1, each T_i was defined in terms of two parameters, a_{i-1} and θ_i . For the 3D case, four parameters will be defined: d_i , θ_i , a_{i-1} , and α_{i-1} . These are referred to as *Denavit-Hartenberg parameters*, or DH parameters for short [316]. The definition of each parameter is indicated in Figure 3.15. Figure 3.15.a shows the definition of d_i . Note that X_{i-1} and X_i contact Z_i at two different places. Let d_i denote signed distance between these points of contact. If X_i is above X_{i-1} along Z_i , then d_i is positive; otherwise, d_i is negative. The parameter θ_i is the angle between X_i and X_{i-1} , which corresponds to the rotation about Z_i that moves X_{i-1} to coincide X_i . In Figure 3.15.b, Z_i is pointing outward. The



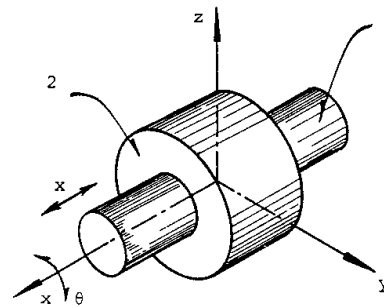
Revolute
1 Degree of Freedom



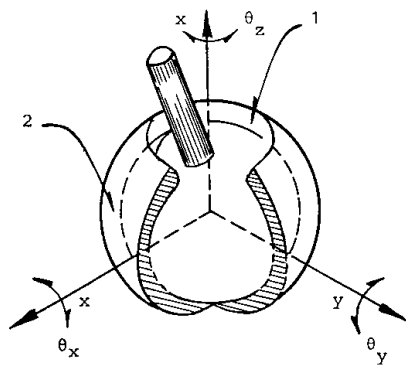
Prismatic
1 Degree of Freedom



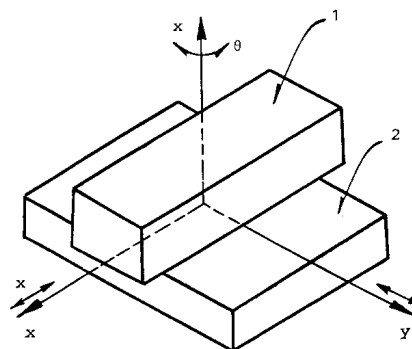
Screw
1 Degree of Freedom



Cylindrical
2 Degrees of Freedom



Spherical
3 Degrees of Freedom



Planar
3 Degrees of Freedom

Figure 3.12: Types of 3D Joints

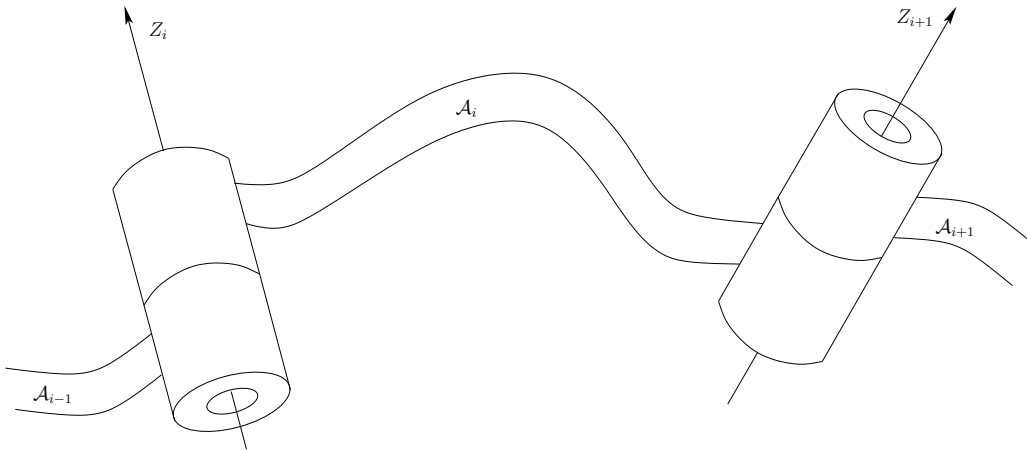
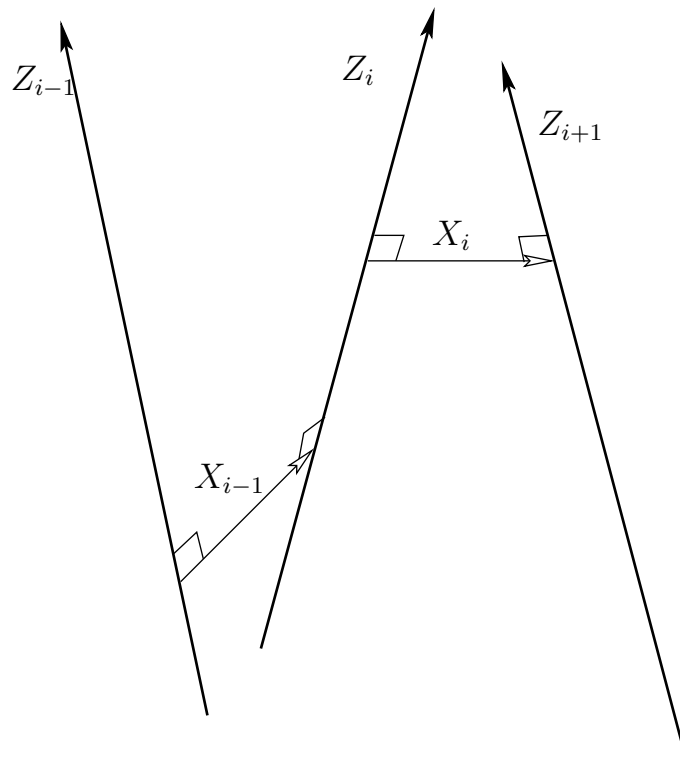


Figure 3.13: The diagram of a generic link.

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

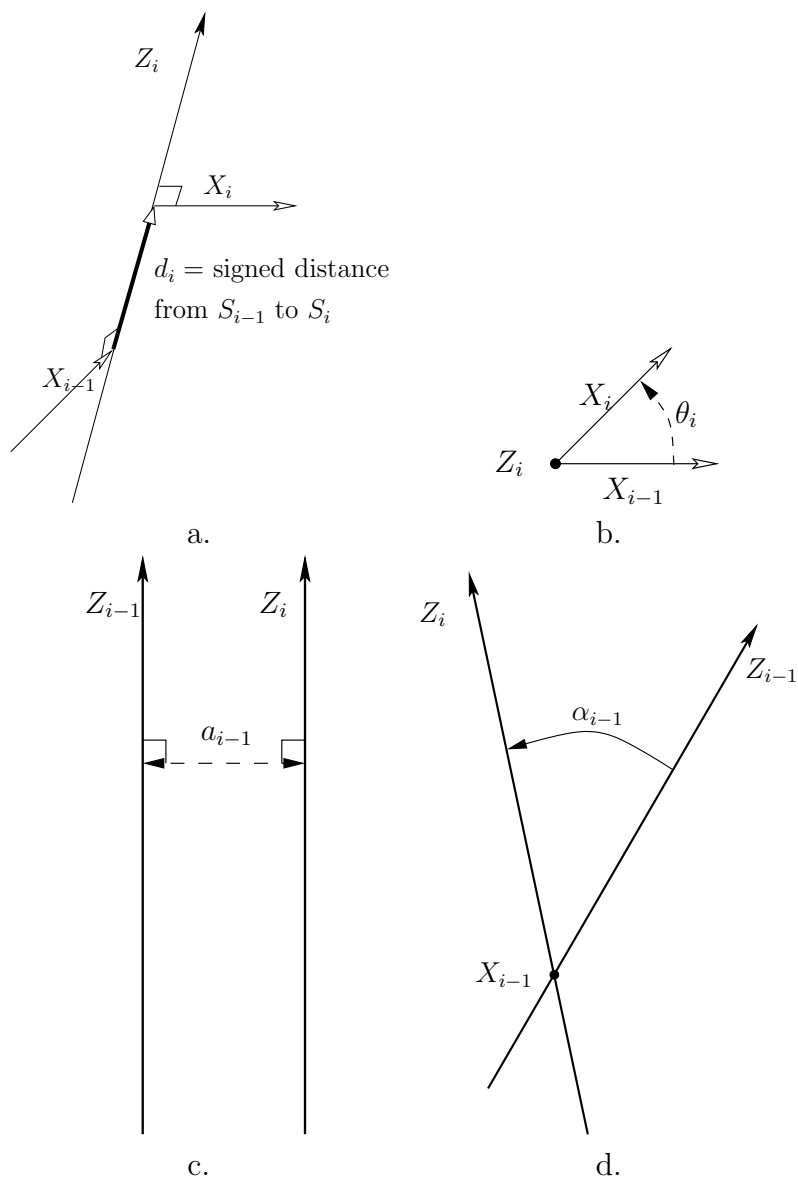


Figure 3.15: Definitions of the four DH parameters: d_i , θ_i , a_{i-1} , α_{i-1}

parameter a_i is the distance between Z_i and Z_{i-1} ; recall these are generally skew lines in \mathbb{R}^3 . The parameter α_{i-1} is the angle between Z_i and Z_{i-1} . In Figure 3.15.d, X_{i-1} is pointing outward.

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

$$R_i = \begin{pmatrix} \cos \theta_i & -\sin \theta_i & 0 & 0 \\ \sin \theta_i & \cos \theta_i & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3.46)$$

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 effect of R_i is independent of the ordering of the rotation by θ_i and the translation by d_i because both 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 Z_i . The second transformation is

$$Q_{i-1} = \begin{pmatrix} 1 & 0 & 0 & a_{i-1} \\ 0 & \cos \alpha_{i-1} & -\sin \alpha_{i-1} & 0 \\ 0 & \sin \alpha_{i-1} & \cos \alpha_{i-1} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad (3.47)$$

which can be considered as a screw about the X_{i-1} axis. A rotation of α_{i-1} about X_{i-1} is followed by a translation of a_{i-1} .

Transformation matrix The homogeneous transformation matrix, T_i , for $1 < i \leq m$, is

$$T_i = Q_{i-1}R_i = \begin{pmatrix} \cos \theta_i & -\sin \theta_i & 0 & a_{i-1} \\ \sin \theta_i \cos \alpha_{i-1} & \cos \theta_i \cos \alpha_{i-1} & -\sin \alpha_{i-1} & -\sin \alpha_{i-1}d_i \\ \sin \theta_i \sin \alpha_{i-1} & \cos \theta_i \sin \alpha_{i-1} & \cos \alpha_{i-1} & \cos \alpha_{i-1}d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (3.48)$$

This can be considered as the 3D counterpart to the 2D transformation matrix, (3.30). The following four operations are performed in succession:

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

| 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 |

Table 3.1: The DH parameters are shown for substitution into each homogeneous transformation matrix (3.48). Note that the parameters a_3 and d_3 must be written as negative values (they are signed displacements, not distances).

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 (3.42). If the first body is only capable of rotation via a revolute joint, then 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_t axes). This implies that Q_0 from (3.47) is the identity matrix, which makes $T_1 = R_1$.

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

$$T_1 T_2 \cdots T_m \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}. \quad (3.49)$$

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 3.3.2 (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 [?, 413]. The procedure is to determine appropriate coordinate frames to represent each one of the links. The first three links allow the hand (called an end-effector) to make many large movements in the \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 coordinate frames are shown in Figure 3.16, and the corresponding DH parameters are given

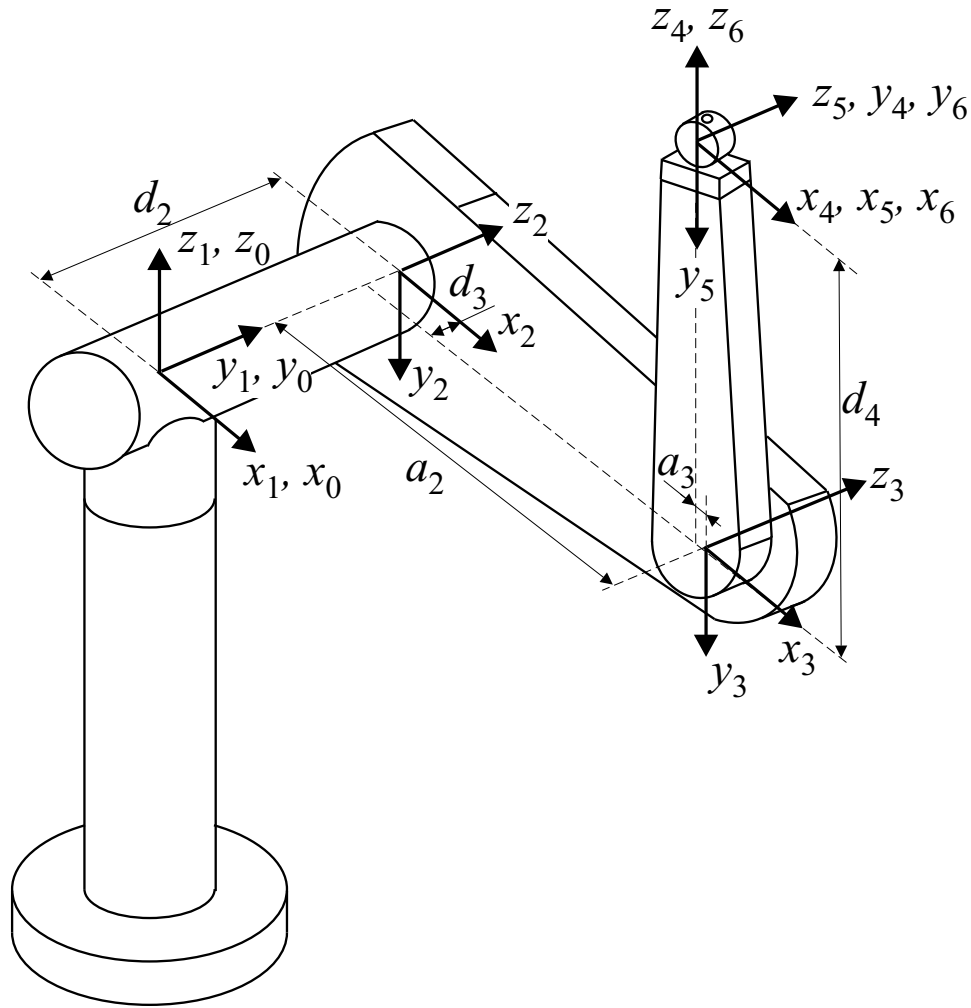


Figure 3.16: The Puma 560 is shown along with the the DH parameters and coordinate frames for each link in the chain. This figure is borrowed from [413] by courtesy of the authors.

in Table 3.1. Each transformation matrix, T_i , may be considered as a function of θ_i ; hence, it is written $T_i(\theta_i)$. The other parameters are fixed for the this example. Only $\theta_1, \theta_2, \dots, \theta_6$ are allowed to vary.

The parameters from Table 3.1 may be substituted into the homogeneous transformation matrices to obtain

$$T_1 = \begin{pmatrix} \cos \theta_1 & -\sin \theta_1 & 0 & 0 \\ \sin \theta_1 & \cos \theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad (3.50)$$

$$T_2 = \begin{pmatrix} \cos \theta_2 & -\sin \theta_2 & 0 & 0 \\ 0 & 0 & 1 & d_2 \\ -\sin \theta_2 & -\cos \theta_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad (3.51)$$

$$T_3 = \begin{pmatrix} \cos \theta_3 & -\sin \theta_3 & 0 & a_2 \\ \sin \theta_3 & \cos \theta_3 & 0 & 0 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad (3.52)$$

$$T_4 = \begin{pmatrix} \cos \theta_4 & -\sin \theta_4 & 0 & a_3 \\ 0 & 0 & -1 & -d_4 \\ \sin \theta_4 & \cos \theta_4 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad (3.53)$$

$$T_5 = \begin{pmatrix} \cos \theta_5 & -\sin \theta_5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin \theta_5 & -\cos \theta_5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad (3.54)$$

and

$$T_6 = \begin{pmatrix} \cos \theta_6 & -\sin \theta_6 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \sin \theta_6 & \cos \theta_6 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (3.55)$$

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

$$T_1(\theta_1)T_2(\theta_2)T_3(\theta_3)T_4(\theta_4)T_5(\theta_5)T_6(\theta_6) \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}. \quad (3.56)$$

■

Example 3.3.3 (Transforming Octane) Figure 3.17 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 3.18. Studying the configurations (called *conformations*) of molecules is an important part of computational 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.

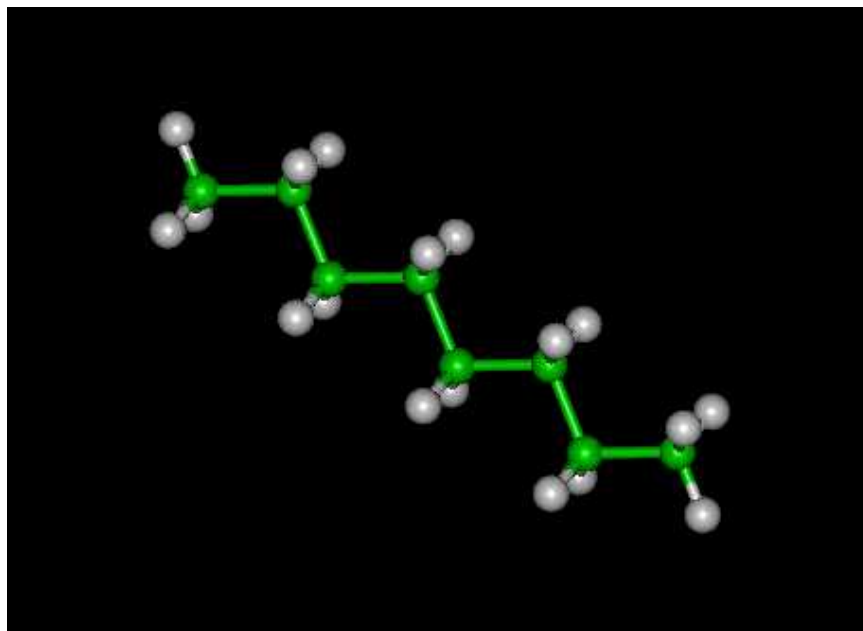


Figure 3.17: A hydrocarbon (octane) molecule with 8 carbon atoms and 18 hydrogen atoms (courtesy of the New York University Molecular Library).

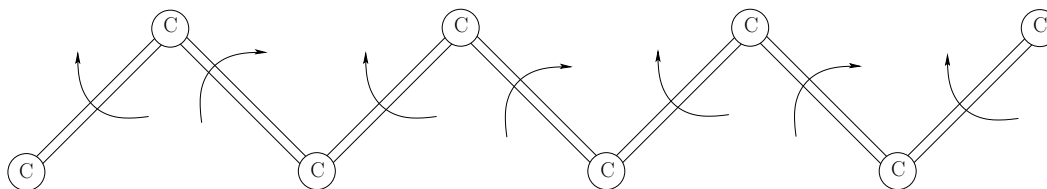


Figure 3.18: Consider transforming the spine of octane by ignoring hydrogen atoms and allowing the bonds between carbons to rotate. You could also construct this easily with Tinkertoys. If the first link is held fixed, then there are six degrees of freedom. The rotation of the last link is ignored.

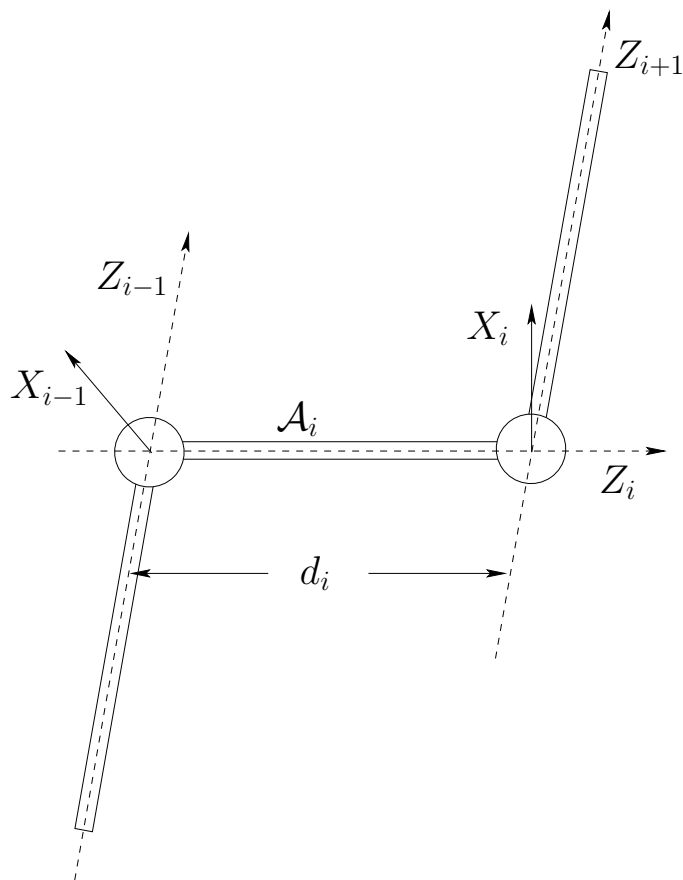


Figure 3.19: Each bond may be interpreted as a “link” of length d_i that is aligned with the Z_i axis. Note that most of \mathcal{A}_i appears in the negative Z_i direction.

Note that the bonds correspond exactly to the axes of rotation. This suggests that Z_i axes should be chosen to coincide with the bonds. Since consecutive bonds meet at atom, there is no distance between them. From Figure 3.15.c, observe that this will make $a_i = 0$ for all i . From Figure 3.15.a, it can be seen that each d_i will correspond to a *bond length*, the distance between consecutive carbon atoms. See Figure ???. 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 3.15.d, 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 3.15.b, observe that the example is degenerate because each X_i 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 Z_i 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 Z_i and Z_{i+1} . Thus, most of the points in \mathcal{A}_i will lie in the $-Z_i$ direction; see Figure ???.

The next task is to write down the matrices. Attach a coordinate 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 ???. To define T_1 , recall that $T_1 = R_1$ from (3.46) because Q_0 is dropped. The parameter d_1 represents the distance between the intersection points of Axis 0 and Axis 2 along Axis 1. Since there is no Axis 0, there is freedom to choose d_1 ; hence, let $d_1 = 0$ to obtain

$$T_1(\theta_1) = R_1(\theta_1) = \begin{pmatrix} \cos \theta_1 & -\sin \theta_1 & 0 & 0 \\ \sin \theta_1 & \cos \theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (3.57)$$

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

$$T_i(\theta_i) = \begin{pmatrix} \cos \theta_i & -\sin \theta_i & 0 & 0 \\ \sin \theta_i \cos \alpha_{i-1} & \cos \theta_i \cos \alpha_{i-1} & -\sin \alpha_{i-1} & -\sin \alpha_{i-1} d_i \\ \sin \theta_i \sin \alpha_{i-1} & \cos \theta_i \sin \alpha_{i-1} & \cos \alpha_{i-1} & \cos \alpha_{i-1} d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad (3.58)$$

for $i \in \{2, \dots, 7\}$. The notation $T_i(\theta_i)$ indicates that θ_i is the only variable. All other parameters of T_i are constants. The position of any point, (x, y, z) on the

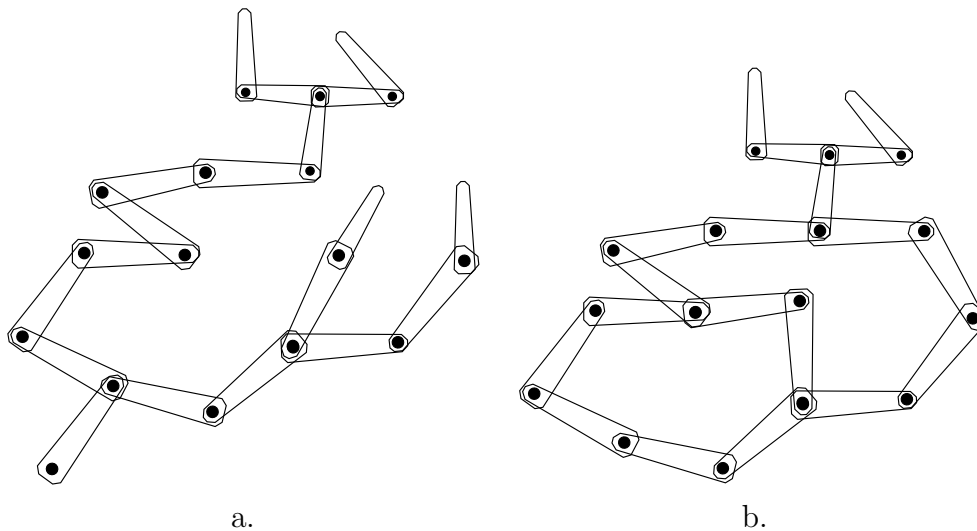


Figure 3.20: General linkages: a) Instead of a chain of rigid bodies, a “tree” of rigid bodies can be considered; b) if there are loops, then parameters must be carefully assigned to ensure that the loops are closed.

last link, \mathcal{A}_7 , is given by

$$T_1(\theta_1)T_2(\theta_2)T_3(\theta_3)T_4(\theta_4)T_5(\theta_5)T_6(\theta_6)T_7(\theta_7) \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}. \quad (3.59)$$

■

3.4 Transformations of Kinematic Trees

Motivation For many interesting problems, the linkage is arranged in a “tree” as shown in Figure 3.20.a. Assume here that the links are not attached in ways that form loops (i.e., Figure 3.20.b); 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 indicated by Figure ??, 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 ?? involved a single chain; however, most organic molecules

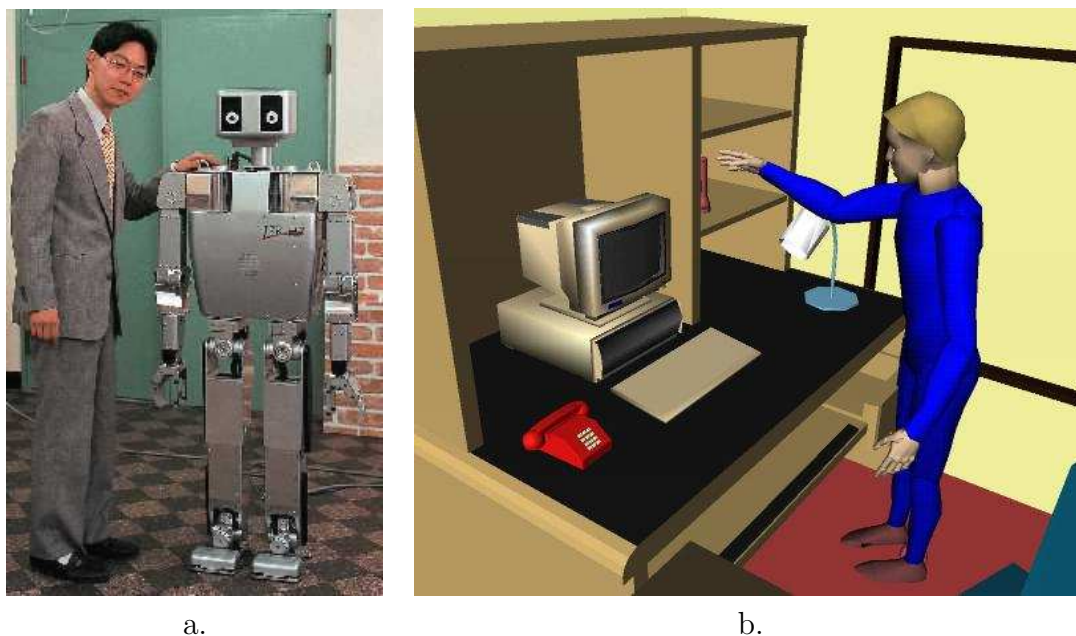


Figure 3.21: a) This is a picture of the H7 humanoid robot and one of its developers, S. Kagami. It was developed in the JSK Laboratory at the University of Tokyo. b) This is a digital actor whose motions were generated by planning algorithms. This was part of the Ph.D. thesis of James Kuffner at Stanford University.

are more complicated, as in the familiar drugs shown in Figure 3.22. 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 or not the molecule can twist the right way so that it docks nicely (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.

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 3.20.a. 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 chain of bodies that connects \mathcal{A}_i to \mathcal{A}_1 , while ignoring all other bodies. When determining the degrees of freedom of the entire tree, there will be one θ_i for each link of the tree. This case seems quite straightforward; unfortunately, it is not this easy in general.

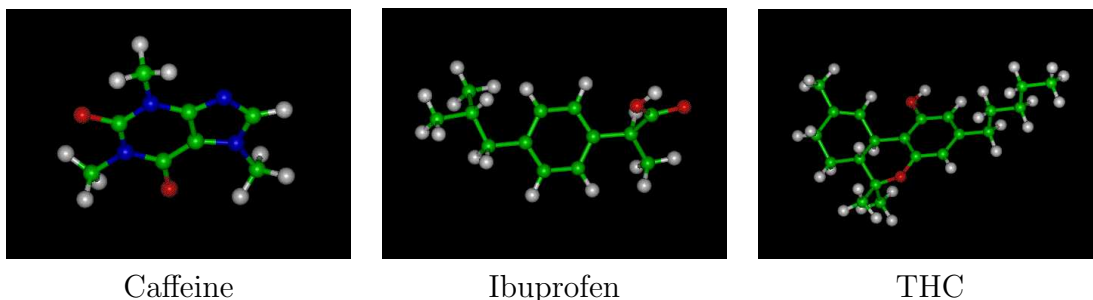


Figure 3.22: Several familiar drugs are pictured as ball-and-stick models (courtesy of the New York University Molecular Library). Analyzing the flexibility of these molecules is an important part of drug design. Note that they can be treated as robots made of many links. Kinematic tree and closed chain issues become important.

Junctions with more than two rotation axes Now consider modeling a more complicated collection of attached links. The main novelty that is that one link may have joints attached to it in more than two locations, as in \mathcal{A}_7 from Figure 3.23. 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, for instance. The legs, arms, and head could all be modeled as independent chains. In each chain, the only concern is that the first link will not necessarily be defined around the coordinate origin. This could be accounted for 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 3.23 represents part of a 2D system of links for which the root, \mathcal{A}_1 is attached to via a chain of bodies to \mathcal{A}_5 . To transform link \mathcal{A}_9 , the tools from Section 3.3.1 may be directly applied to yield a sequence of transformations,

$$T_1 \cdots T_5 T_6 T_7 T_8 T_9 \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad (3.60)$$

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

$$T_1 \cdots T_5 T_6 T_7 T_{12} T_{13} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad (3.61)$$

can be used by ignoring the chain of links \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 3.24, its coordinate

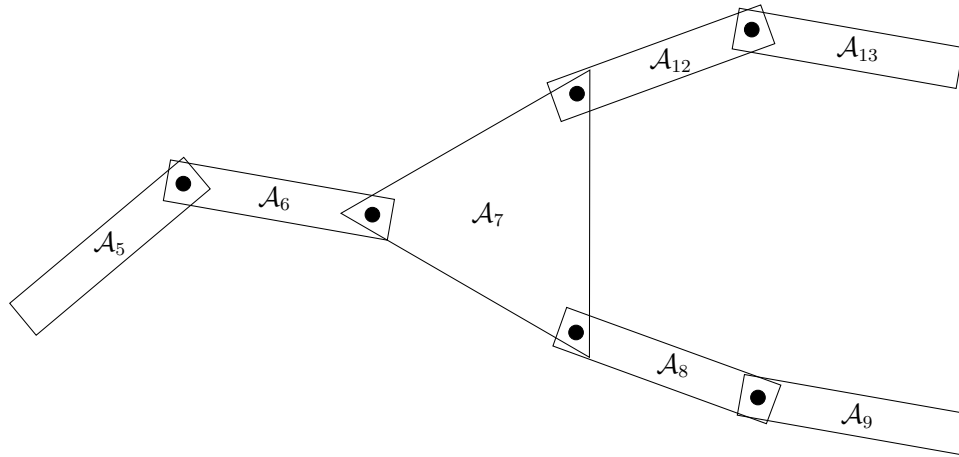


Figure 3.23: Now it is possible for a link to have more than two joints, as in \mathcal{A}_7 .

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.

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 3.25, 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 an independent variable. The parameter θ_{7u} must then be chosen as $\theta_{7l} + \phi$, in which ϕ is shown in Figure 3.24.

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.

Example 3.4.1 Figure 3.26 shows a 2D example that involves six links. To transform \mathcal{A}_6 , the only relevant links are \mathcal{A}_5 , \mathcal{A}_2 , and \mathcal{A}_1 . The chain of transformations

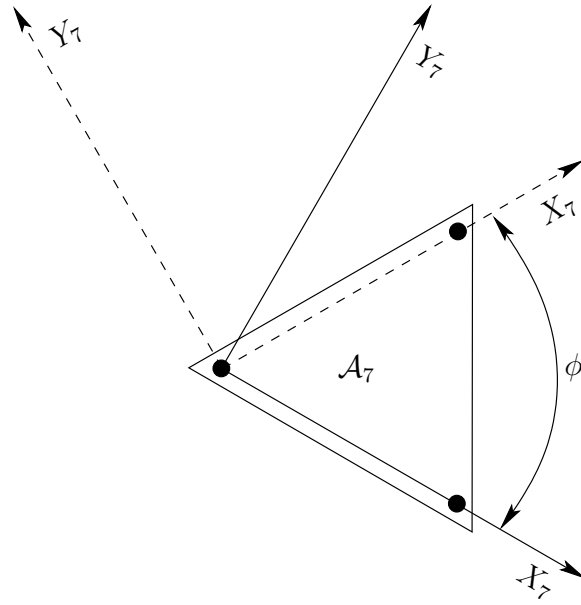


Figure 3.24: The junction is assigned two different frames, depending on which chain was followed. The solid axes were obtained from transforming \mathcal{A}_9 , and the dashed axes were obtained from transforming \mathcal{A}_{13} .

is

$$T_1 T_{2l} T_5 T_6 \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad (3.62)$$

in which

$$T_1 = \begin{pmatrix} \cos \theta_1 & -\sin \theta_1 & x_t \\ \sin \theta_1 & \cos \theta_1 & y_t \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos \theta_1 & -\sin \theta_1 & 0 \\ \sin \theta_1 & \cos \theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad (3.63)$$

$$T_{2l} = \begin{pmatrix} \cos \theta_{2l} & -\sin \theta_{2l} & a_1 \\ \sin \theta_{2l} & \cos \theta_{2l} & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos \theta_2 & -\sin \theta_2 & 1 \\ \sin \theta_2 & \cos \theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad (3.64)$$

$$T_5 = \begin{pmatrix} \cos \theta_5 & -\sin \theta_5 & a_2 \\ \sin \theta_5 & \cos \theta_5 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos \theta_5 & -\sin \theta_5 & \sqrt{2} \\ \sin \theta_5 & \cos \theta_5 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad (3.65)$$

and

$$T_6 = \begin{pmatrix} \cos \theta_6 & -\sin \theta_6 & a_5 \\ \sin \theta_6 & \cos \theta_6 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos \theta_6 & -\sin \theta_6 & 1 \\ \sin \theta_6 & \cos \theta_6 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad (3.66)$$

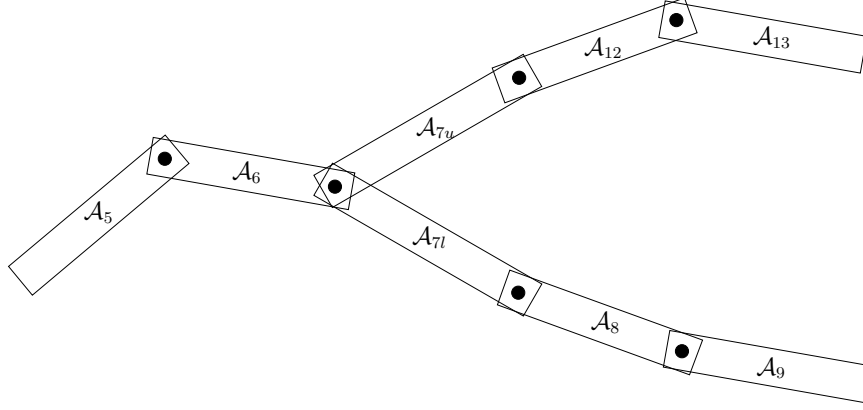


Figure 3.25: Choosing each θ_7 independently would result in a tree that ignores that fact that \mathcal{A}_7 is rigid.

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

$$T_1 T_{2u} T_4 T_5 \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad (3.67)$$

in which T_1 is the same as before, and

$$T_3 = \begin{pmatrix} \cos \theta_3 & -\sin \theta_3 & a_2 \\ \sin \theta_3 & \cos \theta_3 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos \theta_3 & -\sin \theta_3 & \sqrt{2} \\ \sin \theta_3 & \cos \theta_3 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad (3.68)$$

and

$$T_4 = \begin{pmatrix} \cos \theta_4 & -\sin \theta_4 & a_4 \\ \sin \theta_4 & \cos \theta_4 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos \theta_4 & -\sin \theta_4 & 0 \\ \sin \theta_4 & \cos \theta_4 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \quad (3.69)$$

The interesting case is

$$T_{2u} = \begin{pmatrix} \cos \theta_{2u} & -\sin \theta_{2u} & a_1 \\ \sin \theta_{2u} & \cos \theta_{2u} & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta_{2l} + \pi/4) & -\sin(\theta_{2l} + \pi/4) & a_1 \\ \sin(\theta_{2l} + \pi/4) & \cos(\theta_{2l} + \pi/4) & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad (3.70)$$

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

What if there are loops? The most general case includes links that are connected in loops, as shown in Figure 3.27. These are generally referred to as *closed*

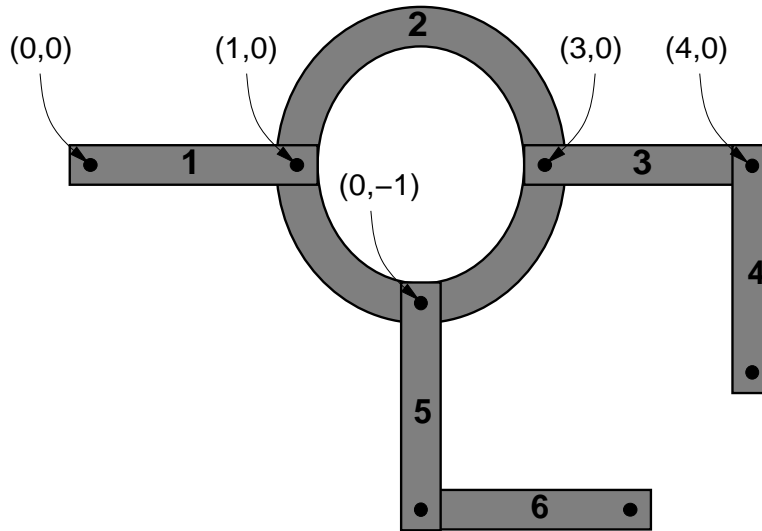


Figure 3.26: A tree of bodies in which the joints are attached in different places.

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. An another example, suppose that two robot manipulators, like the Puma 560 from Example 3.3.2, cooperate together to carry an object. If each robot grasps the same object with its hand, then a loop will be formed. Furthermore, 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 same way as the case of trees. For example, the links in Figure 3.27 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. 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 these any of 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

$$T_1(\theta_1)T_2(\theta_2)T_3(\theta_3)T_4(\theta_4)T_5(\theta_5) \begin{pmatrix} a_5 \\ 0 \\ 1 \end{pmatrix}, \quad (3.71)$$

in which $(a_5, 0) \in \mathcal{A}_5$ is the location of joint of \mathcal{A}_5 that is supposed to connect to

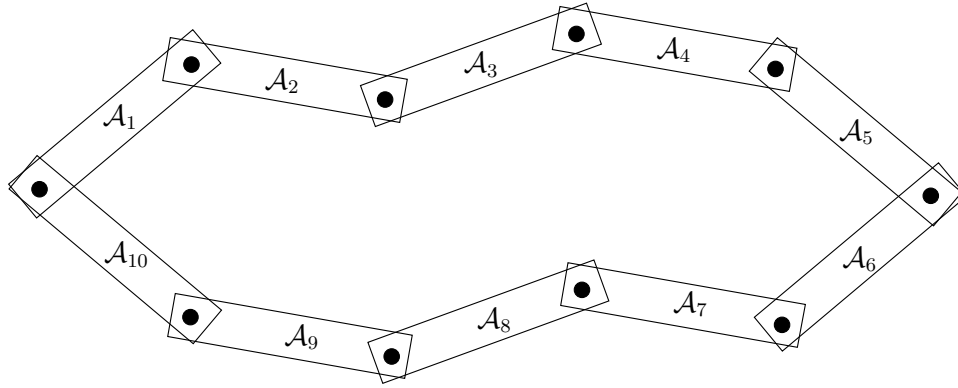


Figure 3.27: There are ten links and ten revolute joints arranged in a loop. This is an example of a closed kinematic chain.

\mathcal{A}_6 . The position of this joint may also be expressed using the lower chain as

$$T_{10}(\theta_{10})T_9(\theta_9)T_8(\theta_8)T_7(\theta_7)T_6(\theta_6) \begin{pmatrix} a_6 \\ 0 \\ 1 \end{pmatrix}, \quad (3.72)$$

with $(a_6, 0)$ representing the position of the joint in the frame of \mathcal{A}_6 . If the loop does not have to be maintained, then any values for $\theta_1, \dots, \theta_{10}$, may be selected, resulting in ten degrees of freedom. However, if a loop must be maintained, then (3.71) and (3.72) must be equal,

$$T_1(\theta_1)T_2(\theta_2)T_3(\theta_3)T_4(\theta_4)T_5(\theta_5) \begin{pmatrix} a_5 \\ 0 \\ 1 \end{pmatrix} = T_{10}(\theta_{10})T_9(\theta_9)T_8(\theta_8)T_7(\theta_7)T_6(\theta_6) \begin{pmatrix} a_6 \\ 0 \\ 1 \end{pmatrix}, \quad (3.73)$$

which is quite a mess of nonlinear, trigonometric equations that must be solved. The set of solutions to (3.73) could be very complicated. For the example, the total degrees of freedom is eight because two were removed by making the joint common. Since the common joint allows the links to rotate, only 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. The problem is a form of the well-known *inverse kinematics problem* \square .

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 (3.73) must be

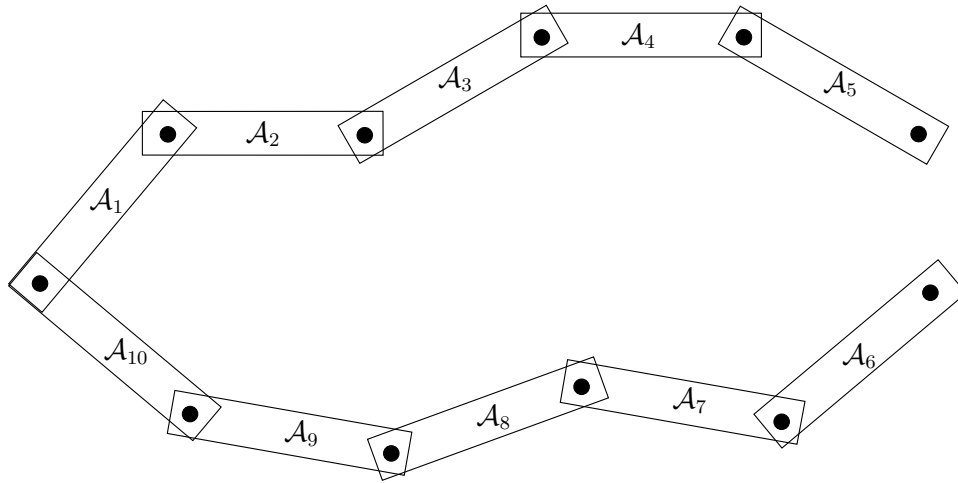


Figure 3.28: Loops may be opened to enable tree-based transformations to be applied; however, a closure constraint must still be satisfied.

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 characterizing this complicated solution space is given 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 Rotations are a special case of linear transformations, which are generally expressed by a $n \times n$ matrix, M , if the transformations are performed over \mathbb{R}^n . Consider transforming points, (x, y) in a 2D robot, \mathcal{A} , as

$$\begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}. \quad (3.74)$$

If M is a rotation matrix, then the “shape” of \mathcal{A} will remain the same. In some applications, however, it may be desirable to distort the shape.

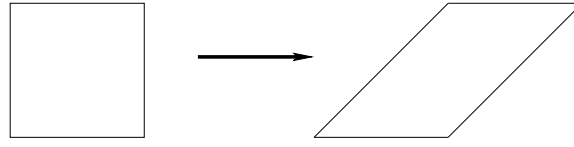


Figure 3.29: Shearing transformations may be performed.

The robot can be *scaled* by m_{11} along the X axis and m_{22} along the Y axis by applying

$$\begin{pmatrix} m_{11} & 0 \\ 0 & m_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}, \quad (3.75)$$

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

$$\begin{pmatrix} 1 & m_{12} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \quad (3.76)$$

for $m_{12} \neq 0$. The case of $m_{12} = 1$ is shown in Figure 3.29.

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 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 a hundred revolute joints together in a chain. The tools from Section 3.3 may be used to transform it with 100 rotation parameters, $\theta_1, \dots, \theta_{100}$, 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 these are designed to provide easy control over the shape of a curve through the adjustment of a small number of parameters. Other possibilities include generalized cylinders 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 [?].

Literature

A thorough coverage of solid and boundary representations, including semi-algebraic models, can be found in [331]. A standard geometric modeling book from a CAD/CAM perspective, including NURBs models is [568]. NURB models are also surveyed in [628].

The logical predicate defined in Section 3.1 can check whether a point lies inside of \mathcal{O} in $O(n)$ time, in which n is the number of primitives. Many algorithms exist that can accomplish this much more quickly. For a convex polygon, it can be determined whether a point lies inside or outside in time $O(\ln n)$ by performing range searching on the upper and lower chains of edges []. *need more refs and info.*

Discussion of optimal decompositions See Suri's survey, pp. 429-444 of CRC Handbook on DCG.

Theoretical algorithm issues regarding semi-algebraic models are covered in [558, 559]. The subject of transformations of rigid bodies and chains of bodies is covered in most robotics texts. Classic references include [180, 618]. The DH parameters were introduced in [316].

Need to talk about half-edge data structures, and related variations.

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, it not best suited for the development of motion planning (sorry!) algorithms. It is the easiest (and safe) to use for making quick 3D animations of motion planning output, but it incorrectly captures the state space for the planning algorithm. 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 many other parameterizations of rotations, such as spherical coordinates, Cayley-Rodrigues parameters, and stereographic projection. Chapter 5 of [155] provides extensive coverage of 3D rotations and different parameterizations.

The coverage of transformations of chains of bodies was heavily influenced by classic robotics texts [180, 618, ?]. The standard approach in these books 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 careful study of the configuration space in Chapter 4.

Interesting Web Pages

NYU Molecular Library: <http://www.nyu.edu/pages/mathmol/library/>

Exercises

- How would you define the semi-algebraic model to remove a triangular “nose” from the region shown in Figure 3.4?
- For distinct values of yaw, pitch, and roll, is it possible to generate the same rotation. In other words, $R(\alpha, \beta, \gamma) = R(\alpha', \beta', \gamma')$, if at least one of the angles is distinct. Characterize the sets of angles for which this occurs.
- Using rotation matrices, prove that 2D rotation is commutative, but 3D rotation is not.
- 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 (3.31) 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 $R_{euler}(\alpha, \beta, \gamma) = R_Z(\alpha)R_Y(\beta)R_Z(\gamma)$.
 - Determine the matrix R_{euler} .
 - Show that $R_{euler}(\alpha, \beta, \gamma) = R_{euler}(\alpha - \pi, -\beta, \gamma - \pi)$.
 - Suppose that a rotation matrix is given as shown in (3.35). Show that the Euler angles are

$$\alpha = \text{atan2}(r_{23}, r_{13}), \quad (3.77)$$

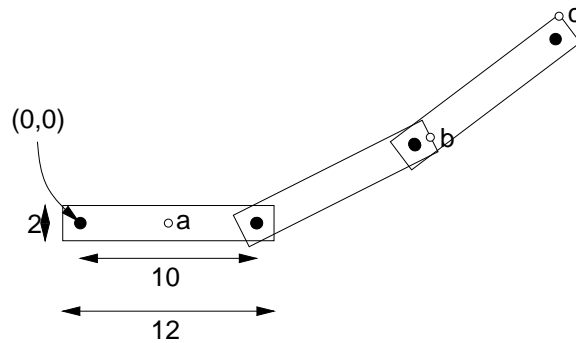
$$\beta = \text{atan2}(\sqrt{1 - r_{33}^2}, r_{33}), \quad (3.78)$$

and

$$\gamma = \text{atan2}(r_{32}, -r_{31}). \quad (3.79)$$

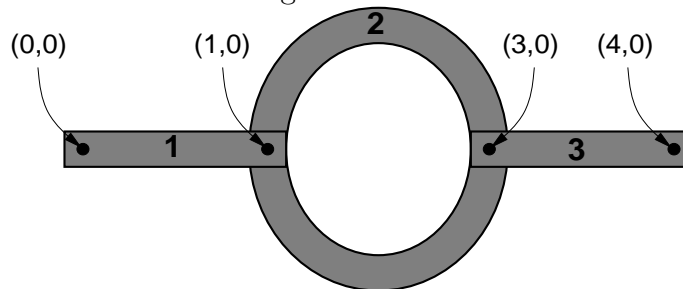
- There are 12 different variants of yaw-pitch-roll (or Euler angles), depending on which axes are used and the order of these axes. Determine all of the possibilities, using only notation such as $R_Z(\alpha)R_Y(\beta)R_Z(\gamma)$ for each one. Give brief arguments that support why or why not specific combinations rotations are included in your list of 12.
- Suppose that \mathcal{A} is a unit disc, centered at the origin and $\mathcal{W} = \mathbb{R}^2$. Assume that \mathcal{A} is represented by a single, semi-algebraic primitive, $H = \{(x, y) \mid x^2 + y^2 \leq 1\}$. Show that the transformed primitive is unchanged after rotation.

7. Consider the articulated chain of bodies shown below. 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 the \mathcal{A}_1 , and \mathcal{A}_3 is attached to the \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 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 $(\pi/4, \pi/2, -\pi/4)$. Use the homogeneous transformation matrices to determine the locations of points a , b , and c . Name the set of all configurations such that 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 articulated body that lives in a 2D world is shown below. The first link is attached at $(0,0)$, but can rotate. Each remaining link is attached to another link with a revolute joint. The second link 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 structure 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 Link 1, θ_2 is the angle of Link 2 with respect to Link 1, and θ_3 is the angle of Link 3 with respect to Link 2. Using homogeneous coordinate transformations, compute the position of the point at $(4,0)$ in

the figure above, when the structure is at configuration $(\frac{\pi}{4}, \frac{\pi}{2}, \frac{\pi}{4})$ (the point is attached to Link 3).

9. Approximate a spherical joint as a chain of three short links that are attached by revolute joints and give the sequence of transformation matrices. If the link lengths approach zero, show that the resulting sequence of transformation matrices can be used to exactly represent the kinematics of a spherical joint.
10. Recall Example 3.4.1. How should the transformations be modified so that the links are in the positions shown in Figure 3.26 precisely when $\theta_i = 0$ for every revolute joint whose angle can be independently chosen.
11. **Project: Virtual Tinkertoys** Design and implement a system in which the user can attach various links to make a 3D kinematic tree. Assume that all joints are revolute. The user should be allowed to change parameters and see the resulting positions of all of the links.
12. **Project: Virtual Human Animation** 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.

Chapter 4

The Configuration Space

| Chapter Status | |
|---|---|
|  | What does this mean? Check http://msl.cs.uiuc.edu/planning/status.html for information on the latest version. |



Chapter 3 only covered how to model and transform a collection of bodies; however, for the purposes of planning it is important to define a whole 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 the seminal work of Lozano-Pérez [507, 503, 504], who introduced this notion in the context of planning. The motion planning literature was further unified around this concept by Latombe's book [437]. 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 of 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. After reading this, you should be able to precisely characterize the configuration space 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 explicitly characterized (i.e., parameterized).

4.1 Basic Topological Concepts

4.1.1 Topological Spaces

Recall from basic mathematics, the concepts of open and closed intervals in the set of real numbers \mathbb{R} . An open interval, such as $(0, 1)$ 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 $1/2, 1/4, \dots, 1/2^i$, converges to 0 as i tends to infinity. This means that we can get within any small, positive distance from 0 or 1, but we cannot stand exactly on the boundary of the interval. For a closed interval, such as $[0, 1]$, these 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 will apply 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* such that the following axioms hold:

1. The union of a countable number of open sets is an open set.
2. The intersection of a finite number of open set 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. This will not necessarily be true for closed sets.

For the special case of $X = \mathbb{R}$, the open sets include open intervals, as expected. Note that many sets that are not intervals are included because taking unions and intersections of open intervals generates many other open sets. For example, the set

$$\bigcup_{i=1}^{\infty} \left(\frac{1}{3^i}, \frac{2}{3^i} \right), \quad (4.1)$$

which is an infinite union of intervals, is open.

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 vice versa. 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; 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.

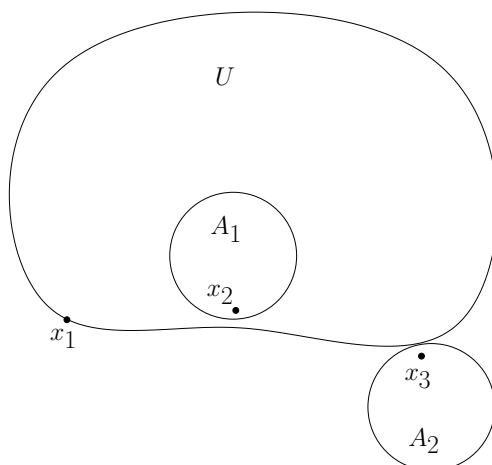


Figure 4.1: An illustration of the boundary definition. Suppose $X = \mathbb{R}^2$, and U is a subset as shown. Three kinds of points appear: 1) x_1 is a boundary point, 2) x_2 is an interior point, and 3) x_3 is an exterior point. Both x_1 and x_2 are limit points.

Here is a question to ponder: 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 (the interval $[2\pi, \infty)$ 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 , and 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 , such that $x \in O$ and $O \subseteq U$, then x is called an *interior point* of U . The set of all interior points in X is called the *interior of U* , and is denoted by $\text{int}(U)$.
- If there exists an open set, O , such that $x \in O$ and $O \subseteq X \setminus U$, 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* of U .

The set of all limit points of U is a closed set called the *closure* of U , and is denoted by $cl(U)$. Note that $cl(U) = int(U) \cup \partial U$.

For the case of $X = \mathbb{R}$, the boundary points are the endpoints of intervals. Thus, 0 and 1 are boundary points of intervals, $(0, 1)$, $[0, 1]$, $[0, 0)$, 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. For example, 0 is the limit point of the sequence generated by $1/2^i$ for each $i \in \mathcal{N}$, 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 4.1.1 ($X = \mathbb{R}^n$) We should expect that \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

$$B(x, \rho) = \{x' \in \mathbb{R}^n \mid \|x' - x\| < \rho\}, \quad (4.2)$$

in which $\|\cdot\|$ denotes the Euclidean norm (or magnitude) of x . Such sets is considered an open set 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} , note that this degenerates to representing all open sets as a union of intervals, which we have 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 $H = \{x \in \mathbb{R}^n \mid f(x) \leq 0\}$, in which $x \in \mathbb{R}^n$, 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 that sets constructed only from primitives that use the $<$ relation are open. ■

Example 4.1.2 (Subspace topology) A new topological space can easily be constructed from a subset of a topological space. This will be very useful in the

¹Such a collection is often referred to as a *basis*.

coming sections. 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 any subset of Y that can be represented as $U \cap Y$ for some open set U of X . Thus, the open sets for Y are almost the same as for X , except 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 which will appear in later in this chapter. ■

Example 4.1.3 (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 4.1.4 ($X = \{cat, dog, tree, house\}$) 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. For this case, suppose that $\{cat\}$ and $\{dog\}$ are open sets. Then, $\{cat, dog\}$ must also be an open set. Closed sets and boundary points can be even 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 A_1 and A_2 such that $x_1 \in A_1$, $x_2 \in A_2$, and $A_1 \cap A_2 = \emptyset$.

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*. The manifold definition that is used in Section 4.1.2 will guarantee that the resulting topological space is a Hausdorff space.

Continuous functions A very simple definition of continuity exists for topological spaces. It nicely generalizes the definitions from standard calculus. Let $f : X \rightarrow Y$ denote a function between topological spaces X and Y . For any set $B \subset Y$, let the *preimage* of B be denoted and defined by

$$f^{-1}(B) = \{x \in X \mid f(x) \in B\}. \quad (4.3)$$

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 the composition of functions is continuous requires only a

one-line argument that the preimage of the preimage will be open. Compare this to the cumbersome classical proof that requires a mess of δ 's and ϵ 's.

Homeomorphism: Making a donut into a coffee cup You might heard the expression that to a topologist, a donut and a coffee cup appear the same². 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 a *isomorphism*. In topology, the most basic equivalence is based on homeomorphism, which allows spaces that appear quite different in most other subjects to be declared equivalent in topology. A donut and coffeecup (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 (1-1 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 is denoted by $X \cong Y$. 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 4.1.5 (Interval homeomorphisms) Any open interval of \mathbb{R} is homeomorphic to any other 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, $X \subset \mathbb{R}^n$ that 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 $x \rightarrow 1/x$ establishes that $(0, 1)$ and $(1, \infty)$ are homeomorphic. The mapping $x \rightarrow \tan^{-1} x$ establishes that $(-\pi/2, \pi/2)$ and all of \mathbb{R} are homeomorphic! ■

Example 4.1.6 (Topological graphs) Let X be a topological space. The previous example can be extended nicely to make homeomorphisms look like graph

²I also heard a vulgar version (from a mathematician) about topologists not knowing their ... from a hole in the ground.

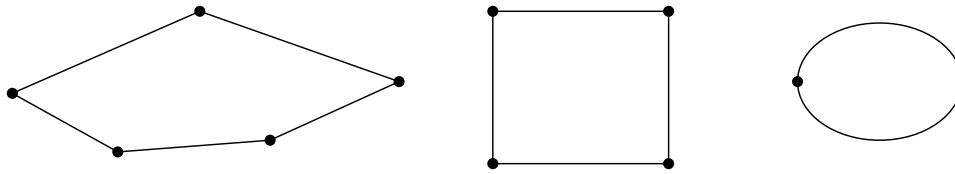


Figure 4.2: Even though the graphs are not isomorphic, the corresponding topological spaces may be homeomorphic due to useless vertices. The example graphs map into \mathbb{R}^2 and are all homeomorphic to a circle.

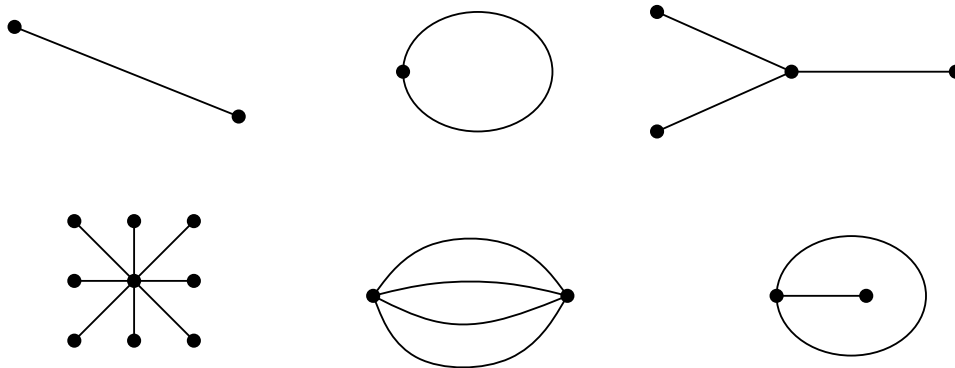


Figure 4.3: The following topological graphs map into subsets of \mathbb{R}^2 that are not homeomorphic to each other.

isomorphisms. Let a *topological graph*³ 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, $\tau : [0, 1] \rightarrow X$. 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, $f : V_1 \rightarrow V_2$ such that if there is an edge between v_1 and v'_1 in G_1 , then 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 will 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.⁴ What if the graphs are not isomorphic?

³In topology this is called a 1-complex [317].

⁴Technically, the images of the topological graphs, as subspaces of X , are homeomorphic, not the graphs themselves.

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, 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*⁵ 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, the space behaves like \mathbb{R}^n ; we can move a small amount in any direction. Several simple examples that may or may not be manifolds are shown in Figure 4.4.

One natural consequence will be that $m \geq n$. According to Whitney’s theorem [], $m \leq 2n$. In other words, \mathbb{R}^{2n} is “big enough” to hold any n -dimensional manifold. 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 boundary points of M are homeomorphic to half spaces of dimension n , which were defined for \mathbb{R}^2 and \mathbb{R}^3 in Section 3.1, and the interior points must be homeomorphic to \mathbb{R}^n .

⁵Manifolds 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 $M \subseteq \mathbb{R}^n$; thus, it avoids these extra complications and is still general enough for our purposes. Some authors use the term *manifold* to refer to a *differentiable manifold*. This requires the definition of an atlas of charts and the homeomorphism is replaced by diffeomorphism. This extra structure is not needed here, but will be introduced when it is needed in Chapter 13.

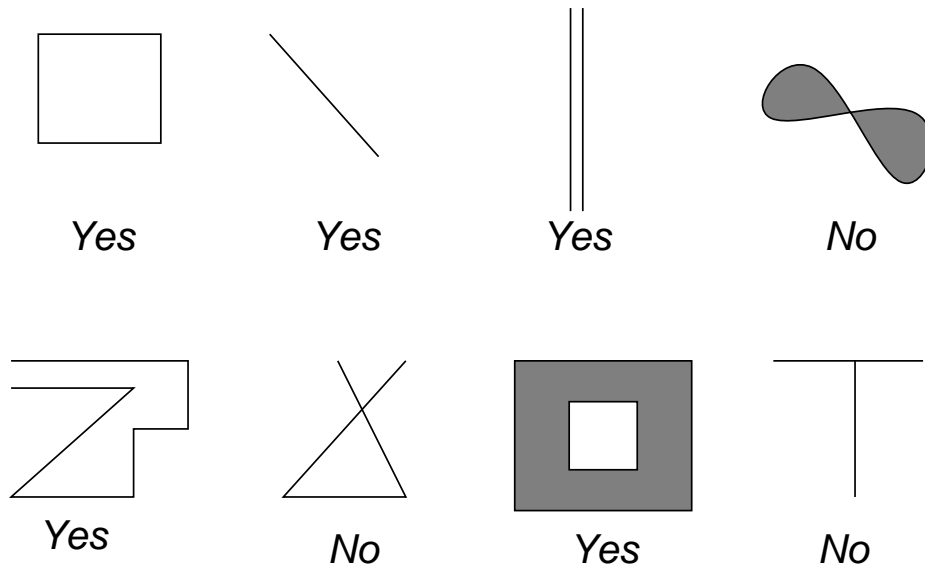


Figure 4.4: Some subsets of \mathbb{R}^2 that may or may not be manifolds.

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 manifolds will be considered equivalent if they are homeomorphic (recall the donut and coffee cup).

Cartesian products The Cartesian product provides 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) exists 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 . No other open sets appear in $X \times Y$; therefore, its open sets are automatically determined.

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, interesting manifolds will be constructed via Cartesian products.

One-dimensional manifolds \mathbb{R} is the most obvious example of a one-dimensional manifold because \mathbb{R} certainly looks like \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 4.1.5).

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

$$\mathbb{S}^1 = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}. \quad (4.4)$$

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 is homeomorphic to a circle that we learned about in high school geometry. Also, when referring to \mathbb{R} , we might instead substitute $(0, 1)$ without any trouble.

Another way to represent \mathbb{S}^1 will be given by *identification*, which is a general method of declaring that some points of a space are identical, although 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 are redefined by directly applying the identification to their elements. Using identification, \mathbb{S}^1 can be defined as $[0, 1]/\sim$, in which the identification declares is 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, just 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 “wrap around” to 0 is a convenient way to represent \mathbb{S}^1 because it does not need to be curved as in (4.4).

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 states that any n -dimensional manifold can be embedded in \mathbb{R}^{2n} [317]. The identifications just cut down on the number of dimensions that are needed for visualization. They are also convenient in the implementation of motion planning algorithms.

Two-dimensional manifolds A variety of interesting, two-dimensional manifolds can be defined by applying the Cartesian product to one-dimensional manifolds. The two-dimensional 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 outer shell of a donut).

Can any other two-dimensional 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 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.

⁶This is usually defined more formally and called a *quotient topology*.

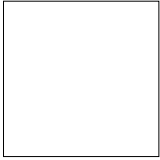
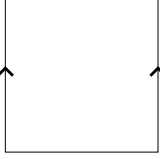
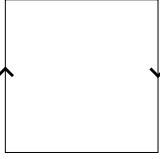
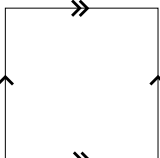
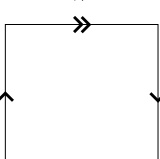
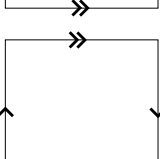
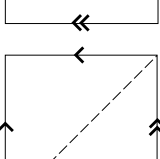
| Identification | Name | Notation |
|---|------------------|----------------------------------|
|  | Plane | \mathbb{R}^2 |
|  | Cylinder | $\mathbb{R} \times \mathbb{S}^1$ |
|  | Möbius band | |
|  | Torus | T^2 |
|  | Klein bottle | |
|  | Projective plane | $\mathbb{R}P^2$ |
|  | Two-sphere | \mathbb{S}^2 |

Figure 4.5: Some common two-dimensional manifolds.

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 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 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 the 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 the 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 two-dimensional 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 . The 3D version, \mathbb{RP}^3 , happens to be one of the most important manifolds for motion planning!

One extremely important two-dimensional manifold remains to be defined. Let \mathbb{S}^2 denote the sphere, which can be easily defined as

$$\mathbb{S}^2 = \{(x, y, z) \in \mathbb{R}^3 \mid x^2 + y^2 + z^2 = 1\}. \quad (4.5)$$

Another way to define \mathbb{S}^2 is by making the identifications shown in the last line 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.

Higher-dimensional manifolds The construction techniques used for the two-dimensional manifolds generalize nicely to higher dimensions. Of course, \mathbb{R}^n , is an n -dimensional manifold. An n -dimensional torus, 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 T^n is used for $(\mathbb{S}^1)^n$. Different kinds of n -dimensional cylinders can be made by forming a Cartesian product $\mathbb{R}^i \times T^j$ for integers i and j such that $i + j = n$. Higher dimensional spheres can be defined as

$$\mathbb{S}^n = \{x \in \mathbb{R}^{n+1} \mid \|x\| = 1\}, \quad (4.6)$$

in which $\|x\|$ denotes the Euclidean norm of x .

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 flat real projective space can be defined in this way, for example. *Lens spaces* are an interesting family manifolds that can be constructed in by identification of polyhedral faces [662].

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 (4.6), note that each one 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 . They are also unique for each line. If we identify all pairs of antipodal points of \mathbb{S}^n , a continuous bijection can be defined between each line in \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 this 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{R}^2 and \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. This also 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

At the core of motion planning is determining whether one part of reachable from another. In Chapter 2, one part of the space was reached from another by applying a sequence of actions. For a continuous state space, we would need a continuum of actions. The application of the continuum of actions produces a path in the state space. This will be formalized in Part IV, but the short explanation is that the path is obtained through the integration of a vector field that is derived from the plan. Here now consider the effect of a plan, which is the continuum of states visited. Therefore, the notion of a continuous path will become very important.

Paths Let X be a topological space, which for our purposes will also be a manifold. A *path*, τ , in X is a continuous function, $\tau : [0, 1] \rightarrow X$. Other intervals of \mathbb{R} may alternatively be used for the domain of τ . Note 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 in that case, a countable set of stages was defined, and the states visited could be represented as x_1, x_2, \dots . In the current setting $\tau(s)$ is used, in which s replaces the stage index. To make 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:

$$X = \{(x, y) \in \mathbb{R}^2 \mid x = 0 \text{ or } y = \sin(1/x)\}. \quad (4.7)$$

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 $\tau(0) = x_1$ 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 (4.7) can happen⁷, and the definitions of connected and path connected coincide []. 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 4.6.a.

Two paths τ_1 and τ_2 are called *homotopic* (with endpoints fixed) if there exists a continuous function $h : [0, 1] \times [0, 1] \rightarrow X$ such that the following four conditions

⁷The 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} .

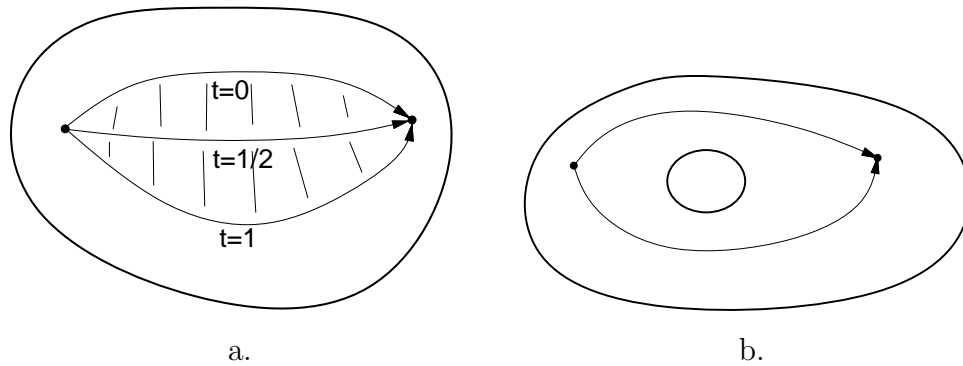


Figure 4.6: a) Homotopy continuously warps one path into another. b) The image of the path cannot be continuously warped over a hole in \mathbb{R}^2 because it causes a discontinuity. In this case, the two paths are not homotopic.

are met:

$$h(s, 0) = \tau_1(s) \quad \text{for all } s \in [0, 1], \quad (4.8)$$

$$h(s, 1) = \tau_2(s) \quad \text{for all } s \in [0, 1], \quad (4.9)$$

$$h(0, t) = h(0, 0) \quad \text{for all } t \in [0, 1], \quad (4.10)$$

and

$$h(1, t) = h(1, 0) \quad \text{for all } t \in [0, 1]. \quad (4.11)$$

The parameter t can be interpreted as a knob that is turned to gradually deform the path from τ_1 into τ_2 . The value $t = 0$ yields τ_1 and $t = 1$ yields τ_2 .

During the warping process, the path image will not be allowed to jump over certain kinds of holes, such as the one shown in Figure 4.6.b. 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 easily go around the air bubbles, but it will not be able to 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 surfaces, as opposed to paths.

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 will be called *simply-connected*. Otherwise, X will be called *multiply-connected*. The case of multiply-connected spaces is very interesting. SAY SOMETHING ABOUT CONTRACTIBLE SPACES?

The fundamental group 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, a brief overview is given here.

At the highest level of abstraction, the task is often considered as a mapping between the category of all topological spaces and a category of some algebraic objects, such as all groups. The fundamental group is the simplest of these mappings to explain. It is often denoted as $\pi_1(X)$, which is the *fundamental group* (first homotopy group) associated with a topological space, X . Let a (continuous) path for which $f(0) = f(1)$ be called a *loop*. Let some $x_t \in X$ be designated as a *base point*. For some arbitrary but fixed based point, x_t , consider the set of all loops such that $f(0) = f(1) = x_t$. 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

$$\tau(t) = \begin{cases} \tau_1(2t) & \text{if } t \in [0, 1/2) \\ \tau_2(2t - 1) & \text{if } t \in [1/2, 1] \end{cases} \quad (4.12)$$

This results in a continuous loop path because τ_1 always terminates at x_t , and τ_2 always begins at x_t . 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_t . 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 paths from two equivalence classes are combined 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. The groups that result provide an interesting characterization of the connectivity of a topological space. Since these groups are based on paths, there is a nice connection to motion planning.

Example 4.1.7 (A simply-connected space) Suppose that a topological space, X , is simply connected. In this case, all loop paths from a based point, x_t , are homotopic, resulting in one equivalence class. The result is $\pi_1(X) = \mathbf{1}_G$, which just contains the identity element. ■

Example 4.1.8 (The circle) Suppose $X = \mathbb{S}^1$. In this case, there is an equivalence class for each $i \in \mathbb{Z}$, the set of integers. Here is one possible definition. If $i > 0$, then it means that the path winds i times around \mathbb{S}^1 in the counterclockwise direction and then returns to the x_t . 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 the base point. The fundamental group is \mathbb{Z} , with respect to the

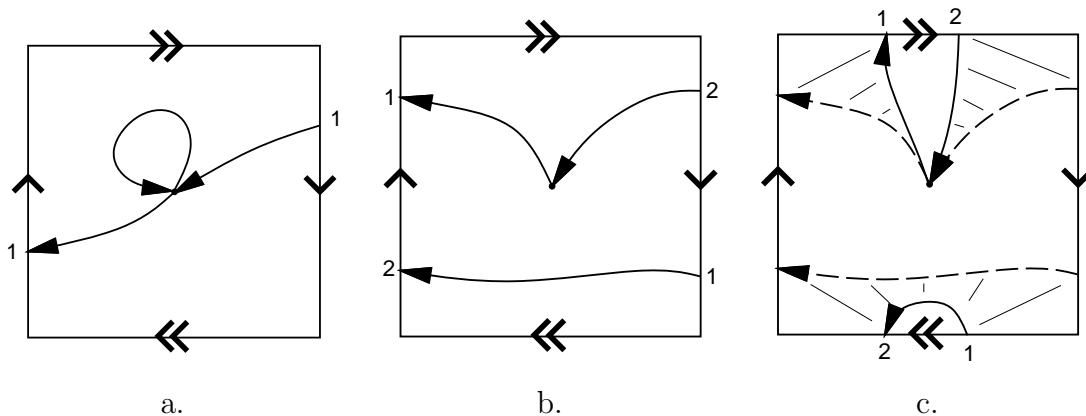


Figure 4.7: An illustration of why $\pi_1(\mathbb{RP}^2) = \mathbb{Z}_2$. (a) Two paths are shown that are not equivalent. The integers 1 and 2 indicate precisely where the path continues when it reaches the boundary. (b) A path that winds around twice. (c) This is homotopic to a loop path that does not wind around at all, as shown in a. Eventually, the part of the path that appears at the bottom is pulled through the top.

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. Think about additive inverses. If a path travels 7 times around \mathbb{S}^1 , and it is combined with a path that travels 7 times in the opposite direction, the result will be homotopic to a path that never leaves the base point. Thus, $\pi_1(\mathbb{S}^1) = \mathbb{Z}$. ■

Example 4.1.9 (The torus) For the torus, $\pi_1(T^n) = \mathbb{Z}^n$, which the i^{th} component of \mathbb{Z}^n corresponds to the number of times a loop path wraps around the i^{th} component of T^n . This makes intuitive sense since T^n is just the Cartesian product of n circles. The fundamental group \mathbb{Z}^n will be obtained if we start with a simply connected subset of the plane and drill out n disjoint, bounded holes. This situation arises frequently in motion planning when a mobile robot must avoid colliding with n disjoint obstacles. ■

By now it seems that the fundamental group simply keeps track of how many times a path loops around holes. This next example yields some very bizarre behavior that helps illustrate some of the interesting structure that arises in algebraic topology.

Example 4.1.10 (\mathbb{RP}^2) Suppose $X = \mathbb{RP}^2$, the projective plane. In this case, there are only two equivalence classes. 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 windings around \mathbb{RP}^2 . The fundamental group is the same for \mathbb{RP}^3 , which will be seen in Section 4.2.2 to be homeomorphic to the set of 3D rotations.

Thus, there are surprisingly only two path classes for the set of 3D rotations. ■

Unfortunately, even if two topological spaces are not homeomorphic, their fundamental groups may be identical. 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 care 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 0, 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. The fundamental group will also neglect bubbles in \mathbb{R}^3 because the homotopy can warp paths around them. (Note that this space is even considered simply connected by our definition.) This last problem 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 will wrap around bubbles in \mathbb{R}^3 , producing a different group. 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 this that a space, X , is *contractible*, which means a homotopy can be constructed that shrinks X to a point [317]. In many motion planning contexts, this notion may be a preferable substitute for simply-connected.

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 is much more powerful for proving topology theorems. 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. For each robot a set of transformations can be made. If the robot has n de-

degrees of freedom, this leads to a manifold of dimension n called the *configuration space* or *C-space*. It will be generally denoted by \mathcal{C} . In the context of this book, the configuration space may be considered as a special form of state space. To solve a motion planning problem, algorithms must conduct a search in this space. The configuration space notion provides a powerful abstraction that converts the complicated models and transformations of Chapter 3 into the general problem of computing a path in 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 this section 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 matrix, T , given by (3.30). 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 a more detailed technical argument will be given. 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.

Groups The first step is to consider the set of transformations as a group, in addition to a topological space.⁸ A *group* is a set, G , together with a binary operation, \circ , such that the *group axioms* are satisfied:

1. (**Closure**) For any $a, b \in G$, the product $x \circ y \in G$.
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 $a \in G$, $e \circ a = e$ and $a \circ e = e$.
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 $a^{-1} \circ a = e$.

⁸The groups considered in this section are actually Lie groups because they are differentiable manifolds. We will not use the name here, however, because the notion of a differentiable structure was not defined. Readers familiar with Lie groups, however, will recognize most of the coming concepts.

Here are some simple examples. 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).

Matrix groups Groups will now be derived 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 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 must satisfy the group axioms. By constructing subgroups, we will arrive at the set of rotation matrices. 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 . Note that matrices will have orthogonal columns (the inner product of any pair is zero) and the determinant will be 1 or -1 . This can be seen by observing that AA^T takes the inner product of every pair of columns. If the columns are different, the result must 0; if they are the same, the result is 1 because the $AA^T = I$. 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”. These can also be considered as topological spaces. 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 there are n^2 entries in the matrix that can be independently chosen. For $GL(n)$, singular matrices are removed, but a n^2 -dimensional manifold is still 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 4.2.1 It is helpful to illustrate the concepts for $n = 2$. The set of all 2×2 matrices may be denoted by

$$\left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \mid \text{for which } a, b, c, d \in \mathbb{R} \right\}, \quad (4.13)$$

and 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 four dimensional manifold.

Next, the constraint $AA^T = I$ is enforced to obtain $O(2)$. This becomes

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} a & c \\ b & d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad (4.14)$$

which directly yields four algebraic equations

$$a^2 + b^2 = 1 \quad (4.15)$$

$$ac + bd = 0 \quad (4.16)$$

$$ca + db = 0 \quad (4.17)$$

$$c^2 + d^2 = 1. \quad (4.18)$$

There are two kinds of equations. There is $\binom{n}{2} = 1$ equation, (4.16), that forces the inner product of the columns to be 0. There are $n = 2$ other constraints, (4.15) and (4.18), which force the columns to be unit vectors. The resulting dimension of the manifold is $\binom{n}{2} = 1$ because we started with \mathbb{R}^4 and lost three dimensions from (4.15), (4.16), and (4.18). 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 b 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, (3.25), 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

$$\left\{ \begin{pmatrix} R & v \\ 0 & 1 \end{pmatrix} \mid \text{for which } R \in SO(n) \text{ and } v \in \mathbb{R}^n \right\}. \quad (4.19)$$

This should look like a generalization of (3.44) and (3.48), 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 at the beginning of this section. Thus, the C-space is

$$\mathcal{C} \cong \mathbb{R}^2 \times \mathbb{S}^1 \quad (4.20)$$

for the case of an unconstrained rigid body.

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 three-dimensional manifold; however, there is a nice interpretation using identification. Start with the open unit cube, $(0, 1)^3 \subset \mathbb{R}^3$. Add in the boundary points of the form $(x, y, 0)$, 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 an “edge” to the configuration space; however, traveling in the Z direction will cause a wraparound.

It is very important for a motion planning algorithm to understand this 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 $\mathcal{C} \cong \mathbb{R} \times \mathbb{S}^1$, 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 obstacle is determining the topology of $SO(3)$. At least we do not have to go beyond $n = 3$ 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

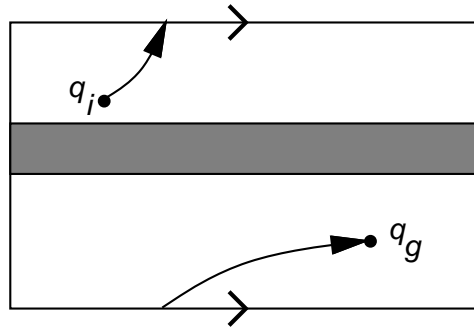


Figure 4.8: A planning algorithm may have to cross the identification boundary to find a solution path.

C-space will be a six-dimensional manifold, $\mathcal{C} \cong \mathbb{R}^3 \times \mathbb{RP}^3$. Three dimensions come from translation and three more 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 were very convenient for visualization, performing transformations in software, and also for deriving the DH parameters. However, these were concerned with 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 special 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 3 independent equations that force the magnitude of each column to be 1. This means that $O(3)$ will have 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 4.2.1 that $SO(2)$ consists of two circles, and the constraint $\det A = 1$ selects one of them. In the case of $O(3)$, there will be 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 represent 3D rotations, consider using complex numbers to represent 2D

rotations. Let the term *unit complex number* refer to any complex number, $a + bi$ for which $a^2 + b^2 = 1$.

Note that 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 $f : G \rightarrow H$ 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 with G for a while, map the result to H , perform more calculations, and map back to G without any trouble. The groups G and H are just two different representations of the same thing.

This is true of the unit complex numbers and $SO(2)$. 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 (3.25).

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 will be 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 $a_1 + b_1i$ and $a_2 + b_2i$ are combined to obtain $(a_1a_2 - b_1b_2) + (a_1b_2 + a_2b_1)i$. Note that we did not use complex numbers to express the solution to a polynomial equation; we simply borrowed their nice algebraic properties. At any time, a complex number $a + bi$ can be converted into the equivalent rotation matrix

$$R(a, b) = \begin{pmatrix} a & -b \\ b & a \end{pmatrix}. \quad (4.21)$$

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 that 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: $i^2 = j^2 = k^2 = -1$, $ij = k$, $jk = i$, and $ki = j$. Using these, multiplication of two quaternions, $h_1 = a_1 + b_1i + c_1j + d_1k$ and $h_2 = a_2 + b_2i + c_2j + d_2k$, can be derived

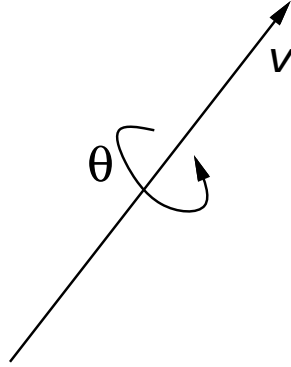


Figure 4.9: Any 3D rotation can be considered as a rotation by an angle θ about the axis given by the unit direction vector $v = [v_1 \ v_2 \ v_3]$.

to obtain $h_1 \cdot h_2 = a_3 + b_3i + c_3j + d_3k$, in which

$$a_3 = a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2 \quad (4.22)$$

$$b_3 = a_1b_2 + a_2b_1 + c_1d_2 - c_2d_1 \quad (4.23)$$

$$c_3 = a_1c_2 + a_2c_1 + b_2d_1 - b_1d_2 \quad (4.24)$$

$$d_3 = a_1d_2 + a_2d_1 + b_1c_2 - b_2c_1. \quad (4.25)$$

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 was 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 $a_1v_2 + a_2v_1 - v_1 \times v_2$.

Just as 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 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

$$R(h) = \begin{pmatrix} 2(a^2 + b^2) - 1 & 2(bc - ad) & 2(bd + ac) \\ 2(bc + ad) & 2(a^2 + c^2) - 1 & 2(cd - ab) \\ 2(bd - ac) & 2(cd + ab) & 2(a^2 + d^2) - 1 \end{pmatrix}, \quad (4.26)$$

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

$$h = \cos \frac{\theta}{2} + v_1 \sin \frac{\theta}{2} i + v_2 \sin \frac{\theta}{2} j + v_3 \sin \frac{\theta}{2} k. \quad (4.27)$$

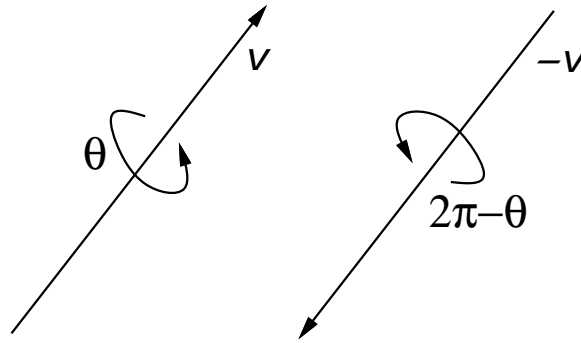


Figure 4.10: There are two ways to encode the same rotation.

Unfortunately, this representation is not unique. It can be verified in (4.26) that $R(h) = R(-h)$. A nice geometric interpretation is given in Figure ???. 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 will be

$$\cos\left(\frac{2\pi - \theta}{2}\right) = \cos\left(\pi - \frac{\theta}{2}\right) = -\cos\left(\frac{\theta}{2}\right) = -a. \quad (4.28)$$

The i , j , and k components will be

$$-v \sin\left(\frac{2\pi - \theta}{2}\right) = -v \sin\left(\pi - \frac{\theta}{2}\right) = -v \sin\left(\frac{\theta}{2}\right) = [-b \quad -c \quad -d]. \quad (4.29)$$

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 (4.26) is two-to-one.

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, then $\mathbb{S}^n / \sim \cong \mathbb{RP}^n$. Hence, $SO(3) \cong \mathbb{RP}^3$, which can be considered as the set of all lines through the original of \mathbb{R}^4 , but this is hard to visualize. An extension of the representation of \mathbb{RP}^2 in Figure 4.5 can be made 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, T^3 would be obtained). For example, in the Z direction, let $(x, y, 0) \sim (1 - x, 1 - y, 1)$ for all $x, y \in [0, 1]$.

A useful way to force uniqueness of rotations is to require staying in the “upper half” of \mathbb{S}^3 . For example, we can 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 we can require that $b \geq 0$. However, if $a = b = 0$, then we must 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 these concepts 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 for the antipodal identification problem for $SO(3)$).

However, one important reason why the quaternion arithmetic was introduced is that the group of unit quaternions 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, (4.26) 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 will be needed. For any $h = a + bi + cj + dk \in \mathbb{H}$, let $h^* = a - bi - cj - dk$. 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 will be new coordinates for the transformed point. It will be 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 (3.35), the yaw, pitch, 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 (4.26).⁹

For a given rotation matrix (3.35), the quaternion parameters, $h = a + bi + cj + dk$ can be computed as follows [155]. The first component is

$$a = \frac{1}{2} \sqrt{r_{11} + r_{22} + r_{33} + 1}, \quad (4.30)$$

⁹Since that function was two-to-one, it is technically not an inverse until the quaternions are restricted to the upper hemisphere, as described previously.

and if $a \neq 0$, then

$$b = \frac{r_{32} - r_{23}}{4a}, \quad (4.31)$$

$$c = \frac{r_{13} - r_{31}}{4a}, \quad (4.32)$$

and

$$d = \frac{r_{21} - r_{12}}{4a}. \quad (4.33)$$

If $a = 0$, then the previously mentioned equator problem occurs. In this case, then

$$b = \frac{r_{13}r_{12}}{\sqrt{r_{12}^2r_{13}^2 + r_{12}^2r_{23}^2 + r_{13}^2r_{23}^2}}, \quad (4.34)$$

$$c = \frac{r_{12}r_{23}}{\sqrt{r_{12}^2r_{13}^2 + r_{12}^2r_{23}^2 + r_{13}^2r_{23}^2}}, \quad (4.35)$$

and

$$d = \frac{r_{13}r_{23}}{\sqrt{r_{12}^2r_{13}^2 + r_{12}^2r_{23}^2 + r_{13}^2r_{23}^2}}. \quad (4.36)$$

This method will fail 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, (3.31), pitch, (3.32), or roll, (3.33), which can be detected in advance.

Special Euclidean group Now that the complicated part of representing $SO(3)$ has been handled, the determination of $SE(3)$ is straightforward. The general form of a matrix in $SE(3)$ is given by (4.19), in which $R \in SO(3)$ and $v \in \mathbb{R}^3$. Since $SO(3) \cong \mathbb{RP}^3$, and the translations are chosen independently, the resulting configuration space for a rigid body that rotates and translates in \mathbb{R}^3 is

$$\mathcal{C} \cong \mathbb{R}^3 \times \mathbb{RP}^3, \quad (4.37)$$

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 configuration spaces can be combined using Cartesian products. Let \mathcal{C}_i denote the configuration 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

$$\mathcal{C} = \mathcal{C}_1 \times \mathcal{C}_2 \times \cdots \times \mathcal{C}_n. \quad (4.38)$$

If the bodies are attached to form a kinematic chain or kinematic tree, then each configuration 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 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 configuration 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 will not be \mathbb{RP}^3 , because part of $SO(3)$ is missing.

Another complication in the process of determining the configuration space is that the DH parameterization of Section 3.3.2 designed to facilitate the assignment of coordinate frames and computation of transformations, but neglects considerations of topology. For example, a common approach to representing a spherical robot wrist is to make three zero-length lengths 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, 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 configuration space is

$$\mathcal{C} \cong \mathbb{S}^1 \times \mathbb{S}^1 \times \cdots \times \mathbb{S}^1 = T^n. \quad (4.39)$$

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 $\mathcal{C} \cong \mathbb{R}^2 \times T^n$. If there are prismatic joints, then each one contributes an \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 an \mathbb{R} . A cylindrical joint contributes an $\mathbb{R} \times \mathbb{S}^1$, unless its rotational motion is restricted. A planar joint contributes $\mathbb{R}^2 \times \mathbb{S}^1$ because any motion $SE(2)$ 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 will rarely occur. 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 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. Section 4.3 removes from \mathcal{C} the configurations that either cause the robot to collide with obstacles or different links of the robot to collide with each other. The removed part is referred to as the obstacle region. The leftover space is precisely where the planning occurs. A motion planning algorithm must find a collision-free path from an initial configuration to a goal configuration. Finally, after the models of Chapter ?? 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 modeled using semi-algebraic primitives (which includes polygonal and polyhedral primitives) 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} , is defined as

$$\mathcal{C}_{obs} = \{q \in \mathcal{C} \mid \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\}, \quad (4.40)$$

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 becomes 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, then \mathcal{C}_{free} must be an open set. This means that in the way the model is defined, the robot can come arbitrarily close to the obstacles and remain in \mathcal{C}_{free} . If \mathcal{A} “touches” \mathcal{O} ,

$$int(\mathcal{O}) \cap int(\mathcal{A}(q)) = \emptyset \text{ and } \mathcal{O} \cap \mathcal{A}(q) \neq \emptyset, \quad (4.41)$$

then $q \in \mathcal{C}_{obs}$. The notion 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. For such extensions, the closure, $cl(\mathcal{C}_{free})$, should 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 (4.40) 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 parts of the robot. Let the robot be modeled as a collection, $\{\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_m\}$,

of m links, which could be attached by joints, or may be unattached. 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 that set of *collision pairs*, in which each *collision pair*, (i, j) , represents a pair of link indices $i, j \in \{1, 2, \dots, m\}$, such that $i \neq j$. If (i, j) appears in P , it means that \mathcal{A}_i and \mathcal{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 usually in contact all of the time because of the joint between them. One common definition for P is that each link must avoid collisions with 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 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 may be added to the definition of \mathcal{C}_{obs} to obtain

$$\mathcal{C}_{obs} = \left\{ \bigcup_{i=1}^m \{q \in \mathcal{C} \mid \mathcal{A}_i(q) \cap \mathcal{O} \neq \emptyset\} \right\} \cup \left\{ \bigcup_{[i,j] \in P} \{q \in \mathcal{C} \mid \mathcal{A}_i(q) \cap \mathcal{A}_j(q) \neq \emptyset\} \right\}. \quad (4.42)$$

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 is neither straightforward nor efficient to construct an explicit boundary or solid representation of either \mathcal{C}_{free} or \mathcal{C}_{obs} .

Formulation 4.3.1 (The Piano Mover's Problem)

1. A *world*, \mathcal{W} , is defined, in which either $\mathcal{W} = \mathbb{R}^2$ or $\mathcal{W} = \mathbb{R}^3$.
2. A semi-algebraic *obstacle region* $\mathcal{O} \subset \mathcal{W}$ is defined 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 links, $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_m$.
4. The *configuration space*, \mathcal{C} , is 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 $q_i \in \mathcal{C}_{free}$ is designated as the *initial configuration*.

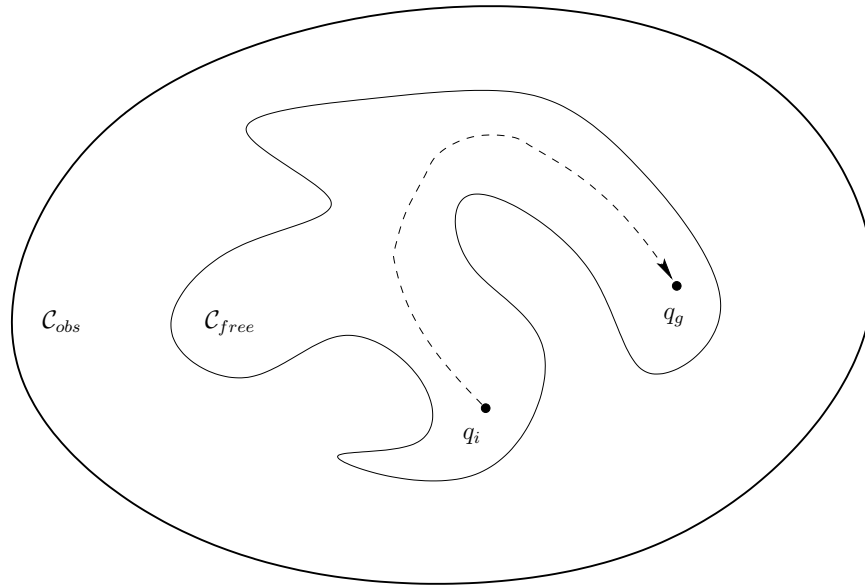


Figure 4.11: The basic motion planning problem is conceptually very simple using the configuration space ideas. The task is to find a path from q_i to q_g in \mathcal{C}_{free} . The entire blob represents $\mathcal{C} = \mathcal{C}_{free} \sqcup \mathcal{C}_{obs}$.

6. A configuration $q_g \in \mathcal{C}_{free}$ is designated as the *goal configuration*.
7. An 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 [651] 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 subsets of $X, Y \subset \mathbb{R}^n$, let their *Minkowski difference*, denoted by \ominus be

$$X \ominus Y = \{x - y \in \mathbb{R}^n \mid x \in X \text{ and } y \in Y\}, \quad (4.43)$$

in which $x - y$ is just vector subtraction on \mathbb{R}^n .

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. The Minkowski difference between

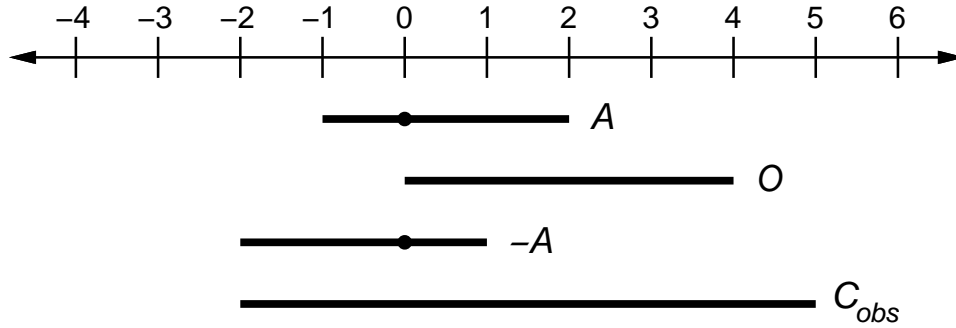


Figure 4.12: A one-dimensional example.

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 , as opposed to subtracting them. The set $-Y$ is obtained by replacing each $y \in Y$ by $-y$. 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 $-\mathcal{A}$, $\mathcal{C}_{obs} = [-2, 4]$.

The Minkowski difference is often considered as a *convolution*. It can even be defined to appear the same as in studied in differential equations and system theory. For the 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 following convolution,

$$h(x) = \int_{-\infty}^{\infty} f(\tau)g(x - \tau)d\tau,$$

will yield a function h of x that is 1 if $x \in \mathcal{C}_{obs}$, and 0 otherwise.

A polygonal C-space obstacle An efficient method of 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} [504]. 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 $\alpha_1, \alpha_2, \dots, \alpha_n$ denote the angles of the inward edge normals in counterclockwise order around \mathcal{A} . Let $\beta_1, \beta_2, \dots, \beta_n$ 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

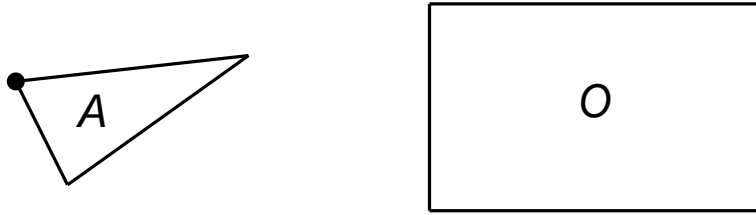


Figure 4.13: A triangular robot and a rectangular obstacle.

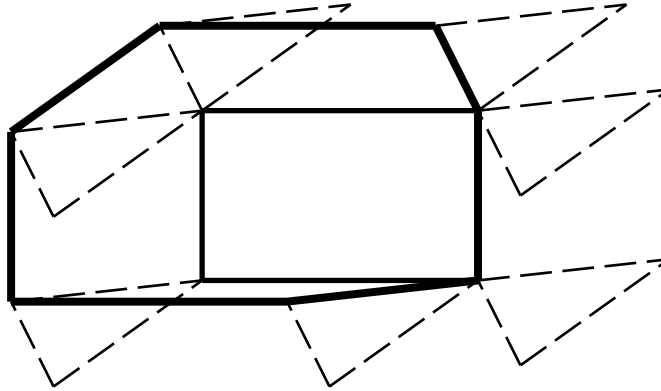


Figure 4.14: Slide the robot around the obstacle while keeping them both in contact.

be constructed incrementally by adding the edges that correspond to the sorted normals, in the order in which they are encountered.

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 of its coordinate frame. Consider sliding the robot around the obstacle in such a way that they are always in contact, as shown in Figure 4.14. This corresponds to the traversal of all of the configurations in $\partial\mathcal{C}_{obs}$. The origin of \mathcal{A} , will trace out the edges of \mathcal{C}_{obs} , as shown in Figure 4.15. There are 7 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 4.16. When sorted as shown in Figure 4.17, 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} ; the 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} .

The previous method quickly identifies each edge that contributes to \mathcal{C}_{obs} . This method can also construct a solid representation \mathcal{C}_{obs} in terms of half planes. This requires defining $n + m$ linear equations (assuming there are no collinear edges).

There are two different ways in which an edge of \mathcal{C}_{obs} is generated, as shown in Figure 4.18 [207, 504]. Type EV contact refers to the case in which an edge

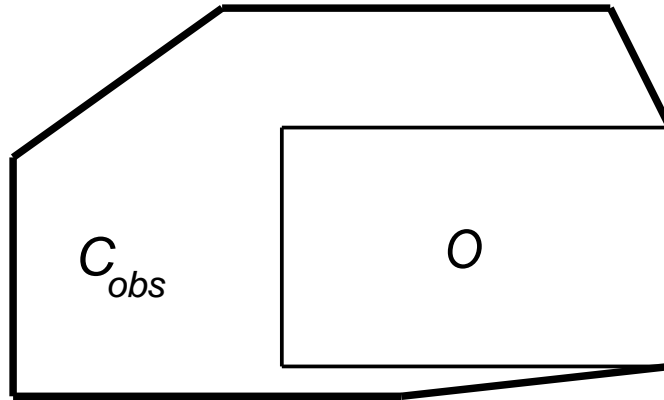


Figure 4.15: The traced out edges of the configuration space obstacle are shown.

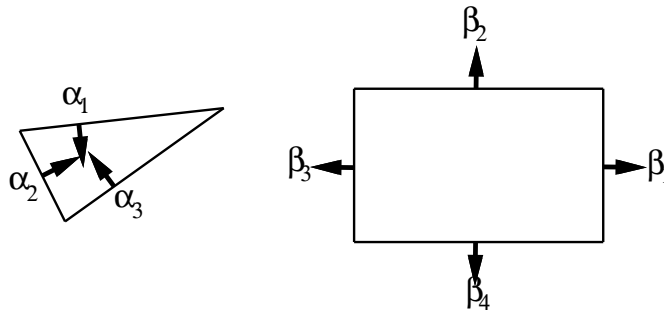


Figure 4.16: The directions of the normals are sorted around the circle.

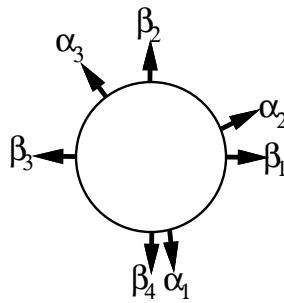


Figure 4.17: The edge normals are sorted by orientation.

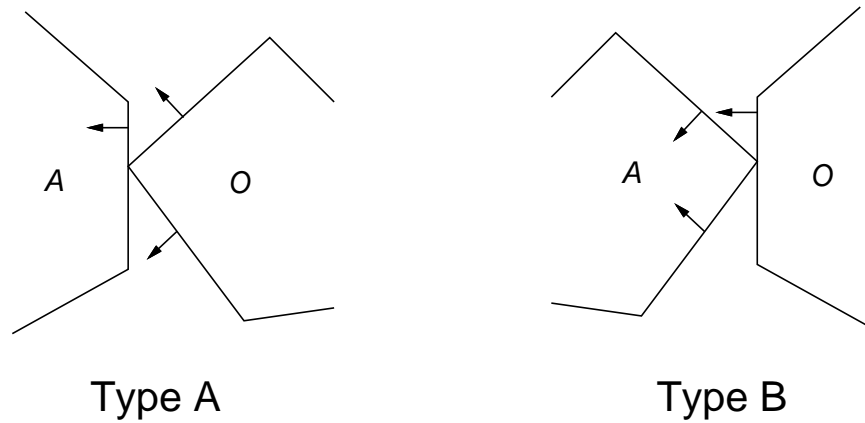


Figure 4.18: Two different types of contact, each of generates a different kind of edge [205, 504].

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 B contact refers to the case in which an edge of \mathcal{A} is in contact with a vertex of \mathcal{O} . This contributes to m edges of \mathcal{C}_{obs} . The relationships between the edge normals are also shown in Figure 4.18. For Type EV, the inward edge normal lies between the outward edge normals of the obstacle edges that share the contact vertex. Likewise for Type B, the outward edge normal of \mathcal{O} lies between the inward edge normals of \mathcal{A} .

Using the ordering shown in Figure 4.17, Type EV contacts occur precisely when an edge normal of \mathcal{A} is encountered, and Type B contacts occur precisely when an edge normal of \mathcal{O} is encountered. The task is to determine a line equation at each of these instances. Consider the case of a Type EV contact; the Type B 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 when the vectors n and v , shown in Figure 4.19 are perpendicular, i.e., $n \cdot v = 0$.

Note that the normal vector, n , does not depend on the configuration of \mathcal{A} because it can only translate. 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, $n \cdot v = 0$ is the equation of a line in \mathcal{C} . For example, if the coordinates of p are $(1, 2)$ when \mathcal{A} is at the origin, then the expression for p at configuration (x_t, y_t) is $(1+x_t, 2+y_t)$. Let $f(x_t, y_t) = n \cdot v$. Let $H = \{(x_t, y_t) \in \mathcal{C} \mid f(x_t, y_t) \leq 0\}$. Observe that the 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 B

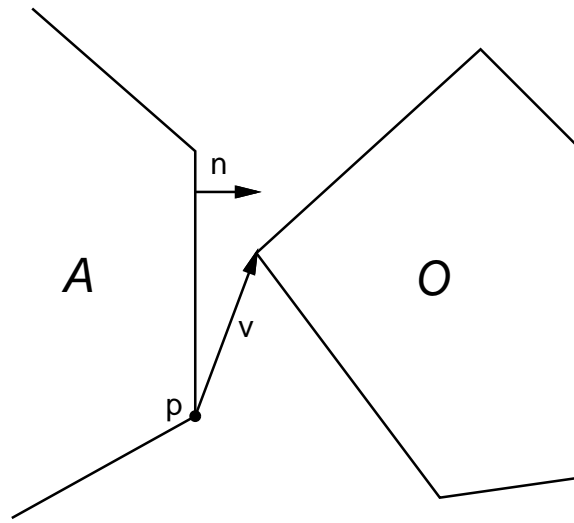


Figure 4.19: Contact occurs when n and v are perpendicular.

contacts. This yields a convex polygon in \mathcal{C} that has $n + m$ sides, as expected.

Example 4.3.1 Consider building a geometric model of \mathcal{C}_{obs} for the example in Figure 4.20. Suppose that the orientation of \mathcal{A} is fixed as shown, and $\mathcal{C} \cong \mathbb{R}^2$. In this case, \mathcal{C}_{obs} will be a convex polygon with seven sides. The contact conditions that occur are shown in Table 4.1. The ordering is given as normals appear as shown in Figure 4.17.

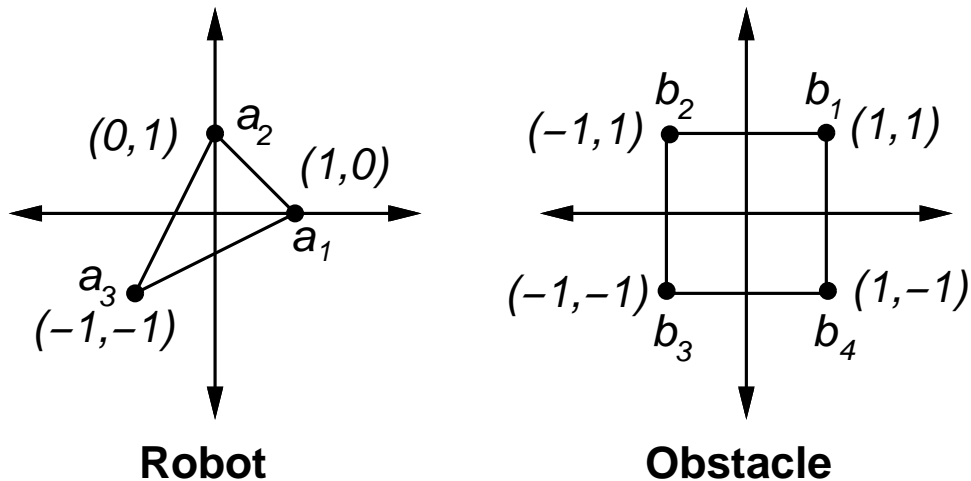


Figure 4.20: Consider constructing the obstacle region for this example.

■

| Type | Vtx. | \mathcal{O} Edge | n | v | Half Plane |
|------|-------|--------------------|-----------|----------------------|---|
| B | a_3 | b_4-b_1 | $[1, 0]$ | $[x_t - 2, y_t]$ | $H_B = \{q \in \mathcal{C} \mid x_t - 2 \leq 0\}$ |
| B | a_3 | b_1-b_2 | $[0, 1]$ | $[x_t - 2, y_t - 2]$ | $H_B = \{q \in \mathcal{C} \mid y_t - 2 \leq 0\}$ |
| A | b_2 | a_3-a_1 | $[1, -2]$ | $[-x_t, 2 - y_t]$ | $H_A = \{q \in \mathcal{C} \mid -x_t + 2y_t - 4 \leq 0\}$ |
| B | a_1 | b_2-b_3 | $[-1, 0]$ | $[2 + x_t, y_t - 1]$ | $H_B = \{q \in \mathcal{C} \mid -x_t - 2 \leq 0\}$ |
| A | b_3 | a_1-a_2 | $[1, 1]$ | $[-1 - x_t, -y_t]$ | $H_A = \{q \in \mathcal{C} \mid -x_t - y_t - 1 \leq 0\}$ |
| B | a_2 | b_3-b_4 | $[0, -1]$ | $[x_t + 1, y_t + 2]$ | $H_B = \{q \in \mathcal{C} \mid -y_t - 2 \leq 0\}$ |
| A | b_4 | a_2-a_3 | $[-2, 1]$ | $[2 - x_t, -y_t]$ | $H_A = \{q \in \mathcal{C} \mid 2x_t - y_t - 4 \leq 0\}$ |

Table 4.1: The various contact conditions are shown in the order as normals appear in Figure 4.17.

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.

There are three different kinds of contacts that lead to half spaces:

- Type FV: A face of \mathcal{A} and a vertex of \mathcal{O}
- Type VF: A vertex of \mathcal{A} and a face of \mathcal{O}
- Type EE: An edge of \mathcal{A} and an edge of \mathcal{O}

Each half space defines a face of the polyhedron. The resulting polyhedron 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 \square .

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, and after applying any of the transformations of Sections 3.2 to 3.4. It might not be true for other kinds of transformations, such as parameters that warp a flexible material [?].

Consider the case of a convex polygonal robot and a convex polygonal obstacle in a 2D world. Any transformation in $SE(2)$ may be applied to \mathcal{A} ; thus, $\mathcal{C} \cong \mathbb{R}^2 \times \mathbb{S}^1$ 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 B contacts. We will describe how to construct the algebraic primitives for the Type EV contacts; Type B can be handled in a similar manner.

For the translation-only case, we were able to determine all of the Type EV conditions 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

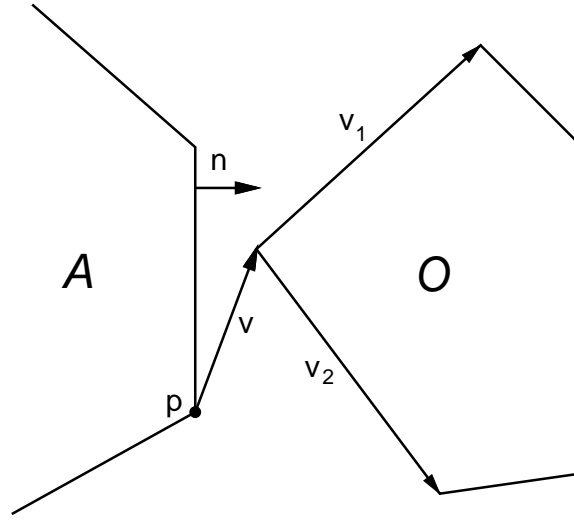


Figure 4.21: An illustration to help in constructing the obstacle representation in the C-space.

depends on θ . Recall the constraint that the inward normal of \mathcal{A} must lie between the outward normals of the edges of \mathcal{O} that contain the vertex of contact. See 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 $\frac{\pi}{2}$. 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 depends on q . For any $q \in \mathcal{C}$, if $n(q) \cdot v_1 \geq 0$, $n(q) \cdot v_2 \geq 0$, and $n(q) \cdot v(q) > 0$, then $q \in \mathcal{C}_{free}$. Let H_f denote the set of configurations that satisfy these conditions. These conditions can be used to determine whether a point is in \mathcal{C}_{free} ; however, it is not a complete characterization of \mathcal{C}_{free} ; any other Type EV and Type B contacts could add more points to \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} , i.e., $\mathcal{C}_{obs} \subset \mathcal{C} \setminus H_f$. Let $H_A = \mathcal{C} \setminus H_f$. Let the following primitives,

$$H_1 = \{q \in \mathcal{C} \mid n(q) \cdot v_1 \leq 0\}, \quad (4.44)$$

$$H_2 = \{q \in \mathcal{C} \mid n(q) \cdot v_2 \leq 0\}, \quad (4.45)$$

and

$$H_3 = \{q \in \mathcal{C} \mid n(q) \cdot v(q) \leq 0\}, \quad (4.46)$$

define $H_A = H_1 \cup H_2 \cup H_3$.

It is known that $\mathcal{C}_{obs} \subseteq H_A$, but H_A may still overlap with \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 B contact, the result will

be \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 lie in \mathcal{C}_{obs} . For any Type EV constraint, $(H_1 \cup H_2) \setminus H_3 \subseteq \mathcal{C}_{free}$. A similar statement can be made for Type B constraints. A logical predicate, similar to that defined in Section 3.1.1, can be constructed to detect whether or not $q \in \mathcal{C}_{obs}$ in time that is linear in the number of \mathcal{C}_{obs} primitives.

One important issue remains. The expression $n(q)$ 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, thus transforming polynomials into polynomials. Such a substitution can be made using stereographic projection [437]; however, a simpler substitution can be made using complex numbers to represent rotation. Recall that when $a + bi$ is used to represent rotation, each rotation matrix in $SO(2)$ is represented as (4.21), and the 3×3 homogeneous transformation matrix becomes

$$T(a, b, x_t, y_t) = \begin{pmatrix} a & -b & x_t \\ b & a & y_t \\ 0 & 0 & 1 \end{pmatrix}. \quad (4.47)$$

Using this matrix to transform a point $[x \ y \ 1]$ results in the point coordinates $(ax - by + x_0, bx + ay + y_0)$. Thus, any transformed point on \mathcal{A} will be 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 will be added to constrain the angles to lie in \mathbb{S}^1 .

By using complex numbers, primitives in \mathbb{R}^4 are obtained for each Type EV and Type B contact. By defining $\mathcal{C} = \mathbb{R}^4$, the following algebraic primitives are obtained for a Type EV contact:

$$H_1 = \{(x_t, y_t, a, b) \in \mathcal{C} \mid n(x_t, y_t, a, b) \cdot v_1 \leq 0\}, \quad (4.48)$$

$$H_2 = \{(x_t, y_t, a, b) \in \mathcal{C} \mid n(x_t, y_t, a, b) \cdot v_2 \leq 0\}, \quad (4.49)$$

and

$$H_3 = \{(x_t, y_t, a, b) \in \mathcal{C} \mid n(x_t, y_t, a, b) \cdot v(x_t, y_t, a, b) \leq 0\}. \quad (4.50)$$

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

$$H_s = \{(x_t, y_t, a, b) \in \mathcal{C} \mid a^2 + b^2 - 1 = 0\} \quad (4.51)$$

is intersected with H_A . This constraint preserves the topology of the original configuration space. The set H_s remains fixed over all Type EV and Type B contacts; therefore, it only needs to be considered once.

Example 4.3.2 Consider adding rotation to the model considered in Example 4.3.1. In this case, all possible contacts must be considered. For this example, there are 12 Type EV contacts and 12 Type B 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 B 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 that share 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, (3.30), 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 (4.26) avoids all trigonometric functions in the same way that (4.21) avoided them for $SO(2)$. Unfortunately, (4.26) is not linear in the configuration variables, as it was for (4.21), but it is at least polynomial. This enables semi-algebraic models to be formed for \mathcal{C}_{obs} . Recall that there will be Type FV, VF, and EE contacts for case of $SE(3)$. From all of the contact conditions, polynomials that correspond to each patch of \mathcal{C}_{obs} can be made. Note that these patches will be polynomials in seven variables: $x_t, y_t, z_t, a, b, c, d$. Once again, a special primitive must be intersected with all others to enforce the constraint that unit quaternions are used. This reduces the dimension from 7 back down to 6. Also, constraints may be added to throw away half of \mathbb{S}^3 , which is redundant because of the identification.

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 (3.45) and (3.49) 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 will consist 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 4.3.1 and 4.3.2, you can imagine that no human would ever want to do all of that by hand, but at least it can be automated. It is also very important for the existence of theoretical algorithms that solve the motion planning problem combinatorially.

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. These 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 when imposing the constraints that all complex numbers must have unit magnitude.

4.4 Kinematic Closure and Varieties

This section continues where the discussion at the end of Section 3.4 finished. Suppose that a collection of links are arranged in a way that forms loops. In this case, the configuration 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 (3.72) and Figure 3.27, 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 NO. However, the configuration 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 will be a system of polynomials whose solution set represents the configuration 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.¹⁰

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 [369] that for every closed, bounded real algebraic variety that can be embedded in \mathbb{R}^n , there exists a linkage whose configuration space is homeomorphic to it. This difficulty implies that most of the time, motion planning algorithms need to manipulate implicit polynomials when searching the space. For the algebraic methods of Section 6.4.2, this will not pose any conceptual difficulty because they methods

¹⁰This is called a Whitney stratification [123, 772]

already work directly with polynomials. Sampling-based methods usually rely on being able to 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 them it may be helpful to 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 six axioms below.

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 $a + b = 0$.
6. (**Multiplicative Inverses:**) For every $a \in \mathbb{F}$, 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, \dots, x_n . A *monomial* over a field, \mathbb{F} , is a product of the form

$$x_1^{d_1} \cdot x_2^{d_2} \cdots x_n^{d_n}, \quad (4.52)$$

in which all of the exponents d_1, d_2, \dots, d_n are positive integers. The *total degree* of the monomial is $d_1 + \cdots + d_n$.

A *polynomial* f in x_1, \dots, x_n with coefficients in \mathbb{F} finite linear combination of monomials that have coefficients in \mathbb{F} . A polynomial can be expressed as

$$\sum_{i=1}^m c_i m_i, \quad (4.53)$$

in which m_i is a monomial as shown in (4.52) 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, \dots, x_n with coefficients in \mathbb{F} is denoted by $\mathbb{F}[x_1, \dots, x_n]$.

Example 4.4.1 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

$$x_1^4 - \frac{1}{2}x_1x_2x_3^3 + x_1^2x_2^2 + 4. \quad (4.54)$$

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 (4.54) is 5 due to the second term. An equivalent polynomial may be written using nicer variables. Using x, y, z as variables yields

$$x^4 - \frac{1}{2}xyz^3 + x^2y^2 + 4, \quad (4.55)$$

which belongs to $\mathbb{Q}[x, y, z]$. ■

The set, $\mathbb{F}[x_1, \dots, x_n]$, 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 $\mathbb{F}[x_1, \dots, x_n]$ 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 requires 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

$$\mathbb{F}^n = \{(c_1, \dots, c_n) \mid c_1, \dots, c_n \in \mathbb{F}\}. \quad (4.56)$$

For our purposes in this section, an affine space can be considered as a vector space (the exact definition appears in []). \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 $\mathbb{F}[x_1, \dots, x_n]$ can be converted into function

$$f : \mathbb{F}^n \rightarrow \mathbb{F}, \quad (4.57)$$

by substituting elements of \mathbb{F} for each variable, and evaluating the expression using the field operations. This can be written as $f(a_1, \dots, a_n) \in \mathbb{F}$, 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 $f(x_1, \dots, x_n) = 0$? We could also ask the question for some nonzero element, but notice that this is not necessary because the polynomial may be redefined for formulate the question with 0. For example, what are the elements of \mathbb{R}^2 such that $x^2 + y^1 = 1$? This familiar equation for \mathbb{S}^1 can be reformulated as 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, \dots, f_k be polynomials in $\mathbb{F}[x_1, \dots, x_n]$. The set

$$V(f_1, \dots, f_k) = \{(a_1, \dots, a_n) \in \mathbb{F}^n \mid f_i(a_1, \dots, a_n) = 0 \text{ for all } 1 \leq i \leq k\}, \quad (4.58)$$

is called the (*affine*) *variety* defined by f_1, \dots, 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 notice that for varieties only equations of the form $f = 0$ are allowed. Consider the varieties $V(f_1, \dots, f_k)$ and $V(g_1, \dots, g_l)$. Their intersection is given by

$$V(f_1, \dots, f_k) \cap V(g_1, \dots, g_l) = V(f_1, \dots, f_k, g_1, \dots, g_l), \quad (4.59)$$

because each element of \mathbb{F}^n must be produce a 0 value for each of the polynomials $f_1, \dots, f_k, g_1, \dots, g_l$.

To obtain unions, the polynomials simply need to be multiplied. For example, consider the varieties $V_1, V_2 \subset \mathbb{F}$ defined as

$$V_1 = \{(a_1, \dots, a_n) \in \mathbb{F}^n \mid f_1(a_1, \dots, a_n) = 0\} \quad (4.60)$$

and

$$V_2 = \{(a_1, \dots, a_n) \in \mathbb{F}^n \mid f_2(a_1, \dots, a_n) = 0\}. \quad (4.61)$$

The set $V_1 \cup V_2 \subset \mathbb{F}^n$ is obtained by forming the polynomial $f = f_1 f_2$. Note that $f(a_1, \dots, a_n) = 0$ if either $f_1(a_1, \dots, a_n) = 0$ or $f_2(a_1, \dots, a_n) = 0$. 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 list of polynomials in $V(\cdot)$ 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, $\mathcal{A}_1, \dots, \mathcal{A}_n$, as considered in Example 3.3.1 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 configuration space

$$\mathcal{C} \cong \mathbb{S}^1 \times \mathbb{S}^1 \times \dots \times \mathbb{S}^1 = T^n, \quad (4.62)$$

the n -dimensional torus

Two links If there are three links, $\mathcal{A}_1, \mathcal{A}_2$, and \mathcal{A}_3 , then the configuration space can be nicely 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. This yields $a_1 = a_2 = 1$. A point, $(x, y) \in \mathcal{A}^3$ is transformed as

$$\begin{pmatrix} \cos \theta_1 & -\sin \theta_1 & 0 \\ \sin \theta_1 & \cos \theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos \theta_2 & -\sin \theta_2 & 1 \\ \sin \theta_2 & \cos \theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \quad (4.63)$$

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

$$\begin{pmatrix} a_1 & -b_1 & 0 \\ b_1 & a_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_2 & -b_2 & 1 \\ b_2 & a_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad (4.64)$$

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 it 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 (4.64) yields the polynomials, $f_1, f_2 \in \mathcal{R}[a_1, b_1, a_2, b_2]$,

$$f_1 = xa_1a_2 - ya_1b_2 - xb_1b_2 + ya_2b_1 + a_1 \quad (4.65)$$

and

$$f_2 = -ya_1a_2 + xa_1b_2 + xa_2b_1 - yb_1b_2 + b_1, \quad (4.66)$$

for the X and Y coordinates, respectively. Note that the polynomial variables are configuration parameters, not x and y . For a given point (x, y) in \mathcal{A}_2 , all coefficients are determined.

Now a kinematic closure constraint will be imposed. Fix the point $(1, 0)$ in \mathcal{A}_2 at $(1, 1)$ in \mathcal{W} . This yields the constraints

$$f_1 = a_1a_2 - b_1b_2 + a_1 = 1 \quad (4.67)$$

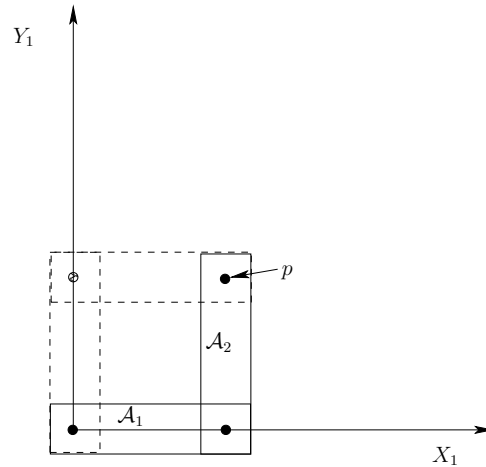


Figure 4.22: There are two configurations that hold the point p at $(1, 1)$.

and

$$f_2 = a_1 b_2 + a_2 b_1 + b_1 = 1, \quad (4.68)$$

by substituting $x = 1$ and $y = 0$ into (4.65) and (4.66). This yields the variety

$$V(a_1 a_2 - b_1 b_2 + a_1 - 1, a_1 b_2 + a_2 b_1 + b_1 - 1, a_1^2 + b_1^2 - 1, a_2^2 + b_2^2 - 1), \quad (4.69)$$

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 (4.69) represents the constrained configuration space for the chain of two links, it is not very explicit. Without an explicit characterization (e.g., 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 (4.69) 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 (4.69), 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} .

The most interesting and complicated situations occur when there are 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

$$a_1 a_2 - b_1 b_2 + a_1 = 0, \quad (4.70)$$

and allow any possible value for y . As shown in Figure 4.23.a, 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 4.23.b shows the resulting variety $V(a_1a_2 - b_1b_2 + a_1)$, 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 4.23.b, 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 4.23.b 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 4.23.a. 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 4.23.b are linear. Figures 4.23.c and 4.23.d show the varieties for the constraints

$$a_1a_2 - b_1b_2 + a_1 = \frac{1}{8}, \quad (4.71)$$

and

$$a_1a_2 - b_1b_2 + a_1 = 1, \quad (4.72)$$

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 4.23.b to 4.23.d can be imagined as part of an animation in which the solution shrinks into a small circle. Eventually, it shrinks to a point for the case $a_1a_2 - b_1b_2 + a_1 = 2$, 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, but allowing one constraint to vary, four different topologies were obtained: 1) two circles joined at a point, 2) a circle, 3) a point, and 4) the empty set.

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 configuration 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

$$\begin{pmatrix} \cos \theta_1 & -\sin \theta_1 & 0 \\ \sin \theta_1 & \cos \theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos \theta_2 & -\sin \theta_2 & 10 \\ \sin \theta_2 & \cos \theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos \theta_3 & -\sin \theta_3 & 10 \\ \sin \theta_3 & \cos \theta_3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \quad (4.73)$$

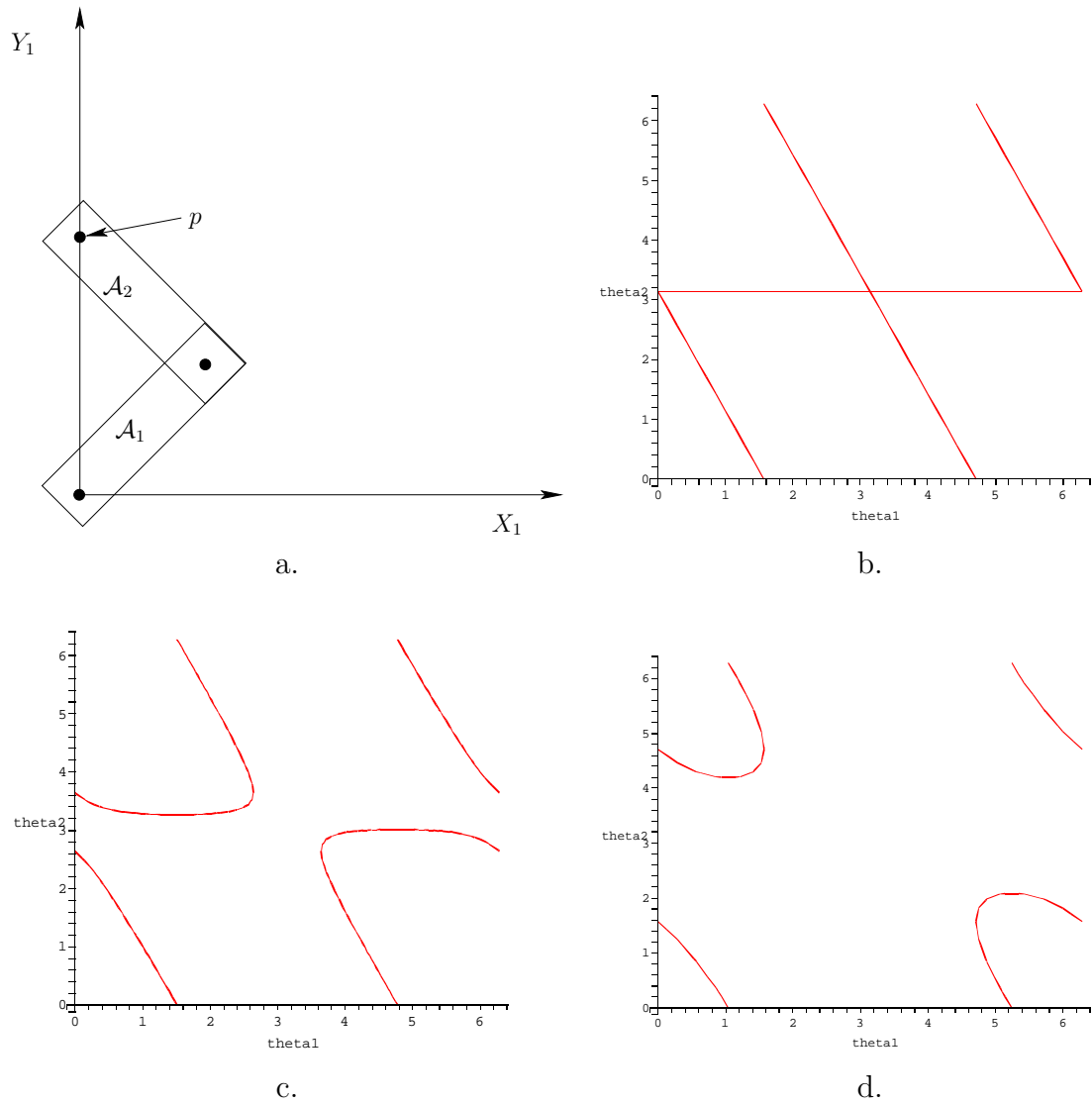


Figure 4.23: A single constraint was added to the point p on \mathcal{A}_2 , as shown in (a). The curves in (b), (c), and (d) depict the variety for the cases of $f_1 = 0$, $f_1 = 1/8$, and $f = 1$, respectively.

To obtain polynomials, let $a_i = \cos \theta_i$ and $b_i = \sin \theta_i$, to obtain

$$\begin{pmatrix} a_1 & -b_1 & 0 \\ b_1 & a_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_2 & -b_2 & 1 \\ b_2 & a_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_3 & -b_3 & 1 \\ b_3 & a_3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad (4.74)$$

for which the constraints $a_i^2 + b_i^2 = 1$ for $i = 1, 2, 3$ must 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

$$f_1 = 2a_1a_2a_3 - a_1b_2b_3 + a_1a_2 - 2b_1b_2a_3 - b_1a_2b_3 + a_1, \quad (4.75)$$

and

$$f_2 = 2b_1a_2a_3 - b_1b_2b_3 + b_1a_2 + 2a_1b_2a_3 + a_1a_2b_3, \quad (4.76)$$

for the X and Y coordinates, respectively.

Again, consider imposing a single constraint,

$$2a_1a_2a_3 - a_1b_2b_3 + a_1a_2 - 2b_1b_2a_3 - b_1a_2b_3 + a_1 = 0, \quad (4.77)$$

which constrains the point $(1, 0) \in \mathcal{A}_3$ to traverse the Y axis. The resulting variety is an interesting manifold, as depicted from three different viewpoints in Figures 4.24 to 4.26 (remember that the sides of the cube are identified).

By increasing the X value for the constraint on the final point, the variety can once again be forced to shrink. Snapshots for $f_1 = 7/8$ and $f_1 = 2$ are shown in Figure 4.27. At $f_1 = 1$, the variety is not a manifold, but 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 two-dimensional 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* [], in which the fourth link, \mathcal{A}_4 is fixed along the X axis from 0 to 1. The resulting variety,

$$V(2a_1a_2a_3 - a_1b_2b_3 + a_1a_2 - 2b_1b_2a_3 - b_1a_2b_3 + a_1 - 1, 2b_1a_2a_3 - b_1b_2b_3 + b_1a_2 + 2a_1b_2a_3 + a_1a_2b_3), \quad (4.78)$$

is depicted in Figure 4.28. Using $\theta_1, \theta_2, \theta_3$ 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 both 2D [] and a 3D extension that uses spherical joints [553].

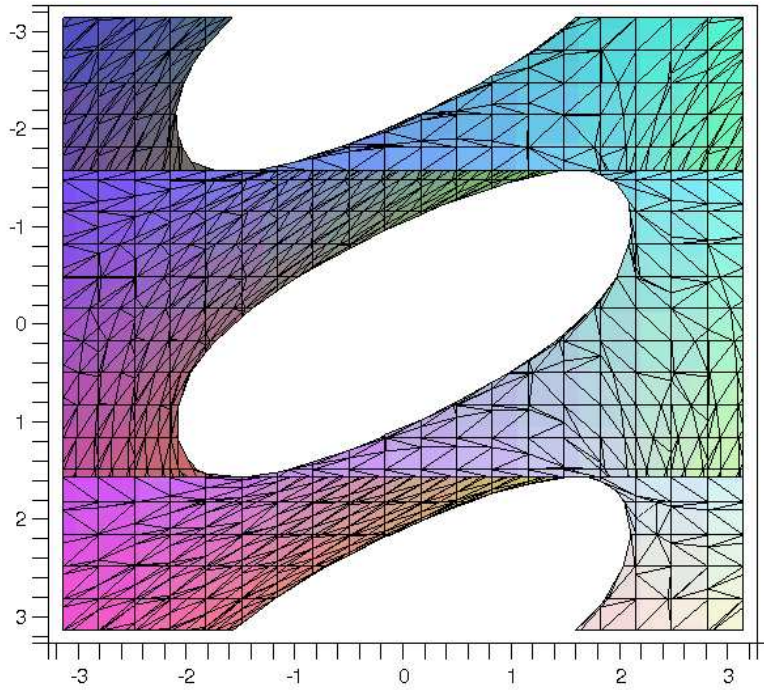


Figure 4.24: The two-dimensional variety for the three-link chain with $f_1 = 0$.

4.4.3 Defining the Variety for General Problems

Now a general methodology for defining the variety will be described. 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 i^{th} coordinate of a point $p \in \mathcal{A}_j$ needs to be held at the value x in the coordinate frame of \mathcal{A}_k .

For the variety in Figure 4.23.b, the first coordinate of a point $p \in \mathcal{A}_2$ was held at the value 0 in \mathcal{W} (which is the same frame as for \mathcal{A}_1). The general form must also allow a point to be fixed with respect to the frame of links other than \mathcal{A}_1 , which did not occur in Section 4.4.2

Suppose that n links, $\mathcal{A}_1, \dots, \mathcal{A}_n$ 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. Let denote a finite set of *joints*, in which each joint is represented as (i, j) , which indicates that \mathcal{A}_i is attached to \mathcal{A}_j by a joint. Is it assumed that $i \neq j$.

A *linkage graph*, $G(V, E)$, is constructed from the links and joints. Each vertex of G represents a link in L . Each edge in G represents a joint. This definition may seem somewhat backwards, especially in the plane because links often look like edges and joints look like vertices. This assignment is also possible, but is

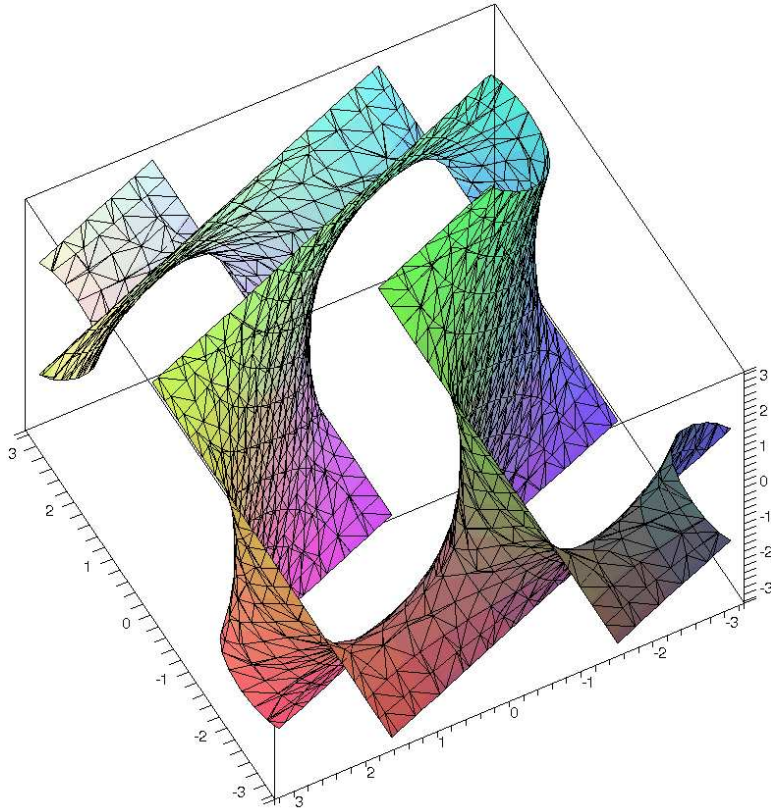


Figure 4.25: Another view of the variety in Figure 4.24.

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 will generate an edge in our representation.

The steps to determine the polynomial constraints that express the variety are:

1. Define the linkage graph, 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 4.29 and 4.30.a. In Figure 4.30, links 4, 13, and 23 were 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, G . This implies that every vertex (or link) is reachable by a path from the

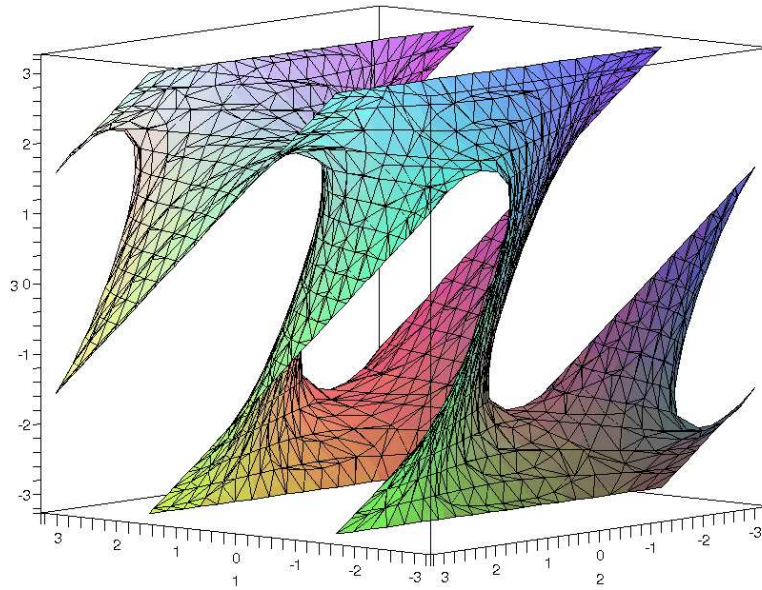


Figure 4.26: A third view of the variety in Figure 4.24.

root). Any spanning tree may be used. Figure 4.30.b shows a resulting spanning tree after deleting the edges shown with dashed lines.

4. Apply the techniques of Section 3.4 to assign frames and transformations to the resulting tree of links.
5. For each edge of G that does not appear in T , write a set of constraints between the two corresponding links. In Figure 4.30.b, 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 3.28, the constraint may be formulated as in (3.73). This is equivalent to what was done to obtain the example in Figure 4.28, 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 4.29 if all joints are revolute. Suppose instead that two bodies, \mathcal{A}_j and \mathcal{A}_k must be rigidly attached. This would require 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 frame of \mathcal{A}_k . If four equations are added, two from each point, then one of them will 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

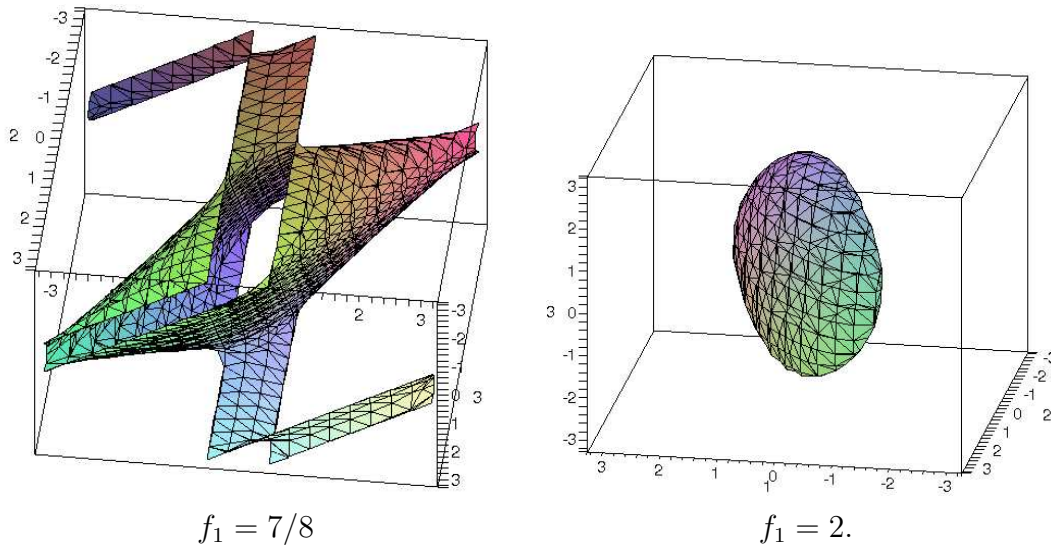


Figure 4.27: If $f_1 > 0$ then variety shrinks. If $1 < p < 3$, the variety is a sphere. At $f_1 = 0$ it is a point, and for $f_1 > 3$ it completely vanishes.

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 will be needed. In total, however, there can 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 transformations, the familiar substitution of complex numbers may be made. If the DH parameterization is used for the 3D case, then each of the $\cos \theta_i, \sin \theta_i$ terms can be parameterized with one complex number, and each of the $\cos \alpha_i, \sin \alpha_i$ terms 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 transformations will result.
7. List the constraints as polynomials 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.

It is 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 chains of links in Section 4.4.2, which produced varieties of various dimensions, depending on the particular equations. Techniques for computing the dimension

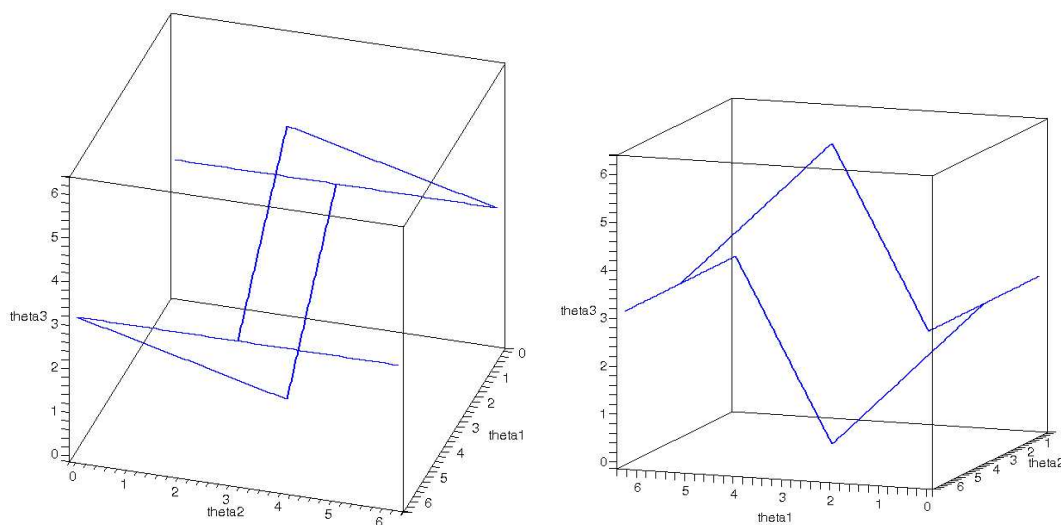


Figure 4.28: If two constraints, $f_1 = 1$ and $f_2 = 0$, are imposed, then the varieties are intersected to obtain a one-dimensional set of solutions. The example is equivalent to a well-studied four-bar mechanism.

exist but require much more machinery than is presented here (see the literature section 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 final concern is the obstacle region, \mathcal{C}_{obs} . Once the variety has been identified, then the obstacle region and motion planning definitions in (4.40) and Formulation 4.3.1 are do not need to be changed, with the understanding that \mathcal{C} represents the linkages that maintain loops while moving.

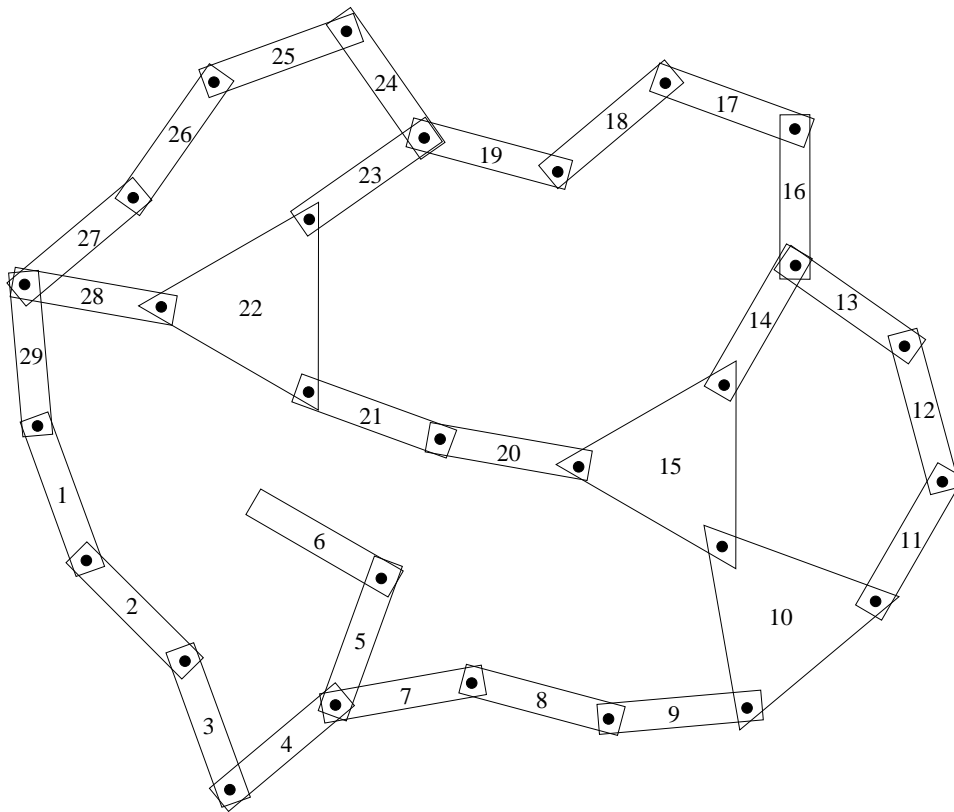


Figure 4.29: A complicated linkage that has 29 links, several loops, links with more than two bodies, and bodies with more than two links. Each integer, i , indicates link \mathcal{A}_i .

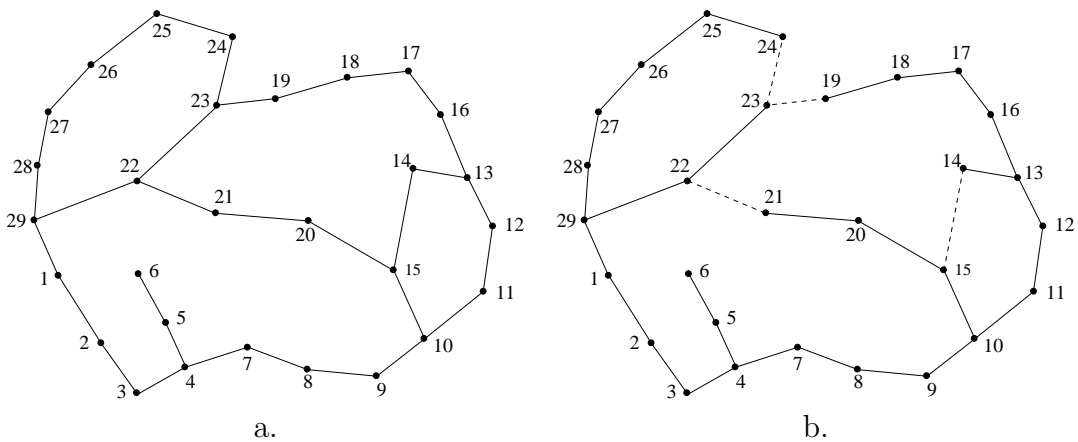


Figure 4.30: a) One way to make the linkage graph that corresponds to the linkage in Figure 4.29. b) A spanning tree is indicated by showing the removed edges with dashed lines.

Literature

Books on basic topology are [23, 328]. An excellent introduction to algebraic topology is the book by Allen Hatcher [317], which is available online at:

<http://www.math.cornell.edu/~hatcher/AT/ATpage.html>

This is a graduate-level mathematics textbook. For an undergraduate-level topology book that covers homology and contains many interesting examples and illustrations, see [399].

Much of the presentation in Section 4.4 was inspired by the nice undergraduate-level introduction to algebraic varieties in [178]. Examples of simple robot arms that form closed chains are also included. In the context of motion planning, an excellent source on motion planning for closed chains is the recent thesis of Juan Cortés [177].

C-space for points moving on a graph[2].

Mention better theoretical algorithms for computing C-space obstacles.

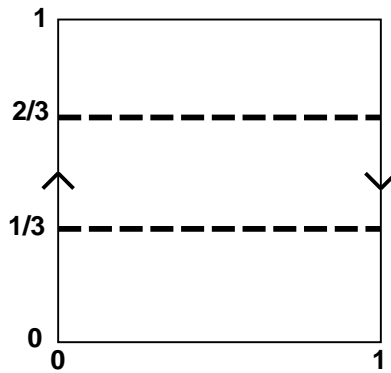
Computing the dimension of algebraic varieties, etc. [178].

Exercises

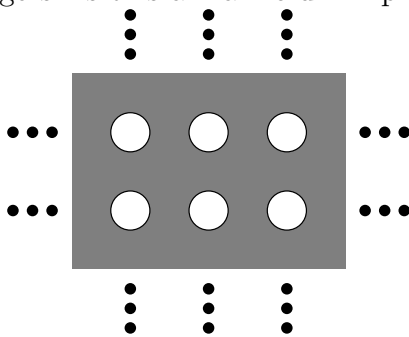
1. Consider the set $X = \{1, 2, 3, 4, 5\}$. Let $X, \emptyset, \{1, 3\}, \{1, 2\}, \{2, 3\}, \{1\}, \{2\},$ and $\{3\}$ be the collection of all subsets of X that are designated as *open sets*. Is X a topological space? Is it a topological space if $\{1, 2, 3\}$ is added to the collection of open sets? Explain. What are the closed sets (assuming $\{1, 2, 3\}$ is included as an open set)? Are any subsets of X neither open nor closed?
2. For the letters of the Russian alphabet А, Б, В, Г, Е, Ё, Ж, З, И, Й, К, Л, М, Н, О, П, Р, С, Т, У, Ф, Ц, Ч, Ш, Щ, Ъ, Ы, Э, Ю, Я determine which pairs are homeomorphic.
3. Prove the homeomorphism yields an equivalence relation on the category of all topological spaces.
4. What is the dimension of the configuration 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 is not changed in any detectable way. Express the configuration 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?
5. Let $\tau_1 : [0, 1] \rightarrow \mathbb{R}^2$ be a loop path in the plane, defined as follows: $\tau_1(s) = (\cos(2\pi s), \sin(2\pi s))$. This path traverses a unit circle. Let $\tau_2 : [0, 1] \rightarrow \mathbb{R}^2$ be another loop path, defined as follows: $\tau_2(s) = (-2 + 3 \cos(2\pi s), \frac{1}{2} \sin(2\pi s))$.

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).

6. Prove that homotopy implies 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.
7. Determine the configuration space for a spacecraft in an asteroids game.
8. Determine the equations for Type B constraints.
9. Determine the configuration 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 sits flatly on the earth without wobbling). [Hint: it is not $\mathbb{S}^2 \times \mathbb{S}^1$]
10. Show that (4.26) is a valid rotation matrix for all unit quaternions.
11. Show that $F[x_1, \dots, x_n]$, the set of polynomials over a field F with variables x_1, \dots, x_n is a group with respect to addition.
12. a) Define a unit quaternion, h_1 , that expresses a rotation of $-\frac{\pi}{2}$ (-90 degrees) 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. Please convert the trig functions whenever possible (for example $\sin \frac{\pi}{6} = \frac{1}{2}$, $\sin \frac{\pi}{4} = \frac{1}{\sqrt{2}}$, and $\sin \frac{\pi}{3} = \frac{\sqrt{3}}{2}$).
13. Suppose there are five polyhedral bodies that can float freely in a 3D world. They are each capable of rotating and translating. If these are treated as "one" composite robot, what would be the topology of the resulting configuration space (assume that the bodies are NOT attached to each other)? What is its dimension?
14. *build the configuration space for containment*
15. The figure below 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?



16. Consider the set of points in \mathbb{R}^2 that are remaining after a closed disk of radius $\frac{1}{4}$ with center (x, y) is removed for every value of (x, y) such that x and y are both integers. Is this a manifold? Explain.



17. Show that the solution curves shown in Figure 4.28 correctly illustrate the variety given in (4.78).

Chapter 5

Sampling-Based Motion Planning

Chapter Status



What does this mean? Check

<http://msl.cs.uiuc.edu/planning/status.html>
for information on the latest version.

There are two main philosophies for addressing the motion planning problem, Formulation 4.3.1 from Section 4.3.1. This chapter presents *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 shown 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, 3D triangles, nonconvex polyhedra, etc. This general philosophy has been very successful in recent years for solving problems from industrial and biological applications that involve thousands and even millions of geometric primitives. Such problems would be practically impossible to solve using explicit \mathcal{C}_{obs} construction techniques.

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 solving 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 initial and goal are given, and the algorithm must search until it finds a

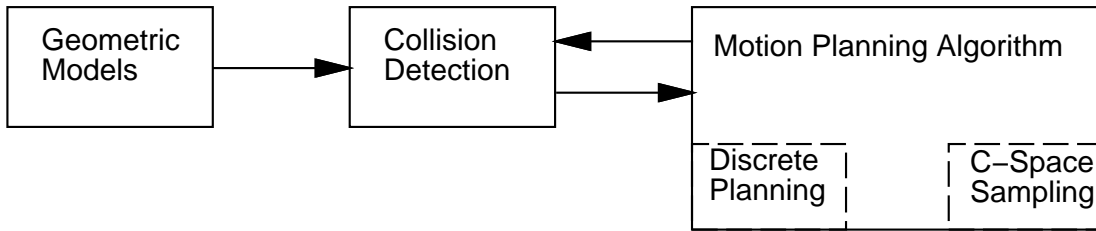


Figure 5.1: The sampling-based planning philosophy uses collision detection as a “black box” that separates the motion planning from the particular geometric and kinematic models. C-space sampling and discrete planning (i.e., searching) are performed.

solution (or it may report early failure). Section 5.5 focuses on Rapidly-exploring Random Trees (RRTs) and Rapidly-exploring Dense Trees, 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 obstacles, \mathcal{Q} remain the same for every query.

5.1 Distance and Volume in C-Space

Virtually all sampling-based planning algorithms require a function that measures 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 to many of these algorithms to have a notion of the volume of a subset of \mathcal{C} . This will result in 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

We are all familiar with the notion of Euclidean distance in \mathbb{R}^n . To define a distance function over \mathcal{C} , it will have to satisfy certain axioms so that it coincides with our expectations about distances 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. **(Non-negativity)** $\rho(a, b) \geq 0$

¹Some topological spaces are not *metrizable*, which means that no function exists that satisfies the axioms. There are many metrization theorems that give sufficient conditions for a topological space to be metrizable [328], and virtually any space that arises in motion planning will be metrizable.

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

$$\rho(x, x') = \left[\sum_{i=1}^n |x_i - x'_i|^p \right]^{\frac{1}{p}}. \quad (5.1)$$

For each value of p , (5.1) is called an L_p metric (pronounced “el pee”). The three most common cases are:

L_2 : The *Euclidean metric*, which is the familiar Euclidean distance in \mathbb{R}^n .

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”.

L_∞ : The L_∞ metric must actually be defined by taking the limit of (5.1) as p tends to infinity. The result is

$$L_\infty(x, x') = \max_{1 \leq i \leq n} |x_i - x'_i|, \quad (5.2)$$

which seems correct because the larger the value of p , the more the largest term of the sum in (5.1) dominates.

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

$$\|x\|_p = \left[\sum_{i=1}^n |x_i|^p \right]^{\frac{1}{p}}. \quad (5.3)$$

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 , of a metric space, (X, ρ) , itself becomes a metric space by restricting the domain of ρ to 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 configuration 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

$$\rho_z(z_1, z_2) = \rho(x_1, y_1, x_2, y_2) = c_1\rho_x(x_1, x_2) + c_2\rho_y(y_1, y_2), \quad (5.4)$$

in which $c_1 > 0$ and $c_2 > 0$ are any positive, real constants, and $x_1, x_2 \in X$ and $y_1, y_2 \in Y$. Other combinations lead to a metric for Z ; for example,

$$\rho_z(z_1, z_2) = (c_1\rho_x^p(x_1, x_2) + c_2\rho_y^p(y_1, y_2))^{1/p}, \quad (5.5)$$

for any positive integer p . In either of these cases, 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

Example 5.1.1 ($SO(2)$ metric using complex numbers) If $SO(2)$ is represented by unit complex numbers, recall that this leads to a subset of \mathbb{R}^2 given by $\{(a, b) \in \mathbb{R}^2 \mid a^2 + b^2 = 1\}$. Therefore, any L_p metric from \mathbb{R}^2 may be used. Using the Euclidean metric,

$$\rho(a_1, b_1, a_2, b_2) = \sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2}. \quad (5.6)$$

for any pair of points (a_1, b_1) and (a_2, b_2) . ■

Example 5.1.2 ($SO(2)$ 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 short cut by computing the length of the line segment 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, since 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:

$$\rho(\theta_1, \theta_2) = \min(|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|), \quad (5.7)$$

for which $\theta_1, \theta_2 \in [0, 2\pi]/\sim$. This may be alternatively be expressed using the complex number representation $a + bi$ as an angle between two vectors:

$$\rho(a_1, b_1, a_2, b_2) = \cos^{-1}(a_1 a_2 + b_1 b_2), \quad (5.8)$$

for two points (a_1, b_1) and (a_2, b_2) . ■

Example 5.1.3 (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 5.1.4 ($SO(3)$ 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 define a metric on \mathbb{S}^3 , but this will not be a metric for $SO(3)$ because antipodal points need to be identified. This leads to a min in the metric. 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). The resulting metric is

$$\rho(h_1, h_2) = \min(\|h_1 - h_2\|, \|h_1 + h_2\|), \quad (5.9)$$

in which the two arguments of the mean 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, $-h_2$.

Just 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 outward onto \mathbb{S}^3 . It is easier to visualize by dropping a dimension. Imagine computing the distance between to 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):

$$\rho(h_1, h_2) = \cos^{-1}(a_1 a_2 + b_1 b_2 + c_1 c_2 + d_1 d_2), \quad (5.10)$$

in which each $h_i = (a_i, b_i, c_i, d_i)$. ■

Example 5.1.5 (Another $SE(2)$ metric) A 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 will be obtained. The units applied to \mathbb{R}^2 and \mathbb{S}^1 are completely incompatible. ■

Example 5.1.6 (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

$$\rho(q_1, q_2) = \max_{a \in \mathcal{A}} \|a(q_1) - a(q_2)\|, \quad (5.11)$$

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 . ■

Example 5.1.7 (T^n metrics) Next consider making a metric over a torus, T^n . The Cartesian product rule (??) 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, \dots, c_n . Robot displacement could be used to determine the coefficients. For example, if robot is a chain of links, it might make sense to weight changes in the first link more heavily because the ensure linkage moves in this case. When the last parameter is changed, only the last link moves; in this case, it might make sense to give less weight. ■

Example 5.1.8 ($SE(3)$ metrics) Metrics for $SE(3)$ can be formed by applying the Cartesian product rules to a metric for \mathbb{R}^3 and the metric for $SO(3)$, which is given in (5.10). Again, this unfortunately leaves coefficients to choose. These issues will arise again in Section 5.3.4, where more details appear on robot displacement. ■

Pseudometrics In many planning algorithms one may want to define 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 by defining the distance to be the optimal cost-to-go for some criterion (recall discrete cost-to-go functions from Section 2.4).

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 will not be symmetric. For example, it requires less energy for a car to travel downhill, as opposed to uphill. 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. This issue arose for the Dubins car, which is covered in Section 13.3.1.

Another example of a pseudometric is the concept of a *potential function* in robotics. This function 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 will generally cause 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 measure 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 [664] for a construction of these bizarre sets. Because of this problem, a workaround was developed that defines a collection of subsets that does avoid the paradoxical ones. 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 countable number of sets in \mathcal{B} , their union must also be in \mathcal{B} .

Note that the last two conditions together 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 . This yields sets of the form

$$B(x, r) = \{x' \in X \mid \rho(x, x') < r\} \quad (5.12)$$

for any $x \in X$ and any $r \in [0, \infty)$. From the open balls, *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 5.1.9 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, $\mu(\{\}) = 0$.
2. For any collection, E_1, E_2, E_3, \dots , of a countable number of pairwise disjoint, measurable sets, let E denote their union. The measure, μ , must satisfy

$$\mu(E) = \sum_i E_i, \quad (5.13)$$

in which i counts over the whole collection.

Example 5.1.10 (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 $n > 3$. One important concept about 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, $A \subseteq \mathbb{R}^m$, is n for some integer $n < m$, then $\mu(A) = 0$, using the Lebesgue measure on \mathbb{R}^m . For example, the set $\mathbb{S}^2 \subset \mathbb{R}^3$ has measure zero because the sphere has no volume. However, we might want to restrict the measure space to be \mathbb{S}^2 and then define surface area. In this case nonzero measure is obtained. ■

Example 5.1.11 (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 5.1.12 (Probability measure) Measure theory even unifies discrete and continuous probability theory. The measure μ can be defined to yield probability mass. The probability axioms are consistent with the measure axioms, while yields a measure space. The integrals and sums needed to define expectations of random variables for continuous and discrete cases, respectively, unify into 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 [253, 664].

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. Fortunately, there is a natural measure, called the Haar measure, for the transformation groups $SO(N)$ and $SE(N)$. 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 5.1.13 (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. These correspond 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. Similarly, invariance for gA may be checked. The transformation g translates the intervals

²Such a measure is unique up to scale, and exists for any locally-compact topological group [253, 664]

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 also 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 5.1.2.

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.

Note that 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 5.1.14 (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. It can be shown that the surface area on \mathbb{S}^3 is the Haar measure. (Since \mathbb{S}^3 is a three-dimensional manifold, it may more appropriately be considered as a boundary 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 any 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 \mathcal{C} -space

is sampled.

Since sampling-based planning algorithms will often be 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 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 $cl(U) = V$ (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 will converge to it. The sequence fractions is a subset of \mathbb{Q} . A sequence will be called *dense* if its underlying set is dense. The bare minimum for sampling methods is that that produce a dense sequence. Stronger requirements, such as uniformity and regularity, will be explained shortly.

A random sequence is probably dense One of the simplest ways conceptually to obtain a dense sequence is to pick points at random in $[0, 1]$. Suppose $I \subset [0, 1]$ is an interval of length e . If k samples are chosen independently at random, the probability that none of them falls into I is e^k . As k approaches infinity, this probability converges to zero. This means that the probability that any interval in $[0, 1]$ contains no points converges to zero. One small technicality exists. The infinite sequence of independently, randomly chosen points is 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 [759]. It exhibits many ideal qualities for applications. At the same time, it is based on a simple

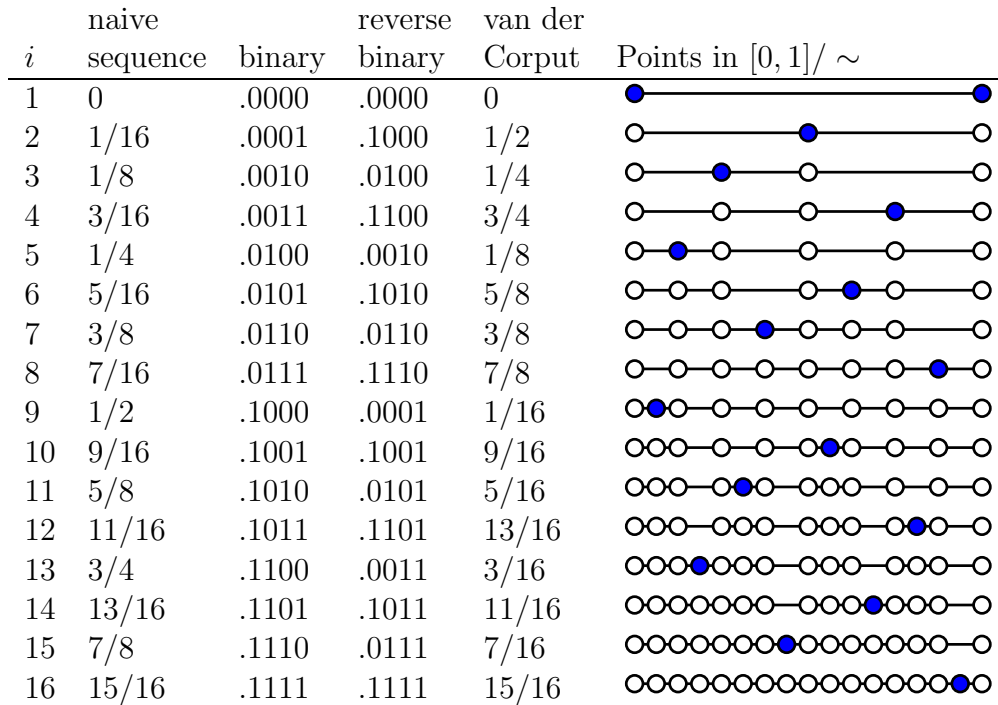


Figure 5.2: The van der Corput sequence is obtained by reversing the bits in the binary decimal representation of the naive sequence.

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 $0 \sim 1$ (recall identifications from Section 4.1.2), 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 sequence by sorting them by increasing value. The problem is that it 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, while 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 i^{th} point of the van der Corput 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 will be 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 was used, the same $\nu(1), \dots, \nu(8)$ would be obtained. Observe in the reverse binary column of Figure 5.2, 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 $\nu(1), \dots, \nu(16)$ will be obtained, and the sequence would continue 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 *a priori* specified. In addition to the nice uniformity properties for every i , the infinite van der Corput sequence is also dense in $[0, 1]/\sim$. There 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 planning.

5.2.2 Random Sampling

Now imagine moving beyond $[0, 1]$ and generating a dense sample sequence for any bounded configuration 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 will work 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 configuration 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 a uniform random samples, x_1 and x_2 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, if 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 will be uniformly randomly distributed over \mathcal{C} . Combining samples over Cartesian products is much more difficult for nonrandom (deterministic) methods, 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 [26], and is reproduced here. Choose three points $u_1, u_2, u_3 \in [0, 1]$ uniformly at random. The random quaternion is given by the simple expression

$$h = (\sqrt{1 - u_1} \sin 2\pi u_2, \sqrt{1 - u_1} \cos 2\pi u_2, \sqrt{u_1} \sin 2\pi u_3, \sqrt{u_1} \cos 2\pi u_3). \quad (5.14)$$

A full explanation of the method is given in [26], 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 S_2 that corresponds to some line of latitude on \mathbb{S}^2 . The second parameter selects the longitude, $2\pi u_2$. Unfortunately, the points will not be 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 will not have equal area; therefore, the points will not be uniformly distributed. However, for \mathbb{S}^3 , the 3D crusts happen to have the same area (or measure); this can be shown by evaluating surface integrals. This implies that a (infinitesimal) slice can be selected uniformly at random with u_1 , and a point on the crust is selected uniformly at random by u_2 and u_3 . For \mathbb{S}^4 and beyond, the measure of the crusts vary, which means this elegant scheme only works for \mathbb{S}^3 . 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 will not affect the uniform random distribution of the samples.

Generating random directions Some sampling-based algorithms require choosing motion directions at random. 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 $n = 3$, then the previously mentioned trick for $SO(3)$ should be used. If $n = 2$ or $n > 3$, then samples can be generated using a slightly more expensive method that exploits spherical symmetries of the multi-dimensional Gaussian density function [251]. 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 ($k \leq 12$ is usually sufficient). The vector $(u_1, u_2, \dots, u_{n+1})$ 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 tradeoffs, 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.³ 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 troubles is the Mersenne twister [540], 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 [478, 579]. 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 $\gcd(M, a) = 1$. The third parameter, c , must be chosen to satisfy $0 \leq c < M$. Using modular arithmetic, a sequence can be generated as

$$y_{i+1} = ay_i + c \pmod{M}, \quad (5.15)$$

by starting with some arbitrary seed $1 \leq y_0 \leq M$. Pseudorandom numbers in $[0, 1]$ are generated by the sequence

$$x_i = y_i/M. \quad (5.16)$$

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 [579].

³There are exceptions which use physical phenomena as a random source.

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 test of uniform randomness are used. One of the simplest, the *chi-square test*, is described here. This test measures how far computed statistics are 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:

$$e(P) = \sum_{i=1}^{100} (b_i - k/100)^2, \quad (5.17)$$

in which b_i is the number of samples that fall into box i . It is shown [391] that $e(P)$ will follow a *chi-squared* distribution. A surprising fact is that the goal is not to minimize $e(P)$. If this value 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 appeared in each of the 100 boxes. This yields $e(P) = 0$, but how likely is this to ever occur? The value must generally be larger (it appears in many statistical tables) to account for the irregularity due to randomness.

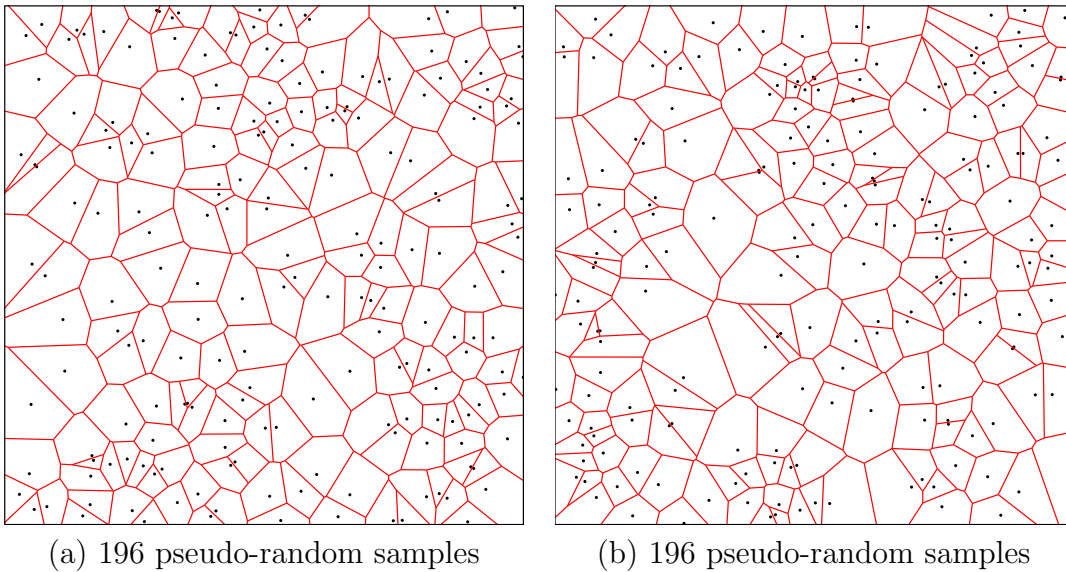


Figure 5.3: Irregularity in a collection of (pseudo)random samples can be nicely observed with Voronoi diagrams.

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*, $Vor(x)$. For any point $y \in Vor(x)$, x is the closest sample to y using Euclidean distance. The different sizes and shapes of these regions gives some indication of the required irregularity of

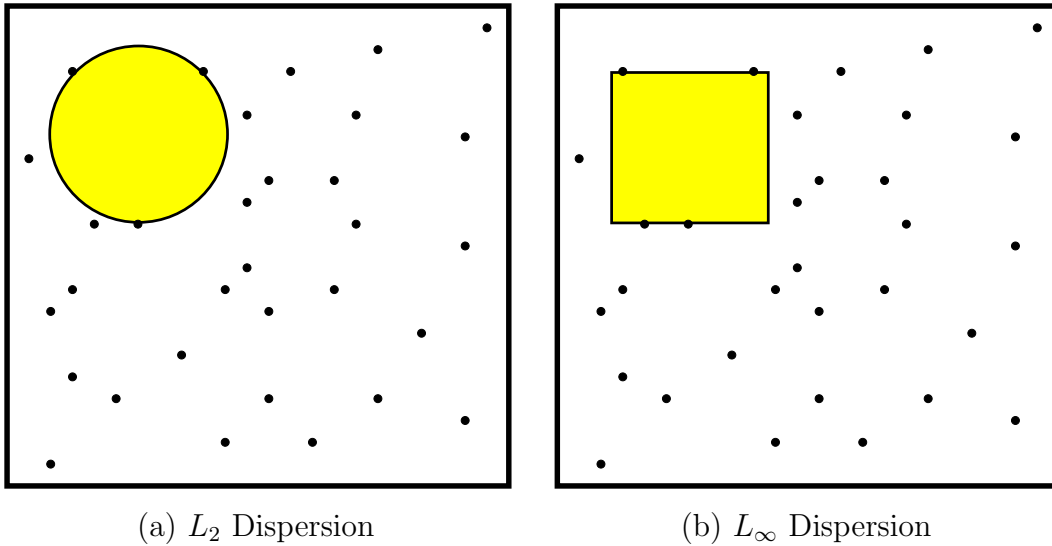


Figure 5.4: Reducing dispersion means reducing the radius of the largest empty ball.

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. Instead, the goal is to optimize a criterion called *dispersion* [579]. Intuitively, the idea is to place samples in a way that makes the largest uncovered area be as small as possible. This will yield a generalization of the idea of *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 *dispersion complete* algorithm. This applies to multiresolution grids or any other dense sample sequence. These concepts are explained further at the end of Section 5.4.2.

Dispersion definition The *dispersion*⁴ of a set P of samples in a metric space (X, ρ) is

$$\delta(P) = \sup_{x \in X} \min_{p \in P} \rho(x, p). \quad (5.18)$$

⁴The definition is unfortunately backwards from intuition. Lower dispersion means that the points are nicely dispersed. Thus, more dispersion is bad, which is counterintuitive.

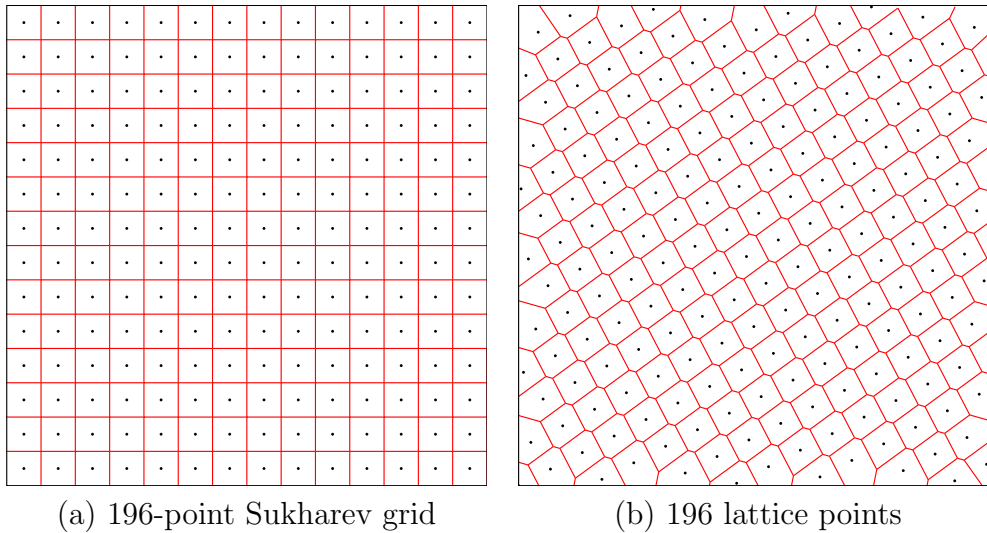


Figure 5.5: The Sukharev grid and a lattice.

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 (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. This is Figure 5.3 for 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 places at all Voronoi vertices is the dispersion. This interpretation extends nicely to higher dimensions.

Making good grids Optimizing dispersion will force 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 a point is placed at the center of each cube, as shown for $n = 2$ and $k = 96$ in Figure 5.5.a. 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 will be 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* [728].

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 [728]:

$$\delta(P) \geq \frac{1}{2 \lfloor N^{\frac{1}{d}} \rfloor}. \quad (5.19)$$

This implies that keeping dispersion fixed *requires* exponentially many points in dimension.

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 for 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 [174] 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 and lift the samples to \mathbb{S}^n with relatively little distortion [787]. 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 5.6.

Example 5.2.1 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 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 remains unchanged. ■

One nice property of grids is that they have a lattice structure. This means that neighboring points can be obtained very easily by adding or subtracting vectors. Let g_j be an n -dimensional vector called a *generator*. A point on a lattice can be expressed as

$$x = \sum_{j=1}^n k_j g_j, \quad (5.20)$$

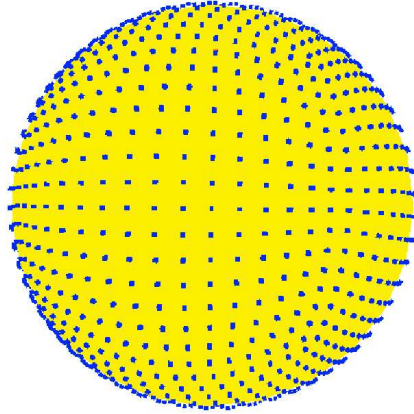


Figure 5.6: A distorted grid can even be placed over spheres and $SO(3)$ by putting grids on faces an inscribed cube and lifting them to the surface.

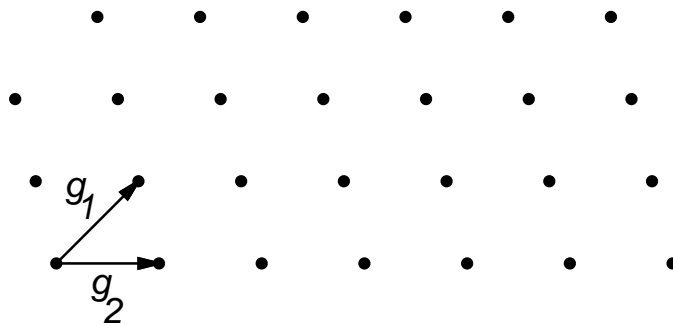


Figure 5.7: A lattice can be considered as a grid in which the generators are not necessarily orthogonal.

for n independent generators, as depicted in Figure 5.7. 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 5.5.b. In Section 5.4.2 lattice structure will become important and convenient for defining the search graph.

Infinite 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 *multiresolution grid*. The resolution can be iteratively doubled. For a multiresolution 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^{in} points for any positive integer i , a grid with 2^i points per axis will be represented. If we are only allowed to use complete grids, 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} , etc., 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 $X = T^2 = [0, 1]^2 / \sim$. 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 will continue recursively to obtain a complete grid after $k = 2^{2i}$ points, for any positive integer i . For any k , however, there will be only a partial grid. The points will be 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 octant. There are many other orderings that produce L_∞ -optimal dispersion. Selecting orderings that additionally optimize other criteria, such as discrepancy or L_2 dispersion are covered in [495, ?]. Unfortunately, it is more difficult to make a multiresolution Sukharev grid. The base becomes 3 instead of 2; after every 3^{in} points a complete grid will be 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 [579]. 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 $f(n) = \frac{1}{\ln 4}$ [579]; 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 an corridor in \mathcal{C}_{free} . In some cases, a 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 [579]. 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* [768] for k samples is defined as

$$D(P, \mathcal{R}) = \sup_{R \in \mathcal{R}} \left| \frac{|P \cap R|}{k} - \frac{\mu(R)}{\mu(X)} \right| \quad (5.21)$$

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 [538]. 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 or not 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.

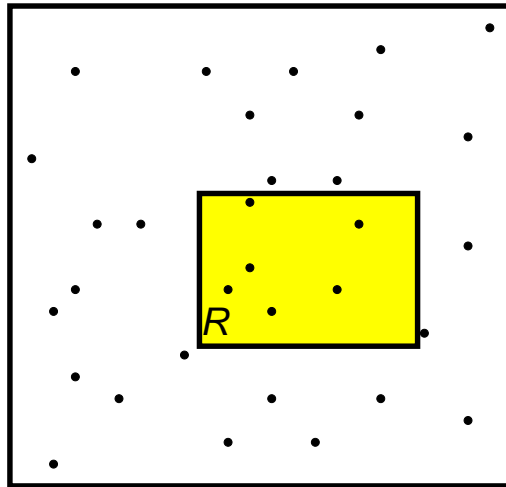


Figure 5.8: Discrepancy measures whether the right number of points fall into boxes. It is related to the chi-square test, but optimizes over all possible boxes.

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 lower bound corresponds to the best that can be achieved by a sequence of point sets, as opposed to a single sequence.

Relating Dispersion and Discrepancy Since balls have volume, there is a close relationship between discrepancy, which is measure-based, and dispersion, which is metric-based. For example, for any $P \subset [0, 1]^n$,

$$\delta(P, L_\infty) \leq D(P, \mathcal{R})^{1/d}, \quad (5.22)$$

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, there is a tradeoff that 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 of the literature has led to low-discrepancy sequences and point sets [579, 712, 744]. 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 pseudo-random 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, based on extending the van der Corput sequence. The *Halton sequence* is an n -dimensional generalization van der Corput sequences, but instead of using binary representations, a different basis is used for each coordinate [311]. 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 5.9.a shows the first 196 Halton points in \mathbb{R}^2 .

Choose n relatively prime integers p_1, p_2, \dots, p_n (usually the first n primes, $p_1 = 2, p_2 = 3, \dots$, are chosen). To construct the i^{th} sample, consider the digits of the base p representation for i in the reverse order (that is, write $i = a_0 + pa_1 + p^2a_2 + p^3a_3 + \dots$, where each $a_j \in \{0, 1, \dots, p\}$) and define the following element of $[0, 1]$:

$$r_p(i) = \frac{a_0}{p} + \frac{a_1}{p^2} + \frac{a_2}{p^3} + \frac{a_3}{p^4} + \dots \quad (5.23)$$

The i^{th} sample in the Halton sequence is

$$(r_{p_1}(i), r_{p_2}(i), \dots, r_{p_n}(i)), \quad i = 0, 1, 2, \dots \quad (5.24)$$

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 is an adaptation of the Halton sequence [312]. Using only $d - 1$ distinct primes, the i^{th} sample in a Hammersley point set with k elements is

$$\left(\frac{i}{k}, r_{p_1}(i), \dots, r_{p_{d-1}}(i) \right), \quad i = 0, 1, \dots, N - 1. \quad (5.25)$$

Figure 5.9.b 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 [579]. The constant for the dispersion also increases exponentially, which is must worst than for the methods of Section 5.2.3.

Improved constants are obtained for sequences and finite points by using (t,s)-sequences, and (t,m,s)-nets, respectively [579]. The key idea is to enforce zero discrepancy over a particular subset 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 [579]). The Niederreiter-Xing (t,s)-sequence has the best-known asymptotic constant, $(a/d)^d$, in which a is a small constant [581].

The third category is *lattices*, which can be considered as a generalization of grids that allows nonorthogonal axes [538, 712, 765]. As an example, consider Figure 5.5.b, which shows 196 lattice points generated by the following technique. Let α be a positive irrational number. For a fixed k (lattices are closed sample sets), generate the i^{th} point according to $(\frac{i}{k}, \{i\alpha\})$, in which $\{\cdot\}$ denotes the fractional part of the real value (modulo-one arithmetic). In Figure 5.5.b, $\alpha = \frac{\sqrt{5}+1}{2}$,

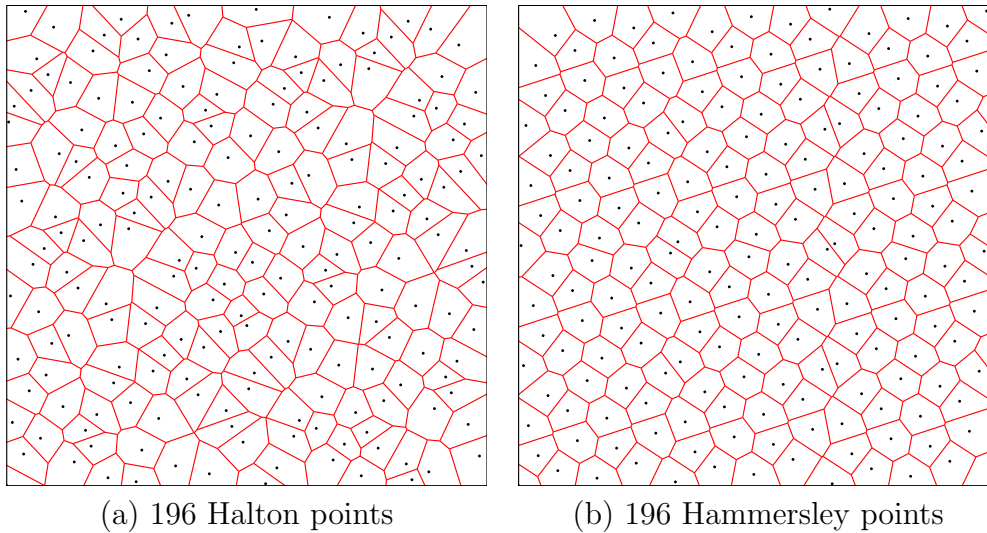


Figure 5.9: The Halton and Hammersley points are easy to construct and provide a nice alternative to random sampling that achieves more regularity. Compare the Voronoi regions to those in Figure 5.3. Beware that although these sequences produce asymptotically optimal discrepancy, their performance degrades substantially in higher dimensions (e.g., beyond 10).

the Golden Ratio. This procedure can be generalized to d dimensions by picking $d - 1$ distinct irrational numbers. A technique for choosing the α parameters by using the roots of irreducible polynomials is discussed in [538]. The i^{th} sample in the lattice is

$$\left(\frac{i}{k}, \{i\alpha_1\}, \dots, \{i\alpha_{n-1}\} \right). \quad (5.26)$$

Recent analysis shows that some lattice sets achieve asymptotic discrepancy that is very close to that of the best-known non-lattice sample sets [323, 745]. Thus, restricting the points to lie on a lattice seems to entail little or no loss in performance, but with 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 [324, 745].

5.3 Collision Detection

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 applications, the majority of computation time is spent in collision checking, as opposed to planning.

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, of course, 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.

5.3.1 Basic Concepts

Just 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, $\phi(q) = \text{FALSE}$.

Hausdorff Distance 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 *Hausdorff distance* is defined as

$$\rho(E, F) = \min_{e \in E} \min_{f \in F} \|e - f\|, \quad (5.27)$$

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, $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_m$, and that \mathcal{O} has k connected components. For this complicated situation, collision detection can be viewed as a two-phase process.

1. In the *broad phase*, the task is to avoid performing expensive computations for bodies that are far 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 [?]. 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. 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.

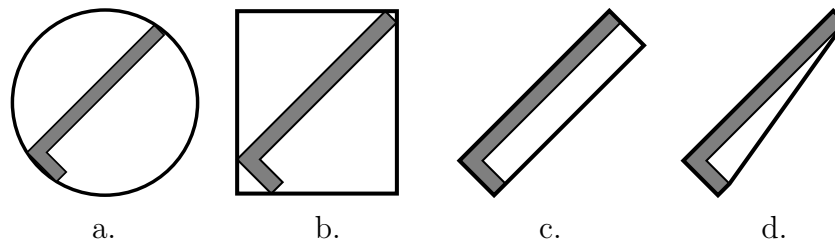


Figure 5.10: Four different kinds of bounding regions: a) sphere, b) axis-aligned bounding box (AABB), c) oriented bounding box (OBB), d) convex hull. Each one usually provides a tighter approximation than the previous one, but is more expensive to test for overlapping pairs.

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 represent each body as a tree in which each node represents a *bounding region* that contains all of the points in one portion of the body. The bounding region of the root node contains all of the points in the body.

There are generally two opposing criteria that guide the selection of the type of bounding region::

1. The region should fit the actual data as tightly as possible.
2. The intersection test for two regions should be as efficient as possible.

Several popular choices are shown in Figure 5.10 for an L-shaped body.

The tree is constructed for a body, E (or alternatively, F) recursively as follows. For each node, consider the set, X , of all points in E that are contained in the bounding region. Two child nodes 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 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 5.11 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 node represents a single triangle. In this case, it is easy to test for intersection of two triangles.

Consider the problem of determining whether bodies E and F are in collision. Suppose that a trees, T_e and T_f , have been constructed for E and F , respectively. If the bounding regions of the root nodes of T_e and T_f do not intersect, then it

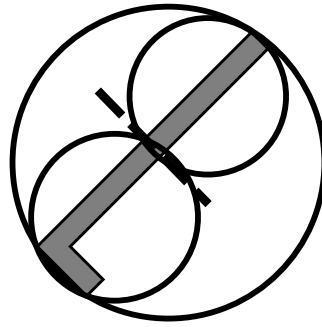


Figure 5.11: The large circle shows the bounding region for a node that covers an L-shaped body. After performing a split along the dashed line, two smaller circles are used to cover the two halves of the body. Each circle corresponds to a child node.

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 will be traversed, until eventually the leaves are reached. If triangle primitives are used for the geometric models, then at the leaves, the algorithm will test the individual triangles for collision, instead of bounding regions. Note that as the trees are traversed, if a bounding region from the node, n_1 , of T_e does not intersect the bounding region from a node, n_2 , of T_f , then no children of n_1 have to be compared to children of n_2 . This can generally result in dramatic reduction in comparison to the amount of comparisons needed in a naive approach that, for example, tests all pairs of triangles for intersection.

It is possible to extend the hierarchical collision detection scheme to also compute distance. If at any time, a pair of bounding regions have a distance greater than the smallest distance computed so far, then their children do not have to be considered [493].

5.3.3 Incremental Methods

This section focuses on a particular approach called *incremental distance computation*, which assumes that between successive calls to a when 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 [492, 556]. 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

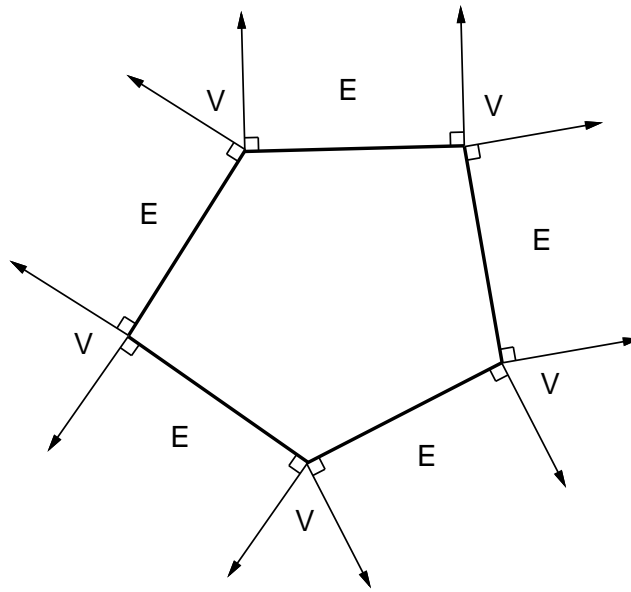


Figure 5.12: The Voronoi regions alternate between being edge-based and vertex-based. The Voronoi regions of vertices are labeled with a “V”, and the Voronoi regions of edges are labeled with an “E”. Note that the Voronoi regions alternate between “V” and “E” (no two Voronoi regions of the same kind are adjacent). The adjacencies between these Voronoi regions follow the same pattern as the adjacencies between vertices and edges in the polygon (a vertex is always between two edges, etc.).

segments, all of the line segments must fit together perfectly to form polygons. There can be no isolated segments or chains of segments. In 3D, polyhedral models are required to have all faces come together perfectly to form the boundaries of three-dimensional shapes. It cannot be an arbitrary collection of 3D triangles.

The method will be explained for the case of 2D convex polygons. Voronoi regions will be defined for a convex polygon, 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, g , the set of all points from which g is the closest feature is the Voronoi region of g , denoted $Vor(g)$. Figure 5.12 shows all ten Voronoi regions for a pentagon.

For any two convex polygons that do not intersect, the closest point will be determined by a pair of points, one on each polygon (usually the points are unique, except in the case of parallel edges). Consider the feature for each point in this pair. There are only three possible combinations:

- **Edge-Edge** Each point of the closest pair each lies on an edge. In this case, the edges must be parallel.
- **Edge-Vertex** One point of the closest pair lies on an edge, and the other

lies on a vertex.

- **Vertex-Vertex** Each point of the closest pair is a vertex of a polygon.

Let g_e and g_f represent any feature pair of E and F , respectively. Let $(x_e, y_e) \in g_e$ and $(x_f, y_f) \in g_f$ denote the closest pair of points, among all pairs of points in g_e and g_f , respectively. The following condition can be used to determine whether the distance between (x_e, y_e) and (x_f, y_f) is the distance between E and F :

$$(x_f, y_f) \in Vor(g_e) \text{ and } (x_e, y_e) \in Vor(g_f) \quad (5.28)$$

If this condition is satisfied for a given feature pair, then the distance between E and F equal to the distance between g_e and g_f . This implies that the distance between E and F can be determined in constant time. The assumption that E moves only a small amount is made to increase the likelihood that the closest feature pair will remain 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 can be 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 to not move far between successively calls.

The same ideas can be applied for the 3D case in which the bodies are convex polyhedra [492, 556]. 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 algorithm has nearly 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 Δq is chosen as the configuration 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 cases 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 Hausdorff 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 5.1.6. Before expressing the general case, first the concept will be explained 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 the path from q to q' is monotonic (if the robot can take any path between q and q' , then no such guarantee could possibly be made).

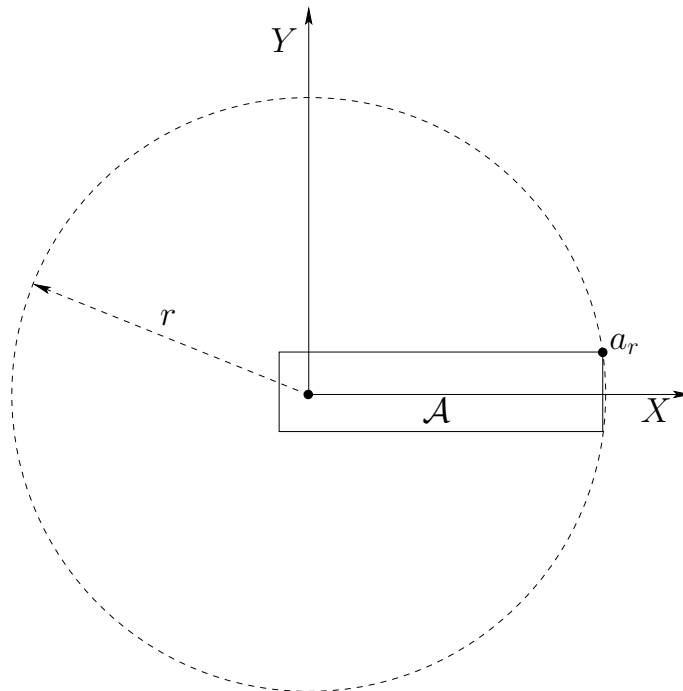


Figure 5.13: The furthest point on \mathcal{A} from the origin travels the fastest when rotated. At most it can be displaced by $2\pi r$, if x_t and y_t are fixed.

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 5.13 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' :

$$|a(x_t, y_t, \theta) - a(x_t, y_t, \theta')| \leq |a_r(x_t, y_t, \theta) - a_r(x_t, y_t, \theta')| < r|\theta - \theta'|, \quad (5.29)$$

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 as long as $r|\theta - \theta'| < d$. Note that this is a loose upper bound since 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 at same time, then a region in \mathcal{C}_{free} may be defined as

$$\{(x'_t, y'_t, \theta') \in \mathcal{C} \mid |x_t - x'_t| + |y_t - y'_t| + r|\theta - \theta'| < d. \quad (5.30)$$

Such bounds can generally be used to set the 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 direction⁵ 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 to obtain a straightforward generalization of (5.30) that has six terms, three for translation and three for rotation. For each of the three rotation parts, a point with 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* bound of the form

$$\|a(q) - a(q')\| < c\|q - q'\|, \quad (5.31)$$

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 c as small as possible to enable larger variations in q .

A better method is to individually bound the link displacement with respect to each parameter,

$$\|a(q_1, \dots, q_{i-1}, q_i, q_{i+1}, \dots, q_n) - a(q_1, \dots, q_{i-1}, q'_i, q_{i+1}, \dots, q_n)\| < c_i|q_i - q'_i|, \quad (5.32)$$

⁵To formally talk about directions, it would be better to define a differentiable structure on \mathcal{C} . This will be deferred to Section ??, where it seems unavoidable.

to obtain the Lipschitz constants c_1, \dots, c_n . The bound on robot displacement becomes

$$\|a(q) - a(q')\| < \sum_{i=1}^n c_i |q_i - q'_i|. \quad (5.33)$$

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 configuration space is 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 $1 \leq i \leq n$), which means that last point is at the point (500, 0). Changes in θ_1 , the orientation of the first link, will dramatically move \mathcal{A}_{50} . However, changes in θ_{50} will move \mathcal{A}_{50} a smaller amount. Therefore, it is advantageous to pick different Δq_i for each $1 \leq i \leq n$. 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 still might be preferable to determine in a local region around $q \in \mathcal{C}$ how much link displacement is possible with respect to each parameter perturbation. A linear method could be obtained by analyzing the Jacobian of the transformations, such as (3.45) and (3.49).

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 recursive binary strategy [272]. This will make no difference if the path is collision-free, but it often saves time when 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: $\tau(0)$, $\tau(1/2)$, $\tau(1/4)$, $\tau(3/4)$, $\tau(1/8)$, \dots . 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 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 and q_g are 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. In fact, these sampling-based planning algorithms are strikingly similar to the family of search algorithms summarized in Section 2.3.4. 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 G is an undirected graph whose edges represent paths, as opposed to a directed graph whose 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 15.

Most single-query sampling-based planning follow this template:

1. **Initialization:** Let $G(V, E)$ represent an undirected *search graph*, for which the node set, V contains a node for q_i and possibly other states in \mathcal{C}_{free} , and the edge set, E , is empty.
2. **Vertex Selection Method (VSM):** Choose a vertex $q_{cur} \in V$ for expansion.
3. **Local Planning Method (LPM):** For some $q_{new} \in \mathcal{C}_{free}$ which 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 Edge in Graph:** Insert τ_s into E , as an edge from q_{cur} to q_{new} . If q_{new} is not already in V , it is added.
5. **Check for Solution:** Determine whether 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 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, G is a topological graph, as defined in Example 4.1.6. 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 might 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

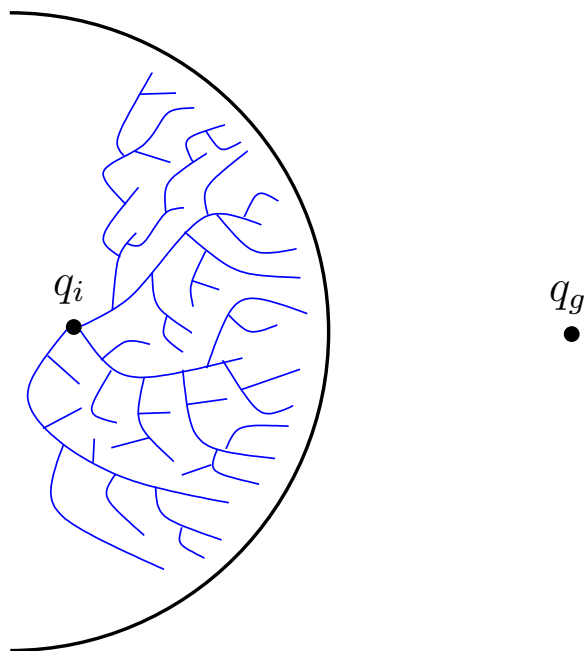


Figure 5.14: Imagine a problem in which the configuration space obstacle is a giant “bowl” that can trap the configuration. This figure is drawn in two dimensions, but imagine that the \mathcal{C} has many dimensions, such as 6 for $SE(3)$ or perhaps dozens for a linkage. If the discrete planning algorithms from Section 2.3 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 states in this bowl would typically be on the order of 100^n , for an n -dimensional configuration space.

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 2.3.1. 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 2.3 may be applied to motion planning by approximating \mathcal{C} with a high-resolution grid. The resulting problem looks like a multidimensional extension of Example 2.2.1 (the “labyrinth” is formed by \mathcal{C}_{obs}). For a high-resolution grid in a high-dimensional space, most classical discrete searching algorithms have trouble becoming trapped in a local minimum. There could be an astronomical number of states that fall within a concavity in \mathcal{C}_{obs} that must be escaped to solve the problem, as shown in Figure 5.14. Therefore,

sampling-based motion planning algorithms combine sampling and searching in a way that attempts to overcome these kinds of difficulties.

Just as in the case of discrete search algorithms, there are several classes of algorithms based on the number of search trees.

Unidirectional (single tree) methods: In this case, the planning appears very similar to discrete forward search, which was given in Figure 2.5. The main difference between algorithms in this category is how they implement the VSM and LPM. Figure 5.15 shows a bug trap⁶ example for which forward search algorithms will have great trouble; however, the problem might not be difficult for backwards search, if the planner incorporates some kind of greedy, best-first behavior.

Bidirectional (two tree) methods: Since it is not known whether or not 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 nodes. 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.

Multidirectional (more than two trees) methods: If the problem is so bad that a double bug trap exists, as shown in Figure 5.16, 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. For which pairs should attempts be made to connect? How often should these attempts be made? Which vertex pairs should be selected. Many heuristic parameters may be needed to answer these questions.

Of course, we can play the devil's advocate and construct the example in Figure 5.17, for which virtually all sampling-based planning algorithms are doomed. Several variations can also be made. For example, the connecting pipe could have a small hold in it; this does not help. The two bug traps could even be disconnected, as long as the entrance to each is hard to find.

5.4.2 Adapting Classical 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 Chapter 2. The resulting planning problem actually looks very

⁶This principle is actually used in real life to trap flying bugs. This example and analogy was suggested by James O'Brien in a discussion with James Kuffner.

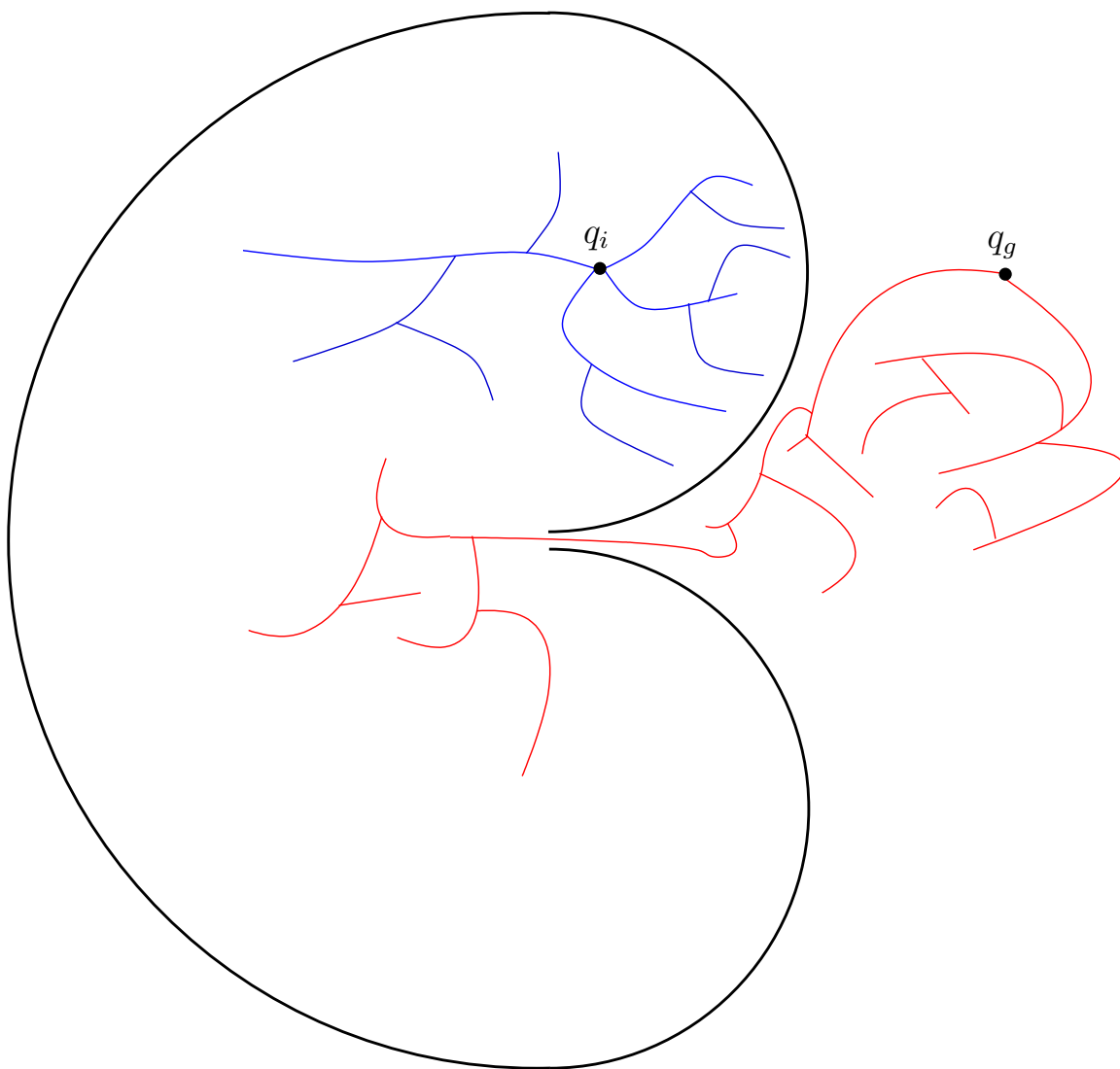


Figure 5.15: This example, again in high dimensions, can be considered as a kind of “bug trap”. To leave the trap, a path must be found from q_i into the narrow opening. Imagine a fly buzzing around through the high-dimensional trap. The escape opening might not look too difficult in two dimensions, but if it has a small range with respect to each configuration parameter, it will be 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 algorithms. It might be easier for a search tree that starts in q_g to arrive in the bug trap.

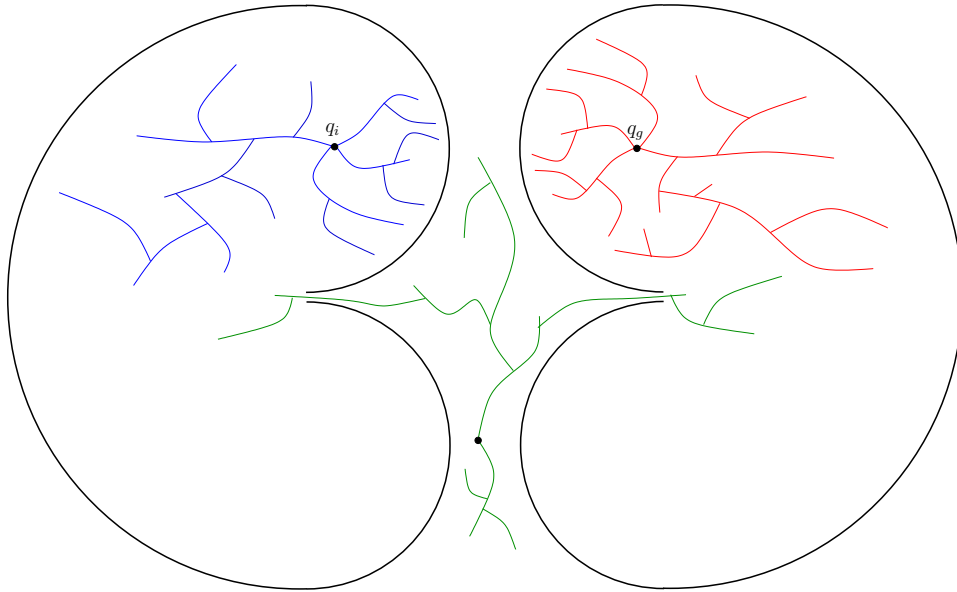


Figure 5.16: The double bug trap is trouble even for bidirectional search. This may motivate the construction of more than two trees.

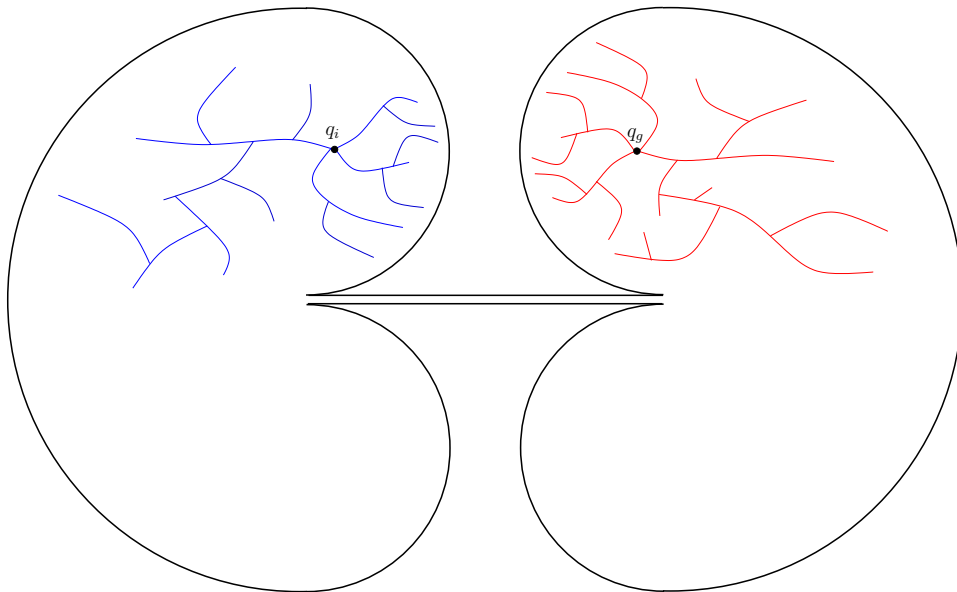


Figure 5.17: A multidimensional search cannot even help with this example, which involves two bug traps connected by a thin tube. We must accept the fact that some problems are hopeless to solve using sampling-based planning methods, unless there is some problem-specific structure that can be additionally exploited.

similar to Example 2.2.1. 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 during the search because an explicit characterization of \mathcal{C}_{obs} is too expensive to construct (recall Section 4.3).

Assume that an n -dimensional configuration 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 will have to be deformed into $[0, 1]^3 / \sim$.

Discretization Assume that \mathcal{C} is *discretized* by using the *resolutions* k_1, k_2, \dots , and k_d , 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 $\Delta q_i = [0 \ \dots \ 0 \ \frac{1}{k_i} \ 0 \ \dots \ 0]$. A *grid point* is a configuration $q \in \mathcal{C}$ that can be expressed in the form⁷

$$\sum_{i=1}^n j_i \Delta q_i, \quad (5.34)$$

in which each $j_i \in \{0, 1, \dots, k_i\}$. The integers j_1, \dots, j_n can be imagined as array indices for the grid. Let the term *boundary grid point* refer to a grid point that has $j_i = 0$ or $j_i = k_i$ for some i . Note that due to identification, boundary grid points might have more than one representation.

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

$$N_1(q) = \{q + \Delta q_1, \dots, q + \Delta q_n, q - \Delta q_1, \dots, q - \Delta q_n\}. \quad (5.35)$$

For an n -dimensional configuration space there are at most $2n$ 1-neighbors. In two dimensions, this yields 4 1-neighbors, which can be thought of as “up”, “down”, “left” and “right”. We say “at most” because some directions may be blocked by the obstacle region.

A *2-neighborhood* is defined as

$$N_2(q) = \{q \pm \Delta q_i \pm \Delta q_j \mid 1 \leq i, j \leq n, i \neq j\} \cup N_1(q). \quad (5.36)$$

Similarly, a *k-neighborhood* can be defined for any positive integer $k \leq n$. For a n -neighborhood, there are at most $3^n - 1$ neighbors; there may be fewer due to collisions. The definitions can be extended in a straightforward way to handle the boundary points.

⁷Alternatively, the general lattice definition in (5.20) could be used.

Obtaining a discrete planning problem Once the grid and neighborhoods have been defined, it is straightforward to define a discrete planning problem. Figure 5.18 depicts the process for a problem in which there are 9 Sukharev grid points in $[0, 1]^2$. Using 1-neighborhoods, the potential edges in the search graph, $G(V, E)$, appear in Figure 5.18.a. Note that G is a topological graph, as defined in Example 4.1.6 because each vertex is a configuration and each edge is a path. If q_i and q_g do not coincide with grid points, they need to be connected to some nearby grid points, as shown in Figure 5.18.b. 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 5.18.a will not appear in G because some will intersect with \mathcal{C}_{obs} . Figure 5.18.c 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 collision checking all potential vertices and edges. 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 G is revealed “on the fly” during the search. This is the same situation that occurs for the discrete planning methods from Section 2.3. In the current setting, the potential edges of 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 5.18.d.

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 problem 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 the right 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 classical 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, this savings diminishes 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^{in} 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.

An similar to iterative deepening from Section 2.3.2 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 will yield grids of sizes 2^n , 3^n , 4^n , etc., 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 this would dominate the running time. An easy way make this efficient is to apply the *union-find* algorithm [176, 655] to iteratively keep track of connected components in G instead of performing explicit searching. If q_i and q_j 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 almost⁸ 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 multiresolution qualities, then it may perform well without the

⁸It is not constant because the running time includes the inverse Ackerman function, which grows very, very slowly. For all practical purposes, the algorithm operates in constant time. See Section ??.

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 search algorithms of Section 5.5 go one step further and search in a multiresolution 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 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 dispersion completeness. 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).

Notions of completeness It is useful to define several notions of completeness for sampling-based algorithms. An algorithm is considered *complete* if for any input it correctly reports whether or not there is a solution in a finite amount of time. If there is a solution, it must return it. Unfortunately, completeness cannot be achieved with sampling-based planning. If α is a deterministic, dense sequence, then the refinement scheme described so far produces a *dispersion complete* algorithm. This means that if a solution exists, then the algorithm will find it; however, if no solution exists, then the algorithm will run forever. If it terminates early without finding a solution, it may declare that either no solution exists, or if the solution exists, it requires sampling with a smaller dispersion. This implies that the path must travel through a narrow passage. A special case of dispersion completeness is when a multiresolution grid or lattice is used. In this case, an algorithm may be called *resolution complete*. Finally, if α is a random sequence that is dense with probability one, then the resolution algorithm is *probabilistically complete*. This means that with enough points, the probability that it will find a solution converges to one. The most relevant information, however, is the rate at which the convergence occurs. This is usually very difficult to establish.

5.4.3 Randomized Potential Fields

Adapting the classical algorithms, as described in Section 5.4.2, 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 dimension, 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 the 50^{10} samples. Planners that exploit best-first heuristics might find the answer without searching most of them; however, for a simple problem

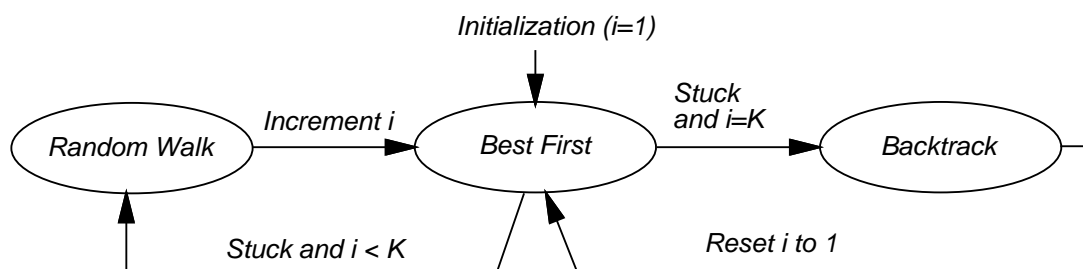


Figure 5.19: The randomized potential field method can be modeled as a three-state machine.

such as that shown in Figure 5.14, the planner will take too long exploring the nodes in the bowl.⁹

The *randomized potential field* approach uses random walks to attempt to escape local minima when best-first search becomes stuck [51, 53, 437], was one of the first sampling-based planners that developed specialized techniques beyond classical 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.¹⁰

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 of any configuration from the goal. In most formulations, there is an *attractive* term that is just a metric on \mathcal{C} which yields distance to the goal, and a *repulsive* term, which penalizes robot as it gets too close to obstacles. The construction of potential functions involves many heuristics and is covered in great detail in [437]. One of the most effective methods involves constructing cost-to-go functions over \mathcal{W} and lifting them to \mathcal{C} [52]. 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 2.3.2. In this case, however, there is no requirement that $g(q)$ is optimal or even an underestimate of the true cost to go.

When a random walk is needed, it is executed for some number of iterations.

⁹Of 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.

¹⁰The exciting results obtained by the method also helped inspire me to work in motion planning.

Using the discretization procedures of Section 5.4.2, a 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 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 5.19. Each state will be 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 may be applied. The method for generating random samples from 5.2.2 can be used. 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.

In the RANDOM WALK mode, a random walk is executed from the newest node. 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. If BEST FIRST fails after K random walks have been attempted, then the BACKTRACK mode is entered. The K is another parameter (a typical value is $K = 20$ [52]). The BACKTRACK mode selects a vertex at random from among the vertices in V there 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 (or geodesic). For example, suppose $t_1, t_2 \in [0, 1]$ are chosen at random and $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$ is the solution path. This path is transformed into a new path

$$\tau'(t) = \begin{cases} \tau(t) & \text{if } 0 \leq t \leq t_1 \\ a\tau(t_1) + (1-a)\tau(t_2) & \text{if } t_1 \leq t \leq t_2 \\ \tau(t) & \text{if } t_2 \leq t \leq 1 \end{cases}, \quad (5.37)$$

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 allowed 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 [544, 543]. There are two modes, SEARCH and EXPLORE, which alternate over successive iterations. In the EXPLORE mode, the VSM simply 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 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 next approach and the methods in Section 5.5 is that some of the time must be spend 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 some time on pure 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 [543], which are generally avoided for motion planning because of the required problem-specific parameter tuning.

Expansive space planner This method [344, 670] 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 algorithms, as covered in Section 2.3.3. The VSM selects a vertex, v_e , in G with a probability that is inversely proportional to the number of other vertices of 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 G with a probability that

is inversely proportional to the number of other vertices of G that lie within a predetermined neighborhood of v_n . For a fixed number of iterations, the VSM will repeatedly choose 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 is essentially tries to determine the appropriate resolution.

Random walk planner A surprisingly simple and efficient algorithm can be made entirely from random walks [127]. 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 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 would be interesting to combine adaptive random walks with other search algorithms, such as the potential field planner, but this has not been attempted to date.

5.5 Rapidly-Exploring Dense Trees

This section introduces an incremental sampling and search approach that yields good performance in practice without any parameter tuning.¹¹ 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 will densely cover the space. Thus, it has properties similar to space filling curves [668], 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 will be 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 the space is obtained. This method was originally developed for problems with differential constraints [463, 466]; that case is covered in Section 15.3.3.

¹¹The original RRT [449] 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   $G.\text{init}(q_0)$ ;
2  for  $i = 1$  to  $k$  do
3       $G.\text{add\_vertex}(\alpha(i))$ ;
4       $q_n \leftarrow \text{NEAREST}(S, \alpha(i))$ ;
5       $G.\text{add\_edge}(q_n, \alpha(i))$ ;

```

Figure 5.20: The basic algorithm for constructing RDTs (including RRTs) when there are no obstacles. It requires the availability of a dense sequence, α , and iteratively connects from $\alpha(i)$ to the closest point among all those reached by G .

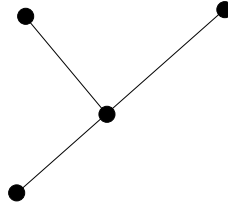


Figure 5.21: Suppose inductively that the following tree has been constructed so far using the algorithm in Figure 5.20.

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 will work for any dense sequence. Therefore, let α denote an infinite, dense sequence of samples in \mathcal{C} . The i^{th} sample is denoted by $\alpha(i)$. Let this also include a uniform, random sequence, which is 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 will actually be a topological graph, $G(V, E)$. Let $S \subset \mathcal{C}_{\text{free}}$ indicate the set of all points reached by G . Since each $e \in E$ is a path, this can be expressed as

$$S = \bigcup_{e \in E} e([0, 1]), \quad (5.38)$$

in which $e([0, 1]) \subseteq \mathcal{C}_{\text{free}}$ is the image of the path e .

The exploration algorithm is first explained in Figure 5.20 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 $\alpha(i)$ to its closest point on 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 5.21-5.23 illustrate an iteration graphically. Suppose the tree has 3 edges and 4 vertices, as shown in Figure 5.21. If the nearest point, $q_n \in S$, to $\alpha(i)$ is a vertex, as shown in Figure 5.22, then an edge is made from

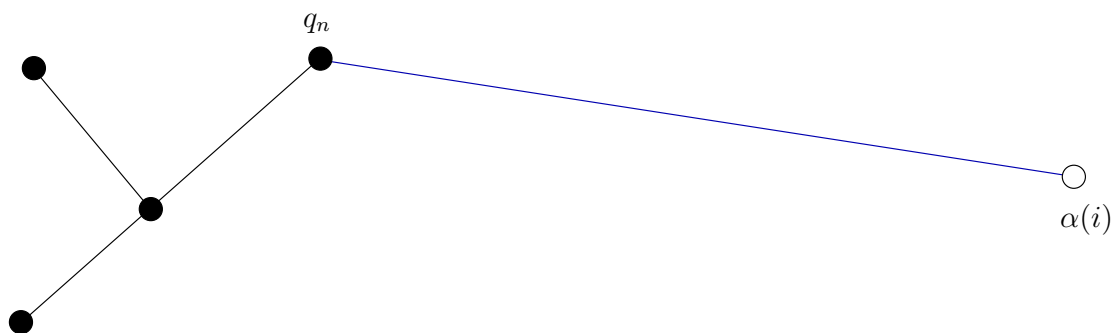


Figure 5.22: A new edge is added, which connects from the sample $\alpha(i)$ to the nearest point in S , which is the vertex q_n .

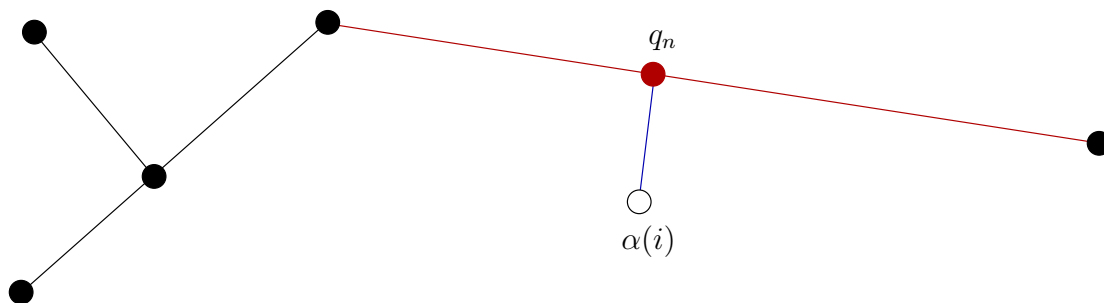


Figure 5.23: If the nearest point S lies in an edge, then the edge is split into two, and a new vertex is inserted into G .

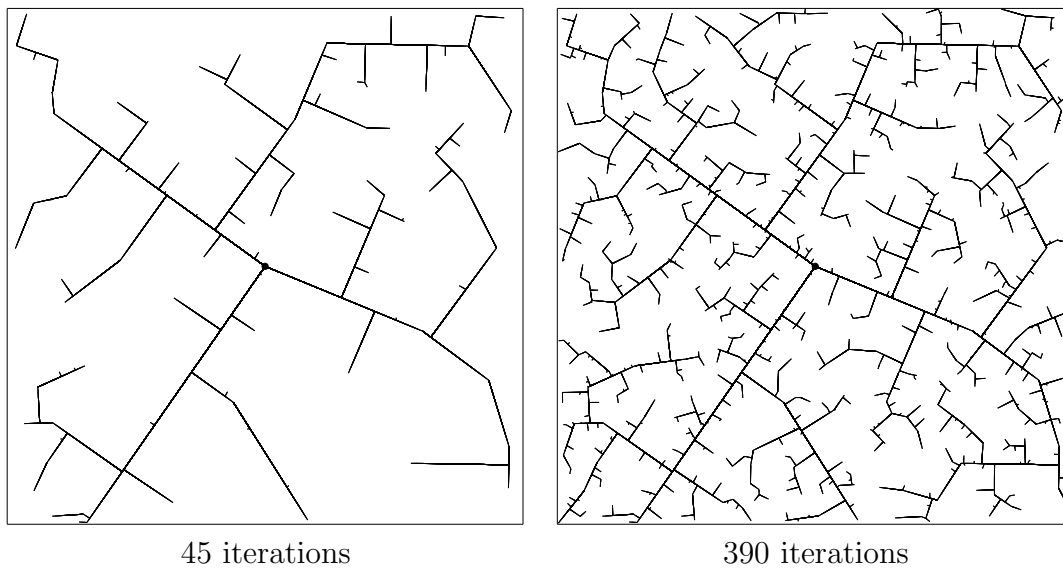
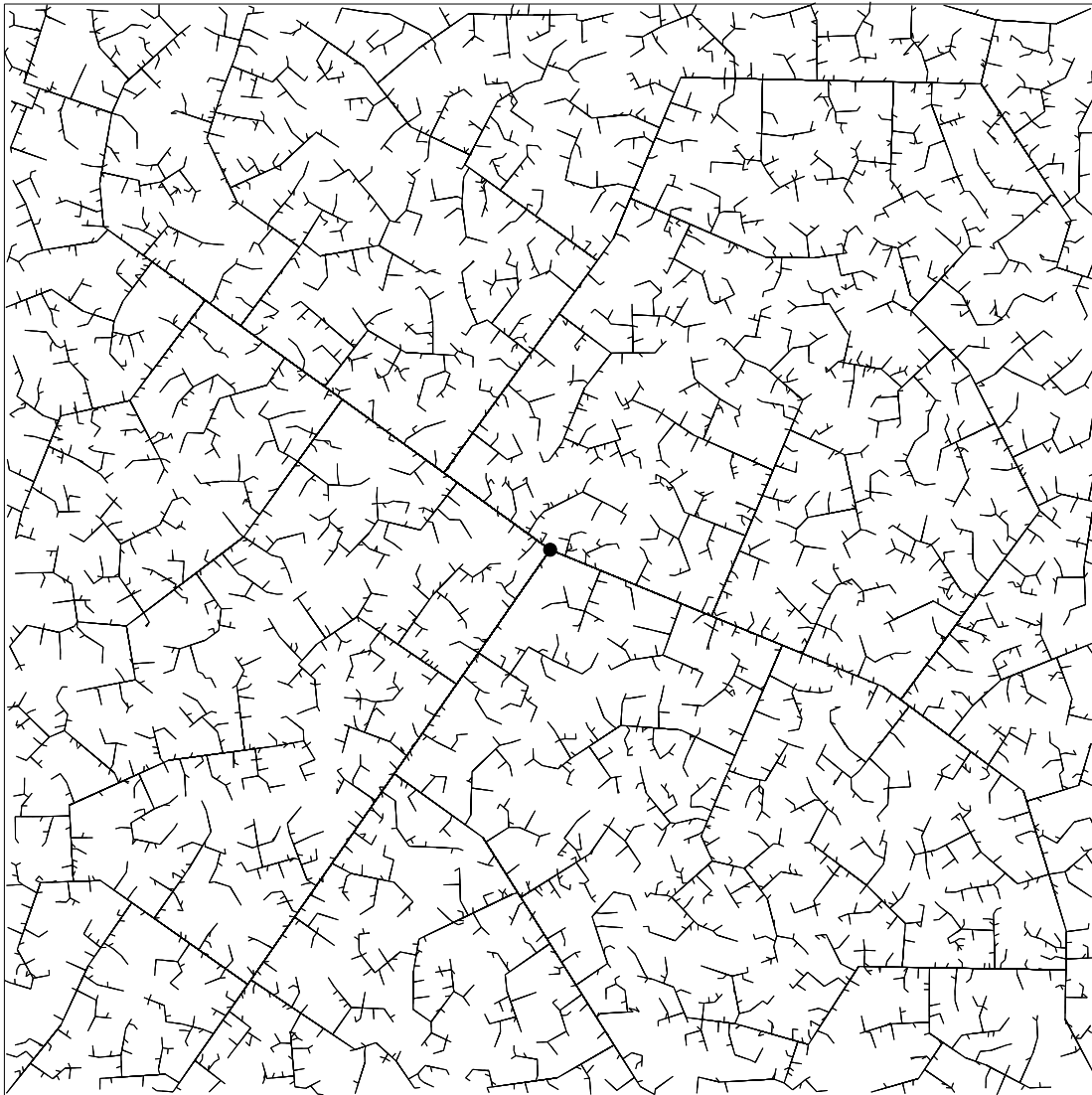


Figure 5.24: The RRT quickly reaches the unexplored parts.



2345 iterations

Figure 5.25: The RRT is dense in the limit (with probability one).

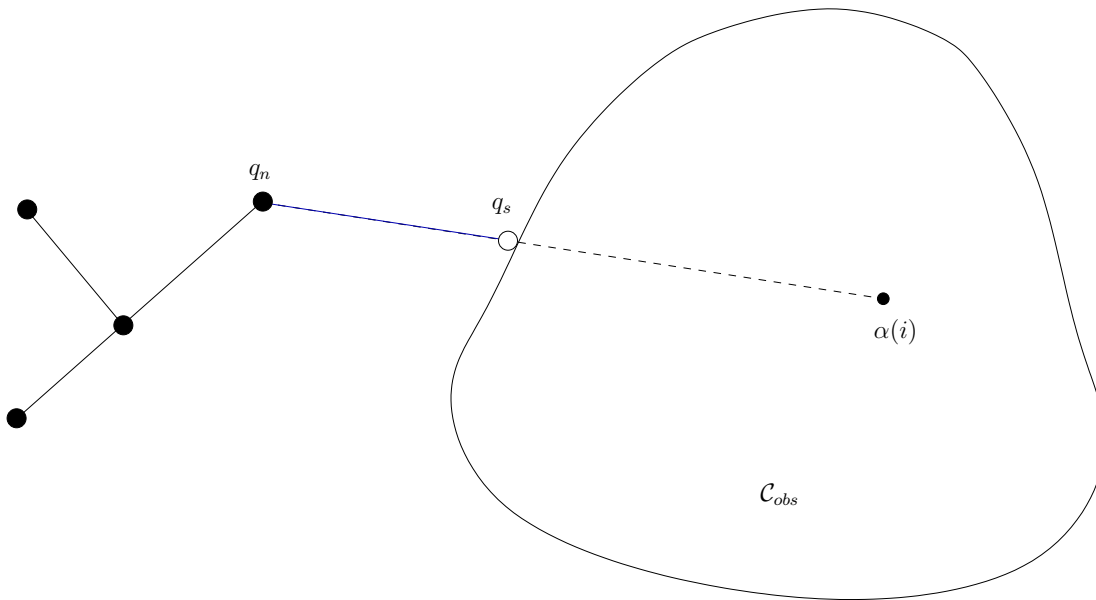


Figure 5.26: If there is an obstacle, the edge travels up to the obstacle boundary, as far as allowed by the collision detection algorithm.

q_n to $\alpha(i)$. However, if the closest point lies in the interior of an edge, as shown in Figure 5.23, 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 method as described here does not fit precisely under the general framework from Section 5.4.1; however, with modifications suggested in Section 5.5.2, it can be adapted to fit. In the present formulation, the NEAREST functions serves the purpose of the VSM, but in this case, a point may be selected from anywhere in the interior of an edge, in addition to a vertex. The LPM tries to connect $\alpha(i)$ to q_n along the shortest path possible in \mathcal{C} .

Figures 5.24 and 5.25 show an implementation of the algorithm in Figure 5.20 for the case in which $\mathcal{C} = [0, 1]^2$, and $q_0 = (1/2, 1/2)$. It exhibits a kind of fractal behavior.¹² 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 will densely fill 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 5.26 indicates how to modify the algorithm in Figure 5.20 so that collision checking is taken into account. The pseudocode for the modi-

¹²If α is uniform, random, then a *stochastic fractal* [435] is obtained. Deterministic fractals can be constructed using sequences that have appropriate symmetries.

```

RDT( $q_0$ )
1   $G.\text{init}(q_0)$ ;
2  for  $i = 1$  to  $k$  do
3       $q_n \leftarrow \text{NEAREST}(S, \alpha(i))$ ;
4       $q_s \leftarrow \text{STOPPING-CONFIGURATION}(q_n, \alpha(i))$ ;
5      if  $q_s \neq q_n$  then
6           $G.\text{add\_vertex}(q_s)$ ;
7           $G.\text{add\_edge}(q_n, q_s)$ ;

```

Figure 5.27: The RDT with obstacles.

fied algorithm appears in Figure 5.27. The procedure STOPPING-CONFIGURATION yields the closest configuration possible to the boundary of \mathcal{C}_{free} , along the direction toward $\alpha(i)$. The closest 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 5.20. There are generally two families of methods: *exact* or *approximate*. First consider the exact case.

Exact solutions Suppose that all edges in 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 5.24. 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 computations. Using this primitive, NEAREST is implemented by iterating over all of the 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 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 .

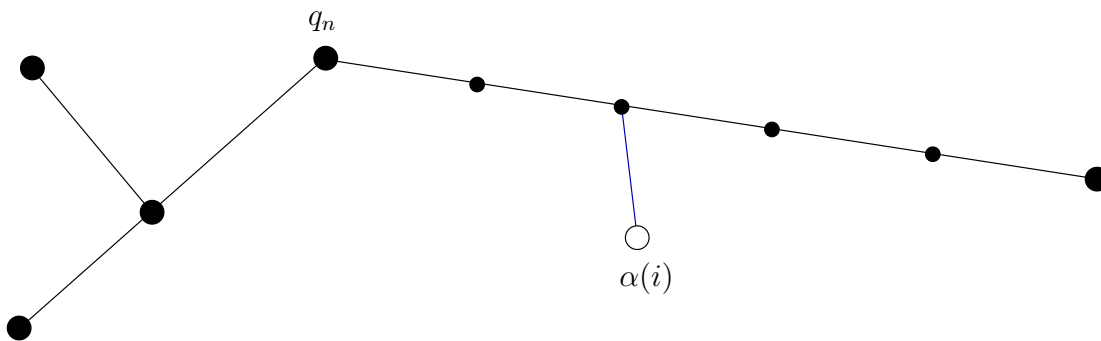


Figure 5.28: For implementation ease, intermediate vertices can be inserted to avoid checking for close points along line segments. The tradeoff is that the number of vertices is increased.

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 . Then straight line segments can 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. Because of antipodal identification, only 4 of the 8 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 5.28. The intermediate vertices should be added each time a new sample, $\alpha(i)$, is inserted into G . A parameter Δq can be defined, and intermediate samples are inserted to ensure that no two consecutive vertices in G are ever further than Δq from each other. Using intermediate vertices, the interiors of the edges in 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 G . This approach is by far the simplest to implement (in fact, it was done to obtain the results in

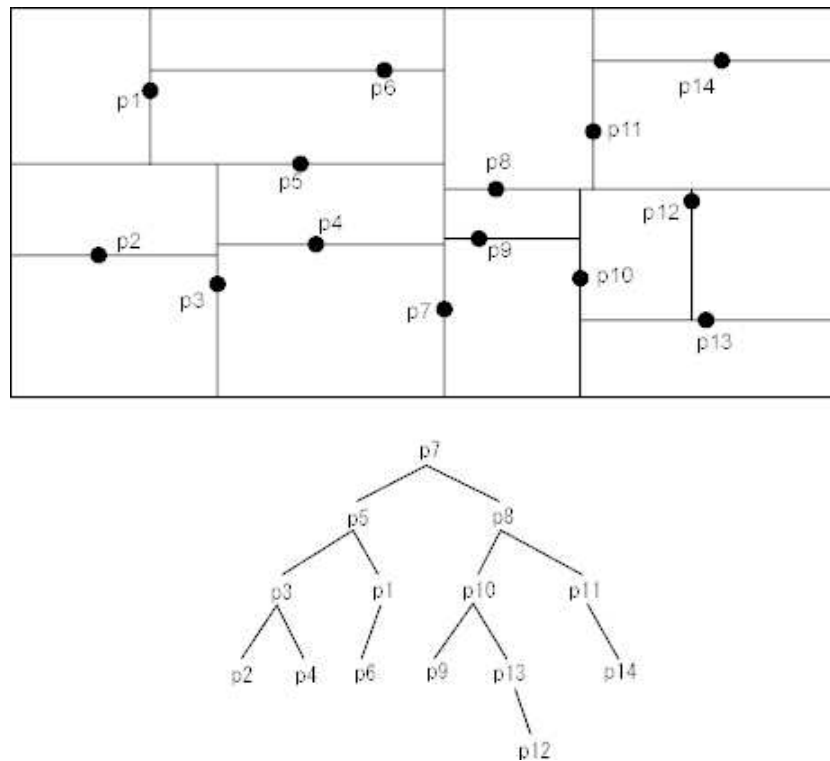


Figure 5.29: The Kd-tree can be used for efficient nearest neighbor computations.

Figure 5.24). It also fits precisely under the incremental sampling and searching framework from Section 5.4.1.

When using intermediate vertices, the tradeoffs 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 also dramatically increases running time. One way to recover some of this cost of the insert the vertices into an efficient data structure for nearest-neighbor searching. One of the most practical and widely-used data structures is the Kd-tree [189, 263, 599]. A depiction is shown in Figure 5.29 for 14 points in \mathbb{R}^2 . The Kd-tree can be considered as a multidimensional 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 the 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 forth. 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 [32], 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. The Kd-tree can be constructed in $O(n \lg k)$

time. The topology must be carefully considered when traversing the tree. When a query is made, a point, $q \in T$, is given, and the closest point to q is found. At first the query algorithm descends to a leaf node which contains the query point, finds all distances from the data points in this leaf to the query point, and picks up the closest one. Then, it recursively visits those surrounding leaf nodes which are further from the query point than the closest point found so far [32]. The nearest point can be found in time logarithmic in k .

Unfortunately, these bounds hide a constant that increases exponentially with 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 tradeoffs 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. There are many ways that RRTs and RDTs in general can be used 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 5.27, if the sequence α contains the appropriate bias. If the sample sequence is random, which generates an RRT, then the following modification will work well. In each iteration, toss a biased coin that has probably 49/50 of being HEADS, and 1/50 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 will force the RDT to occasionally attempt making a connection to the goal, q_g . Of course, 1/50 is arbitrary, but it is in a range that works well experimentally. If the bias is too strong, then the RDT will become too greedy like the randomized potential field. If the bias is not strong enough, then there will be no incentive to connect the tree to q_g .

If $\alpha(i)$ is a deterministic sequence, then q_g can be selected with a fixed frequency. For example, the Halton sequence can be used, but for every positive integer i , q_g is inserted into the Halton sequence between points $50i$ and $50i + 1$. Thus, in every 50th iteration, the RDT will attempt to connect to the goal. Of course, the fixed frequency could also be combined with the random sampling.

Other variations can be made by using a dense, but nonuniform sequence in \mathcal{C} . For example, in the case of random sampling, the probability density function

```

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       $q_n \leftarrow$  NEAREST( $S_a, \alpha(i)$ );
4       $q_s \leftarrow$  STOPPING-CONFIGURATION( $q_n, \alpha(i)$ );
5      if  $q_s \neq q_n$  then
6           $T_a$ .add_vertex( $q_s$ );
7           $T_a$ .add_edge( $q_n, q_s$ );
8           $q'_n \leftarrow$  NEAREST( $S_b, q_s$ );
9           $q'_s \leftarrow$  STOPPING-CONFIGURATION( $q'_n, q_s$ );
10         if  $q'_s \neq q'_n$  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 5.30: A bidirectional RDT-based planner.

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 ¹³

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, the 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 [], which will ensure that both trees are the same size.

Figure 5.30 gives an outline of the algorithm. The graph, 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 will be 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 5.20. 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 will cause T_b to try to grow towards T_a . If the two connect, which is tested in Line 13, then a solution has been found.

¹³This particular planner is due to an unpublished collaborative effort with James Kuffner.

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 5.15 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 5.20, 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 through 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 without 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 5.16. 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 a difficult. Some promising experimental work has been done in this direction, but it currently requires substantial parameter tuning [62].

These additional trees could be started at arbitrary (possible 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 leads to 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 that numerous initial-goal queries will be given 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 invent substantial time to preprocess the models so that future queries can be answered efficiently. The goal is to construct a *roadmap* that can be used

```

BUILD_ROADMAP
1  G.init();
2  for i = 1 to N
3      if  $\alpha(i) \in \mathcal{C}_{free}$  then
4          G.add_vertex( $\alpha(i)$ );
5          for each  $q \in \text{NEIGHBORHOOD}(\alpha(i), G)$ 
6              if ((not G.same_component( $\alpha(i), q$ )) and CONNECT( $\alpha(i), q$ )) then
7                  G.add_edge( $\alpha(i), q$ );

```

Figure 5.31: The basic construction algorithm for sampling-based roadmaps.

to efficiently solve queries. Intuitively, the paths on the roadmap will be easy to reach from each of q_i and q_g , and the network of paths in the roadmap can be quickly searched for a solution. The general framework presented here was mainly introduced in [387] 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 $G(V, E)$ 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 G in a way that will be useful for quickly answering future queries. For this reason, it is called a *roadmap*, which in some sense should be capable of reaching 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 G using a local planner. Following this, a discrete search is performed using any of the algorithms in Section 2.3 to obtain a sequence of edges that forms a path from q_i to q_g .

Generic preprocessing phase Figure 5.31 presents an outline of the basic preprocessing phase. Figure 5.32 illustrates the algorithm. As seen throughout this chapter, the algorithm utilizes a uniform, dense sequence α . In each iteration, the algorithm must check whether $q \in \mathcal{C}_{free}$. If $q \in \mathcal{C}_{obs}$, then it must continue to iterate until a collision-free sample is found. Once $\alpha(i) \in \mathcal{C}_{free}$, then it is inserted into G , in Line 4. The next step is to try to connect $\alpha(i)$ to some nearby vertices, q , of 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 will simply test the shortest path between $\alpha(i)$ and q . Experimentally, it seems

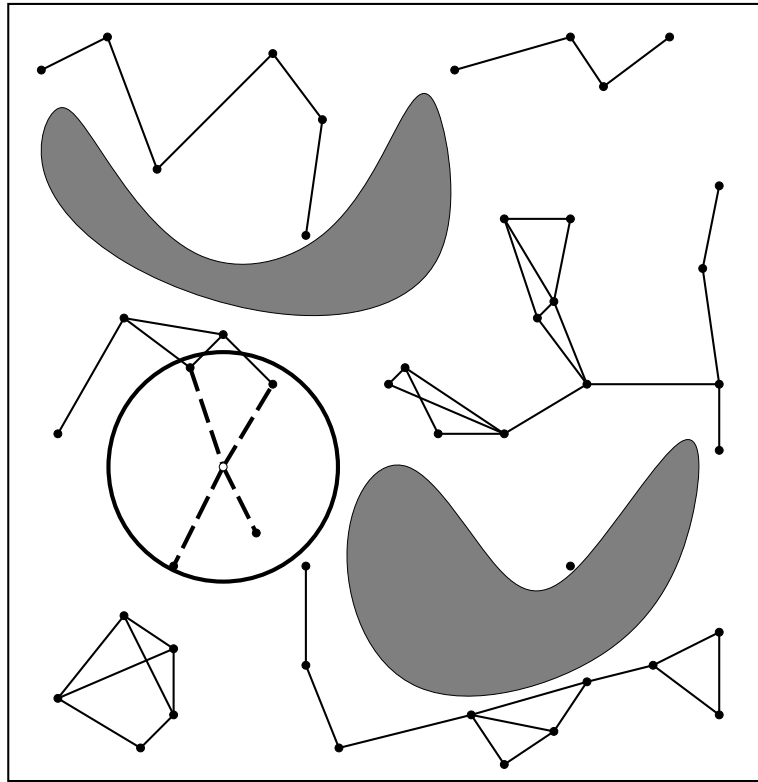


Figure 5.32: The sampling-based roadmap is constructed incrementally by attempting to connect each new sample, $\alpha(i)$, to nearby vertices in the roadmap.

most efficient to use the multiresolution, van der Corput-based method described at the end of Section 5.3.4 [272]. Instead of the shortest path, it is possible to use more sophisticated connection methods, such as the bidirectional algorithm in Figure 5.30. 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 G before wasting time on collision checking. This will ensure that every time a connection is made, the number of connected components of G is decreased. This can be implemented very efficiently (near constant time) using the previously-mentioned *union-find* algorithm [176, 655]. In some implementations this step may be ignored, especially if it is important to keep multiple solutions. For example, it may be desirable to generate solution paths from different homotopy classes. In this case the condition (`not G.same_component($\alpha(i)$, q)`) may be replaced with `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 in increasing distance from $\alpha(i)$. This makes

sense because shorter paths are usually less costly to check for collision, and they also have a high 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 G . A reasonable value is $K = 1$ in this case; 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 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.

Query phase In the query phase, it is assumed that 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 algorithm pretends as if q_i and q_g were chosen from α for connection to G . This requires running two more iterations of the algorithm in Figure 5.31. If q_i and q_g are successfully connected to other vertices in 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 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 [453].

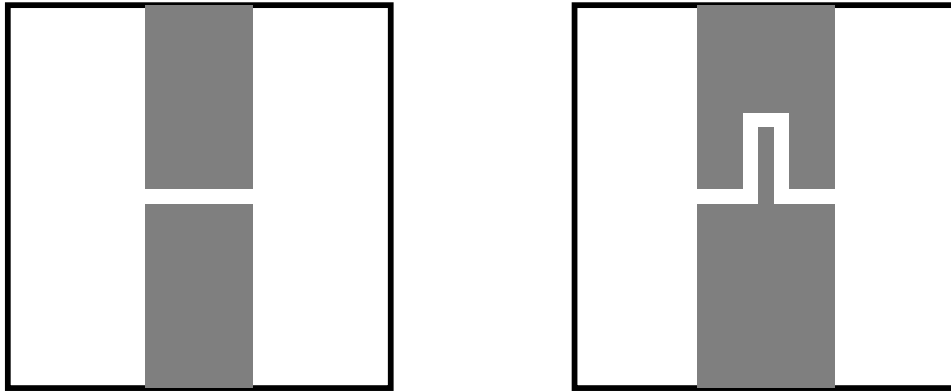


Figure 5.33: Examples such as these are difficult because of the narrow corridor that links two portions of \mathcal{C}_{free} .

Some analysis There have been many works that analyze the performance of sampling-based roadmaps. The basic idea from one of them [49] is briefly presented here. Consider problems such as those in Figure 5.33, in which the CONNECT method will mostly likely fail, 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 will generally cause 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 through a corridor is hard (unless more-sophisticated LPMS are used).

Let $V(q)$ denote the set of all configurations that can be connected to q using the CONNECT method. Intuitively, this can be considered as the set of all configurations that can be $V(q)$ “seen” using line-of-sight visibility, as shown in Figure 5.34.a

The ϵ -goodness of \mathcal{C}_{free} is defined as

$$\epsilon = \min_{q \in \mathcal{C}_{free}} \frac{\mu(V(q))}{\mu(\mathcal{C}_{free})}, \quad (5.39)$$

in which μ represents the measure. Intuitively, ϵ represents the small fraction of \mathcal{C}_{free} that is visible from any point. In terms of ϵ and the number of vertices in G , bounds can be established that yield the probability that a solution will be found [49]. 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 are interesting for gaining a better understanding of sampling-based planning, but such bounds are difficult to use in a particular application to determine whether an algorithm will perform well.

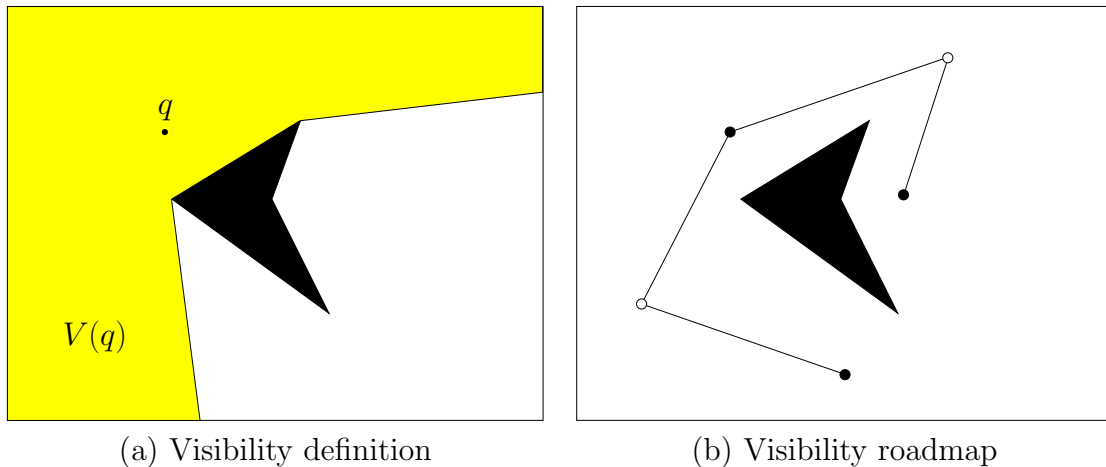


Figure 5.34: a) $V(q)$ is the set of points reachable by the LPM from q . b) A visibility roadmap has two kinds of vertices: guards, which are shown in black, and connectors, shown in white. Guards are not allowed to see other guards. Connectors must see at least two guards.

5.6.2 Visibility Roadmap

One of the most interesting variations of sampling-based roadmaps is the *visibility roadmap* [705]. 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 5.31, but the extra expense is usually worthwhile if the multiple query philosophy is taken to its fullest extent.

The idea is to define two different kinds of vertices in G :

Guards: To become a *guard*, a vertex, q must not be able to see over guards. Thus, the visibility region, $V(q)$, must be empty of guards.

Connectors: To become a *connector*, a vertex, q , must see at least two guards. Thus, there exists 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 5.31. The *neighborhood* function returns all vertices in G . Therefore, for each new sample $\alpha(i)$, an attempt is made to connect it to every other vertex in G .

The main novelty of the visibility roadmap is that a strong criterion exists to determine whether to keep $\alpha(i)$ and its associated edges in 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 G .
2. The new sample can connect to guards from at least two different connected components of G . In this case, it becomes a connector, and is inserted into G along with its associated edges that 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.

Figure 5.35 shows the dramatic reduction in the number of vertices for two different examples.¹⁴ Each column from top to bottom shows the problem, a basic sampling-based roadmap, and the visibility roadmap. The first example is for a point robot, and the second example is for a rectangular robot that can translate or rotate.

One problem with the method described 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 could allow guards to be reassigned in later iterations as better positions become available. Accomplishing this efficiently remains an open problem.

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 specific to the shape of the configuration 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.

Original node enhancement [387] This heuristic strategy focuses effort on nodes that were difficult to connect to other nodes in the roadmap construction algorithm in Figure 5.31. 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 5.31. A recommended heuristic [387] 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 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 serves to normalize the statistics to obtain a valid probability distribution).

Sampling on the \mathcal{C}_{free} boundary [13, 16] This scheme is based on the intuition that it is sometimes better to sample along the boundary, $\partial\mathcal{C}_{free}$, rather than

¹⁴These examples are taken from a class project of Andrew Olson and Kevin Crotty completed at Iowa State University.

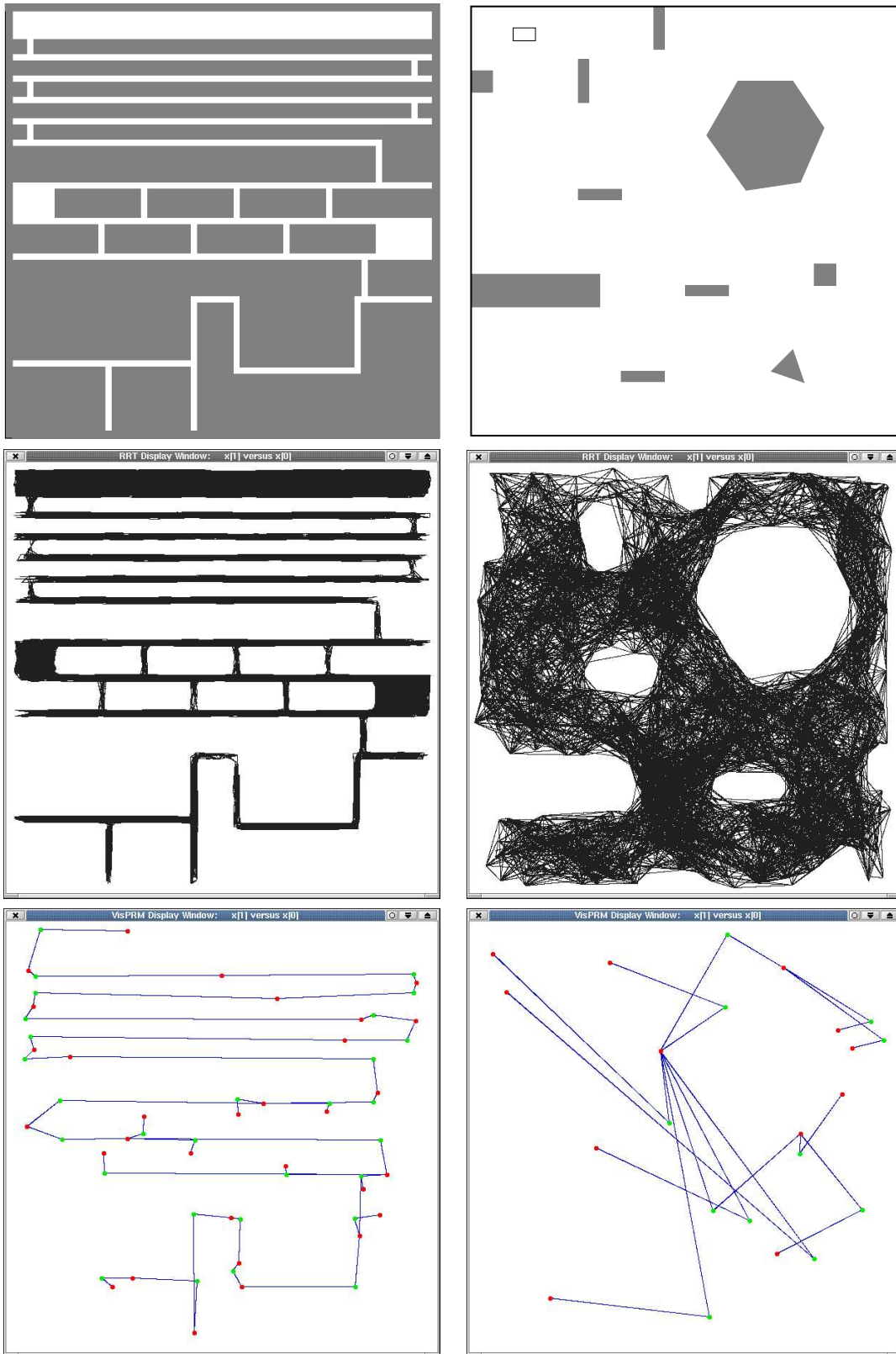


Figure 5.35: The visibility roadmap is more costly to construct, but can dramatically reduce the number of vertices.

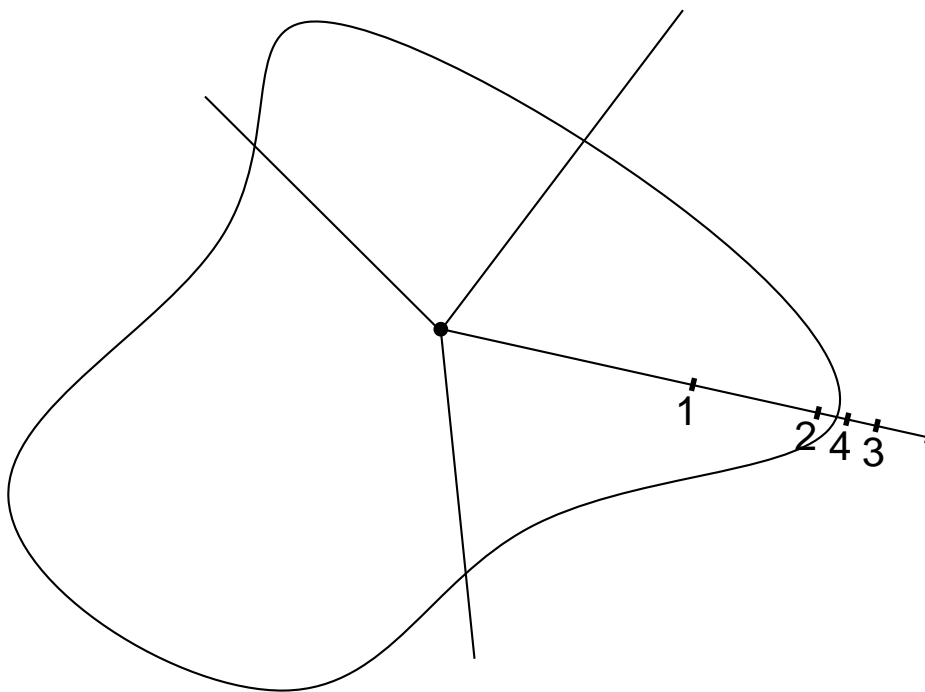


Figure 5.36: To obtain samples along the boundary, binary search is used along random directions from a sample in \mathcal{C}_{obs} .

wasting samples on large areas of \mathcal{C}_{free} that might be free of obstacles. Figure 5.36 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 method in 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 5.36. 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 iterations 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 continuously recursively until the desired resolution is obtained.

Gaussian sampling [86] 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 in which biases the samples to

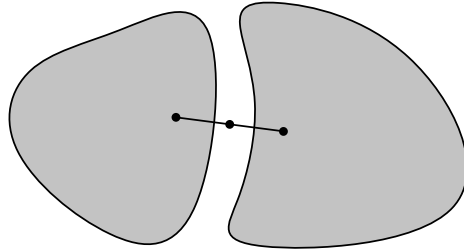


Figure 5.37: The bridge test finds narrow corridors by examining a triple of samples.

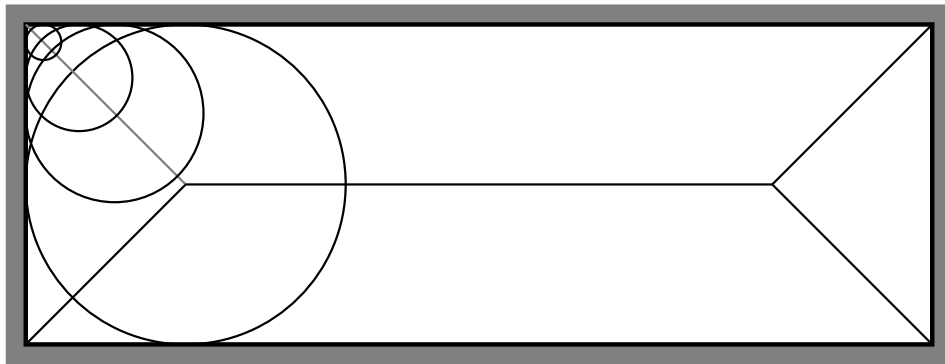


Figure 5.38: The medial axis is traced out by the centers of the largest inscribed balls. The five line segments inside of the rectangle correspond to the medial axis.

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 used as a vertex in the roadmap. For some examples, this dramatically prunes the number of required vertices.

Bridge test sampling [341] The Gaussian sampling strategy decides to keep a point based on 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 5.37, then the middle sample becomes a vertex. This is based on the intuition that narrow corridors are thin in at least one direction. The bridge test indicates that there a corridor is thin, while is a difficult and important place to locate a vertex.

Medial axis sampling [332, 490, 774] 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 5.38. The medial axis in \mathcal{C}_{free} is based on the largest balls that can be inscribed in $cl(\mathcal{C}_{free})$. 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 $\partial\mathcal{C}_{free}$ [490, 774]. Sampling on the medial axis of $W \setminus \mathcal{O}$ has also been proposed [332]. In this case, the medial axis in $W \setminus \mathcal{O}$ is easier to compute, and can be used to heuristically guide the placement of good roadmap vertices in \mathcal{C}_{free} .

Literature

Explain [81] somewhere.

Should say something about disconnection proofs.

Need to cite the exact collision detection method of Latombe et al from WAFR 2002.

The following is from the section entitled “The Rise of Sampling-Based Motion Planning” in an ISSR 2003 paper coauthored with Steve Lindemann. It needs to be shortened here.

To fully understand the continuing evolution of sampling-based motion planning and its current issues, it is helpful to understand how sampling-based algorithms have developed and changed over time. In this section, we will describe how sampling-based algorithms began to emerge, and how they have continued to develop up to the present time.

In the 1980s, constructing a representation of \mathcal{C}_{obs} , either completely or in part, was the predominate approach to motion planning. Examples include the planner by Brooks and Lozano-Pérez for a polygon rotating and translating in the plane [102], work by Donald for planning for a 3D rigid body [205, 207], and a planner by Lozano-Pérez for manipulator arms [505]. References to many combinatorial planners and a few early sampling-based ones can be found in Hwang and Ahuja’s survey [357]. Glimpses of sampling-based motion planning began to emerge in the late 1980s. These algorithms typically centered around advances in efficient calculation of distance between polyhedra. Faverjon and Tournassoud introduced a manipulator planner which computed local collision-free motions using distance computation and hierarchical CAD models [243, 242]. The introduction of algorithms such as the Gilbert-Johnson-Keerthi algorithm [279] made sampling-based approaches more common. A good example of an approach is the manipulator planner of Paden *et al.* [602]. They create a 2^d -tree representation of the configu-

ration space, labelling cells as “freespace,” “obstacle,” or “not sure or mixed.” To classify cells correctly (or at least, conservatively), they find the uniform bound on the Jacobian for the given manipulator. Then, based on this information and the workspace distance returned by the GJK algorithm, they can determine whether or not an entire cell can be classified as freespace or obstacle. If neither apply, then the cell is labelled mixed and will be subdivided, if a predefined minimum resolution has not yet been reached. After preprocessing the environment in such a way, it is simple to find a path, if one exists in the tree, or to determine that greater resolution is required to resolve small mixed cells.

The use of distance information from a collision detector permits hierarchical grid-based approaches as in Paden *et al.*, but computing this information is more expensive than simply returning the boolean result of an intersection test (the most basic form of collision detection). A less-expensive grid-based approach might discretize the space at a sufficiently fine resolution and use an inexpensive collision detection method to determine whether each cell belongs to \mathcal{C}_{free} , thus creating a bitmap of C-space. The resulting data structure can then be searched by classical AI search techniques to find a path, if one exists. In fact, this very approach was taken by Lengyel *et al.* [479]. Their algorithm uses graphics hardware to plan for a polygonal robot translating and rotating in the plane. They divide the rotational degree of freedom, θ , into a number of slices, and use graphics hardware to calculate the Minkowski sum of the robot and obstacles for a particular value of θ . They combine all resulting slices and have a bitmap representation of the three-dimensional C-space, which they then search with a dynamic programming technique.

In general, however, this kind of approach is limited to lower dimensions since the number of resultant grid cells grows exponentially with the number of DOFs of the problem, and the a fine resolution is required. Hence, checking them all for collision becomes impractical. Nevertheless, when general sampling-based motion planning algorithms began to proliferate in the early 90’s, several of these were clearly influenced by the grid search approach. We will consider two of this type, along with two other early sampling-based algorithms, before describing several more recent, state-of-the-art sampling-based motion planners.

One early planner that strongly reflects classical grid search techniques is that of Kondo [406]. Kondo’s planner is based on the observation that even if a fine grid is placed over the configuration space, it may be possible to find a solution without visiting large portions of that grid. Hence, if one delays collision checking until needed—a “lazy” approach—only (relatively) few collision checks will need to be performed, thus avoiding the expensive preprocessing step of naive grid search. The planner searches a grid bidirectionally, assigning cost $f(C) = g(C) + h(C)$ to each expanded grid cell, in which $g(C)$ is the standard cost-to-come and $h(C)$ is a heuristic weighted sum-of-squares cost. Kondo’s planner uses multiple heuristics (i.e., different assignments of the heuristic weight constants), and adaptively selects between them based on an estimate of their effectiveness. Hence, the effec-

tiveness of the planner strongly depends on the quality of the heuristic functions, and on the planner's ability to choose the appropriate one to apply. If either of these are poor, then performance will degrade greatly. Kondo gives several six-dimensional examples, with the resolution of the grid being 2^7 points per axis yielding 2^{42} total grid cells. However, for the results reported, typically less than 20000 collision checks were needed to solve the problem. The influence of Kondo's multiple-heuristic approach can be seen in recent PRM-related work by Isto [362].

In 1990, Barraquand and Latombe introduced the planner that came to be called the Randomized Path Planner [51]. This planner is important for three primary reasons: first, it was the perhaps the first well-known sampling-based motion planner; second, it solved problems with many DOFs, typically many more than other planners at the time were capable of handling; and third, it advocated randomization as a means of efficiently finding solutions in the high-dimensional configuration space. Its influence in this third respect can hardly be overestimated, since for the following decade virtually every significant sampling-based motion planning algorithm used randomization. In fact, only recently has the role of randomization in sampling-based motion planning begun to be studied in depth. We will discuss this issue in some depth in subsequent sections. RPP operates as follows: first, the planner defines several potential fields over a grid imposed on the workspace; each potential field corresponds to a "control point" on the robot. A finer-resolution grid is also defined over the configuration space, and the potential value of each configuration-space grid cell is defined by the following non-negative, real-valued function on C_{free} :

$$U(q) = G(U_{p_1}(X(p_1, q)), \dots, U_{p_n}(X(p_n, q))),$$

in which p_1, \dots, p_n are the control points, X is a function mapping a point on the the robot to its position in the workspace at the given configuration, and G is an arbitration function. Then, beginning at the initial state, the planner descends the gradient of the C-space potential field, until a local minimum is reached. If the minimum is the global minimum, the goal state has been attained; else, the planner executes a series of random walks with the aim of escaping the local minimum. After this, the planner again descends the potential field gradient, continuing this process until the goal state has been reached or a user-specified amount of time has elapsed. This latter condition is necessary because unlike combinatorial planners, sampling-based planners are typically unable to recognize that a problem has no solution; in such a situation, they will never terminate. The key to this planner's performance is the construction of good potential fields and a good arbitration function, which can be quite difficult to construct in practice. If the potential fields result in many local minima, the planner can perform poorly.

Another early sampling-based motion planner is the SANDROS planner of Chen and Hwang [140], which was developed for manipulator arms. This planner searches in a multi-resolution manner over a non-uniform grid (i.e., the resolution on the coordinate axes may differ). The axes are given different resolutions because

for manipulator arms, links near the base have the greatest impact on end effector position. Just as Paden *et al.*, this algorithm uses the GJK algorithm [279] for collision detection. It also uses the distance information to place links of the arm at maximal distance from the workspace obstacles.

Finally, a planner (later termed the ZZ-method) was introduced by Glavina in 1990 [281] which foreshadows PRMs in many respects. The ZZ-method first attempts to connect the initial and goal queries using a “straight-and-slide” local planner (a method which does not allow backtracking but is more powerful than the straight-line local planner). If this fails, which is usually the case, then a new configuration is chosen as a subgoal (Glavina advocates using jittered sampling), and attempts to connect the subgoal to the initial and goal configurations using the same local planner. If this fails, new subgoals are added and attempts are made to connect them with previously existing subgoals, as well as the initial and goal configurations. Edges between subgoals are checked for collisions at a pre-defined subsampling resolution. Glavina also identifies the well-known “narrow corridor” problem and uses connected component analysis to speed up his planner. However, he uses a primitive collision detection method which prevents him from applying his algorithm to challenging, high-DOF problems (this was remedied in some extensions of his work [38, 39]); also, the straight-and-slide local planner becomes expensive in high dimensions. In principle, however, the ZZ-method contains many elements which have become common in more recent algorithms.

Since the introduction of these early algorithms, sampling-based motion planning has continued to develop. Changes have been made to deal with failings of previous planners, and new exploration paradigms have been investigated. We discuss four well-known recent motion planning algorithms: PRMs, Ariadne’s Clew, the expansive-space planner by Hsu *et al.*, and RRTs.

In recent years, the most popular paradigm for sampling-based motion planning has the probabilistic roadmap [387]. The original PRM, along with its numerous extensions and variants (e.g., [13, 81, 482, 629, 705, 774, 780]), have been successfully applied to problems in robotics, computer animation, and computational biology [404, 627, 715]. While there is a strong connection to Glavina’s work, there are several important differences. Foremost among these is that the PRM is designed for multiple-queries rather than a single-query. Hence, the placement of landmarks is seen as constructing a reusable roadmap in the PRM method, not as generating query subgoals as in the ZZ-method. Second, the ZZ-method attempted to connect each new landmark (subgoal) to all previous ones; PRMs attempt to connect to a more carefully-chosen subset of these, which is typically the K nearest landmarks from each connected component, or all subgoals within some specified radius. Third, the PRM uses a more simple local planner, often either straight-line or rotate-at-s [18], unlike the ZZ-method’s more expensive straight-and-slide local planner. Finally, methods are used to identify difficult regions of C-space and sample in those regions (the “roadmap enhancement” phase). Along with the use of more sophisticated collision detection methods, these factors make

the PRM more effective for challenging motion planning problems.

Ariadne’s Clew is a single-query algorithm that grows a tree from the initial configuration toward the goal configuration [543, 544]. At each step, it searches for a new “landmark,” reachable from a current landmark by a Manhattan path of a certain order, which is maximally distant from the set of all current landmarks. They use highly-parallelized genetic algorithms to search for a solution to this optimization problem. Once a new landmark has been added to the tree, the planner attempts to connect this new landmark to the goal. To improve performance, when the algorithm encounters an obstacle in trajectory calculation it “bounces” off it. Experimental results give fast solution times for motion of a 6-DOF arm in a dynamic environment. One limitation, however, is the difficult heuristic choices required for the genetic algorithm.

Hsu *et al.* introduced a single-query path planner¹⁵ for “expansive” configuration spaces in [344]. The notion of expansiveness is related to how much of the free space is visible from a single free configuration or connected set of free configurations, and extends the idea of ϵ -goodness [49]. The expansive-space planner grows a tree from the initial configuration. Each node x in the tree has an associated weight, which is defined to be the number of nodes inside $N_d(x)$, the ball of radius d centered at x . At each iteration, it picks a node to extend; the probability that a given node x will be selected is $1/w(x)$, in which w is the weight function. Then, K points are sampled from $N_d(x)$ for the selected node x , and the weight function value for each is calculated. Each new point y is retained with probability $1/w(y)$, and the planner attempts to connect each retained point to the node x . Hence, we see a similarity between this planner and Ariadne’s clew, in that they each try to “push” the tree toward unexplored areas of free space. The main drawback of the approach is that the required d and K parameters may vary dramatically across problems, and they are difficult to estimate for a given problem.

Finally, we describe Rapidly-exploring Random Trees (RRTs) [449, 466], which were developed for problems with differential constraints, such as kinodynamic planning and nonholonomic planning. Its introduction has stimulated a flurry of recent applications and extensions (e.g., [94, 109, 145, 164, 192, 261, 371, 375, 395, 486, 756, 780]). In its basic form, the RRT attempts to grow a tree from the initial configuration to the goal configuration as follows: take a random sample, and find its nearest neighbor in the search tree. Then, grow toward the sample from its nearest neighbor. This process is repeated until the initial and goal configurations are connected. The best-performing RRT planner uses a more greedy connection strategy (at each iteration, attempt to make a complete connection from the nearest neighbor to the sample) and searches bidirectionally. This planner rapidly explores the configuration space because it is Voronoi-biased: at each iteration it tends to grow from the node with the largest Voronoi area. This is because the

¹⁵Some authors refer to this and virtually all planning algorithms that use randomization as PRMs. To avoid confusion, we do not use this term for single-query planners, such as the planner of Hsu *et al.*, even though it is called a PRM by its authors.

probability that a node is selected for expansion is directly proportional to the volume of its Voronoi cell. In contrast to Ariadne's Clew and the expansive-space planner, which work hard to push the tree toward unexplored regions, RRTs are *pulled* into these regions by virtue of the sampling and connection strategy. This avoids the need for complicated parameter tuning, but comes at the expense of performing nearest neighbor queries.

Hierarchical collision detection is covered in [556, 493, 293]. The incremental collision detection ideas are borrowed from the Lin-Canny algorithm [492] and V-Clip [556]. [639, 293, 556, 300, 221] Survey: [493]

Nearest Neighbors:

[28, 27, 263, 401, 599, 721, 359, 788].

Exercises

There are merely sketches of ideas here. Needs to be updated...

1. Show that using uniform mass over \mathbb{S}^3 yields the Haar measure for $SO(3)$.
2. Show some unidimensional dispersion bounds.
3. Construct a bound on distance traveled by points on \mathcal{A} when a quaternion is perturbed.
4. Make up some bug trap examples with real geometry.
5. (Open problem) Prove there are $d + 1$ branches for an RRT in an “infinite” disc.
6. Do something with Cantor sets.
7. Devise a good way to select a subset of neighbors in a high-dim grid.
8. Something with average-case dispersion.
9. Try RRTs with more-powerful descent functions
10. Experiment with visibility pruning in the RRT.

Chapter 6

Combinatorial Motion Planning

Chapter Status



What does this mean? Check
<http://msl.cs.uiuc.edu/planning/status.html>
for information on the latest version.

Combinatorial approaches to motion planning find paths through the continuous configuration space without resorting to approximations. Because of 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, dispersion 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-hard¹; 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.

Reasons to study combinatorial methods Based on these observations, 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 two-dimensional, 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 incomplete 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 about making 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 simple² polygon into triangles in $O(n)$ time for a polygon with n edges [137]. 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 tradeoffs 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.

¹This implies NP-hard. An overview of such complexity statements appears in Section 6.5.1.

²A polygonal region that has no holes.

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 Chapter 6 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 will first construct a cell decomposition of \mathcal{C}_{free} from which the roadmap is consequently derived. Other methods directly construct a roadmap without consideration of cells.

Let G be a topological graph (defined in Example 4.1.6), that maps into \mathcal{C}_{free} . Furthermore, let $S \subset \mathcal{C}_{free}$ be the set of all points reached by G , as defined in (5.38). The graph G is called a *roadmap* if it satisfies two important conditions:

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.

Connectivity Preserving: Using the first condition, it will be 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 will not be missed because the G fails to capture the connectivity of \mathcal{C}_{free} . This ensures that complete algorithms will be 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 and q_g is solved using G by connecting each query point to the roadmap, which relies on accessibility, and then performing a discrete graph search on G , which relies on G being connectivity preserving to ensure that a solution will be found when one 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 which may be modeled as a point moving on a building floor that can be modeled with a 2D polygonal floor plan.

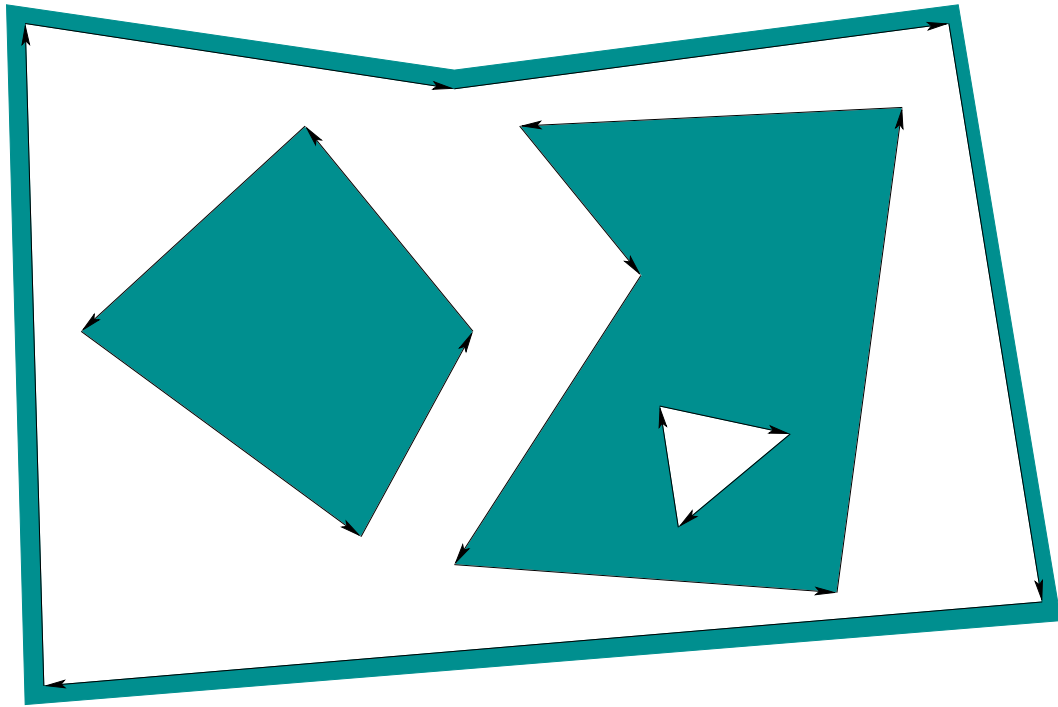


Figure 6.1: A polygonal model specified by four oriented simple polygons.

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 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 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, (4.43), 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, as in Figure 6.1. An efficient algorithm to perform this merging is given in Section 2.4 of [189]. 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 6.1, and any other information that planning algorithms would like 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 component (e.g., a 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. Each half-edge also contains pointers to the next and previous half edges in the circular chain. Such chains are oriented so that the obstacle portion (or a twin half-edge) is always to its left. Half-edges are always arranged in circular chains to form the boundary of a face. The half-edge must also store a pointer to this face.

For the example in Figure 6.1, there are four circular chains of half-edges, which 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. Note that 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.

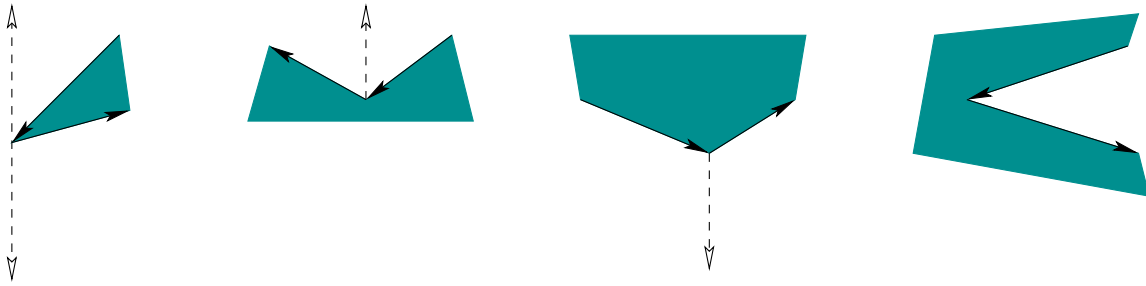


Figure 6.2: There are four general cases: 1) extending upward and downward, 2) upward only, 3) downward only, and 4) no possible extension.

6.2.2 Vertical Cell Decomposition

Cell decompositions will be defined formally (as cell complexes) in Section 6.3, but here the notion will be used informally. Combinatorial methods must construct a finite data structure that exactly represents the planning problem. Cell decomposition algorithms achieve this partitioning \mathcal{C}_{free} into a finite set of regions called *cells*. The term *k-cell* will be used to refer 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.
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 2.3 may be applied; however, in the current setting, the entire graph, G , is usually known in advance.³ This was not assumed for discrete planning problems.

Defining the vertical decomposition An algorithm will next be presented that constructs a *vertical cell decomposition* [136], which partitions \mathcal{C}_{free} into a finite collection of 2-cells and 1-cells. Each 2-cell will be either a trapezoid that has vertical sides, or it will be 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

³Once exception to this are the algorithms mentioned in Section 6.5.3, which obtain greater efficiency by only maintaining one connected component of \mathcal{C}_{obs} .

the boundary of \mathcal{C}_{obs} is hit. There are four possible cases, as shown in Figure 6.1, 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 will result. Extending these rays for the example in Figure 6.3.a leads to the decomposition of \mathcal{C}_{free} shown in Figure 6.3.b. Note that only trapezoids and triangles are obtained for the 2-cells in \mathcal{C}_{free} .

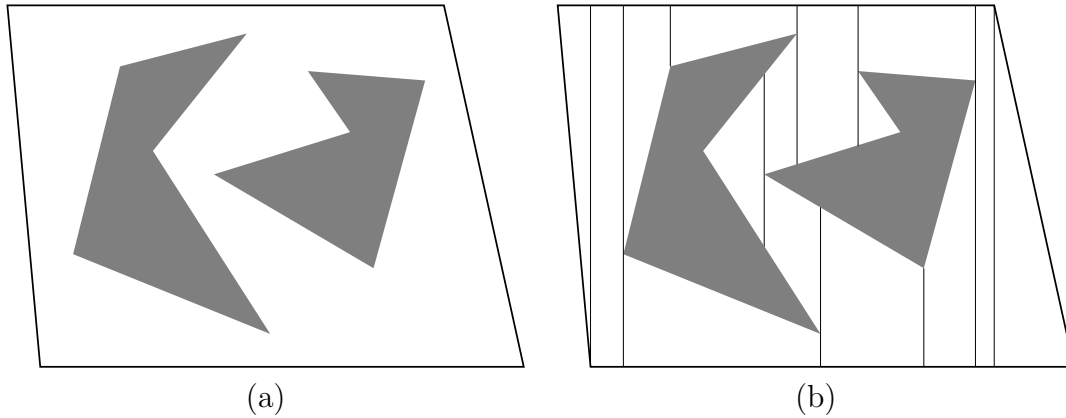


Figure 6.3: The vertical cell decomposition method uses the cells to construct a roadmap, which is searched to yield a solution to a query.

Every 1-cell is a vertical segment that serves as the border between two 2-cells. We must ensure that topology is correctly handled. 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 will not be allowed because they lie in \mathcal{C}_{obs} .

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 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 considered as part of 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 (this is requested in Exercise 1); however, in general, it is important to be aware of this difficulty, which is not as easy to fix in most other settings.

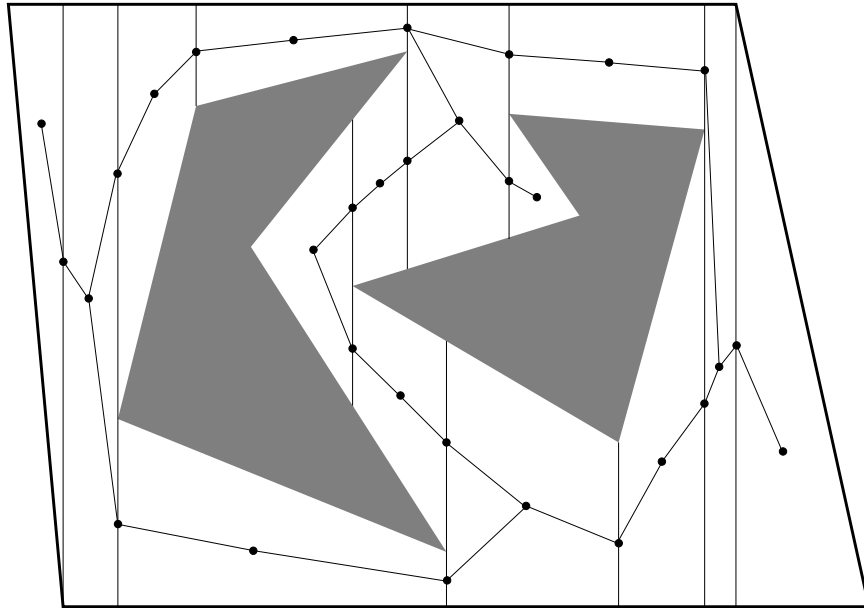


Figure 6.4: The roadmap derived from the vertical cell decomposition.

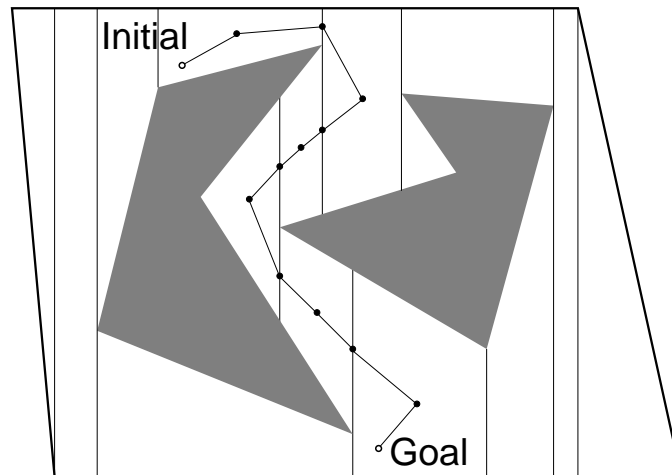


Figure 6.5: An example solution path.

Defining the roadmap To enable the handling of 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 choice is not too important. Let

$G(V, E)$ be a topological graph defined as follows. For every cell, define a vertex $v \in V$ that corresponds to its sample point. There is a vertex for every 1-cell and every 2-cell. For each 2-cell, define an $e \in E$ 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 satisfied because G is derived directly from the cell decomposition, which also preserved 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 and q_g . Let C_0 and C_f denote the cells that contain q_i and q_g , respectively. In the graph, G , search for a path that connects the sample point of C_0 to the sample point of C_f . 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, \dots, C_{k-1} denote the sequence of 1-cells and 2-cells visited along the computed path in G from C_0 to C_k .

A solution path can be formed by simply “connecting the dots”. Let $q_0, q_1, q_2, \dots, q_{k-1}, q_k$, denote the sample points along the path in 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 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 $O(n \lg n)$.

⁴This is the reason why the approach is defined differently from Chapter 1 of [437]. In that case, sample points were not placed in the interiors of the 2-cells, and collision could result for some queries.

Plane sweep principle The algorithm is based on the *plane sweep* or *line sweep* principle from computational geometry [83, 189, 219], 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, it deals with carefully “sorting” 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 in places where some critical changes occur in the information. This gives the intuition, but the sweeping line is not explicitly represented by the algorithm. To construct the vertical decomposition, we imagine that a vertical line sweeps from $x = -\infty$ to $x = \infty$, using (x, y) to denote a point in $\mathcal{W} = \mathbb{R}^2$.

From Section 6.2.1, note that the set P of \mathcal{C}_{obs} vertices is the only data in \mathbb{R}^2 that is explicitly referenced. 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 will 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 \mathcal{C}_{obs} some edges will be maintained. This list must be maintained at all times in the order that the edges appear if stabbed by the vertical sweep line. The ordering is maintained from lower to higher.

Algorithm execution Figure 6.6 and Table 6.1 show how the algorithm proceeds. Initially, L is empty and a double-connected edge list is used to represent \mathcal{C}_{free} . Each connected component of \mathcal{C}_{free} will be 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 [176], 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 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 or last cases of Figure 6.2, two edges must be inserted into L (the mirror images of these cases will cause two edges to be deleted from L). If the middle two cases occur, then one edge is replaced another in L . These insertion and deletion

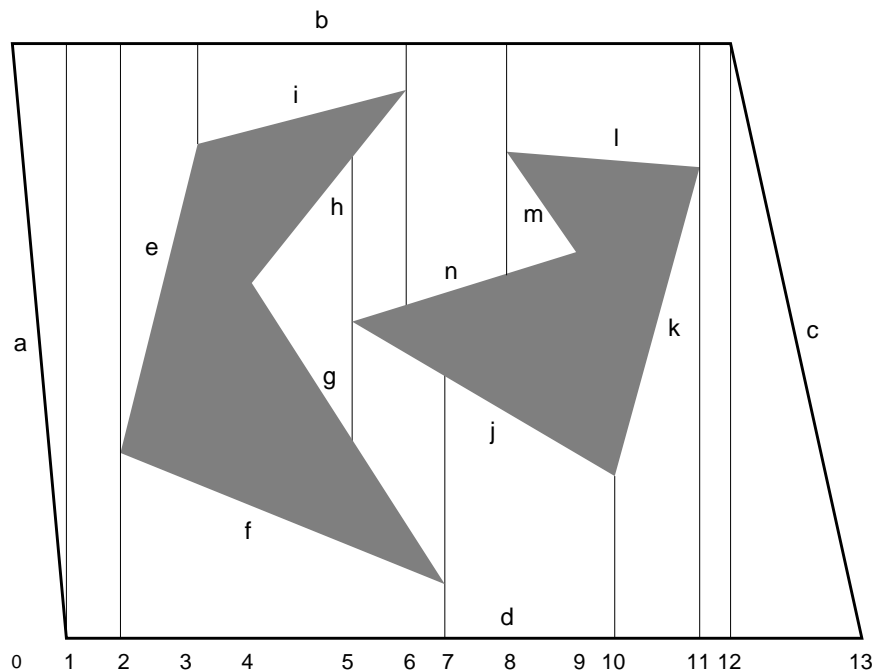


Figure 6.6: There are 14 events in this example.

| Event | Sorted Edges in L |
|-------|------------------------------|
| 0 | $\{a, b\}$ |
| 1 | $\{d, b\}$ |
| 2 | $\{d, f, e, b\}$ |
| 3 | $\{d, f, i, b\}$ |
| 4 | $\{d, f, g, h, i, b\}$ |
| 5 | $\{d, f, g, j, n, h, i, b\}$ |
| 6 | $\{d, f, g, j, n, b\}$ |
| 7 | $\{d, j, n, b\}$ |
| 8 | $\{d, j, n, m, l, b\}$ |
| 9 | $\{d, j, l, b\}$ |
| 10 | $\{d, k, l, b\}$ |
| 11 | $\{d, b\}$ |
| 12 | $\{d, c\}$ |
| 13 | $\{\}$ |

Table 6.1: The status of L is shown after each of 14 events occurs. Before the first event, L is empty.

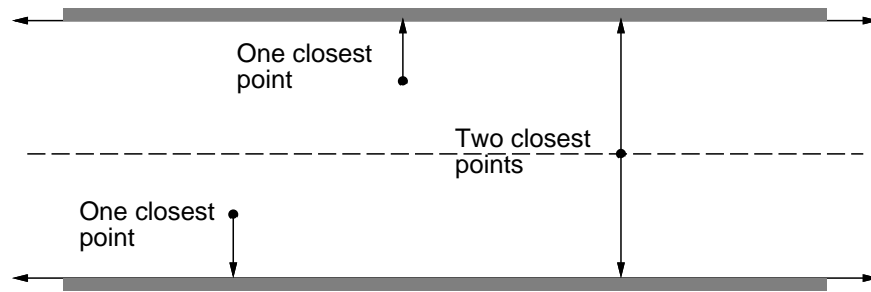


Figure 6.7: The maximum clearance roadmap keeps as far away from the \mathcal{C}_{obs} as possible. This involves traveling along points that are equidistant from two or more points on the boundary of \mathcal{C}_{obs} .

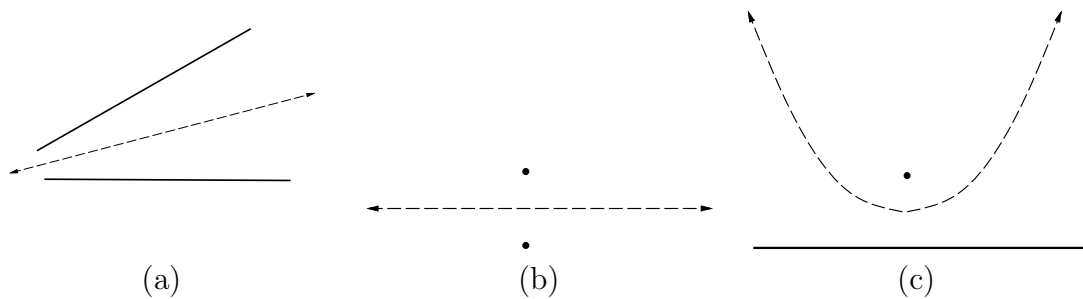


Figure 6.8: Voronoi roadmap pieces are generated in one of three possible cases: a) between two edges, b) between two points, and c) between a point and an edge. The third case leads to a quadratic curve.

operations can be performed in $O(\lg n)$ time, assuming L is implemented using a balanced binary search tree. Since there are n events, the running time for the construction algorithm is $O(n \lg n)$.

The roadmap, G , can be computed from the face pointers of the doubly-connected edge list. A more elegant approach is to incrementally build G in 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 G is correctly built, and the sample point is obtained for each cell along the way. We can even go one step further, and forget about the cell decomposition, and instead build a topological graph line segment paths between all sample points of adjacent cells.

6.2.3 Maximum Clearance Roadmaps

This method directly produces a roadmap without the consideration of cells. A *maximum clearance roadmap* tries to keep as far as possible from \mathcal{C}_{obs} , as shown for the corridor in Figure 6.7. 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* [590]. It is considered as a generalization of Voronoi diagrams, which were considered in Section 5.2.2, from the case of points to the case of polygons. Each point along a roadmap edge is equidistant from two edges of \mathcal{C}_{obs} . Each roadmap vertex corresponds to the intersection of two or more roadmap map edges, and is therefore equidistant from three or more edges 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 [328]:

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 repeatedly iteratively, the maximum clearance roadmap is the only part that will remain (assuming we prevent the remaining slivers from being shaved away).

To construct the maximum clearance roadmap, the concept of *features* from Section 5.3.3 will be 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 6.8.a. For every vertex-vertex pair, generate a line as shown in Figure 6.8.b. Finally, for every edge-point pair, generate a parabolic curve as shown in Figure ?? (The maximum clearance path between a point and a line is a parabola.) 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 time [474, 481], 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 [686].

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 [437]. The idea was first introduced in [582] and may perhaps be the first example of a motion planning algorithm. This is in direct conflict with maximum clearance because shortest paths tend to graze the corners. In fact, the problem is ill posed because \mathcal{C}_{free} is an open set. For any path

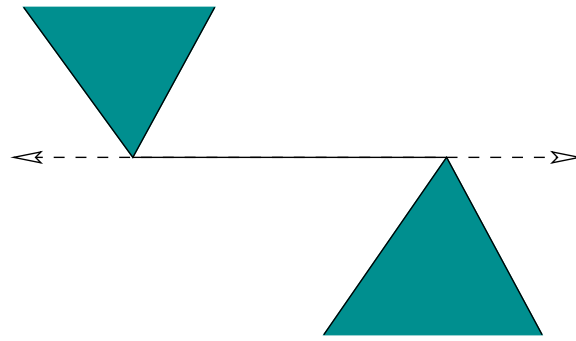


Figure 6.9: A bitangent edge must touch two reflex vertices that are mutually visible from each other, and the line must extend outward past each of them without poking into \mathcal{C}_{obs} .

$\tau : [0, 1] \rightarrow \mathcal{C}_{free}$, it is always possible to find a shorter one. For this reason, we must consider the problem of determining shortest paths in $cl(\mathcal{C}_{free})$, 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 will slightly increase the path length, but this additional cost can be made arbitrarily small as the path approaches touching \mathcal{C}_{obs} .

The *shortest path* roadmap, 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 (in general position) are reflex vertices. The vertices of G are the reflex vertices. Edges of 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 a corresponding edge is made in G .

Bitangent edges: If a *bitangent line* can be drawn through a pair of reflex vertices, then a corresponding edge is made in G . A bitangent line, depicted in Figure ??, is a line that is incident to two or more reflex vertices and does not poke into the interior of \mathcal{C}_{obs} at any of these vertices. Furthermore, all of these vertices must be mutually visible from each other.

An example of the resulting roadmap is shown in Figure 6.10. 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 , both configurations are connected to all roadmap vertices that are visible; this is shown in Figure 6.11. 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 will be the shortest path between q_i and q_g . The shortest path for the example Figure 6.11 is shown in Figure 6.12.

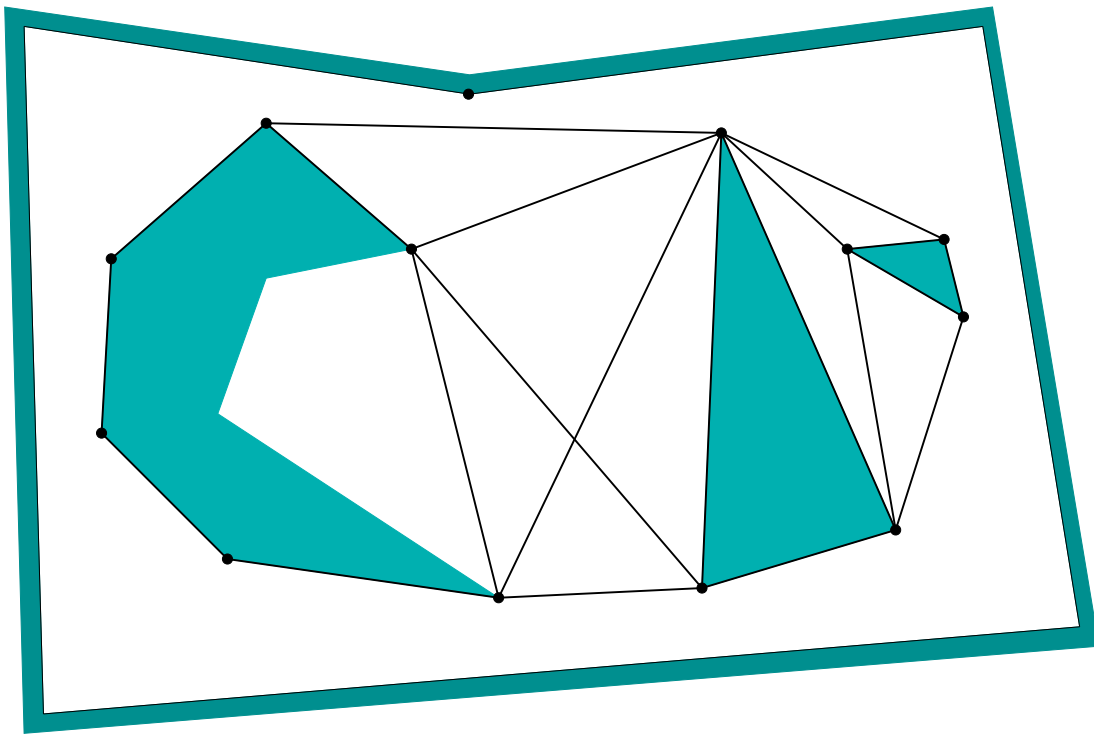


Figure 6.10: The shortest path roadmap includes edges between consecutive reflex vertices on \mathcal{C}_{obs} and also bitangent edges.

If the bitangent tests are performed naively, then the resulting algorithm will require $O(n^3)$ 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, 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 [189] for more details. There exists an algorithm that can compute the shortest path roadmap in time $O(n^2 + m)$, in which m is the total number of edges in the roadmap [595].

The shortest path roadmap can be implemented without the use of trigonometric functions. This greatly improves the numerical robustness of the algorithm. 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 $f_l(p_1, p_2, p_3) = \text{TRUE}$ 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 $f_l(p_1, p_2, p_3) = \text{TRUE}$, 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 6.13. A pair, p_2, p_5 , of vertices

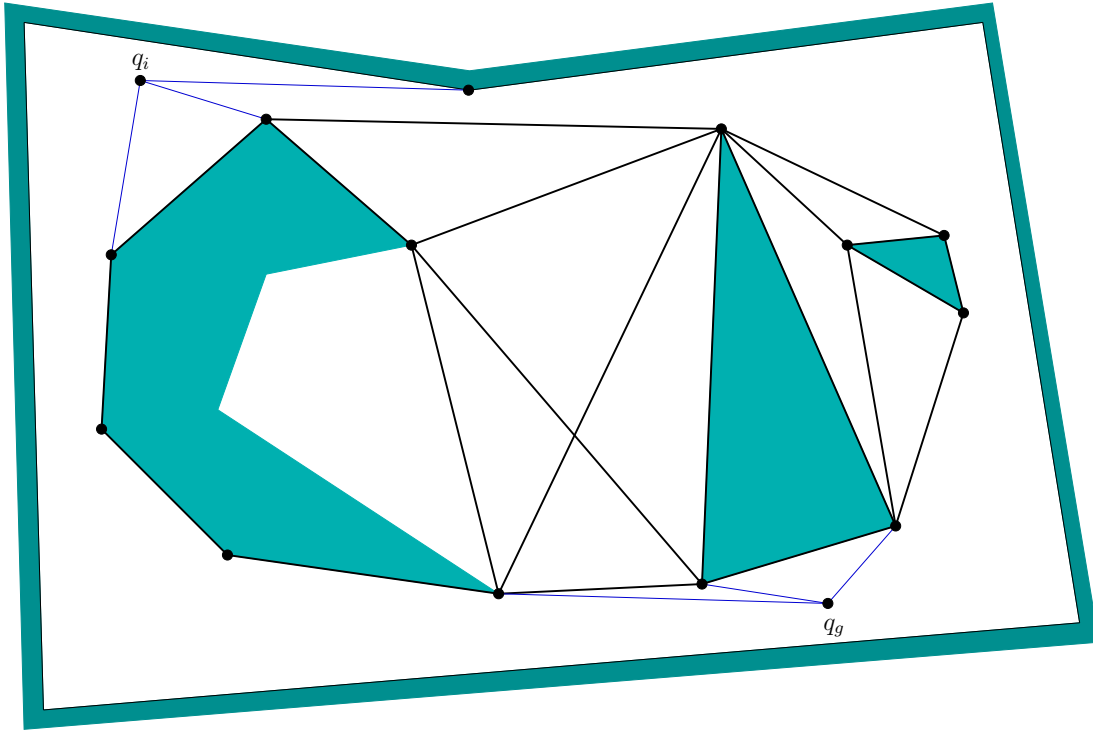


Figure 6.11: To solve a query, q_i and q_g are connected to all visible roadmap vertices, and graph search is performed.

should receive a bitangent edge if the following sentence is FALSE :

$$[f_l(p_1, p_2, p_3) \oplus f_l(p_1, p_2, p_5)] \vee [f_l(p_2, p_5, p_6) \oplus f_l(p_4, p_5, p_6)], \quad (6.1)$$

in which \oplus denotes logical “exclusive or”. The f_l predicate can be implemented without trigonometric functions by defining

$$M(p_1, p_2, p_3) = \begin{pmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{pmatrix}, \quad (6.2)$$

in which $p_i = (x_i, y_i)$. If $\det(M) > 0$, then $l_f(p_1, p_2, p_3) = \text{TRUE}$; otherwise, $l_f(p_1, p_2, p_3) = \text{FALSE}$.

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, which

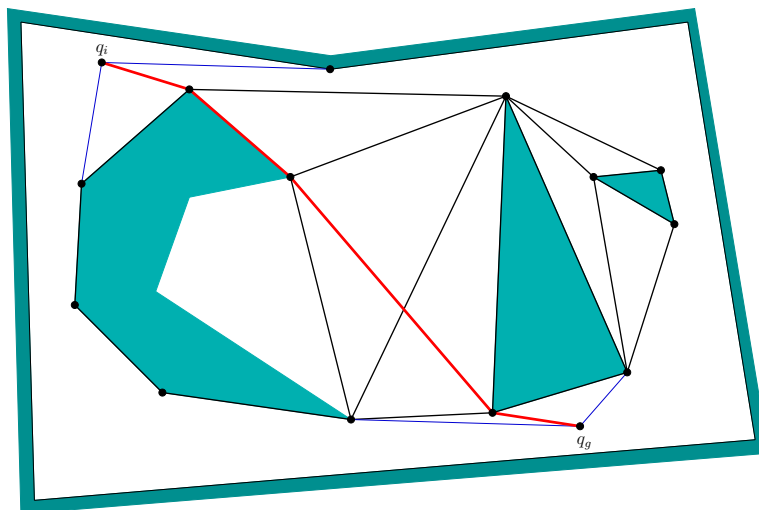


Figure 6.12: The shortest path in the extended roadmap is the shortest path between q_i and q_g .

provides 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 models. Along the way, a cylindrical decomposition will appear in Section 6.3.4 for the special case of a line-segment robot in $W = \mathbb{R}^2$.

6.3.1 General Definitions

In this section, the term *complex* will be used to refer 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* will still refer 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 will be more useful to build decompositions out of simpler cells, such as ones that contain no holes. Formally, we will require 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 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

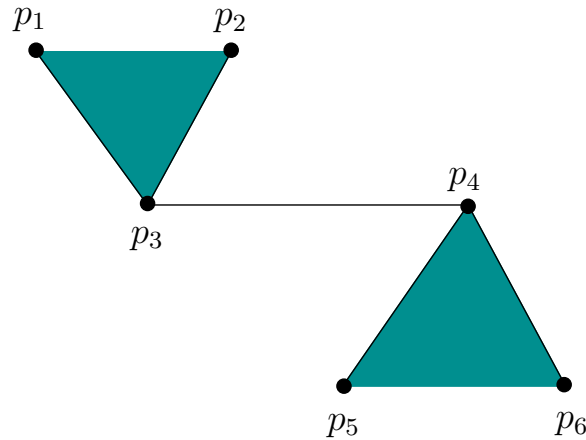


Figure 6.13: The bitangents can be determined by checking for left turns, which avoids the use of trigonometric functions and their associated numerical problems.

this one is more complicated to define, it encompasses all of the cell complexes that are of interest in this book. It provides an elegant way to represent topological spaces. Another important cell complex which is not covered here is the *CW-complex* [317].

Simplicial Complex For this definition, it is assumed that $X = \mathbb{R}^n$. Let p_1, p_2, \dots, p_{k+1} , be $k \leq n + 1$ linearly-independent⁵ points in \mathbb{R}^n . A k -simplex, $[p_1, \dots, p_{k+1}]$ is formed from these points as

$$[p_1, \dots, p_{k+1}] = \left\{ \sum_{i=1}^{k+1} \alpha_i p_i \in \mathbb{R}^n \mid 0 \leq \alpha_i \leq 1 \text{ for any } 1 \leq i \leq k + 1 \right\}, \quad (6.3)$$

in which $\alpha_i p_i$ is 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, set one of the α_i to 0 for any i for which $1 \leq i \leq k + 1$. This yields a $(k - 1)$ -dimensional simplex which 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*.

To form a complex, the simplexes will be required to fit together in a nice way. This yields a high-dimensional notion of a *triangulation*, which in \mathbb{R}^2 is a tiling of triangular regions. A *simplicial complex*, \mathcal{K} , is a finite set of simplexes that satisfies the following:

⁵Form k vectors by subtracting p_1 from the other k points. 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 3 points cannot be coplanar.

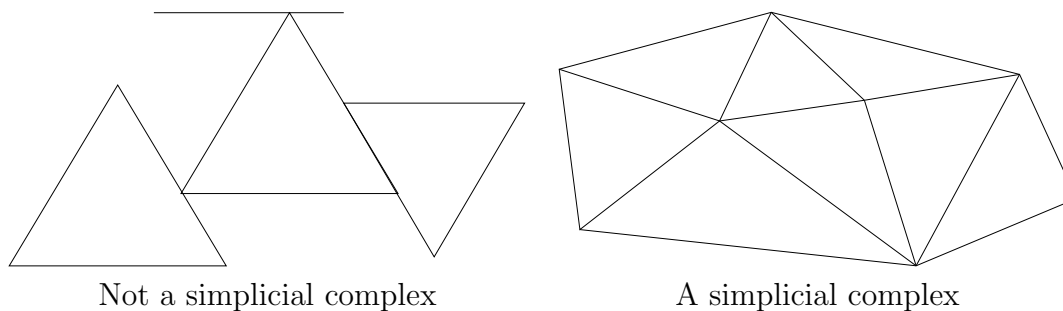


Figure 6.14: To become a simplicial complex, the simplex faces must fit together nicely.

1. Any face of a simplex in \mathcal{K} is also in \mathcal{K} .
2. The intersection, $\Delta_1 \cap \Delta_2$, of any two simplexes $\Delta_1, \Delta_2 \in \mathcal{K}$ is either empty, or $\Delta_1 \cap \Delta_2$ is a common face of both Δ_1 and Δ_2 .

Figure 6.14 illustrates these requirements. For $k > 0$, a k -cell of \mathcal{K} is defined to be interior $\text{int}([p_1, \dots, p_{k+1}])$ of any k -simplex. For $k = 0$, every 0-simplex can also be considered as 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 and decomposition algorithms that arise in motion planning, they will be insufficient for the most general problems. A *singular complex* is a generalization of the *simplicial complex*. Instead of being limited to \mathbb{R}^n , let a singular complex be defined for any (Hausdorff) topological space, X . 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 (this was introduced in Example 4.1.6, and has been used extensively). 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 $G(V, E)$ 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 can be 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

$$\{x \in X \mid \tau(s) = x \text{ for some } s \in (0, 1)\}. \quad (6.4)$$

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 a 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 $B^k \subset \mathbb{R}^n$ for $k \leq n$ denote a closed, d -dimensional unit ball,

$$B^k = \{x \in \mathbb{R}^n \mid \|x\| \leq 1\}, \quad (6.5)$$

in which $\|\cdot\|$ is the Euclidean norm. A *singular k -simplex* is a continuous mapping $\sigma : B^k \rightarrow X$. Let $\text{int}(B^k)$ refer to the interior of B^k . For $k \geq 1$, the *k -cell*, C , corresponding to a singular k -simplex, σ , is the image $C = \sigma(\text{int}(B^d)) \subseteq X$. 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 . When σ is restricted to $\text{int}(B^d)$, it actually defines a homeomorphism between B^d and C . Note that both of these are open sets if $d > 0$.

A simplicial complex required 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 d -cell is an open set in the topological subspace S_d . Note that 0-cells are open in S_0 , even though they are usually closed in X .

Example 6.3.1 (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 \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 does satisfy 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 $cl(\mathcal{C}_{free})$, 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 $S_3 = cl(\mathcal{C}_{free})$, 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 d -dimensional cells, however, is open in the topological space S_d . 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 less paths would be needed in the search graph. 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 [499, 390]. Therefore, we are willing to tolerate non-optimal decompositions.

Triangulation One alternative to vertical decomposition is to perform a *triangulation*, which yields a simplicial complex over \mathcal{C}_{free} . Figure 6.15 shows an example. Because \mathcal{C}_{free} is an open set, there are no 0-cells. Each 2-simplex (triangle) has either three, two, or one face, 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 6.16. Note that there are many ways to triangulate \mathcal{C}_{free} for a given problem. The problem of finding good triangulations, which for example means trying to avoid thin triangles, is given considerable attention in computational geometry [83, 189, 219].

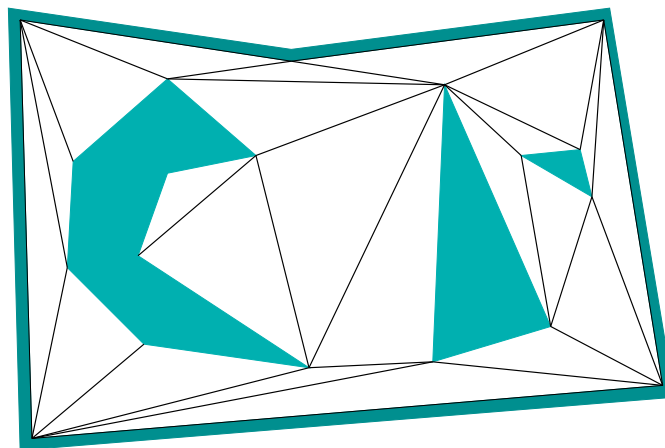


Figure 6.15: A triangulation of \mathcal{C}_{obs} .

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

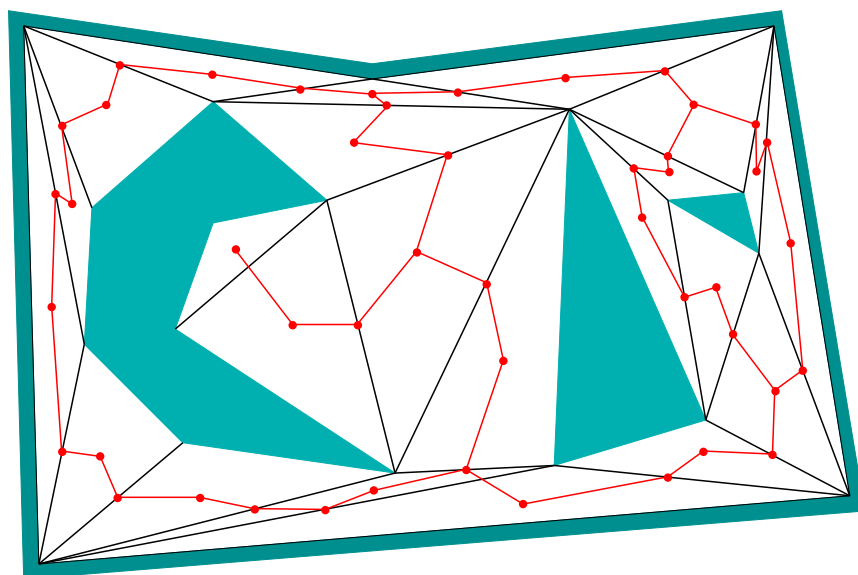


Figure 6.16: A roadmap obtained from the triangulation.

two triangles. Even though this leads to triangular cells, it does not produce a simplicial complex (two triangles could abut the same edge of a triangle). 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 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 [189] 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 [137]. Unfortunately, this algorithm is too complicated to use in practice (there are, however, simpler algorithms whose complexity is close to $O(n)$; see the end of Chapter 3 of [189] for a survey).

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 of the way from $y = -\infty$ to $y = \infty$. The result is shown in Figure 6.17, 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, configuration space topology, and semi-algebraic models. Therefore, it is presented here to ease the transition to the general decompositions.

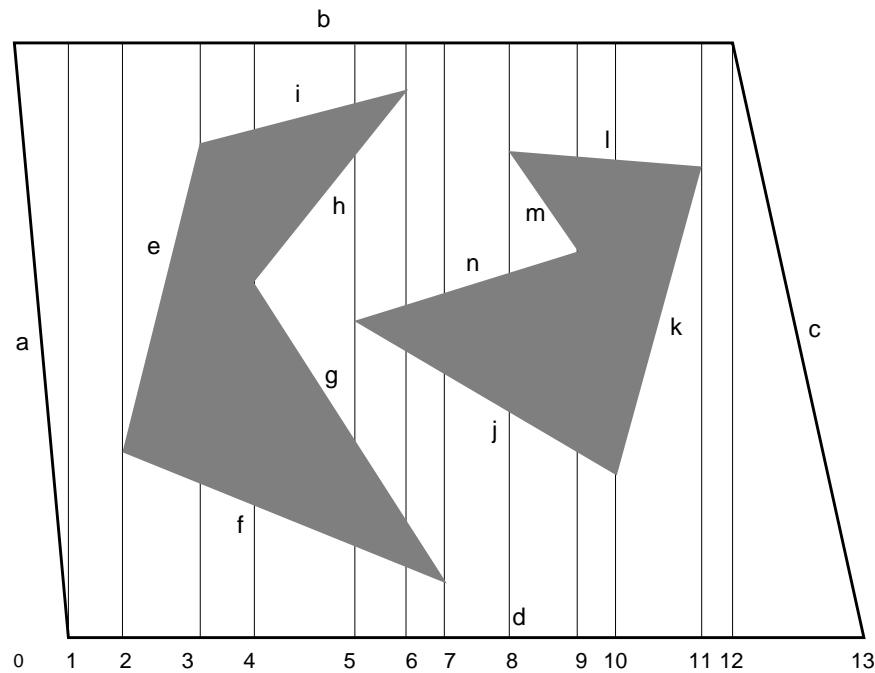


Figure 6.17: The cylindrical decomposition differs from the vertical decomposition in that the rays continue forever instead of stopping at the nearest edge. Compare this figure to Figure 6.6.

The most important property of the cylindrical decomposition is shown in Figure 6.18. Consider each vertical strip between two events. When traversing a strip from $y = -\infty$ to ∞ , 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 (3.5) 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 very valuable 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 n by recursively applying the sweeping idea. The method requires, however, that \mathcal{C}_{obs} must be 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

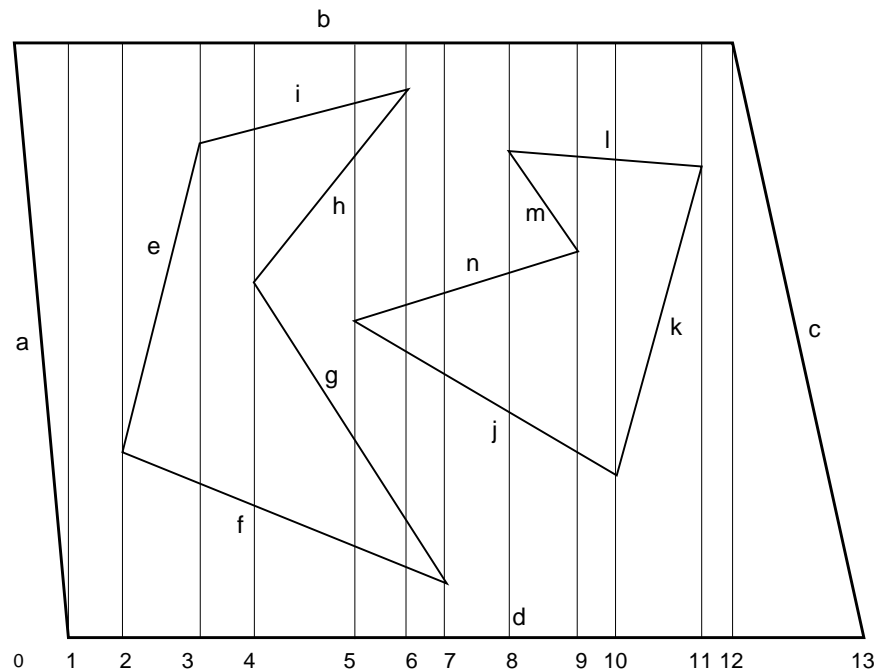


Figure 6.18: The cylindrical decomposition produces vertical strips. Inside of a strip, there is a stack of collision-free cells, separated by \mathcal{C}_{obs} .

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 interesting planning problem in which \mathcal{C}_{obs} is piecewise linear is for a polyhedral robot that can translate in \mathbb{R}^3 , and the obstacles in \mathcal{W} are polyhedra. Because 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 6.19 illustrates the algorithm that constructs the 3D vertical decomposition. Compare this with the algorithm in Section 6.2.2. Let (x, y, z) denote points 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 6 planes. The cross section of a 3-cell, for some fixed x value will yield 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. It is bounded above and below by polygonal 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 two-dimensional, polygonal slice of \mathcal{C}_{obs} . Three such slices are shown at the bottom of Figure 6.19. Each slice is parallel to the YZ plane, and appears to look exactly like a problem that can be solved by the 2D

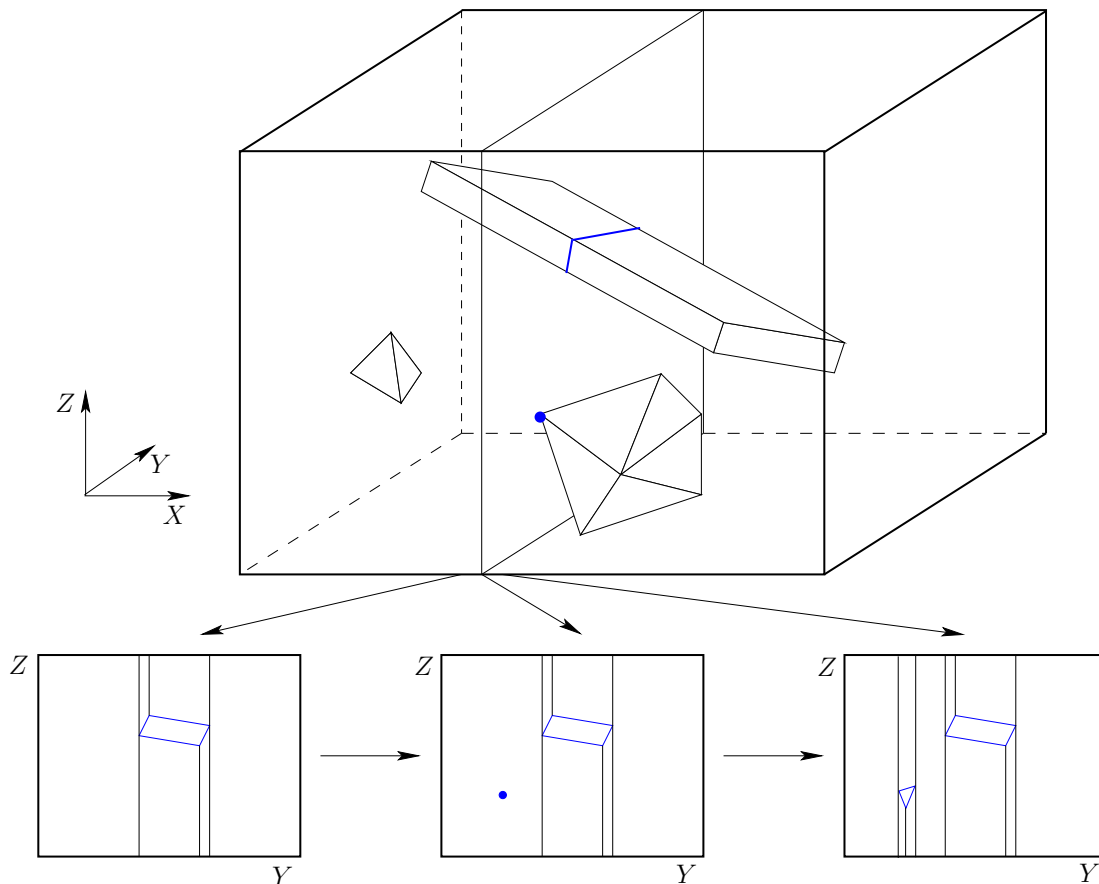


Figure 6.19: In higher dimensions, the sweeping idea can be applied recursively.

vertical decomposition method. 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 change in an important way is when the sweeping plane stops at some x value. The center slice in Figure 6.19 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., it 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 the critical changes. During this process, the three dimensional cell decomposition and roadmap can be incrementally constructed, just as in the 2D case.

The roadmap is constructed by placing a sample point in the center of each 3-cell and each 2-cell. The vertices are the sample points, and edges are added to the roadmap by connecting the sample points of adjacent pairs 3-cells and 2-cells.

This same principle can be extended to any dimension, but the applications to motion planning are limited because the method requires linear models (or at

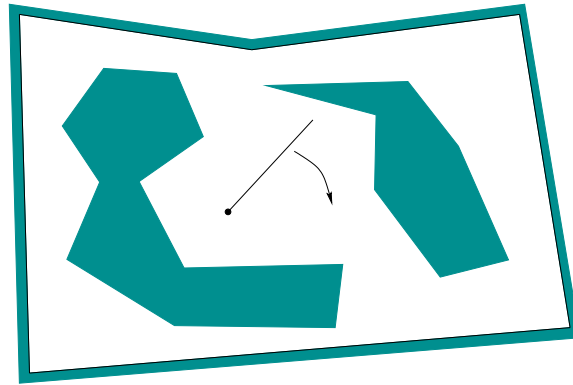


Figure 6.20: Motion planning for a line segment that can translate and rotate in a 2D world.

least is very challenging to adapt to nonlinear models; in some special cases, this can be done).

6.3.4 A Decomposition for a Line-Segment Robot

This section presents a 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. Suppose the planning problem is as shown in Figure 6.20. 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 configuration 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) .

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 may inspire 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 $\Delta\theta = 2\pi/K$ [11]. The obstacle region, \mathcal{C}_{obs} , will be 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 connecting samples between corresponding cells in neighboring slices. If K is large enough, this strategy could work quite well, but the method is not complete because a sufficient value for K cannot be determined in advance. The method is actually an interesting hybrid between being combinatorial and sampling-based

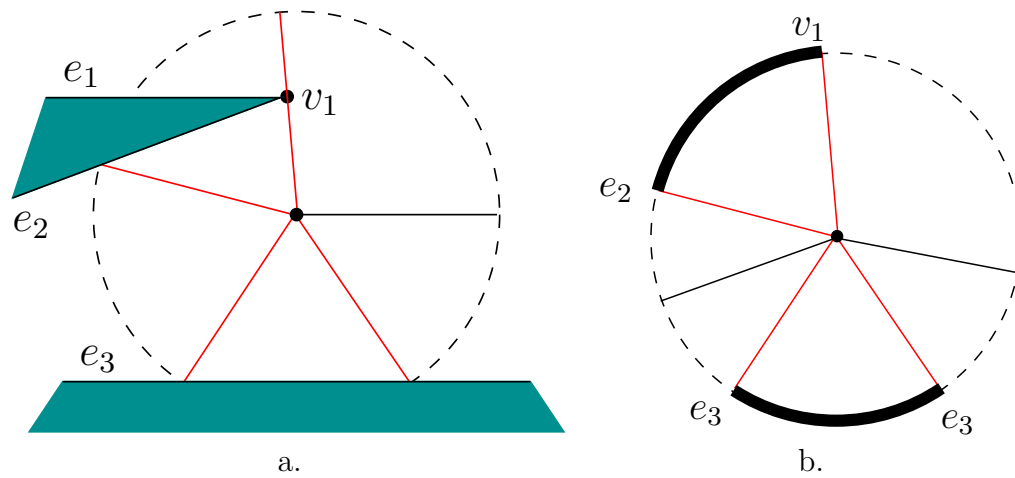


Figure 6.21: Fix (x, y) , and swing the segment around for all values of $\theta \in [0, 2\pi]/\sim$. a) Note the vertex and edge features that are hit by the segment. b) Record orientation intervals over which the robot is not in collision.

motion planning. A resolution complete version can be imagined.

In the limiting case as K tends to infinity, the surfaces of \mathcal{C}_{obs} will be 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 this 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 to 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 will then be lifted into \mathcal{C} by considering θ as a third axis that is “above” the XY plane. The result will be a cylindrical decomposition in which each cell in the XY plane produces a cylindrical stack of cells for different θ values. Recall the cylinders in Figures and 6.17 and 6.18. 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 6.21. In Figure 6.21.a, 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

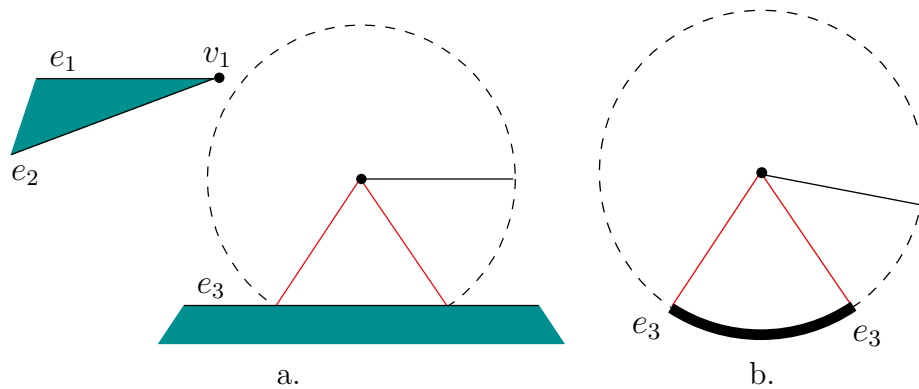


Figure 6.22: If x is increased enough, a critical change occurs in the radar map because v_1 can no longer be reached by the robot.

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 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.

Figure 6.21.b 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 $[e_2, e_3]$ can be used to identify 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 changing the cell.

Of course, if (x, y) is changed enough, then at some point 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 6.21.a. The black band between v_1 and e_2 in Figure 6.21.b will shrink in length. Eventually, when the distance from (x', y') to v_1 is greater than the length of \mathcal{A} , the black band will disappear. This situation is shown in Figure 6.22. 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, will produce 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 6.22 into the positive Y direction. The remaining black band will begin to shrink, and will finally disappear 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

$$([f_1, f_2], [f_3, f_4], [f_5, f_6], \dots, [f_{2k-1}, f_{2k}]), \quad (6.6)$$

when there are k orientation intervals over which the configurations lie in \mathcal{C}_{free} . For the radar map in Figure 6.21.b, 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 6.22.b is $([e_3, e_3])$. The intervals are specified in counterclockwise order around the radar map. Because 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 can 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 just write \emptyset .

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 6.21.b. We say that a *critical change* occurs each time the circular ordering representation of (6.6) is forced to change. 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 , will represent 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 three dimensional region in \mathcal{C} 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 will precisely define the boundaries of our desired 2-cells in \mathbb{R}^2 . Let L denote the length of the line segment, \mathcal{A} .

Two of the five cases have already been observed in Figures 6.21 and 6.22. These appear in Figures 6.23.a and Figures 6.23.b, and occur if (x, y) is within L of an edge or a vertex. The third and fourth cases are shown in Figures 6.23.c and 6.23.d, 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 resurfacing of the bitangent case from Figure 6.9, which arose for the shortest path graph. If the vertices are within

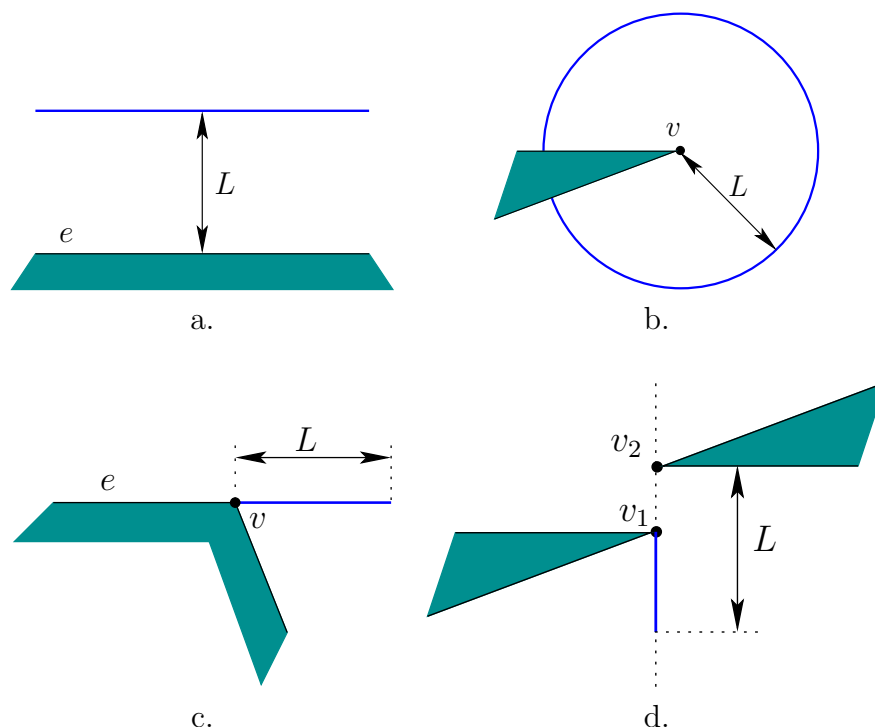


Figure 6.23: Four of the five cases that produce critical curves in \mathbb{R}^2 .

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 6.24, is the most complicated. It is a 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 the $(0, 0)$ origin, then the equation of the curve is

$$(x^2 - y^2)(y + d)^2 - y^2 L^2 = 0, \quad (6.7)$$

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), which form the 2-cells. The boundaries between adjacent 2-cells are sections of the critical curves, and form 1-cells. There are also 0-cells at places where critical curves intersect. Figure 6.25 shows an example adapted from [437]. 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 will be 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, it seems best to avoid wasting time on decomposing it. These unnecessary cases can be detected by imagining that \mathcal{A} is a laser with

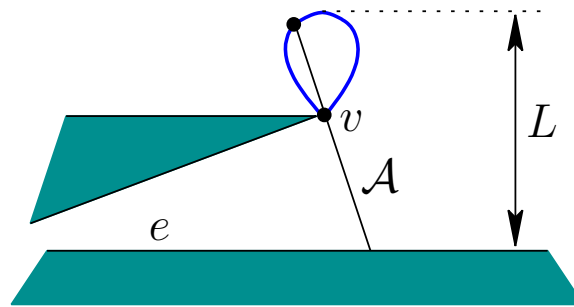


Figure 6.24: The fifth case is the most complicated. It results in a fourth degree algebraic curve called the Conchoid of Nicomedes.

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 will not be generated here to make the coming example easier to understand. Each 3-cell, $\{R, [f_i, f_{i+1}]\}$, will correspond to the vertex in a roadmap. The roadmap edges will 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 1D portion of a critical curve.

The problem is to determine which 3-cells are actually adjacent. Figure 6.26 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 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, then 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 6.26 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 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 6.27 depicts the cells for the example in Figure 6.25. 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

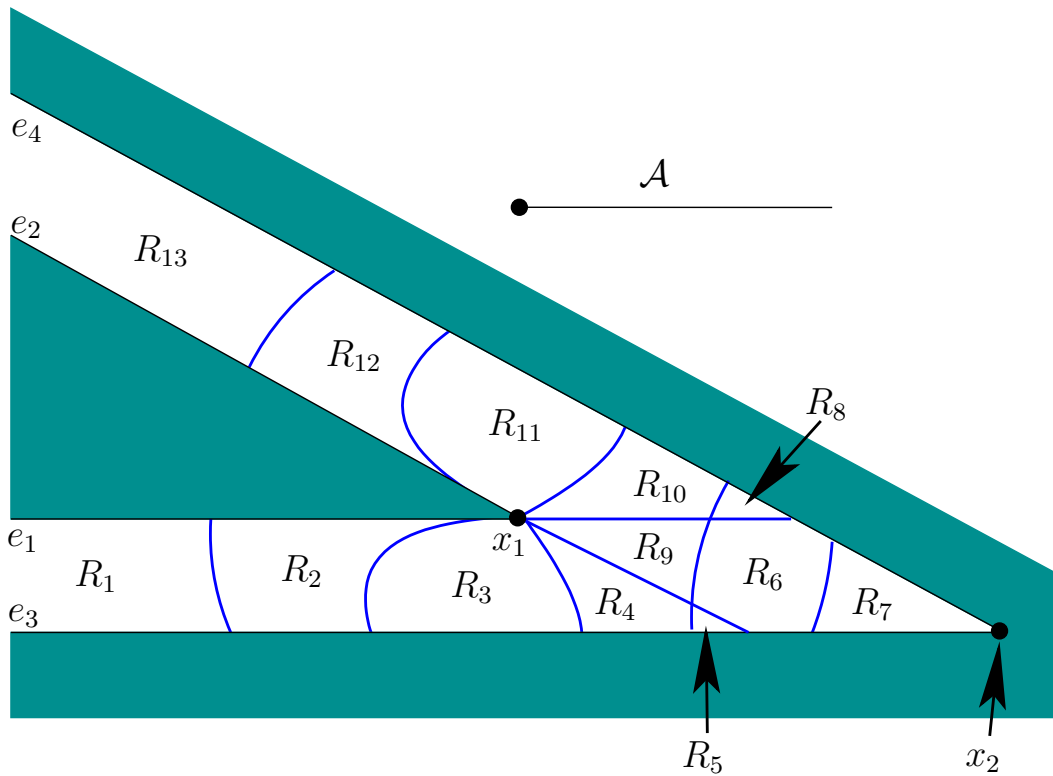


Figure 6.25: The critical curves form the boundaries of the noncritical regions in \mathbb{R}^2 .

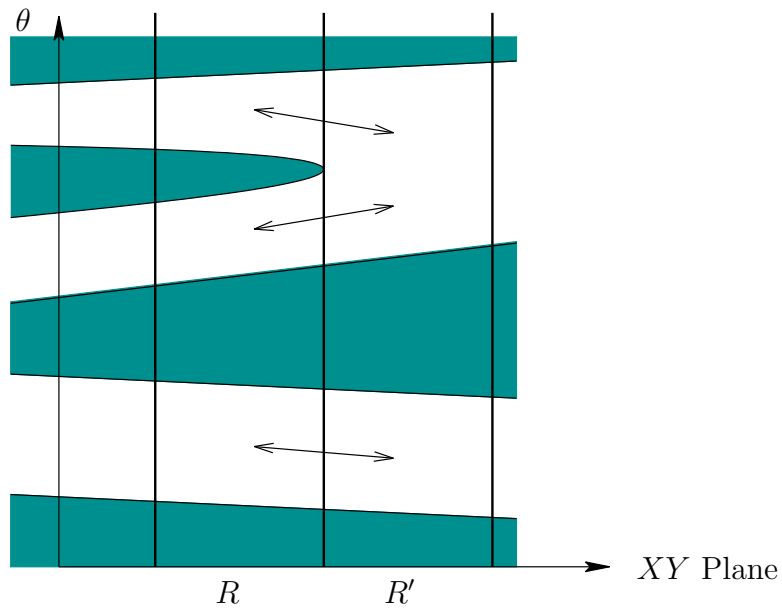


Figure 6.26: Connections are made between neighboring 3-cells that lie above neighboring noncritical regions.

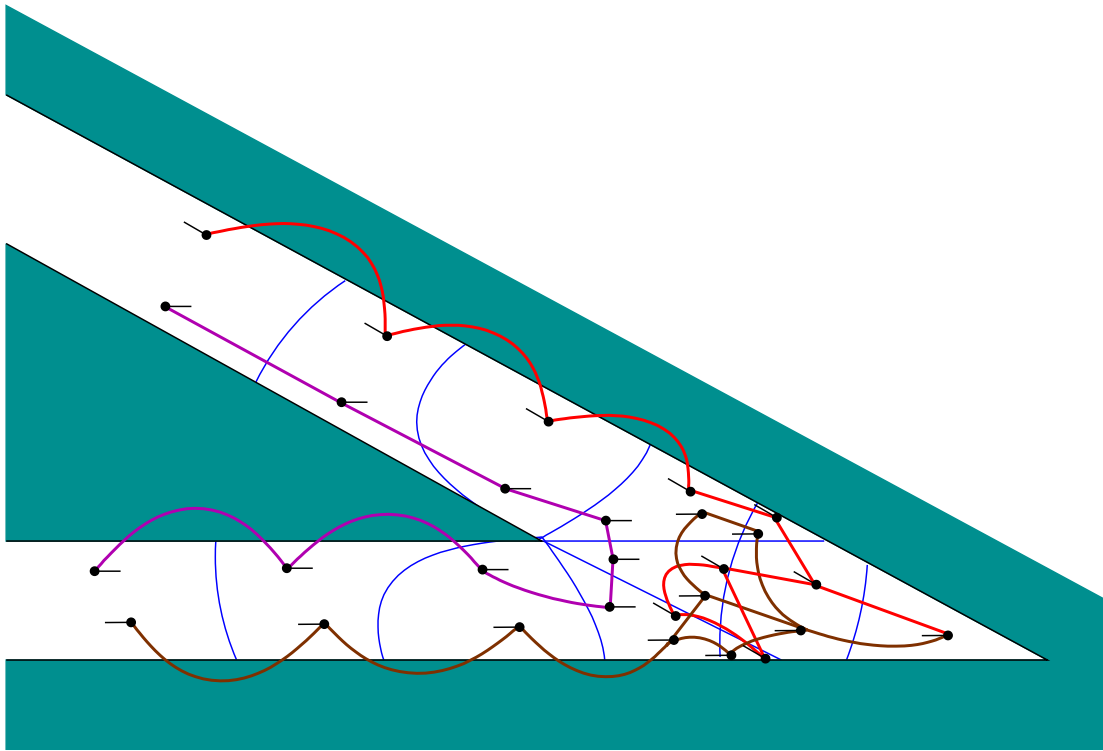


Figure 6.27: A depiction of the 3-cells above the noncritical regions.

cell. The connections between the cells are also shown. Using the noncritical region and feature names from Figure 6.25, the resulting roadmap is depicted abstractly in Figure 6.28. 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 to 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. Because of 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 [34]. Some of these details are also summarized in [437].

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 correspond 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

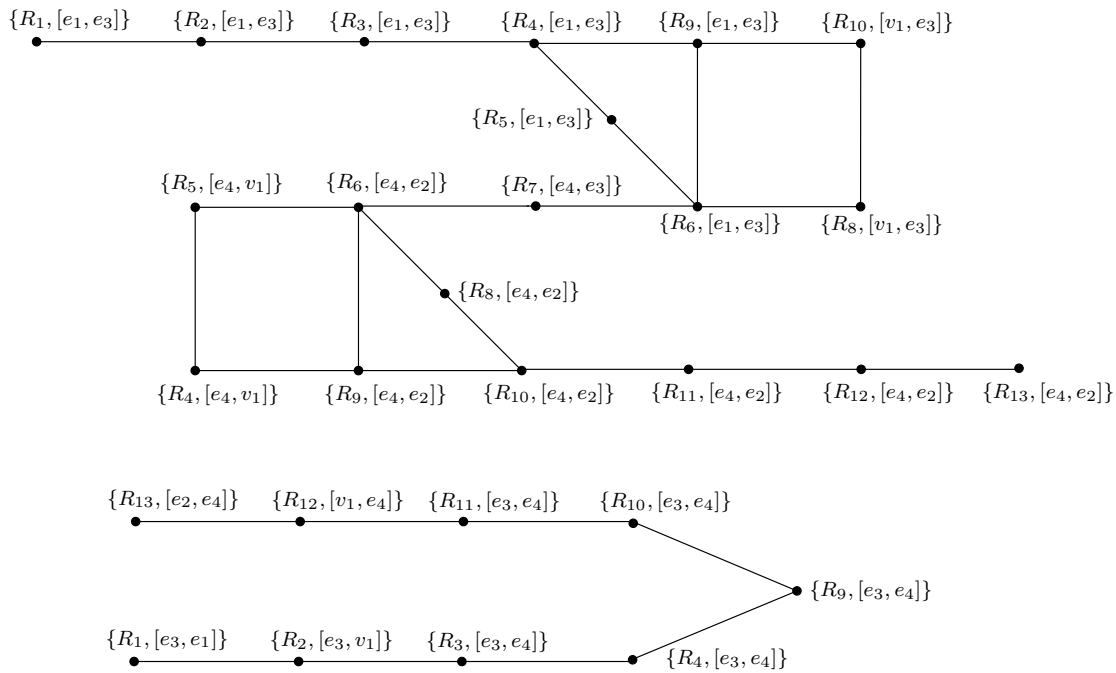


Figure 6.28: The roadmap corresponding to the example in Figure 6.25.

occur, and the number of cells is expected to be reasonable. In [675], an $O(n^5)$ -time algorithm is given to construct the cell decomposition. Other algorithms, which have much better running time are mentioned in Section 6.5.3, but they are much 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 4.3.1 and even the kinematic closure 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 here were mostly developed in the context of computational real algebraic geometry [58, 178]. They are powerful enough to conquer numerous problems in robotics, computer vision, geometric modeling, computer-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 [676].

6.4.1 Basic Definitions and Concepts

This section builds on the semi-algebraic 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 $\mathcal{C} \cong \mathbb{R}^3 \times \mathbb{RP}^3$, which is a six-dimensional manifold, but it can be embedded in \mathbb{R}^{12} , which is obtained from the Cartesian product of all 3×3 matrices and \mathbb{R}^3 . The required constraints that for rotation matrices to lie $SO(2)$ or $SO(3)$ are polynomials, and 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 will be discussed briefly in Section 6.4.2.

Suppose that the models in \mathbb{R}^n are all expressed using polynomials from $\mathbb{Q}[x_1, \dots, x_n]$, the set of polynomials⁶ over the field of rational numbers \mathbb{Q} . Let $f \in \mathbb{Q}[x_1, \dots, x_n]$ denote a polynomial.

Tarski sentences Recall the logical predicates that were formed in Section 3.1. They will be used again here, but here they are defined with a little more flexibility. For any $f \in \mathbb{Q}[x_1, \dots, x_n]$, 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* $\phi(x_1, \dots, x_n)$ 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 which yields TRUE if and only if the relation is satisfied when the polynomial is evaluated at the point $(x_1, \dots, x_n) \in \mathbb{R}^n$.

Example 6.4.1 An example of a predicate ϕ over \mathbb{R}^3 is

$$\phi(x_1, x_2, x_3) = (x_1^2 x_3 - x_2^4 < 0) \vee [-(3x^2 x^3 \neq 0) \wedge (2x_3^2 - x_1 x_2 x_3 + 2 \geq 0)]. \quad (6.8)$$

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

$$\Phi(x_1, \dots, x_{n-k}) = (\mathcal{Q}z_1)(\mathcal{Q}z_2) \dots (\mathcal{Q}z_k) [\phi(z_1, \dots, z_k, x_1, \dots, x_{n-k})], \quad (6.9)$$

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

⁶It will be explained shortly why $\mathbb{Q}[x_1, \dots, x_n]$ is preferred over $\mathbb{R}[x_1, \dots, x_n]$.

at the left to be a valid Tarski sentence; however, such expressions can always 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 [559]: 1) eliminate any redundant quantifiers; 2) rename some of the variables to ensure that the same variable does not end up appearing both free and bound; 3) move negation symbols as far inward as possible; 4) push the quantifiers to the left.

Example 6.4.2 (Tarski sentences) Several examples are given. Tarski sentences that have no free variables are either TRUE or FALSE in general because there are no arguments on which the results depend. Here is an example,

$$\Phi = \forall x \exists y (x^2 - y < 0), \quad (6.10)$$

which 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 another 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:

$$\Phi(z) = \exists y \forall x (x^2 - zx^2 - y < 0). \quad (6.12)$$

This yields a function $\Phi : \mathbb{R} \rightarrow \{\text{TRUE}, \text{FALSE}\}$, in which

$$\Phi(z) = \begin{cases} \text{TRUE} & \text{if } z > 1 \\ \text{FALSE} & \text{if } z \leq 1 \end{cases}. \quad (6.13)$$

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 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 examples 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 the familiar issue faced throughout motion planning. 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, \dots, 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, \dots, 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

$$\Phi(x_1, \dots, x_n) = \phi(x_1, \dots, x_n) \quad (6.14)$$

for all $(x_1, \dots, x_n) \in \mathbb{R}^n$. This is equivalent to constructing a semi-algebraic model because ϕ can always be expressed in the form

$$\phi(x_1, \dots, x_n) = \bigvee_{i=1}^k \bigwedge_{j=1}^{m_i} (f_{i,j}(x_1, \dots, x_n) \bowtie 0), \quad (6.15)$$

in which \bowtie may be either $<$, $=$, $>$. This appears the same (3.5), except that (6.15) uses relations $<$, $=$, and $>$ to allow open and closed semi-algebraic sets, whereas (3.5) 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

$$Y = \bigcup_{i=1}^k \bigcap_{j=1}^{m_i} \{(x_1, \dots, x_n) \in \mathbb{R}^n \mid f_{i,j}(x_1, \dots, x_n) = s_{i,j}\}, \quad (6.16)$$

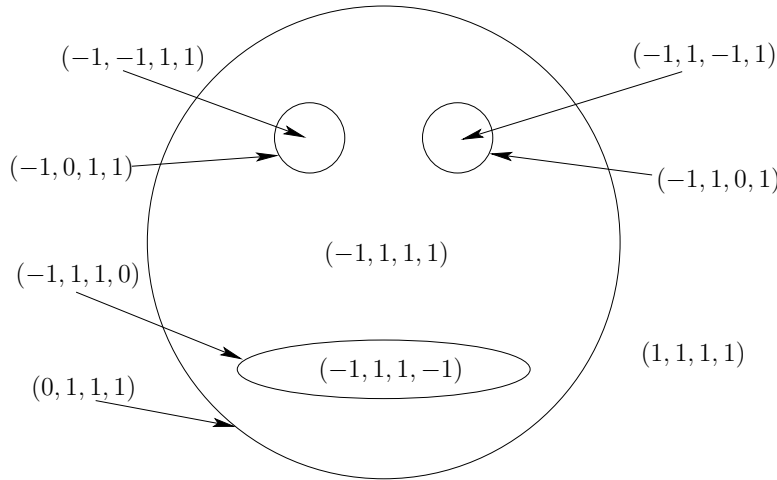


Figure 6.29: A semi-algebraic decomposition of the gingerbread face yields 9 sign-invariant regions.

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 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 .

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, $\text{sgn}_{\mathcal{F}} : \mathbb{R}^n \rightarrow \{-1, 0, 1\}^m$. Each $f \in \mathcal{F}$, has a corresponding position in the sign assignment vector. At this position, the sign, $\text{sgn}(f(x_1, \dots, x_n)) \in \{-1, 0, 1\}$, 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 $\text{sgn}_{\mathcal{F}}$ is must remain constant. The regions will not be referred to as 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 6.4.3 (Sign assignment) Recall Example 3.1.1 and Figure 3.4 from Section 3.1.2. Figure 3.4.a 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

$$\text{sgn}_{\mathcal{F}}(x, y) = \begin{cases} -1 & \text{if } x^2 + y^2 - 4 < 0 \\ 0 & \text{if } x^2 + y^2 - 4 = 0 \\ 1 & \text{if } x^2 + y^2 - 4 > 0 \end{cases} . \tag{6.17}$$

Now consider the sign assignment, $\text{sgn}_{\mathcal{F}}$, shown in Figure 6.29 for the gingerbread face of Figure 3.4.b. The polynomials of the semi-algebraic model are $\mathcal{F} = \{f_1, f_2, f_3, f_4\}$, as defined in Example 3.1.1. In order, these are the “head”, “left eye”, “right eye”, and “mouth”. The sign assignment produces a four-dimensional vector of signs. Note that if (x, y) lies on one of the zeros of a polynomial in \mathcal{F} , a 0 appears in the sign assignment. If the curves of two or more

of the polynomials had intersected, then the sign assignment would produce more than one 0 at the intersection points.

For the semi-algebraic decomposition for the gingerbread face in Figure 6.29, there are nine regions. Five two-dimensional regions correspond to 1) being outside of the face, 2) inside of the left eye, 3) inside of the right eye, 4) inside of the mouth, and 5) inside of the face, but outside of the mouth and eyes. There are four one-dimensional regions, each of which corresponds to points that lie on one of the zero sets of a polynomial. The resulting decomposition is not a singular complex because the $(-1, 1, 1, 1)$ region contains three holes. ■

A decomposition such as the one in Figure 6.29 would not be very useful for motion planning because of the holes in the regions. Further refinement will be needed for motion planning, which is fortunately produced by cylindrical algebraic decomposition. On the other hand, any semi-algebraic decomposition is quite useful for solving the decision problem. Only one point needs to be checked inside of each region to determine whether some Tarski sentence that has no free variables is true. Why? Observe that if the polynomial signs cannot change over some region, then the TRUE /FALSE value of the corresponding logical predicate, Φ cannot change. Therefore, it sufficient only to check one point per sign-invariant region.

6.4.2 Cylindrical Algebraic Decomposition

Cylindrical algebraic decomposition is a general method that produces a cylindrical decomposition in the same sense considered in Section 6.3.2 for polygons in \mathbb{R}^2 , and also the decomposition in Section 6.3.4 for the line-segment robot. It is sometimes referred to as *Collins decomposition* after its original developer [24, 168, 169]. In fact, the decomposition in Figure 6.18 can be considered as a cylindrical algebraic decomposition for a semi-algebraic set in which every geometric primitive is a linear polynomial. In this section, such a decomposition is generalized to any semi-algebraic set in \mathbb{R}^n .

The idea is to develop a sequence of projections that drops the dimension of the semi-algebraic set by one each time. Initially, the set is defined over \mathbb{R}^n , and after one projection, a semi-algebraic set is obtained in \mathbb{R}^{n-1} . Eventually, the projection reaches \mathbb{R} , and a univariate polynomial is obtained for which the zeros are at the critical places where cell boundaries need to be formed. A cell decomposition of 1-cells (intervals) and 0-cells is formed by partitioning \mathbb{R} . The sequence is then reversed, and decompositions are formed from \mathbb{R}^2 up to \mathbb{R}^n . Each iteration starts with a cell decomposition in \mathbb{R}^i and lifts it to obtain a cylinder of cells in \mathbb{R}^{i+1} . Figure 6.34 shows how the decomposition looks for the gingerbread example; since $n = 2$, it only involved one projection and one lifting.

Semi-algebraic projections are semi-algebraic The following is implied by the Tarski-Seidenberg Theorem [58]:

A projection of a semi-algebraic set from dimension n to dimension $n - 1$ is a semi-algebraic set

This gives a kind of closure of semi-algebraic sets under projection, which is required to ensure that every projection of a semi-algebraic set in \mathbb{R}^i leads to a semi-algebraic set in \mathbb{R}^{i-1} . This property is actually not true for (real) algebraic varieties, which were introduced in Section 4.4.1. These are defined using only the $=$ relation, and are not closed under the projection operation. Therefore, it is a good thing (not just a coincidence!) that we are using semi-algebraic sets.

Real algebraic numbers As stated previously, the sequence of projections ends with a univariate polynomial over \mathbb{R} . The sides of the cells will be defined based on the precise location of roots of this polynomial. Furthermore, representing a sample point for a cell of dimension k in a complex in \mathbb{R}^n for $k < n$, will require perfect precision. If the coordinates are slightly off, the point will lie in a different cell. This raises the complicated issue of how these roots are represented and manipulated in a computer.

For univariate polynomials of degree 4 or less, formulas exist to compute all of the roots in terms of functions of square roots and higher-order roots. From Galois theory [351, 611], it is known that such formulas and nice expressions for roots do not exist for higher-degree polynomials, which can certainly arise in the complicated semi-algebraic models formulated in motion planning. The roots in \mathbb{R} could be any real number, and many real numbers require infinite representations.

One way of avoiding this mess is to assume that only polynomials in $\mathbb{Q}[x_1, \dots, x_n]$ will be used, instead of the more general $\mathbb{R}[x_1, \dots, x_n]$. The field \mathbb{Q} is not *algebraically closed* because zeros the polynomial lie outside of \mathbb{Q}^n . For example, if $f(x_1) = x_1^2 - 2$, then $f = 0$ for $x_1 = \pm\sqrt{2}$, and $\sqrt{2} \notin \mathbb{Q}$. However, some elements of \mathbb{R} can never be a root of a polynomial in $\mathbb{Q}[x_1, \dots, x_n]$.

The set, \mathbb{A} , of all real roots to all polynomials in $\mathbb{Q}[x]$ is called the set of *real algebraic numbers*. The set $\mathbb{A} \subset \mathbb{R}$ of actually represents a field (recall from Section 4.4.1). Several nice algorithmic properties of the numbers in \mathbb{A} are: 1) they all have finite representations, 2) addition and multiplication operations on elements of \mathbb{A} can be computed in polynomial time, and 3) conversions between different representations of real algebraic numbers can be performed in polynomial time. This means that all operations can be done without resorting to some kind of numerical approximation. In some applications, such approximations are fine; however, for algebraic decompositions, they destroy critical information by potentially confusing roots (e.g., how can we know for sure whether a polynomial has multiple roots, or just two roots that are very close together?).

The details are not presented here, but there are several methods for representing real algebraic numbers and corresponding algorithms for manipulating them efficiently. The running time of cylindrical algebraic decomposition ultimately

depends on this representation. In practice, a numerical root finding method that has a precision parameter, ϵ , can be used by choosing ϵ small enough to ensure that roots will not be confused. A sufficiently small value can be determined by applying *gap theorems*, which give lower bounds on the amount of real root separation, expressed in terms of the polynomial coefficients [123]. Some methods avoid requiring a precision parameter. One well-known example is the derivation of a Sturm sequence of polynomials based on the given polynomial. The polynomials in the Sturm sequence are then used to find isolating intervals for each of the roots [58]. The polynomial, together with its isolating interval, can be considered as an example root representation. Algebraic operations can even be formed using this representation in time $O(d \lg^2 d)$, in which d is the degree of the polynomial [676]. See [58, 123, 676] for detailed presentations on the exact representation and calculation with real algebraic numbers.

One-dimensional decomposition To explain the method, we first perform a semi-algebraic decomposition of \mathbb{R} , which is the final step in the projection sequence. Once this is explained, then the multi-dimensional case will follow more easily.

Let \mathcal{F} be a set of m univariate polynomials

$$\mathcal{F} = \{f_i \in \mathbb{Q}[x] \mid i = 1, \dots, m\}, \quad (6.18)$$

that are used to define some semi-algebraic set in \mathbb{R} . The polynomials in \mathcal{F} could come directly from a quantifier-free formula ϕ (which could even appear inside of a Tarski sentence, as in (6.9)).

Define a single polynomial f as $f = \prod_{i=1}^m f_i$. Suppose that f has k distinct, real roots, which are sorted in increasing order:

$$-\infty < \beta_1 < \beta_2 < \dots < \beta_{i-1} < \beta_i < \beta_{i+1} < \dots < \beta_k < \infty. \quad (6.19)$$

The one-dimensional semi-algebraic decomposition is given by the following sequence of alternating 1-cells and 0-cells:

$$(-\infty, \beta_1), [\beta_1, \beta_1], (\beta_1, \beta_2), \dots, (\beta_{i-1}, \beta_i), [\beta_i, \beta_i], (\beta_i, \beta_{i+1}), \dots, [\beta_k, \beta_k], (\beta_k, \infty). \quad (6.20)$$

Any semi-algebraic set can be expressed using the polynomials in \mathcal{F} can be expressed as the union some of the 0-cells and 1-cells given in (6.20). This can also be considered as a singular complex (it can even be considered as a simplicial complex, but this will not be true in higher dimensions).

Sample points can be generated for each of the cells as follows. For the unbounded cells, $[-\infty, \beta_1)$ and $(\beta_k, \infty]$, valid samples are $\beta_1 - 1$ and $\beta_k + 1$, respectively. For each finite 1-cell, (β_i, β_{i+1}) , the midpoint $(\beta_i + \beta_{i+1})/2$ produces a sample point. For each 0-cell, $[\beta_i, \beta_i]$, the only choice is to use β_i as the sample point.

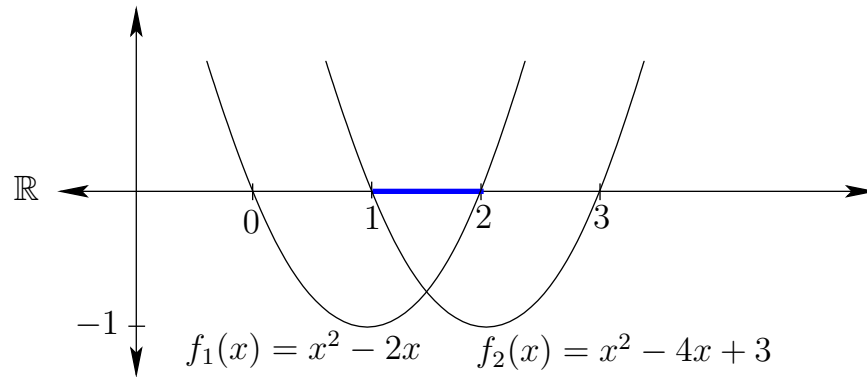


Figure 6.30: Two parabolas are used to define the semi-algebraic set $[1, 2]$.

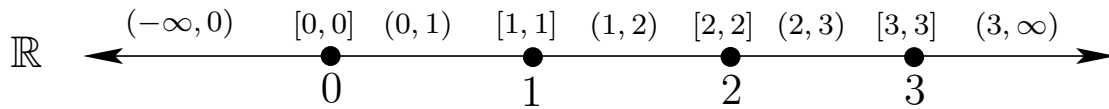


Figure 6.31: A semi-algebraic decomposition for the polynomials in Figure 6.30.

Example 6.4.4 Figure 6.30 shows a semi-algebraic subset of \mathbb{R} that is defined by two polynomials, $f_1(x) = x^2 - 2x$ and $f_2(x) = x^2 - 4x + 3$. Thus, $\mathcal{F} = \{f_1, f_2\}$. Consider quantifier-free formula

$$\phi(x) = (x^2 - 2x \geq 0) \wedge (x^2 - 4x + 3 \geq 0) \quad (6.21)$$

The semi-algebraic decomposition into 5 1-cells and 4 0-cells is shown in Figure 6.31. Note that each cell is sign-invariant. The sample points for the 1-cells are -1 , $1/2$, $3/2$, $5/2$, and 4 , respectively. The sample points for the 0-cells are 0 , 1 , 2 , and 3 , respectively.

A decision problem can be nicely solved using the decomposition. Suppose a Tarski sentence that uses the polynomials in \mathcal{F} has been given. Here is one possibility:

$$\Phi = \exists x[(x^2 - 2x \geq 0) \wedge (x^2 - 4x + 3 = 0)] \quad (6.22)$$

The sample points alone are sufficient to determine whether Φ is TRUE or FALSE. Once $x = 1$ is attempted, it is discovered that Φ is TRUE. The quantifier elimination problem cannot yet be considered because more dimensions are needed. ■

The inductive step to higher dimensions Now consider constructing a cylindrical (semi-)algebraic decomposition for \mathbb{R}^n . Figure 6.34 shows an example for \mathbb{R}^2 . First consider how to iteratively project the polynomials down to \mathbb{R} to ensure that when the decomposition of \mathbb{R}^n is constructed, the sign invariant property is maintained. It will also be the case that the resulting decomposition will correspond directly to a singular complex.

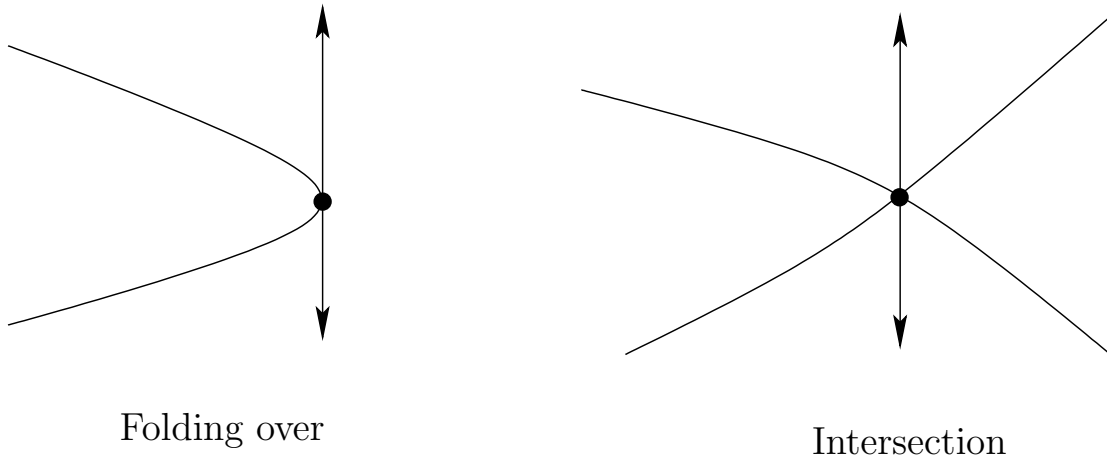


Figure 6.32: Critical points occur either when the surface folds over in the vertical direction or when surfaces intersect.

Let \mathcal{F}_n denote the original set of polynomials in $\mathbb{Q}[x_1, \dots, x_n]$ that are used to define the semi-algebraic set (or Tarski sentence) in \mathbb{R}^n . Form a single polynomial $f = \prod_{i=1}^m f_i$. Let $f' = \partial f / \partial x_n$, which is also a polynomial. Let $g = \text{GCD}(f, f')$, which is the greatest common divisor of f and f' . The set of zeros of g are all points which are both zeros of both f and f' . Being a zero of f' means that the surface given by $f = 0$ does not vary locally when perturbing x_n . These are places where a cell boundary needs to be formed because the surface may fold over itself in the x_n direction, which is not permitted for a cylindrical decomposition. Another place where a cell boundary needs to be formed is at the intersection of two or more polynomials in \mathcal{F}_n . The projection technique from \mathbb{R}^n to \mathbb{R}^{n-1} generates a set, \mathcal{F}_{n-1} , of polynomials in $\mathbb{Q}[x_1, \dots, x_{n-1}]$, that satisfy these requirements. The polynomials \mathcal{F}_{n-1} have the property that at least one contains a zero point below every point in $x \in \mathbb{R}^n$ for which $f(x) = 0$ and $f'(x) = 0$, or polynomials in \mathcal{F}_n intersect. The projection method that constructs \mathcal{F}_{n-1} involves computing *principle subresultant coefficients*, which are covered in [58, 677]. Resultants, of which the subresultants are an extension, are covered in [178].

The polynomials in \mathcal{F}_{n-1} are then projected to \mathbb{R}^{n-2} to obtain \mathcal{F}_{n-2} . This process continues until \mathcal{F}_1 is obtained, which is a set of polynomials in $\mathbb{Q}[x_1]$. A one-dimensional decomposition is formed, as defined earlier. From \mathcal{F}_1 , a single polynomial is formed by taking the product, and \mathbb{R} is partitioned into 0-cells and 1-cells. We next describe the process of lifting a decomposition over \mathbb{R}^{i-1} up to \mathbb{R}^i . This technique is applied iteratively until \mathbb{R}^n is reached.

Assume inductively that a cylindrical algebraic decomposition has been computed for a set of polynomials \mathcal{F}_{i-1} in $\mathbb{Q}[x_1, \dots, x_{i-1}]$. The decomposition consists of k -cells for which $0 \leq k \leq i$. Let $p = (x_1, \dots, x_{i-1}) \in \mathbb{R}^{i-1}$. For each one of the k -cells, C_{i-1} , a *cylinder* over C_{i-1} is defined as the $(k+1)$ -dimensional set

$$\{(p, x_i) \in \mathbb{R}^i \mid p \in C_{i-1}\} \quad (6.23)$$

The cylinder is sliced into a strip of k -dimensional and $k + 1$ -dimensional cells by using polynomials in \mathcal{F}_i . Let f_j denote one of the ℓ slicing polynomials in the cylinder, sorted in increasing x_i order as $f_1, f_2, \dots, f_j, f_{j+1}, \dots, f_\ell$. The following kinds of cells are produced (see Figure 6.33):

Lower unbounded sector:

$$\{(p, x_i) \in \mathbb{R}^i \mid p \in C_{i-1} \text{ and } x_i < f_1(p)\} \quad (6.24)$$

Section:

$$\{(p, x_i) \in \mathbb{R}^i \mid p \in C_{i-1} \text{ and } x_i = f_i(p)\} \quad (6.25)$$

Bounded sector:

$$\{(p, x_i) \in \mathbb{R}^i \mid p \in C_{i-1} \text{ and } f_j(p) < x_i < f_{j+1}(p)\} \quad (6.26)$$

Upper unbounded sector:

$$\{(p, x_i) \in \mathbb{R}^i \mid p \in C_{i-1} \text{ and } f_\ell(p) < x_i\}. \quad (6.27)$$

There is one degenerate possibility in which there are no slicing polynomials, and the cylinder over C_{i-1} can be extended into one unbounded cell. In general, the sample points are computed by picking a point in $p \in C_{i-1}$ and making a vertical column of samples of the form (p, x_i) . A polynomial in $\mathbb{Q}[x_i]$ can be generated, and the samples are placed using the same assignment technique as used for the one-dimensional decomposition.

Example 6.4.5 (Mutilating the gingerbread face) Figure 6.34 shows a cylindrical algebraic decomposition of the gingerbread face. It can be seen that the resulting complex is very similar to that obtained in Figure 6.18. ■

It is important to note that the cells do not necessarily project onto a rectangular set, as in the case of the higher-dimensional vertical decomposition. For example, a generic n -cell, C_n , for a decomposition of \mathbb{R}^n is described as the open set of $(x_1, \dots, x_n) \in \mathbb{R}^n$ such that

- $C_0 < x_n < C'_0$ for some 0-cells $C_0, C'_0 \in \mathbb{R}$ which are roots of some $f, f' \in \mathcal{F}_1$.
- (x_{n-1}, x_n) lies between C_1 and C'_1 for some 1-cells C_1, C'_1 which are zeros of some $f, f' \in \mathcal{F}_2$.
- \vdots
- (x_{n-i}, \dots, x_n) lies between C_{i-1} and C'_{i-1} for some i -cells C_{i-1}, C'_{i-1} which are zeros of some $f, f' \in \mathcal{F}_i$.

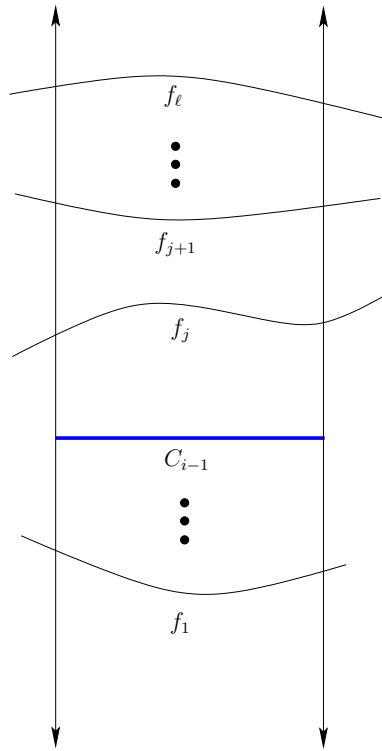


Figure 6.33: A cylinder over every k -cell C_{i-1} is formed. A sequence of polynomials, f_1, \dots, f_ℓ , slices the cylinder into k -dimensional sections and $(k + 1)$ -dimensional sectors.

\vdots

- (x_1, \dots, x_n) lies between C_{n-1} and C'_{n-1} for some $(n - 1)$ -cells C_{n-1}, C'_{n-1} which are zeros of some $f, f' \in \mathcal{F}_n$.

The resulting decomposition is sign-invariant, which allows the decision and quantifier elimination problems to be solved in finite time. To solve a decision problem, the polynomials in \mathcal{F}_n are evaluated at every sample point to determine whether one of them satisfies the Tarski sentence. To solve the quantifier elimination problem, note that any semi-algebraic sets that can be constructed from \mathcal{F}_n can be defined as a union of some cells in the decomposition. For the given Tarski sentence, \mathcal{F}_n is formed from all polynomials that are mentioned in the sentence, and the cell decomposition is performed. Once obtained, the sign information is used to determine which cells need to be included in the union. The resulting union of cells is designed to include only the points in \mathbb{R}^n at which the Tarski sentence is TRUE .

Solving a motion planning problem The cylindrical algebraic decomposition is also capable of solving any motion planning problems formulated in Chapter 4.

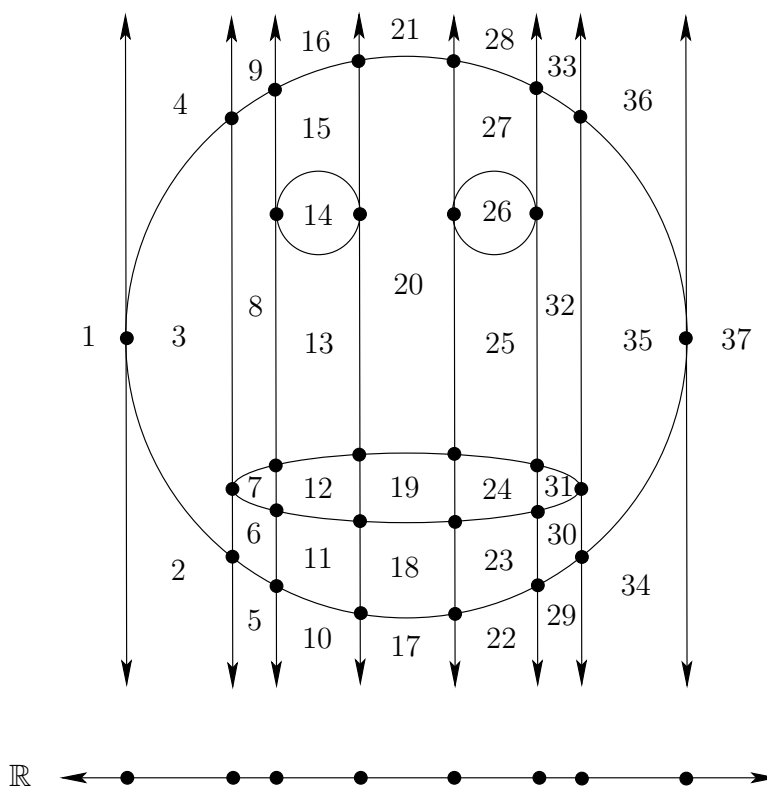


Figure 6.34: A cylindrical algebraic decomposition of the gingerbread face. There are 37 2-cells, 64 1-cells, and 28 0-cells. The straight 1-cells are intervals of the vertical lines, and the curved ones are portions of the zero set of a polynomial in \mathcal{F} . The decomposition of \mathbb{R} is also shown.

First assume that $\mathcal{C} = \mathbb{R}^n$. Just as for other decompositions, a roadmap is formed in which every vertex is an n -cell, and edges connect every pair of adjacent n -cells by traveling through an $(n - 1)$ -cell. It is straightforward to determine adjacencies inside of a cylinder, but there are several technical details associated with determining adjacencies of cells from different cylinders [58] (pages 152-154 present an example that illustrates the problem). The cells of dimension less than $n - 1$ are not needed for motion planning purposes (just as vertices were not needed for the vertical decomposition in Section 6.2.2). The query points, q_i and q_g are connected to the roadmap depending on the cell in which they lie, and a discrete search is performed.

If $\mathcal{C} \subset \mathbb{R}^n$ and its dimension is k for $k < n$, then all of the interesting cells are of lower dimension. This occurs, for example, due to the constraints on the matrices to force them to lie in $SO(2)$ or $SO(3)$. This may also occur for problems from Section 4.4, in which closed chains reduce the degrees of freedom. The cylindrical algebraic decomposition method can still solve such problems; however, the exact root representation problem becomes more complicated when determining the cell adjacencies. A discussion of these issues appears in [676]. For the case of $SO(2)$

and $SO(3)$, this complication can be avoided by using *stereographic projection* to map \mathbb{S}^1 or \mathbb{S}^3 to \mathbb{R} or \mathbb{R}^3 , respectively. This mapping removes a single point from each, but the connectivity of \mathcal{C}_{free} remains unharmed. The antipodal identification problem for unit quaternions represented by \mathbb{S}^3 also does not present a problem; there is a redundant copy of \mathcal{C} , which does not affect the connectivity.

The running time for cylindrical algebraic decomposition depends on many factors, but in general it is polynomial in the number of polynomials in \mathcal{F}_n , polynomial in the maximum algebraic degree of the polynomials, and doubly-exponential in the dimension. Complexity issues will be covered in more detail in Section 6.5.3.

6.4.3 Canny's Roadmap Algorithm

The doubly-exponential running time of cylindrical algebraic decomposition inspired researchers to do better. It has been shown that quantifier elimination requires doubly-exponential time [187]; however, motion planning is a different problem. Canny introduced a method that produces a roadmap directly from the semi-algebraic set, rather than constructing a cell decomposition along the way. Since there are doubly-exponentially many cells in the cylindrical algebraic decomposition, avoiding this construction pays off. The resulting roadmap method of Canny solves the motion planning problem in time that is again polynomial in the number of polynomials, polynomial in the algebraic degree, but is only singly-exponential in dimension [123].

Much like the other combinatorial motion planning approaches, it is based on finding critical curves and points. The main idea is to construct linear mappings from \mathbb{R}^n to \mathbb{R}^2 that produce *silhouette curves* of the semi-algebraic sets. Performing one such mapping on the original semi-algebraic set will yield a roadmap, but it might not preserve the original connectivity. Therefore, linear mappings from \mathbb{R}^{n-1} to \mathbb{R}^2 are performed on some $(n-1)$ -dimensional slices of the original semi-algebraic set to yield more roadmap curves. This process is applied recursively until the slices are already one-dimensional. The resulting roadmap is formed from the union of all the pieces obtained in the recursive calls. The resulting roadmap was shown to have the same connectivity as the original semi-algebraic set [123].

Suppose that $\mathcal{C} = \mathbb{R}^n$. Let $\mathcal{F} = \{f_1, \dots, f_m\}$ denote the set of polynomials that define the semi-algebraic set, which is assumed to be represented as a disjoint union of manifolds. Assume that each $f_i \in \mathbb{Q}[x_1, \dots, x_n]$. First, a small perturbation to the input polynomials \mathcal{F} is performed to ensure that every sign-invariant set of \mathbb{R}^n is a manifold. This forces the polynomials into a kind of general position, which can be achieved with probability one using random perturbations; there are also deterministic methods to solve this problem. The general position requirements on the input polynomials and the 2D projection directions are fairly strong, which has stimulated more recent work that eliminates many of the problems [58]. From this point onward, it will be assumed that the polynomials are in

general position.

Recall the sign assignment function from Section 6.4.1. Each sign-invariant set is a manifold because of the general position assumption. Canny's method computes a roadmap for any k -dimensional manifold for $k < n$. Such a manifold will have precisely $n - k$ signs that are 0 (which means that points lie precisely on the zero sets of $n - i$ polynomials in \mathcal{F}). At least one of the signs must be 0, which means that Canny's roadmap actually lies in $\partial\mathcal{C}_{free}$ (this technically is not permitted, but nevertheless the algorithm correctly decides whether a solution path exists through \mathcal{C}_{free}).

Recall that each f_i is a function $\mathbb{R}^n \rightarrow \mathbb{R}$. Let x denote $(x_1, \dots, x_n) \in \mathbb{R}^n$. The k polynomials that have zero signs can be put together sequentially to produce a mapping $\psi : \mathbb{R}^n \rightarrow \mathbb{R}^k$. The i^{th} component of the vector $\psi(x)$ is $\psi_i(x) = f_i(x)$. This is closely related to the sign assignment function of Section 6.4.1, except that now the real value from each polynomial is directly used, rather than taking its sign.

Now introduce a function, $g : \mathbb{R}^n \rightarrow \mathbb{R}^j$, in which either $j = 1$ or $j = 2$ (the general concepts presented below will work for other values of j , but 1 and 2 are the only values needed for Canny's method). The function g will serve the same purpose as a projection in cylindrical algebraic decomposition, but note that g immediately drops from dimension n to dimension 2 or 1, instead of dropping to $n - 1$ as in the case of cylindrical projections and liftings.

Let $h : \mathbb{R}^n \rightarrow \mathbb{R}^{k+j}$ denote a mapping that constructed directly from ψ and g as follows. For the i^{th} component, if $i \leq k$, then $h_i(x) = \psi_i(x) = f_i(x)$. Assume that $k + j \leq n$. If $i > k$, then $h_i(x) = g_{i-k}(x)$. Let $J_x(h)$ denote the *Jacobian* of h , at x be defined as

$$J_x(h) = \begin{pmatrix} \frac{\partial h_1(x)}{\partial x_1} & \dots & \frac{\partial h_1(x)}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial h_{m+k}(x)}{\partial x_1} & \dots & \frac{\partial h_{m+k}(x)}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1(x)}{\partial x_1} & \dots & \frac{\partial f_1(x)}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_k(x)}{\partial x_1} & \dots & \frac{\partial f_k(x)}{\partial x_n} \\ \frac{\partial g_1(x)}{\partial x_1} & \dots & \frac{\partial g_1(x)}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial g_j(x)}{\partial x_1} & \dots & \frac{\partial g_j(x)}{\partial x_n} \end{pmatrix}. \quad (6.28)$$

A point $x \in \mathbb{R}^n$ at which $J_x(h)$ is singular is called a *critical point*. The matrix is defined to be *singular* if every $(m+k) \times (m+k)$ subdeterminant is zero. Each of the first k rows of $J_x(h)$ calculates the surface normal to $f_i(x) = 0$. If these normals are not linearly independent of the directions given by the last j rows, then the matrix becomes singular. The following example from [119] nicely illustrates this principle.

Example 6.4.6 Let $n = 3$, $k = 1$, and $j = 1$. The zeros of a single polynomial f_1 define a two-dimensional subset of \mathbb{R}^3 . Let f_1 be the unit sphere, \mathbb{S}^2 , defined as the zeros of the polynomial

$$f_1(x_1, x_2, x_3) = x_1^2 + x_2^2 + x_3^2 - 1. \quad (6.29)$$

Suppose that $g : \mathbb{R}^3 \rightarrow \mathbb{R}$ is defined as $g(x_1, x_2, x_3) = x_1$. The Jacobian (6.28) becomes

$$\begin{pmatrix} 2x_1 & 2x_2 & 2x_3 \\ 1 & 0 & 0 \end{pmatrix}, \quad (6.30)$$

and is singular when all 3 of the possible 2×2 subdeterminants are zero. This occurs if and only if $x_2 = x_3 = 0$. This yields the critical points $(-1, 0, 0)$ and $(1, 0, 0)$ on \mathbb{S}^2 . Note that this is precisely when the surface normals of \mathbb{S}^2 are parallel to the vector $[1 \ 0 \ 0]$.

Now suppose that $j = 2$ to obtain $g : \mathbb{R}^3 \rightarrow \mathbb{R}^2$, which is defined as $g(x_1, x_2, x_3) = (x_1, x_2)$. In this case, (6.28) becomes

$$\begin{pmatrix} 2x_1 & 2x_2 & 2x_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \quad (6.31)$$

which is singular if and only if $x_3 = 0$. The critical points are therefore the X_1X_2 -plane intersected with \mathbb{S}^3 , which yields the equator points (all $(x_1, x_2) \in \mathbb{R}^2$ such that $x_1^2 + x_2^2 = 1$). In this case, more points are generated because the matrix becomes degenerate for any surface normal of \mathbb{S}^2 that is parallel to $[1 \ 0 \ 0]$, $[0 \ 1 \ 0]$, or any linear combination of them. ■

The first mapping in Example 6.4.6 yielded two isolated critical points, and the second mapping yielded a one-dimensional set of critical points, which is referred to as a *silhouette*. The union of the silhouette and the isolated critical points yields a roadmap for \mathbb{S}^2 . Now consider generalizing this example to obtain the full algorithm for general n and k . A linear mapping $g : \mathbb{R}^n \rightarrow \mathbb{R}^2$ is constructed, which might not be axis-aligned as in Example 6.4.6 because it must be chosen in general position (otherwise degeneracies might arise in the roadmap). Define ψ to be the set of polynomials that become zero on the desired manifold on which to construct a roadmap. Form the matrix (6.28), and determine the silhouette. This is accomplished in general using subresultant techniques which were also needed for cylindrical algebraic decomposition; see [58, 123] for details. Let g_1 denote the first component of g , which yields a mapping $g_1 : \mathbb{R}^n \rightarrow \mathbb{R}$. Forming (6.28) using g_1 yields a finite set of critical points. Taking the union of the critical points and the silhouette produces part of the roadmap.

So far, however, there are no guarantees that the connectivity is preserved. To handle this problem, the Canny's algorithm proceeds recursively. For each of the critical points, $x \in \mathbb{R}^n$, an $n - 1$ -dimensional hyperplane through x is chosen for

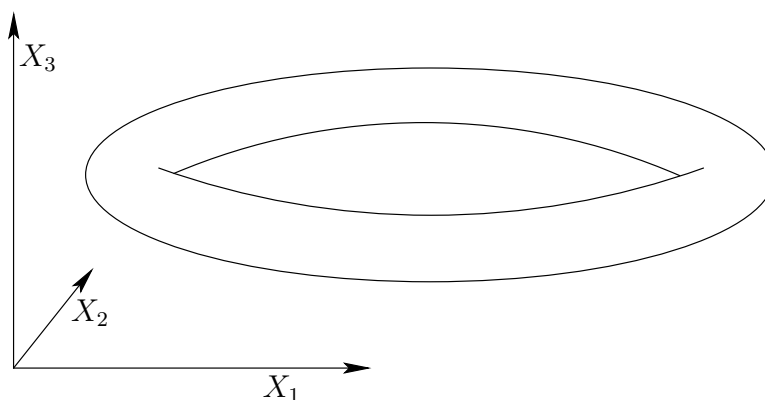


Figure 6.35: Suppose that the semi-algebraic set is a solid torus in \mathbb{R}^3 .

which the g_1 row of (6.28) is the normal (hence it is perpendicular in some sense to the flow of g_1). Inside of this hyperplane, a new g mapping is formed. This time a new direction is chosen, and the mapping takes the form $g : \mathbb{R}^{n-1} \rightarrow \mathbb{R}^2$. Once again, the silhouettes and critical points are founded and added to the roadmap. This process is repeated recursively until the base case in which the silhouettes and critical points are directly obtained without forming g .

It is helpful now to consider an example. Since the method involves a sequence of 2D projections, it is difficult to visualize. Examples in \mathbb{R}^4 and higher involve more than one of the 2D projections. An example over \mathbb{R}^3 is presented here; see [123] for another example over \mathbb{R}^3 .

Example 6.4.7 (The solid torus in \mathbb{R}^3) Consider three-dimensional algebraic set shown in Figure 6.35. After defining the mapping $g(x_1, x_2, x_3) = (x_1, x_2)$, the roadmap shown in Figure 6.36 is obtained. The silhouette are obtained from g , and the critical points are obtained from g_1 . Note that the original connectivity of the solid torus is not preserved because the inner ring does not connect to the outer ring. This illustrates the need to also compute the roadmap for lower-dimensional slices. For each of the four critical points, the critical curves are computed for a plane that is parallel to the X_2X_3 plane, and for which the x_1 position is determined by the critical point. The slice for one of the inner critical points is shown in Figure 6.37. In this case, the slice already has two dimensions. New silhouette curves are added to the roadmap to obtain the final result shown in Figure 6.38. ■

To solve a planning problem, the query points q_i and q_g are artificially declared to be critical points in the top level of recursion. This forces the algorithm to generate curves that connect them to the rest of the roadmap.

The completeness of the method requires very careful analysis, which is thoroughly covered in [58, 123]. The main elements to the analysis are: 1) showing that the polynomials can be perturbed and g can be chosen to ensure general position, 2) the singularity conditions on (6.28) lead to algebraic sets (varieties), and

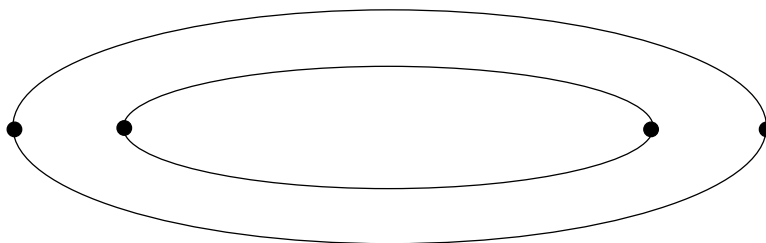


Figure 6.36: The projection into the X_1X_2 plane yields silhouettes for the inner and outer rings, and also four critical points.

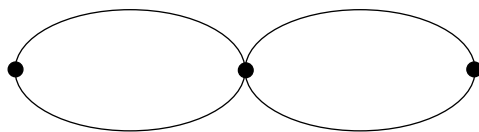


Figure 6.37: A slice taken for the inner critical points is parallel to the X_2X_3 plane. The roadmap for the slice connects to the silhouettes from Figure 6.36, which preserves the connectivity of the original set in Figure 6.35.

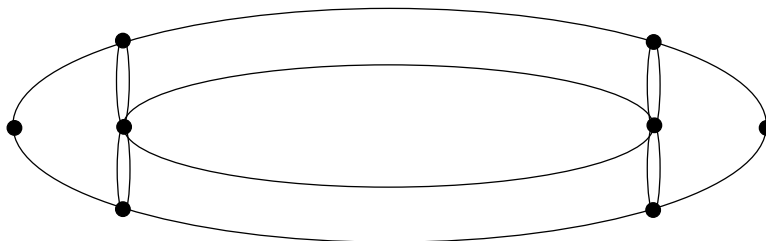


Figure 6.38: All of the silhouettes and critical points are merged to obtain the roadmap.

3) the resulting roadmap has the required properties mentioned in Section 6.1 of being accessible and connectivity-preserving for \mathcal{C}_{free} (actually it is shown for the $\partial\mathcal{C}_{free}$). The method above explained how to compute the roadmap for each sign-invariant set, but to obtain a roadmap for the planning problem, the roadmaps from each sign-invariant set must be connected together correctly; fortunately, this was established. See the Linking Lemma of [119].

6.5 Complexity of Motion Planning

This section summarizes some theoretical work that characterizes the complexity of motion planning problems. Note this not equivalent to characterizing the running time of particular algorithms. The existence of an algorithm serves as an *upper bound* on the problem difficulty because it is a proof by example that solving the problem requires no more time than what is needed by the algorithm. On the other hand, *lower bounds* are also very useful because they give an indication of the difficulty of the problem itself. Suppose, for example, you are given an algorithm that solves a problem in time $O(n^2)$. Does it make sense to try to find a more efficient algorithm? Does it make sense to try to find a general-purpose motion algorithm that runs in time that is polynomial in the dimension? Lower bounds provide answers to questions such as this. Usually lower bounds are obtained by concocting bizarre, complicated examples that are allowed by the problem definition, but probably not considered by the person who first formulated the problem. In this line of research, progress is made by either raising the lower bound (unless it is already tight), or by showing that a narrower version of the problem is still allows such bizarre examples. The latter case occurs often in motion planning.

6.5.1 Lower Bounds

Lower bounds have been established for a variety of motion planning problems, and also a wide variety of planning problems in general. To interpret these bounds a basic understanding of the *theory of computation* is required [339, 711]. This fascinating subject will be unjustly summarized in a few paragraphs. A *problem* is a set of *instances* that each are carefully encoded as a binary string. An *algorithm* is formally considered as a *Turing machine*, which is a finite-state machine that can read and write bits to an unbounded piece of tape. Algorithms are usually formulated to make a binary output, which involves *accepting* or *rejecting* a problem instance that is initially written to the tape and given to the algorithm. In motion planning, this amounts to deciding whether or not a solution path exists for a given problem instance.

Languages A *language* is a set of binary strings associated with a problem. It represents the complete set of instances of a problem. An algorithm is said to *decide* a language if in finite time it correctly report accepts all strings that belong

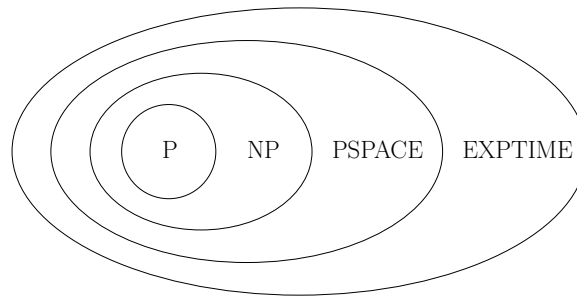


Figure 6.39: It is known that $P \subset EXPTIME$ is a strict subset; however, it is not known precise how large NP and $PSPACE$ are.

to it, and rejects all others. The interesting question is how much time or space is required to decide a language? This question is asked of the problem, under the assumption that the best possible algorithm would be used to decide it. (We can easily think of inefficient algorithms that waste resources.)

A *complexity class* is a set of languages that can all be decided within some specified resource bound. The class P is the set of all languages (and hence problems) for which a polynomial-time algorithm exists (i.e., the algorithm runs in time $O(n^k)$ for some integer k). By definition, an algorithm is called *efficient* if it decides its associated language in polynomial time.⁷ If no efficient algorithm exists, then the problem is called *intractable*. The relationship between several other classes that often emerge in theoretical motion planning is shown in Figure ???. The class NP is the set of languages that can be solved in polynomial time by a *nondeterministic Turing machine*. Some discussion of nondeterministic machines appears in Section ??. Intuitively, it means that solutions can be verified in polynomial time because the machine magically knows which choices to make while trying to make the decision. The class $PSPACE$ is the set of languages that can be decided with no more than a polynomial amount of storage space during the execution of the algorithm ($NPSPACE=PSPACE$, so there is no nondeterministic version). The class $EXPTIME$ is the set of languages that can be decided in time $O(2^{n^k})$ for some integer k . It is known that $EXPTIME$ is larger than P , but it is not known precisely where NP and $PSPACE$ lie. It might be the case that $P = NP = PSPACE$ (although hardly anyone believes this), or it could be that $NP = PSPACE = EXPTIME$. Because of this uncertainty, one cannot say that a problem is intractable if it is NP -hard or $PSPACE$ -hard; one can, however, if the problem is $EXPTIME$ -hard. One additional remark: it is convenient to remember that $PSPACE$ -hard implies NP -hard.

Hardness and completeness Since an easier class is included as a subset of a harder one, it is helpful to have a notion of a language (i.e., problem) being

⁷Note that this definition may be absurd in practice; an algorithm that runs in time $O(n^{90125})$ would probably not be too efficient for most purposes.

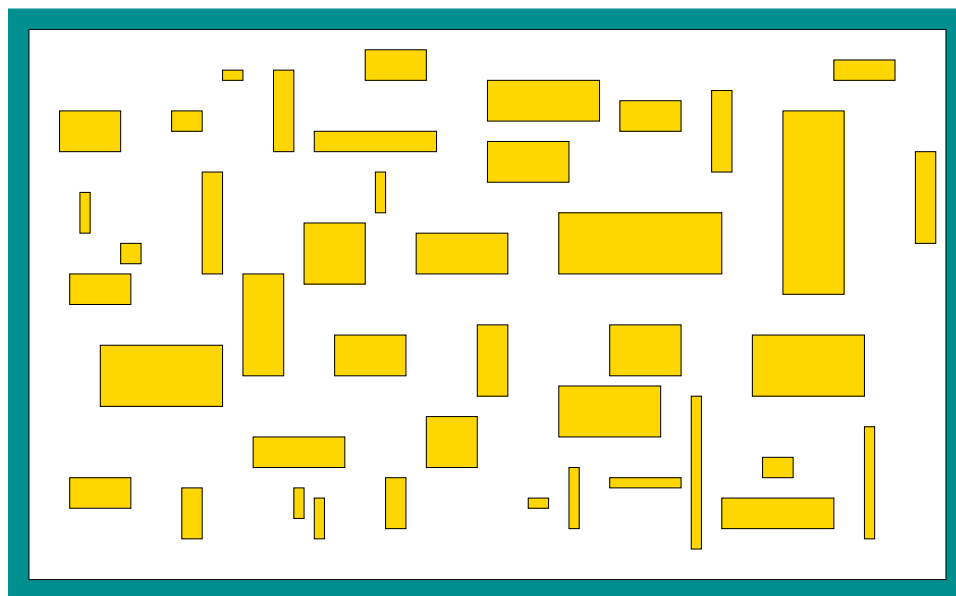


Figure 6.40: Even motion planning for a bunch of translating rectangles inside of a rectangular box in \mathbb{R}^2 is PSPACE-hard.

among the hardest possible within a class. Let X refer to either P, NP, PSPACE, or EXPTIME. A language A is called X -hard if every language, B , in class X is *polynomial time reducible* to A . In short, this means that in polynomial time, any language in B can be translated into instances for language A , and then the decisions for A can be correctly translated back in polynomial time to correctly decide B . Thus, if A can be decided, then within a polynomial-time factor, every language in X can be decided. The hardness concept can even be applied to a language (problem) that does not belong to the class. For example, we can declare that a language A is NP-hard even if $A \notin NP$ (it could be harder, and lie in EXPTIME, for example). If it is known that the language is both hard for some class X and is also a member of X , then it is called X -complete (i.e., NP-complete, PSPACE-complete, etc.).⁸

Lower bounds for motion planning The general motion planning problem, Formulation 4.3.1, was shown in 1979 to be PSPACE-hard by Reif [651]. In fact, the problem was restricted to polyhedral obstacles and a finite number of polyhedral robot bodies attached by spherical joints. The coordinates of all polyhedra are assumed to be in \mathbb{Q} (this enables a finite-length string encoding of the problem instance. The proof introduces a fascinating motion planning instance that involves many attached, dangling robot parts that must work their way through a

⁸If you remember hearing that a planning problem is NP-something, but cannot remember whether it was NP-hard or NP-complete, then it is safe to say NP-hard because NP-complete implies NP-hard. This can similarly be said for other classes, such as PSPACE-complete vs. PSPACE-hard.

complicated system of tunnels, which together simulates the operation of a *symmetric Turing machine*. Canny later established that the problem in Formulation 4.3.1 (with rational polynomial coefficients) lies in PSPACE. Therefore, the general motion planning problem is PSPACE-complete.

Many other lower bounds have been shown for a variety of planning problems. One famous example is the Warehouseman's problem shown in Figure 6.40. This problem involves a finite number of translating, axis-aligned rectangles in a rectangular world. It was shown in [338] to be PSPACE-hard. This example is a beautiful illustration of how such a deceptively simple problem formulation can lead to such a high lower bound. More recently, it was even shown that planning for Sokoban, which is a warehouseman's problem on a discrete 2D grid, is also PSPACE hard [182]. Other general motion planning problems that were shown to be PSPACE-hard include motion planning for a chain of bodies in the plane [337, 370], and motion planning for a chain of bodies among polyhedral obstacles in \mathbb{R}^3 . Many lower bounds have been established for a variety of extensions and variations of the general motion planning problem. For example, in [122] it was established that a certain form of planning under uncertainty for a robot in a 3D polyhedral environment is NEXPTIME-hard, which is harder than any of the classes shown in Figure 6.39; the hardest problems in this NEXPTIME hard are believed to require doubly-exponential time to solve.

These lower-bound or hardness results depend significantly on the precise representation of the problem. For example, it is possible to make problems look easier by making instance encodings that are exponentially longer than they should be. The running time or space required is expressed in terms of n , the input size. If the motion planning problem instances are encoded with exponentially more bits than necessary, then a language that belongs to P will be obtained. As long as the instance encoding is within a polynomial factor of the optimal encoding, then this bizarre behavior is avoided. Another important part of the representation is to pay attention to how parameters in the problem formulation can vary. We can redefine motion planning to be all instances for which the dimension of \mathcal{C} is never greater than 2^{1000} . The number of dimensions is sufficiently large for virtually any application. The resulting language for this problem belongs to P because cylindrical algebraic decomposition and Canny's algorithm can solve any motion planning problem in polynomial time. Why? Because now the dimension parameter in the time complexity expressions can be replaced by 2^{1000} , which is a constant. This formally implies that an *efficient* algorithm exists for any motion planning problem that we would ever care about. This implication has no practical value, however. Thus, be very careful when interpreting theoretical bounds.

The lower bounds may appear discouraging. There are two general directions to go from here. One is weaken the requirements, and tolerate algorithms that yield some kind of resolution, dispersion, or probabilistic completeness. This approach was taken in Chapter 5, and leads to many efficient algorithms. Another direction is to define narrower problems that do not include the bizarre construc-

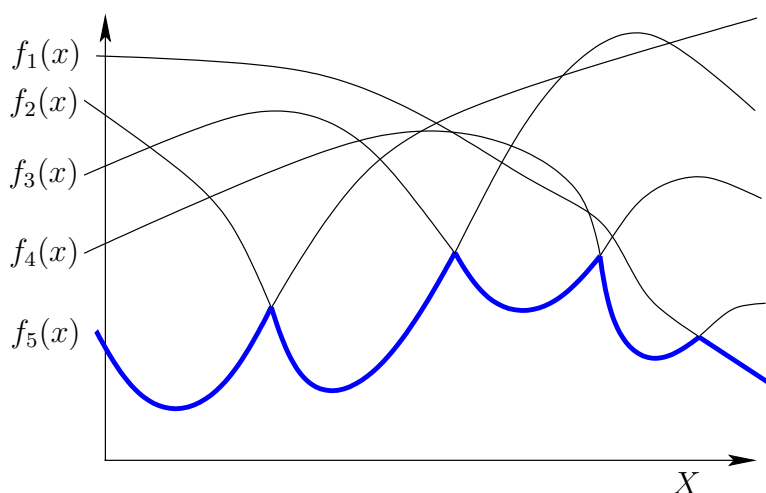


Figure 6.41: The lower envelope of a collection of functions.

tions that led to bad lower bounds. For the narrower problems, it may be possible to design interesting, efficient algorithms. This approach was taken for the methods in Sections 6.2 and 6.3. In Section 6.5.3, upper bounds for some algorithms that address these narrower problems will be presented, along with bounds for the general motion planning algorithms. Several of the upper bounds involve Davenport-Schinzel sequences, which are therefore covered next.

6.5.2 Davenport-Schinzel Sequences

Davenport-Schinzel sequences provide a powerful characterization of the structure that arises from the lower or upper envelope of a collection of functions. The lower envelope of five functions is depicted in Figure 6.41. Such envelopes arise in many problems throughout computational geometry, including many motion planning problems. They are an important part of the design and analysis of many modern algorithms, and the resulting algorithm time-complexity usually involves terms that follow directly from the sequences. Therefore, it is worthwhile to understand some of the basics before interpreting some of the results of Section 6.5.3. Much more information on Davenport-Schinzel sequences and their applications appears in [687]. The brief introduction presented here is based on [686].

For positive integers n and s , an (n, s) *Davenport-Schinzel sequence* is a sequence (u_1, \dots, u_m) composed from a set of n symbols such that

1. The same symbol may not appear consecutively in the sequence. In other words, $u_i \neq u_{i+1}$ for any i such that $1 \leq i < m$.
2. The sequence does not contain any alternating subsequence that uses two symbols and has length $s + 2$. A subsequence can be formed by deleting any elements in the original sequence. The condition can be expressed as there

does not exist $s+2$ indices $i_1 < i_2 < \dots < i_{s+2}$ for which $u_{i_1} = u_{i_3} = u_{i_5} = a$ and $u_{i_2} = u_{i_4} = u_{i_6} = b$, for some symbols a and b .

As an example, an $(n, 3)$ sequence cannot appear as $(a \dots b \dots a \dots b \dots a)$, in which each \dots is filled in with any sequence of symbols. Let $\lambda_s(n)$ denote the maximum possible length of an (n, s) Davenport-Schinzel sequence.

The connection between Figure 6.41 can now be explained. Consider the sequence of function indices that visit the lower envelope. In the example, this sequence is $(5, 2, 3, 4, 1)$. Suppose it is known that each pair of functions intersects in at most s places. If there are n real-valued continuous functions, then the sequence of function indices must be an (n, s) Davenport-Schinzel sequence. It is amazing that such sequences cannot be very long. For a fixed s , they are close to being linear.

The standard bounds for Davenport-Schinzel sequences are [686]⁹:

$$\lambda_1(n) = n \tag{6.32}$$

$$\lambda_2(n) = 2n - 1 \tag{6.33}$$

$$\lambda_3(n) = \Theta(n\alpha(n)) \tag{6.34}$$

$$\lambda_4(n) = \Theta(n \cdot 2^{\alpha(n)}) \tag{6.35}$$

$$\lambda_{2s}(n) \leq n \cdot 2^{\alpha(n)^{s-1} + C_{2s}(n)} \tag{6.36}$$

$$\lambda_{2s+1}(n) \leq n \cdot 2^{\alpha(n)^{s-1} \lg \alpha(n) + C'_{2s+1}(n)} \tag{6.37}$$

$$\lambda_{2s}(n) = \Omega(n \cdot 2^{\frac{1}{(s-1)!} \alpha(n)^{s-1} + C'_{2s}(n)}). \tag{6.38}$$

In the expressions above $C_r(n)$ and $C'_r(n)$ are terms that are smaller than their leading exponents. The $\alpha(n)$ term is the inverse Ackerman function, which is an extremely slow-growing function that appears frequently in algorithms. The *Ackerman function* is defined as follows. Let $A_1(m) = 2m$ and $A_{n+1}(m)$ represent m applications of A_n . Thus, $A_1(m)$ performs doubling, $A_2(m)$ performs exponentiation, and $A_3(m)$ performs *tower exponentiation*, which makes a stack of 2's,

$$2^{2^{\vdots^2}}, \tag{6.39}$$

which has height m . The *Ackerman function* is defined as $A(n) = A_n(n)$. This function grows so fast that $A(4)$ is already an exponential tower of 2's that has height 65536. Thus, the *inverse Ackerman function*, α , grows very slowly. If n is less than or equal to an exponential tower of 65536 2's, then $\alpha(n) \leq 4$. Even when it appears in exponents of the Davenport-Schinzel bounds, it does not represent a significant growth rate.

⁹The following asymptotic notion is used: $O(f(n))$ denotes an upper bound, $\Omega(f(n))$ denotes a lower bound, and $\Theta(f(n))$ means that the bound is tight (both upper and lower). This notation is used in most algorithms books [176].

Example 6.5.1 (Lower envelope of line segments) One interesting application of Davenport-Schinzel applications is to the lower envelope of a set of line segments in \mathbb{R}^2 . Because segments in general position can intersect in at most one place, the number of edges in the lower envelope is $\Theta(\lambda_3(n)) = \Theta(n\alpha(n))$. There are actually arrangements of segments in \mathbb{R}^2 that reach this bound; see [687]. ■

6.5.3 Upper Bounds

The upper bounds for motion planning problems arise from the existence of complete algorithms that solve them. This section proceeds by starting with the most general bounds, which are based on the methods of Section 6.4, and concludes with bounds for simpler motion planning problems.

General algorithms The first upper bound for the general motion planning problem of Formulation 4.3.1 came from the application of cylindrical algebraic decomposition [676]. Let n be the dimension of \mathcal{C} . Let m be the number of polynomials in \mathcal{F} , which are used to define \mathcal{C}_{obs} . Recall from Section 4.3.3 how quickly this grows for simple examples. Let d be the maximum degree among the polynomials in \mathcal{F} . The maximum degree of the resulting polynomials is bounded by $O(d^{2^{n-1}})$, and the total number of polynomials is bounded by $O((md)^{3^{n-1}})$. The total running time required to use cylindrical algebraic decomposition for motion planning is bounded by $(md)^{O(1)^n}$.¹⁰ Note that the algorithm is doubly-exponential in dimension, n , but polynomial in m and d . It can theoretically be declared to be *efficient* on a space of motion planning problems of bounded dimension (although, it certainly is not efficient for motion planning in any practical sense).

Since the general problem is PSPACE-complete, it appears unavoidable that a complete, general motion planning algorithm will require a running time that is exponential in dimension. Since cylindrical algebraic decomposition is doubly-exponential, it led many in the 1980s to wonder whether whether this upper bound can be lowered. This was achieved by Canny’s roadmap algorithm, for which the running time is bounded by $m^n(\lg m)d^{O(n^4)}$. Hence, it is singly-exponential, which appears very close to optimal because it is up against the lower bound seems to be implied by PSPACE-hardness (and the fact that problems exist that require a roadmap with $(md)^n$ connected components [58]). Much of the algorithm complexity is due to finding a suitable deterministic perturbation to put the input polynomials into general position. A randomized algorithm can alternatively be used, for which the randomized expected running time is bounded by $m^n(\lg m)d^{O(n^2)}$. For a *randomized algorithm* [569], the *randomized expected* running time is still a worst-case upper bound, but averaged over random “coin

¹⁰It may seem odd for $O(\cdot)$ to appear in the middle of an expression. In this context, it means that there exists some $c \in [0, \infty)$ such that the running time is bounded by $(md)^{c^n}$.

tosses” that are introduced internally in the algorithm; it does *not* reflect any kind of average over the expected input distribution. Thus, these two bounds represent the best known upper bounds for the general motion planning problem. Canny’s algorithm may also be applied to solve the kinematic closure problems of Section 4.4, but the complexity does not reflect the fact that the dimension, k , of the algebraic variety is less than n , the dimension of \mathcal{C} . A roadmap algorithm that is particularly suited for this problem is introduced in [57], and its running time is bounded by $m^{k+1}d^{O(n^2)}$. This serves as the best-known upper bound for the problems of Section 4.4.

Specialized algorithms Now upper bounds are summarized for some narrower problems, which are easier to solve than the general problem. All of the problems involve either two or three degrees of freedom. Therefore, we expect that the bounds are much lower than those for the general problem. In many cases, the Davenport-Schinzle sequences of Section 6.5.2 arise. Most of the bounds presented here are based on algorithms that are not practical to implement; they mainly serve to indicate the best asymptotic performance that can be obtained for a problem. Most of the bounds mentioned here are included in [686].

Consider the problem from Section 6.2, in which the robot translates in $\mathcal{W} = \mathbb{R}^2$ and \mathcal{C}_{obs} is polygonal. Suppose that \mathcal{A} is a convex polygon that has k edges, and \mathcal{O} is the union of m disjoint, convex polygons with disjoint interiors, and their total number of edges n . In this case, the boundary of \mathcal{C}_{free} (computed by Minkowski difference; see Section 4.3.2) will have at most $6m - 12$ nonreflex vertices (interior angle less than π), and $n + km$ reflex vertices (interior angle greater than π). The free space, \mathcal{C}_{free} can be decomposed and searched in time $O((n + km) \lg^2 n)$ [389, 686]. Using randomized algorithms, the bound reduces to $O((n + km) \cdot 2^{\alpha(n)} \lg n)$ randomized expected time. Now suppose that \mathcal{A} is a single nonconvex polygonal region described by k edges, and that \mathcal{O} is a similar polygonal region described by n edges. The Minkowski difference could yield $\Theta(k^2 n^2)$ edges for \mathcal{C}_{obs} . This can be avoided if the search is performed within a single connected component of \mathcal{C}_{free} . Based on analysis that uses Davenport-Schinzle sequences, it can be shown that the worst connected component may have complexity $\Theta(kn\alpha(k))$, and the planning problem can be solved in time $O(kn \lg^2 n)$ deterministically, or for a randomized algorithm, $O(kn \cdot 2^{\alpha(n)} \lg n)$ randomized expected time is needed. More generally, if \mathcal{C}_{obs} consists of n algebraic curves in \mathbb{R}^2 , each with degree no more than d , then the motion planning problem for translation only can be solved deterministically in time $O(\lambda_{s+2}(n) \lg^2 n)$, or with a randomized algorithm in $O(\lambda_{s+2}(n) \lg n)$ randomized expected time. In these expressions, $\lambda_{s+2}(n)$ is the bound (6.37) obtained from the $(n, s + 2)$ Davenport-Schinzle sequence, and $s \leq d^2$.

For the case of the line-segment robot of Section 6.3.4 in an obstacle region described with n edges, an $O(n^5)$ -time algorithm was given. This is not the best possible running time for solving the line-segment problem, but the method is

easier to understand than others that are more efficient. In [589], a roadmap algorithm based on retraction is given that solves the problem in $O(n^2 \lg n \lg^* n)$ time, in which $\lg^* n$ is the number of times that \lg has to be iterated on n to yield 1 (i.e., it is a very small, insignificant term; for practical purposes, you can imagine that the running time is $O(n^2 \lg n)$). The tightest known upper bound is $O(n^2 \lg n)$ [480]. It is established in [388] that there exist examples for which the solution path requires $\Omega(n^2)$ length to encode.

Now consider the case for which $\mathcal{C} = SE(2)$, and \mathcal{A} is a convex polygon with k edges and \mathcal{O} is a polygonal region described by n edges. The boundary of \mathcal{C}_{free} has no more than $O(kn\lambda_6(kn))$ edges, and can be computed to solve the motion planning problem in $O(kn\lambda_6(kn) \lg kn)$ [5]. An algorithm that runs in time $O(k^4 n \lambda_3(n) \lg n)$ and provides better clearance between the robot and obstacles is given in [151]. In [33] (some details also appear in [437]), an algorithm is presented, and even implemented, that solves the problem in time $O(k^3 n^3 \lg(kn))$, for the more general case in which \mathcal{A} is nonconvex. The number of faces of \mathcal{C}_{obs} could be as high as $\Omega(k^3 n^3)$ for this problem. By explicitly representing and searching only one connected component, the best-known upper bound for the problem is $O((kn)^{2+\epsilon})$, in which $\epsilon > 0$ may be chosen arbitrarily small [310].

In the final case, suppose that \mathcal{A} translates in $\mathcal{W} = \mathbb{R}^3$ to yield $\mathcal{C} = \mathbb{R}^3$. For a polyhedron or polyhedral region, let its *complexity* be the total number of faces, edges, and vertices. If \mathcal{A} is a polyhedron with complexity k , and \mathcal{O} is a polyhedral region with complexity n , then the boundary of \mathcal{C}_{free} is polyhedral surface that has of complexity $\Theta(k^3 n^3)$. As for other problems, if the search is restricted to a single component, then the complexity reduces. The motion planning problem in this case can be solved in time $O((kn)^{2+\epsilon})$ [25]. If \mathcal{A} is convex, and there are m convex obstacles, then the best-known bound is $O(kmn \lg^2 m)$ time. If more generally, \mathcal{C}_{obs} is bounded by n algebraic patches of constant maximum degree, then a vertical decomposition method can be used to solve the motion planning problem within a single connected component of \mathcal{C}_{free} in time $O(n^{2+\epsilon})$.

Literature

A nice collection of early papers appears in [679]; this includes [589, 590, 651, 675, 676, 677].

A excellent reference for material in combinatorial algorithms, computational geometry, and complete algorithms for motion planning is the collection of survey papers in [292].

If you need more algebra background, try reading [178] and [611] before trying to tackle books such as [58] and [351].

Say why we did not follow Latombe's naming of roadmap vs. cell decomp. Since all cell decomposition methods produce a roadmap, they can be considered as a special class of roadmap algs.

Exercises

1. Extend the vertical decomposition algorithm to correctly handle the case in which \mathcal{C}_{obs} has two or more points that lie on the same vertical line. This includes the case of vertical segments. Random perturbations are not allowed.
2. Describe in detail how to use the vertical decomposition and line-sweep idea to compute \mathcal{C}_{obs} in $O(n \lg n)$ time.
3. Propose a complete motion planning algorithm for a polygonal \mathcal{C}_{obs} based on decomposing \mathcal{C}_{obs} into triangles. What is the running time of your algorithm?
4. Explain how to use the plane sweep idea to efficiently merge two nonconvex polygons.
5. Extend vertical decomposition to work for circular arcs and line segments.
6. Extend the bitangent graph algorithm to work for obstacle boundaries that are either pieces of circular arcs or line segments.
7. Derive the Conchoid of Nicomedes equation for the segment robot.
8. Make a resolution-complete version of the slicing/dscretizing method for the line segment robot.
9. Determine the cells for a line segment robot example.
10. Make some 1D decompositions to determine truth for a Tarski sentence with no free variables. Maybe a 2D example?
11. Construct a cad for \mathbb{S}^1 , \mathbb{S}^2 , \mathbb{S}^3 . (Give them cell numbers for the first two.)

12. Show using the matrix (6.28) that the Canny's roadmap for the torus, shown in Figure 6.38, is correct. (need to give torus equation)
13. A semester project idea is to implement Canny's algorithm. (please tell me if you succeed)

Chapter 7

Extensions of Basic Motion Planning

Chapter Status



What does this mean? Check <http://msl.cs.uiuc.edu/planning/status.html> for information on the latest version.

This chapter presents many extensions and variations of the motion planning problem considered in Chapters 3 to 6. Each one of these can be considered as a “spin-off” that is fairly straightforward to describe using the mathematical concepts and algorithms introduced so far. Unlike previous chapters, there is not much continuity in Chapter 7. Each problem is treated independently; therefore, it is safe to jump to whatever sections in the chapter you find interesting without fear of missing important details.

In many places throughout the chapter, a state space, X will arise. This is consistent with the general planning notation used throughout the book. In Chapter 4, the configuration space, \mathcal{C} , was introduced, which can be considered as a special state space: it encodes the set of transformations that can be applied to a collection of bodies. Hence, Chapters 5 and 6 addressed planning in $X = \mathcal{C}$. The configuration space alone will be insufficient for many of the problems in this chapter; therefore, X will be used because it appears to be more general. For most cases in this chapter, however, X is derived from one or more configuration spaces. Thus, configuration space and state space terminology will be used in combination.

7.1 Time-Varying Problems

This section brings time into the motion planning formulation. Although the robot has been allowed to move, it has been assumed so far that the obstacle region, \mathcal{O} , and the goal configuration, $q_g \in \mathcal{C}_{free}$ are stationary for all time. It is now assumed that these entities may vary over time, although their motions are predictable. If the motions are not predictable, then some form of feedback is needed to respond to observations that are made during execution. Such problems are much more difficult, and will be handled in Chapters 8, 10, and elsewhere throughout the book. The current formulation is designed to allow the tools and concepts learned so far to be directly applied to generate path.

7.1.1 Problem Formulation

Let $T \subset \mathbb{R}$ denote the *time interval*, which may be *bounded* or *unbounded*. If T is bounded, then $T = [0, t_f]$, in which 0 is the *initial time*, and t_f is the *final time*. If T is unbounded, then $T = [0, \infty)$. An initial time other than 0 could alternatively be defined without difficulty, but this will not be done here.

Let the state space, X be defined as $X = \mathcal{C} \times T$, in which \mathcal{C} is the usual configuration space of the robot, as defined in Chapter 4. A state, x , can be represented as $x = (q, t)$, to indicate the configuration, q , and time, t , components of the state vector. The planning will occur directly in X , and in many ways it can be treated as any configuration space seen to far, but there is one critical difference: *time marches forward*. Imagine a path that travels through X . If it first reaches a state $(q_1, 5)$, and then later some state $(q_2, 3)$, then traveling backwards though time is required! There is no mathematical problem with allowing such time travel, but it is not realistic for most applications. Therefore, paths in X will be forced to follow a constraint that they must move forward in time. Such a constraint can be considered *nonholonomic* because it restricts the way the states can flow through X ; this notion will be formally considered in much greater generality in Chapter 14.

Now consider making time-varying versions of the items used in Formulation 4.3.1 for motion planning:

Formulation 7.1.1 (The Time-Varying Motion Planning Problem)

1. A *world*, \mathcal{W} , is defined, in which either $\mathcal{W} = \mathbb{R}^2$ or $\mathcal{W} = \mathbb{R}^3$. This is the same as in Formulation 4.3.1.
2. A *time interval*, $T \subset \mathbb{R}$, is defined, which is either *bounded* to yield $T = [0, t_f]$ for some *final time*, $t_f > 0$, or *unbounded* to yield $T = [0, \infty)$.
3. A semi-algebraic, time-varying *obstacle region* $\mathcal{O}(t) \subset \mathcal{W}$ is defined for every $t \in T$. It is assumed that the obstacle region is a finite collection of rigid

bodies that undergoes continuous, time-dependent rigid body transformations.

4. The *robot*, \mathcal{A} (or $\mathcal{A}_1, \dots, \mathcal{A}_m$ for a linkage), and *configuration space*, \mathcal{C} , definitions are the same as in Formulation 4.3.1.
5. The *state space*, X , is defined as the Cartesian product, $X = \mathcal{C} \times T$, and a state, $x \in X$ may be denoted as $x = (q, t)$ to denote the configuration, q , and time, t components. See Figure 7.1. The obstacle region, X_{obs} , in state space is defined as

$$X_{obs} = \{(q, t) \in X \mid \mathcal{A}(q) \cap \mathcal{O}(t) \neq \emptyset\}, \quad (7.1)$$

and $X_{free} = X \setminus X_{obs}$. For a given $t \in T$, slices of X_{obs} and X_{free} are obtained. These are denoted as $\mathcal{C}_{obs}(t)$ and $\mathcal{C}_{free}(t)$, respectively, in which (if \mathcal{A} is one body)

$$\mathcal{C}_{obs}(t) = \{q \in \mathcal{C} \mid \mathcal{A}(q) \cap \mathcal{O}(t) \neq \emptyset\}, \quad (7.2)$$

and $\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}$.

6. A state $x_i \in X_{free}$ is designated as the *initial state*, with the constraint that $x_i = (q_i, 0)$ for some $q_i \in \mathcal{C}_{free}(0)$. In other words, at the initial time the robot cannot be in collision.
7. A subset $X_g \subset X_{free}$ is designated as the *goal region*. A typical definition is to pick some $q_g \in \mathcal{C}$ and let $X_g = \{(q_g, t) \in X_{free} \mid t \in T\}$, which means that the goal is *stationary* for all time.
8. A complete algorithm must compute a continuous, time-monotonic *path*, $\tau[0, 1] \rightarrow X_{free}$ such that $\tau(0) = x_i$ and $\tau(1) \in X_g$, or correctly report that such a path does not exist. To be *time monotonic*, we require that $t_1 < t_2$, which are obtained from $(q_1, t_1) = \tau(s_1)$ and $(q_2, t_2) = \tau(s_2)$, for any $s_1, s_2 \in [0, 1]$ such that $s_1 < s_2$.

Example 7.1.1 Figure 7.1 shows an example for a convex, polygonal robot, \mathcal{A} that translates in $\mathcal{W} = \mathbb{R}^2$. There is a single, convex, polygonal obstacle, \mathcal{O} . The two of these together yield a convex, polygonal configuration space obstacle, $\mathcal{C}_{obs}(t)$, which is shown for times t_1, t_2 , and t_3 . The obstacle moves with a *piecewise-linear motion model*, which means that transformations applied to \mathcal{O} are a piecewise-linear function of time. For example, let (x, y) be a fixed point on the obstacle. To be a linear motion model, this point must transform as $(x + c_1 t, y + c_2 t)$ for some constants $c_1, c_2 \in \mathbb{R}$. To be piecewise linear, it may change to a different linear motion at a finite number of critical times. Between these critical times, the motion must remain linear. There are two critical times in the example. If $\mathcal{C}_{obs}(t)$ is polygonal, and a piecewise-linear motion model is used, then X_{obs} will

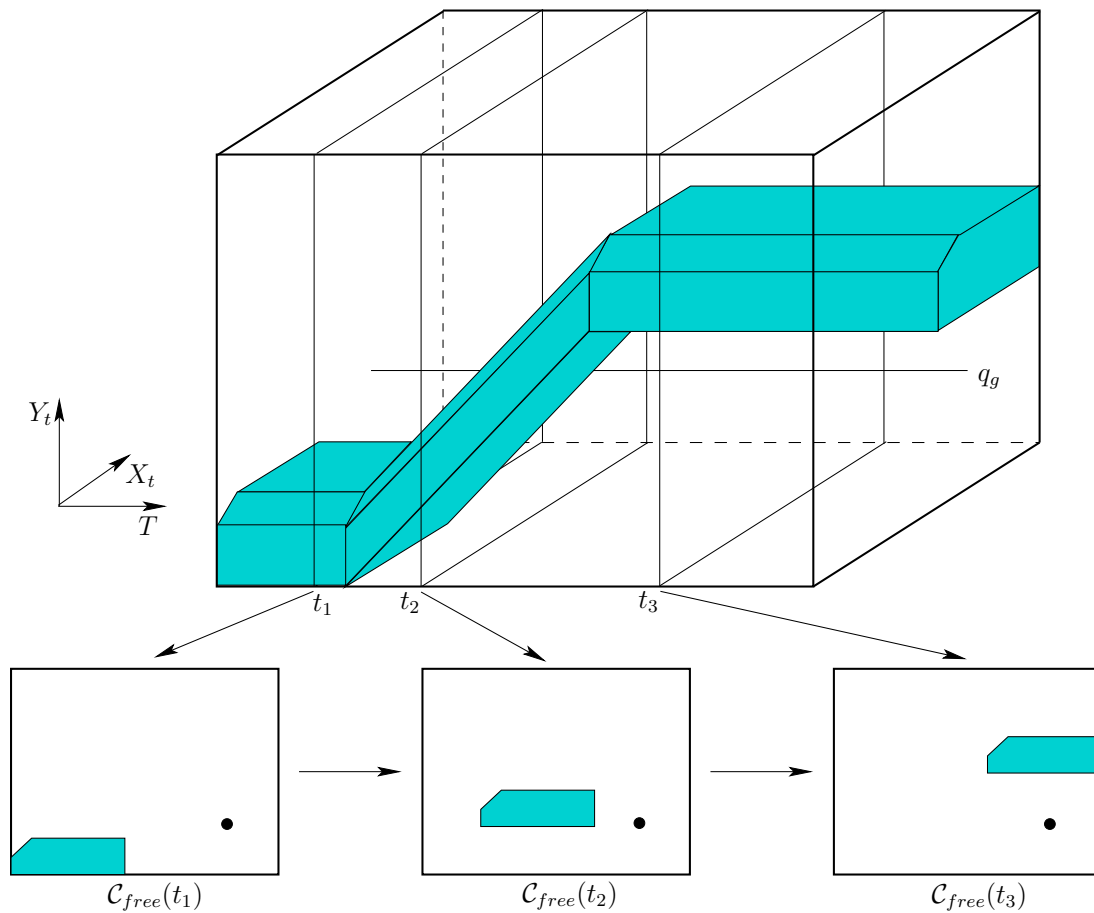


Figure 7.1: A time-varying example with linear obstacle motion.

be polyhedral, which is depicted in Figure 7.1. A stationary goal is also shown, which appears as a line that is parallel to the T axis. ■

In the general formulation, there are no additional constraints on τ , which means that the robot motion model allows infinite acceleration and unbounded speed. The robot velocity may change instantaneously, but the path through \mathcal{C} must always be continuous. These issues did not arise in Chapter 4 because there was no need to mention time. Now it becomes necessary.¹

¹The infinite acceleration and unbounded speed assumptions may annoy those with mechanics and control background. In this case, assume that the present models approximate the case in which every body moves slowly, and the dynamics can be consequently neglected. If this is still not satisfying, then jump ahead to Chapters 13 to 15, where general nonlinear systems are considered. It is still helpful to consider the implications derived from the concepts in this chapter because the issues remain for more complicated problems that involve dynamics.

7.1.2 Direct Solutions

Sampling-based methods Many sampling-based methods can be adapted from \mathcal{C} to X without much difficulty. The time-dependency of obstacle models must be taken into account when verifying that path segments are collision free; the techniques from Section 5.3.4 and be extended to handle this. One important concern is the metric for X . For some algorithms, it may be important to be use a pseudometric because symmetry is broken by time (going back in time is not as easy as going forward).

For example, suppose that the configuration space, \mathcal{C} is a metric space, (\mathcal{C}, ρ) . This metric can be extended across time to obtain a pseudometric, ρ_X as follows. For a pair of states, $x = (q, t)$ and $x' = (q', t')$, let

$$\rho_X(x, x') = \begin{cases} 0 & \text{if } q = q' \\ \infty & \text{if } q \neq q' \text{ and } t' \leq t \\ \rho(q, q') & \text{otherwise} \end{cases} . \quad (7.3)$$

Using ρ_X , several sampling-based methods will naturally work. For example, the rapidly-exploring dense trees from Section 5.5 can be adapted to X . Using ρ_X for a single-tree approach will ensure that all path segments travel forward in time. Using bidirectional approaches is more difficult for time-varying problems, because X_g is usually not a single point. It is not clear which (q, t) should be the starting vertex for the tree from the goal. The sampling-based roadmap methods of Section 5.6 are perhaps the most straightforward to adapt. The notion of a *directed roadmap* is needed, in which every edge must be directed to yield a time-monotonic path. For each pair of states, (q, t) and (q', t') , such that $t \neq t'$, exactly one valid direction exists for making a potential edge. If $t = t'$, then no edge can be attempted because it would require the robot to instantaneously “teleport” from one part of \mathcal{W} to another. Because forward time progress is already taken into account by the directed edges, a symmetric metric may be preferable instead of (7.3) for the sampling-based roadmap approach.

Combinatorial methods In some cases, combinatorial methods can be used to solve time-varying problems. If the motion model is *algebraic* (i.e., expressed with polynomials) then X_{obs} will be semi-algebraic. This enables the possibility of applying the general planners from Section 6.4, which are based on computational real algebraic geometry. The key issue once again is that the resulting roadmap must be directed with all edges being time monotonic. For Canny’s method, this requirement seems difficult to ensure. Cylindrical algebraic decomposition is straightforward to adapt if T is chosen as the last variable to be considered in the sequence of projections. This will yield polynomials in $\mathbb{Q}[t]$, and \mathbb{R} will be nicely partitioned into time intervals and time instances. Connections can then be made for a cell of one cylinder to an adjacent cell of a cylinder that occurs later in time.

If X_{obs} is polyhedral as depicted in Figure 7.1, then vertical decomposition can be used. It is best to first sweep the plane along the T axis, stopping at the critical

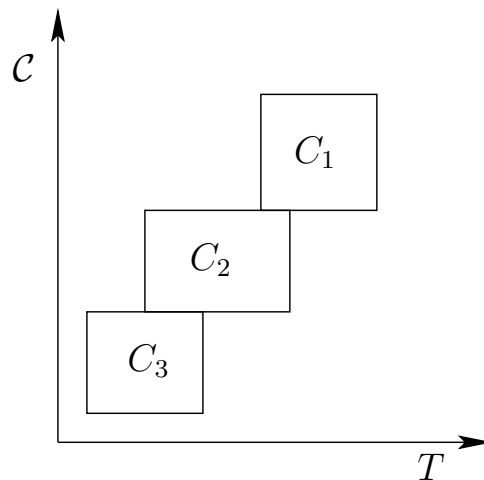


Figure 7.2: Transitivity is broken if the cells are not formed in cylinders over T . A time-monotonic path exists from C_1 to C_2 , and from C_2 to C_3 , but this does not imply that one exists from C_1 to C_3 .

times when the linear motion changes. This will yield nice sections which are further decomposed recursively, as explained in Section 6.3.3, and also facilitates the connection of adjacent cells to obtain time monotonic path segments. It is not too difficult to imagine the approach working for a four-dimensional state space, X , for which $\mathcal{C}_{obs}(t)$ is polyhedral as in Section 6.3.3, and time adds the fourth dimension. Again, performing the first sweep with respect to the T axis is preferable.

If X is not decomposed into cylindrical slices over each noncritical time interval, then cell decompositions may still be used, but one has to be more concerned about correctly connecting cells. Figure 7.2 illustrates the problem, for which transitivity among adjacent cells is broken. This complicates sample point selection for the cells.

Bounded speed There has been no consideration so far of the speed at which the robot must move to avoid obstacles. It is obviously impractical in many applications if the solution requires the robot to move arbitrarily fast. One step towards making a realistic model is to enforce a bound on the speed of the robot. (More steps towards realism are taken in Chapter 13.) For simplicity, suppose $\mathcal{C} = \mathbb{R}^2$, which corresponds to a translating rigid robot, \mathcal{A} , that moves in $\mathcal{W} = \mathbb{R}^2$. A configuration, $q \in \mathcal{C}$ can be represented as $q = (y, z)$ (since x already refers to a state vector). The *robot velocity* can be expressed as $v = (\dot{y}, \dot{z}) \in \mathbb{R}^2$, in which $\dot{y} = dy/dt$ and $\dot{z} = dz/dt$. The *robot speed* is $\|v\| = \sqrt{\dot{y}^2 + \dot{z}^2}$. A *speed bound*, b , is a positive constant, $b \in (0, \infty)$, for which $\|v\| \leq b$.

In terms of Figure 7.1 this means that the slope of a solution path τ must be constrained. Suppose that the domain of τ is $T = [0, t_f]$ instead of $[0, 1]$. This yields $\tau : T \rightarrow X$, and $\tau(t) = (y, z, t)$. Using this representation, $d\tau_1/dt = \dot{y}$ and

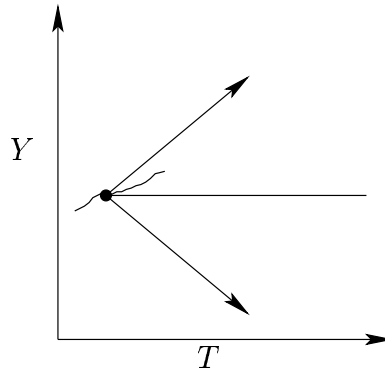


Figure 7.3: A projection of the cone constraint for the bounded speed problem.

$d\tau_2/dt = \dot{z}$, in which τ_i denotes the i^{th} component of i (because it is a vector-valued function). Thus, it can be seen that b constrains the slope of $\tau(t)$ in X . To visualize this, imagine that only motion in the Y direction occurs and suppose $b = 1$. If τ holds the robot fixed, then the speed is zero, which satisfies any bound. If the robot moves at speed 1, then $d\tau_1/dt = 1$ and $d\tau_2/dt = 0$, which satisfies the speed bound. In Figure 7.1 this generates a path that has slope 1 in the YT plane and is horizontal in the ZT plane. If both $d\tau_1/dt = d\tau_2/dt = 1$, then the bound is exceeded because the speed is $\sqrt{2}$. In general, the velocity vector at any state (y, z, t) points into a cone that starts at (y, z) and is aligned in the positive t direction; this is depicted in Figure 7.3. At time $t + \Delta t$, the state must stay within the cone, which means that

$$[y(t + \Delta t) - y(t)]^2 + [z(t + \Delta t) - z(t)]^2 \leq b^2(\Delta t)^2. \quad (7.4)$$

This constraint makes it considerably more difficult to adapt the algorithms of Chapters 5 and 6. Even for piecewise-linear motions of the obstacles, the problem has been established to be PSPACE-hard [652, 653, 731], for $\mathcal{W} = \mathbb{R}^2$. A complete algorithm that builds a kind of visibility graph is presented in [653]. The sampling-based roadmap of Section 5.6 is perhaps one of the easiest of the sampling-based algorithms to adapt for this problem. The neighbors of point q , which are determined for attempted connections, must lie within the cone that represents the speed bound. If this constraint is enforced, a dispersion-complete or probabilistically-complete planning algorithm results.

7.1.3 The Velocity Tuning Method

An alternative to defining the problem in $\mathcal{C} \times T$ is to decouple it into a *path planning* part and a *motion timing* part. Algorithms based on this method cannot be complete, but velocity tuning is an important idea that can be applied elsewhere. Suppose there are both *stationary* obstacles and *moving obstacles*. For the stationary obstacles, suppose that some path $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$ has been computed using any of the techniques in Chapters 5 and 6.

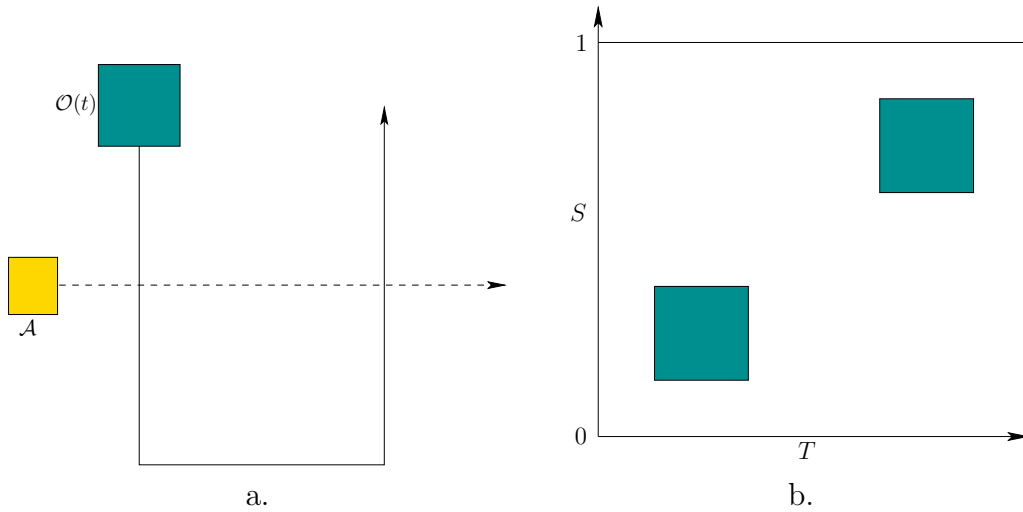


Figure 7.4: An illustration of path tuning: a) If the robot follows its computed path, it may collide with the moving obstacle; b) the resulting state space.

The timing part is then handled in a second phase. This amounts to designing a timing function, $\sigma : T \rightarrow [0, 1]$ that indicates for time t , the location of the robot along the path, τ . This is achieved by defining the composition $\phi = \tau \circ \sigma$, which maps from T to \mathcal{C}_{free} via $[0, 1]$. Thus, $\phi : T \rightarrow \mathcal{C}_{free}$. The configuration at time $t \in T$ may be expressed as $\phi(t) = \tau(\sigma(t))$.

A two-dimensional state space can be defined as shown in Figure 7.4. The purpose is to convert the design of σ (and consequently ϕ) into a familiar planning problem. The robot must move along its path from $\tau(0)$ to $\tau(1)$, an obstacle \mathcal{O} moves along its path over the time interval T . Let $S = [0, 1]$ denote the domain of τ . A state space, $X = T \times S$ is shown, in which each point (t, s) means indicates the time $t \in T$, and the position along the path, $s \in [0, 1]$. See Figure 7.4.b. An obstacle region X_{obs} is defined as

$$X_{obs} = \{(t, s) \in X \mid \mathcal{A}(\tau(s)) \cap \mathcal{O}(t) \neq \emptyset\}. \quad (7.5)$$

Once again, X_{free} is defined as $X_{free} = X \setminus X_{obs}$. The task is to find a continuous path $g : [0, 1] \rightarrow X_{free}$. If g is time monotonic, then a position $s \in S$ is assigned for every time, $t \in T$. These assignments can be nicely organized into a function, $\sigma : T \rightarrow S$, from which ϕ is obtained by $\phi = \tau \circ \sigma$ to determine where the robot will be at each time. Being time monotonic in this context means that the path must always progress from left to right in Figure 7.4.b. It can, however, be nonmonotonic in the S direction. This corresponds to moving back and forth along τ , causing some configurations to be revisited.

Any of the methods described in Formulation 7.1.1 can be applied here. The dimension of X in this case is always two. Note that X_{obs} is polygonal if the paths taken by \mathcal{A} and \mathcal{O} are both piecewise linear, and both are polygonal regions. In

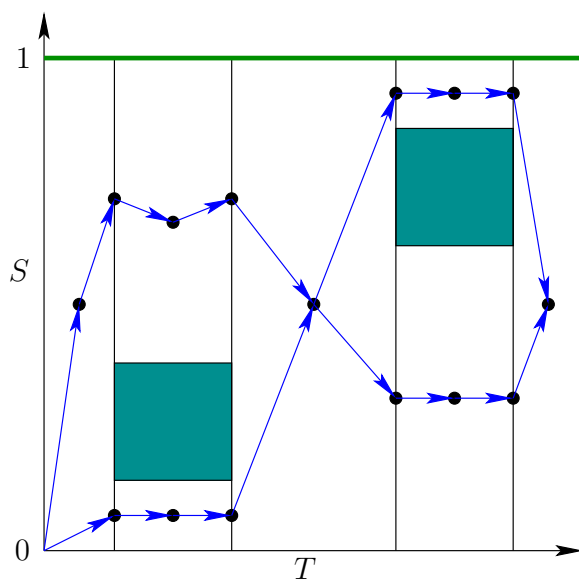


Figure 7.5: Vertical decomposition can solve the path tuning problem. Note that this example is not in general position because vertical edges exist. The goal is to reach the green line at the top, which can be accomplished from any adjacent 2-cell. For this example, it may even be accomplished from the first 2-cell if the robot is able to move quickly enough.

this case, the vertical decomposition method of Section 6.2.2 can be applied by sweeping along the time axis to yield a complete algorithm (after having committed to τ , but it is not complete for Formulation 7.1.1). The result is shown in Figure 7.5. The cells are connected only if it is possible to reach one from the other by traveling in the forward time direction. As an example of a sampling-based approach, which may be suitable when X_{obs} is not polygonal, is to place a grid over X and apply one of the classical search algorithms described in Section 5.4.2. Once again, only path segments in X that move forward in time are allowed.

7.2 Multiple Robots

This section supposes that there are multiple robots that share the same world, \mathcal{W} . A path must be computed for each one that avoids collisions with obstacles and with other robots. In Chapter 4, each robot could be a rigid body, \mathcal{A} , or be made of k attached bodies, $\mathcal{A}_1, \dots, \mathcal{A}_k$. To avoid confusion, superscripts will be used in this section to denote different robots. The i^{th} robot will be denoted by \mathcal{A}^i . Suppose there are m robots, $\mathcal{A}^1, \mathcal{A}^2, \dots, \mathcal{A}^m$. Each robot, \mathcal{A}^i , has its associated configuration space, \mathcal{C}^i , and its initial and goal configurations, q_{init}^i and q_{goal}^i .

7.2.1 Problem Formulation

A state space can be defined that considers the configurations of all of the robots simultaneously,

$$X = \mathcal{C}^1 \times \mathcal{C}^2 \times \dots \times \mathcal{C}^m. \quad (7.6)$$

A state $x \in X$ specifies all robot configurations, and may be expressed as $x = (q^1, q^2, \dots, q^m)$. Let N denote the dimension of X , which is given by $\sum_{i=1}^m \dim(\mathcal{C}^i)$.

There are two sources of obstacle regions in the state space: 1) *robot-obstacle* collisions, and 2) *robot-robot* collisions. For each i such that $1 \leq i \leq m$, the subset of X that corresponds to robot \mathcal{A}^i in collision with the obstacle region, \mathcal{O} , is defined as

$$X_{obs}^i = \{x \in X \mid \mathcal{A}^i(q^i) \cap \mathcal{O} \neq \emptyset\}. \quad (7.7)$$

This models the robot-obstacle collisions.

For each pair, \mathcal{A}^i and \mathcal{A}^j , of robots, the subset of X that corresponds to \mathcal{A}^i in collision with \mathcal{A}^j is given by

$$X_{obs}^{ij} = \{x \in X \mid \mathcal{A}^i(q^i) \cap \mathcal{A}^j(q^j) \neq \emptyset\}. \quad (7.8)$$

Both (7.7) and (7.8) will be combined in (7.10) to yield X_{obs} .

Formulation 7.2.1 (Multiple-Robot Motion Planning)

1. The *world*, \mathcal{W} and *obstacle region*, \mathcal{O} are the same as in Formulation 4.3.1.
2. There are m robots, $\mathcal{A}^1, \dots, \mathcal{A}^m$, which each may consist of one or more moving bodies.
3. Each robot, \mathcal{A}^i , for $1 \leq i \leq m$ has an associated *configuration space*, \mathcal{C}^i .
4. The *state space*, X , is defined as the Cartesian product

$$X = \mathcal{C}^1 \times \mathcal{C}^2 \times \dots \times \mathcal{C}^m. \quad (7.9)$$

The obstacle region in X is

$$X_{obs} = \left(\bigcup_{i=1}^m X_{obs}^i \right) \cup \left(\bigcup_{ij, i \neq j} X_{obs}^{ij} \right), \quad (7.10)$$

in which X_{obs}^i and X_{obs}^{ij} are the robot-obstacle and robot-robot collision states from (7.7) and (7.8), respectively.

5. A state $x_i \in X_{free}$ is designated as the *initial state*, in which $x_i = (q_i^1, \dots, q_i^m)$. For each i such that $1 \leq i \leq m$, q_i^i specifies the initial configuration of \mathcal{A}^i .
6. A subset $x_g \in X_{free}$ is designated as the *goal state*, in which $x_g = (q_g^1, \dots, q_g^m)$.
7. The task is to compute a continuous path, $\tau : [0, 1] \rightarrow X_{free}$ such that $\tau(0) = x_{init}$ and $\tau(1) \in x_{goal}$.

An ordinary motion planning problem? On the surface it may appear that there is nothing unusual about the multiple robot problem because the formulations used in Chapter 4 already cover the case in which the robot consists of multiple bodies. They do not have to be attached; therefore, X can be considered as an ordinary configuration space. The planning algorithms of Chapters 5 and 6 may be applied without adaptation. The main concern, however, is that the dimension of X grows linearly in the number of robots. For example, if there are 12 rigid bodies for which each has $\mathcal{C}^i = SE(3)$, then the dimension of X is $6 \cdot 12 = 72$. Complete algorithms require time that is at least exponential in dimension, which makes them unlikely candidates for such problems. Sampling-based algorithms are more likely to scale well in practice when there many robots, but the resulting dimension might still be too high.

Reasons to study multi-robot motion planning In spite of the fact multiple-robot motion planning can be handled like any other motion planning problem, there are several reasons to study it separately:

1. The motions of the robots may be decoupled in many interesting ways. These leads to several interesting methods that first develop some kind of partial plan for the robots independently, and then consider the plan interactions to produce a solution. This idea is referred to as *decoupled planning*.
2. The part of X_{obs} due to robot-robot collisions has a cylindrical structure, depicted in Figure 7.6, which can be exploited by planning algorithms to make them more efficient. Each X_{obs}^{ij} defined by (7.8) depends only on two robots. A point, $x = (q^1, \dots, q^N)$, is in X_{obs} if there exists i, j such that $1 \leq i, j \leq m$ such that $\mathcal{A}^i(q^i) \cap \mathcal{A}^j(q^j) \neq \emptyset$, regardless of the configurations of the other $m - 2$ robots. For some decoupled methods, this even implies that X_{obs} can be completely characterized by 2D projections, as depicted in Figure 7.10.
3. If optimality is important, then a unique set of issues arises for the case of multiple robots. It is not a standard optimization problem because the performance of each robot has to be optimized. There is no clear way to combine these objectives into a single optimization problem without losing some critical information. It will be explained in Section 7.7.2 that Pareto optimality naturally arises as the appropriate notion of optimality for multiple-robot motion planning.

Assembly Planning One important variant of multiple-robot motion planning is called *assembly planning* [132, 309, 330, 336, 403, 775, 776]. In automated manufacturing, many complicated objects are assembled step-by-step from individual parts. It is convenient for robots to manipulate the parts one-by-one to insert them into the proper locations (see Section 7.3.2). Imagine a collection of parts,

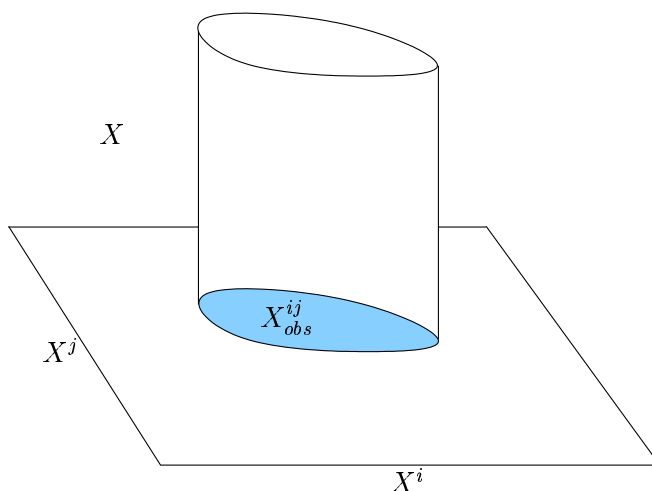


Figure 7.6: The set X_{ij} and its cylindrical structure on X .

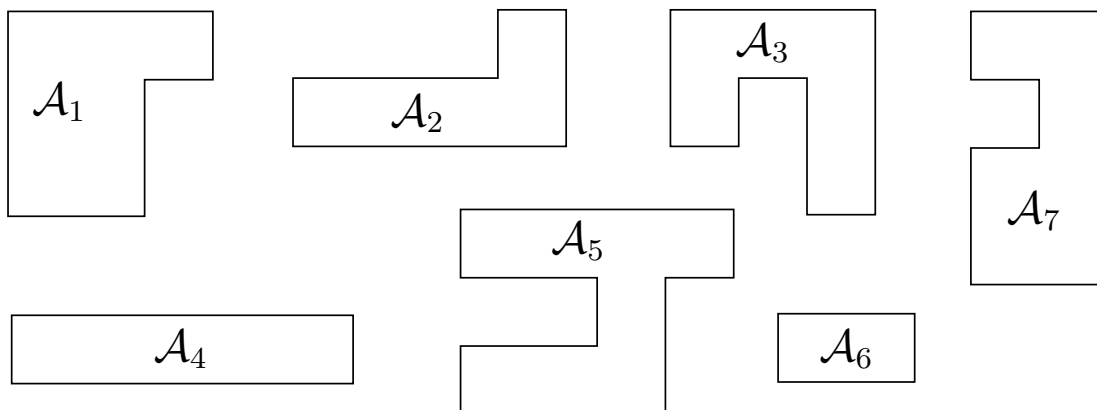


Figure 7.7: A collection of pieces used to define the assembly planning problem in Figure 7.8.

each of which is interpreted as a robot, as shown in Figure 7.7. The goal is to assemble the parts into one coherent object, such as that shown in Figure 7.8. The problem is generally approached by starting with the goal configuration, which is tightly constrained, and working outward. The problem formulation may allow that the parts touch, but their interiors cannot overlap. The assembly planning problem with arbitrarily many parts is NP-hard []. Interesting special cases have been considered. In one such case, for which parts can be removed by a sequence of straight-line paths, a polynomial-time algorithm is given in [775, 776].

7.2.2 Decoupled Planning

Decoupled approaches first design motions for the robots while ignoring robot-robot interactions. Once these interactions are considered, the choices available

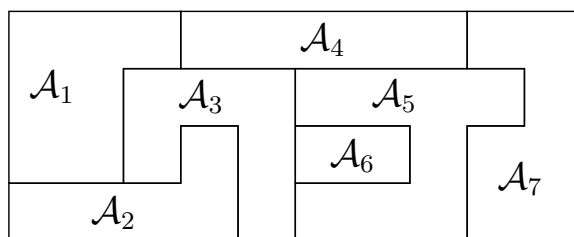


Figure 7.8: Assembly planning involves determining a sequence of motions that assembles the parts. The object shown here is assembled from the parts in Figure 7.8.

to each robot are already constrained by the designed motions. If a problem arises, these approaches are typically unable to reverse their commitments. Therefore, completeness is lost. Nevertheless, these approaches are quite practical, and in some cases completeness can be recovered.

Prioritized planning A straightforward approach to decoupled planning is to sort the robots by priority, and plan for higher-priority robots first [233]. Lower-priority robots plan by viewing the higher-priority robots as moving obstacles. Suppose the robots are sorted as $\mathcal{A}^1, \dots, \mathcal{A}^m$, in which \mathcal{A}^1 has the highest priority.

The prioritized planning approach proceeds inductively as follows:

Base case: Use any motion planning algorithm from Chapters 5 and 6 to compute a collision-free path, $\tau_1 : [0, 1] \rightarrow \mathcal{C}_{free}^i$ for \mathcal{A}^1 . Compute a timing function, σ_1 , for τ_1 , to yield $\phi_1 = \tau_1 \circ \sigma_1 : T \rightarrow \mathcal{C}_{free}^i$.

Inductive case: Suppose that $\phi_1, \dots, \phi_{i-1}$ have been designed for $\mathcal{A}^1, \dots, \mathcal{A}^{i-1}$, and that these timing functions avoid robot-robot collisions between any of the first $i - 1$ robots. Formulate the first $i - 1$ robots as moving obstacles in \mathcal{W} . For each $t \in T$ and $j \in \{1, \dots, i - 1\}$, the configuration, q^j of each \mathcal{A}^j is $\tau_j(\phi_j(t))$. This yields $\mathcal{A}^j(\tau_j(\phi_j(t))) \subset \mathcal{W}$, which can be considered as a subset of the obstacle $\mathcal{O}(t)$. Design a path, τ_i and timing function ϕ_i using any of the time-varying motion planning methods from Section 7.1.

A special case of prioritized planning would be to design all of the paths, $\tau_1, \tau_2, \dots, \tau_m$, in the first phase, then formulate each inductive step as a velocity tuning problem. This yields a sequence of 2D planning problems, which can be solved quite easily. This will come at a greater expense, however, because the choices are even more constrained. The idea of preplanning paths, and even roadmaps, for all robots independently can lead to a powerful solution if the coordination of the robots is approached more carefully. This is the next topic.

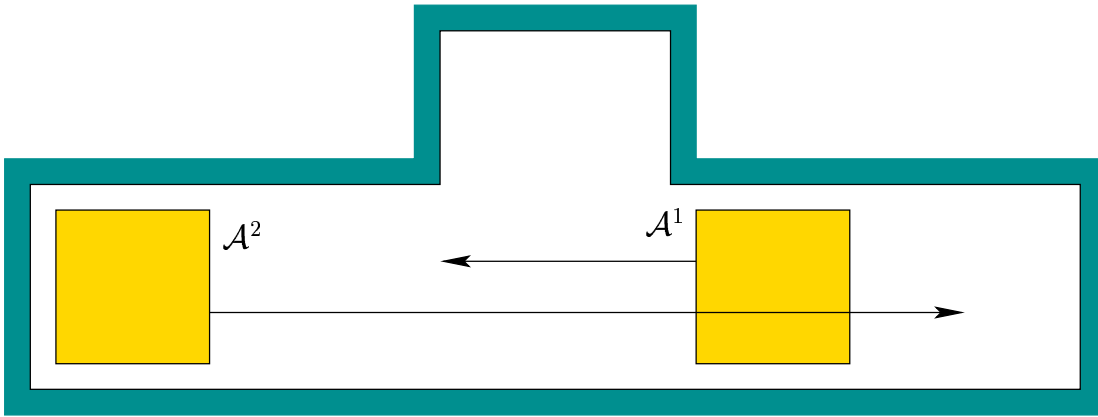


Figure 7.9: If \mathcal{A}^1 neglects the query for \mathcal{A}^2 , then completeness is lost when using the prioritized planning approach. This example has a solution in general, but prioritized planning fails to find it.

Fixed-path coordination Suppose that each robot, \mathcal{A}^i is constrained to follow a path $\tau_i : [0, 1] \rightarrow \mathcal{C}_{free}^i$, which can be computed using any ordinary motion planning technique. For m robots, an m -dimensional state space called a *coordination space* can be formed which schedules the motions of the robots along their paths so that they will not collide [587]. One interesting feature to note carefully is that time will only be *implicitly* represented in the coordination space. The task will be to compute a path in the coordination space, from which explicit timings can be easily extracted.

For m robots, the *coordination space*, X , is defined as the m -dimensional unit cube $X = [0, 1]^m$. Figure 7.10 depicts an example for which $m = 3$. The i^{th} coordinate of X represents the domain, $S_i = [0, 1]$, of the path τ_i . A state, $x \in X$, therefore indicates the configuration of every robot. For each i , the configuration $q^i \in \mathcal{C}^i$ is given by $q^i = \tau_i(x_i)$. At state $(0, \dots, 0) \in X$, every robot is in its initial configuration, $q_{init}^i = \tau_i(0)$, and at state $(1, \dots, 1) \in X$, every robot is in its goal configuration $q_{goal}^i = \tau_i(1)$. Any continuous path, $\sigma : [0, 1] \rightarrow X$, for which $\sigma(0) = (0, \dots, 0)$ and $\sigma(1) = (1, \dots, 1)$, will move the robots to their goal configurations. The path σ does not even need to be monotonic, in contrast to prioritized planning.

One important concern has been neglected so far. What prevents us from designing σ as a straight-line path between the opposite corners of $[0, 1]^m$? We have not yet taken into account the collisions between the robots. This forms an obstacle region, X_{obs} that must be avoided when designing a path through X . Thus, the task is to design $\sigma : [0, 1] \rightarrow X_{free}$, in which $X_{free} = X \setminus X_{obs}$.

The definition of X_{obs} is very similar to (7.8) and (7.10), except that here the state space dimension is much smaller. Each q^i is replaced by a single parameter. The cylindrical structure, however, is still retained, as shown in Figure 7.10. Each

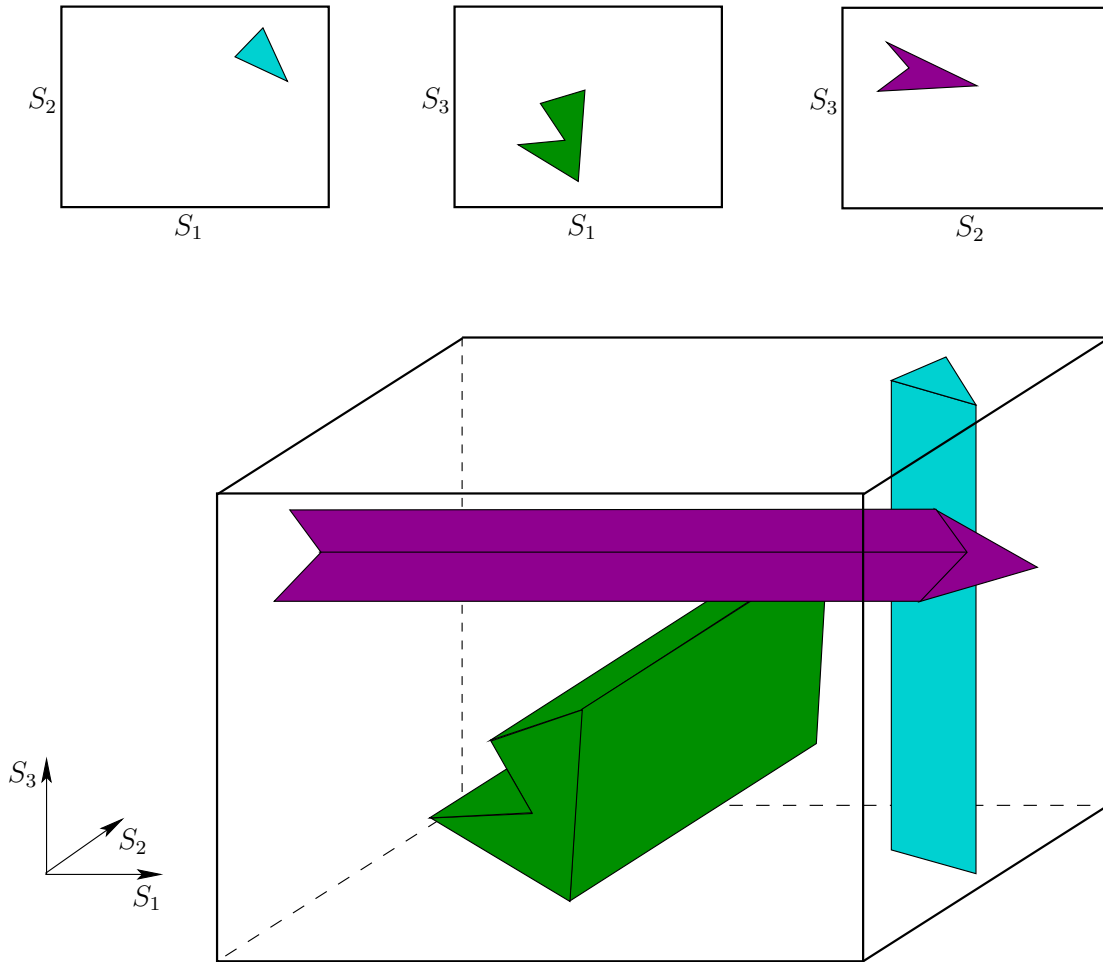


Figure 7.10: The obstacles that arise from coordinating m robots are always cylindrical. The set of all 2D projections completely characterizes X_{obs} .

cylinder of X_{obs} is given by

$$X_{obs}^{ij} = \{(s_1, \dots, s_m) \in X \mid \mathcal{A}^i(\tau_i(s_i)) \cap \mathcal{A}^j(\tau_j(s_j)) \neq \emptyset\}, \quad (7.11)$$

which are combined to yield

$$X_{obs} = \bigcup_{ij, i \neq j} X_{obs}^{ij}. \quad (7.12)$$

Standard motion planning algorithms can be applied to the coordination space because there is no monotonicity requirement on σ . If 1) $\mathcal{W} = \mathbb{R}^2$, 2) $m = 2$ (two robots), 3) the obstacles and robot are polygonal, and 4) the paths, τ_i , are piecewise linear, then X_{obs} will be a polygonal region in X . This enables the methods of Section 6.2, for a polygonal \mathcal{C}_{obs} , to directly apply after the representation of X_{obs} is explicitly constructed. For $m > 2$, the multidimensional version of vertical cell decomposition, given for $m = 3$ in Section 6.3.3, can be applied. For general

coordination problems, cylindrical algebraic decomposition or Canny's algorithm can be applied. For the problem of robots in $\mathcal{W} = \mathbb{R}^2$ that either translate or move along circular paths, a resolution-complete planning method based on exact determination of X_{obs} using special collision detection methods is given in [706].

For very challenging coordination problems, sampling-based solutions may yield practical solutions. Perhaps one of the simplest solutions is to place a grid over X and adapt the classical search algorithms, as described in Section 5.4.2 [461, 587]. Other possibilities include using RDTs of Section 5.5, or if the multiple-query framework is appropriate, then the sampling-based roadmap methods of 5.6 may be suitable. Methods for validating the path segments, which were covered in Section 5.3.4, can be adapted without trouble to the case of coordination spaces.

Thus far, the particular speeds of the robots have been neglected. For explanation purposes, consider the case of $m = 2$. Moving vertically or horizontally in X holds one robot fixed while the other moves at some maximum speed. Moving diagonally in X moves both robots, and the relative speeds depends on the slope of the path. To carefully regulate these speeds, it may be necessary to reparameterize the paths by distance. In this case each axis of X represents the distance traveled, instead of $[0, 1]$.

Fixed-roadmap coordination The fixed-path coordination approach still cannot solve the problem in Figure 7.9 if the paths are designed independently. Fortunately, fixed-path coordination can be extended to enable each robot to move over a roadmap other topological graph. This still yields a coordination space that has only one dimension per robot, and the resulting planning methods are much closer to being complete, assuming each robot utilizes a roadmap that has many alternative paths. There is also motivation to study this problem by itself because of autonomous guided vehicles (AGVs), which often move in factories on a network of predetermined paths []. In this case, coordinating the robots *is* the planning problem, as opposed to being a simplification of Formulation 7.2.1.

One way to obtain completeness for Formulation 7.2.1 is to design the independent roadmaps so that each robot has its own *garage* configuration. The conditions for a configuration, q^i , be a *garage* for \mathcal{A}^i are: 1) while at configuration q^i , it is impossible for any other robots to collide with it (i.e., for all coordination states for which the i^{th} coordinate is q^i , no collision occurs). 2) q^i is always reachable by \mathcal{A}^i from q_{init}^i , and its presence at q^i does not block other robots from reaching their garages. If each robot has a roadmap and a garage, and if the planning method for X is complete, then the overall planning algorithm is complete. If the planning method in X uses some weaker notion of completeness, then this is also maintained. For example, a resolution-complete sampling-based planner for X will be yield a resolution-complete approach to the problem in Formulation 7.2.1 (or to the problem of planning for an AGV).

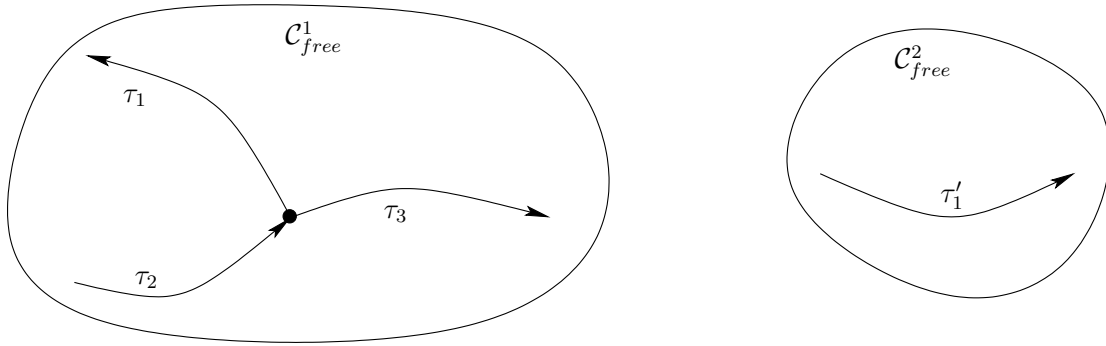


Figure 7.11: An example in which \mathcal{A}^1 moves along three paths, and \mathcal{A}^2 moves along one.

Cube complex How is the coordination space represented when there are multiple paths for each robot? It turns out that a *cube complex* is obtained, which is a special kind of singular complex (recall from Section 6.3.1). The coordination space for fixed paths can be considered as a singular m -simplex. For example, the problem in Figure 7.10, can be considered as a 3-simplex, $[0, 1]^3 \rightarrow X$. In Section 6.3.1 the domain of a k -simplex was defined using B^k , a k -dimensional ball; however, a cube, $[0, 1]^k$ will also work because B^k and $[0, 1]^k$ are homeomorphic.

For a topological space, X , k -cube (which is also a singular k -simplex), \square_k , is a continuous mapping $\sigma : [0, 1]^k \rightarrow X$. A cube complex is obtained by connecting together k -cubes of different dimensions. Every k -cube for $k \geq 1$ has $2k$ faces, which are $(k - 1)$ -cubes that are obtained as follows. Let (s_1, \dots, s_k) denote a point in $[0, 1]^k$. For each $i \in \{1, \dots, k\}$, one face is obtained by setting $s_i = 0$, and another is obtained by setting $s_i = 1$.

The cubes must fit together nicely, much in the same way that the simplexes of a simplicial complex were required to fit together. To be a *cube complex*, \mathcal{K} , be a collection of simplexes must satisfy these requirements:

1. Any face \square_{k-1} of a cube $\square_k \in \mathcal{K}$ is also in \mathcal{K} .
2. The intersection of the images of any two k -cubes $\square_k, \square'_k \in \mathcal{K}$ is either empty, or there exists some cube $\square_i \in \mathcal{K}$ for $i < k$, which is a common face of both \square_k and \square'_k .

Let G_i denote a topological graph (which may also be a roadmap) for robot \mathcal{A}^i . The graph can be designed by constructing paths of the form $\tau : [0, 1] \rightarrow \mathcal{C}_{free}^i$. Before covering formal definitions of the resulting complex, consider Figure 7.11, in which \mathcal{A}^1 moves along three paths connected in a “T” junction, and \mathcal{A}^2 moves along one path. In this case, there are three two-dimensional fixed-path coordination spaces, which are attached together along one common edge, as shown in Figure 7.12. The resulting cube complex is defined by three 2-cubes (i.e., squares), one 1-cube (i.e., line segment), and eight 0-cubes (i.e., corner points).

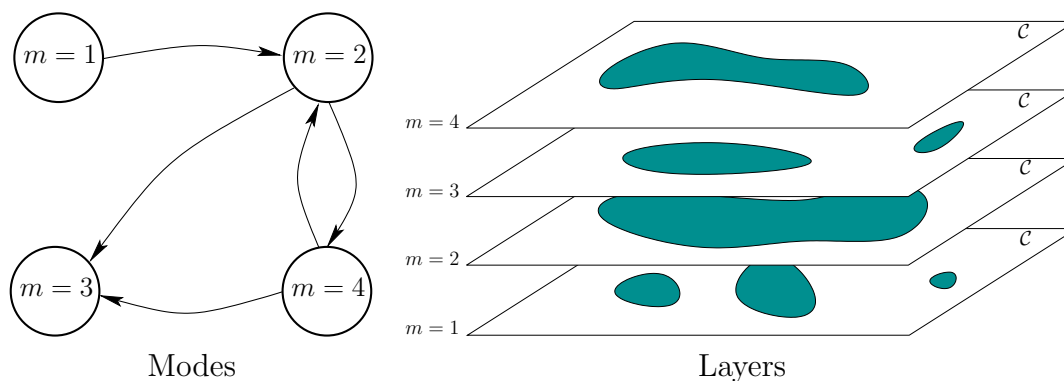


Figure 7.13: A hybrid state space can be imagined as having layers of configuration spaces which are indexed by modes.

7.3 Hybrid Systems: Discrete and Continuous

Many important applications involve a mixture of discrete and continuous variables. This results in a state space that is a Cartesian product of a finite set called the *mode space*, and a continuous set called the *configuration space*. The resulting *hybrid system* can be visualized as having layers of configurations spaces that are indexed by the modes, as depicted in Figure 7.13. The main application given in this section is manipulation planning; many others exist, especially when other complications such as dynamics and uncertainties are added to the problem. The framework of this section is inspired mainly from *hybrid systems* in the control theory community [93], which is usually models mode-dependent dynamics. The main concern in this section is that the allowable robot motions and/or the obstacles depend on the mode.

7.3.1 General Framework

As illustrated in Figure 7.13, a hybrid system involves interaction between discrete and continuous spaces. The formal model will be first be given, followed by some explanation. This formulation can be considered as a synthesis of the components from discrete feasible planning, Formulation 2.2.1, and basic motion planning, Formulation 4.3.1.

Formulation 7.3.1 (Hybrid System Motion Planning)

1. The \mathcal{W} and \mathcal{C} components from Formulation 4.3.1 are included.
2. A nonempty *mode space*, M , is defined which is a finite or countably infinite set of *modes*.
3. A semi-algebraic *obstacle region* $\mathcal{O}(m)$ is defined for each mode $m \in M$.

4. A semi-algebraic *robot*, $\mathcal{A}(m)$, is defined for each $m \in M$. It may be a rigid robot or a collection of links. It will be assumed here that the configuration space is not mode-dependent; only the geometry of the robot can depend on the mode. When the robot is transformed to configuration q , it will be denoted as $\mathcal{A}(q, m)$.
5. A *state space*, X , is defined as the Cartesian product $X = \mathcal{C} \times M$. A state may be represented as $x = (q, m)$, in which $q \in \mathcal{C}$ and $m \in M$. Let

$$X_{obs} = \{(q, e) \in X \mid \mathcal{A}(q, m) \cap \mathcal{O}(m) \neq \emptyset\}, \quad (7.13)$$

and $X_{free} = X \setminus X_{obs}$.

6. For each state, $x \in X$, a finite *action space*, $U(x)$. Let U denote the set of all possible actions (the union of $U(x)$ over all $x \in X$).
7. A *mode transition function*, f_m , which produces a mode, $f(x, u) \in M$, for every $x \in X$ and $u \in U(x)$. It is that f is defined in a way that does not produce race conditions. This means that if q is fixed, the mode can change at most once. It then remains constant, and may only change if q is changed.
8. A *state transition function*, f , which is derived from f_m by changing the mode and holding the configuration fixed. Thus, $f((q, m), u) = (q, f_m(q, m))$.
9. A configuration $x_i \in X_{free}$ is designated as the *initial state*.
10. A configuration $X_g \in X_{free}$ is designated as the *goal region*. A region is defined instead of a point to facilitate the specification of a goal configuration that does not depend on the final mode.
11. An algorithm must compute a (continuous) *path*, $\tau : [0, 1] \rightarrow X_{free}$ and *action function* $\sigma : [0, 1] \rightarrow U$ such that $\tau(0) = x_i$ and $\tau(1) \in X_g$, or correctly report that such a combination path and action function does not exist.

The obstacle region and robot may or may be mode-dependent, depending on the problem. Examples of each will be given shortly. Changes in the mode depend on the action taken by the robot. From most states, it is usually assumed that a “do nothing” action exists, which leaves the mode unchanged. From certain states, the robot may select an action that changes the mode as desired. An interesting degenerate case exists, in which there is only a single action available. This means that the robot has no control over the mode from that state. If the robot arrives in such states, a mode change could automatically occur.

The solution requirement is somewhat more complicated because both a path and action function need to be specified. It is insufficient to specify a path because it is important to know what action was applied to induce the correct mode transitions. Therefore, σ , is used to indicate when these occur. Note that τ and

σ are closely coupled; one cannot simply associate any σ with a path τ ; it must correspond to the actions required to generate τ .

Example 7.3.1 (The Power of the Portiernia) In this example, a robot, \mathcal{A} , is modeled as a square that translates in $\mathcal{W} = \mathbb{R}^2$. Therefore, $\mathcal{C} = \mathbb{R}^2$. The obstacle region in \mathcal{W} is mode-dependent because of two doors, which are numbered “1” and “2” in Figure 7.14.a. In the upper left sits the portiernia,² which is able to give a key to the robot, if the robot is in a configuration as shown in Figure 7.14.b. The portiernia only trusts the robot with one key at a time, which may be either for Door 1 or Door 2. The robot can return a key by revisiting the portiernia. As shown in Figures 7.14.c and 7.14.d, the robot can open a door by making contact with it, as long as it holds the correct key.

The set, M , of modes needs to encode which key, if any, the robot holds, and also it must encode the status of the doors. The robot may either have: 1) the key to Door 1; 2) the key to Door 2; or 3) no keys. The doors may have the status: 1) both open; 2) Door 1 open, Door 2 closed; 3) Door 1 closed, Door 2 open; or 4) both closed. Considering keys and doors in combination yields 12 possible modes.

If the robot is at a portiernia configuration as shown in Figure 7.14.b, then its available actions correspond to different ways to pick up and drop off keys. For example, if the robot is holding the key to Door 1, it can drop it off and pick up the key to Door 2. This changes the mode, but the door status and robot configuration must remain unchanged when f_m and f are applied. The other locations in which the robot may change the mode are when in contact with Door 1 or Door 2. The mode changes the mode only if the robot is holding the proper key. In all other configurations, the robot only has a single action (i.e., no choice), which keeps the mode fixed.

The task is to reach the configuration shown in the lower right with dashed lines. The problem is solved by: 1) picking up the key for Door 1 at the portiernia; 2) opening Door 1; 3) swapping the key at the portiernia to obtain the key for Door 2; 4) entering the innermost room to reach the goal configuration. As a final condition, we might want to require that the robot returns the key to the portiernia.

Example 7.3.1 allows the robot to change the obstacles in \mathcal{O} . The next example involves a robot that can change its shape. This is an illustrative example of a *reconfigurable robot*. The study of such robots has become a popular topic of research [154, 277, 410, 789]; the reconfiguration possibilities in that research area are much more sophisticated than the simple example considered here.

Example 7.3.2 (Reconfigurable Robot) To solve the problem shown in Figure 7.15, the robot must change its shape. There are two possible shapes, which correspond directly to the modes: *elongated* and *compressed*. Examples of each are shown in the figure. Figure 7.16 shows how $\mathcal{C}_{free}(m)$ appears for each of the

²These are groups of people who guard the keys at some public facilities in Poland.

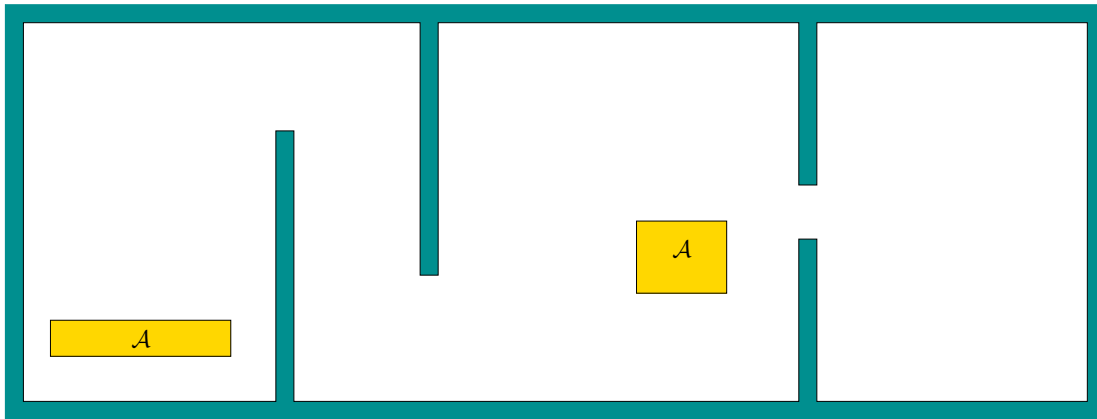


Figure 7.15: An example in which the robot must reconfigure itself to solve the problem. There are two modes: *elongated* and *compressed*.

two modes. Suppose the robot starts initially from the left while in the elongated mode, and must travel to the last room on the right. This problem must be solved by: 1) reconfiguring the robot into the compressed mode; 2) passing through the corridor into the center; 3) reconfiguring the robot into the elongated mode; 4) passing through the corridor to the rightmost room. The robot has actions that directly change the mode by reconfiguring itself. To make the problem more interesting, we could require that robot is only able to reconfigure itself in specific locations (e.g., where there is enough clearance, or possibly at a location where another robot can assist it).

The examples presented so far barely scratch the surface on the possible hybrid problems that can be defined. Many such problems can arise, for example, in the context making automated video game characters or digital actors. To solve these problems, standard motion planning algorithms can be adapted if they are given information about how to change the modes. Locations in X from which the mode can be changed may be expressed as subgoals. Much of the planning effort should then be focused on attempting to change modes, in addition to trying to directly reach the goal. Applying sampling-based methods requires the definition of a metric on X that accounts for both changes in the mode and the configuration. A wide variety of hybrid problems can be formulated, ranging from ones that are impossible to solve in practice to others that are straightforward extensions of standard motion planning. One particularly interesting class of problems, for which successful algorithms have been developed, will be covered next.

7.3.2 Manipulation Planning

This section presents an overview of manipulation planning; the concepts explained here are mainly due to [9, 10]. Returning to Example 7.3.1, imagine that the robot must carry a key that is so large that it changes the connectivity of

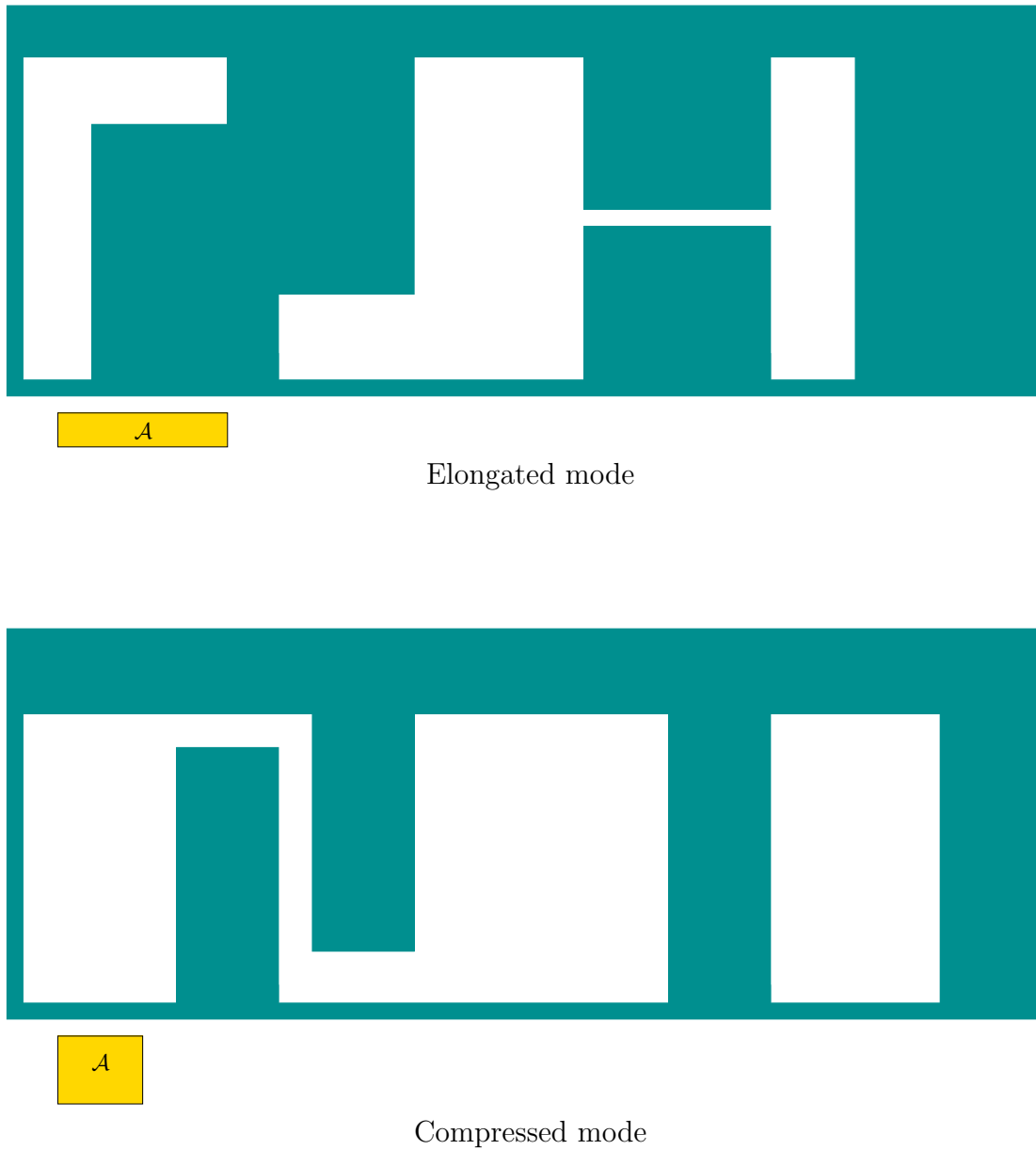


Figure 7.16: When the robot changes its configuration, $\mathcal{C}_{free}(m)$ changes, enabling it to solve the problem.

\mathcal{C}_{free} . For the manipulation planning problem, the robot will be called a *manipulator* that interacts with a *part*. In some configurations it is able to *grasp* the part and move it to other locations in the environment. The *manipulation task* usually requires moving the part to a specified location in \mathcal{W} , without particular regard to how the manipulator can accomplish the task. The model considered here greatly simplifies the problems of grasping, stability, friction, mechanics, and uncertainties, and instead focuses on the geometric aspects (some of these issues will be addressed in Sections ??). For a thorough introduction to these other important aspects of manipulation planning, see [536].

Admissible configurations Assume that following components from Formulation 4.3.1 are used here: \mathcal{W} , \mathcal{O} , and \mathcal{A} . For manipulator planning, \mathcal{A} will be called the *manipulator*, and let \mathcal{C}^a refer to the *manipulator configuration space*. Let \mathcal{P} denote a *part*, which is a rigid body modeled in terms of geometric primitives, as described in Section 3.1. It is assumed that \mathcal{P} is allowed to undergo rigid body transformations, and will therefore have its own *part configuration space*, $\mathcal{C}^p = SE(2)$ or $\mathcal{C}^p = SE(3)$. Let $q^p \in \mathcal{C}^p$ denote a *part configuration*. The transformed part model is denoted as $\mathcal{P}(q^p)$.

The combined *configuration space*, \mathcal{C} , is defined as the Cartesian product

$$\mathcal{C} = \mathcal{C}^a \times \mathcal{C}^p, \quad (7.14)$$

in which each configuration $q \in \mathcal{C}$ is of the form $q = (q^a, q^p)$. The first step is to remove all configurations that must be avoided. Parts of Figure 7.17 show examples of these sets. Configurations for which the manipulator collides with obstacles are

$$\mathcal{C}_{obs}^a = \{(q^a, q^p) \in \mathcal{C} \mid \mathcal{A}(q^a) \cap \mathcal{O} \neq \emptyset\}. \quad (7.15)$$

The next logical step is to remove configurations for which the part collides with obstacles. It will make sense to allow the part to “touch” the obstacles. For example, this could model a part sitting on a table. Therefore, let

$$\mathcal{C}_{obs}^p = \{(q^a, q^p) \in \mathcal{C} \mid \text{int}(\mathcal{P}(q^p)) \cap \mathcal{O} \neq \emptyset\}, \quad (7.16)$$

denote the open set for which the interior of the part intersects \mathcal{O} . Certainly if the part penetrates \mathcal{O} , the configuration should be avoided.

Consider $\mathcal{C} \setminus (\mathcal{C}_{obs}^a \cup \mathcal{C}_{obs}^p)$. The configurations that remain ensure that the robot and part do not inappropriately collide with \mathcal{O} . Next consider the interaction between \mathcal{A} and \mathcal{P} . The manipulator must be allowed to touch the part, but penetration will once again not be allowed. Therefore, let

$$\mathcal{C}_{obs}^{ap} = \{(q^a, q^p) \in \mathcal{C} \mid \mathcal{A}(q^a) \cap \text{int}(\mathcal{P}(q^p)) \neq \emptyset\}. \quad (7.17)$$

Removing all of these bad configurations yields

$$\mathcal{C}_{adm} = \mathcal{C} \setminus (\mathcal{C}_{obs}^a \cup \mathcal{C}_{obs}^p \cup \mathcal{C}_{obs}^{ap}), \quad (7.18)$$

which will be called the set of *admissible configurations*.

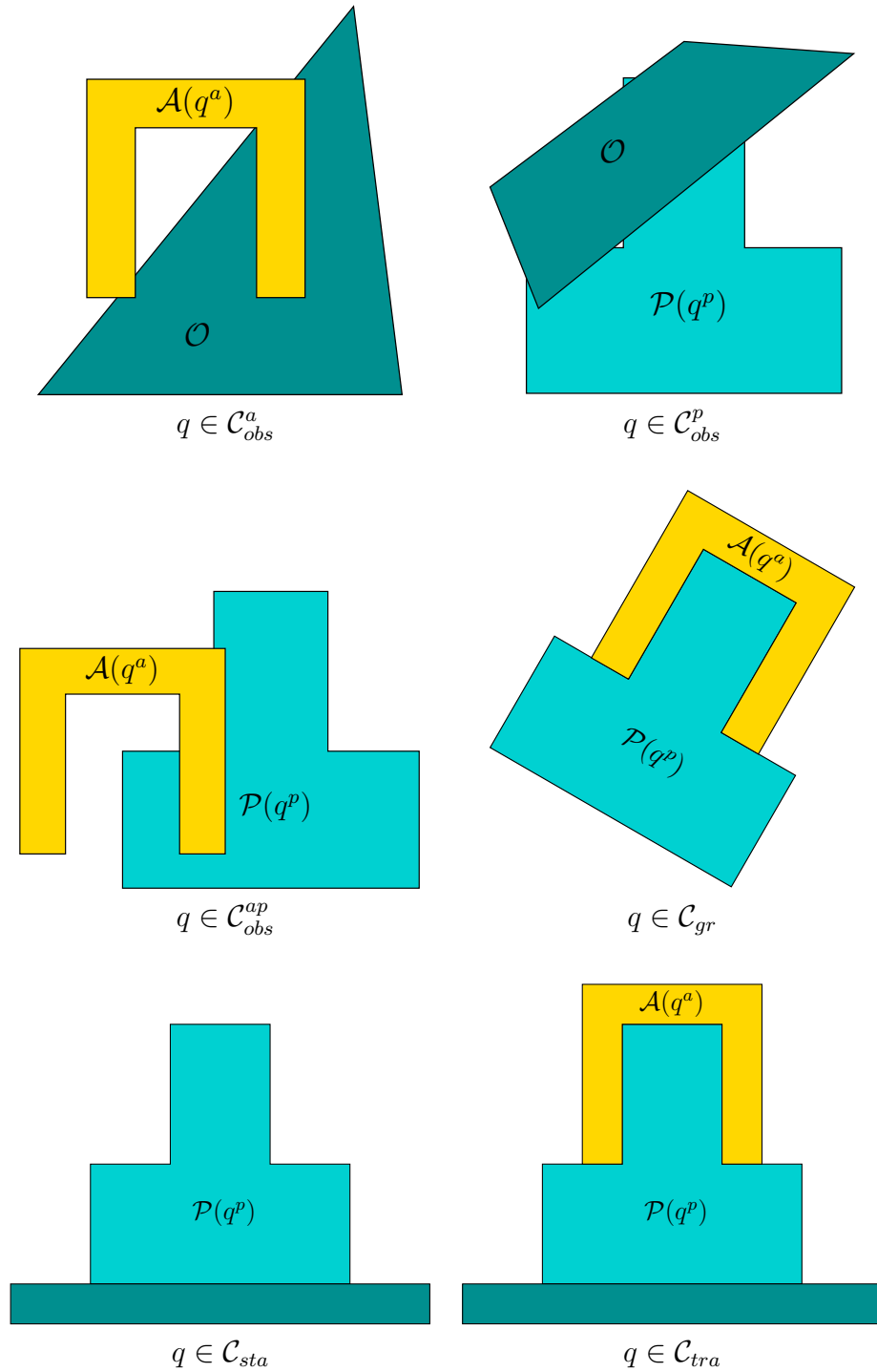


Figure 7.17: Examples of several important subsets of \mathcal{C} for manipulation planning.

Stable and grasped configurations Two important subsets of \mathcal{C}_{adm} will be used in the manipulation planning problem. See Figure 7.17. Let \mathcal{C}_{sta}^p denote the set of *stable part configurations*, which are configurations at which the part can safely rest without any forces being applied by the manipulator. This means that a part cannot, for example, float in the air. It also cannot be in a configuration from which it might fall. The particular stable configurations depend on properties such as the part geometry, friction, mass distribution, etc. These issues are not considered here. From this, let $\mathcal{C}_{sta} \subseteq \mathcal{C}_{adm}$ be the corresponding *stable configurations*, defined as

$$\mathcal{C}_{sta} = \{(q^a, q^p) \in \mathcal{C}_{adm} \mid q^p \in \mathcal{C}_{sta}^p\}. \quad (7.19)$$

The other important subset of \mathcal{C}_{adm} is the set of all configurations in which the robot is grasping the part (and is capable of carrying it, if necessary). Let this denote the *grasped configurations*, denoted by $\mathcal{C}_{gr} \subseteq \mathcal{C}_{adm}$. For every configuration, $(q^a, q^p) \in \mathcal{C}_{gr}$, we require that the manipulator touches the part. This means that $\mathcal{A}(q^a) \cap \mathcal{P}(q^p) \neq \emptyset$ (penetration is still not allowed because $\mathcal{C}_{gr} \subseteq \mathcal{C}_{adm}$). In general many configurations at which $\mathcal{A}(q^a)$ contacts $\mathcal{P}(q^p)$ will not necessarily be in \mathcal{C}_{gr} . The conditions for a point to lie in \mathcal{C}_{gr} depend on the particular characteristics of the manipulator, the part, and the contact surface between them. For example, a typical manipulator would not be able to pick up a block by making contact with only one corner of it. This level of detail is not defined here; see [] for more information about grasping.

We must always ensure that either $x \in \mathcal{C}_{sta}$ or $x \in \mathcal{C}_{gr}$. Therefore, let $\mathcal{C}_{free} = \mathcal{C}_{sta} \cup \mathcal{C}_{gr}$, to reflect the subset of \mathcal{C}_{adm} which will actually be allowed for manipulation planning.

The mode space, M , contains two modes, which are named the *transit mode* and the *transfer mode*. In the transit mode, the manipulator is not carrying the part, which requires that $q \in \mathcal{C}_{sta}$. In the transfer mode, the manipulator carries the part, which requires that $q \in \mathcal{C}_{gr}$. Based on these simple conditions, the only way the mode can change is if $q \in \mathcal{C}_{sta} \cap \mathcal{C}_{gr}$. Therefore, the manipulator is given two actions only when in these configurations. In all other configurations the mode must remain constant. For convenience, let $\mathcal{C}_{tra} = \mathcal{C}_{sta} \cap \mathcal{C}_{gr}$ denote the set of *transition configurations*, which are the places in which the mode may change.

Using the framework of Section 7.3.1, the mode space, M , and configuration space, \mathcal{C} , are combined to yield the *state space*, $X = \mathcal{C} \times M$. Since there are only two modes, there are only two copies of \mathcal{C} , one for each mode. State-based sets, X_{free} , X_{tra} , X_{sta} , and X_{gr} , are directly obtained from \mathcal{C}_{free} , \mathcal{C}_{tra} , \mathcal{C}_{sta} , and \mathcal{C}_{gr} by ignoring the mode. For example,

$$X_{tra} = \{(q, m) \in X \mid q \in \mathcal{C}_{tra}\}. \quad (7.20)$$

The sets X_{free} , X_{sta} and X_{gr} are similarly defined.

The task can now be defined. An *initial part configuration*, $q_{init}^p \in \mathcal{C}_{sta}$ and *goal part configuration*, $q_{goal}^p \in \mathcal{C}_{sta}$ are specified. Compute a path $\tau : [0, 1] \rightarrow X_{free}$

such that $\tau(0) = q_{init}^p$ and $\tau(1) = q_{goal}^p$. Furthermore, the *action function* $\sigma : [0, 1] \rightarrow U$ must be specified to indicate the appropriate mode changes whenever $\tau(s) \in X_{tra}$. A solution can be considered as an alternating sequence of *transit paths* and *transfer paths*, whose names follow from the mode. This is depicted in Figure 7.18.

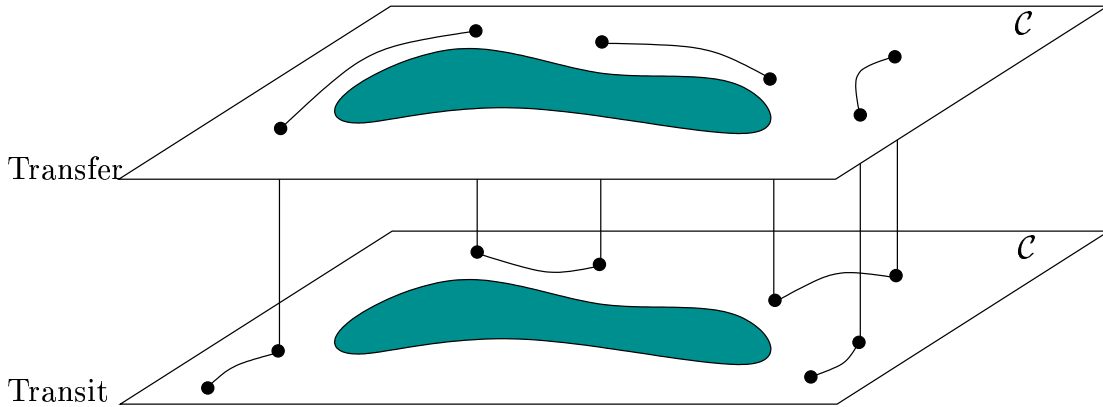


Figure 7.18: The solution to a manipulation planning problem alternates between the two layers of X . The transitions can only occur when $x \in X_{tra}$.

Manipulation graph The manipulation planning problem can generally be solved by forming a manipulation graph, G_m [9, 10]. Let a *connected component* of X_{tra} refer to any connected component of \mathcal{C}_{tra} that is lifted into the state space by ignoring the mode. In other words, there are two copies of the connected component of \mathcal{C}_{tra} , one for each mode. For each connected component of X_{tra} , a vertex exists G_m . An edge is defined for each transfer path or transit path that connects two connected components of X_{tra} . The general approach to manipulation planning then becomes:

1. Compute the connected components of X_{tra} .
2. Compute the edges of G_m by applying ordinary planning methods to each pair of vertices of G_m .
3. Apply planning methods to connect the initial and goal states to every possible vertex of X_{tra} that can be reached without a mode transition.
4. Search G_m for a path that connects the initial and goal states. If one exists, then extract the corresponding solution as a sequence of transit and transfer paths (this implies the actions taken by the robot, to execute the required mode changes).

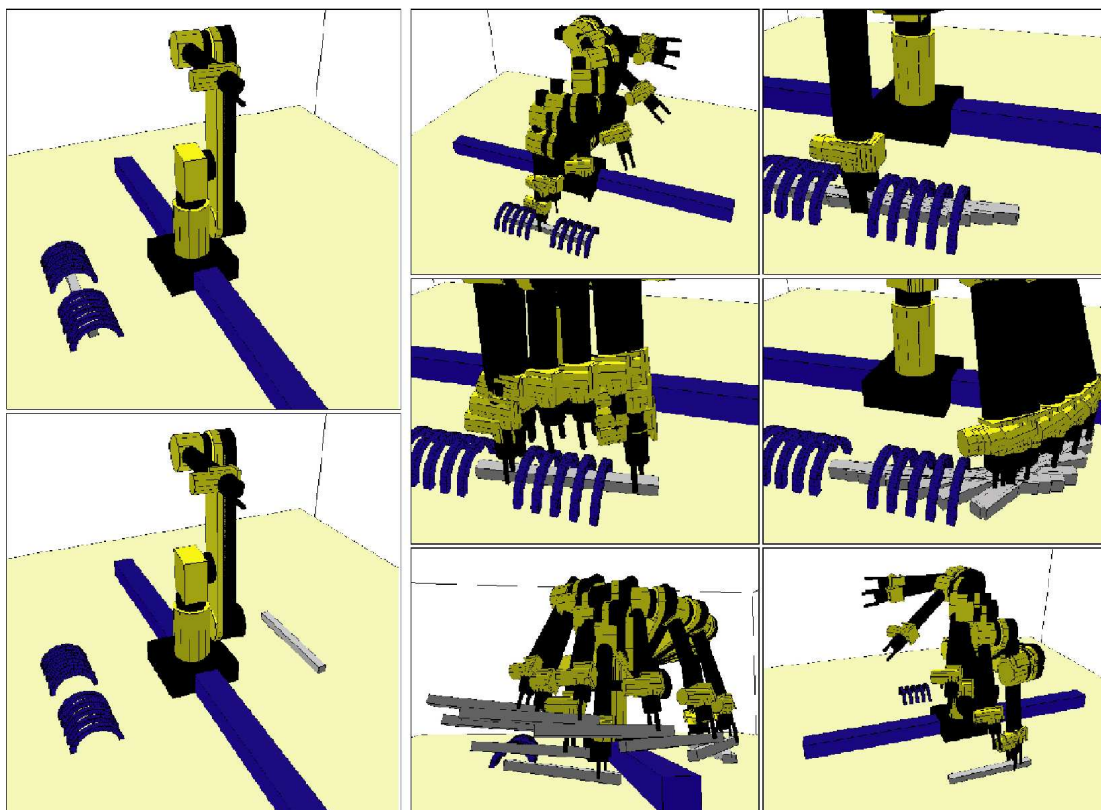


Figure 7.19: This example was solved in [177] using the manipulation planning framework and the visibility-based roadmap planner. It is very challenging because the same part must be regrasped in many places.

Multiple parts The manipulation planning framework nicely generalizes to multiple parts, $\mathcal{P}_1, \dots, \mathcal{P}_k$. Each part has its own part configuration space, and \mathcal{C} is formed by taking the Cartesian product of all part configuration spaces with the manipulator configuration space. The set \mathcal{C}_{adm} is defined in a similar way, but now part-part collisions also have to be removed, in addition to part-manipulator, manipulator-obstacle, and part-obstacle collisions. The definition of \mathcal{C}_{sta} requires that all parts are in stable configurations; the parts may even be allowed to stack on top of each other. The definition of \mathcal{C}_{gr} requires that one part is grasped and all other parts are stable. There are still two modes, depending on whether or not the manipulator is grasping a part. Transitions once again only occur when the robot is in $\mathcal{C}_{tra} = \mathcal{C}_{sta} \cap \mathcal{C}_{gr}$. The task involves moving all parts from one configuration to another. This is achieved once again by defining a manipulation graph, and obtaining a sequence of transit paths (in which no parts move) and transfer paths (in which one part is carried, and all other parts are fixed). A challenging problem solved by a motion planning algorithm is shown in Figure 7.19.

Other generalizations are possible. A generalization to k robots would lead to 2^k modes, in which each mode indicates whether or not each robot is grasping. Multiple robots could even grasp the same object. Another generalization could allow a single robot to grasp more than one object.

7.4 Planning for Closed Kinematic Chains

This section continues where Section 4.4 finished. The subspace of \mathcal{C} that results from maintaining kinematic closure was defined and illustrated through some examples. Planning in this context requires that paths remain on a lower-dimensional variety for which a parameterization is not available. Many important applications require motion planning while maintaining these constraints. For example, consider a manipulation problem that involves multiple manipulators grasping the same object forms a closed loop, as shown in Figure 7.21. A loop exists because both manipulators are attached to the ground, which may itself be considered as a link. The development of virtual actors for movies and video games also involves related manipulation problems. Loops also arise in this context when more than one human limb is touching a fixed surface (e.g., two feet on the ground). A class of robots called *parallel manipulators* are designed with internal closed loops [550]. For example, consider the Stewart-Gough platform [296, 724] illustrated in Figure 7.20. The lengths of each of the six arms, $\mathcal{A}_1, \dots, \mathcal{A}_6$ can be independently varied, while they remain attached via spherical joints to the ground and to the *platform*, which is \mathcal{A}_7 . Each arm can actually be imagined as two links that are connected by a prismatic joint. Due to the total of 6 degrees of freedom introduced by the variable lengths, the platform actually achieves the full 6 degrees of freedom (hence, some neighborhood in $SE(3)$ is obtained for \mathcal{A}_7). Planning the motion of the Stewart-Gough platform, or robots

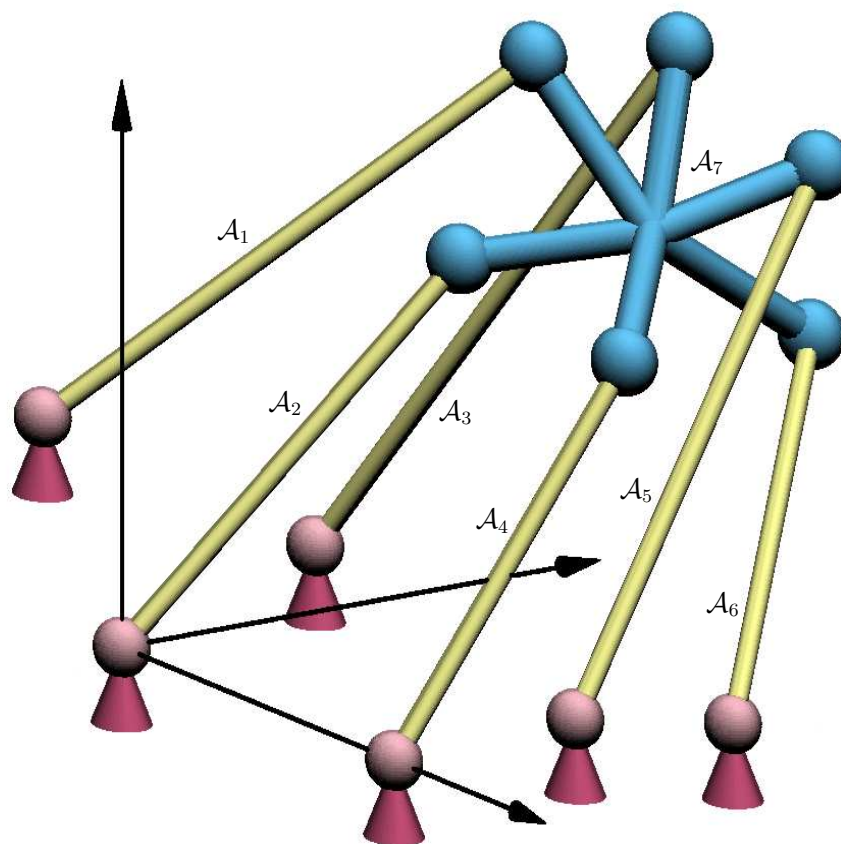


Figure 7.20: An illustration of the Stewart-Gough platform (adapted from a figure made by Frank Sottile).

that are based on the platform (the robot shown in Figure 7.29 that uses a stack of several of these mechanisms), requires handling many closure constraints that must be maintained simultaneously. Another application is computational biology, in which the configuration space of molecules is searched, many of which are derived from molecules that have closed, flexible chains of bonds [].

7.4.1 Adaptation of Motion Planning Algorithms

First, the planning problem will be precisely defined. All of the components from the general motion planning problem of Model 4.3.1 are included: \mathcal{W} , \mathcal{O} , $\mathcal{A}_1, \dots, \mathcal{A}_r$, \mathcal{C} , q_i , and q_g . It is assumed that the robot is a collection of r links that are possibly attached in loops.

It will be assumed in this section that $\mathcal{C} = \mathbb{R}^n$. If this is not satisfactory, there are two ways to overcome the assumption. The first is to represent $SO(2)$ and $SO(3)$ as \mathbb{S}^1 and \mathbb{S}^3 , respectively, and include the circle or sphere equation as part of the constraints considered here. This avoids the topology problems. The other option

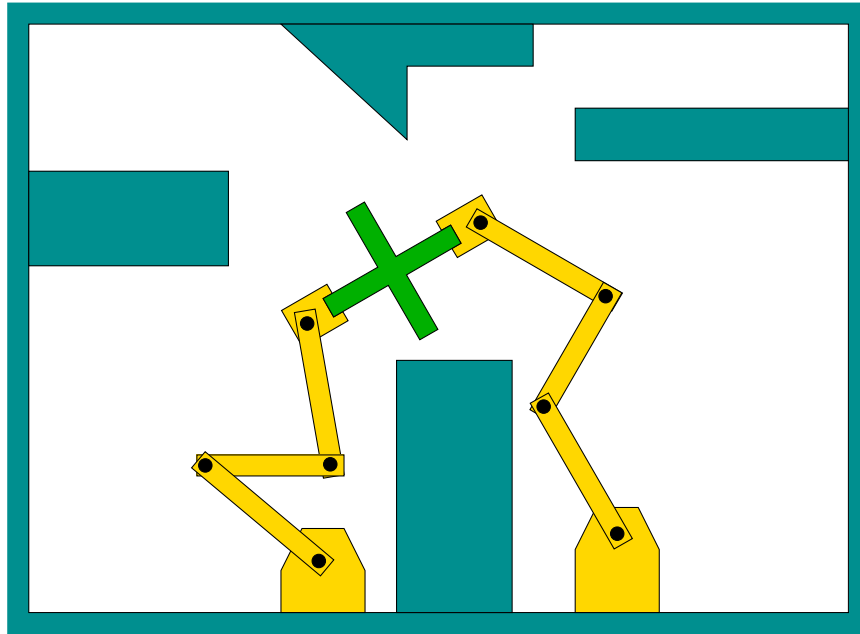


Figure 7.21: Two or more manipulators manipulating the same object causes closed kinematic chains. Each black disc corresponds to a revolute joint.

is to use abandon the restriction of using \mathbb{R}^n , and instead use a parameterization of \mathcal{C} that is of the appropriate dimension. To perform calculus on such manifolds, it *differentiable structure* is required, which is introduced in Section ???. In the presentation here, however, vector calculus on \mathbb{R}^n is sufficient, which intentionally avoids these extra technicalities.

Closure constraints The closure constraints introduced in Section 4.4, can be summarized as follows. There is a set \mathcal{P} of polynomials f_1, \dots, f_k , which belong to $\mathbb{Q}[q_1, \dots, q_n]$ and express the constraints for particular points on the links of the robot. The determination of these is detailed in Section 4.4.3. As mentioned above, polynomials that force points to lie on a circle or sphere in the case of rotations, may also be included in \mathcal{P} .

The *closure space*, \mathcal{C}_{clo} , is defined as

$$\mathcal{C}_{clo} = \{q \in \mathcal{C} \mid \forall f_i \in \mathcal{P}, f_i(q_1, \dots, q_n) = 0\}, \quad (7.21)$$

which is an m -dimensional subspace of \mathcal{C} that corresponds to all configurations that satisfy the closure constants. The obstacle set must also be taken into account. Once again, \mathcal{C}_{obs} and \mathcal{C}_{free} can be defined using (4.40). The *feasible space*, \mathcal{C}_{fea} is defined as $\mathcal{C}_{fea} = \mathcal{C}_{clo} \cap \mathcal{C}_{free}$, which are the configurations that satisfy closure constraints and avoid collisions.

Let n denote the dimension of \mathcal{C} . The motion planning problem then becomes the task of finding a path $\tau : [0, 1] \rightarrow \mathcal{C}_{fea}$ such that $\tau(0) = q_i$ and $\tau(1) = q_g$. The new challenge is that there is no explicit parameterization of \mathcal{C}_{fea} , which is further complicated by the fact that $m < n$.

Combinatorial methods Since the constraints are expressed with polynomials, it may not be surprising that the computational algebraic geometry methods of Section 6.4 can solve the general motion planning problem with closed kinematic chains. Either cylindrical algebraic decomposition or Canny's roadmap algorithm may be applied. As mentioned in Section 6.5.3, an adaptation of the roadmap algorithm which is optimized for problems in which $m < n$ is given in [57].

Sampling-based methods Most of the methods of Section 5 are not easy to adapt because they require sampling in \mathcal{C}_{fea} , for which a parameterization does not even exist. If points in a bounded region of \mathbb{R}^n are chosen at random, the probability is zero that a point on \mathcal{C}_{fea} will be hit. Some incremental sampling and searching methods can, however, be adapted by the construction of a local planning method (LPM) that is suited for problems with closure constraints. The sampling-based roadmap methods require many samples to be generated directly on \mathcal{C}_{fea} . Section 7.4.2 presents some techniques that can be used to generate such samples for certain classes of problems, enabling the development of efficient sampling-based planners, and also improving the efficiency of incremental search planners. Before covering these techniques, we first present a method that leads to a more-general sampling based planner and is easier to implement. However, if designed well, planners based on the techniques of Section 7.4.2 are more efficient.

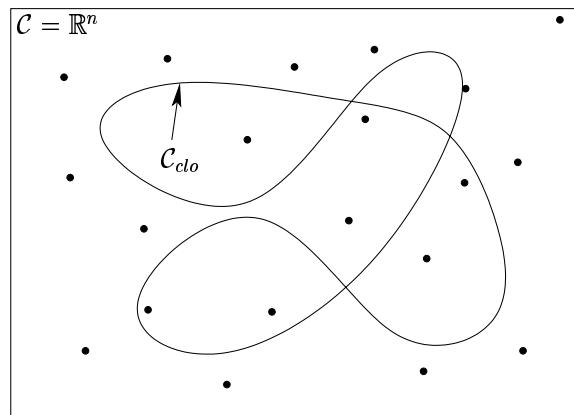


Figure 7.22: For the RDT, the samples can be drawn from a region in \mathbb{R}^n , the space in which \mathcal{C}_{clo} is embedded.

We now consider adapting the RDT of Section 5.5 to work for problems with closure constraints. Similar adaptations may be possible for other incremental sampling and searching methods, covered in Section 5.4, such as the randomized

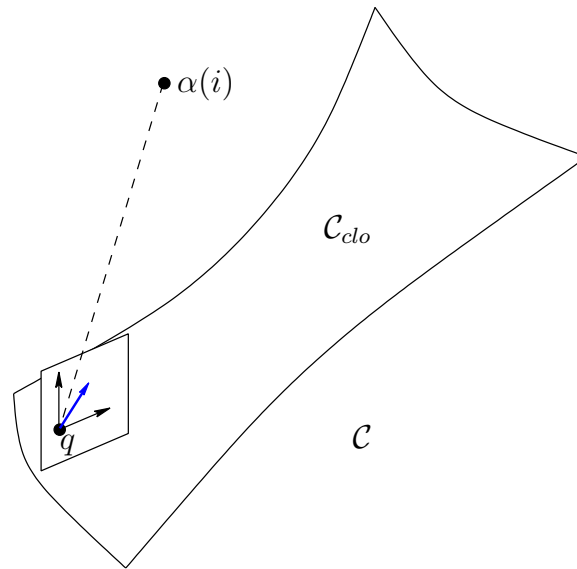


Figure 7.23: For each sample $\alpha(i)$, the nearest point, $q_n \in \mathcal{C}$ is found, and then the local planner generates a motion that lies in the local tangent plane. The motion is the project of the vector from q_n to $\alpha(i)$ onto the tangent plane.

potential field planner. A dense sampling sequence, α , is generated over a bounded n -dimensional subset of \mathbb{R}^n , such as a rectangle or sphere, as shown in Figure 7.22. The samples are not actually required to lie on \mathcal{C}_{clo} because they do not necessarily become part of the topological graph, G . They mainly serve to pull the search tree in different directions. One concern in choosing the bounding region is to make it large enough to include \mathcal{C}_{clo} (at least the connected component that includes q_i), but as small as possible while satisfying this requirement. Such bounds by carefully analyzing the motion limits for a particular linkage.

Stepping along \mathcal{C}_{clo} The RDT algorithm given Figure 5.27 can be applied directly; however, the STOPPING-CONFIGURATION function in Line 4 must be changed to account for both obstacles and the constraints that define \mathcal{C}_{clo} . Figure 7.23 shows the general approach, which is based on *numerical continuation* [?]. The nearest RDT vertex, $q \in \mathcal{C}$, to the sample $\alpha(i)$, is first computed. Let $v = \alpha(i) - q$, which represents the direction in which an edge would be made from q if there were no constraints. A local motion is then computed by projecting v into the tangent plane of \mathcal{C}_{clo} at the point q . Since \mathcal{C}_{clo} is generally nonlinear, the local motion will produce a point that is not precisely on \mathcal{C}_{clo} . Some numerical tolerance is generally accepted, and a small enough step is taken to ensure that the tolerance is maintained. The process iterates by computing v with respect to the new point, and moving in the direction of v projected into the new tangent plane. If the error threshold is reached, then motions must be executed in the normal direction to return to \mathcal{C}_{clo} . This process terminates when progress can no longer

be made, either due to the alignment of the tangent plane (nearly perpendicular to v) or due to an obstacle. This finally yields q_s , the stopping configuration. The new path followed in \mathcal{C}_{fea} is no longer a “straight line” as was possible for some problems in Section 5.5; therefore, the approximate methods in Section 5.5.2 should be used to create intermediate vertices along the path.

In each iteration, the tangent plane computation is computed at some $q \in \mathcal{C}_{clo}$ as follows. The differential configuration vector dq lies in the tangent space of a constraint $f_i(q) = 0$ if

$$\frac{\partial f_i(q)}{\partial q_1} dq_1 + \frac{\partial f_i(q)}{\partial q_2} dq_2 + \cdots + \frac{\partial f_i(q)}{\partial q_n} dq_n = 0. \quad (7.22)$$

This leads to the following homogeneous system for all of the k polynomials in \mathcal{P} that define the closure constraints:

$$\begin{pmatrix} \frac{\partial f_1(q)}{\partial q_1} & \frac{\partial f_1(q)}{\partial q_2} & \cdots & \frac{\partial f_1(q)}{\partial q_n} \\ \frac{\partial f_2(q)}{\partial q_1} & \frac{\partial f_2(q)}{\partial q_2} & \cdots & \frac{\partial f_2(q)}{\partial q_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_k(q)}{\partial q_1} & \frac{\partial f_k(q)}{\partial q_2} & \cdots & \frac{\partial f_k(q)}{\partial q_n} \end{pmatrix} \begin{pmatrix} dq_1 \\ dq_2 \\ \vdots \\ dq_n \end{pmatrix} = \mathbf{0}. \quad (7.23)$$

If the rank of the matrix is $m \leq n$, then m configuration displacements can be chosen independently, and the remaining $n - m$ parameters must satisfy Equation 7.23. This can be solved using linear algebra techniques, such as singular value decomposition (SVD) [287], to compute an orthonormal basis for the tangent space at q . Let e_1, \dots, e_m , denote these n -dimensional basis vectors. The components of the motion direction are obtained from $v = \alpha(i) - q_n$. First, construct the inner products, $a_1 = v \cdot e_1$, $a_2 = v \cdot e_2$, \dots , $a_m = v \cdot e_m$. Using these, the projection of v in the tangent plane is the n -dimensional vector w given by

$$w = \sum_i^m a_i e_i. \quad (7.24)$$

This is used as the direction of motion. The magnitude must be appropriately scaled to take sufficiently small steps. Because \mathcal{C}_{clo} is nonlinear, the direction of motion will leave \mathcal{C}_{clo} . A motion in the inward normal direction is then required to move back onto \mathcal{C}_{clo} .

Because the dimension, m , of \mathcal{C}_{clo} is less than n , the procedure described above can only produce numerical approximations to paths in \mathcal{C}_{clo} . This problem also arises in implicit curve tracing in graphics and geometric modeling [331]. Therefore, each constraint $f_i(q_1, \dots, q_n) = 0$, is actually slightly weakened to $|f_i(q_1, \dots, q_n)| < \epsilon$ for some fixed tolerance $\epsilon > 0$. This essentially “thickens” \mathcal{C}_{clo}

so that its dimension is n . As an alternative to computing the tangent plane, motion directions can be sampled directly inside of this thickened region without computing tangent planes. This results in an easier implementation, but it is not as efficient [780].

7.4.2 Active-Passive Link Decompositions

An alternative sampling-based approach is to perform an *active-passive decomposition*, which is used to generate samples in \mathcal{C}_{clo} by directly sampling *active* variables, and computing the closure values for *passive* variables using inverse kinematics methods. This method was introduced in [313], and subsequently improved through the development of the *random loop generator* in [177, 363]. The method serves as a general framework that can adapt virtually any of the methods of Section 5 to handle closed kinematic chains, and experimental evidence suggests that performance is better than the method of Section 7.4.1. One drawback is that the method requires some careful analysis of the linkage to determine the best decomposition and also bounds on its mobility. Such analysis exists for very general classes of linkages [177]; however, many challenging cases remain unsolved.

Active and passive variables In this section, let \mathcal{C} denote the configuration space obtained from all of the joint variables, instead of requiring $\mathcal{C} = \mathbb{R}^n$, as in Section 7.4.1. This means that \mathcal{P} includes only polynomials that encode closure constraints, as opposed to allowing constraints that represent rotations. Using the tree representation from Section 4.4.3, this means that \mathcal{C} is of dimension n , arising from assigning one variable for every joint of the linkage in the absence of any constraints. Let $q \in \mathcal{C}$ denote this vector of configuration variables. The *active-passive* decomposition partitions the variables of q to form two vectors, q^a , called the *active variables* and q^p , called the *passive variables*. The values of passive variables will always be determined from the active variables by enforcing the closure constraints and using inverse kinematics techniques to compute their values. If m is the dimension of \mathcal{C}_{clo} , then there are always m active variables and $n - m$ passive variables.

Temporarily, suppose that the linkage forms a single loop as shown in Figure 7.24. One possible decomposition into active, q^a , and passive, q^p , variables is given in Figure 7.25. The linkage, when constrained to form a loop, has four degrees of freedom, assuming the bottom link is rigidly attached to the ground. This means that values can be chosen for four active joint angles, q^a , and the remaining three, q^p , can be derived from solving the inverse kinematics. To determine q^p , note that there will be three equations and three unknowns. Unfortunately, these equations are nonlinear and fairly complicated. Nevertheless, efficient solutions exist for this case, and the three-dimensional generalization [529]. For a three-dimensional loop formed of revolute joints, there are six passive variables. The number, 3, of passive links in \mathbb{R}^2 and the number 6 for \mathbb{R}^3 arise from the dimensions of $SE(2)$

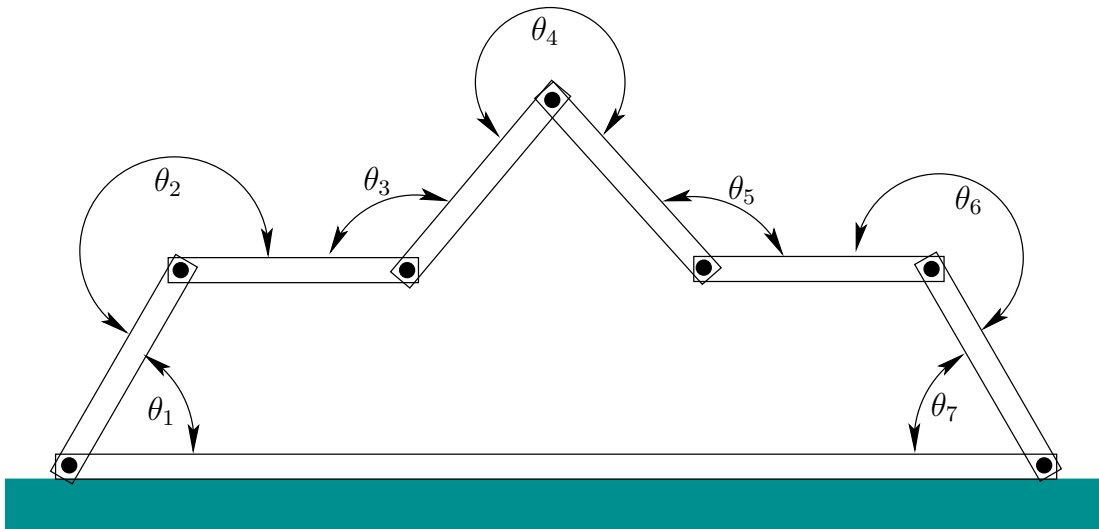


Figure 7.24: A chain of links in the plane. There are 7 links and 7 joints, which are constrained to form a loop. The dimension of \mathcal{C} is 7, but the dimension of \mathcal{C}_{clo} is 4.

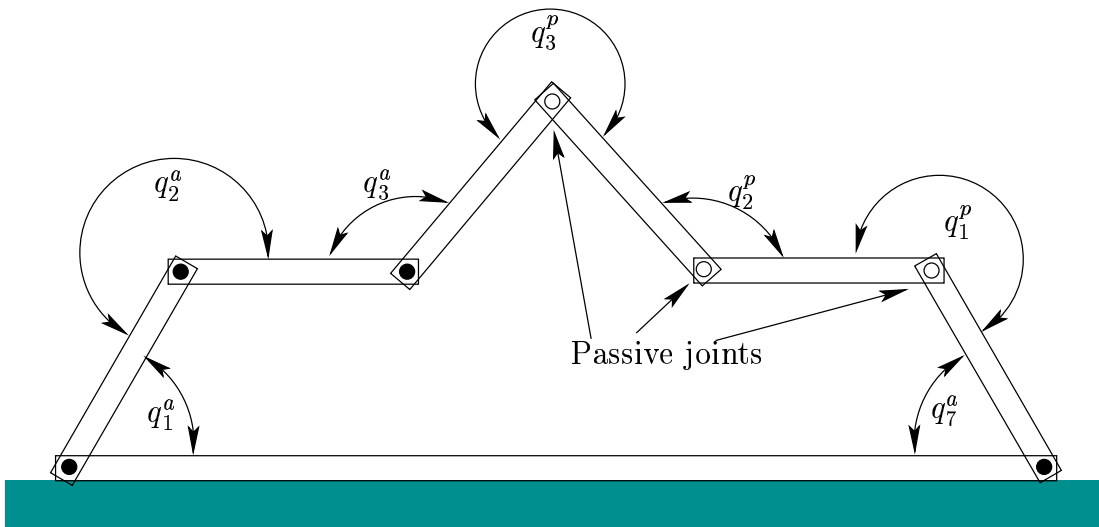


Figure 7.25: Three of the joint variables can be determined automatically by inverse kinematics. Therefore, 4 of the joints be designated as *active*, and the remaining 3 will be passive.

and $SE(3)$, respectively. This is the freedom that is stripped away from system by enforcing the closure constraints. Methods for efficiently computing inverse kinematics in two and three dimensions are given in [20].

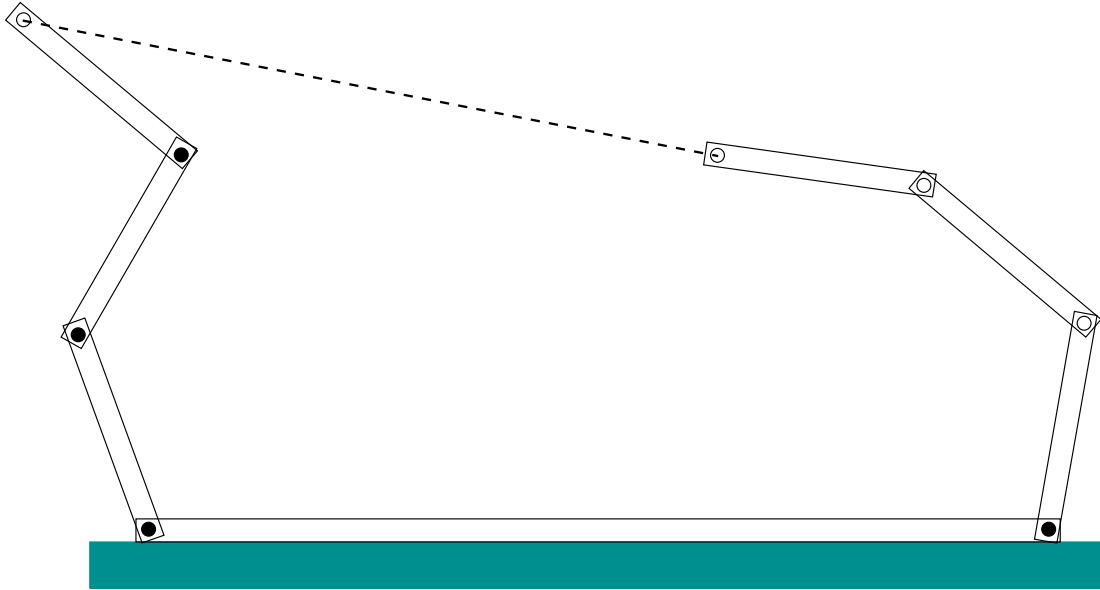


Figure 7.26: In this case, the active variables are chosen in a way that makes it impossible to assign passive variables that close the loop.

There will be at most a finite number of solutions to the inverse kinematics problem, often leading to several choices for the passive variables. It could also be the case that for some assignments of active variables, there are no solutions to the inverse kinematics. An example is depicted in Figure 7.26. Suppose that we want to generate samples in \mathcal{C}_{clo} by selecting random values for q^a , and then using inverse kinematics for q^p . What is the probability that a solution to the inverse kinematics exists? For the example shown, it appears that most of time solutions would not exist.

Loop generator The sampling method in [177, 363] (termed the *random loop generator*) greatly improves the chance of obtaining closure by iteratively restricting the range on each of the active variables. The method requires that the active variables appear sequentially along the chain (i.e., there is no interleaving of active and passive variables). The m coordinates of q^a are obtained sequentially as follows. First, compute an interval, I_1 , of allowable values for q_1^a . The interval serves as a loose bound in the sense for any value $q_1^a \notin I_1$, it is known for certain that closure cannot be obtained. This is ensured by performing careful geometric analysis of the linkage, which will be explained shortly. The next step is to generate a sample in $q_1^a \in I_1$, which is accomplished in [177] by picking a random point in I_1 . Using the value q_1^a , a bounding interval I_2 is computed for allowable

values of q_2^a . The value q_2^a is obtained by sampling in I_2 . This process continues iteratively until I_m and q_m^a is obtained, unless it terminates early because some $I_i = \emptyset$ for $i < m$. If successful termination occurs, then the active variables q^a are used to find values q^p for the passive variables. This step still might fail, but the probability of success is now much higher. The method can also apply to linkages in which there are multiple, common loops, as in the Stewart-Gough platform, by breaking the linkage into a tree, and closing loops one at a time using the loop generator. The performance depends on how the linkage is decomposed [177].

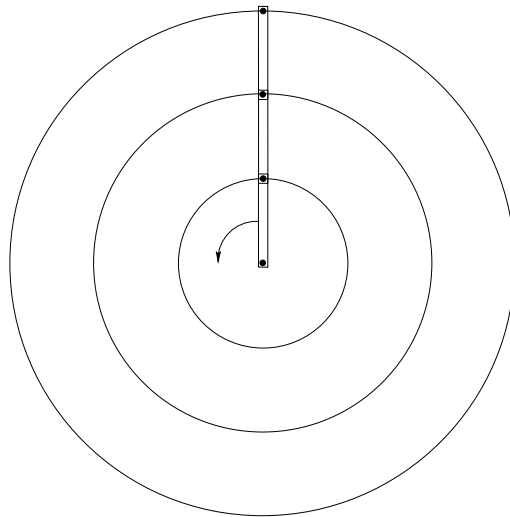


Figure 7.27: If any joint angle is possible, then the links sweep out a circle in the limit.

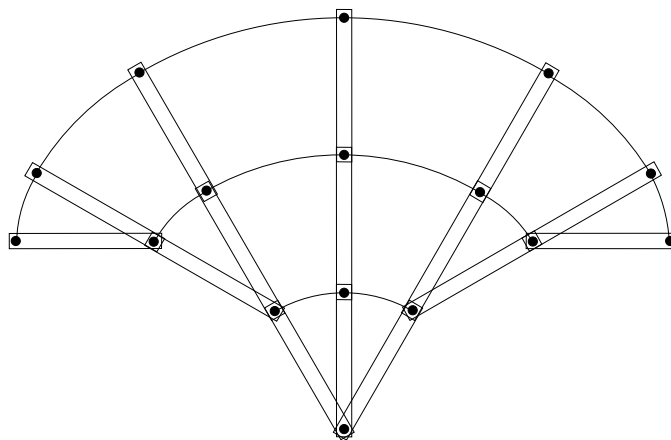


Figure 7.28: If there are limits on the joint angles, then a tighter bound can be obtained for the reachability of the linkage.

Computing bounds on joint angles The main requirement for successful application of the method is the ability to compute bounds on how far a chain of links can travel in \mathcal{W} over some range of variables. For example, for a planar chain that has revolute joints with no limits, the chain can sweep out a circle as shown in Figure 7.27. Suppose, it is known that the angle between links must remain between $-\pi/6$ and $\pi/6$. A tighter bounding region can be obtained, as shown in Figure 7.28. Three-dimensional versions of these bounds, along with many necessary details, are included in [177]. These bounds are then used to compute I_i in each iteration of the sampling algorithm.

Now that there is an efficient method that generates samples directly in \mathcal{C}_{clo} , it is straightforward to adapt any of the sampling-based planning methods of Chapter 5. In [177] many impressive results are obtained for challenging problems which have the dimension of \mathcal{C} up to 97 and the dimension of \mathcal{C}_{clo} up to 25; see Figure 7.29. These methods are based on applying these sampling technique to the RDTs of Section 5.5 and the visibility sampling-based roadmap of Section 5.6.2. For these algorithms, the local planning method is applied to the active variables, and inverse kinematics algorithms are used for the passive variables in the path validation step. This means that inverse kinematics and collision checking are performed together, instead of only collision checking, as described in Section 5.3.4.

One important issue that has been neglected in this section is the existence of *kinematic singularities*, which cause the dimension of \mathcal{C}_{clo} to drop in the vicinity of certain points. The methods presented here have assumed that solving the motion planning problem does not require passing through the singularity. This assumption is reasonable for robot systems that have many extra degrees of freedom, but it is important understand that completeness is lost in general because the sampling-based methods do not explicitly handle these degeneracies. In a sense, they occur below the level of sampling resolution. For more information on kinematic singularities and related issues, see [550].

7.5 Folding Problems in Robotics and Biology

A growing number of motion planning applications involve some form of folding. Examples include automated carton folding, computer-aided drug design, protein folding, modular reconfigurable robots, and even robotic origami. These problems are generally modeled as a linkage in which all bodies are connected by revolute joints. In robotics, self-collision between pairs of bodies usually must be avoided. In biological applications, energy functions replace obstacles. Instead of crisp obstacle boundaries, energy functions can be imagined as “soft” obstacles, in which a real-value is defined for every $q \in \mathcal{C}$, instead of defining a set $\mathcal{C}_{obs} \subset \mathcal{C}$. For a given threshold value, such energy functions can be converted into an obstacle region by defining \mathcal{C}_{obs} to be the configurations that have energy above the threshold. However, the energy function contains more information because such thresholds

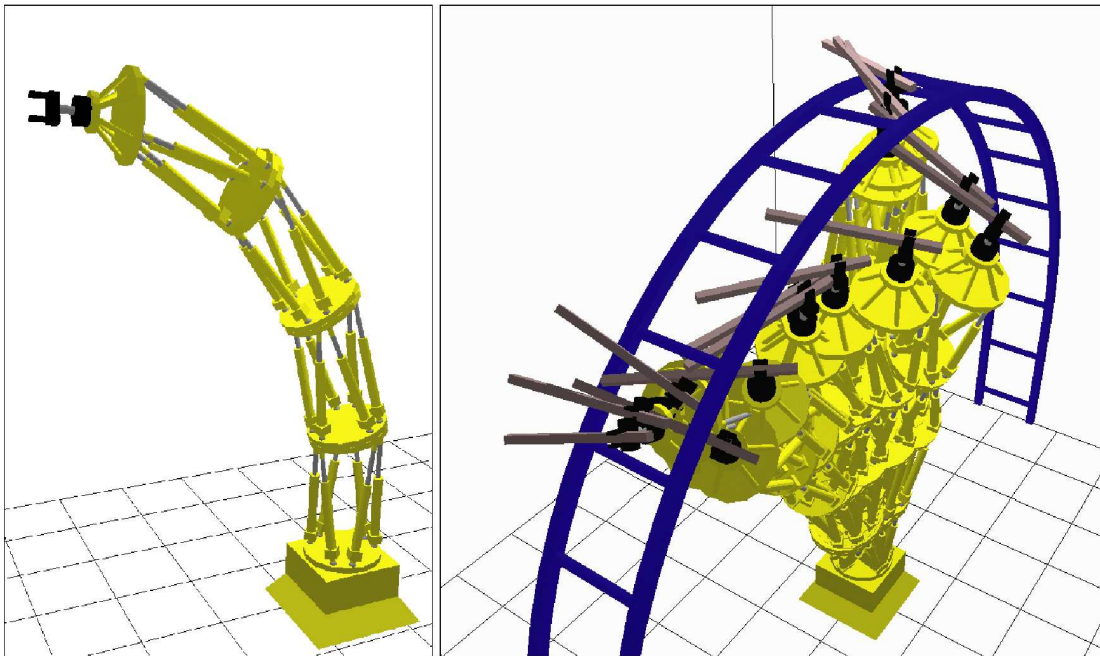


Figure 7.29: Planning for the Logabex LX4 robot [?]. This solution was computed in less than a minute in [177] by applying active-passive decomposition to an RDT-based planner. In this example, the dimension of \mathcal{C} is 97 and the dimension of \mathcal{C}_{clo} is 25.

are arbitrary. This section briefly shows some examples of folding problems and techniques from the recent motion planning literature.

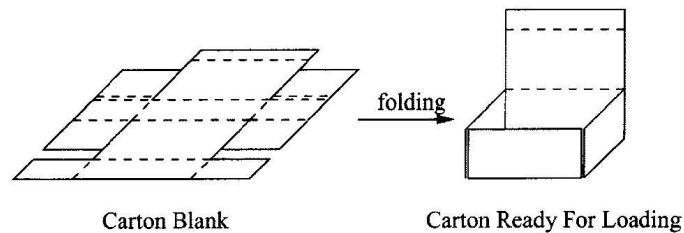


Figure 7.30: An important packaging problem is to automate the folding of a perforated sheet of cardboard into a carton.

Carton folding An interesting application of motion planning to the automated folding of boxes is presented in [509]. Figure 7.30 shows a carton in its original flat form and in its folded form. As shown in Figure 7.31, the problem can be modeled as tree of bodies connected by revolute joints. Once this model has been formulated, many methods from Chapters 5 and 6 can be adapted for this problem. In [509], a planning algorithm optimized particularly for box folding is presented. It is an adaptation of an approximate cell decomposition algorithm developed

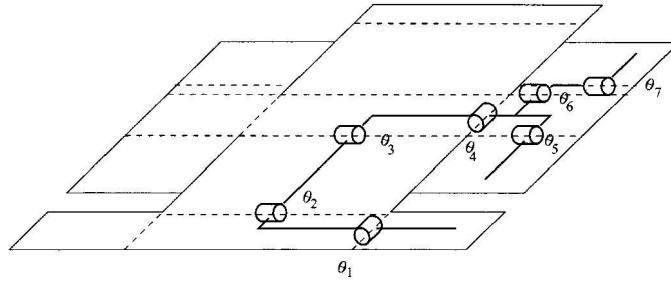


Figure 7.31: The carton can be cleverly modeled as a tree of bodies that are attached by revolute joints.

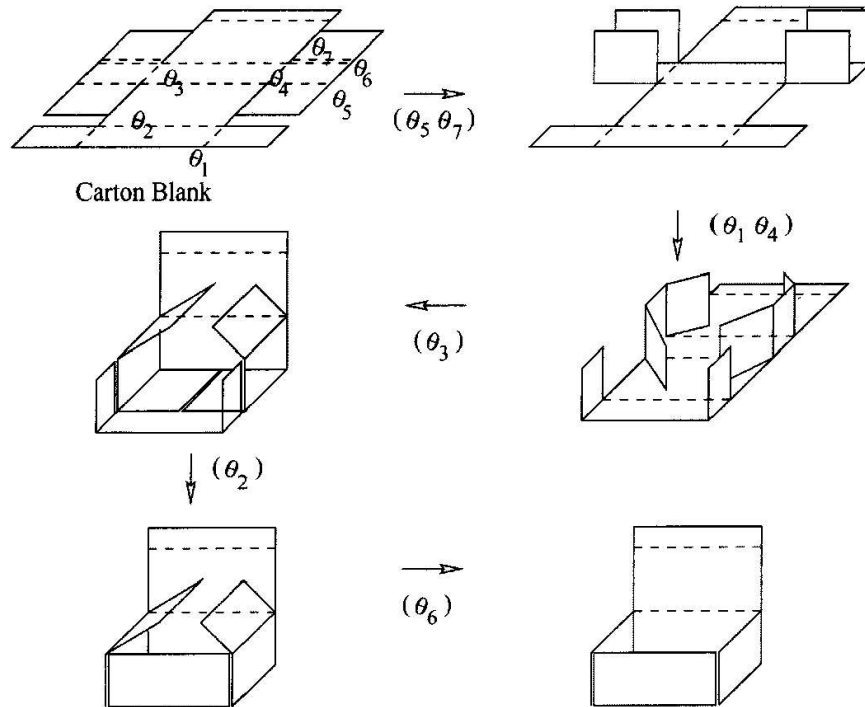


Figure 7.32: A folding sequence that was computed using the algorithm in [509].

for kinematic chains in [505]. Its complexity is exponential in the degrees of freedom of the carton, but gives good performance on practical examples. One such solution that was found by motion planning is shown in Figure 7.32. To use these solutions in a factory, the manipulation problem has to be additionally considered. For example, as demonstrated in [509], a manipulator arm robot can be used in combination with a well-designed set of fixtures. The fixtures help hold the carton in place while the manipulator applies pressure in the right places, which yields the required folds. Since the feasibility with fixtures depends on the particular folding path, the planning algorithm generates all possible distinct paths from the initial, flat configuration.

Simplifying knots A *knot* is defined as a closed curve that does not intersect itself, is embedded in \mathbb{R}^3 , and cannot be untangled to produce a simple loop. If the knot is allowed to intersect itself, then any knot can be untangled; therefore, a careful definition of what it means to untangle a knot is needed. For a closed curve, $\tau : [0, 1] \rightarrow \mathbb{R}^3$, embedded in \mathbb{R}^3 (it cannot intersect itself), let the set $\mathbb{R}^3 \setminus \tau([0, 1])$ of points not reached by the curve be called the *ambient space* of τ . In knot theory, an *ambient isotopy* between two closed curves, τ_1 and τ_2 , embedded in \mathbb{R}^3 , is a homeomorphism between their ambient spaces. Intuitively, this means that τ_1 can be warped into τ_2 without allowing any self-intersections. Therefore, determining whether two loops are equivalent seems closely related to motion planning. Such equivalence gives rise to groups that characterize the space of knots, and are closely related to the fundamental group described in Section 4.1.3. For more information on knot theory, see [3, 328, 382].

A motion planning approach was developed in [424] to determine whether a closed curve is equivalent to the *unknot*, which is completely untangled. This can be expressed as a curve that maps onto \mathbb{S}^1 embedded in \mathbb{R}^3 . The algorithm takes as input a knot expressed as a circular chain of line segments embedded in \mathbb{R}^3 . In this case, the unknot can be expressed as a triangle in \mathbb{R}^3 . One of the most challenging examples solved by the planner is shown in Figure 7.33. The planner is sampling-based and shares many similarities with the RDT algorithm of Section 5.5, and the Ariadne's clew and expansive space planners described in Section 5.4.4. Since the task is not to produce a collision-free path, there are several unique aspects in comparison to motion planning. An energy function is defined over the collection of segments to try to guide the search toward simpler configurations. There are two kinds of local operations that are made by the planner: 1) Try to move a vertex toward a selected subgoal in the ambient space. This is obtained by using random sampling to grow a search tree. 2) Try to delete a vertex, and connect the neighboring vertices by a straight line. If no collision occurs, then the knot has been simplified. The algorithm terminates when it is unable to further simplify the knot.

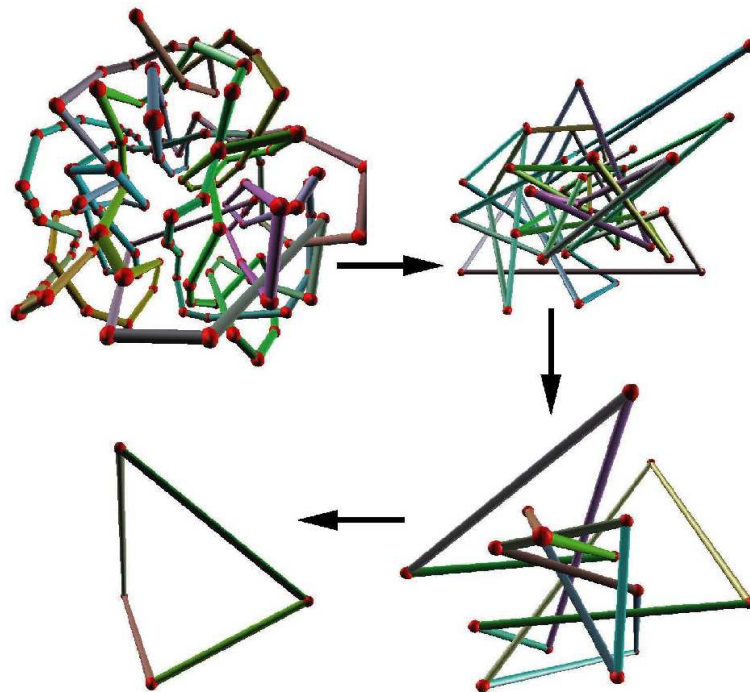


Figure 7.33: The planner in [424] untangles the famous Ochiai unknot benchmark in a few minutes on a standard PC.

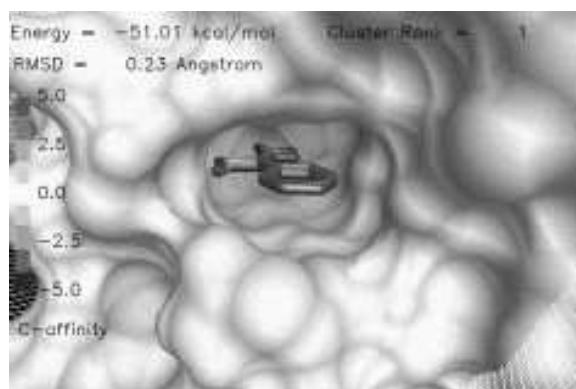


Figure 7.34: A 3D model of protein-ligand docking.

Drug Design A sampling-based motion planning approach to pharmaceutical drug design is taken in [455]. The development of a drug is a long, incremental process, typically requiring years of research and experimentation. The goal is to find a relatively small molecule (called a *ligand*) typically comprising a few dozen atoms, that docks with a receptor cavity in a specific protein [473]; Figure 7.34 shows an illustration. Examples of drug molecules were given in Figure 3.22. Protein-ligand docking can stimulate or inhibit some biological activity, ultimately leading to the desired pharmacological effect. The problem of finding suitable ligands is complicated due to both energy considerations and the flexibility of the ligand. In addition to satisfying structural considerations, factors such as synthetic accessibility, drug pharmacology and toxicology greatly complicate and lengthen the search for the most effective drug molecules.

One popular model used by chemists in the context of drug design is a *pharmacophore*, which serves as a template for the desired ligand [167, 249, 275, 681]. The pharmacophore is expressed as a set of *features* that an effective ligand should possess and a set of *spatial constraints* among the features. The features can be specific atoms, centers of benzene rings, positive or negative charges, hydrophobic or hydrophilic centers, hydrogen bond donors or acceptors, and others. These features generally require that parts of the molecule must remain fixed in \mathbb{R}^3 , which induces kinematic closure constraints. These features are developed by chemists to encapsulate the assumption that ligand binding is due primarily to the interaction of some features of the ligand to “complementary” features of the receptor. The interacting features are included in the pharmacophore, which is a template screening candidate drugs, and the rest of the ligand atoms merely provide a scaffold for holding the pharmacophore features in their spatial positions. Figure 7.35 illustrates the pharmacophore concept.

Candidate drug molecules (ligands), such as the ones shown in Figure 3.22, can be modeled as a tree of bodies as shown in Figure 7.36. Some bonds can rotate, which yields a revolute joint in the model. Other bonds must remain fixed. The drug design problem amounts to searching the space of configurations

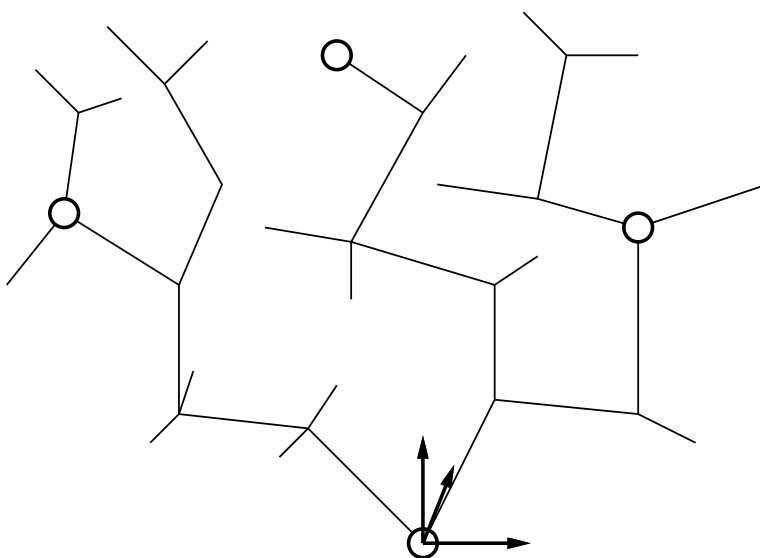


Figure 7.35: A pharmacophore is a model used by chemists to simplify the interaction process between a ligand (candidate drug molecule) and a protein. It often amounts to holding certain features of the molecule fixed in \mathbb{R}^3 . In this example, the positions of three atoms must be fixed relative to the atom to which the coordinate frame is assigned. It is assumed that these features interact with some complementary features in the cavity of the protein.

(called *conformations*) to try to find a low-energy configuration that also places certain atoms in specified locations in \mathbb{R}^3 . This additional constraint arises from the pharmacophore, and causes the planning to occur on \mathcal{C}_{clo} from Section 7.4 because the pharmacophores can be expressed as closure constraints.

An energy function serves a purpose similar that of a collision detector. The evaluation of a ligand for drug design requires determining whether it can achieve low-energy conformations that satisfy the pharmacophore constraints. Thus, the task is different from standard motion planning in that there is no predetermined goal configuration. One of the greatest difficulties is that the energy functions are extremely complicated, nonlinear, and empirical. Here is an example used in [455]:

$$\begin{aligned}
 e(q) = & \sum_{bonds} \frac{1}{2} K_b (R - R')^2 + \sum_{ang} \frac{1}{2} K_a (\alpha - \alpha')^2 + \\
 & \sum_{torsions} K_d [1 + \cos(p\theta - \theta')] + \\
 & \sum_{i,j} \left\{ 4\epsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^6 \right] + \frac{c_i c_j}{\epsilon r_{ij}} \right\}.
 \end{aligned} \tag{7.25}$$

The energy if $q \in \mathcal{C}$ accounts for torsion-angle deformations, van der Waals potential, and Coulomb potentials. In (7.25), the first sum is taken over all bonds, the second over all bond angles, the third over all rotatable bonds, and the last sum of is taken over all pairs of atoms. The variables are: 1) force constants,

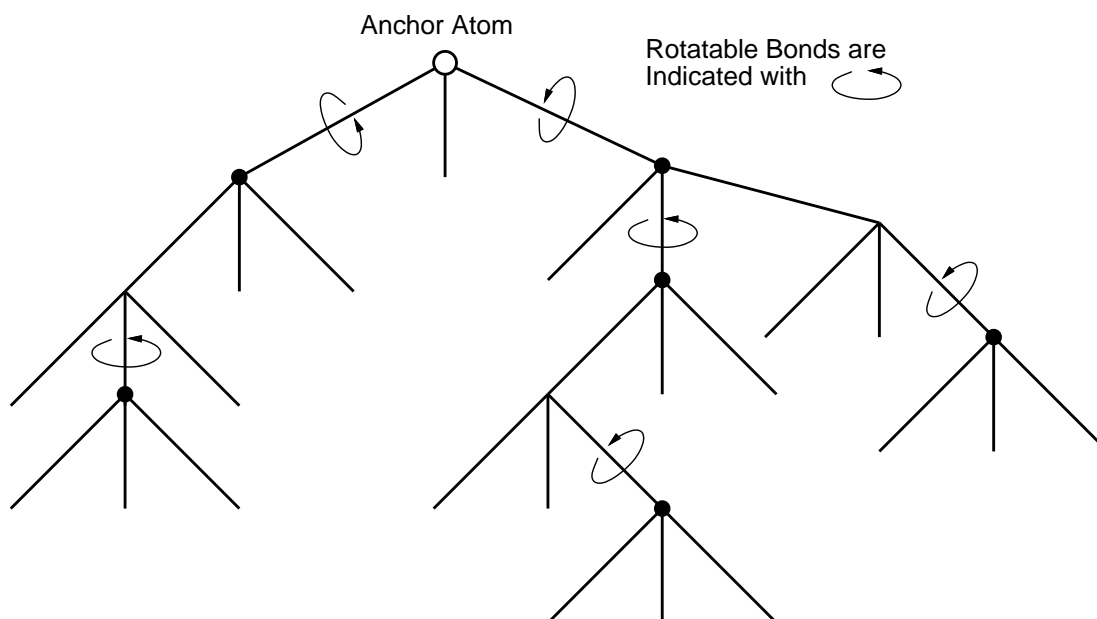


Figure 7.36: The modeling of a flexible molecule is similar to that of a robot. One atom is designated as the root, and the remaining bodies are arranged in a tree. If there are cyclic chains in the molecules, then constraints as described in Section 4.4 must be enforced. Typically, only some bonds are capable of rotation, while others must remain rigid.

$K_b, K_a,$ and K_d ; 2) the dielectric constant, ϵ ; 3) a periodicity constant, p ; 4) the Lennard-Jones radii, σ_{ij} ; 5) well depth, ϵ_{ij} ; 6) partial charge, c_i ; 7) measured bond length, R ; 8) equilibrium bond length, R' ; 9) measured bond angle, α ; 10) equilibrium bond angle, α' ; 11) measured torsional angle, θ ; 12) equilibrium torsional angle, θ' ; 13) distance between atom centers, r_{ij} . Although the energy expression is very complicated, it only depends on the configuration variables; all others are constants that are estimated in advance.

Protein folding In computational biology, the problem of protein folding shares many similarities with drug design in that the molecules have rotatable bonds and energy functions are used to express good configurations. The problems are much more complicated, however, because the protein molecules are generally much larger than drug molecules. Instead of a dozen degrees of freedom, which is typical for a drug molecule, proteins have hundreds or thousands of degrees of freedom. When proteins occur in nature, they are usually in a folded, low-energy configuration. The *structure problem* involves determining precisely how the protein is folded so that its biological activity can be completely understood. In some studies, biologists are even interested in the pathway that a protein takes to arrive in its folded state [14, 15]. This leads directly to an extension of motion planning that involves arriving at a goal state in which the molecule is folded. In [14, 15],

sampling-based planning algorithms were applied to compute folding pathways for proteins. The protein starts in an unfolded configuration and must arrive in a specified folded configuration without violating energy constraints along the way. Figure 7.37 shows an example from [15]. That work also draws interesting connections between protein folding and box folding, which was covered previously.

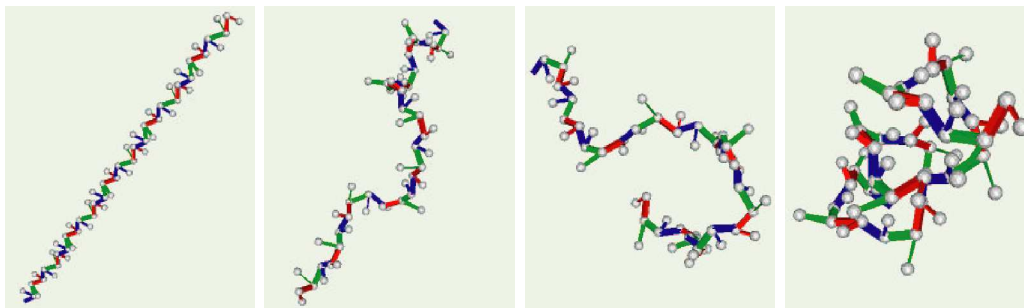


Figure 7.37: Protein folding for a polypeptide, computed by a sampling-based roadmap planning algorithm [14]

7.6 Coverage Planning

Imagine automating the motion of a lawnmower for an estate that has many obstacles, such as a house, trees, garage, and an complicated property boundary. What are the best zig-zag motions for the lawnmower? Can the amount of redundant traversals be minimized? Can the number of times the lawnmower needs to be stopped and rotated be minimized? This is one example of *coverage planning*, which is motivated by applications such as lawn mowing, automated farming, painting, vacuum cleaning, and mine sweeping. A survey of this area appears in [161]. Even for a region in $\mathcal{W} = \mathbb{R}^2$, finding an optimal-length solution to coverage planning is NP-hard, by reduction to the closely-related Traveling Salesman Problem [22, 563]. Therefore, we are willing to tolerate approximate or even heuristic solutions to the general coverage problem, even in \mathbb{R}^2 .

Boustrophedon decomposition One approach to the coverage problem is to decompose \mathcal{C}_{free} into cells, and perform boustrophedon (from Greek “ox turning”) motions in each cell as shown in Figure 7.38 [163]. It is assumed that the robot is a point in $\mathcal{W} = \mathbb{R}^2$, but it carries a *tool* of thickness ϵ that hangs evenly over the sides of the robot. This enables it to paint or mow part of \mathcal{C}_{free} up to distance $\epsilon/2$ from either side of the robot as it moves forward. Such motions are often used in printers to minimize the number of carriage returns.

If \mathcal{C}_{obs} is polygonal, a reasonable decomposition can be obtained by adapting the vertical decomposition method of Section 6.2.2. In that algorithm, critical

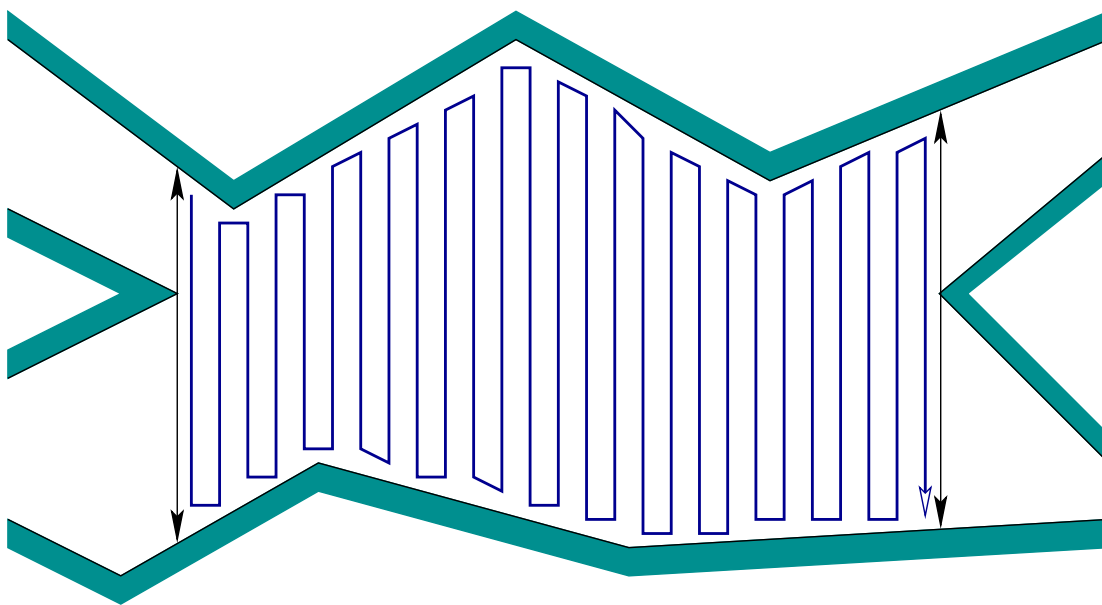


Figure 7.38: An example of the ox plowing motions.

events were defined for several cases, some of which are not relevant for the boustrophedon motions. The only events that need to be handled are shown in Figure 7.39.a [160]. This produces a decomposition that has fewer cells, as shown in Figure 7.39.b. Even though the cells are nonconvex, they can always be sliced nicely into vertical strips, which makes them suitable for boustrophedon motions. The original vertical decomposition could also be used, but the extra cell boundaries would cause unnecessary repositioning of the robot. A similar method, which furthermore optimizes the number of robot turns, is presented in [349].

Spanning tree covering An interesting approximate method can be made by placing a tiling of squares inside of \mathcal{C}_{free} , and computing the spanning tree of the resulting connectivity graph [268, 269]. Suppose again that \mathcal{C}_{free} is polygonal. Consider the example shown in Figure 7.40. The first step is to tile the interior of \mathcal{C}_{free} with squares, as shown in Figure 7.41. Each square should be of width ϵ . Next, construct a roadmap, G , by placing a vertex in the center of each square, and by defining an edge that connects the centers of each pair of adjacent cubes. The next step is to compute a *spanning tree* of G . This is a connected subgraph that has no cycles and touches every vertex of G , and can be easily computed in $O(n)$ time, if G has n edges [539]. There are many possible spanning trees, and a criterion can be defined and optimized to induce preferences. One possible spanning tree is shown Figures 7.42 and 7.43.

Once the spanning tree is made, the robot path is obtained by starting at a point near the spanning tree and following along its perimeter as shown in Figure 7.44. This path can be precisely specified as shown in Figure 7.45. Double

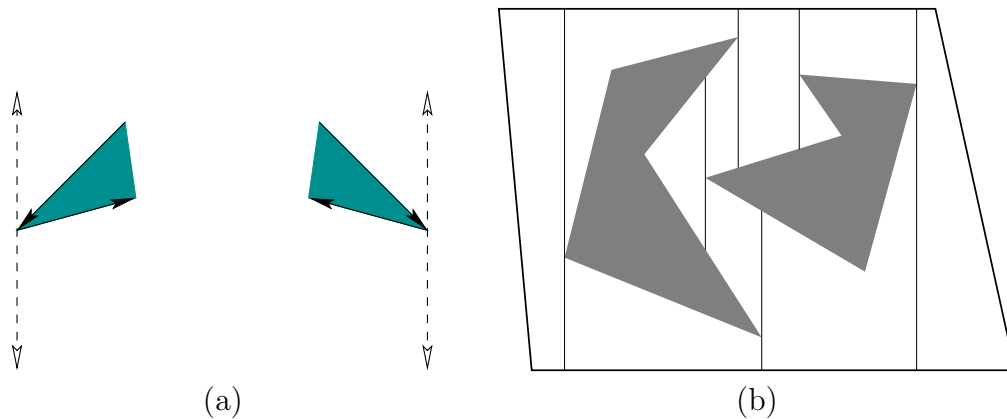


Figure 7.39: a) Only the first case from Figure 6.2 is needed: extend upward and downward. All other cases are neglected. b) The resulting decomposition is shown, which has fewer cells than that of the vertical decomposition in Figure 6.3.

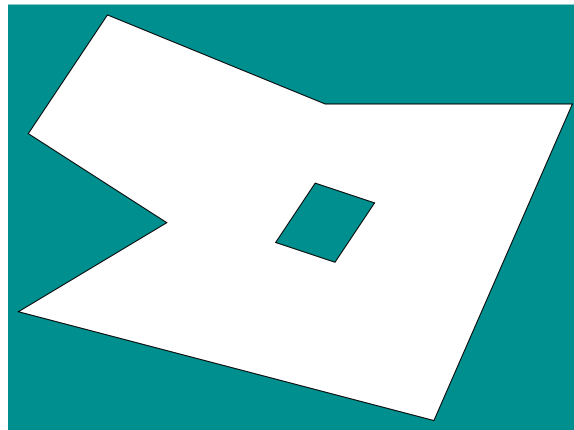


Figure 7.40: An example used for spanning tree covering.

the resolution of the tiling, and form the corresponding roadmap. Part of the roadmap will correspond to the spanning tree, but also included is a loop path that surrounds the spanning tree can be extracted. This path visits the centers of the new squares. The resulting path for the example of Figure 7.40 is shown in Figure 7.46. In general, the method yields an optimal route, once the approximation is given. A bound on uncovered area due to approximation is given in [268]. Versions of the method that do not require an initial map are also given in [268, 269]; this involves reasoning about information spaces, which are covered in Chapter 11.

7.7 Optimal Motion Planning

This section can be considered transitional in many ways. The main concern so far with motion planning has been *feasibility* as opposed to *optimality*. This placed

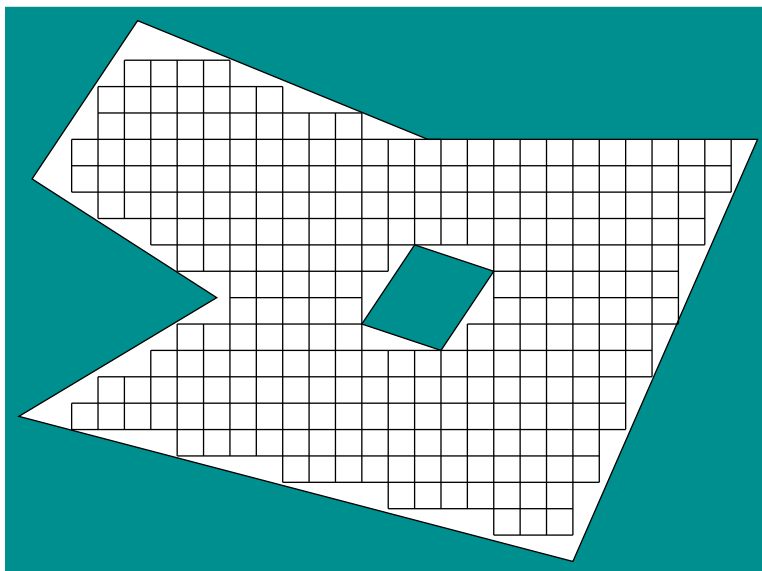


Figure 7.41: The first step is to tile the interior with squares.

the focus on finding any solution, rather than adding the additional requirement that a solution be optimal. In later parts of the book, especially as uncertainty is introduced, optimality will receive more attention. Even the most basic forms of decision theory, the topic of Chapter 9, center on making optimal choices. The requirement of optimality in very general settings usually requires an exhaustive search over the state space, which amounts to computing continuous cost-to-go functions. Once such functions are known, a feedback strategy is obtained, which is much more powerful than having only a path. Thus, optimality will also appear frequently in the design of feedback strategies because it sometimes comes at no additional cost. This will become clearer in Chapter 8. The quest for optimal solutions also raises interesting issues about how to approximate a continuous problem as a discrete problem. The interplay between time discretization and space discretization become very important in relating continuous and discrete planning problems.

7.7.1 Optimality for One Robot

Euclidean shortest paths One of the most straightforward notions of optimality is Euclidean shortest paths in \mathbb{R}^2 or \mathbb{R}^3 . Suppose that \mathcal{A} is a rigid body that translates only in either $\mathcal{W} = \mathbb{R}^2$ or $\mathcal{W} = \mathbb{R}^3$, which contains an obstacle region $\mathcal{O} \subset \mathcal{W}$. Recall that normally, \mathcal{C}_{free} , is an open set, which means that one can take any path, $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$, and make it shorter. Therefore, shortest paths for motion planning must be considered on the closure, $cl(\mathcal{C}_{free})$, which allows the robot to make contact with the obstacles; however, their interiors must not intersect.

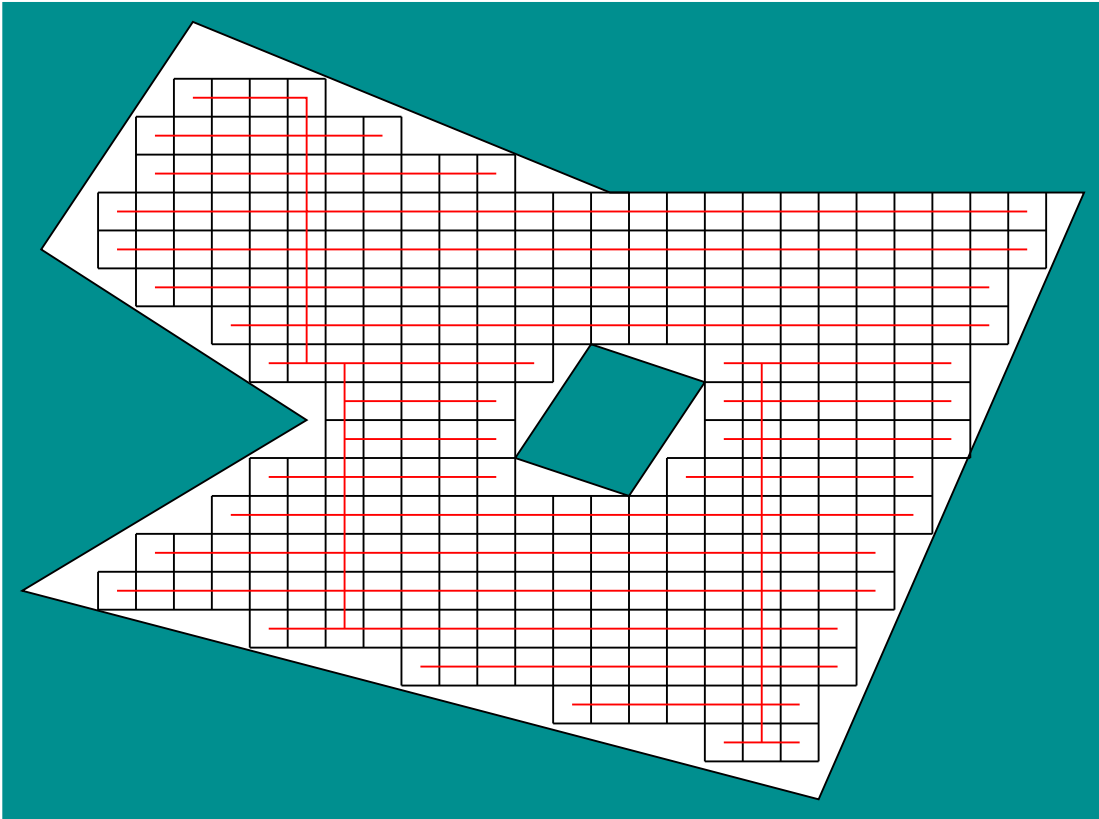


Figure 7.42: A roadmap is formed based on the grid adjacencies, followed by computation of a spanning tree.

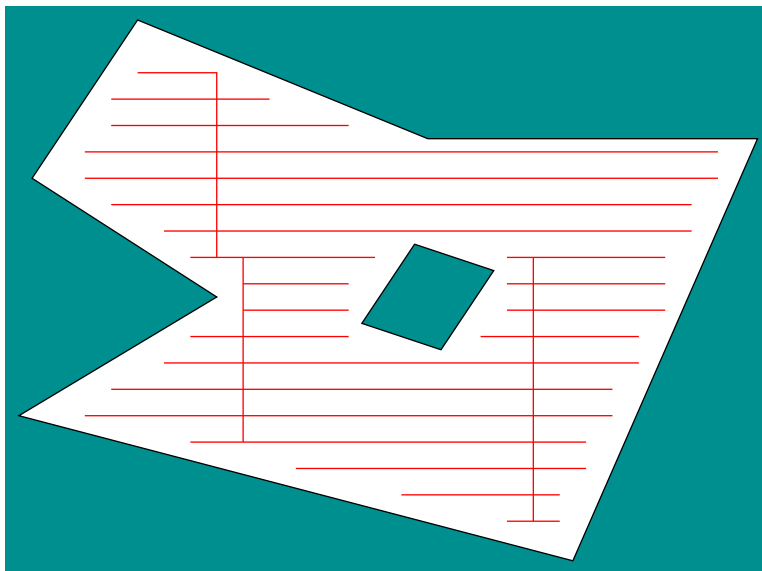


Figure 7.43: The resulting spanning tree is shown without the grid.

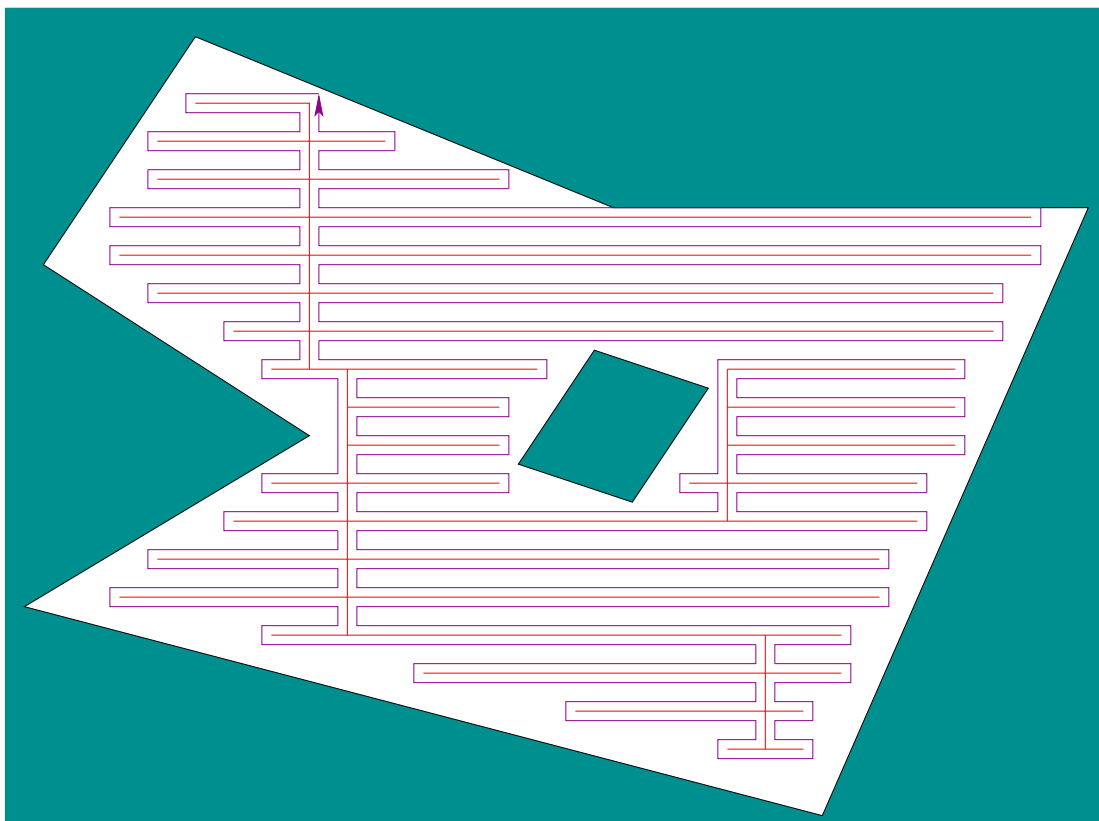


Figure 7.44: A circular path is made that follows the perimeter of the spanning tree.

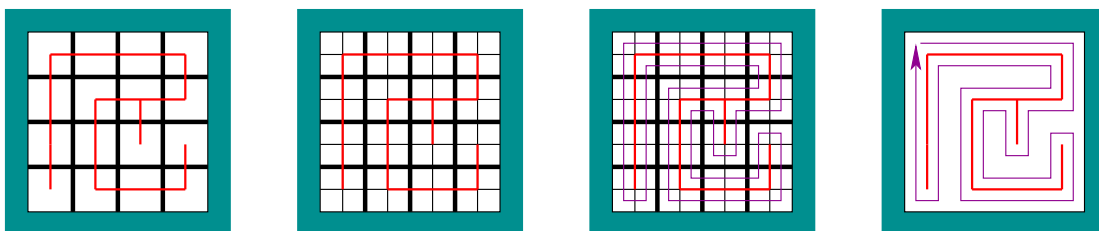


Figure 7.45: A circular path is made by doubling the resolution and following the perimeter of the spanning tree.

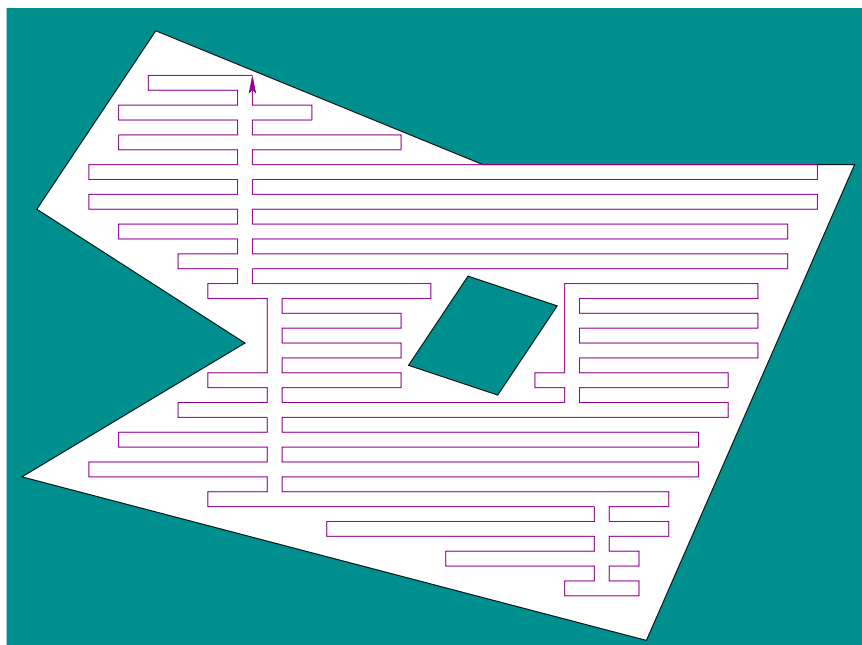


Figure 7.46: The resulting spanning tree covering for the problem in Figure 7.40.

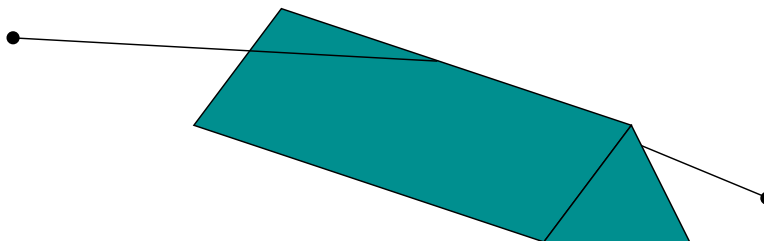


Figure 7.47: For a polyhedral environment, the shortest paths do not have to cross vertices. Therefore, the shortest path roadmap method from Section ?? does not extend to three dimensions.

For the case in which \mathcal{C}_{obs} is a polygonal region, the shortest path roadmap method of Section 6.2.4 has already been given. This can be considered as a kind of multiple-query approach because the roadmap completely captures the structure needed to construct the shortest path for any query. It is possible to make a single-query algorithm using the *continuous Dijkstra paradigm* [562, 321]. This method propagates a *wavefront* from q_i , and keeps track of critical events in maintaining the wavefront. As events occur, the wavefront becomes composed of *wavelets*, which are arcs of circles centered on obstacle vertices. The possible events that can occur are: 1) a wavelet disappears, 2) a wavelet collides with an obstacle vertex, 3) a wavelet collides with another wavelet, or 4) a wavelet collides with a point in the interior of an obstacle edge. The method can be made to run in time $O(n \lg n)$ and uses $O(n \lg n)$ space. A roadmap is constructed that uses $O(n)$ space.

Such elegant methods leave the impression that finding shortest paths is not very difficult, but unfortunately, they do not generalize nicely to \mathbb{R}^3 and a polyhedral \mathcal{C}_{obs} . Figure 7.47 shows a simple example in which the shortest path does not have to cross a vertex of \mathcal{C}_{obs} . It may cross anywhere in the interior of an edge; therefore, it is not clear where to draw the bitangent lines that would form the shortest path roadmap. The lower bounds for this problem are also discouraging. It was shown in [122] that the three-dimensional shortest path problem in a polyhedral environment is NP-hard. Most of the difficulty arises because of the precision required to represent three-dimensional shortest paths. Therefore, efficient polynomial-time approximation algorithms exist [158, 159, 606].

General optimality criteria It is difficult to even define optimality for more general configuration spaces. What does it mean to have a shortest path in $SE(2)$ or $SE(3)$? Consider the case of planar, rigid robot that can translate or rotate. One path could try to minimize amount of rotation, while another tries to minimize the amount of translation. Without more information, there is no clear choice. Ulam's distance is one possibility, which is to minimize the distance traveled by k fixed points [358]. In Chapter ??, differential models will be introduced, which greatly facilitate the natural expression of optimal paths. For example, the shortest paths for a car-like robot are shown in Section ??, but these require a precise specification of the constraints on the motion of a car (it is naturally more costly to move a car sideways than forward; hence, parallel parking is difficult).

In this section, we take some steps in this direction to formulate optimal motion planning problems, to provide a kind of smooth transition toward the later concepts. Up to now, actions were used in Chapter 2 for discrete planning problems, but were successfully avoided for basic motion planning by directly describing paths that map into \mathcal{C}_{free} . It will be convenient to use them once again. Recall that they were convenient for defining costs and optimal planning in Section 2.4.

To avoid for now the complications of differential equations, consider making an approximate model of motion planning in which every path must be composed of a sequence of shortest-path segments in \mathcal{C}_{free} . Most often these will be line segments; however, for the case of $SO(3)$, circular segments obtained by spherical linear interpolation may be preferable. Consider extending Formulation 2.4.2 from Section 2.4.2 to the problem of motion planning.

Let the configuration space, \mathcal{C} be embedded in \mathbb{R}^m (i.e. $\mathcal{C} \subset \mathbb{R}^m$). An action will be defined shortly as an m -dimensional vector. Given a scaling constant, ϵ and a configuration, q , an action, u , will produce a new configuration, $q' = q + \epsilon u$. This can be considered as a *configuration transition equation*, $q' = f(q, u)$. The path segment represented by the action u is the shortest path (usually a line segment) between q and q' . Following Section 2.4, let π_K denote a K -step plan, which is a sequence (u_1, u_2, \dots, u_K) of K actions. Note that if π_K and q_i are given, then a sequence of states, q_1, q_2, \dots, q_{K+1} , can be derived using the state transition equation, f . Initially, $q_1 = q_i$, and each following state is obtained by

$q_{k+1} = f(q_k, u_k)$. This also leads to a path $[0, 1] \rightarrow \mathcal{C}$.

The approximate optimal planning problem is now formalized as follows:

Formulation 7.7.1 (Approximate Optimal Motion Planning)

1. A number, K , of *stages*. The current state, k , is indicated by a subscript, to obtain q_k and u_k .
2. The following components are defined the same as in Model 4.3.1: \mathcal{W} , \mathcal{O} , \mathcal{A} , \mathcal{C} , \mathcal{C}_{obs} , \mathcal{C}_{free} , and q_i . It is assumed that $\mathcal{C} \subseteq \mathbb{R}^m$, for some positive integer m .
3. For each $q \in \mathcal{C}$, a possibly-infinite *action space*, $U(q)$. Each $u \in U$ is an m -dimensional vector.
4. A positive constant, $\epsilon > 0$, called the *step size*.
5. A *configuration transition function*, $f(q, u) = q + \epsilon u$, in which $q + \epsilon u$ is computed by vector addition on \mathbb{R}^m .
6. Instead of a goal state, a goal region, X_G is defined.
7. Let L denote a real-valued, additive cost (or loss) functional, which is applied to a K -step plan, π_K . This means that the sequence, (u_1, \dots, u_K) , of actions and the sequence, (q_1, \dots, q_{K+1}) , of configurations may appear in an expression of L . For convenience, let $F \equiv K + 1$, to denote the *final state* (note that the application of u_K advances the stage to $K + 1$). The *cost functional* is

$$L(\pi_K) = \sum_{k=1}^K l(q_k, u_k) + l_F(q_F). \quad (7.26)$$

The final term, $l_F(q_F)$, is outside of the sum, and is defined as $l_F(q_F) = 0$ if $q_F \in X_G$, and $l_F(q_F) = \infty$, otherwise. Just as in Formulation 2.4.2, it is assumed that K is not necessarily a constant.

8. Each $U(q)$ contains a special *termination action*, u_T , which behaves the same way as in Formulation 2.4.2. If u_T is applied to q_k , at stage k , then the action is repeatedly applied forever, the configuration remains in q_k forever, and no more cost accumulates.

Formulation 7.7.1 can be used to define a variety of optimal planning problems. The parameter ϵ can be considered as the resolution of the approximation. In many formulations it can be interpreted as a *time step*, $\epsilon = \Delta t$; however, note that no explicit time reference is necessary because the problem only requires constructing a path through \mathcal{C}_{free} . As ϵ approaches zero, the formulation approaches an exact optimal planning problem. To properly express the exact problem, differential equations are needed. This is deferred until Section ??.

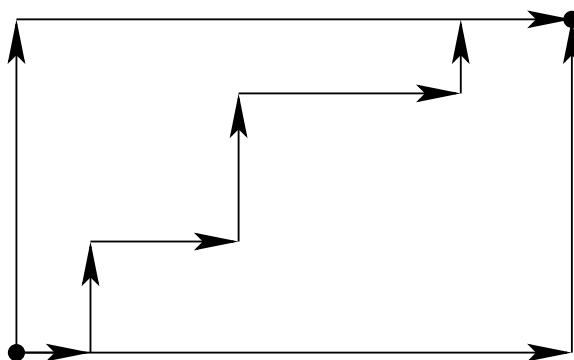


Figure 7.48: Under the Manhattan (L_1) motion model, all monotonic paths that follow the grid directions have equivalent length.

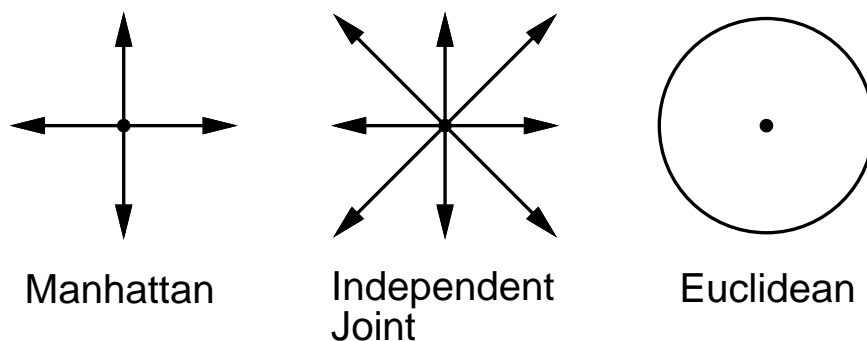


Figure 7.49: Depictions for the actions sets, $U(q)$ for Examples 7.7.1, 7.7.2, and 7.7.3.

Example 7.7.1 (Manhattan Motion Model) Suppose that the $U(q)$ is defined as a set of $2m$ vectors in which only one component is nonzero and must take the value 1 or -1 . For example, if $m = 2$, then

$$U(q) = \{(1, 0), (-1, 0), (0, -1), (0, 1)\}. \quad (7.27)$$

When used in the configuration transition equation, this set of actions produces “up”, “down”, “left”, and “right” motions. The action set for this example and the following two examples are shown in Figure ?? for comparison. The loss $l(q_k, u_k)$ is defined to be 1 for all $q_k \in \mathcal{C}_{free}$ and u_k . If $q_k \in \mathcal{C}_{obs}$, then $l(q_k, u_k) = \infty$. Note that the set of configurations reachable by these actions will lie on a grid, in which the spacing between 1-neighbors is ϵ . This corresponds to a convenient special case in which time-discretization (implemented by ϵ) leads to a nice space-discretization. Consider Figure 7.48. It is impossible to take a shorter path along a diagonal because the actions do not allow it. Therefore, all monotonic paths along the grid produce the same costs.

Optimal paths can be obtained by simply applying the dynamic programming algorithms of Chapter 2. This example provides a nice unification of concepts from

Section 2.3, which introduced grid search, and Section 5.4.2, which explained how to adapt search methods to motion planning. In the current setting, only algorithms that produce optimal solutions on the corresponding graph are acceptable.

This form of optimization might not seem to relevant since it does not represent the Euclidean shortest path problem for \mathbb{R}^2 . This next model adds more actions, and does correspond to an important class of optimization problems in robotics. ■

Example 7.7.2 (Independent Joint Model) Now suppose that $U(q)$ is the set of all 3^m vectors for which every element is either -1 , 0 , or 1 . Now a path can be taken along any diagonal. This still does not change the fact that all reachable configurations lie on a grid. Therefore, the standard grid algorithms can once again be applied. The difference is that now there are now $3^n - 1$ edges emanating from every vertex, as opposed to $2n$ in Example 7.7.1. This model is appropriate for robots that are constructed from a collection of links attached by revolute joints. If each joint is operated independently, then it makes sense that each joint could either be moved forward, moved backwards, or held stationary. This corresponds exactly to the actions. However, this model cannot nicely approximate Euclidean shortest paths; this motivates the next example. ■

Example 7.7.3 (Euclidean Motion Model) To approximate Euclidean shortest paths, let $U(q) = \mathbb{S}^{m-1}$, in which \mathbb{S}^{m-1} is the m -dimensional unit sphere, centered at the origin of \mathbb{R}^m . This means that in k stages, any piecewise-linear path in which each segment has length ϵ can be formed by a sequence of inputs. Therefore, the set of reachable states is no longer confined to a grid. Consider taking $\epsilon = 1$, and pick any point, such as $(\pi, \pi) \in \mathbb{R}^2$. How close can you come to this point? It turns out that the set of points reachable with this model is dense in \mathbb{R}^m if obstacles are neglected. This means that we can come arbitrarily close to any point in \mathbb{R}^m . Therefore, a finite grid cannot be used to represent the problem. Approximate solutions can still be obtained by computing a numerical approximation to an optimal cost-to-go defined over \mathcal{C} . This approach will be presented in Section ??.

One additional issue for this problem is the precision defined for the goal. If the goal region is very small relative to ϵ , then complicated paths may have to be taken to arrive precisely at the goal. ■

Example 7.7.4 (Weighted Region Problem) In outdoor and planetary navigation applications, it does not make sense to define obstacles in the crisp way that has been used so far. It is more convenient to associate a cost with each patch of terrain, which indicates the estimated difficulty of traversal. This is

sometimes considered as a “gray scale” model of obstacles. The model can be easily captured in the cost term $l(q_k, u_k)$. The action spaces can be borrowed from Examples 7.7.1 or 7.7.2. A grid-based search algorithm called D^* is introduced in [?] which generates optimal navigation plans for this problem, assuming that the terrain is initially unknown. Theoretical bounds for optimal weighted-region planning problems are given in [563]. ■

7.7.2 Multiple-Robot Optimality

Suppose that there are two robots as shown in Figure 7.50. There is just enough room to enable the robots to translate along the corridors. Each will like to arrive at the bottom, as indicated by arrows; however, only one at a time can pass through the horizontal corridor. Suppose that at any instant each robot can either be *on* or *off*. When it is *on*, it moves at its maximum speed, and when it is *off*, it is stopped.³ Now suppose that each robot would like to reach its goal as quickly as possible. This means each would like to minimize the total amount of time that it is *off*. In this example, there appears to be only two sensible choices: 1) \mathcal{A}_1 stays *on* and moves straight to its goal while \mathcal{A}_2 is *off* just long enough to let \mathcal{A}_1 pass, and then moves to its goal. 2) The opposite situation occurs, in which \mathcal{A}_2 stays *on* and \mathcal{A}_1 must wait. Note that when a robot waits, there are multiple locations at which it could wait and still yield the same time to reach the goal. The only important information is how long the robot was *off*.

Thus, the two intersecting strategies are that either \mathcal{A}_2 is *off* for some amount of time, $t_{off} > 0$, or \mathcal{A}_1 is *off* for time t_{off} . Consider a vector of costs of the form (L^1, L^2) , in which each component represents the cost for each robot. The costs of the strategies could be measured in terms of time wasted by waiting. This yields $(0, t_{off})$ and $(t_{off}, 0)$ for the cost vectors associated with the two strategies (we could equivalently define cost to be the total time traveled by each robot; the time on is the same for both robots and can be subtracted from each for this simple example). The two strategies are better than or equivalent to any others. Strategies with this property are called *nondominated* or *Pareto optimal*. For example, if \mathcal{A}_2 waits 1 second too long for \mathcal{A}_1 to pass, then the resulting costs are $(0, t_{off} + 1)$, which is clearly worse than $(0, t_{off})$. The resulting strategy is not Pareto optimal.

Another way to solve the problem is to scalarize the costs by mapping them to a single value. For example, we could find strategies that optimize the average wasted time. In this case, one of the two best strategies would be obtained, yielding t_{off} average wasted time. However, no information is retained about which robot had to make the sacrifice. Scalarizing the costs usually imposes some kind of artificial preference or prioritization among the robots. Ultimately, only

³This model allows infinite acceleration. Imagine that the speeds are slow enough to allow this approximation. If this is still not satisfactory, then jump ahead to Chapter 13.

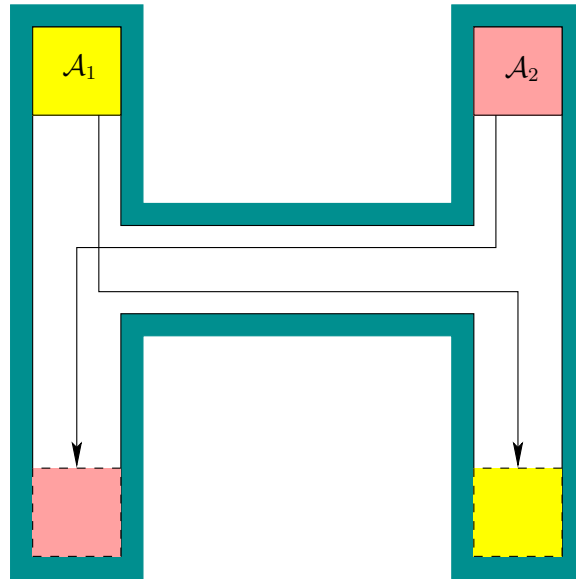


Figure 7.50: There are two Pareto-optimal coordination strategies for this problem, depending on which robot has to wait.

one strategy can be chosen, which might make it seem inappropriate to maintain multiple solutions. However, finding and presenting the alternative Pareto optimal solutions could provide valuable information if, for example, these robots are involved in a complicated application that involves many other time-dependent processes. Presenting the Pareto optimal solutions is equivalent to discarding all of the worse strategies, and showing the best alternatives. In some applications, priorities between robots may change, and if a scheduler of robots has access to the Pareto optimal solutions, it is easy to change priorities by switching between Pareto optimal strategies without having to generate new plans each time.

Now the Pareto optimality concept will be made more precise and general. Suppose there are m robots, $\mathcal{A}^1, \dots, \mathcal{A}^m$. Let γ refer a motion strategy that gives the paths and timing functions for all robots. For \mathcal{A}^i , let \mathcal{L}^i denote its cost-functional, which yields a value $L^i(\gamma) \in [0, \infty]$ for a given strategy, γ . An m -dimensional vector, $L(\gamma)$, is defined as

$$L(\gamma) = (L^1(\gamma), L^2(\gamma), \dots, L^m(\gamma)). \quad (7.28)$$

Two strategies, γ and γ' are called *equivalent* if $L(\gamma) = L(\gamma')$. A strategy γ is said to *dominate* a strategy γ' if they are not equivalent and $L_i(\gamma) \leq L_i(\gamma')$ for all i such that $1 \leq i \leq m$. A strategy is called *Pareto optimal* if it is not dominated by any others. Since many Pareto-optimal strategies may be equivalent, the task is to determine one representative from each equivalence class. This will be called finding the *unique* Pareto-optimal strategies. For the example in Figure 7.50, there are two unique Pareto-optimal strategies, which were already given.

Scalarization For the motion planning problem, a Pareto-optimal solution is also optimal for a scalar cost functional that is constructed as a linear combination of the individual costs. Let $\alpha_1, \dots, \alpha_m$ be positive real constants. Let

$$l(\gamma) = \sum_{i=1}^m \alpha_i L^i(\gamma). \quad (7.29)$$

It can be shown that any strategy that is optimal with respect to (7.29) is also a Pareto-optimal solution [461]. If a Pareto optimal solution is generated this way, however, there is no easy way to determine what alternatives exist.

Computing Pareto-optimal strategies Since optimization for one robot is already very difficult, it may not be surprising that computing Pareto-optimal strategies is even harder. For some problems, it is even possible that a continuum of Pareto-optimal solutions exist [], which is very discouraging. Fortunately, for the problem of coordinating robots on topological graphs, as considered in Section 7.2.2, there is only a finite number of solutions. An efficient grid-based algorithm, which based on dynamic programming and computes all unique Pareto-optimal coordination strategies is presented in [461]. For special cases that involve polygonal robots moving on a tree of piecewise-linear paths, complete algorithms are presented in [].

Literature

Ref. modular robotics, and also Ghrist's work (esp. Ghrist, Abrams).

A survey of coverage planning appears in [161]

Extensions of the spanning tree method, especially for on-line problems: [270].

Earlier work on this topic [586].

Give more computational biology references. There are many problems that are loosely related to motion planning.

Exercises

1. To yield polyhedral obstacles for time-varying motion planning, what is the general form for which linear geometric primitives H_i that define \mathcal{O} can be transformed? To yield semi-algebraic models?
2. Give a method for computing the obstacle region for two translating polygonal robots that follow a linear path.
3. Construct the cube complex for some examples...
4. Try numerical continuation a surface in \mathbb{R}^3 .

Chapter 8

Feedback Motion Strategies

Chapter Status



What does this mean? Check <http://msl.cs.uiuc.edu/planning/status.html> for information on the latest version.

Up to now, it has been assumed that the robot motions are completely predictable. In many applications, this assumption is too strong. A collision-free path might be insufficient as a representation of a motion strategy. This chapter addresses the problem of computing a motion strategy that uses feedback. During execution, the action taken by the robot will depend only on the measured configuration or state.

8.1 Feedback in Discrete Planning

8.2 Vector Fields on Manifolds

8.3 Feedback Strategies in Motion Planning

If a path is insufficient, what form should a motion strategy take? Suppose that a world with a robot and obstacles is defined. This leads to the definition of configuration space, \mathcal{C} , and its collision-free subset \mathcal{C}_{free} . Suppose the goal configuration is q_{goal} . We might also consider a goal region \mathcal{C}_{goal} .

One possible representation of a feedback motion strategy is a velocity field, \vec{V} over \mathcal{C} . At any $q \in \mathcal{C}$, a velocity vector $\vec{V}(q)$ is given, which indicates the how the configuration should change. A successful motion strategy is one in which the velocity field, when integrated from any initial configuration, lead to a configuration in \mathcal{C}_{goal} .

For problems in Section 15, a feedback motion strategy can take the form of a function $\mathcal{C} \rightarrow U$, in which U is the set of inputs, applied in the state transition equation, $\dot{x} = f(x, u)$. However, nonholonomic feedback motion strategies will not be considered in this chapter.

vector field characterization

action bundle characterization

8.3.1 Navigation Functions

potential function, navigation function characterization

Connect this explanation back to cost-to-go functions from Chapter 2.

One convenient way to generate a velocity field is through the gradient of a scalar-valued function. Let $E : \mathcal{C} \rightarrow \mathbb{R}$ denote a real-valued, differentiable *potential function*. Using E , a feedback motion strategy can be defined as $\vec{V} = -\nabla E$, in which ∇ denotes the gradient. If designed appropriately, the potential function can be viewed as a kind of “ski slope” that guides the robot to a specified goal.

As a simple example, suppose $\mathcal{C} = \mathbb{R}^2$, and that there are no obstacles. Let (x, y) denote a configuration. Suppose that the goal $q_{goal} = (0, 0)$. A quadratic function $E(x, y) = x^2 + y^2$ serves as a good potential function to guide the configuration to the goal. The feedback motion strategy is defined as $\vec{V} = -\nabla E = [-2x \quad -2y]$.

If the goal is at any (x_0, y_0) , then a potential function that guides the configuration to the goal is $E(x, y) = (x - x_0)^2 + (y - y_0)^2$.

Suppose the configuration space contains point obstacles. The previous function E can be considered as an attractive potential because the configuration is attracted to the goal. One can also construct a repulsive potential that repels the configuration from the obstacles to avoid collision. If E_a denotes the attractive component, and E_r denotes the repulsive potential, then a potential function of the form $E = E_a + E_r$ can be defined to combine both effects. The robot should be guided to the goal while avoiding obstacles. The problem is that there is no way in general to insure that the potential function will not contain multiple local minima. The configuration could become trapped at a local minimum that is not in the goal region.

Rimon and Koditschek [659] presented a method for designing potential functions that contain only one minimum, which is precisely at the goal. These special potential functions are called *navigation functions*. Unfortunately, the technique applies only when \mathcal{C}_{free} is of a special form. In general, there are no known ways to efficiently compute a potential function that contains only one local minimum which is at the goal. This is not surprising given the difficulty of the basic path planning problem.

cost-to-go functions

8.4 Combinatorial Algorithms

8.4.1 Harmonic Functions

8.4.2 Feedback Strategies over Complexes

Cell based conversions

Star-shaped regions: Rimon and Koditschek

8.5 Sampling-Based Algorithms

8.5.1 Sampling-Based Neighborhood Graph

This is pasted from a paper of ours; it definitely needs to be substantially shortened and written from a more general perspective

Sampling-based techniques can be used to compute a navigation function (a potential function with one local minimum, which is at the goal) over most of \mathcal{C}_{free} . One method presented here is called the Sampling-Based Neighborhood Graph (SNG).

A Sampling-Based Neighborhood Graph (SNG) is an undirected, graph, $G = (V, E)$, in which V is the set of vertices and E is the set of edges. Each vertex represents an n -dimensional neighborhood that lies entirely in \mathcal{C}_{free} . In this paper, an n -dimension ball is used. For any vertex, v , let c_v denote the center of its corresponding ball, r_v denote the radius of its ball, and let B_v be the set of points, $B_v = \{q \in \mathcal{C} \mid \|q - c_v\| \leq r_v\}$. We require that $B_v \subset \mathcal{C}_{free}$. The definition of B_v assumes that \mathcal{C} is an n -dimensional Euclidean space; however, minor modifications can be made to include other frequently-occurring topologies, such as $\mathbb{R}^2 \times S^1$ and $\mathbb{R}^3 \times P^3$.

An edge, $e \in E$, exists for each pair of vertices, v_i , and v_j , if and only if their balls intersect, $B_i \cap B_j \neq \emptyset$. Assume that no balls are contained within another ball, $B_i \not\subset B_j$, for all v_i and v_j in V . Let \mathcal{B} represent the subset of \mathcal{C}_{free} that is occupied by balls, $\mathcal{B} = \bigcup_{v \in V} B_v$. Suppose that the graph G has been given; an algorithm that constructs G is presented in Section 8.5.1. For a given goal, the SNG will be used to represent a feedback strategy, which can be encoded as a real-valued navigation function, $\gamma : \mathcal{B} \rightarrow \mathbb{R}$. This function will have only one minimum, which is at the goal configuration. If the goal changes, it will also be possible to quickly “reconfigure” the SNG to obtain a new function, γ' , which has its unique minimum at the new goal.

Let G be a weighted graph in which $l(e)$ denotes the cost assigned to an edge $e \in E$. Assume that $1 \leq l(e) < \infty$ for all $e \in E$. The particular assignment of costs can be used to induce certain preferences on the type of solution (e.g., maximize clearance, minimize distance traveled). Let B_{v_g} denote any ball that contains the goal, q_{goal} , and let v_g be its corresponding vertex in G . Let $L^*(v)$ denote be the optimal cost in G to reach v_g from v . The optimal costs can be recomputed

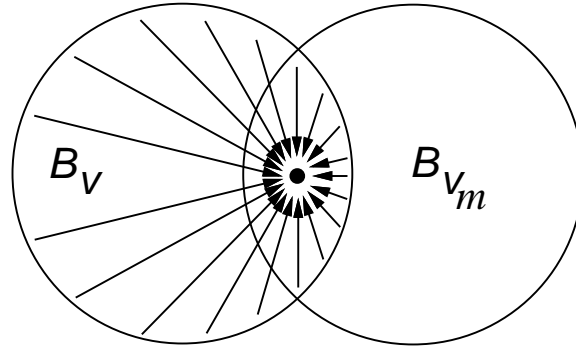


Figure 8.1: The negative gradient of a partial navigation function sends the robot to a lower-cost ball.

for each vertex in V in $O(V^2)$ or $O(V \lg V + E)$ time using Dijkstra’s algorithm; alternatively, an all-pairs shortest paths algorithm can be used to implicitly define solutions for all goals in advance.

Assume that G is connected; if G is not connected, then the following discussion can be adapted to the connected component that contains v_g . Define a strict linear ordering, $<_v$, over the set of vertices in V using $L^*(v)$ as follows. If $L^*(v_1) < L^*(v_2)$ for any $v_1, v_2 \in V$, then $v_1 <_v v_2$. If $L^*(v_1) = L^*(v_2)$, then the ordering of v_1 and v_2 can be defined in an arbitrary way, while ensuring that $<_v$ remains a linear ordering. The ordering $<_v$ can be adapted directly to the set of corresponding balls to obtain an ordering $<_b$ such that: $B_{v_1} <_b B_{v_2}$ if and only if $v_1 <_v v_2$. Note that the smallest element with respect to $<_b$ always contains the goal.

For a given goal, the SNG will be used to represent a mapping $\gamma : \mathcal{B} \rightarrow \mathbb{R}$ that serves as a global potential or navigation function. For each vertex, $v \in V$, let $\gamma_v : B_v \rightarrow \mathbb{R}$ represent a *partial strategy*. Among all balls that intersect B_v , let B_{v_m} denote the ball that is minimal with respect to $<_b$. It is assumed that γ_v is a differentiable function that attains a unique minimum a point in the interior of $B_v \cap B_{v_m}$. Intuitively, each partial strategy guides the robot to a ball that has lower cost.

The partial strategies are combined to yield a global strategy in the following way. Any configuration, $q \in \mathcal{B}$, will generally be contained in multiple balls. Among these balls, let B_v be the minimal ball with respect to $<_b$ that contains q . The navigation function at q is given by $\gamma_v(q)$, thus resolving any ambiguity. Note that the robot will typically not reach the minimum of a partial strategy before “jumping” to a ball that is lower with respect to $<_b$.

SNG Construction Algorithm An outline of the SNG construction algorithm follows:

GENERATE_SNG(α, P_c)

```

1  G.init( $q_{init}$ );
2  while (TerminationUnsatisfied( $G, \alpha, P_c$ )) do
3    repeat
4       $q_{new} \leftarrow \alpha(i)$ ;
5       $d \leftarrow \text{DistanceComputation}(q_{new})$ ;
6      until (( $d > 0$ ) and ( $q_{new} \notin \mathcal{B}$ ))
7       $r \leftarrow \text{ComputeRadius}(d)$ ;
8       $v_{new} \leftarrow G.\text{AddVertex}(q_{new}, r)$ ;
9      G.AddEdges( $v_{new}$ );
10 G.DeleteEnclaves();
11 G.DeleteSingletons();
12 Return G

```

The inputs are $\alpha \in (0, 1)$ and $P_c \in (0, 1)$ (the obstacle and robot models are implicitly assumed). For a given α and P_c , the algorithm will construct an SNG such that with probability P_c , the ratio of the volume of \mathcal{B} to the volume of \mathcal{C}_{free} is at least α .

Each execution of Lines 3-9 corresponds to the addition of a new ball, $B_{v_{new}}$, to the SNG. This results in a new vertex in G , and new edges that each corresponds to another ball that intersects $\mathcal{B}_{v_{new}}$. Balls are added to the SNG until the Bayesian termination condition is met, causing TerminationUnsatisfied to return FALSE. The Bayesian method used in the termination condition is presented in Section 8.5.1. The **repeat** loop from Lines 3 to 6 generates a new sample in $\mathcal{C}_{free} \setminus \mathcal{B}$, which might require multiple iterations. Collision detection and distance computation are performed in Line 5. Many algorithms exist that either exactly compute or compute a lower bound on the closest distance in \mathcal{W} between \mathcal{A} and \mathcal{O} [492, 556, 639], $d(q_{new}) = \min_{a \in \mathcal{A}(q_{new})} \min_{o \in \mathcal{O}} \|a - o\|$. If d is not positive, then q_{new} is in collision, and another configuration is chosen. The new configuration must also not be already covered by the SNG before the *repeat* loop terminates. This forces the SNG to quickly expand into \mathcal{C}_{free} , and leads to few edges per vertex in G .

Distance computation algorithms are very efficient in practice, and their existence is essential to our approach. The distance, d , is used in Line 7 by the ComputeRadius function, which attempts to select r to create the largest possible ball that is centered at q_{new} and lies entirely in \mathcal{C}_{free} . A general technique for choosing r is presented in Section 8.5.1.

The number of iterations in the **while** loop depends on the Bayesian termination condition, which in turn depends on the outcome of sampled events during execution and the particular \mathcal{C}_{free} for a given problem. The largest two computational expenses arise from the distance computation and the test whether q_{new} lies in \mathcal{B} . Efficient algorithms exist for both of these problems.

Radius selection For a given q_{new} , the task is to select the largest radius, r , such that the ball $B_v = \{q \in \mathcal{C} \mid \|q_{new} - q\| \leq r\}$ is a subset of \mathcal{C}_{free} . If DistanceComputation(q_{new}) returns d , then $\max_{a \in \mathcal{A}} \|a(q_{new}) - a(q)\| < d$ for all

$q \in B_v$ implies that $B_v \subset \mathcal{C}_{free}$. For many robots one can determine a point, a_f , in \mathcal{A} that moves the furthest as the configuration varies. For a rigid robot, this is the point that would have the largest radius if polar or spherical coordinates are used to represent \mathcal{A} . The goal is to make r as large as possible to make the SNG construction algorithm more efficient. The largest value of r is greatly affected by the parameterization of the kinematics. For example, if a_f is far from the origin, points on the robot will move very quickly as the rotation angle changes.

Although many alternatives are possible, one general methodology for selecting r for various robots and configuration spaces is to design a parameterization by bounding the arc length. Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ denote the expression of the kinematics that maps points from an n -dimensional configuration space to an m D world. In general, arc length in the world, based on differential changes in configuration, is specified by a metric tensor. If the transformation f is orthogonal, the arc length is

$$\sqrt{\int ds^2} = \sqrt{\int \sum_{i=1}^n \left\| \frac{\partial f(x_1, \dots, x_m)}{\partial q_i} \right\|^2 dq_i^2}, \quad (8.1)$$

in which each term represents the squared magnitude of a column in the Jacobian of f . Using the bound $\sqrt{\int ds^2} < d$, (8.1) expresses the equation of a solid ellipsoid in the configuration space. Obviously, that solid ellipsoid will be significantly different according to different kinematic expressions. The key is to choose kinematic expressions that keep the eccentricity as close as possible to representing a sphere.

For a 2D rigid robot with translation and rotation, $\mathcal{C} = \mathbb{R}^2 \times S^1$, let $r_m = \|a_f(0)\|$. If the standard parameterization of rotation was used, the effects of rotation would dominate, resulting in a smaller radius, $r = d/r_m$. But if a scaled rotation, $q_3 = r_m\theta$, is used, (8.1) will yield that $r = d$, which is a sphere. Although the relative fraction of S^1 that is covered is the same in either case, the amount of \mathbb{R}^2 that is covered is increased substantially. For a 3D rigid robot with translation and rotation, $\mathcal{C} = \mathbb{R}^3 \times P^3$, the same result can be obtained if roll, pitch, and yaw are used to represent rotation. The reason for not using quaternions is because (8.1) will not yield a simple expression. For problems that involve articulated bodies, it is preferable to derive expressions that consider the distance in the world of each rigid body.

A Bayesian termination condition The above algorithm decides to terminate based on a statistical estimate of the fraction of \mathcal{C}_{free} that is covered by the SNG. The volumes of \mathcal{C}_{free} and \mathcal{B} , denoted by $\mu(\mathcal{C}_{free})$ and $\mu(\mathcal{B})$ are assumed unknown. Although it is theoretically possible to incrementally compute $\mu(\mathcal{B})$, it is generally too complicated. A Bayesian termination condition can be derived based on the number of samples that fall into \mathcal{B} , as opposed to $\mathcal{C}_{free} \setminus \mathcal{B}$. For a given α and P_c , the algorithm will terminate when 100 α percent of the volume of \mathcal{C}_{free} has been covered by the SNG with probability P_c .

Let $p(x)$ represent a probability density function that corresponds to the fraction $\mu(\mathcal{B})/\mu(\mathcal{C}_{free})$. Let y_1, y_2, \dots, y_k represent a series of k observations, each of which corresponds for a random configuration, drawn drawn from \mathcal{C}_{free} . Each observation has two possible values: either the random configuration, q_{new} , is in \mathcal{B} or in $\mathcal{C}_{free} \setminus \mathcal{B}$. Let $y_k = 1$ denote $q_{new} \in \mathcal{B}$, and let $y_k = 0$ denote $q_{new} \in \mathcal{C}_{free} \setminus \mathcal{B}$.

For a given α and P_c , we would like to determine whether $P[x > \alpha] \geq P_c$. Assume that the prior $p(x)$ is a uniform density over $[0, 1]$. By iteratively applying Bayes' rule, for a chain of k successive samples we have $P[x > \alpha] = 1 - \alpha^{k+1}$.

The algorithm terminates when the number of successive samples that lie in \mathcal{B} is k , such that $\alpha^{k+1} \leq 1 - P_c$. One can solve for k and the algorithm will terminate when $k = \frac{\ln(1-P_c)}{\ln \alpha} - 1$. During execution, a simple counter records the number of consecutive samples that fall into \mathcal{B} (ignoring samples that fall outside of \mathcal{C}_{free}).

Some Computed Examples Figure 8.2.a shows the balls of the SNG for a point robot in a 2D environment. Figure 8.2.b shows the SNG edges as line segments between ball centers. The SNG construction required 23s, and the algorithm terminated after 500 successive failures ($k = 500$) to place a new ball. The SNG contained 535 nodes, 525 of which are in a single connected component. There were 1854 edges, resulting in an average of only 3.46 edges per vertex. We have observed that this number remains low, even for higher-dimensional problems. This is an important feature for maintaining efficiency because of the graph search operations that are needed to build navigation functions.

Figures 8.2.c and 8.2.d show level sets of two different potential functions that were quickly computed for two different goal (each in less than 10ms). The first goal is in the largest ball, and the second goal is in the upper right corner. Each ball will guide the robot into another ball, which is one step closer to the goal. Using this representation, the particular path taken by the robot during execution is not critical. For higher-dimensional configuration spaces, we only show robot trajectories, even though much more information is contained in the SNG.

8.6 Computing Optimal Feedback Strategies

Numerical dynamic programming: refer back to last section of Chapter 7, and Section 2.4.

Give standard iterative technique

Then give Dijkstra-like algorithm

Maybe also give wavefront propagation (Dial's alg)

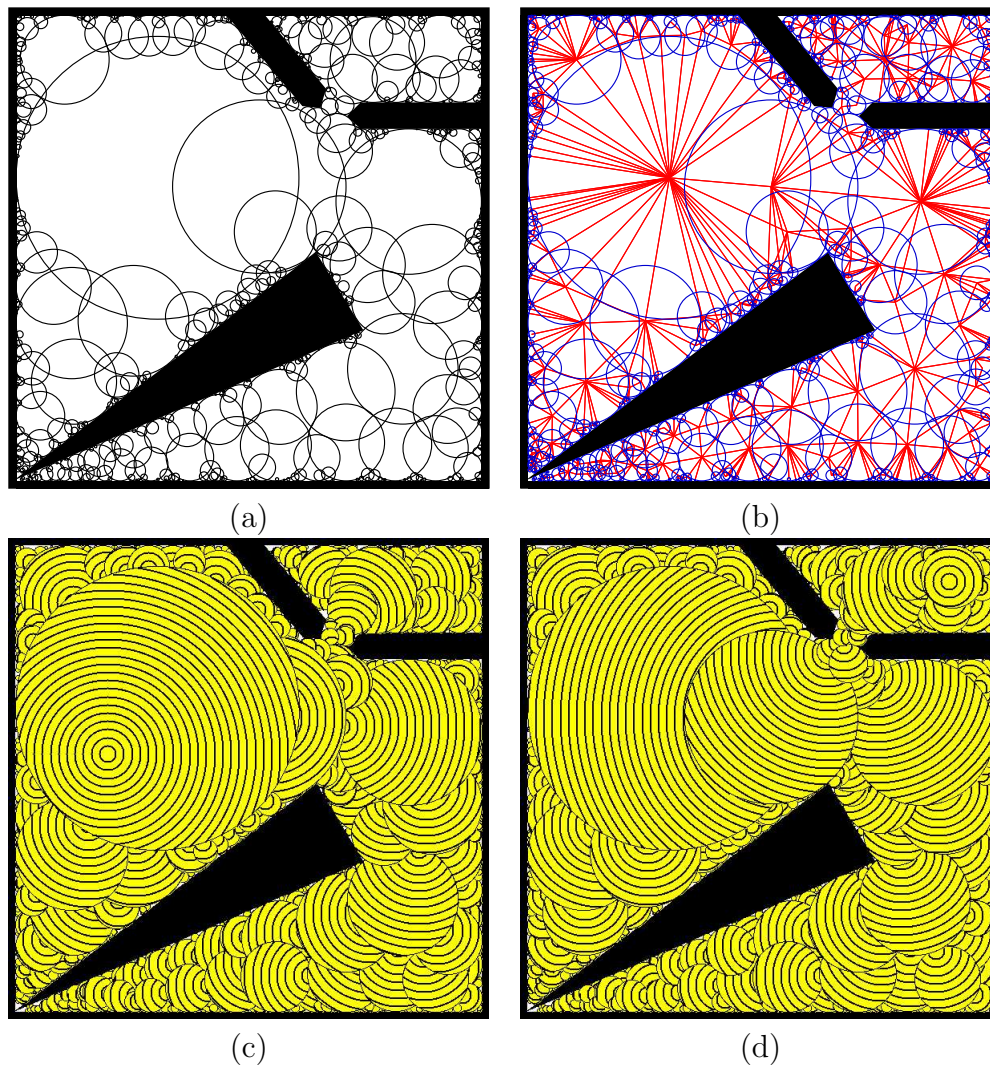


Figure 8.2: (a) The computed neighborhoods for a 2D configuration space; (b) the corresponding graph superimposed on the neighborhoods; (c), (d) the level sets of two navigation functions computed from a single SNG.

Part III

Decision-Theoretic Planning

Overview of Part III: Decision-Theoretic Planning

Planning under Uncertainty

Just as in Part II, it also seems appropriate to give two names to Part III: 1) *decision-theoretic planning*, and 2) *planning under uncertainty*. *explain...*

refer back to computation models of Chapter 1

Chapter 9 addresses the problem of how to model and solve the problem of making a single decision while facing uncertainties. No state space is necessary in this case because the “plan” in this case has only one step. One purpose of the chapter is to introduce and carefully evaluate the assumptions that are typically made in different forms of decision theory. This forms the basis of more complicated problems that follow, especially sequential decision making and control theory.

Chapter 10 extends the tools from Chapter 9 from a single decision to multiple decisions. In this case, a state space is needed once again, and the problems can be considered as a generalizations of the discrete planning problems of Chapter 2. It is assumed that the state can always be perfectly sensed; however, there are uncertainties about what future states will occur.

Chapter 11 introduces perhaps the most important concept of this book: the *information space*. If there is uncertainty in sensing the current state, then the planning problem naturally lives in the information space. An analogy can be made to the configuration space and motion planning. Before efforts to unify motion planning by using configuration space concepts [437, 504, ?], most algorithms were developed on a case by case basis, especially for robot manipulators and mobile robots, which appear to have very different characteristics in the world. However, once viewed in the configuration space, it is possible to construct general algorithms, such as those from Chapters 6 and 5.

A similar kind of unification should be possible for planning problems that involve sensing uncertainties (i.e., are unable to determine the current state). Presently, the literature appears to be mostly on a case by case basis, as basic motion planning once was. Therefore, it is difficult to provide a prespective as unified as the techniques in Part I. Nevertheless, the concepts from Chapter 11 are used to provide a unified introduction to many planning problems that involve sensing uncertainties in Chapter ???. Just as in the case of configuration space, some effort is required to learn the information space concepts, but it will pay great dividends if the investment is made. Honestly!

Chapter 9

Basic Decision Theory

Chapter Status



What does this mean? Check

<http://msl.cs.uiuc.edu/planning/status.html>
for information on the latest version.

These are class notes from CS497 Spring 2003, parts of which were scribed by Steve Lindemann, Shai Sachs.

9.1 Basic Definitions

To introduce some of the basic concepts in single-stage decision making, consider the following scenario:

- Scenario 0**
1. Let U be a set of possible choices: $\{u_1, u_2, \dots, u_n\}$.
 2. Let $L : U \rightarrow \mathbb{R}$ be a loss function or cost function.
 3. Select a $u \in U$ that minimizes $L(u)$.

In this scenario, we see that the set U consists of all choices that we can make; these are also called *actions* or *inputs*. The loss function L represents the cost associated with each possible choice; another approach is to define a *reward function* R which represents the gain or benefit of each choice. These approaches are equivalent, since one can simply take $R(u) = -L(u)$.

A method used to make a decision is called a *strategy*. In this scenario, our strategy was *deterministic*; that is, given some set U and function L , our choice is completely determined. Alternatively, we could have taken a *randomized* strategy, in which our decision also depended on the outcome of some random events. In this strategy, we define a function $p : U \rightarrow \mathbb{R}$ such that the probability of selecting a particular choice u is $p(u)$; denote $p(u_i) = p_i$. The ordinary rules

governing probability spaces apply (e.g., $\sum_{i=1}^n p_i = 1, p_i \geq 0 \forall i$). Randomized and deterministic strategies are also called *mixed* and *pure*, respectively. For purposes of notation, we will use u^* to refer to a randomized strategy and \mathcal{U} to refer to the set of all randomized strategies.

Example 9.1.1 Let the input set $U = \{a, b\}$. Then one can choose a randomized strategy u^* in the following way:

1. Flip a fair H/T coin.
2. If the result is H, choose a ; if T, choose b .

Since the coin is fair, this corresponds to choosing $p(a) = 0.5, p(b) = 0.5$.

Consider the following scenario:

- Scenario 1**
1. $U = \{u_1, u_2, \dots, u_n\}$
 2. $L : U \rightarrow \mathbb{R}$
 3. Select $u^* \in \mathcal{U}$ that minimizes $E[L] = \sum_{i=1}^n L(u_i)p_i$.

$E[L]$ reflects the average loss if the game were to be played many times. Now, Scenarios 0 and 1 are identical, with the exception that one uses a deterministic strategy, and one uses a randomized strategy. Which is better? To help answer this, we give the following example:

Example 9.1.2 Let $U = \{1, 2, 3\}$, and $L(1) = 2, L(2) = 3, L(3) = 5$ (we may write this in vector notation as $L = [2 \ 3 \ 5]$). Following the deterministic strategy from Scenario 0, we choose $u = 1$. What if we use the strategy from Scenario 1? By inspection we can see that we need $p = [1 \ 0 \ 0]$; thus, the randomized strategy results in the same choice as the deterministic one.

We have seen in the above example that a randomized strategies and deterministic ones can produce identical results. However, what if for some input set U and loss function L , we have $L(u_i) = L(u_j)$? Then, there can be randomized strategies which act differently than deterministic ones. However, if one only considers the minimum loss attained, they are not better because both types of strategies will select actions resulting in minimum loss. Thus, in this case we find that Scenario 1 is useless! However, randomized strategies are very useful in general, as shown in the following example.

Example 9.1.3 (Matching Pennies) Consider a game in which two players simultaneously choose H or T. If the outcome is HH or TT (the players choose the same), then Player 1 pays Player 2 \$1; if the outcome is HT or TH, then Player 2 pays Player 1 \$1. What happens if Player 1 uses a deterministic strategy? If Player 2 can determine what that strategy is, then he can choose his strategy so that he always wins the game. However, if Player 1 chooses a randomized strategy, he can at least expect to break even (what randomized strategy guarantees this?).

So far, we have examined scenarios in which there were only a finite number of possible choices. Many problems, however, have a continuum of choices, as does the following:

- Scenario 2**
1. $U \subseteq \mathbb{R}^d$ (usually, U is closed and bounded)
 2. $L : U \rightarrow \mathbb{R}$
 3. Select $u \in U$ to minimize L

This is a classical optimization problem.

Example 9.1.4 Let $U = [-1, 1] \subset \mathbb{R}$ and $L(u) = u^2$. To attain minimum cost we choose $u = 0$.

However, what if in the example above we chose $U = (0, 1)$? Then the minimum is not well-defined. However, we can introduce the concept of the *infimum*, which is the greatest lower bound of a set. Similarly, we can introduce the *supremum*, which is the least upper bound of a set. Then, we can still say $\inf_{u \in U} L(u) = 0$.

9.2 A Game Against Nature

In the previous scenarios, we have assumed complete knowledge about the loss function L . This need not be the case, however; in particular situations, there may be uncertainty involved. One convenient way to describe this uncertainty is to introduce a special decision-maker, called *nature*. Nature is an unreasoning entity (i.e., it is not an adversary), and we do not know what decision nature will make (or has made). We call the set Θ the set of choices for nature (alternatively, the *parameter space*), and $\theta \in \Theta$ is a particular choice by nature. The parameter space may be either discrete or continuous; in the discrete case, we have $\Theta = \{\theta_1, \theta_2, \dots, \theta_n\}$, and in the continuous case we have $\Theta \subseteq \mathbb{R}^d$. Then, we can define the loss function to be $L : U \times \Theta \rightarrow \mathbb{R}$, in which the operator $\cdot \times \cdot$ is the Cartesian product.

Example 9.2.1 Let L be specified by the following table:

| | | | |
|----------|-----|----|----|
| | U | | |
| | 1 | -1 | 2 |
| Θ | -1 | 2 | -1 |
| | 0 | -2 | 1 |

The best strategy to adopt depends on what model we have of what nature will do:

- **Nondeterministic:** I have no idea.
- **Probabilistic:** I have been observing nature and gathering statistics.

In the first case, one might assume Murphy’s Law (“If anything can go wrong, it will”); then, one would choose the column with the least maximum value. Alternatively, one might assume that nature’s decisions follow a uniform distribution, all choices being equally likely. Then one would choose the column with the least average loss (this approach was taken by Laplace in 1812). In the second case, one could use Bayesian analysis to calculate a probability distribution $P(\theta)$ of the actions of nature, and use that to make decisions. The following two scenarios formalize these approaches.

Scenario 3 (Minimax solution) 1. $U = \{u_1, \dots, u_n\}$

2. $\Theta = \{\theta_1, \dots, \theta_m\}$

3. $L : U \times \Theta \rightarrow \mathbb{R}$

4. Choose u to minimize $\max_{\theta \in \Theta} L(u, \theta)$.

Scenario 4 (Expected optimal solution) 1. $U = \{u_1, \dots, u_n\}$

2. $\Theta = \{\theta_1, \dots, \theta_m\}$

3. $P(\theta)$ given $\forall \theta \in \Theta$

4. $L : U \times \Theta \rightarrow \mathbb{R}$

5. Choose u to minimize $E^\theta[L] = \sum_{\theta \in \Theta} L(u, \theta)P(\theta)$.

Again consider Example 9.2.1. If the strategy from Scenario 3 is adopted, then we would choose u_1 so that we would pay loss 1 in the worst case. If the strategy from Scenario 4 is chosen, and assuming $P(\theta_1) = 1/5$, $P(\theta_2) = 1/5$, $P(\theta_3) = 3/5$, we find that u_2 has the lowest expected loss, and so would take that action. If the probability distribution had been $P = [1/10 \ 4/5 \ 1/10]$, then simple calculations show that u_1 is the best choice. Hence our decision depends on $P(\theta)$; if this information is statistically valid, then better decisions are made. If it is not, then potentially worse decisions can be made.

Another strategy is to minimize “regret”, the amount of loss you could have eliminated if you had chosen differently, given the action of nature. A regret matrix corresponding to Example 9.2.1 can be found in Figure 9.1. Given some regret matrix, one can adopt a minimax or expected optimal strategy.

9.2.1 Having a single observation

Let y be an *observation*; this could be some data, a measurement, or a sensor reading. Let Y be the *observation space*, the set of all possible y . Now, we can make a decision based on y ; let $\gamma : Y \rightarrow U$ denote a *decision rule* (strategy, plan). Then modify our decision strategies as follows:

| | | | |
|----------|-----|---|---|
| | U | | |
| | 2 | 0 | 3 |
| Θ | 0 | 3 | 0 |
| | 2 | 0 | 3 |

Figure 9.1: A regret matrix corresponding to Example 9.2.1.

- **Nondeterministic:** Assume there is some $F(y) \subseteq \Theta$, which is known for every $y \in Y$. Choose some γ that minimizes $\max_{\theta \in F(y)} L(\gamma(y), \theta)$ for each $y \in Y$.
- **Probabilistic:** Assume that $P(y|\theta)$ is known, $\forall y \in Y, \forall \theta \in \Theta$. Then Bayes rule yields $P(\theta|y) = P(y|\theta)P(\theta)/P(y)$, in which $P(y) = \sum_{\theta \in \Theta} P(y|\theta)P(\theta)$ ¹. Then choose γ so that it minimizes the *conditional Bayes risk* $R(u|y) = \sum_{\theta \in \Theta} L(u, \theta)P(\theta|y)$, for every $y \in Y$.

Formally, we have the following scenarios:

Scenario 5 (Nondeterministic) 1. $U = \{u_1, \dots, u_n\}$

2. $\Theta = \{\theta_1, \dots, \theta_m\}$

3. $Y = \{y_1, \dots, y_l\}$

4. $F(y)$ given $\forall y \in Y$

5. $L : U \times \Theta \rightarrow \mathbb{R}$

6. Choose γ to minimize $\max_{\theta \in F(y)} L(\gamma(y), \theta)$ for each $y \in Y$.

Scenario 6 (Bayesian decision theory) 1. $U = \{u_1, \dots, u_n\}$

2. $\Theta = \{\theta_1, \dots, \theta_m\}$

3. $Y = \{y_1, \dots, y_l\}$

4. $P(\theta)$ given $\forall \theta \in \Theta$.

5. $P(y|\theta)$ given $\forall y \in Y, \theta \in \Theta$

6. $L : U \times \Theta \rightarrow \mathbb{R}$

7. Choose γ to minimize $R(\gamma(y)|y)$ for every $y \in Y$.

Extending the former case, we may imagine that we have k observations: y_1, \dots, y_k . Then, $R(u|y_1, \dots, y_k) = \sum_{\theta \in \Theta} L(u, \theta)P(\theta|y_1, \dots, y_k)$. If we assume that $P(y_i|\theta)$ is known for each $i \in \{1, \dots, k\}$ and that conditional independence holds, we have $P(\theta|y_1, \dots, y_k) = \left(\prod_{i=1}^k P(y_i|\theta)\right) P(\theta)/P(y_1, \dots, y_k)$.

¹For the purposes of decision-making, $P(y)$ is simply a scaling factor and may be omitted.

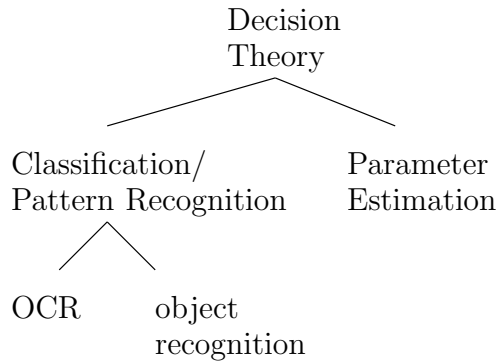


Figure 9.2: An overview of decision theory.

9.3 Applications of Optimal Decision Making

An overview of the field of decision theory and its subfields is pictured in Figure 9.2.

9.3.1 Classification

Let $\Omega = \{\omega_1, \dots, \omega_n\}$ denote a set of *classes*, and let y denote a *feature* and Y a *feature space*. For this type of problem, we have $\Theta = U = \Omega$, since nature selects an object from one of the classes, and we attempt to identify the class nature has selected. The feature set Y represents useful information that can help us identify which class an object belongs to.

The basic task of classification can be described as follows. We are given y , a *feature vector*, where $y \in Y$, and Y is the set of all possible feature vectors. The set of possible classes is Ω . Given an object with a feature vector y , we wish to determine the correct class $\omega \in \Omega$ of the object.

Ideally, we are given $P(y|\omega)$ and $P(\omega)$, the *prior distribution* over the classes. The probability $P(\omega)$ gives the probability that an object falls in the class ω .

A reasonable cost function is

$$L(u, \theta) = \begin{cases} 0 & \text{if } u = \theta \text{ (the classification is correct)} \\ 1 & \text{if } u \neq \theta \text{ (the classification is incorrect)} \end{cases}$$

If the Bayesian decision strategy is adopted, it will result in choices that minimize the expected probability of misclassification.

Example 9.3.1 (Optical Character Recognition) Let $\Omega = \{A, B, C, D, E, F, G, H\}$. Further, imagine that we our image processing algorithms can extract the following features:

| | | |
|-------|---|-----------|
| Shape | 0 | A E F H |
| | 1 | B C D G |
| Ends | 0 | B D |
| | 1 | |
| | 2 | A C G |
| | 3 | F E |
| | 4 | H |
| Holes | 0 | C E F G H |
| | 1 | A D |
| | 2 | B |

Assuming that the image processing algorithms never err, we can use a minimax strategy to make our decision. Are there any letters which the features do not distinguish? If so, what enhancements might we make to our image processing algorithms to distinguish them? If we assume that the image processing algorithms sometimes make mistakes, then we can use a Bayesian strategy. After running the algorithms thousands of times and gathering statistics, we can learn the necessary conditional probabilities and use them to make the decision with the highest expectation of success.

9.3.2 Parameter Estimation

One subfield of Decision Theory is parameter estimation. The goal is to estimate the value of some parameter, given some observation of the parameter. We consider the parameter to be some fixed constant, and denote the set of all possible parameters (the *parameter space*) as X .

Using our notation from decision theory, we have $\Theta = X$. The parameter we are trying to estimate is nature's choice.

Since the goal is the guess the correct value of the parameter, the set of actions U is also equal to X ; that is, the human player's action is to choose some valid value of the parameter as her guess. Therefore, we have $X = \Theta = U$.

Further, we have an observation about the parameter, y ; we denote the set of all possible observations by Y . Clearly, $X \subseteq Y$.

Suppose we have $X = [0, 1] \subseteq \mathbb{R}$, $p(y|x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(x-y)^2}{2\sigma^2}\right)$, and $p(x) = 1$. We interpret these probability density functions as follows: $p(y|x)$ tells us that there is some Gaussian noise in the observation (that is, our observations of the parameters, over many trials, will be concentrated around the true parameter in a Gaussian distribution); further, $p(x)$ tells us that each parameter is equally likely.

Finally, we choose a loss function which measures the estimation error (that is, the difference between our estimate and the true parameter). We use $L(u, x) = (u - x)^2$.

We wish to choose the input u which minimizes our risk; we therefore choose u which minimizes

$$R(u) = \int_x L(u, x)p(y|x)p(x)dx \quad (9.1)$$

Note that when Equation 9.1 is multiplied by $\frac{1}{p(y)}$, by Bayes' rule we have

$$R(u) = \int_x L(u, x)p(y|x)p(x)\frac{1}{p(y)}dx = \int_x L(u, x)p(x|y)dx$$

Then the expression for $R(u)$ becomes exactly analogous to the discrete form of the risk function from our previous lecture. Since $p(y)$ is constant over X , we may remove it from the integral without affecting the correct choice of u . Therefore $R(u)$ in Equation 9.1 is not exactly the risk function, but it is closely related.

9.4 Utility Theory

Utility theory asks: Where does the loss function L come from? In other words, how useful is one loss compared against another?

In utility theory we replace “loss” with “reward”; the human wants to maximize reward. Note that this convention can easily be inverted to return the the usual “loss” convention in decision theory.

9.4.1 Choosing a Good Reward

Let u_1 be some fairly unpleasant task (such as writing scribe notes), and u_2 be the act of doing nothing. We consider the problem of choosing a good reward function using the following examples.

1. Let $R(u_1) = 1000$, and $R(u_2) = 0$. For a poor graduate student, it may be worthwhile to write scribe notes for \$ 1000, so the student will probably do u_1 in this scenario. One difficulty in this scenario is that we haven't considered the possible cost to the human of each action.
2. Let $R(u_1) = 10001000$, and $R(u_2) = 10000000$. Although the relative reward is the same, the action chosen is probably different! This is so because the value (or utility) of money decreases as we have more of it.
3. Let $R(u_1) = 10000$ and $R(u_2) = 25,000$ with probability $\frac{1}{2}$, and 0 with probability $\frac{1}{2}$. In this scenario some conservative students may choose u_1 , to guarantee a reward; while more adventurous gamblers may choose u_2 , as the expected gain is greater.
4. Let $R(u_1) = 100$, and $R(u_2) = 250$ with probability $\frac{1}{2}$, and 0 otherwise; allow the student to choose an action (and collect the corresponding reward) 100 times. The expected reward for each action remains the same, but we

| | | |
|------------|-------|-------|
| | u_1 | u_2 |
| θ_1 | 1 | 0 |
| θ_2 | 3 | 2 |

Figure 9.3:

imagine more students will choose u_2 100 times than those that will choose u_1 100 times. This is so because “repeatability” is important in games of expectation; that is, the true outcome of a game is more likely nearer the expected value if the number of trials increases.

The goal of utility theory is to construct reward functions that give the right expected value for a game, given the preferences of the human player.

We call the set of all possible rewards for a given game the *reward space*, and denote it by \mathcal{R} . For example, in scenario 3, we have $\mathcal{R} = \{0, 10000, 25000\}$.

Consider game with nature depicted in Figure 9.3. Suppose $P(\theta_1) = \frac{1}{4}$ and $P(\theta_2) = \frac{3}{4}$. Then choosing u_1 implies we get $R(u_1) = 1$ with probability $\frac{1}{4}$, and $R(u_1) = 3$ with probability $\frac{3}{4}$. We may consider the prior distribution over Θ as giving us a probability distribution over \mathcal{R} .

In general, we let \mathcal{P} be the set of all probability distributions over \mathcal{R} , and let $P \in \mathcal{P}$ be one such distribution. We expect the human player to express her preference between any two such probability distributions, and we denote these preferences with the usual inequality operations. Thus $P_1 \leq P_2$ indicates that the human prefers P_1 no more than P_2 , $P_1 = P_2$ indicates that the human has no preference among P_1, P_2 , and so on.

We may then express the goal of utility theory as follows: we wish to find some function $V : \mathcal{R} \mapsto \mathbb{R}$ such that $P_1 < P_2$ iff $E^{P_1}[V(r)] < E^{P_2}[V(r)]$. That is, the expected value of a reward is greater under more preferred distributions over the reward space.

Note that computing V is difficult. However, we know (but will not prove) that V exists when the human is rational – that is, when her choices obey the Axioms of Rationality.

9.4.2 Axioms of Rationality

We say that a human is *rational* when the preferences she expresses among probability distributions over \mathcal{R} obey the following axioms.

1. If $P_1, P_2 \in \mathcal{P}$, then either $P_1 \leq P_2$ or $P_2 \leq P_1$.
2. If $P_1 \leq P_2$ and $P_2 \leq P_3$ then $P_1 \leq P_3$.
3. If $P_1 < P_2$ then $\alpha P_1 + (1 - \alpha)P_3 < \alpha P_2 + (1 - \alpha)P_3$, for all $P_3 \in \mathcal{P}$ and $\alpha \in (0, 1)$.

| | | |
|------------|-------|-------|
| | u_1 | u_2 |
| θ_1 | 2 | 1000 |
| θ_2 | 2 | 0 |

Figure 9.4: A scenario in which worst-case decision making might yield undesirable results.

This axiom is strange, but it merely means that no matter how much we “blend” P_1 and P_2 with some other distribution P_3 , we will still prefer the “blended” P_2 to the “blended” P_1 .

4. If $P_1 < P_2 < P_3$ then $\exists \alpha \in (0, 1), \beta \in (0, 1)$ such that $\alpha P_1 + (1 - \alpha)P_3 < P_2$ and $P_2 < \beta P_1 + (1 - \beta)P_3$.

This axiom means that no matter how good P_3 is, we can always blend a bit of it with P_1 to get a distribution less preferable than P_2 ; similarly, no matter how bad P_1 is, we can always blend a bit of it with P_3 to get a distribution more preferable than P_2 .

9.5 Criticisms of Decision Theory

We consider a few criticisms of decision theory:

1. The values of rewards are subjective. If they are provided by the human, then the process of making a decision may amount to “garbage in, garbage out.”
2. It is difficult to assign losses.
3. Badly chosen loss functions can lead to bad decisions.

One response to this criticism is *sensitivity analysis*, which claims that the decisions are not hypersensitive to the loss functions. Of course, if this argument is taken to far, then the value of decision theory itself is thrown into question.

9.5.1 Nondeterministic decision making

There are two main criticisms of nondeterministic decision making: first, “worst-case” analysis can yield undesirable results in practice; and second, the same decisions can often be acquired through Bayesian decision making by manipulating the prior distribution.

Consider the rewards in Figure 9.4. Worst-case analysis causes us to choose u_1 , although in practice we may want to choose u_2 if we know the risk of event θ_2 is low (equivalently, the probability of θ_2 must be rather high for the expected

| <u>Bayesian</u> | <u>frequentist</u> |
|--|--|
| A probability is a belief about the outcome of a single trial. | A probability of an event is the limit of the frequency of that event in many repeated trials. |
| subjective | objective, minimalist |
| practical | rigorous, but often useless |

Figure 9.5: A comparison of the Bayesian and frequentist interpretations of probabilities.

value of the scenario to favor choosing u_1 .) Further, we can simulate the result of the nondeterministic decision as a Bayesian decision by correctly assigning prior distributions - for example, by setting $P(\theta_1) \ll P(\theta_2)$.

9.5.2 Bayesian decision making

A common criticism of Bayesian decision making centers around the Bayesian interpretation of probabilities. We compare the Bayesian and frequentist interpretations of probabilities in Figure 9.5.

While frequentists do not incorporate prior beliefs into decisions, they do incorporate observations. Thus, a frequentist risk function might be:

$$R(\theta, \gamma) = \int_y L(\theta, \gamma(y))p(y|\theta)dy$$

Prior distributions One problem with this function is that both θ and γ are unknown. If we are to choose a γ which minimizes $R(\theta, \gamma)$, then our choice of θ might considerably influence the choice of γ .

A considerable difficulty for Bayesian decision making is determining the prior distributions. One common distribution is the Laplace distribution. Using the principle of insufficient reason, this distribution makes each θ equally likely.

The Laplace distribution has some justification from information theory. Lacking any information about Θ , we may wish to choose the most “non-informative” prior - that is, the probability distribution which contains the least information.

The entropy contained in a probability distribution over Θ can be computed using the Shannon Entropy Equations:

$$E = - \sum_{\theta} P(\theta) \log P(\theta) \tag{9.2}$$

$$E = - \int_{\theta} p(\theta) \log p(\theta) d\theta \tag{9.3}$$

Equation 9.2 is for discrete probability mass functions; Equation 9.3 is for continuous probability density functions.

Using Shannon's Entropy Equations, we can show that the probability distribution which yields the least information (the highest entropy) is that which assigns equal probabilities to all events in Θ – that is, the Laplace distribution.

The structure of Θ We encounter several problems with the Laplace distribution as we consider the structure of Θ .

Suppose $\Theta = \mathbb{R}$. The Laplace distribution assigns 0 probability to any bounded interval of \mathbb{R} . This difficulty is mostly mechanical however; the use of generalized probability density functions solves this problem.

Often, we can structure Θ in arbitrary ways that significantly affect the prior distribution. Suppose we let $\Theta = \{\theta_1, \theta_2\}$, where θ_1 indicates “no precipitation” and θ_2 indicates “some precipitation”. The Laplace distribution assigns $P(\theta_1) = P(\theta_2) = \frac{1}{2}$.

Suppose instead we let $\Theta' = \{\theta_1, \theta_2, \theta_3\}$, where θ_1 indicates “no precipitation”, θ_2 indicates “rain”, and θ_3 indicates “snow”. Clearly, Θ' describes the same set of events as Θ . But in this scenario the Laplace distribution assigns $P(\theta_1) = P(\theta_2) = P(\theta_3) = \frac{1}{3}$. The combined probability of precipitation is $\frac{2}{3}$. Which characterization of nature is correct – Θ or Θ' ?

The following is an interesting practical example of arbitrary choices about the structure of Θ . Suppose we wish to fit a line to a set of points. The equation for the line is $\theta_1 x + \theta_2 y + \theta_3 = 0$. What prior distribution should we choose for θ_1 , θ_2 and θ_3 ? We could choose to “spread the probability” around the unit sphere, by requiring $\theta_1^2 + \theta_2^2 + \theta_3^2 = 1$. However, this choice is entirely arbitrary; we could have also spread the probability around the unit cube, with very different results.

9.6 Multiobjective Optimality

For now, we concentrate on multiple-objective decisions with no uncertainty. Thus, we have U (the input space), and the loss function $L : U \mapsto \mathbb{R}^d$.

The goal is to find all $u \in U$ such that there is no $u' \in U$ with $L(u') \leq L(u)$. That is, we wish to compute the set of “minimal” inputs u , using the partial ordering \leq . This set is called the Pareto optimal set of inputs.

Let $L(u) = \langle L_1(u), \dots, L_d(u) \rangle$. Then we define $L(u') \leq L(u)$. Then we define $L(u') \leq L(u)$ iff $L_i(u') \leq L_i(u)$ for all i .

Consider the multi-decision problem depicted in Figure 9.6. Two robots, indicated by hollow circles, wish to travel along the paths designated. Suppose the path for and speed for each robot is fixed; the only actions possible are starting and stopping at various points in time. Suppose further that the loss function computes the time for each robot to travel along its designated path.

Clearly, many possible inputs are possible. For example, one robot could wait until the other robot has reached its goal before starting to move. Alternately,

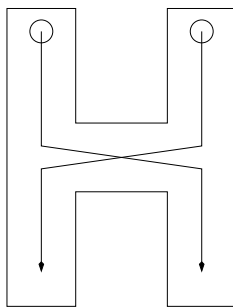


Figure 9.6: A multi-objective decision problem in which, although there are many conceivable inputs, there are only two Pareto optimal loss values.

both robots could move until just before collision, at which point one robot stops and the other continues moving; the stopped robot continues moving once collision is avoided. Nevertheless, there will only be two Pareto optimal loss values for this problem, such as $\langle 4, 6 \rangle$ and $\langle 6, 4 \rangle$.

One problem with Pareto optimality is that it might yield an “optimal” set which is identical to U . For example, consider $U = [0, 1]$ and $L(u) = \langle u, 1 - u \rangle$. It is easy to see that the optimal set for this scenario is just U , since whenever one component of $L(u)$ increases, the other decreases by the same amount.

9.6.1 Scalarizing L

If we can “scalarize” L , then we can find a single optimal value of $L(u)$, rather than many possible optimal values.

We can scalarize L as follows. Choose $\alpha_1, \dots, \alpha_d \in (0, 1)$. Let $l(u) = \sum_i \alpha_i L_i(u)$. Note that $l(u)$ is just the dot product of $L(u)$ and the α vector.

We can make a multi-objective decision by choosing the u which minimizes $l(u)$. It turns out that this u must also be in the Pareto optimal set. Note that it is possible that more than one u yields the minimum $l(u)$.

We might interpret the α_i as a set of “priorities” over the components of L – higher α_i ’s are more important, and higher losses in the corresponding components of $L(u)$ should be avoided.

9.7 Two-Player Zero Sum Games

9.7.1 Overview of game theory

In a game, several decision makers strive to maximize their (expected) utility by choosing particular courses of action, and each decision maker’s final utility pay-offs depend on the courses of action chosen by all decision makers. The interactive situation specified by the set of participants, the possible courses of action of each

decision maker, and the set of all possible utility payoffs, is called a **game**; the decision makers 'playing' a game are called the **players**.

Game theory is a set of analytical tools designed to help us understand the phenomena that we observe when decision-makers interact. The basic assumptions that underlie the theory are that decision-makers pursue well-defined exogenous objectives (they are rational) and take into account their knowledge or expectations of other decision-makers' behavior (they reason strategically).

Some of the areas of game theory that we are going to look into are:

- **Multiple Decision Makers:** There will be two or more decision makers, trying to make decisions at the same time.
- **Single stage v Multiple stage**
- **Zero sum v Non zero sum games:** Zero-sum games are games where the amount of "winnable goods" (or resources) is fixed. Whatever is gained by one decision maker, is therefore lost by the other decision maker: the sum of gained (positive) and lost (negative) is zero.
In non-zero-sum games there is no universally accepted solution. That is, there is no single optimal strategy that is preferable to all others, nor is there a predictable outcome. Non-zero-sum games are also non-strictly competitive, as opposed to the completely competitive zero-sum games, because such games generally have both competitive and cooperative elements. Players engaged in a non-zero sum conflict have some complementary interests and some interests that are completely opposed.
- **Different Information States for each player:** Each player has an information set corresponding to the decision nodes, which are used to represent situations where the player may not have complete knowledge about everything that happens in a game. Information sets are unique for each player.
- **Deterministic v Randomized Strategies:** When the player uses a deterministic or pure strategy, the player specifies a choice from his information set. When a player uses a mixed strategy, he plays unpredictably in order to keep the opponent guessing.
- **Cooperative v Noncooperative:** A player may be interpreted as an individual or as a group of individuals making a decision. Once we define the set of players, we may distinguish between two types of models: those in which the sets of possible actions of individual players are primitives and those in which the sets of possible joint actions of groups of players are primitives. Models of the first type can be referred to as "noncooperative", while those of the second type can be referred to as "cooperative".

There are two main assumptions in game theory:

- Players know each other's loss functionals.

- Players are rational decision makers.

The following table summarizes some of the above mentioned features:

| # of players | # of steps | Nature ? | Cost Functionals | Example |
|--------------|------------|----------|------------------|------------------------------|
| 1 | 1 | N | 1 | Classical Optimization |
| 1 | 1 | Y | 1 | Basic Decision Theory |
| > 1 | 1 | N | > 1 | Matrix Games |
| > 1 | 1 | Y | > 1 | Markov Games (probabilistic) |
| 1 | >1 | N | 1 | Optimal Control Theory |
| 1 | >1 | Y | 1 | Stochastic Control |
| >1 | >1 | N/Y | > 1 | Dynamic Game Theory |
| 1 | 1 | N | > 1 | Multi-objective Optimality |
| >1 | >1 | N/Y | 1 | Team Theory |

The most elementary type of two-player zero sum games are *matrix games*. The main features of such games are:

- There are two players P_1 and P_2 and an $(m \times n)$ dimensional loss matrix $A = \{a_{ij}\}$.
- Each entry of the matrix is an outcome of the game, corresponding to a particular pair of decisions made by the players.
- For P_1 , the alternatives are the m rows of the matrix and for P_2 , the alternatives are the n columns of the matrix. These are also known as the strategies of the players and can be expressed in the following way:

$$\begin{aligned} U^1 &= u_1^1, u_2^1, \dots, u_m^1 \\ U^2 &= u_1^2, u_2^2, \dots, u_n^2 \end{aligned}$$

- Both players play simultaneously.
- If P_1 chooses the i th row and P_2 chooses the j th column, then a_{ij} is the outcome of the game and P_1 pays this amount to P_2 . In case a_{ij} is negative, this should be interpreted as P_2 paying P_1 the positive amount corresponding to this entry.

More formally, for each pair $\langle U_i^1, U_j^2 \rangle$,

P_1 has loss $L_1(U_i^1, U_j^2)$ and

P_2 has loss $L_2(U_i^1, U_j^2) = -L_1(U_i^1, U_j^2)$

We can write the loss functional as simply L , where P_1 tries to minimize L and P_2 tries to maximize L .

Example: Suppose the loss matrix for players P_1 and P_2 is as below:

| # of players | # of steps | Nature ? | Loss Functionals | Example |
|--------------|------------|----------|------------------|------------------------------|
| 1 | 1 | N | 1 | Classical Optimization |
| 1 | 1 | Y | 1 | Decision Theory |
| > 1 | 1 | N | > 1 | Matrix Games |
| > 1 | 1 | Y | > 1 | Markov Games (probabilistic) |
| 1 | >1 | N | 1 | Optimal Control Theory |
| 1 | >1 | Y | 1 | Stochastic Control |
| >1 | >1 | N/Y | > 1 | Dynamic Game Theory |
| 1 | 1 | N | > 1 | Multi-objective Optimality |
| >1 | >1 | N/Y | 1 | Team Theory |

In order to illustrate the above mentioned features of matrix games, let us consider the following (3×4) matrix.

| | | | | |
|-------|-------|----|---|---|
| | P_2 | | | |
| | 1 | 3 | 3 | 2 |
| P_1 | 0 | -1 | 2 | 1 |
| | -2 | 2 | 0 | 1 |

In this case, P_2 , who is the maximizer, has a unique *security strategy*, “column 3” ($j^* = 3$), securing him a gain-floor $\underline{V} = 0$. P_1 , who is the minimizer, has two security strategies, “row 2” and “row 3” ($i_1^* = 2, i_2^* = 3$) yielding him a loss ceiling of $\overline{V} = \max_j a_{2j} = \max_j a_{3j} = 2$ which is above the security level of P_2 .

We can express this more formally in the following notation:

Security strategy for $P_1 = \operatorname{argmin}_i \max_j L(U_i^1, U_j^2)$

Therefore, loss-ceiling or upper value $\overline{V} = \min_i \max_j L(U_i^1, U_j^2)$

Security strategy for $P_2 = \operatorname{argmax}_j \min_i L(U_i^1, U_j^2)$

Therefore, gain-floor or lower value $\underline{V} = \max_j \min_i L(U_i^1, U_j^2)$

9.7.2 Regret

If P_2 plays first, then he chooses column 3 as his security strategy and P_1 's unique response would be row 3, yielding an outcome of $\underline{V} = 0$. However, if P_1 plays first, then he is indifferent between his two security strategies. In case he chooses row 2, P_2 will respond with choosing column 2 and if P_1 chooses row 3, then P_2 chooses column 2, both strategies resulting in in outcome of $\overline{V} = 2$.

This means that when there is a definite order of play, security strategies of the player who acts first make complete sense and they can be considered to be in equilibrium with the corresponding response strategies of the other player. By the

two strategies being in equilibrium, it is meant that after the game is over and its outcome is observed, the players should have no ground to regret their past actions. Therefore, in a matrix game with a fixed order of play, for example, there is no justifiable reason for a player who acts first to regret his security strategy.

In matrix games where players *arrive at their decisions independently* the security strategies cannot possibly possess any sort of equilibrium. To illustrate this, we look at the following matrix:

| | | | |
|----------------|----------------|----|----|
| | P ₂ | | |
| | 4 | 0 | -1 |
| P ₁ | 0 | -1 | 3 |
| | 1 | 2 | 1 |

We assume that the players act independently and the game is to be played only once. Both players have unique security strategies, “row 3” for P₁ and “column 1” for P₂, with the upper and lower values of the game being $\bar{V} = 2$ and $\underline{V} = 0$ respectively. If both players play according to their security strategies, then the outcome of the game is 1, which is midway between the security strategies of the players. But after the game is over, both P₁ and P₂ might have regrets. This indicates that in this matrix game, the security strategies of the players cannot possibly possess any equilibrium property.

9.7.3 Saddle Points

For a class of matrix games with equal upper and lower values, a dilemma regarding regret does not arise. If there exists a matrix game where $\bar{V} = \underline{V} = V$ then we say that the strategies are in equilibrium, since each one is optimal against the other. The strategy pair (row x, col y), possessing all the favorable features is clearly the only candidate that can be considered as the equilibrium of the matrix game.

Such equilibrium strategies are known as *saddle point strategies* and the matrix game in question is said to have a *saddle point in pure strategies*.

There can also be multiple saddle points as shown in the following figure:

| | | | | |
|---|---|---|---|---|
| | ≥ | | ≥ | |
| ≤ | V | ≤ | V | ≤ |
| | ≥ | | ≥ | |
| ≤ | V | ≤ | V | ≤ |
| | ≥ | | ≥ | |

9.7.4 Mixed Strategies

Another approach to obtain equilibrium in a matrix game that does not possess a saddle point and in which players act independently is to enlarge the strategy

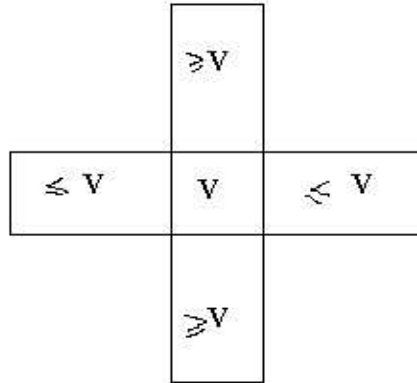


Figure 9.7: Saddle point

spaces so that the players can base their decisions on the outcome of the random events - this strategy is called *mixed strategy* or *randomized strategy*. Unlike the pure strategy case, here the same game is allowed to be played over and over again, and the final outcome, sought to be minimized by P_1 or maximized by P_2 is determined by averaging the outcomes of the individual outcomes.

A strategy of a player can be represented by probability vectors. Suppose the strategy for P_1 is represented by

$$y = [y_1, y_2, \dots, y_n]^T \text{ where } y_i \geq 0 \text{ and } \sum y_i = 1$$

and the strategy for P_2 is represented by

$$z = [z_1, z_2, \dots, z_n]^T \text{ where } z_i \geq 0 \text{ and } \sum z_i = 1$$

Let A be the loss matrix. Therefore,

Expected loss for P_1 is,

$$\begin{aligned} E[L_1] &= \sum_{i=1}^n \sum_{j=1}^m a_{ij} y_i z_j \\ &= y^T A z \end{aligned}$$

Note: Az is the expected losses over nature's choices, given P_1 's actions. Az makes P_2 look like nature (probabilistic) to P_1 .

Expected loss for P_2 is,

$$E[L_2] = -E[L_1]$$

It turns out that we can always find a saddle point in the space of mixed strategies.

Mixed Security Strategy A vector $y^* \in Y$ is called a *mixed security strategy* for P_1 in the matrix game A , if the following inequality holds $\forall Y$:

$$\bar{V}_m(A) = \max_{z \in Z} y^{*'}Az \leq \max_{z \in Z} y'Az \quad y \in Y \quad (9.4)$$

Here, the quantity \bar{V}_m is known as the average security level of P_1 .

Analogously, a vector $z^* \in Z$ is called a *mixed security strategy* for P_2 in the matrix game A , if the following inequality holds $\forall Z$:

$$\underline{V}_m(A) = \min_{y \in Y} y^{*'}Az \leq \min_{y \in Y} y'Az \quad z \in Z \quad (9.5)$$

Here, the quantity \underline{V}_m is known as the average security level of P_2 .

From eq. (1), we have,

$$\bar{V}_m \leq \bar{V} \quad (9.6)$$

Similarly, from eq. (2):

$$\underline{V} \leq \underline{V}_m \quad (9.7)$$

Therefore, combining eq. (3) and eq. (4), we have:

$$\underline{V} \leq \underline{V}_m \leq \bar{V}_m \leq \bar{V} \quad (9.8)$$

According to Von Neumann, \underline{V}_m and \bar{V}_m always equal. So eq. (5) can be written as :

$$\underline{V} \leq \underline{V}_m = \bar{V}_m \leq \bar{V} \quad (9.9)$$

which essentially means that there always exists a saddle point for mixed strategies.

9.7.5 Computation of Equilibria

It has been shown that a two person zero-sum matrix game always admits a saddle point equilibrium in mixed strategies. An important property of mixed saddle point strategies is that, for each player there is a mixed security strategy and for each mixed security strategy there is a corresponding mixed saddle point strategy.

Using this property, there is a possible way of obtaining the saddle point solution of a matrix game, which can be used to determine the mixed security strategies for each player.

Let us consider the following (2×2) matrix game:

| | | |
|-------|-------|---|
| | P_2 | |
| P_1 | 3 | 0 |
| | -1 | 1 |

Let the mixed strategies of y and z be defined as follows:

$$y = [y_1, y_2]^T$$

$$z = [z_1, z_2]^T$$

For P_1 , our goal is to find the y^* that optimizes $y^T A z$ while P_2 is trying to do his best, i.e. P_2 uses only pure strategies. Therefore, P_2 can be expected to play either $(z_1 = 1, z_2 = 0)$ or $(z_1 = 0, z_2 = 1)$ and under different choices of mixed strategies for P_1 , we can determine the average outcome of the game as shown in Fig 3 by the bold line, which forms the upper envelope to the straight lines drawn. Now, if the mixed strategy $(y_1^* = \frac{2}{5}, y_2^* = \frac{3}{5})$ corresponds to the lowest point of that envelope adopted by P_1 , then the average outcome will be no greater than $\frac{3}{5}$. This implies that the strategy $(y_1^* = \frac{2}{5}, y_2^* = \frac{3}{5})$ is a mixed security strategy for P_1 (and his only one), and thereby, it is his mixed saddle point strategy. From the figure, we can see that the mixed saddle point value is $\frac{3}{5}$.

In order to find z^* , we assume the P_1 adopts pure strategies. Therefore for different mixed strategies of P_2 , the average outcome of the game can be determined to be the bold line, shown in Fig. 4, which forms the lower envelope to the straight lines drawn. Since P_2 is the maximizer, the highest point on this envelope is his average security level. This he can guarantee by playing the mixed strategy which is also his saddle point strategy.

Solving matrix games with larger dimensions One alternative to the graphical solution described above when the dimensions are large (i.e. $n \times m$ games) is to convert the original matrix game into a linear programming model and make use of the powerful algorithms for linear programming in order to obtain the saddle point solutions.

This equivalency of games and LP may be surprising, since a LP problem involves just one decision-maker, but it should be noted that with each LP problem there is an associated problem called the dual LP. The optimal values of the objective functions of the two LPs are equal, corresponding to the value of the game. When solving LP by simplex-type methods, the optimal solution of the dual problem also appears as part of the final tableau.

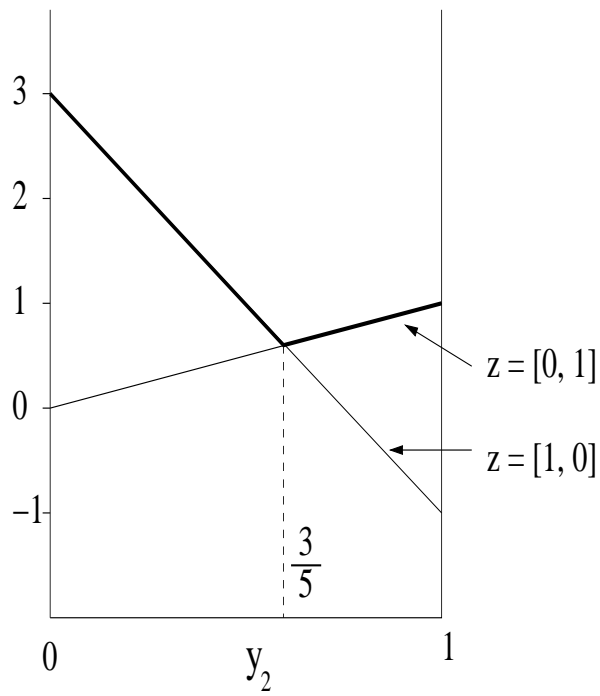


Figure 9.8: Mixed Security strategy for P_1 for the matrix game

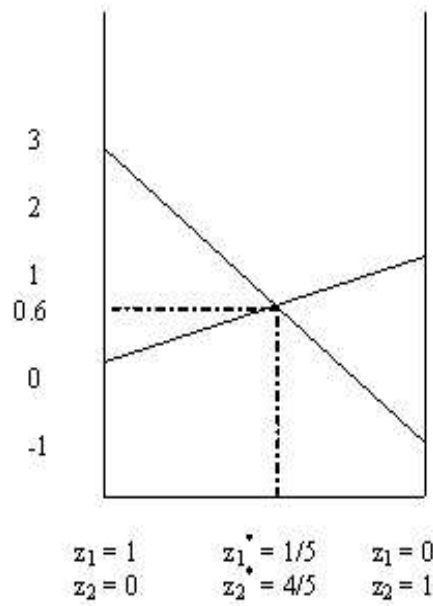


Figure 9.9: Mixed Security strategy for P_2 for the matrix game

9.8 Nonzero Sum Games

The branch of Game Theory that better represents the dynamics of the world we live in is called the theory of non-zero-sum games. Non-zero-sum games differ from zero-sum games in that there is no universally accepted solution. That is, there is no single optimal strategy that is preferable to all others, nor is there a predictable outcome. Non-zero-sum games are also non-strictly competitive, as opposed to the completely competitive zero-sum games, because such games generally have both competitive and cooperative elements. Players engaged in a non-zero sum conflict have some complementary interests and some interests that are completely opposed.

9.8.1 Nash Equilibria

A bi-matrix game is comprised of two $(m \times n)$ dimensional matrices $A = \{a_{ij}\}$ and $B = \{b_{ij}\}$ where each pair of entries $\{a_{ij} b_{ij}\}$ denote the outcome of the game corresponding to a particular pair of decisions made by the players. Being a rational decision maker each player will strive for an outcome which provides him with the lowest possible loss.

Assuming that there are no cooperations between the players and the players make their decisions independently, we now try to find out a noncooperative equilibrium solution. The notion of saddle points in zero sum games is also relevant in non zero sum games, where the equilibrium solution is expected to exist if there is no incentive for any unilateral deviation for the players. Therefore, we have the following definition:

Definition 3.1 *A pair of strategies {row i^* , column j^* } is said to constitute a **Nash Equilibrium** if the following pair of inequalities is satisfied for all $i = 1, \dots, m$ and all $j = 1, \dots, n$:*

$$\begin{aligned} a_{i^*j^*} &\leq a_{ij^*} \\ b_{i^*j^*} &\leq b_{ij^*} \end{aligned}$$

We use a 2 player, single stage game to illustrate the features of a non zero sum game. A and B are the two players, each of them have individual loss functions P_1 and P_2 respectively. The loss functions are represented by the following two matrices:

For A:

| | | |
|-------|---|-------|
| | | P_2 |
| | 1 | 0 |
| P_1 | 2 | -1 |

and for B:

$$\begin{array}{c}
 P_2 \\
 \begin{array}{|c|c|}
 \hline
 2 & 3 \\
 \hline
 1 & 0 \\
 \hline
 \end{array} \\
 P_1
 \end{array}$$

It admits two Nash equilibria, {row 1, col 1} and {row 2, col 2}. The corresponding Nash equilibria is (1,2) and (-1,0).

9.8.2 Admissibility

The previous example shows that a bi-matrix game can admit more than one Nash equilibrium solution, with the equilibrium outcomes being different in each case. This raises the question whether there is a way of choosing one equilibrium over the other. We introduce the concept admissibility as follows:

Better A pair of strategies {row i_1 , column j_1 } is said to be better than another pair of strategies {row i_2 , column j_2 } if $a_{i_1j_1} \leq a_{i_2j_2}$ and $b_{i_1j_1} \leq b_{i_2j_2}$ and if at least one of these inequalities is strict.

Admissibility A Nash equilibrium strategy pair is said to be admissible if there exists no better Nash equilibrium strategy pair.

In the given example, {row 2 ,column 2} is the one that is admissible out of the two Nash equilibrium solutions, since it provides lower costs for both players. This pair of strategies can be described as the most reasonable noncooperative equilibrium solution of the bi-matrix game. In the case when a bimatrix game admits more than one admissible Nash equilibrium the choice becomes more difficult. If the two matrices are as follows:

For A:

$$\begin{array}{c}
 P_2 \\
 \begin{array}{|c|c|}
 \hline
 -2 & 1 \\
 \hline
 -1 & -1 \\
 \hline
 \end{array} \\
 P_1
 \end{array}$$

and for B:

$$\begin{array}{c}
 P_2 \\
 \begin{array}{|c|c|}
 \hline
 -1 & 1 \\
 \hline
 2 & -2 \\
 \hline
 \end{array} \\
 P_1
 \end{array}$$

there are two admissible Nash equilibrium solutions { row 1, column 1}, {row 2, column 2} with the equilibrium outcomes being (-2,-1) and (-1,-2) respectively. This game can lead to regret unless some communication and negotiation is possible.

However if the equilibrium strategies are interchangeable then the ill-defined equilibrium solution accruing from the existence of multiple admissible Nash equilibrium solution can be resolved. This necessarily requires the corresponding outcomes to be the same. Since zero sum matrix games are special types of bi-matrix games (in which case the equilibrium solutions are known to be interchangeable), it follows that there exists some non empty class of bi-matrix games whose equilibrium solutions possess such a property. More precisely :

Multiple Nash equilibria of a bimatrix game (A,B) are interchangeable if (A,B) is strategically equivalent to $(A,-A)$.

9.8.3 The Prisoner's Dilemma

The following example shows how by using Nash's equilibrium, the prisoners can achieve results that yield no regrets, but how by cooperating, they could have done much better. We show the cost of cooperation and denial of wrong doing in form of the following two matrices:

For A:

| | | | |
|----------------|----|----------------|--|
| | | P ₂ | |
| | 8 | 0 | |
| P ₁ | 30 | 2 | |

and for B:

| | | | |
|----------------|---|----------------|--|
| | | P ₂ | |
| | 8 | 30 | |
| P ₁ | 0 | 2 | |

Using Nash equilibrium, the choice is (8,8) which yields no regret for either A or B. However, if the prisoners had cooperated then they would have ended up with (2,2) which is much better for both of them.

9.8.4 Nash Equilibrium for mixed strategies

Nash showed that every non-cooperative game with finite sets of pure strategies has at least one mixed strategy equilibrium pair. We define such pair as a *Nash equilibrium*. For a two-player game, where the matrices A and B define the cost for players 1 and 2 respectively, the strategy (y^*, z^*) is a Nash equilibrium if:

$$\begin{aligned} y^{*T} A z^* &\leq y^T A z^* \quad \forall y \in Y \\ y^{*T} B z^* &\leq y^{*T} B z \quad \forall z \in Z \end{aligned}$$

in which Y and Z are the sets of possible mixed strategies for players 1 and 2 respectively. Remember that the elements of Y and Z are vectors defining

the probability of choosing different strategies. For example, for a $y \in Y$, $y = [y_1, y_2, \dots, y_m]^T$, we have $\sum_{i=1}^m y_i = 1$, in which $y_i \geq 0$ defines the probability of choosing the strategy i .

If a player plays the game according with the strategy defined by the Nash equilibrium, then we say that the player is using a *Nash strategy*. The Nash strategy safeguards each player against attempts by any one player to further improve on his individual performance criterion. Moreover, each player knows the expected cost for the game solution (y^*, z^*) . For player 1 the expected cost is $y^{*T} A z^*$, and for player 2 is $y^{*T} B z^*$. For the case of two players, the Nash equilibrium can be found using quadratic programming. In general, for multi-player games, the Nash equilibrium is found using non-linear programming.

This solution generally assumes that the player know each other's cost matrices and that, when the strategies have been calculated, they are announced at the same instant of time.

Note that the Nash strategy does not correspond in general with the security strategy. When the game has a unique Nash equilibrium for pure strategies, then the Nash equilibrium maximizes the security strategy.

Chapter 10

Sequential Decision Theory

Chapter Status



What does this mean? Check

<http://msl.cs.uiuc.edu/planning/status.html>
for information on the latest version.

These are class notes from CS497 Spring 2003, parts of which were scribed by Xiaolei Li, Warren Shen, Sherwin Tam.

10.1 Basic Definitions

Notation

- $U(x_k)$ the set of decision maker actions from the state x_k
 $u_k \in U(x_k)$
 $\Theta(x_k)$ the set of actions nature can perform in state x_k
 $\theta_k \in \Theta(x_k)$
 $\theta_k \in \Theta(x_k, u_k)$ like above, except that nature responds to decision maker

The state transition equation: $x_{k+1} = f(x_k, u_k, \theta_k)$

The cost functional: $L = \sum_{i=1}^K l(x_i, u_i, \theta_i) + l_F(x_F)$

Use termination actions as before. Also, assume the current state is always known.

10.1.1 Non-Deterministic Forward Projection

If nature is non-deterministic, what will our next state be given our current state and our action that we apply?

Let θ_k be whatever action nature does after we apply u_k in state x_k , and $F(x_k, u_k)$ be the set of states that we can be in after u_k and θ_k is applied when

we were in state x_k . So, we have $\theta_k \in \Theta(x_k, u_k)$ and $F(x_k, u_k) \subseteq X$, where X is the set of all states.

In the non-deterministic case, we have

$$f(x_k, u_k) = \{x_{k+1} \mid \exists \theta_k \in \Theta(x_k, u_k) \text{ such that } x_{k+1} = f(x_k, u_k, \theta_k)\}$$

This is a 1-stage forward projection, where we project all possible states one step forward. Here is a 2-stage forward projection:

$$F(F(x_k, u_k), u_{k+1})$$

This can be further expanded to any number of stages.

10.1.2 Probabilistic Forward Projection

In the probabilistic case, assume we have a probability distribution over the possible actions nature can do given our action u_k in state x_k . Thus, the probability that nature performs action θ_k could be written as

$$P(\theta_k \mid x_1 \cdots x_k, u_1, \cdots, u_k, \theta_1 \cdots \theta_{k-1})$$

which is the probability given everything that has happened in the past.

This is too big - so our solution is to arbitrarily knock out stuff until we're happy. To make it not so arbitrary, we'll go by the Markovian assumption and say that the probability depends only on local values. So, now we have

$$P(\theta_k \mid x_k)$$

Now, given this probability as well as the fact that we're in state x_k and apply action u_k , we want to get the probability that we'll get to state x_{k+1} . We can simply combine the state transition function, $x_{k+1} = f(x_k, u_k, \theta_k)$, and $P(\theta_k \mid x_k)$ to get $P(x_{k+1} \mid x_k, u_k)$. This is a 1-state probabilistic forward projection.

A 2-stage probabilistic forward projection is the probability that we'll get to a state x_{k+2} from x_k . This probability is $P(x_{k+2} \mid x_k, u_k, u_{k+1})$. In order to get this to a form we know, we can marginalize variables and get

$$P(x_{k+2} \mid x_k, u_k, u_{k+1}) = \sum_{x_{k+1}} P(x_{k+2} \mid x_{k+1}, u_{k+1})P(x_{k+1} \mid x_k, u_k)$$

10.1.3 Strategies

A strategy $\gamma : x \rightarrow u$ is a plan that tells the decision maker what action to take given a state ($u_k = \gamma(x_k)$). Remember that our cost functional is $L = l_i(x_i) + \sum_{i=1}^k l(x_i, u_i, \theta_i) + l_f(x_f)$. How should we choose our strategy?

In the non-deterministic case, we can think of our strategy problem as a table, where each row specifies a state we could be in right now, and each column specifies states we could go to if we applied $\gamma(x_i)$. Wherever there is a 1, that means that the state is reachable from our current state.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | . | . | . | n |
| 1 | 0 | 1 | 1 | . | . | . | 1 |
| 2 | 1 | 0 | 1 | . | . | . | 1 |
| 3 | 0 | 1 | 0 | . | . | . | 0 |
| . | | | | | | | |
| . | | | | | | | |
| . | | | | | | | |
| n | 0 | 0 | 1 | . | . | . | 0 |

For example, if we were in state 2 and we applied the strategy that this table represents, we could end up in 1, 3...n, but not 2 because there is a 0 in that position. Since nature is non-deterministic, we don't know which state we'll end up in.

So, given our cost functional L , we want to choose γ to minimize the worst-case cost.

In the probabilistic case, we can also think of our strategy in terms of a table, except that instead of having boolean values in the entries, we have probabilities:

| | | | | | | | |
|---|-----|-----|-----|---|---|---|-----|
| | 1 | 2 | 3 | . | . | . | .n |
| 1 | .6 | .02 | .01 | . | . | . | .3 |
| 2 | .02 | .4 | .2 | . | . | . | .13 |
| 3 | .2 | .4 | .02 | . | . | . | .2 |
| . | | | | | | | |
| . | | | | | | | |
| . | | | | | | | |
| n | 0.0 | .14 | .33 | . | . | . | .43 |

For example, in the entry (2, 3), we have $P(x_{k+1} = 3 \mid x_k = 2, u_k = \gamma(2)) = .2$.

Given our cost functional L , in the probabilistic case, we want to choose γ to minimize the expected cost.

10.2 Dynamic Programming over Discrete Spaces

Setting it up As before, we have stages $1, 2, \dots, K - 1, K, F$. Here are some definitions, which have been adapted from our dynamic programming for games without nature:

In the non-deterministic case:

$$\begin{aligned}
 L_{F,F}^*(x_F) &= l_f(x_f) \\
 L_{K,F}^*(x_K) &= \min_{u_K} \max_{\theta_K} \{l(x_K, u_K, \theta_K) + l_F(x_F)\} \\
 L_{1,F}^*(x_1) &= \min_{u_1} \max_{\theta_1} \min_{u_2} \max_{\theta_2} \cdots \min_{u_K} \max_{\theta_K} \left\{ l_I(x_1) + \sum_{i=1}^K l(x_i, u_i, \theta_i) + l_F(x_F) \right\}
 \end{aligned}$$

In the probabilistic case:

$$\begin{aligned} L_{F,F}^*(x_F) &= l_f(x_f) \\ L_{K,F}^*(x_K) &= \min_{u_K} \{E_{\theta_K} [l(x_K, u_K, \theta_K) + l_F(x_F)]\} \\ L_{1,F}^*(x_1) &= \min_{u_1 \dots u_k} \left\{ E_{\theta_1 \dots \theta_K} \left[l_I(x_1) + \sum_{i=1}^K l(x_i, u_i, \theta_i) + l_F(x_F) \right] \right\} \end{aligned}$$

where E_{θ_K} is the expectation of θ_K , and $E_{\theta_1 \dots \theta_K}$ is the expectation over $\theta_1 \dots \theta_K$

Finding the minimum loss Now, to find the loss from any stage k , we use dynamic programming, as before. In the non-deterministic case we have

$$L_{k,F}^*(x_k) = \min_{u_k \in U(x_k)} \max_{\theta_K \in \Theta(x_k, u_k)} \{L_{k+1,F}^*(x_{k+1}) + l(x_k, u_k, \theta_k)\}$$

where x_{k+1} is defined by our state transition equation $x_{k+1} = f(x_k, u_k, \theta_k)$.

In the probabilistic case we have

$$\begin{aligned} L_{k,F}^*(x_k) &= \min_{u_k \in U(x_k)} \{E_{\theta_k} [L_{k+1,F}^*(x_{k+1}) + l(x_k, u_k, \theta_k)]\} \\ &= \min_{u_k \in U(x_k)} \left\{ \sum_{\theta_k} (L_{k+1,F}^*(x_{k+1}) + l(x_k, u_k, \theta_k)) P(\theta_k | x_k, u_k) \right\} \end{aligned}$$

However, if $l(x_k, u_k, \theta_k) = l(x_k, u_k)$, then our formula becomes

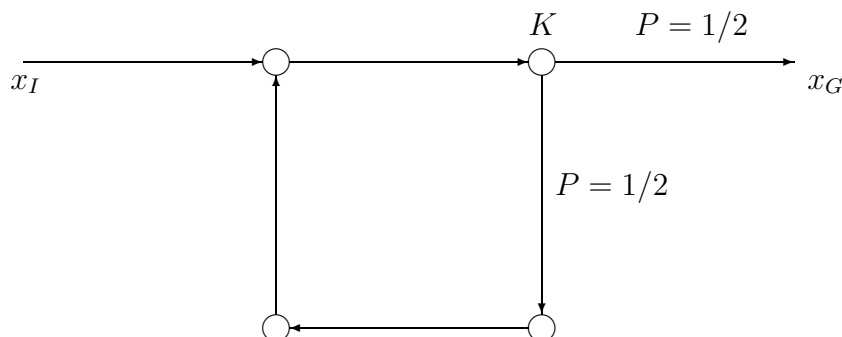
$$L_{k,F}^*(x_k) = \min_{u_k \in U(x_k)} \left\{ l(x_k, u_k) + \sum_{x_{k+1}} (L_{k+1,F}^*(x_{k+1})) P(x_{k+1} | x_k, u_k) \right\}$$

Here, we've just made the θ go away, but in reality it's just hiding inside of $P(x_{k+1} | x_k, u_k)$.

Issues with Cycles As before assume termination actions to end our game when we reach a goal state, and also assume that K is unknown. What if there are cycles in our problem, where a series of actions could potentially bring you back to the same state? How do we make sure that our program terminates?

In the non-deterministic case, there must be no negative cycles (in reality, there must be no minimax cycles), and there also must be a way to escape or avoid all positive cycles. If there were negative cycles, meaning that even with nature we can perform actions in a cycle such that we still have negative loss, then the optimal strategy would be to go around forever to minimize loss. If there were positive cycles that we couldn't escape from or avoid, then it would be possible for nature to keep on sending us through the cycle forever.

In the probabilistic case, as long as no transitions at the start of a cycle is 1, then we will terminate. For example, suppose we had this graph, where nodes are states and edges are transitions:



Assume that all transitions have a loss of 1. If we were at state K , we would want to go to x_G . If we go straight from x_1 to x_g , we we'll have a total loss of 3. However, at K there is a chance of $1/2$ that we will go around the cycle. If we go around, we'll acquire an additional loss of 4 each time. Thus, the expected loss becomes

$$\begin{aligned} E[L] &= \frac{1}{2}(3) + \frac{1}{4}(7) + \dots \\ &= 3 + \sum_{i=0}^{\infty} \frac{1}{2^{i+1}}(4i) \\ &< \infty \end{aligned}$$

If the probabilities are less than 1, then the expected loss converges on some finite value, meaning that we will terminate. However, no matter how far we go in the future, there will be some exponentially small chance that we will keep going around the cycle. To calculate L_k^* , when do we stop? We can pick some threshold ϵ and terminate when $\max_{x \in X} |L_{k+1}^*(x) - L_k^*(x)| < \epsilon$.

10.3 Infinite Horizon Problems

In an infinite-horizon MDP, K (number of stages) is infinite and there are no termination actions. In this situation, the accumulated loss ($\sum_{i=1}^{\infty} l(x_i, u_i)$) will be ∞ . This means that we will end up with an ∞ -loss plan, which would be quite useless. There are two solutions to the problem and they are described below. The first one is to average the loss-per-stage and derive the limit. The second is to discount losses in the future. We will look at both of them but focus in depth on the latter.

10.3.1 Average Loss-Per-Stage

The intuition behind this idea is to basically limit the horizon. In this manner, we could figure out the average loss per stage and calculate the limit as $K \rightarrow \infty$. The exact equation is shown below.

$$\lim_{K \rightarrow \infty} \frac{1}{K} E \left\{ \sum_{i=1}^{K-1} l(x_i, u_i, \theta_i) \right\}$$

10.3.2 Discounted Loss

An alternative to the average loss-per-stage scheme is the concept of *discounted loss*. The intuition is that losses in the far future do not count too much. So the discounted loss scheme will gradually reduce the losses in the future to zero. This will force $\sum_{i=1}^{\infty} l(x_i, u_i, \theta_i)$ to converge. The exact definition of the discounted loss functional shown below. The α is known as the *discount factor*. A larger α gives more weight to the future.

$$L = \lim_{K \rightarrow \infty} E \left\{ \sum_{i=1}^{K-1} \alpha^{i-1} \times l(x_i, u_i, \theta_i) \right\} \quad 0 < \alpha < 1 \quad (10.1)$$

With the above definition, it is clear that $\lim_{i \rightarrow \infty} \alpha^{i-1} = 0$. Thus as i approaches ∞ , the term inside the summation will be 0. Therefore, the entire equation will converge.

10.3.3 Optimization in the Discounted Loss Model

Using the discounted loss model described in the previous section, it is now possible to optimize infinite-horizon MDPs using dynamic programming (DP). We need to find the best policy (γ) such that L is minimized (i.e., optimize Equation (10.1)). Before we look at dynamic programming, let us examine how L accumulates as K increases. When $K = 1$, there are no losses. As K increments, additional loss terms are attached on as shown below.

| Stage | L_K^* |
|---------|-----------------------------------|
| $K = 1$ | 0 |
| $K = 2$ | l_1 |
| $K = 3$ | $l_1 + \alpha l_2$ |
| $K = 4$ | $l_1 + \alpha l_2 + \alpha^2 l_3$ |
| | \vdots |

Figure 10.1: Discounted Loss Growth

10.3.4 Forward Dynamic Programming

From Figure (10.1), we can easily envision how forward dynamic programming can solve for L . We can set L_1^* to 0 and at each iteration after, find the best next step. In other words, search through all possible γ 's and find the one that gives the least l_{i+1} where i is the current stage. As i increases, each $|L_{i+1}^* - L_i^*|$ will get smaller and smaller because $\lim_{i \rightarrow \infty} \alpha^{i-1} = 0$. And we can use a condition similar to Equation (??) that will allow the DP to stop after so many stages. This process sounds fairly easy on paper but turns out to be rather difficult in practice. Therefore, we will instead use backwards dynamic programming.

10.3.5 Backwards Dynamic Programming

Similar to forward dynamic programming, the backwards method will work in an iterative fashion. The main difference is that it will start at the end. What is the end for our problem? It's stage K . But in the infinite-horizon MDP, K is equal to ∞ . This presents a problem in that we cannot annotate stage ∞ ; we will use a notational trick to get around this problem.

Recall in Figure 10.1 that each dynamic programming step added a term to L . In the forward DP method, we can envision this process as shown in Figure 10.2. In the backward DP method, we can envision the growth pattern in Figure 10.2 as being flipped upside down in Figure 10.3.



Figure 10.2: **FDP Growth** Figure 10.3: **BDP Growth**

An observation we could make about Figure 10.3 is that the bottom of the *stage list* is growing into the past. In other words, the stages in the previous step of the DP is being slid into the future. Due to discounted loss, we will need to multiple them by α because they're now further in the future. To make this process natural in terms of notation, we will define a new term J^* as below.

$$J_{K-k}^*(x_k) = \alpha^{-k} L_k^*(x_k)$$

For example, if K was equal to 5, L_5^* will be equal to J_0^* and L_1^* will be equal to J_4^* . Intuitively, J_i^* is the expected loss for an i -stage optimal strategy. Recall that the original dynamic programming had the solution of:

$$L_K^*(x) = 0 \quad \forall x \in X$$

$$L_k^*(x) = \min_{u_k \in U(x_k)} E_{\theta_k} \{ \alpha^k l(x_k, u_k, \theta_k) + L_{k+1}^*(f(x_k, u_k, \theta_k)) \} \quad (10.2)$$

Equipped with the new J notation, we will re-write Equation (10.2) as the following by replacing all L 's with J 's.

$$\alpha^k J_{K-k}^*(x_k) = \min_{u_k \in U(x_k)} E_{\theta_k} \{ \alpha^k l(x_k, u_k, \theta_k) + \alpha^{k+1} J_{K-k-1}^*(f(x_k, u_k, \theta_k)) \}$$

We will then divide out α^k from the equation and also re-write $(K - k)$ as i . This will leave us with the following.

$$J_i^*(x_k) = \min_{u_k \in U(x_k)} E_{\theta_k} \{ l(x_k, u_k, \theta_k) + \alpha J_{i-1}^*(f(x_k, u_k, \theta_k)) \}$$

And more generally,

$$J^*(x) = \min_{u \in U(x)} E_{\theta} \{ l(x, u, \theta) + \alpha J^*(f(x, u, \theta)) \}$$

Note that now it is possible to enumerate through the backwards DP by starting at J_0^* . It would be just like solving the original BDP by starting at L_K^* . Furthermore, if we removed the *min* term in front of the equation, it will also allow us to evaluate a particular strategy:

$$J_{\gamma}(x) = E_{\theta} \{ l(x, u, \theta) + \alpha J^*(f(x, u, \theta)) \}$$

It is also possible that our loss function could be independent of nature. That is $l(u, x, \theta) = l(x, u)$. We can then further simplify the last pair of equations to the following. For simplicity, we will rewrite $f(x, u, \theta)$ as x' .

$$J^*(x) = \min_{u \in U(x)} \left\{ l(x, u) + \alpha \sum_{x'} P(x'|x, u) J^*(x') \right\} \quad (10.3)$$

$$J_{\gamma}(x) = l(x, u) + \alpha \sum_{x'} P(x'|x, u) J_{\gamma}(x') \quad (10.4)$$

Notice that the loss function no longer has θ as a parameter. This allows us to remove the expectation of nature from the equation. However, since x' still depends on nature, we simply wrote out the definition of expectation as a weighted sum of all x' 's. This hides θ amongst the probabilities. For a given fixed strategy, it is now possible to find J^* by iteratively evaluating Equation (10.4) until a condition such as Equation (??) is satisfied. This is known as *value iteration*.

10.3.6 Policy Iteration

The method of finding an optimal strategy, γ^* , using Equation (10.3) is known as *policy iteration*. The process can be summarized below.

1. Guess a strategy γ .
2. Evaluate γ using Equation (10.4).
3. Use Equation (10.3) to find an improved γ' .
4. Go back to Step 2 and repeat until no improvements occur in Step 3.

Example We shall illustrate the above algorithm through a simple example. Suppose we have $X = \{1, 2\}$ and $U = \{a, b\}$. Let Figures 10.4 and 10.5 be the probabilities of actions a and b . In addition, let the discount factor α equal $\frac{9}{10}$.

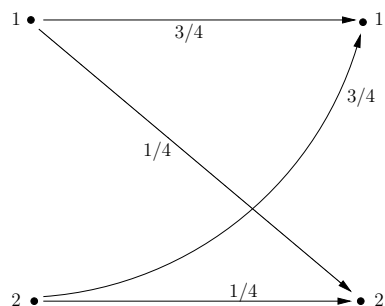


Figure 10.4: **Action a**

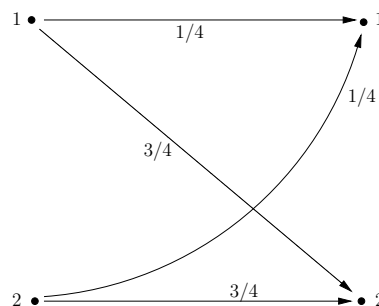


Figure 10.5: **Action b**

Assuming that $l(x, u, \theta) = l(x, u)$, we have the following loss values.

$$\begin{aligned} l(1, a) &= 2 & l(1, b) &= \frac{1}{2} \\ l(2, a) &= 1 & l(2, b) &= 3 \end{aligned}$$

Now, let us follow the algorithm described earlier. **Step 1** is to choose an initial γ . We will randomly choose one that is $\gamma(1) = a$ and $\gamma(2) = b$. In other words, choose action a when in state 1 and choose action b when in state 2.

Step 2 is to evaluate γ using Equation (10.4). This results in the following pair of equations. With them, we see that there are two unknowns with two equations and thus can be easily solved.

$$J_\gamma(1) = l(1, a) + \frac{9}{10} \left(\frac{3}{4} J_\gamma(1) + \frac{1}{4} J_\gamma(2) \right)$$

$$J_\gamma(2) = l(2, b) + \frac{9}{10} \left(\frac{1}{4} J_\gamma(1) + \frac{3}{4} J_\gamma(2) \right)$$

$$J_\gamma(1) = 24.12 \quad J_\gamma(2) = 25.96$$

Step 3 is to minimize J_γ . With the answers above, we can now evaluate Equation (10.3) by putting them in place of $J^*(x')$. This will let us find a new γ which we can repeat in Step 2 (which will turn out to be $\gamma(1) = b$ and $\gamma(2) = a$). This process is relatively simple and is guaranteed to find a global minimum. However, when the number of states are large and the number of actions are large, the system of equations can become impossible to solve practically.

10.4 Dynamic Programming over Continuous Spaces

I wrote this in 1997 for CS326a at Stanford. It needs to be better integrated in to the current context. It should also be enhanced to allow nature to be added.

This section describes how the dynamic programming principle can be used to compute optimal motion plans. Optimality is expressed with respect to a desired criterion, and the method can only provide a solution that is optimal for a specified resolution. The method generally applies to a variety of holonomic and nonholonomic problems, and can be adapted to many other problems that involve complications such as stochastic uncertainty in prediction. The primary drawback of the approach is that the computation time and space are exponential in the dimension of the C-space. This limits its applicability to three or four-dimensional problems (however, for many problems it is the only known method to obtain optimal solutions). Although there are connections between dynamic programming in this context and in graph search, its use in these notes applies to continuous spaces. The dynamic programming formulation presented here is more similar to what appears in optimal control literature [19, 110, 433].

10.4.1 Reformulating Motion Planning

Recall that the goal of the basic motion planning problem is to compute a path $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$ such that $\tau(0) = q_{init}$ and $\tau(1) = q_{goal}$, when such a path exists. In the case of nonholonomic systems, velocity constraints must additionally be satisfied.

We are next going to add some new concepts to the standard motion planning formulation. First, it will be helpful to define time, both for motion planning problems that vary over time and to help in the upcoming concepts. Since time is irrelevant for basic motion planning, it can be considered in this case as an auxiliary variable that only assists in the formulation. Suppose that there is some initial time, $t = 0$, at which the robot is at q_{init} . Suppose also that there is some final time, T_f (one would like to at least have the robot at the goal before $t = T_f$).

Recall that \dot{q} represents the velocity of the robot in the configuration space. Suppose that \mathcal{C} is an m -dimensional configuration space, and that u is a continuous, vector-valued function that depends on time: $u : [0, T_f] \rightarrow \mathbb{R}^m$. If we select u , and let $\dot{q}(t) = u(t)$, then a trajectory has been specified for the robot: $q(0) = q_{init}$

and for $0 < t \leq T_f$, we can obtain

$$q(t) = q(0) + \int_0^t \dot{q}(t') dt' = q(0) + \int_0^t u(t') dt'. \quad (10.5)$$

The function u can be considered as a *control input*, because it allows us to move the robot by specifying its velocity. As will be seen shortly, the case of $\dot{q}(t) = u(t)$ corresponds to a holonomic planning problem. Suppose that we can choose any control input such that for all t it is either normalized, $\|u(t)\| = 1$, or $u(t) = 0$. This implies that we can locally move the robot in any allowable direction from its tangent space. For nonholonomic problems, one will only be allowed to move the robot through a function of the form $\dot{q} = f(q(t), u(t))$. For example, as described in [437], p. 432, the equations for the nonholonomic car robot can be expressed as $\dot{x} = v \cos(\theta)$, $\dot{y} = v \sin(\theta)$, and $\dot{\theta} = \frac{v}{L} \tan(\phi)$. Using the notation in these notes, $(\dot{x}, \dot{y}, \dot{\theta})$ becomes $(\dot{q}_1, \dot{q}_2, \dot{q}_3)$, and (v, ϕ) becomes (u_1, u_2) . The function f can be considered as a kind of interface between the user and the robot. Commands are specified through $u(t)$, but the resulting velocities in the configuration space get transformed using f (which in general prevents the user from directly controlling velocities).

Incorporating optimality If we want to consider optimality, then it will be helpful to define a function that assigns a cost to a given trajectory that is executed by the robot. One can also make this cost depend on the control function. For example, if the control is an accelerator of a car, then one might want to penalize rapid accelerations which use more fuel. A *loss functional* is defined that evaluates any configuration trajectory and control function:

$$L = \int_0^{T_f} l(q(t), u(t)) dt + Q(q(T_f)). \quad (10.6)$$

The integrand $l(q(t), u(t))$ represents an instantaneous cost, which when integrated can be imagined as the total amount of energy that is expended. The term $Q(q(T_f))$ is a final cost that can be used to induce a preference over trajectories that terminate in a goal region of the configuration space.

The loss functional can be reduced to a binary function when encoding a basic path planning problem that does not involve optimality. The loss functional can be simplified to $L = Q(q(T_f))$. We take $Q(q(T_f)) = 0$ if $q(T_f) = q_{goal}$, and $Q(q(T_f)) = 1$ otherwise. This partitions the space of control functions into two classes: control functions that cause the basic motion planning problem to be solved receive zero loss; otherwise, unit loss is received.

The previous formulation considered all control inputs that achieve the goal to be equivalent. As another example, the following measures the path length for control inputs that lead to the goal:

$$L = \begin{cases} \int_0^{T_f} \|\dot{q}(t)\| dt & \text{if } q(T_f) = q_{goal} \\ \infty & \text{otherwise} \end{cases}. \quad (10.7)$$

The term $\int_0^{T_f} \|\dot{q}(t)\| dt$ measures path length, and recall that $\dot{q}(t) = u(t)$ for all t .

There is a small technicality about considering optimal collision-free paths. For example, the visibility-graph method produces optimal solutions, but these paths must graze the obstacles. Any path that maps into \mathcal{C}_{free} can be replaced by a shorter path that still maps into \mathcal{C}_{free} , but might come closer to obstacles. The problem exists because \mathcal{C}_{free} is an open set, and can be fixed by allowing the path to map into \mathcal{C}_{valid} (which is the closure of \mathcal{C}_{free}). If one still must use \mathcal{C}_{free} , then the optimal path that maps into \mathcal{C}_{valid} will represent an *infimum* (a lower bound that can't quite be reached) over paths that only map into \mathcal{C}_{free} .

A discrete-time representation The motion planning problem can alternatively be characterized in discrete time. For the systems that we will consider, discrete-time representations can provide arbitrarily close approximations to the continuous case, and facilitate the development of the dynamic programming algorithm.

With the discretization of time, $[0, T_f]$ is partitioned into *stages*, denoted by $k \in \{1, \dots, K + 1\}$. Stage k refers to time $(k - 1)\Delta t$. The final stage is given by $K = \lfloor T_f/\Delta t \rfloor$. Let q_k represent the configuration at stage k . At each stage k , an *action* u_k can be chosen from an *action space* U . Because

$$\frac{dq}{dt} = \lim_{\Delta t \rightarrow 0} \frac{q(t + \Delta t) - q(t)}{\Delta t}, \quad (10.8)$$

the equation $\dot{q}(t) = u(t)$ can be approximated as

$$q_{k+1} = q_k + \Delta t u_k, \quad (10.9)$$

in which $q_k = q(t)$, $q_{k+1} = q(t + \Delta t)$, and $u_k = u(t)$. As an example of how this representation approximates the basic motion planning problem, consider the following example. Suppose $\mathcal{C}_{free} \subseteq \mathbb{R}^2$. It is assumed that $\|u_k\| = 1$ and, hence, the space of possible actions can be sufficiently characterized by the parameter $\phi_k \in [0, 2\pi)$. The discrete-time transition equation becomes

$$q_{k+1} = q_k + \Delta t \begin{bmatrix} \cos(\phi_k) \\ \sin(\phi_k) \end{bmatrix}. \quad (10.10)$$

At each stage, the direction of motion is controlled by selecting ϕ_k . Any K -segment polygonal curve of length $K\Delta t$ can be obtained as a possible trajectory of the system. If an action is included that causes no motion, shorter polygonal curves can also be obtained.

In general, a variety of holonomic and nonholonomic problems can also be approximated in discrete time. The equation $\dot{q} = f(q(t), u(t))$ can be approximated by a transition equation of the form $q_{k+1} = f_k(q_k, u_k)$.

A discrete-time representation of the loss functional can also be defined:

$$L(q_1, \dots, q_F, u_1, \dots, u_K) = \sum_{k=1}^K l_k(q_k, u_k) + l_{K+1}(q_F), \quad (10.11)$$

in which l_k and l_{K+1} serve the same purpose as l and Q in the continuous-time loss functional.

The basic motion planning problem can be represented in discrete time by letting $l_k = 0$ for all $k \in \{1, \dots, K\}$, and defining the final term as $l_{K+1}(q_F) = 0$ if $q_k = q_{goal}$, and $l_{K+1}(q_F) = 1$ otherwise. This gives equal preference to all trajectories that reach the goal. To approximate the problem of planning an optimal-length path, $l_k = 1$ for all $k \in \{1, \dots, K\}$. The final term is then defined as $l_{K+1}(q_F) = 0$ if $q_k = q_{goal}$, and $l_{K+1}(q_F) = \infty$ otherwise.

10.4.2 The Algorithm

This section presents algorithm issues that result from computing approximate optimal motion strategies. Variations of this algorithm, which apply to a variety of motion planning problems are discussed in detail in [448]. The quality of this approximation depends on the resolution of the representation chosen for the configuration space and action space. The efforts are restricted to obtaining approximate solutions for three primary reasons: 1) known lower-bound hardness results for basic motion planning and a variety of extensions; 2) exact methods often depend strongly on specialized analysis for a specific problem class; and 3) the set of related optimal-control and dynamic-game problems for which analytical solutions are available is quite restrictive. The computational hardness results have curbed many efforts to find efficient, complete algorithms to general motion planning problems. In [651] the basic motion planning problem was shown to be PSPACE-hard for polyhedral robots with n links. In [122] it was shown that computing minimum-distance paths in a 3-D workspace is NP-hard. It was also shown that the compliant motion control problem with sensing uncertainty is nondeterministic exponential time hard. In [652] it was shown that planning the motion of a disk in a 3-D environment with rotating obstacles is PSPACE-hard. In [654], a 3-D pursuit-evasion problem is shown to be exponential time hard, even though there is perfect sensing information. Such results have turned motion planning efforts toward approximate techniques. For example, a polynomial-time algorithm is given in [606] for computing epsilon approximations of minimum-distance paths in a 3-D environment. Also, randomized techniques are used to compute solutions for high degree-of-freedom problems that are unapproachable by complete methods [16, 51, 383, 734].

The second motivation for considering approximate solutions is to avoid specialized analysis of particular cases, with the intent of allowing the algorithms to be adaptable to other problem classes. Of course, in many cases there is great value in obtaining an exact solutions to a specialized class of problems. The approach described in this paper can be considered as a general way to approximate solutions that might be sufficient for a particular application, or the approach might at least provide some understanding of the solutions.

The final motivation for considering approximate solutions is that the class of

related optimal-control and dynamic-game problems that can be solved directly is fairly restrictive. In both control theory and dynamic game theory, the classic set of problems that can be solved are those with a linear transition equation and quadratic loss functional [19, 37, 110].

The algorithm description is organized into three parts. First, the general principle of optimality is described, which greatly reduces the amount of effort that is required to compute optimal strategies. The next part describes how cost-to-go functions are computed as an intermediate representation of the optimal strategy. The third part describes how the cost-to-go is used as a navigation function to execute the represented strategy (i.e., selecting optimal actions during on-line execution). Following this, basic complexity assessments are given.

Exploiting the principle of optimality Because the decision making expressed in $q_{k+1} = f_k(q_k, u_k)$ is iterative, the dynamic programming principle can generally be employed to avoid brute-force enumeration of alternative strategies, and it forms the basis of our algorithm. Although there are obvious connections to dynamic programming in graph search, it is important to note the distinctions between Dijkstra's algorithm and the usage of the dynamic programming principle in these notes. In optimal control theory, the dynamic programming principle is represented as a differential equation (or difference equation in discrete time) that can be used to directly solve a problem such as the linear-quadratic Gaussian regulator [420], or can be used for computing numerical approximations of optimal strategies [431]. In the general case, the differential equation is expressed in terms of time-dependent *cost-to-go* functions. The cost-to-go is a function on the configuration space that expresses the cost that is received under the implementation of an optimal strategy from that particular configuration and time. In some cases, the time index can be eliminated, as in the special case of values stored at vertices in the execution of Dijkstra's algorithm.

For the discrete-time model, the dynamic programming principle is expressed as a difference equation (in continuous time it becomes a differential equation). The cost-to-go function at stage k is defined as

$$L_k^*(q_k) = \min_{u_k, \dots, u_K} \left\{ \sum_{i=k}^K l_i(q_F, u_i) + l_{K+1}(q_F) \right\}. \quad (10.12)$$

The cost-to-go can be separated:

$$L_k^*(q_k) = \min_{u_k} \min_{u_{k+1}, \dots, u_K} \left\{ l_k(q_k, u_k) + \sum_{i=k+1}^K l_i(q_F, u_i(q_F)) + l_{K+1}(q_F) \right\}. \quad (10.13)$$

The second *min* does not affect the l_k term; thus, it can be removed to obtain

$$L_k^*(q_k) = \min_{u_k} \left[l_k(q_k, u_k) + \min_{u_{k+1}, \dots, u_K} \left\{ \sum_{i=k+1}^K l_i(q_F, u_i(q_F)) + l_{K+1}(q_F) \right\} \right]. \quad (10.14)$$

The second portion of the *min* represents the cost-to-go function for stage $k + 1$, yielding [63]:

$$L_k^*(q_k) = \min_{u_k} \{l_k(q_k, u_k(q_k)) + L_{k+1}^*(q_{k+1})\}. \quad (10.15)$$

This final form represents a powerful constraint on the set of optimal strategies. The optimal strategy at stage k and configuration q depends only cost-to-go values at stage $k + 1$. Furthermore, only the particular cost-to-go values that are reachable from the transition equation, $q_{k+1} = f(q_k, u_k)$, need to be considered. The dependencies are local; yet, the globally-optimal strategy is characterized.

Iteratively approximating cost-to-go functions An optimal strategy can be computed by successively building approximate representations of the cost-to-go functions. One straightforward way to represent a cost-to-go function is to specify its values at each location in a discretized representation of the configuration space. Note that this requires visiting the entire configuration space to determine a strategy. Instead of a path, however, the resulting solution can be considered as a feedback strategy. From any configuration, the optimal action will be easily determined. Note that the cost-to-go function is encoding a globally-optimal solution which must take into account all of the appropriate geometric and topological information at a given resolution. Artificial potential functions have often been constructed very efficiently in motion planning approaches; however, these approaches heuristically estimate the cost-to-go and are typically prone to have local minima [51, 394].

The first step is to construct a representation of L_{K+1}^* . The final term, $l_{K+1}(q_{K+1})$, of the loss functional is directly used to assign values of $L_{K+1}^*(q_F)$ at discretized locations. Typically, $l_{K+1}(q_F) = 0$ if q_F lies in the goal region, and $l_{K+1}(q_F) = \infty$ otherwise. This only permits trajectories that terminate in the goal region. If the goal is a point, it might be necessary to expand the goal into a region that includes some of the quantized configurations.

The dynamic programming equation (10.15) is used to compute the next cost-to-go function, L_K^* , and subsequent cost-to-go functions. For each quantized configuration, q_k , a quantized set of actions $u_k \in U$ are evaluated. For a given action u_k , the next configuration obtained by $q_{k+1} = f(q_k, u_k)$ generally might not lie on a quantized configuration. See Figure 10.6.a. Linear interpolation between neighboring quantized configurations can be used, however, to obtain the appropriate loss value without restricting the motions to the grid (see Figure 10.6.a). Suppose for example, that for a one-dimensional configuration space, $L_{k+1}^*[i]$ and $L_{k+1}^*[i + 1]$ represent the loss values for some configurations q_i and q_{i+1} . Suppose that the transition equation, f_k , yields some q that is between q_i and q_{i+1} . Let

$$\alpha = \frac{q_{i+1} - q}{q_{i+1} - q_i}. \quad (10.16)$$

Note that $\alpha = 1$ when $q = q_i$ and $\alpha = 0$ when $q = q_{i+1}$. The interpolated loss can

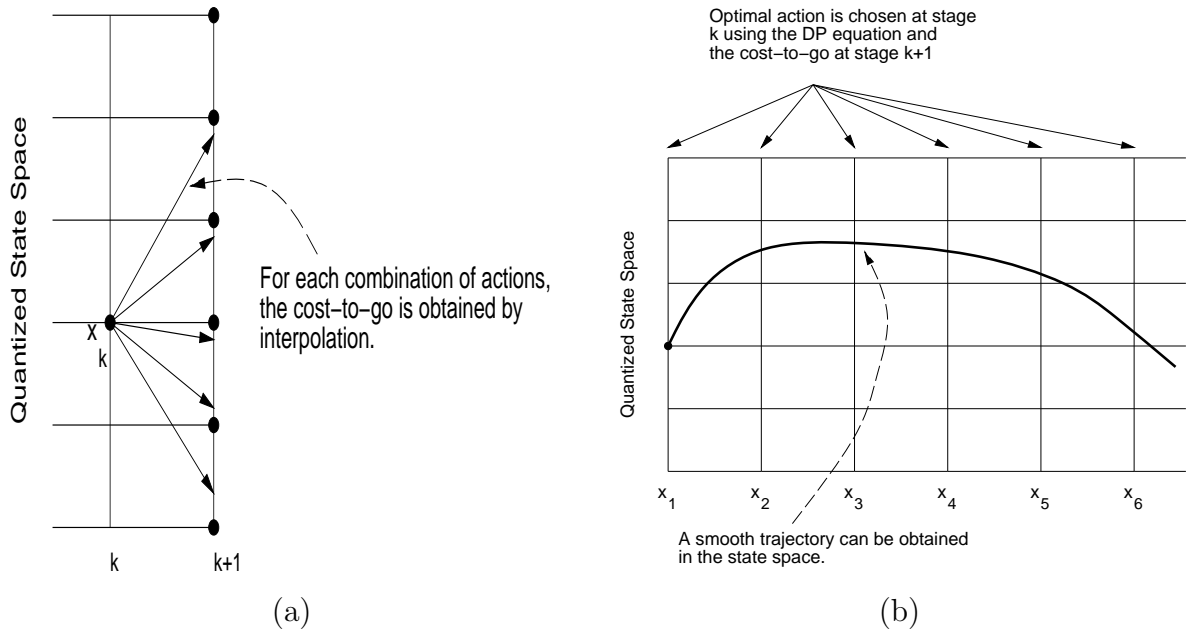


Figure 10.6: The computations are illustrated with a one-dimensional configuration space. (a) The cost-to-go is obtained from at the next stage by interpolation of the values at the neighboring quantized configurations. (b) During execution, interpolation can also be used to obtain a smooth trajectory.

be expressed as

$$L_{k+1}^*(q_{k+1}) \approx \alpha L_{k+1}^*[i] + (1 - \alpha) L_{k+1}^*[i + 1]. \quad (10.17)$$

In an m -dimensional C-space, interpolation can be performed between 2^m neighbors. For example, if $\mathcal{C} = \mathbb{R}^2$, the interpolation can be computed as

$$L_{k+1}^*(q_{k+1}) \approx \alpha\beta L_{k+1}^*[i, j] + (1 - \alpha)\beta L_{k+1}^*[i + 1, j] + \alpha(1 - \beta) L_{k+1}^*[i, j + 1] + (1 - \alpha)(1 - \beta) L_{k+1}^*[i + 1, j + 1] \quad (10.18)$$

in which $\alpha, \beta \in [0, 1]$ are coefficients that express the normalized distance to the neighbors in the q_1 and q_2 directions, respectively. For example $\alpha = 1$, and $\beta = 0$ when q_{k+1} lies at the configuration represented by index $[i, j + 1]$. Other schemes, such as quadratic interpolation, can be used to improve numerical accuracy at the expense of computation time [433]. Convergence properties of the quantization and interpolation are discussed in [63, 69]. Interpolation represents an important step that overcomes the problems of measuring Manhattan distance due to quantization. Note that for some problems, however, interpolation might not be necessary. Suppose for example, that the robot is a manipulator that has independently-controlled joints. During each stage, each joint can be moved clockwise, counterclockwise, or not at all. These choices will naturally result in motions that fall directly onto a grid in the configuration space.

For a motion planning problem, the obstacle constraints must additionally be taken into account. The constraints can be directly evaluated each time to determine whether each q_{k+1} lies in the free space, or a bitmap representation of the configuration space can be used for quick evaluations (an efficient algorithm for building a bitmap representation of \mathcal{C}_{free} is given in [385]).

Note that L_K^* represents the cost of the optimal one-stage strategy from each configuration q_k . More generally, L_{K-i}^* represents the cost of the optimal $(i+1)$ -stage strategy from each configuration q_{K+1} . For a motion planning problem, one is typically concerned only with strategies that require a finite number of stages before terminating in the goal region. For a small, positive δ the dynamic programming iterations are terminated when $|L_k^*(q_k) - L_{k+1}^*(q_{k+1})| < \delta$ for all values in the configuration space. This assumes that the robot is capable of selecting actions that halt it in the goal region. The resulting stabilized cost-to-go function can be considered as a representation of the optimal strategy. Note that no choice of K is necessary because termination occurs when the loss values have stabilized. Also, only the representation of L_{k+1}^* is retained while constructing L_k^* ; earlier representations can be discarded to save storage space.

The general advantages of these kinds of computations were noted long ago in [431]: 1) extremely general types of system equations, performance criteria, and constraints can be handled; 2) particular questions of existence and uniqueness are avoided; 3) a true feedback solution is directly generated.

Using the cost-to-go as a navigation function To execute the optimal strategy, an appropriate action must be chosen using the cost-to-go representation from any given configuration (see Figure 10.6.b). One approach would be to simply store the action that produced the optimal cost-to-go value, for each quantized configuration. The appropriate action could then be selected by recalling the stored action at the nearest quantized configuration. This method could cause errors, particularly since it does not utilize any benefits of interpolation. A preferred alternative is to select actions by locally evaluating (10.15) at the exact current configuration. Linear interpolation can be used as before. Note that although the approach to select the action is local (and efficient), the global information is still taken into account (it is encoded in the cost-to-go function). This concept is similar to the use of a numerical navigation function in previous motion planning literature [51, 659] (such as NF1 or NF2), and the cost-to-go is a form of *progress measure*, as considered in [230]. When considering the cost-to-go as a navigation function, it is important to note that it does not contain local minima because it is constructed as a by-product of determining the optimal solution. Once the optimal action is determined, an exact next configuration is obtained (i.e., not a quantized configuration). This form of iteration continues until the goal is reached or a termination condition is met. During the time between stages, the trajectory can be linearly interpolated between the endpoints given by the discrete-time transition equation, or can be integrated using an original continuous-time transition

equation.

Computational expense Consider the computation time for the dynamic programming algorithm for the basic case modeled by (10.15). Let c denote the number of quantized values per axis of the configuration space. Let m denote the dimension of the configuration space. Let a denote the number of quantized actions. Each stage of the cost-to-go computations takes time $O(c^m a)$, and the number of stages before stabilization is nearly equal to the longest optimal trajectory (in terms of the number of stages) that reaches the goal. The space complexity is obviously $O(c^m)$. The algorithm is efficient for fixed dimension, yet suffers from the exponential dependence on dimension that appears in most deterministic motion planning algorithms. The utilization of the cost-to-go function during execution requires $O(a)$ time in each stage. These time complexities assume constant evaluation time of the cost-to-go at the next stage; however, if multilinear interpolation is used, then additional exponential-time computation is added because 2^m neighbors are evaluated.

10.5 Reinforcement Learning

We can now extend the infinite-horizon MDP problem by assuming that $P(x'|x, u)$ in Equations (10.3) and (10.4) is unknown. This is essentially saying that we have no idea what the distributions of nature are. Traditionally, this hurdle is handled by the following steps.

1. Learning phase (Travel through the states in X , try various actions, and gather statistics.)
2. Planning phase (Use value iteration or policy iteration to compute J^* and γ^* .)
3. Execution phase.

In the learning phase, if the number of trials is sufficiently large, $P(x'|x, u)$ can be estimated relatively well. Also during the learning phase, we can observe the losses associated with states and actions. If we combine the three steps above and run the world through a Monte Carlo simulator, we get *reinforcement learning*. Figure 10.7 shows an outline of the architecture.

A major issue of reinforcement learning is the problem of *exploration vs. exploitation*. The goal of exploration is to try to gather more information about $P(x'|x, u)$, but it might end up choosing actions that yield high losses. The goal of exploitation is to make good decisions based on knowledge of $P(x'|x, u)$, but it might fail to learn a better solution. Pure exploitation is vulnerable to getting stuck to a bad solution while pure exploration requires lots of resources and might never be used.

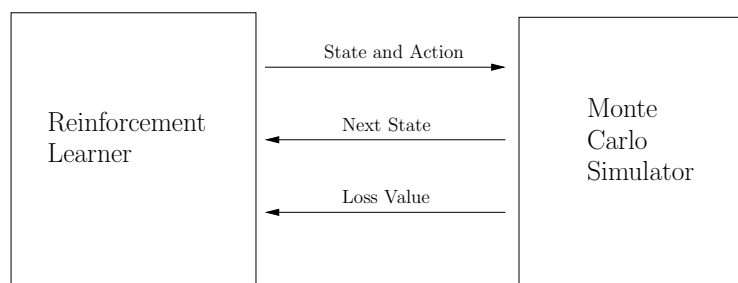


Figure 10.7: Reinforcement Learning Architecture

10.5.1 Stochastic Iterative Algorithms

Recall that the original evaluation of a particular strategy was:

$$J_\gamma(x) = l(x, u) + \alpha \sum_{x'} P(x'|x, u) J_\gamma(x')$$

But the problem now is that $P(x'|x, u)$ is unknown. Instead, we use what is called a *stochastic iterative algorithm*. $J_\gamma(x)$ will be updated with the following equation. ρ is the learning rate.

$$\hat{J}_\gamma(x) = (1 - \rho)\hat{J}_\gamma(x) + \rho(l(x, \gamma(x)) + \alpha\hat{J}_\gamma(x'))$$

In this equation, x' is now observed instead of calculated from $f(x, u, \theta)$. A question a keen reader might ask is where have the probabilities gone? They're conspicuously missing in the above equation. The answer is that they're really embedded in the observations of x' from nature. In the Monte Carlo simulation, states that have high probability will occur more often and thus will make a bigger influence to \hat{J}_γ . In this manner, over time the probabilities distribution of x' will be stored in \hat{J}_γ .

10.5.2 Finding an Optimal Strategy: Q-learning

So how do we find the optimal strategy? The answer lies in Q : rather than using just $J^* : X \rightarrow \mathbb{R}$, the expected loss of a particular strategy, now we use $Q^* : X \times U \rightarrow \mathbb{R}$. $Q^*(x, u)$ represents the optimal cost-to-go from applying u and then continuing on the optimal path after that. Note that Q is independent of the policy being followed.

Using $Q^*(x, u)$ in the dynamic programming equation yields:

$$Q^*(x, u) = l(x, u) + \alpha \sum_{x'} P(x'|x, u) \min_{u' \in U(x')} (Q^*(x', u'))$$

If we make $J^*(x)$ the expected cost for optimal strategy given state x , and $Q^*(x, u)$ be the expected cost for optimal strategy given state x and using cost u ,

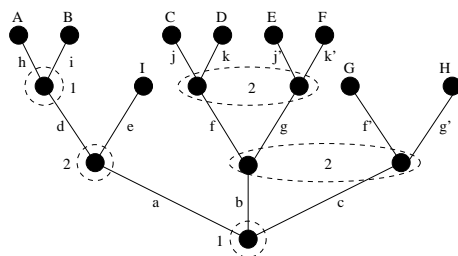


Figure 10.8: A tree for the extensive form.

then

$$J^*(x) = \min_{u \in U(x)} Q^*(x, u)$$

However, for reinforcement learning, the probability $P(x'|x, u)$ is unknown, so we can bring in the stochastic iterative idea again and get

$$\hat{Q}^*(x, u) := (1 - \rho)\hat{Q}^*(x, u) + \rho(l(x, u) + \alpha \min_{u' \in U(x')} \hat{Q}^*(x, u))$$

10.6 Sequential Game Theory

Until now we have used matrices to describe the games. This representation is called *normal form*. For sequential games (i.e., parlor games), in which a player take a decision based on the outcome of previous decisions of all the players, we can use the *extensive form* to describe the game.

The rules of a sequential game specify a series of well defined *moves*, where each move is a point of decision for a given player from among a set of alternatives. The particular alternative chosen by a player in a given decision point is called *choice*, and the totality of choices available to him at the decision point is defined as the *move*. A sequence of choices, one following another until the game is terminated is called a *play*. The extensive form description of a sequential game consist of the following:

- A finite tree that describes the relation of each move to all other moves. The root of the tree is the first move of the game.
- A partition of the nodes of the tree that indicates which of the players takes each move.
- A refinement of the previous partition into information sets. Nodes that belong to the same information set are indistinguishable to the player.
- A set of outcomes to each of the plays in the game.

Figure 10.8 shows an example of a tree for a sequential game. The numbers beside the nodes indicates which player takes the corresponding move. The edges are labeled by the corresponding choice selected. The leaves indicate the outcome of the play selected (a root-leaf path in the tree). The information sets are shown with dashed ellipses around the nodes. Nodes inside the same ellipse are indistinguishable for the players, but the players can differentiate nodes from one information set to another. If every ellipse enclose only one node, then we say that the players have *perfect information* of the game, which leads to a “feedback strategy”.

In the extensive form all games are described with a tree. For games like chess this may not seem reasonable, since the same arrangement of pieces on the board can be generated by several different routes. However, for the extensive form, two moves are different if they have different past histories, even if they have exactly the same possible future moves and outcomes.

10.6.1 Dynamic Programming over Sequential Games

10.6.2 Algorithms for Special Games

Chapter 11

The Information Space

Chapter Status



What does this mean? Check

<http://msl.cs.uiuc.edu/planning/status.html>
for information on the latest version.

Up to now it has been assumed everywhere that the current state is known. Suppose instead that the state is not exactly known. In this case, information regarding the state is obtained from sensors during the execution of a plan. This situation arises in most applications that involve interaction with the physical world. For example in robotics, it is virtually impossible for a robot to precisely know its state, except in some limited cases. What should be done if there is limited information regarding the state? A classical approach is to take all of the information available and try to estimate the state. If the estimates are sufficiently reliable, then we may safely pretend that there is no uncertainty in state information. This enables many of the planning methods introduced so far to be applied with only minor adaptation.

The more interesting case occurs when state estimation is altogether avoided. It may be surprising, but many important tasks can be defined and solved without ever requiring that specific states are reached, even though a state space is defined for the planning problem. To achieve this, the planning problem will be expressed in terms of an *information space*. The information space serves the same purpose for sensing problems as the configuration space of Chapter 4 did for problems that involve geometric transformations. The information space represents the place where problems that involve sensing uncertainty naturally live. Successfully formulating and solving such problems will depend on our ability to manipulate, simplify, and control the information space. In some cases, elegant solutions exist, and in others there appears to be no hope at present of efficiently solving them. There are many exciting open research problems associated with

information spaces and sensing uncertainty in general.

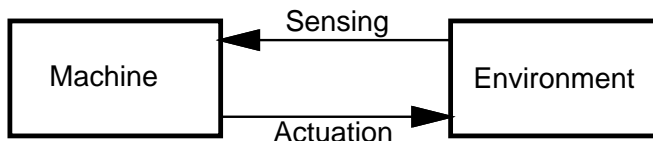


Figure 11.1: The state of the environment is not known. The only information available to make inferences regarding the state is the history of sensor observations, actions that have been applied, and the initial conditions. This history becomes the *information state*.

Recall the situation depicted in Figure 11.1, which was also shown in Section 1.4. It is assumed that the state of the environment is not known. There are three general sources of information regarding the state:

1. The *initial conditions* can provide powerful information regarding the state before any actions are applied. It might even be the case that the initial state is given. At the other extreme, the initial conditions might contain no information.
2. The *sensor observations* provide measurements of the state during execution. These measurements are usually incomplete or involve disturbances that distort their values.
3. The *actions* already executed in the plan provide valuable information regarding the state. For example, if a robot is commanded to move east (with no other uncertainties except an unknown state), then it is expected that the state is further east than it was previously. Thus, previously applied actions provide important clues for deducing the state.

Section 11.1 will formalize these concepts for the case of discrete state spaces. OTHER SECTIONS.

There are generally two ways to use the information space:

1. Take all of the information available, and try to estimate the state.
This is the classical approach. Pretend that there is no longer any uncertainty in state, but hope (or prove) that the resulting motion strategy or control law works under reasonable estimation error.
A plan is generally expressed as $\pi : X \rightarrow U$.
2. Solve the task entirely in terms of an information space.
Many robot tasks may be achieved without ever knowing the exact state. The goals and analysis are formulated in the information space, without the need to achieve particular states.
A plan is generally expressed as $\pi : \mathcal{I} \rightarrow U$, for an information space, \mathcal{I} .

The first approach may be considered somewhat classical. Most of the focus of the chapter is on the second approach, which represents a powerful way to express and solve problems.

GUIDE TO THE SECTIONS: Section 11.1 will formalize these concepts for the case of discrete state spaces.

11.1 Discrete State Spaces

11.1.1 Sensors

As the name suggests, *sensors* are designed to sense the state. Throughout all of Section 11.1 it is assumed that the state space, X is finite or countably infinite, as in Formulations 2.2.1 and 2.4.2. A *sensor* is defined in terms of two components: 1) the *observation space*, which is the set of possible readings for the sensor, and 2) *sensor mapping*, which characterizes the readings that can be expected if the state is given. In the planning model, the state will not be given, it is only assumed to be given when modeling a sensor.

Let Y denote an *observation space*, which is a finite or countably infinite set. Let h denote the *sensor mapping*. Three different kinds of sensor mappings will be considered, each of which is more complicated and general than the previous one:

1. **State sensor mapping:** In this case, $h : X \rightarrow Y$, which means that given the state, the observation is completely determined.
2. **State-nature sensor mapping:** In this case, a finite set, $\Psi(x)$, of *nature sensing actions* are defined for each $x \in X$. Each nature sensing action, $\psi \in \Psi(x)$ interferes with the sensor observation. Therefore, the state-nature mapping, h , produces an observation, $y = h(x, \psi) \in Y$ for every $x \in X$ and $\psi \in \Psi(x)$. The particular ψ chosen by nature is assumed to be unknown during planning and execution. However, it is specified as part of the sensing model.
3. **History-based sensor mapping:** In this case, the observation could be based on the current state or any previous states. Furthermore, a nature sensing action could be applied. Suppose that the current stage is k . The set of nature sensing actions is denoted by $\Psi_k(x)$, and the particular nature sensing action is $\psi_k \in \Psi_k(x)$. This yields a very general sensor mapping, defined as

$$y_k = h(x_1, \dots, x_k, \psi_k), \quad (11.1)$$

in which y_k is the observation obtained in stage k .

Many examples of sensors will now be given. These are provided to illustrate the definitions and to provide building blocks that will be used in later examples of information spaces. Examples 11.1.1 to 11.1.5 all involve state sensor mappings.

Example 11.1.1 (Odd/even sensor) Let $X = \mathbb{Z}$, the set of integers. Let $Y = \{0, 1\}$. A sensor mapping can be defined as

$$y = h(x) = \begin{cases} 0 & \text{if } x \text{ is even.} \\ 1 & \text{if } x \text{ is odd.} \end{cases} \quad (11.2)$$

The limitation of this sensor is that it only tells whether $x \in X$ is odd or even. When combined with other information, this might be enough to infer the state, but in general it provides incomplete information. ■

Example 11.1.2 (Mod sensor) Example 11.1.1 can be easily generalized yield the remainder when x is divided by k , for some fixed integer k . Let $X = \mathbb{Z}$, and let $Y = \{0, 1, \dots, k - 1\}$. The sensor mapping is defined as

$$y = h(x) = x \pmod{k}. \quad (11.3)$$

■

Example 11.1.3 (Sign sensor) Let $X = \mathbb{Z}$, and let $Y = \{-1, 0, 1\}$. The sensor mapping is defined as

$$y = h(x) = \text{sgn}x. \quad (11.4)$$

This sensor provides very limited information because it only indicates on which side of the boundary $x = 0$ the state may lie. The one exception is that it can precisely determine whether $x = 0$ or $x \neq 0$. ■

Example 11.1.4 (Selective sensor) Let $X = \mathbb{Z} \times \mathbb{Z}$, and let $(i, j) \in X$ denote a state, in which $i, j \in \mathbb{Z}$. Suppose that only one component of (i, j) can be observed. This yields the sensor mapping

$$y = h(i, j) = i. \quad (11.5)$$

An obvious generalization can be made for any state space that is formed from Cartesian products. The sensor reveals the values of one or more components, and other rest remain hidden. ■

Example 11.1.5 (Bijective sensor) Let X be any state space, and let $Y = X$. Let the sensor mapping be defined as any bijective function $h : X \rightarrow Y$. This sensor provides information that is equivalent to having knowledge of the state. Because h is bijective, it can be inverted to obtained $h^{-1} : Y \rightarrow X$. For any $y \in Y$, the state can be determined as $x = h^{-1}(y)$.

A special case of the *bijective sensor* is the *identity sensor*, for which h is the identity function. This was essentially assumed to exist for all planning problems

covered before this chapter because it immediately yields the state. However, any bijective sensor would serve the same purpose. ■

Example 11.1.6 (Null sensor) Let X be any state space, and let $Y = \{0\}$. The *null sensor* is obtained by letting defining the sensor mapping as any function $h : X \rightarrow Y$. The sensor reading remains fixed, and hence contains no information regarding the state. ■

From the examples so far, it is tempting to think about partitioning X based on sensor observations. Suppose that in general a state mapping, h , is not bijective, and let $H(y)$ denote the following subset of X :

$$H(y) = \{x \in X \mid y = h(x)\}, \quad (11.6)$$

called the *preimage* of y . The set of preimages, one for each $y \in Y$, form a partition of X . In some sense, this indicates the “resolution” of the sensor. A bijective sensor partitions X into singleton sets because it contains perfect information. At the other extreme, the null sensor partitions X into a single set, X itself. The sign sensor appears slightly more useful because it partitions X into three sets: $H(1) = \{1, 2, \dots\}$, $H(-1) = \{\dots, -2, -1\}$, and $H(0) = \{0\}$. The preimages of the selective sensor are particularly interesting. For each $i \in \mathbb{Z}$, $H(i) = \mathbb{Z}$. EXPLAIN CONNECTION TO QUOTIENT GROUPS FOR MOD SENSOR.

Next consider some examples that involve a state-action sensor mapping. There are two different possibilities regarding the model for the nature sensing action:

1. **Nondeterministic:** In this case, there is no additional information regarding which $\psi \in \Psi(x)$ will be chosen.
2. **Probabilistic:** A probability distribution is known. In this case, the probability, $P(\psi|x)$, that ψ will be chosen is known for each $\psi \in \Psi(x)$.

These two possibilities also appeared in Section ?? for nature actions that interfere with the state transition equation.

It is sometimes useful to consider the state-action sensor model as a probability distribution over Y for a given state. Suppose that when the domain of h is restricted to some $x \in X$, then it forms an injective mapping from Ψ to X . In other words, every nature action leads to a unique observation, assuming x is fixed. Using $P(\psi)$ and h , one can easily derive $P(y|x)$ as

$$P(y|x) = \begin{cases} P(\psi) & \text{for a unique } \psi \text{ such that } y = h(x, \psi). \\ 0 & \text{if no such } \psi \text{ exists.} \end{cases} \quad (11.7)$$

If the injective assumption is lifted, then $P(\psi)$ is replaced by a sum over all ψ for which $y = h(x, \psi)$.

Example 11.1.7 (Sensor disturbance) Let $X = \mathbb{Z}$, $Y = \mathbb{Z}$, and $\Psi = \{-1, 0, 1\}$. The idea is to construct a sensor that would be the identity sensor if it were not for the interference of nature. The sensor mapping is

$$y = h(x, \psi) = x + \psi. \quad (11.8)$$

It is always known that $|x - y| \leq 1$. Therefore, if y is received as a sensor reading, one of the following must be true: $x = y - 1$, $x = y$, or $x = y + 1$. ■

Example 11.1.8 (Disturbed sign sensor) Let $X = \mathbb{Z}$, $Y = \{-1, 0, 1\}$, and let $\Psi = \{-1, 0, 1\}$. Let the sensor mapping be defined as

$$y = h(x, \psi) = \text{sgn}(x + \psi). \quad (11.9)$$

In this case, if $y = 0$, it is no longer known for certain whether $x = 0$. It is possible that $x = -1$ or $x = 1$. If $x = 0$, then it is possible for the sensor to read -1 , 0 , or 1 . ■

Example 11.1.9 (Disturbed odd/even sensor) It is not hard to construct examples for which some mild interference from nature destroys all of the information. Let $X = \mathbb{Z}$, $Y = \{0, 1\}$, and $\Psi = \{0, 1\}$. Let the sensor mapping be defined as

$$y = h(x, \psi) = \begin{cases} 0 & \text{if } x + \psi \text{ is even.} \\ 1 & \text{if } x + \psi \text{ is odd.} \end{cases} \quad (11.10)$$

If the value of ψ is not known, then the sensor provides no useful information regarding the state. For example, it may yield $y = 0$, but it not known whether x is even or odd. If there is a probabilistic model for the nature sensing action, then this sensor may provide some useful information. ■

It is once again informative to consider preimages. For a state-action sensor mapping, the preimage is defined as

$$H(y) = \{x \in X \mid \exists \psi \in \Psi(x) \text{ for which } y = h(x, \psi)\}. \quad (11.11)$$

In comparison to state sensor mappings, the preimage sets are larger for state-action sensor mappings. They also do not generally form a partition of X . For example, the preimages of the Example 11.1.8 are: $H(1) = \{0, 1, \dots\}$, $H(0) = \{-1, 0, 1\}$, and $H(-1) = \{\dots, -2, -1, 0\}$. This is not a partition because every preimage contains 0.

Finally, one example of a history-based sensor mapping is given.

Example 11.1.10 (Delayed-observation sensor) Let $X = Y = \mathbb{Z}$. A *delayed-observation sensor* can be defined for some fixed positive integer i as $y_k = x_{k-i}$.

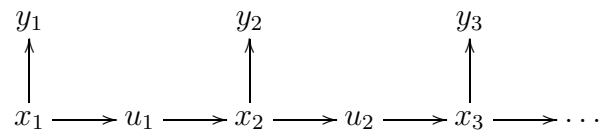


Figure 11.2: In each stage, k , an observation, $y_k \in Y$ is received, and an action $u_k \in U$ is applied. The state, x_k , however, is hidden from the decision maker.

Thus, it indicates what the state was i stages ago. In this case, it gives a perfect measurement of the old state value. Many other variants are possible. For example, it might only give the sign of the state, i stages ago. ■

11.1.2 Defining the Information Space

Suppose that X , U , and f have been defined as in Formulation ??, and the notion of stages has been defined as in Formulation 1. This yields state sequences x_1, x_2, \dots and action sequences u_1, u_2, \dots during the execution of a plan. However, in the current setting, the state sequence is not known. Instead, at every stage, an observation, y_k , is obtained. The process depicted in Figure 11.2.

In previous formulations, the action space, $U(x)$, was generally allowed to depend on x . Since x is currently unknown, it would seem strange to all the actions to depend on x . This would mean that inferences could be made regarding the state simply by noticing which actions are available. Instead, it will be assumed that U is fixed for all $x \in X$.

Initial conditions As stated at the beginning of the chapter, the initial conditions provide one of the three general sources of information regarding the state. Three alternative types of initial conditions will be allowed:

1. The initial state, $x_1 \in X$ is given. This initializes the problem with perfect state information. Assuming nature actions interfere with the state transition equation f , uncertainty in the current state will generally develop.
2. A set of states, $X_1 \subset X$ is given. In this case, the initial state is only known to lie within a particular subset of X . This can be considered as a generalization of the first type, which only allowed singleton subsets.
3. A probability distribution, $P(x)$, over X is given.

In general, let η_0 denote the initial condition, which may be any one of the three alternative types.

History Suppose that the k^{th} stage has passed. What information is available? It will be assumed that at every stage, a sensor observation is made. This yields a *sensing history*, (y_1, y_2, \dots, y_k) . At every stage an action also be applied. This yields an *action history*, $(u_1, u_2, \dots, u_{k-1})$. Note that the sequence only runs to u_{k-1} , instead of u_k , because once u_k is applied, state x_{k+1} and stage $k + 1$ is obtained.

By combining the sensing and action histories, the *history*, λ_k , at stage k is the sequence

$$\lambda_k = (u_1, \dots, u_{k-1}, y_1, \dots, y_k). \quad (11.12)$$

Information state The history, λ_k , in combination with the initial condition, \mathcal{I}_0 , yields the *information state*, which is denoted by η_k . This corresponds to all information that is known up to stage k . In spite of the fact that the states, x_1, \dots, x_k , might not be known, the information states are always known because they are defined directly in terms of available information. The information state may be denoted as

$$\eta_k = (\eta_0, u_1, \dots, u_{k-1}, y_1, \dots, y_k), \quad (11.13)$$

or in short form, $\eta_k = (\eta_0, \lambda_k)$. When representing information spaces, we will generally ignore the problem of nesting parentheses; the short form actually expresses a sequence of two sequences, while (11.13) is a single sequence. This distinction is insignificant for the purposes of decision making.

The information state, η_k , can also be expressed as

$$\eta_k = (\eta_{k-1}, u_{k-1}, y_k), \quad (11.14)$$

by noticing that the information state at stage k contains all of the information from the information state at stage $k - 1$. The only new information is the previously applied action, u_{k-1} and the current sensor observation, y_k .

Information space The information space will simply be the set of all possible information states. Although the information states appear to be quite complicated, it is helpful to think of them abstractly as points in a set that is called the information space. To define the set of all possible information states, we will need careful definitions of the set of all initial conditions, actions, and observations.

The set of all observations is always Y . Therefore, the set of all observation histories is Y^k , which is obtained by a Cartesian product of k copies of the observation space, Y :

$$Y^k = Y \times Y \dots \times Y. \quad (11.15)$$

Similarly, the set of all action histories is given by U^{k-1} , the Cartesian product of $k - 1$ copies of the action space, U .

It is slightly more complicated to define the set of all possible initial conditions because three different types of initial conditions were possible. Let \mathcal{I}_0 denote the *initial condition space*. Depending on which of the three types of initial conditions are used, one of the following three definitions of \mathcal{I}_0 is used:

1. If the initial state, x_1 , is given, then $\mathcal{I}_0 \subseteq X$. Typically, $\mathcal{I}_0 = X$; however, it might be known in some instances that certain initial states are impossible. Therefore, it is generally written that $\mathcal{I}_0 \subseteq X$.
2. If X_1 is given, then $\mathcal{I}_0 \subseteq \text{pow}(X)$, in which pow denotes the power set. Again, a typical situation is $\mathcal{I}_0 \subseteq \text{pow}(x)$; however, it might be known that certain subsets of X are impossible as initial conditions.
3. Finally, if $P(x)$ is given, then $\mathcal{I}_0 \subseteq \mathcal{P}(X)$ in which $\mathcal{P}(x)$ is the set of all probability distributions over X .

The *information space at stage k* can be expressed as

$$\mathcal{I}_k = \mathcal{I}_0 \times U^{k-1} \times Y^k. \quad (11.16)$$

Thus, each $\eta_k \in \mathcal{I}_k$ yields an initial condition, action history, and observation history.

It will be convenient to consider information spaces that do not depend on k . This will be defined by simply taking a union. If there are K stages, then the *information space, \mathcal{I}* , is

$$\mathcal{I} = \mathcal{I}_1 \cup \mathcal{I}_2 \cup \cdots \cup \mathcal{I}_K. \quad (11.17)$$

If the number of stages is not fixed, then \mathcal{I} is defined to be the union of \mathcal{I}_k over all $k \in \mathbb{N}$. The situation is similar the state space obtained for time-varying motion planning in Section 7.1. The information space is naturally time-dependent because information accumulated over time. In our discrete model, the reference to time is only implicit through the use of stages. Therefore, stage-dependent information spaces were defined. Taking the union of all of these is similar to what the state space was formed in Section 7.1 by making time one axis of the state space. For the information space, \mathcal{I} , the stage index, k , can be imagined as an “axis”.

One immediate concern regarding the information space, \mathcal{I} , is that the information states may be arbitrarily long because the history grows linearly with the number of stages. For now, it is helpful to simply imagine \mathcal{I} abstractly as another kind of state space, without paying close attention to how complicated each $\eta \in \mathcal{I}$ may be to represent. In many contexts, there exist ways to simplify the information state representation. This will be the topic of Section 11.2.

11.1.3 Defining a Planning Problem

Now that the information space has been defined, in many ways it can be considered as another kind of state space; however, it is important to keep in mind that the information space was derived from another state space for which perfect state observations could not be obtained. This next task is to define planning problems on the information space.

In Section ??, a feedback plan was defined as a function of the state. Here a feedback plan is instead a function of the information state. Decisions cannot be

based on the state because it will be generally unknown during execution of the plan. However, the information state is always known. Therefore, it is logical to base decisions on the information state.

Let π_K denote a K -step information-feedback plan, which is a sequence $(\pi_1, \pi_2, \dots, \pi_K)$ of K functions, $\pi_k : \mathcal{I}_k \rightarrow U$. Thus, at every stage, k , the information state $\eta_k \in \mathcal{I}_k$ is used as a basis for choosing the action $u_k = \pi_k(\eta_k)$. Due to interference of nature through both the state transition equation and the sensor mapping, the action sequence, (u_1, \dots, u_K) produced by a plan, π_K , will not be known until the plan terminates.

Just as in Formulation 2.4.2, it will be convenient to assume that U contains a *termination action*, u_T . If u_T is applied to η_k , at stage k , then u_T is repeatedly applied forever. It is assumed once again that the state, x_k , remains fixed after the termination condition is applied. Remember, however, the x_k is still unknown in general; it becomes fixed and unknown. Technically, based on the definition of information spaces, the information state must change after u_T is applied because the history grows. These information states can be ignored, however, because no new decisions are made after u_T is applied. A plan that uses a termination condition can be specified as $\pi = (\pi_1, \pi_2, \dots)$, because the number of stages may vary each time the plan is executed.

We are almost ready to define the planning problem. This will require the specification of a cost functional. The cost will depend on the history, σ , of states and actions, as in Section ???. The planning formulation involves the following components, summarizing most of the concepts introduced so far in Section 11.1:

Formulation 11.1.1 (Discrete Information Space Planning)

1. A nonempty *state space*, X , which is either finite or countably infinite.
2. A finite *action space*, U . It is assumed that U contains a special *termination action*, which has the same effect as defined in Formulation 2.4.2.
3. A finite *nature action space*, $\Theta(x, u)$ for each $x \in X$ and $u \in U$.
4. A *state transition equation*, f , that produces a state, $f(x, u, \theta)$ for every $x \in X$, $u \in U$, and $\theta \in \Theta(x, u)$.
5. A finite or countably infinite *observation space*, Y .
6. A finite *nature observation action space*, $\Psi(x)$ for each $x \in X$.
7. An *sensor mapping*, h , which produces an observation, $y = h(x, \psi)$ for each $x \in X$ and $\psi \in \Psi$. This definition assumed a state-nature sensor mappings. A state sensor mapping or history-based sensor mapping, as defined in Section 11.1.1 may alternatively be used.
8. A set of *stages*, each denoted by k , which begins at $k = 1$ and continues indefinitely.

9. An *initial condition*, η_0 , which is an element of an *initial condition space*, \mathcal{I}_0 .
10. A *goal set*, $X_G \subset X$.
11. An *information space*, \mathcal{I} , which is the union of the information spaces, $\mathcal{I}_k = \mathcal{I}_0 \times U^{k-1} \times Y^k$, for each stage k .
12. Let L denote a real-valued, additive cost functional, which may be applied to any state-action history, $\sigma_K = (x_1, \dots, x_{K+1}, u_1, \dots, u_K)$, to yield

$$L(\sigma_K) = \sum_{k=1}^K l(x_k, u_k) + l_F(x_{K+1}). \quad (11.18)$$

If the termination action, u_T , is applied at some stage k , then for all $i \geq k$, $u_i = u_T$, $x_i = x_k$, and $l(x_i, u_T) = 0$.

Using Formulation 11.1.1, either a feasible or optimal planning problem can be defined. To obtain a feasible planning problem, let $l(x_k, u_k) = 0$ for all $x_k \in X$ and $u_k \in U$, and let

$$l_F(x_{K+1}) = \begin{cases} 0 & \text{if } x_{K+1} \in X_G \\ \infty & \text{otherwise} \end{cases}. \quad (11.19)$$

To obtain an optimal planning problem, then in general $l(x_k, u_k)$ may assume any nonnegative, finite value.

The Information Space is Just Another State Space It will become important throughout this chapter and Chapter 12 to realize that in many ways the information space can be treated as an ordinary state space. It only seems special because it is itself derived from another state space, but once this is forgotten, it exhibits many properties of an ordinary state space in planning. One nice feature is that the state in this new space is always known. Thus, by converting from an original state space to its information space, we also convert from having imperfect state information to always knowing the (information) state.

One important consequence of this interpretation is that the state transition equation can be lifted into the information space to obtain an *information transition equation*, $f_{\mathcal{I}}$. Suppose there are no nature actions. In this case, future states are predictable, which leads to

$$\eta_{k+1} = f_{\mathcal{I}}(\eta_k, u_k). \quad (11.20)$$

The function $f_{\mathcal{I}}$ generates η_{k+1} by concatenating u_k and $y_{k+1} = h(x_{k+1}) = h(f(x_k, u_k))$ to η_k . If there are nature actions, θ_k and/or nature sensing actions ψ_{k+1} , then

$$\eta_{k+1} = f_{\mathcal{I}}(\eta_k, u_k, \theta_k, \psi_{k+1}), \quad (11.21)$$

which reflects the fact that future information states are unpredictable. Once θ_k and ψ_{k+1} are chosen by nature, then η_{k+1} is obtained by concatenating u_k and

$$y_{k+1} = h(x_{k+1}, \psi_{k+1}) = h(f(u_k, x_k, \theta_k), \psi_{k+1}) \quad (11.22)$$

to the history. Note, however, that even though nature causes future information states to be unpredictable, the current information state is always known. A plan, $\mathcal{I} \rightarrow U$ now seems like a state-feedback plan, if the information space is viewed as a state space. The transitions are all specified by $f_{\mathcal{I}}$.

11.2 Alternative Representations of Information Spaces

The information space in its original form appears to be quite complicated. Every information state corresponds to a history of actions and observations. The length of the information state vector unfortunately grows linearly with the number of stages. This motivates many methods that try to reduce or simplify the representation of the information space in some way. In many applications, the ability to perform this simplification is critical to finding a practical solution. In some cases, the simplification preserves the structure of the original information space, meaning that completeness, and optimality if applicable, will not be lost by using the simpler representation. In other cases, we might be willing to tolerate a simplification that results in an approximation of the information space. Such an approach may be the only way to handle the most challenging problems.

This section involves a substantial amount of notation. It is easy to become lost without frequent consideration of examples. Section ?? will present several detailed examples that illustrate the concepts presented in Sections 11.1 and 11.2. In this section, Example 11.2.1, which is very simple and less interesting, will be used to provide immediate illustration of some notation and concepts.

11.2.1 Nondeterministic Derived Information States

This section assumes that nature is modeled nondeterministically, which means that there is no information about what actions nature will choose, other than the actions will be chosen from Θ and Ψ . Further assume that the state-action sensor mapping from Section 11.1.1 is used. Consider what inferences that may be drawn from an information state, $\eta_k = (\eta_0, \lambda_k)$. Since the model does not involve probabilities, suppose that η_0 represents a set $X_1 \subseteq X$. Using the history, λ_k , together with several components from Formulation 11.1.1, we can calculate a minimal subset of X in which x_k is known to lie. Let $X_k(\eta_k)$ refer to this subset, which will be referred to as a *derived information state*. It is always true that $x_k \in X$. Thus, it is important to make X_k as small as possible by removing any states that are impossible values for x_k .

Recall from (11.11), that for every observation, y_k , a set $H(y_k) \subseteq X$, of possible values for x_k , can be inferred. This could serve as a crude estimate of the derived information state. It is certainly known that $X_k(\eta_k) \subseteq H(y_k)$; otherwise, the current state, x_k , would not be consistent with the current sensor observation. If we carefully progress from the initial conditions, while applying constraints due to the state transition equation, the appropriate subset of $H(y_k)$ will be obtained.

From the state transition equation, f , define a set-valued function, F , which yields a subset of X for every $x \in X$ and $u \in U$ as

$$F(x, u) = \{x' \in X \mid \exists \theta \in \Theta(x) \text{ for which } x' = f(x, u, \theta)\}. \quad (11.23)$$

Note that both F and H are set-valued functions that eliminate the direct appearance of nature actions. The effect of nature is taken into account in the set that is obtained when these functions are applied. This will be very convenient for computing the derived information state.

It will be convenient to generally use the notation $X(\cdot)$ as a subset of X that is derived using whatever information appears in the place of \cdot . It may sometimes be denoted as $X_k(\cdot)$ to additionally denote the particular stage, k .

An inductive process will now be described that results in computing the derived information state, $X_k(\eta_k)$, for any stage k . The base case, $k = 1$, of the induction proceeds as

$$X_1(\eta_1) = X_1(\eta_0, y_1) = X_1 \cap H(y_1). \quad (11.24)$$

The first part of the equation replaces η_1 with (η_0, y_1) , which is the long form of the information state. There are not yet any actions in the history. The second part applies set intersection to make consistent the two pieces of information: 1) the initial state lies in X_1 , which is the initial condition, and 2) the states in $H(y_1)$ are possible given the observation y_1 .

Now assume inductively that the derived information state $X_k(\eta_k) \subseteq X$ has been computed, and the task is to compute the derived information state, $X_{k+1}(\eta_{k+1})$. Recall that $\eta_{k+1} = (\eta_k, u_k, y_{k+1})$. Thus, the only new pieces of information are that u_k was applied and y_{k+1} was observed. These will be considered one at a time.

Consider computing $X_{k+1}(\eta_k, u_k)$. If x_k was known, then after applying u_k , the state could lie anywhere within $F(x_k, u_k)$, using (11.23). Although x_k is actually not known, it is, however, known that $x_k \in X_k(\eta_k)$. Therefore,

$$X_{k+1}(\eta_k, u_k) = \bigcup_{x_k \in X_k(\eta_k)} F(x_k, u_k). \quad (11.25)$$

This can be considered as the set of all states that can be reached by starting from some state in $X_k(\eta_k)$, and applying actions $u_k \in U$ and $\theta_k \in \Theta(x_k)$.

The next step is to take into account the observation y_{k+1} . This information alone indicates that x_{k+1} lies in $H(y_{k+1})$. Therefore, an intersection is performed

to obtain the derived information state,

$$X_{k+1}(\eta_{k+1}) = X_{k+1}(\eta_k, u_k, y_{k+1}) = X_{k+1}(\eta_k, u_k) \cap H(y_{k+1}). \quad (11.26)$$

Now that it has been shown how to compute $X_{k+1}(\eta_{k+1})$ from $X_k(\eta_k)$. After starting with (11.24), the derived information states at any stage can be computed by iterating (11.25) and (11.26) as many times as necessary.

Because the derived information state is always a subset of X , a *derived information space*, denoted by \mathcal{I}° , can be defined as $\mathcal{I}^\circ = \text{pow}(X)$. If X is finite, then \mathcal{I}° is also finite, which was not the case with \mathcal{I} because the histories continued to grow with the number of stages. Thus, if the number of stages is unbounded or large in comparison to the size of X , then derived information states seem preferable. It is also convenient that in \mathcal{I}° there does not need to be an explicit reference to stages. It truly appears to be the appropriate “state space” for the problem. For the planning problem, the goal region, X_G , can be expressed directly as a derived information state. In this way, the planning task is to terminate in a derived information state X_K for which $X_K \subseteq X_G$. The history does not even have to be explicitly maintained. All computations can be performed directly in terms of derived information states.

The following example is not very interesting in itself, but it is simple enough to illustrate the concepts introduced so far.

Example 11.2.1 (Three-State Example) Consider the following components:

1. A state space, $X = \{0, 1, 2\}$.
2. An action space, $U = \{-1, 0, 1\}$.
3. A nature action space, $\Theta(x) = \{0, 1\}$ for all $x \in \Theta$.
4. A state transition equation $f(x, u, \theta) = (x + u + \theta) \pmod 3$.
5. An observation space, $Y = \{0, 1, 2, 3, 4\}$.
6. A nature observation action space, $\Psi(x) = \{0, 1, 2\}$ for all $x \in X$.
7. A sensor mapping, $y = h(x, \psi) = x + \psi$.

The original information state representation based on histories appears very cumbersome for this example, which only involves three states. The derived information space for this example is

$$\mathcal{I}^\circ = \{\emptyset, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{1, 2\}, \{0, 2\}, \{0, 1, 2\}\}, \quad (11.27)$$

which is the power set of $X = \{0, 1, 2\}$. Note, however, that the emptyset, \emptyset , can usually be deleted from \mathcal{I}° .¹ Suppose that the initial condition is $X_1 = \{0, 2\}$,

¹One notable exception is in the theory of nondeterministic finite automata, in which it is possible that all copies of the machine die, and there is no possible current state [711].

and that the initial state is $x_1 = 0$. The initial state is unknown to the decision maker, but it is needed to make an example because we need to make sure that valid observations will be made.

Now consider the execution over a number of stages. Suppose that the first observation, y_1 , is received as $y_1 = 2$. Based on the sensor mapping, $H(y_1) = H(3) = \{1, 2, 3\}$, which is not very helpful since $H(3) = X$. Applying (11.24) yields $X_1(\eta_1) = \{0, 2\}$. Now suppose that the decision maker applies the action $u_1 = 1$, and nature applies $\theta_1 = 1$. Using f , this yields $x_2 = 2$. The decision maker does not know θ_1 , and must therefore take into account any nature action that could have been applied. It uses (11.26) to infer that

$$X_2(\eta_1, u_1) = F(0, 0) \cup F(0, 1) = \{0, 1\} \cup \{1, 2\} = \{0, 1, 2\}. \quad (11.28)$$

Now suppose that $y_2 = 3$. From the sensor mapping, $H(3) = \{1, 2\}$. Applying (11.26) yields

$$X_2(\eta_2) = X_2(\eta_1, u_1) \cap H(y_2) = \{1, 2, 3\} \cap \{1, 2\} = \{1, 2\}. \quad (11.29)$$

This process may be repeated for as many stages as desired. It can be seen that a path generated through \mathcal{I}° be visiting a sequence of derived information states. Note that if the observation $y_k = 4$ is every received, the state, x_k , will become immediately known because $H(4) = \{2\}$. ■

Is the derived information space, \mathcal{I}° , equivalent in some way to the original information space, \mathcal{I} ? The derived information space appears to be simpler; therefore, it seems that some information was lost. The construction of \mathcal{I}° was obtained by mapping information states, η_k to derived information states, $X_k(\eta_k)$. It is certainly possible that many information states could map to the same derived information state. When using the derived information space, it is important to answer the following question:

For the purposes of decision making, it is sufficient to know the set of possible states, or is it important to additionally know what history led to this set of possible states?

The answer to this question is usually no. If it is known that x_k lies within a particular subset of X , given by the derived information state, there is nothing else to learn from the history of how the subset was derived. Note that it is generally impossible to recover the history from a derived information state.

11.2.2 Probabilistic Derived Information States

If nature is modeled probabilistically, it turns out that the derived information states can be determined once again. In this case, each derived information state is a probability distribution, as opposed to a set. The set union and intersection of (??) and (??) are replaced by in this section by marginalization and Bayes

rule, respectively. In a sense, these are the probabilistic equivalents of union and intersection. It will be very helpful to compare the expressions from this section to those of Section 11.2.1. Most expressions in this section of the form $P(x_k|\cdot)$, will have an equivalent expression in Section 11.2.1 of the form $X_k(\cdot)$.

The first step is to use make probabilistic versions of H and F . These are $P(x_k|y_k)$ and $P(x_{k+1}|x_k, u_k)$. The latter term was given in Section ???. To obtain $P(x_k|y_k)$, recall from Section ?? that $P(y_k|x_k)$ can be easily derived from $P(\psi_k|x_k)$. To obtain $P(x_k|y_k)$, Bayes rule can be applied. Recall from basic probability theory that

$$P(x_k, y_k) = P(x_k|y_k)P(y_k) = P(y_k|x_k)P(x_k). \quad (11.30)$$

Solving for $P(x_k|y_k)$ yields

$$P(x_k|y_k) = \frac{P(y_k|x_k)P(x_k)}{P(y_k)} = \frac{P(y_k|x_k)P(x_k)}{\sum_{x_k \in X} P(y_k|x_k)P(x_k)}. \quad (11.31)$$

In the last step, $P(y_k)$, was rewritten using marginalization. Note that in this case x_k appears as the sum index; therefore, the denominator is only a function of y_k , as required. Bayes rule requires knowing the prior, $P(x_k)$. In the coming derivation, this will be replaced by a derived information state.

Next consider defining derived information states for the probabilistic case. Each state is a probability distribution over X , and can be written as $P(x_k|\eta_k)$, if derived from η_k . The initial condition produces $P(x_1)$. Once again, derived information states can be computed inductively. For the base case, the only new piece of information is y_1 . Thus, the derived information state, $P(x_1|\eta_1)$, is $P(x_1|y_1)$. This is computed by letting $k = 1$ in (11.31) to yield

$$P(x_1|\eta_1) = P(x_1|y_1) = \frac{P(y_1|x_1)P(x_1)}{\sum_{x_1 \in X} P(y_1|x_1)P(x_1)}. \quad (11.32)$$

Now consider the inductive step by assuming that $P(x_k|\eta_k)$ is given. The task is to determine $P(x_{k+1}|\eta_{k+1})$, which is equivalent to $P(x_{k+1}|\eta_k, u_k, y_{k+1})$. Just as in Section 11.2.1, this will proceed in two parts by first considering the effect of u_k , followed by y_{k+1} . The first step is to determine $P(x_{k+1}|\eta_k, u_k)$ from $P(x_k|\eta_k)$. First, note that

$$P(x_{k+1}|\eta_k, x_k, u_k) = P(x_{k+1}|x_k, u_k) \quad (11.33)$$

because η_k does contain any additional information regarding the prediction of x_{k+1} since x_k is given. Marginalization from probability theory, can be used to eliminate x_k from $P(x_{k+1}|x_k, u_k)$. This must be eliminated because it is not given. Putting these steps together yields

$$P(x_{k+1}|\eta_k, u_k) = \sum_{x_k \in X} P(x_{k+1}|x_k, u_k, \eta_k)P(x_k|\eta_k) = \sum_{x_k \in X} P(x_{k+1}|x_k, u_k)P(x_k|\eta_k), \quad (11.34)$$

which expresses $P(x_{k+1}|\eta_k, u_k)$ in terms of given quantities.

The next step is to take into account the observation, y_{k+1} . This is accomplished by making a version of (11.31) that is conditioned on the information accumulated so far: η_k and u_k . Also, k is replaced with $k + 1$. The result is

$$P(x_{k+1}|y_{k+1}, \eta_k, u_k) = \frac{P(y_{k+1}|x_{k+1}, \eta_k, u_k)P(x_{k+1}|\eta_k, u_k)}{\sum_{x_{k+1} \in X} P(y_{k+1}|x_{k+1}, \eta_k, u_k)P(x_{k+1}|\eta_k, u_k)}. \quad (11.35)$$

The left side of (11.35) is equivalent to $P(x_{k+1}|\eta_{k+1})$, which is the derived information state for stage $k + 1$, as desired. There are two different kinds of terms on the right. The expression for $P(x_{k+1}|\eta_k, u_k)$ was given in (11.34). Therefore, the only remaining term to calculate is $P(y_{k+1}|x_{k+1}, \eta_k, u_k)$. Note that

$$P(y_{k+1}|x_{k+1}, \eta_k, u_k) = P(y_{k+1}|x_{k+1}) \quad (11.36)$$

because the sensing mapping depends only on the state (and the probability model for the nature observation action, which also depends only on the state). Since $P(y_{k+1}|x_{k+1})$ is specified as part of the sensor model, we are finished deriving the computation of $P(x_{k+1}|\eta_{k+1})$ from $P(x_k|\eta_k)$.

For the probabilistic case, the derived information space, \mathcal{I}° , is the set, $\mathcal{P}(X)$, of all probability distributions over X . Again, the planning problem can be expressed entirely in terms of the derived information space, instead of maintaining histories. A goal region can be specified as constraints on the probabilities. For example, for some particular $x \in X$, the goal might be to reach any derived information state for which $P(x|\eta_k) > 1/2$.

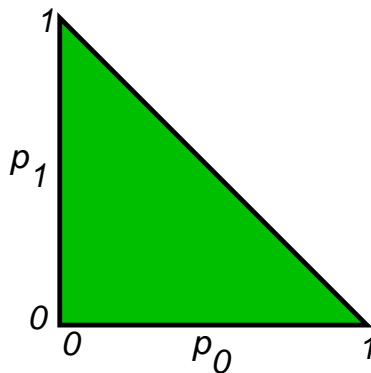


Figure 11.3: The probabilistic derived information space for the three-state example is a 2-simplex embedded in \mathbb{R}^3 .

Example 11.2.2 (Three-State Example Revisited) Now return to Example 11.2.1, but this time use probabilistic models. For a derived information state, let p_i denote the probability that the current state is $i \in X$. The derived information

state can be expressed as $(p_0, p_1, p_2) \in \mathbb{R}^3$. This implies that the information space can be nicely embedded in \mathbb{R}^3 . By the axioms of probability, $p_0 + p_1 + p_2 = 1$, which in \mathbb{R}^3 can be interpreted as a plane that slices diagonally through the origin. This restricts the \mathcal{I}° to a two-dimensional set. Also following the axioms of probability, for each $i \in \{0, 1, 2\}$, $0 \leq p_i \leq 1$. This means that \mathcal{I}° is restricted to a triangular region in \mathbb{R}^3 . The vertices of this triangular region are $(0, 0, 1)$, $(0, 1, 0)$, and $(1, 0, 0)$; these corresponds to the three different ways to have perfect state information. In a sense, the distance away from these points corresponds to the amount of uncertainty in the state. The uniform probability distribution $(1/3, 1/3, 1/3)$ is equidistant from the three vertices. A projection of the triangular region into \mathbb{R}^2 is shown in Figure 11.3. The interpretation in this case is that p_1 and p_2 give a point in \mathbb{R}^2 , and p_3 is automatically determined from $p_3 = 1 - p_1 - p_2$.

The triangular region in \mathbb{R}^3 corresponds to an uncountably infinite set, even though the original information space is countably infinite for a fixed initial condition. This may seem strange, but there is no problem because for a fixed initial condition, it is generally impossible to reach all of the points in $\mathcal{P}(X)$. If the initial condition allows any point in $\mathcal{P}(X)$, then all of the derived information space is covered.

NEED TO SHOW A COUPLE OF STEPS OF THE COMPUTATIONS. ■

11.2.3 Collapsing the Information Space

The mappings from \mathcal{I} to \mathcal{I}° , which were presented in Sections 11.2.1 and 11.2.2, are special cases of a very general and powerful principle called *collapsing*. For a given problem, there are numerous possible mappings that can be developed to further reduce the size of the information space. The general idea is to map the original information space to a smaller space, but to ensure that whenever a successful plan exists over the original space, one will also exist over the smaller space. This idea will now be formalized.

Let $\Phi : \mathcal{I} \rightarrow \mathcal{I}^c$ denote a surjective (onto) mapping from an information space, \mathcal{I} , to a *collapsed information space*, \mathcal{I}^c . Usually, \mathcal{I}^c is selected to be as small as possible while ensuring that satisfactory plans still exist. To make this precise, some definitions are needed to relate the set of possible plans over \mathcal{I} to the plans over \mathcal{I}^c , which is generally considered to be smaller. For a given information space, \mathcal{I} , let $\Pi(\mathcal{I})$ denote the set all possible plans, $\pi : \mathcal{I} \rightarrow U$. This notation can also be applied to derived and collapsed information states, to yield $\Pi(\mathcal{I}^\circ)$ and $\Pi(\mathcal{I}^c)$, respectively.

One must be very careful in designing Φ because it may potentially destroy possible solutions. In the worst case, Φ can map \mathcal{I} to a set \mathcal{I}^c that contains only one state. Clearly this is a bad idea. Let $\mathcal{I}^c = \{\eta_0\}$. The set of all plans of the form $\mathcal{I}^c \rightarrow U$ is dramatically reduced. There are only $|U|$ possible plans, each of which applies a fixed $u \in U$ over all stages. The set, $\Pi(\mathcal{I}^c)$, of all plans over \mathcal{I}^c

can generally be considered as subset of $\Pi(\mathcal{I})$, defined as

$$\Pi_{\Phi}(\mathcal{I}) = \{\pi \in \Pi(\mathcal{I}) \mid \exists \pi' \in \Pi(\mathcal{I}^c) \text{ such that } \forall \eta \in \mathcal{I}, \pi(\eta) = \pi'(\Phi(\eta))\}. \quad (11.37)$$

In words this means that any plan in $\Pi(\mathcal{I})$ can be represented as a plan in $\Pi_{\Phi}(\mathcal{I}^c)$. In general, there will be many information states in \mathcal{I} that map to a single information state in \mathcal{I}^c . For these information states, the action, $\pi(\eta)$ must remain fixed.

A useful way to interpret $\Pi_{\Phi}(\mathcal{I})$ is obtained by considering the partition of \mathcal{I} that is induced by Φ as follows. For each $\eta \in \mathcal{I}^c$, let $\Phi^{-1}(\eta)$ denote the set of $\eta' \in \mathcal{I}$ for which $\Phi(\eta') = \eta$. By constructing this set for each $\eta \in \mathcal{I}^c$, a partition of \mathcal{I} is formed. For a plan to lie in $\Pi_{\Phi}(\mathcal{I})$, it must hold a constant value over each set in the partition. In the extreme case in which \mathcal{I}^c contains only one element, the partition contains one set, \mathcal{I} itself. In the other extreme, Φ is the identity mapping; in this case, the partition contains only singleton elements, one for each $\eta \in \mathcal{I}$. Intuitively, the partition represents the “resolution” over which a plan can be defined. The two given extremes represent the lowest and highest possible resolutions. Once Φ is selected, every plan must be adapted to \mathcal{I}^c by keeping a fixed value over each set in the induced partition.

The main concern when selecting Φ is that the restriction to $\Pi_{\Phi}(\mathcal{I})$ does not severely limit the quality of solutions that can be produced. In the context of feasible planning, one must ensure that Φ does not destroy feasibility. If a feasible plan exists in $\Pi(\mathcal{I})$, then one must also exist in $\Pi_{\Phi}(\mathcal{I})$. This condition might be required to hold for all initial conditions, $\eta_0 \in \mathcal{I}_0$, or maybe Φ is designed to preserve feasibility for a particular initial condition. For problems that involve optimality, we may require that among the set of optimal plans in $\Pi_{\Phi}(\mathcal{I})$, at least one must lie in $\Pi_{\Phi}(\mathcal{I})$. This requirement could also be weakened by requiring that only an approximately-optimal plan exist in $\Pi_{\Phi}(\mathcal{I})$.

It is important to be aware that a plan in $\Pi_{\Phi}(\mathcal{I})$ might not behave the same way as the corresponding plan in $\Pi(\mathcal{I}^c)$. This can be seen by recalling the information transition equation, $f_{\mathcal{I}}$, from (11.20) and (11.21). Once Φ is applied, then the plan causes transitions to occur over the collapsed information space. Suppose for illustration purposes that there are no nature actions, which yields (11.20). Let $\eta_k \in \mathcal{I}$ denote an information state in the original information space. According to (11.20), applying some u_k yields $\eta_{k+1} = f_{\mathcal{I}}(\eta_k, u_k)$ on the original information space. Let $\eta'_k \in \mathcal{I}^c$ denote the collapsed information state for which $\eta'_k = \Phi(\eta_k)$. If the same action, u_k , is applied using the collapsed information space, then $\eta'_{k+1} = f_{\mathcal{I}}(\eta'_k, u_k)$ is obtained. The problem is that η'_{k+1} might not be the same as $\Phi(\eta_{k+1})$. Algebraically, this means that $f_{\mathcal{I}}$ and Φ generally do not commute. Applying $f_{\mathcal{I}}$ and then Φ to an information state η_k is not necessarily equivalent to applying Φ and then $f_{\mathcal{I}}$. This problem will be illustrated in Example 11.3.3.

The derived information states represent ideal examples of collapsing the information space. Using \mathcal{I}^c instead of \mathcal{I} preserves feasibility and optimality for virtually any planning problem. For particular problems, however, it may be pos-

sible to obtain much smaller information spaces that also preserve these properties. An interesting example of this is given in Section ??.

11.2.4 Limited Memory Models

One general way to reduce the size of the information states is to limit the amount of memory. Except in special cases, this usually does not preserve the feasibility or optimality of the original problem. Nevertheless, such models are very useful in practice when there appears to be no other way to reduce the size of the information space. Furthermore, these models occasionally do preserve the desired properties of feasibility, and maybe also optimality.

Previous i stages Under this model, the history is truncated. Any actions or observations received earlier than i stages ago are dropped from memory. This yields an information state defined as

$$\eta_k = (u_{k-i}, \dots, u_{k-1}, y_{k-i+1}, \dots, y_k), \quad (11.38)$$

assume that $i > 0$ and $k > i$. If $i \leq k$, then the information state is defined in the usual way, given by (11.13). In general, the action and observation histories could be truncated at different stages. The advantage of this approach, if it leads to a solution, is that the length of the information state no longer grows with the dimension of the space. If X and U are finite, then the information space will also be finite, even without using derived information states.

Sensor feedback An interesting case is obtained by removing all but the last sensor observation from the information state. This yields $\eta_k = y_k$, which is referred to as *sensor feedback*. In this case, all decisions are made directly in terms of the sensor reading. A plan, π , can therefore be considered as a mapping: $\pi : Y \rightarrow U$. In some contexts, this may be referred to as a *purely reactive plan*. There are generally many problems which have solutions when information spaces are used, but there exist no solutions that use sensor feedback. However, it may be worth determining whether such a solution exists. Such solutions tend to be simpler to implement in practice. Certainly, if a sensor-feedback exists for a problem, and feasibility is the only concern, then it is pointless to design and implement a plan in terms of the entire information space.

EXAMPLE?

11.3 Examples for Discrete State Spaces

11.3.1 Basic Nondeterministic Examples

First, consider a simple example that uses the sign sensor of Example 11.1.3.

Example 11.3.1 (Using the Sign Sensor) Let $X = \mathbb{Z}$, $U = \{-1, 1, u_T\}$, $Y = \{-1, 0, 1\}$, and $y = h(x) = \text{sgn}x$. For the state transition equation, $x_{k+1} = f(x_k, u_k) = x_k + u_k$. There are no nature actions that interfere with the state transition equation or the sensor mapping. Therefore, future information states are predictable. The information transition equation, $f_{\mathcal{I}}$, is $\eta_{k+1} = f_{\mathcal{I}}(\eta_k, u_k)$. Suppose that initially, $\eta_0 = X$, which means that any initial state is possible. The goal is to reach and terminate at $0 \in X$.

An information state at stage k appears as:

$$\eta_k = (X, u_1, \dots, u_{k-1}, y_1, \dots, y_k). \quad (11.39)$$

A typical value appears as $\eta_5 = (X, -1, 1, 1, -1, 1, 1, 1, 0)$. Using the derived information space, \mathcal{I}° , from Section 11.2.1, $\mathcal{I}^\circ = \text{pow}(X)$, which is uncountably infinite. By looking carefully at the problem, however, it can be seen that most of the derived information states are not reachable. If $y_k = 0$, it is known that $x_k = 0$; hence, $\eta_k = \{0\}$. If $y_k = 1$, it will always be the case that $\eta_k = \{1, 2, \dots\}$. If $y_k = -1$, then $\eta_k = \{\dots, -2, -1\}$. From this, a plan, π , can be specified over these three derived information states. For the first one, $\pi(\eta_k) = u_T$. For the other two, $\pi(\eta_k) = -1$ and $\pi(\eta_k) = 1$, respectively. Based on the sign, the plan tries to move towards 0. If different initial conditions are allowed, then more derived information states can be reached, but this was not required as the problem was defined. Note that optimal-length solutions are produced by the plan. ■

The next example provides a simple illustration of solving a problem without ever knowing the exact state. This leads to the *goal recognizability* problem [?].

Example 11.3.2 (Goal Regonizability) Let $X = \mathbb{Z}$, $U = \{-1, 1, u_T\}$, and $Y = \mathbb{Z}$. For the state transition equation, $x_{k+1} = f(x_k, u_k) = x_k + u_k$. Now suppose that for sensing, a variant of Example 11.1.7, sensor disturbance is used, $y = h(x, \psi)$, and $\Psi = \{-5, \dots, 5\}$. Suppose that once again, $\eta_0 = X$. In this case, it is possible to guarantee that a goal, $X_G = \{0\}$, is reached because of the *goal recognizability problem*. The disturbance in the sensor mapping does not allow precise enough state measurements to deduce the precise goal state. If the goal region, X_G is enlarged to $\{-5, 5\}$, then the problem can be solved. Due to the disturbance, the derived information state will always be a subset of consecutive sequence of 11 states. It is simple to derive a plan that moves this interval until the derived information state becomes a subset of X_G . When this occurs, then the plan applies u_T . In solving this problem, the exact state never had to be known. ■

The problem shown in Figure 11.4 will serve two purposes. First, it is an example of *sensorless planning*, which means that there are no observations. This is an interesting class of problems because it appears that no information can be gained regarding the state. Counterintuitively, it turns out for this example

and many others that the plans can be designed that estimate the state. The second purpose is to illustrate how the information space can be dramatically collapsed using the concepts of Section 11.2.3. The derived information space for this example initially contains 2^{19} states, but it can be nicely collapsed to a small number of states for planning purposes.

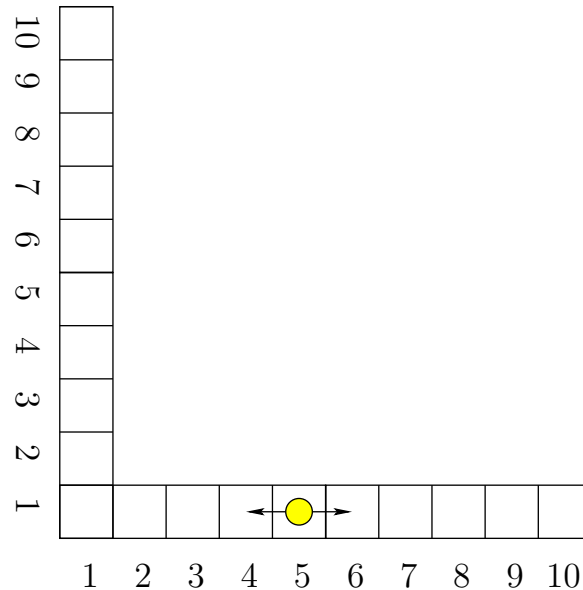


Figure 11.4: An example that involves 19 states. There are no sensor observations; however, actions can be chosen that enable the state to be estimated. The example provides an illustration of collapsing the information space.

Example 11.3.3 (Moving in an L-shaped Corridor) State space, X , for example shown in Figure 11.4 has 19 states, each of which corresponds to a location on one of the white tiles. For convenience, let each state be denoted by (i, j) . There are 10 *bottom states*, denoted by $(1, 1), (2, 1), \dots, (10, 1)$, and 10 *left states*, denoted by $(1, 1), (1, 2), \dots, (1, 10)$. Since $(1, 1)$ is both a bottom state and a left state, it will be called the *corner state*.

It is assumed for this problem that there are no sensor observations. Nature, however, interferes with the state transitions, which leads to a form of nondeterministic uncertainty. If we try to apply an action that takes one step, nature may cause two or three steps to be taken, if possible. This can be modeled as follows. Let

$$U = \{(1, 0), (-1, 0), (0, 1), (0, -1)\} \quad (11.40)$$

and let $\Theta = \{1, 2, 3\}$. The state transition equation is defined as $f(x, u, \theta) = x + \theta u$, unless it is possible to move to the required location. For example, if $x = (5, 1)$, $u = (-1, 0)$, and $\theta = 2$, then the resulting next state is $(5, 1) + 2(-1, 0) = (3, 1)$. If it is not possible to move to the state $x + \theta u$, then the state remains fixed, $f(x, u, \theta) = x$.

It is assumed for this problem that there are no sensor observations. Therefore, the information state at stage k is

$$\eta_k = (u_1, \dots, u_{k-1}). \quad (11.41)$$

Now use the derived information space, $\mathcal{I}^\circ = \text{pow}(X)$. The initial state, $x_1 = (10, 1)$ is given, which means that the initial information state, η_1 , is $\{(10, 1)\}$. The goal is to arrive at the information state, $\{(1, 10)\}$. This means that the task is to design a plan that moves from the lower right to the upper left.

With perfect information, this would be trivial; however, without sensors the uncertainty may grow very quickly. For example, after applying the action $u_1 = (-1, 0)$ from the initial state, the derived information state becomes $\{(7, 1), (8, 1), (9, 1)\}$. After $u_2 = (-1, 0)$ it becomes $\{(4, 1), \dots, (8, 1)\}$. A nice feature of this problem, however, is that uncertainty can be reduced without sensing. Suppose that for 100 stages, we continue to apply $u_k = (-1, 0)$. What is the resulting information state? As the corner state is approached, the uncertainty is reduced because the state cannot be further changed by nature. It is known that each action, $u_k = (-1, 0)$, decreases the X coordinate by at least one each time. Therefore, after 9 or more stages, it is known that $\eta_k = \{(1, 1)\}$. Once this is known, then the action $(0, 1)$ can be applied. This will again increase uncertainty as the state moves through the set of left states. If $(0, 1)$ is applied 9 or more times, then it is known for certain that $x_k = (1, 10)$, which is the required goal state.

A successful plan has now been obtained: 1) apply $(-1, 0)$ for 9 stages, 2) then apply $(0, 1)$ for 9 stages. Recall from Section 11.1.3 that a strategy is generally specified as $\pi : \mathcal{I} \rightarrow U$; however, for this example, it appears that only a sequence of actions is needed. The actions do not depend on the information state. Why did this happen? If no observations are obtained during execution, then there is no way to use feedback. There is nothing to learn by executing the plan. In general, for problems that involve no sensors and a fixed initial information state, a *path* in the information space can be derived from a plan. It is somewhat strange that this path is completely predictable, even though the original problem may involve substantial uncertainties. We always know precisely what will happen in terms of the information states.

To make the situation more interesting, assume that any subset of X could be used as the initial condition. In this case, a plan $\pi : \mathcal{I} \rightarrow U$ must be formulated to solve the problem. From each initial information state η , a path in \mathcal{I}° can still be computed from π . Specifying a plan over all of \mathcal{I}° appears quite complicated, which motivates the next consideration.

The ideas from Section 11.2.3 can be applied here to collapse the information down from 2^{19} (over half of a billion) to 3. The mapping $\Phi\mathcal{I} \rightarrow \mathcal{I}^c$ must be constructed. We have already mapped \mathcal{I} to \mathcal{I}° by using derived information states; therefore, the collapsed information space will be obtained by defining $\Phi : \mathcal{I}^\circ \rightarrow \mathcal{I}^c$. We first make a naive attempt to collapse the information state

down to only three states. This illustrates the issue mentioned near the end of Section 11.2.3. Let $\mathcal{I}^c = \{g, l, a\}$, in which g denotes “goal”, l denotes “left”, and a denotes “any”. The mapping is

$$\Phi(\eta) = \begin{cases} g & \text{if } \eta = \{(10, 1)\} \\ l & \text{if } \eta \text{ is a subset of the set of left states} \\ a & \text{otherwise} \end{cases} \quad (11.42)$$

It might seem that this collapsed information space will lead to a very compact plan for solving the problem. Based on the successful plan described so far, the plan on \mathcal{I}^c can be defined as $\pi(g) = u_T$, $\pi(l) = (0, 1)$, and $\pi(a) = (-1, 0)$. What is wrong with this? Suppose that the initial state is $(10, 1)$. There is no way to require that $u_k = (-1, 0)$ is applied 9 times to reach the l state. If $(-1, 0)$ is applied to the a state, then it is not possible to determine when the transition to l will occur.

Now consider a different collapsed information space. Suppose that are 19 collapsed information states, which includes g as defined previously, l_i for $1 \leq i \leq 9$, and a_i for $2 \leq i \leq 10$. The mapping Φ is defined as $\Phi(\eta) = g$ if $\eta = \{(10, 1)\}$. Otherwise, $\Phi(\eta) = l_i$, for the largest value of i such that η is a subset of $\{(i, 1), \dots, (10, 1)\}$. If there is no such value for i , then $\Phi(\eta) = a_i$, for the smallest value of i such that η is a subset of $\{(1, 1), \dots, (1, 10), (2, 1), \dots, (i, 1)\}$. Now the plan may be defined as $\pi(g) = u_T$, $\pi(l_i) = (0, 1)$, and $\pi(a_i) = (-1, 0)$. Although it might not appear to be any better than the plan obtained from collapsing \mathcal{I}_0 to three states, the important difference is that the correct information state transitions occur. For example, if $u_k = (-1, 0)$ is applied at a_5 , then a_4 is obtained. If $u = (-1, 0)$ is applied at a_2 , then l_1 is obtained. From there, $u = (0, 1)$ is applied to yield l_2 . These actions can be repeated until eventually l_9 and g are reached. ■

11.3.2 Nondeterministic Finite Automata

An interesting connection lies between the ideas of this chapter and the theory of finite automata, which is part the theory of computation (see [339, 711]). In Section ??, it was mentioned that determining whether there exists some string that is accepted by a deterministic finite automaton (DFA) is equivalent to a discrete fesaible planning problem. If unpredictability is introduced into the model, then a nondeterministic finite automaton (NFA) is introduced, as depicted in Figure 11.5. This represents one of the simplest examples of nondeterminism in theoretical computer science. Such nondeterministic models in general serve as a powerful tool for defining models of computation and their associated complexity classes. It turns out that these models give rise to interesting examples of information spaces.

A *nondeterministic finite automaton* (NFA) is typically described using a directed graph as shown in Figure ??,b, and is considered as a special kind of finite

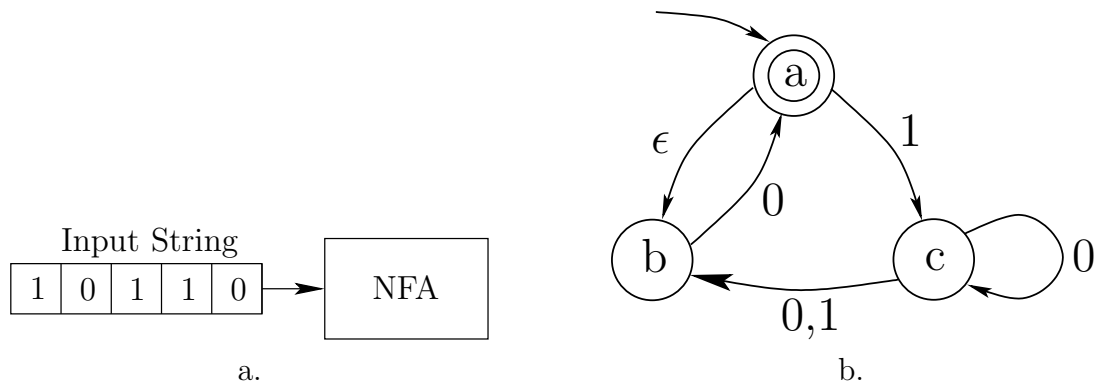


Figure 11.5: a) An NFA is a state machine that reads an input string and decides whether or not to accept it. b) A graphical depiction of a nondeterministic finite automaton (NFA).

state machine. Each vertex of the graph represents a state, and edges represent possible transitions. An *input string* of finite length is read by the machine. For the example, the input string is assumed to be a binary sequence that consists of 0s and 1s. The initial state is designed by an inward arrow that has no source vertex, as shown pointing into state a in Figure ??b. The machine starts in this state and reads the first symbol of the input string. Based on its value, it makes appropriate transitions. For a deterministic finite automaton (DFA), the next state must be specified for each of the two inputs, 0 and 1, from each state. From state in an NFA, there may be any number of outgoing edges (including none) that represent the response to a single input. For example, there are two outgoing edges if 0 is read from state c (the arrow from c to b actually corresponds to two directed edges, one for 0 and the other for 1). There are also edges designated with a special ϵ symbol. If a state has an outgoing ϵ , the state may immediately transition along the edge without reading another symbol. This may be iterated any number of times, for any outgoing ϵ edges that may be countered, without reading the next input symbol. The nondeterminism arises from the fact that there are multiple choices for possible next states due to multiple edges for the same input and ϵ transitions. There is no sensor that indicates which state is actually chosen. The interpretation in the theory of computation is that when there are multiple choices, the machine clones itself, and one copy runs each choice. It is like having multiple universes in which each different possible action of nature is occurring simultaneously. If there are no outgoing edges for a certain combination of state and input, then the clone dies. Any states that are a double boundary, such as state a in Figure 11.5, indicate *accept states*. When the input string ends, the NFA is said to *accept* the input string if there exists at least one alternate universe in which the final machine state is an accept state.

The formulation usually given for NFAs seems very close to Formulation ??, for discrete feasible planning. Here is a typical NFA formulation [711], which

formalizes the ideas depicted in Figure 11.5:

Formulation 11.3.1 (Nondeterministic Finite Automaton)

1. A finite state space, X .
2. A finite *alphabet*, Σ , which represents the possible input symbols. Let $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$.
3. A *transition function*, $\delta : X \times \Sigma_\epsilon \rightarrow \text{pow}(X)$. For each state and symbol, a set of outgoing edges is specified by indicating the states that are reached.
4. A *start state*, $x_0 \in X$.
5. A set, $A \subseteq X$ of *accept states*.

Example 11.3.4 (Three-State NFA) The example in Figure 11.5 can be expressed using Formulation 11.3.1. The components are $X = \{a, b, c\}$, $\Sigma = \{0, 1\}$, $\Sigma_\epsilon = \{0, 1, \epsilon\}$, $x_0 = a$, and $A = \{a\}$. The state transition equation requires the specification of a state for every $x \in X$ and symbol in Σ_ϵ :

| | | | | |
|---|-------------|-------------|-------------|---------|
| | 0 | 1 | ϵ | |
| a | \emptyset | $\{c\}$ | $\{b\}$ | |
| b | $\{a\}$ | \emptyset | \emptyset | (11.43) |
| c | $\{b, c\}$ | $\{b\}$ | \emptyset | |

■

Now consider reformulating the NFA and its acceptance of strings as a kind of planning problem. An input string can be considered as a plan that uses no form of feedback; it is fixed sequence of actions. The planning problem is to determine whether a string exists that is accepted by the NFA. Because there is no feedback, there is no sensing model. The initial state is known, but subsequent states cannot be measured. The history at stage k reduces to $\lambda_k = U^{k-1} = (u_1, \dots, u_{k-1})$, the sequence actions that have been applied so far. The nondeterminism can be accounted for by defining nature actions that interfere with state transitions. This results in the following formulation, which is described in terms of Formulation 11.3.1:

Formulation 11.3.2 (An NFA Planning Problem)

1. A finite state space, X .
2. An action space $U = \Sigma \cup \{u_T\}$.
3. A *state transition function*, $F : X \times U \rightarrow \text{pow}(X)$. For each state and symbol, a set of outgoing edges is specified by indicating the states that are reached.

4. An *initial state*, $x_0 = x_i$.
5. A set, $X_g = A$ of *goal states*.

The information space, \mathcal{I} , is defined using

$$\mathcal{I}_k = U^{k-1} \quad (11.44)$$

for each $k \in \mathbb{N}$, and taking the union as defined in (11.17). It is assumed that the initial state of the NFA is always fixed; therefore, X does not appear in the definition of \mathcal{I} . Because there is no feedback, a plan, π , is just a sequence of actions, as defined for the problems in Chapter 2.

For expressing the planning task, it is best to use the derived information space, $\mathcal{I}^\circ = \text{pow}(X)$, from Section 11.2.1. Thus, each information state, $\mathcal{I} \in \mathcal{I}^\circ$ is a subset of X which corresponds to the possible current states of the machine. The initial condition could be any subset of X because ϵ transitions can occur from x_i . Subsequent derived information states follow directly from F . The task is to compute a plan of the form

$$\pi = (u_1, u_2, \dots, u_K, u_T, u_T, \dots), \quad (11.45)$$

which results in an information state $\eta_{K+1} \in \mathcal{I}^\circ$ for which $\eta_{K+1} \cap X_g \neq \emptyset$. This means that at least one possible state of the NFA must lie in X_g after the termination action is applied. This condition is much weaker than a typical planning requirement. Using worst-case analysis, a typical requirement would be that *every* possible NFA state lies in X_g .

The problem given in Formulation 11.3.2 does not precisely a specialization of Formulation ?? because of the state transition function. For convenience, F was directly defined, instead of explicitly requiring that f is defined in terms of nature actions, $\Theta(x, u)$, which in this context depend on both x and u for an NFA. There is one other small issue regarding this formulation. In the planning problems considered in this book, it is always assumed that there is a current state. For an NFA, it was already mentioned that if there are no outgoing edges for a certain input, then the clone of the machine dies. This means that potential current state ceases to exist. It is even possible that every clone dies, which leaves no current state for the machine. This can be easily enabled by directly defining F ; however, planning problems must always have a current state. To resolve this issue, we could augment X in Formulation 11.3.2 to include an extra *dead* state, which signifies the death of a clone when there are no outgoing edges. A dead state can never lie in X_g , and once a transition to a dead state occurs, the state remains dead for all time. In this section, the state space will not be augmented in this way; however, it is important to note that the formulation can easily be made consistent with Formulation 11.3.2.

The planning model can now be compared to the standard use of NFAs in the theory of computation. A *language* of an NFA is defined to the set of all input

strings that it accepts. The planning problem formulated here determines whether there exists a string (which is a plan that ends with termination actions) that is accepted by the NFA. Equivalently, a planning algorithm determines whether or not the language of an NFA is empty. Constructing the set of all successful plans is equivalent to determining the language of the NFA.

Example 11.3.5 (Planning for the Three-State NFA) The example in Figure 11.5 can be expressed using Formulation 11.3.1. The components are $X = \{a, b, c\}$, $\Sigma = \{0, 1\}$, $\Sigma_\epsilon = \{0, 1, \epsilon\}$, $x_0 = a$, and $F = \{a\}$. The function $F(x, u)$ is defined as

$$\begin{array}{c|cc} & 0 & 1 \\ \hline a & \emptyset & \{c\} \\ b & \{a, b\} & \emptyset \\ c & \{b, c\} & \{b\} \end{array} \quad (11.46)$$

The derived information space is

$$\mathcal{I}^\circ = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\} \quad (11.47)$$

in which the initial condition is $\eta_0 = \{a, b\}$ because an ϵ transition occurs immediately from a . An example plan that solves the problem is $(1, 0, 0, u_T, \dots)$. This corresponds to sending an input string 110 through the NFA depicted in Figure 11.5. The sequence of information states obtained during the execution of the plan is

$$\{a, b\} \xrightarrow{1} \{c\} \xrightarrow{0} \{b, c\} \xrightarrow{0} \{a, b, c\} \xrightarrow{u_T} \{a, b, c\}. \quad (11.48)$$

■

A basic theorem from finite automata states that for the set of strings accepted by an NFA, there exists a DFA (deterministic) that accepts the same set [711]. This is proven by constructing a DFA directly from the derived information space. Each derived information state can be considered as a state of a DFA. Thus, the DFA has 2^n states, if the original NFA has n states. The state transitions of the DFA are derived directly from the transitions between derived information states. When an input (or action) is given, then a transition occurs from one subset of X to another. A corresponding transition is made between the two corresponding states in the DFA. This construction is an interesting example of how the information space is new state space that arises when the states of the original state space are unknown. Even though the information space is usually larger than the original state space, its states are always known. Therefore, the behavior appears the same as in the case of perfect state information. This idea is very general, and may be applied to many problems beyond DFAs and NFAs.

11.3.3 Probabilistic Examples

POMDPs

Exploring an $(n - 1)$ -simplex embedded in \mathbb{R}^n .

Let the vertices be $(0, 0, \dots, 0, 1)$, $(0, 0, \dots, 0, 1, 0)$, \dots , $(1, 0, \dots, 0)$.

Each point in the simplex corresponds to a probability distribution over X .

It is specified by the barycentric coordinates.

A Sensor Planning Problem Can the actions control the sensor? We must allow the case of actions to determine where to sense.

Need a good example of goal recognizability and the termination problem.

11.4 Continuous State Spaces

This section takes many of the concepts that have been developed in Sections 11.1 and 11.2, and generalizes them to continuous state spaces. This represents an important generalization because the configuration space concepts, on which motion planning was based in Part II, are all based on continuous state spaces. In this section, the state space might be a configuration space, $X = \mathcal{C}$, as defined in Chapter ??, or any other continuous state space. Because it may be a configuration space, many interesting problems can be drawn from robotics.

During the presentation of the concepts of this section, it will be helpful to recall analogous concepts that were already developed for discrete state spaces. In many cases, the formulations appear identical. In others, the continuous case is more complicated, but usually maintains some of the concepts from the discrete case. It will be seen after introducing many continuous sensing models in Section 11.5 that many problems formulated in continuous spaces are even more elegant and easy to understand than their discrete counterparts.

11.4.1 Discrete-Stage Information Spaces

It is assumed here that there are discrete stages, k . Let $X \subseteq \mathbb{R}^m$ be an n -dimensional manifold, for $n \leq m$, called the *state space*.² Let $Y \subseteq \mathbb{R}_m$ be an n_y -dimensional manifold, for $n_y \leq m$, called the *observation space*. For each $x \in X$, let $\Psi(x) \subseteq X$ be an n_n -dimensional manifold, for $n_n \leq m$, called the set of *nature observation actions*. The three kinds of sensors mappings, h , defined in Section 11.1.1 are possible, to yield either *state mapping*, $y = h(x)$, *state-sensor mapping* $y = h(x, \psi)$, or *history-based*, $y = h(x_1, \dots, x_k, y)$. For the case of a state mapping, the preimages, $H(y)$, once again induce a partition of X . Preimages can also be defined for state-action mappings, but they do not necessarily induce a partition of X .

²If you did not read Chapter 4, and are not familiar with manifold concepts, then assume $X = \mathbb{R}^n$; it will not make much difference. Make similar assumptions for Y and $\Psi(x)$.

Many interesting sensing models can be formulated in continuous state spaces. Section 11.5 provides a kind of sensor catalog. In general, there is once again the choice of nondeterministic or probabilistic uncertainty if nature observation actions are used. If nondeterministic uncertainty is used, there is nothing more to define. Probabilistic models are defined in terms of a probability density function, $p : \Psi \rightarrow [0, \infty)$.³

The information space definitions from Section 11.1.2 remain the same, with the understanding that all of the variables are continuous. Thus, (??) and (??) serve as the definitions of \mathcal{I}_k and \mathcal{I} . Let $U \subseteq \mathbb{R}^m$ be an n_u -dimensional manifold for $n_u \leq m$. For each $x \in X$ and $u \in U$, let $\Theta(x, u)$ be an n_θ -dimensional manifold for $n_\theta \leq m$. A discrete-stage information space planning problem over continuous state spaces can be easily formulated by taking Formulation 11.1.1 and replacing each discrete variable by its continuous counterpart that uses the same notation. Therefore, the full formulation is not given.

11.4.2 Continuous-Time Information Spaces

Now assume that there is a continuum of stages. Most of the components of Section 11.4.1 remain the same. The spaces, $X, Y, \Psi(x), U, \Theta(x, u)$, remain the same (REALLY???). The sensor mapping also remains the same. The main difference occurs in the state transition equation. To specify it correctly in the most general form, differential equations are necessary. To make the modeling problem worse, expressing the effect of nature actions requires differential inequalities \square in the case of nondeterministic uncertainty, and stochastic differential equations \square in the probabilistic case. Both of these concepts are generalizations of differential equations that are well beyond the scope of this book. The ideas presented here can be generalized to these cases, once the appropriate technical considerations required for these advanced topics are resolved.

The approach taken here is to assume a specialized formulation to avoid these technical difficulties.

Need to avoid actions. This means that plans directly specify the state. Does this even make sense for state feedback? It is like the path is specified in a coordinate frame, but the true frame is not known...

Let t denote time, $t \in T = [0, \infty)$.

Let $U \subseteq \mathbb{R}^m$ be the *input space*.

Let $u : [0, \infty) \rightarrow U$ be called the *input history*.

Let a *state trajectory*, $x : [0, \infty) \rightarrow X$, denote a solution to the following system of n differential equations,

$$\frac{dx}{dt} = f(x(t), u(t)),$$

in which f is a smooth mapping on X and U .

³We assume that all continuous spaces are measure spaces, and all p functions are measurable functions over these spaces.

Note: $u(t)$ could be derived from a state-feedback mapping $\gamma : X \rightarrow U$ to obtain $f(x(t), \gamma(x(t)))$.

We will not be able to do this because of the next topic...

Sensor Model:

Let $Y \subseteq \mathbb{R}^k$ be the *sensor space*.

The sensor space models the set of possible instantaneous sensor readings.

For each $t \in [0, \infty)$, the sensor value $y(t)$ is given by

$$y(t) = h(x(t)),$$

for some specified mapping $h : X \rightarrow Y$.

Often, h is not injective, which causes information loss. (projection, fibration)

Let $y : [0, \infty) \rightarrow Y$ be called the *sensor history*.

The Information State:

Let x_t , u_t , and y_t denote the restrictions of x , u , and y , respectively, to the domain $[0, t]$.

The information state, η_t is given by $\eta_t = (u_t, y_t)$.

In other words, (input history, sensor history).

Note: Many restricted forms are possible: limited memory, sensorless, no knowledge of inputs, etc.

The *information space*, \mathcal{I} , is the set of all possible information states.

Remember that \mathcal{I} is a function space that is determined once X , U , Y , f and h are given.

Modeling Disturbances:

Let “nature” interfere with motions and sensors.

Let $V \subseteq \mathbb{R}^p$, $W \subseteq \mathbb{R}^q$, be *disturbance spaces*. Let $v(t) \in V$ and $w(t) \in W$ for all $t \in [0, \infty)$.

State transition equation:

$$\frac{dx}{dt} = f(x(t), u(t), v(t))$$

Sensing model:

$$y(t) = h(x(t), w(t))$$

The disturbances, $v(t)$ and $w(t)$, may be either

1. Simply unknown, or
2. Modeled as a random process.

Examples of Unknown Disturbance

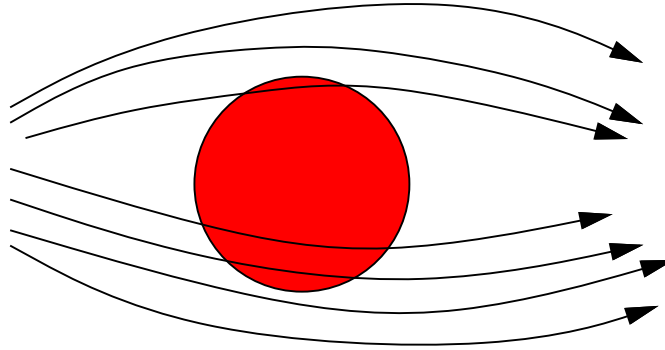
Let B denote a disc of unit radius, centered at the origin of \mathbb{R}^2 .

Suppose $X \subset \mathbb{R}^2$, $U = B$, $Y = \mathbb{R}^2$, $f(x(t), u(t)) = u(t)$, $V = B$, $h(x(t)) = x(t) + v(t)$.

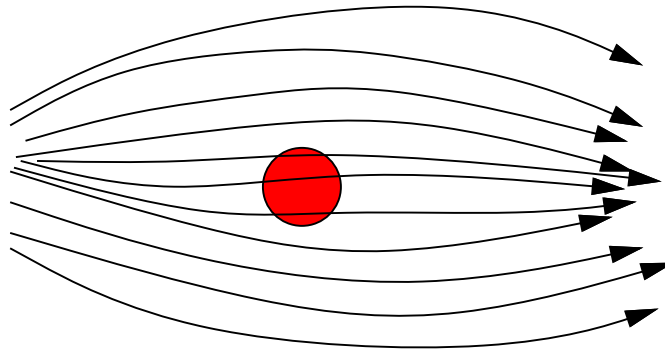
Suppose further that y is continuous.

Consider possible $\eta_t = y_t$ (input history is assumed unknown):

Case 1: A “big” hole in X (radius > 1)



Case 2: A “small” hole in X (radius < 1)



11.4.3 Alternative Representations

Nondeterministic Case:

Given an information state η_t and initial set of states $X_0 \subseteq X$, derive a set $F(\eta_t, X_0) \subseteq X$.

$F(\eta_t, X_0)$ represents a *derived information state*, the set of all possible $x(t) \in X$, given η_t and X_0 .

The *derived information space* is the set of all possible F that can be derived from $\eta_t \in \mathcal{I}$.

Probabilistic Case:

Given an information state η_t an initial probability measure, $p(x(0))$, derive a conditional probability measure, $p(x(t)|\eta_t)$.

$p(x(t)|\eta_t)$ represents a *derived information state* in the probabilistic sense.

The *derived information space* in this case is the set of all probability measures that can be derived from $\eta_t \in \mathcal{I}$.

11.4.4 Approximating Information States

bounding volumes for nondeterministic uncertainty
 moments for probabilistic uncertainty

11.5 Sensors for Continuous Spaces

Many examples can be defined, most of which are for an oriented point in a 2D world, yielding $q = (x, y, \theta)$ and $\mathcal{C} = \mathbb{R}^2 \times S^1$.

1. **Perfect State Measurement:** $h(q) = q$.
2. **Compass:** $Y = S^1$, $h(q) = \theta$. A gyroscope is the 3D version—these work because of precession, which is the effect that keeps bicycles from falling over.
3. **Positioning:** $Y = \mathbb{R}^2$, $h(q) = (x, y)$. Like GPS (but without orientation information).
4. **Contact:** $h(q) = 1$ if $q \in \partial\mathcal{C}_{free}$, and $h(q) = 0$ if $q \in \text{int}(\mathcal{C}_{free})$.
5. **Proximity:** Like contact sensor, but triggers when within a specified range of the wall.
6. **Wheel Odometry:** If accurate, it measures how far the robot has traveled. This is used for *dead reckoning*.
7. **Homing Beacon:** The direction to the goal is known. $H = S^1$, $h(q) = \text{atan2}(x_g - x, y_g - y)$. This was used in bug algorithms, and also is popular in the competitive ratio framework in algorithms. Rather than the goal, beacons may be placed anywhere. They could be individually coded, or confusable.
8. **Geiger Counter:** Gets stronger as the distance to the goal is decreased. Also similar to the “specter detector” used in Scooby Doo to detect ghosts. Again, there could be multiple (radioactive?) sources distributed in the environment.
9. **Speedometer:** Measures the robot speed. Could also measure angular velocity.
10. **Clock:** Measure the elapsed time from the initial state.
11. **Accelerometer:** Measures only acceleration. Only relevant for problems that involve dynamics.
12. **Time-of-Flight:** A unidirectional depth measurement. This is usually obtained from a sonar.

13. **Range Scanner:** Omnidirectional depth map (or limited direction range). Like the SICK laser. Also could characterize stereo vision.
14. **Gap Sensor:** Gives orientations of discontinuities around S^1 . This is used in [?].
15. **Landmarks:** It is known that the robot is within a subset of the state space. Many variations are possible. A whole family of sensors can be obtained by placing static cameras or other sensors around the environment. These can detect the robot and possibly give configuration information when it is within the sensor's active range.
16. **Pebble:** Like the one used in the old maze-searching papers. These can be dropped to mark places where you have been before. Part of the state space might encode the positions of these.
17. **Joint Encoders:** Measures the position of a single manipulator joint.
18. **Force Sensor:** Like the contact sensor, but provides the direction and/or magnitude of the force.

11.6 Examples for Continuous State Spaces

11.6.1 Projection Sensors

State space: $X = \{(x_1, x_2) \in \mathbb{R}^2 \mid x_2 = \sin x_1\}$

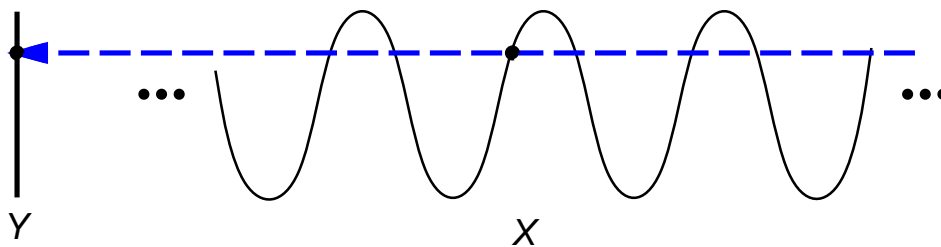
Sensor space: $Y = [-1, 1] \subset \mathbb{R}$

Control model: $f(x(t), u(t)) = u(t)$ and $U(t) = \{u(t) \in \mathbb{R}^2 \mid u \in T(x(t)) \text{ and } \|u\| = 1\}$

Solutions to $\dot{x} = f(x(t), u(t))$ yield continuous state trajectories for each choice of u .

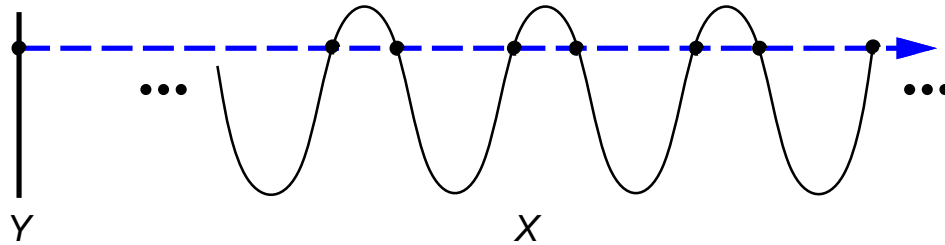
Sensor model: $h(x(t)) = x_2$

Information state: $\eta_t = y_t$ (input history is assumed unknown)



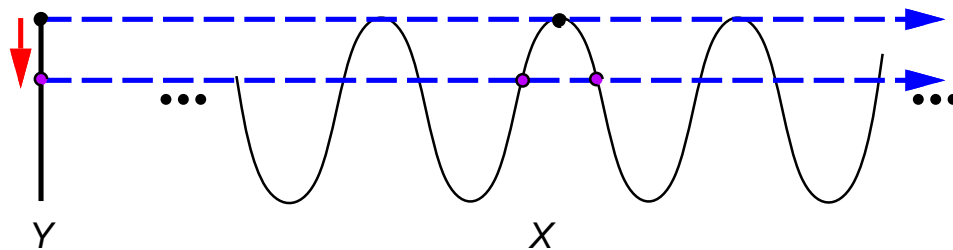
Tracking the Information State

Imagine observing some $y_t \dots$



The derived information state with initial condition $X_0 = X$.

Assume $X_0 = x(0)$, some particular, given initial state.

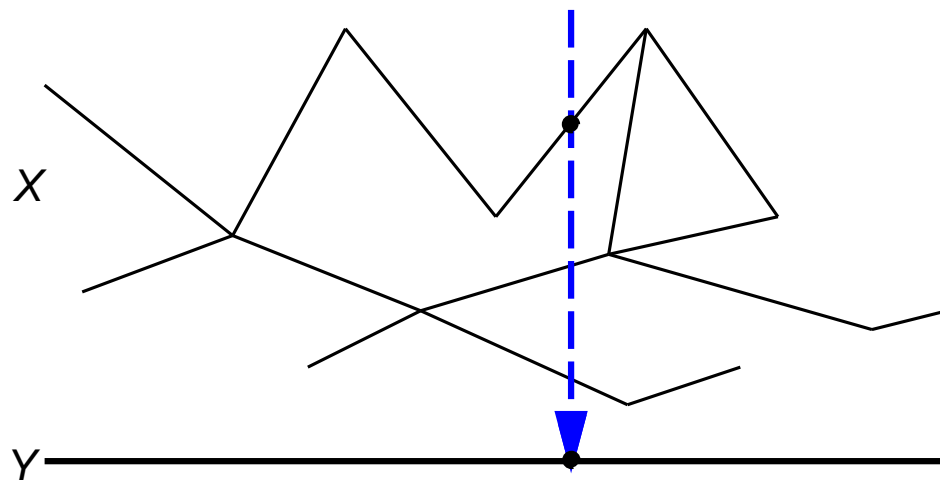


Bifurcations occur when after passing through sensor readings of 1 or -1 .

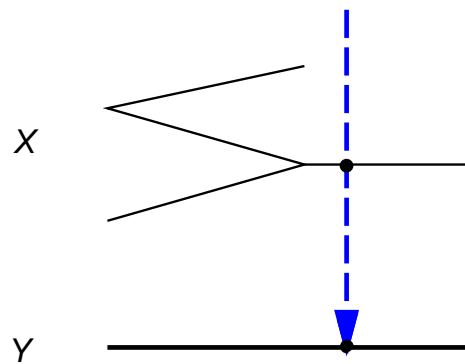
What is the topology of the derived information space?

Traversing a Graph

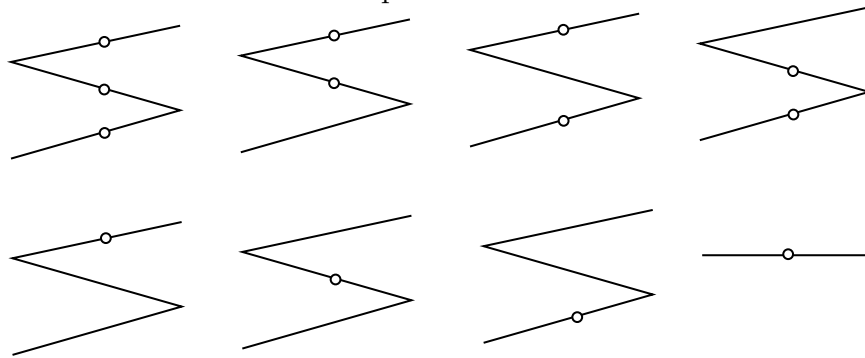
X is a planar graph (connected, 1-dimensional CW complex) embedded in \mathbb{R}^2 :



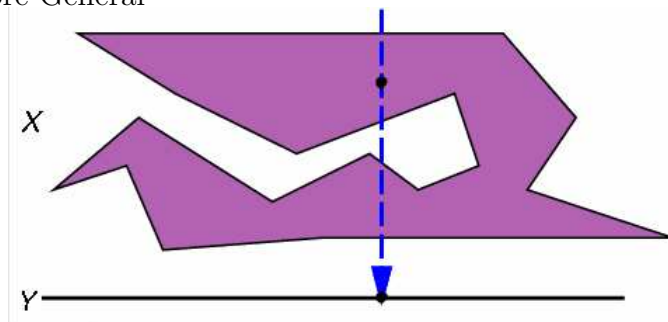
Consider a Simple Example



How is the derived information space connected?



A Little More General



These models are too contrived for real robotics applications.
However, higher-dimensional generalizations are quite relevant.

11.6.2 Sensorless Manipulation

$$\eta_t = u_t$$

A motion strategy is specified by a prescribed input history u .

Sensorless Manipulation (Erdmann, Mason, 1986; Mason, Goldberg, 1990; Akella, Huang, Lynch, Mason, 1997)

Example 11.6.1 See Figure 11.6.

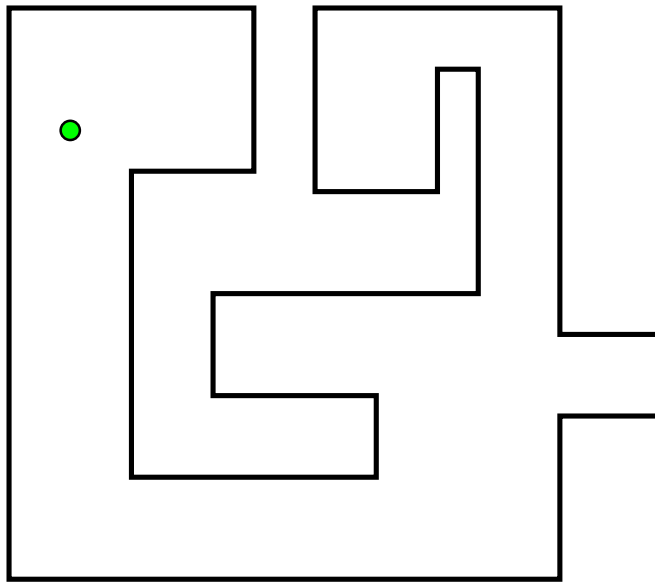


Figure 11.6: Tilt the tray to roll the ball into the desired corner.

Imagine a ball with unknown initial position rolling in a tray.
 Find a sequence of tray tilts that places the ball in a known location.
 Think about nondeterministic derived information states.
 (Example inspired by Mason, Erdmann, 1988 – polygonal part orienting)

Example 11.6.2 (Orienting Parts) See Figure ??.

Mechanical compliance reduces uncertainty.
 Initially, orientation of a planar part is unknown.
 Find a sequence of squeezes that enables the orientation to be known.
 (Mason and Goldberg, 1990)

11.6.3 Environment Spaces

11.7 State Estimation

Need big warnings about how this is classical, but generally not needed in many circumstances. In some sense, estimation defeats the purpose of reasoning about information spaces.

11.7.1 Mapping Histories to States

11.7.2 Kalman Filtering

Linear Gaussian: information space collapses to mean and covariance.

$$X = U = W = V = \mathbb{R}^n.$$

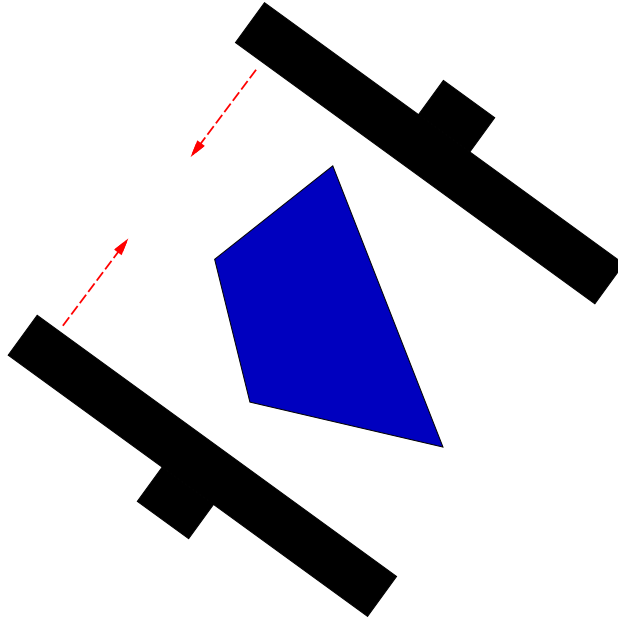


Figure 11.7: A part can be oriented without sensing by performing squeezing operations.

w and v are defined by sampling from a zero-mean Gaussian i.i.d. sequence of random variables on W and V , respectively.

$$x(k+1) = Ax(k) + Bu(k) + v(k)$$

$$y(k) = Cx(k) + w(k)$$

A , B , and C are $n \times n$ matrices with full rank.

If the initial probability measure over X is Gaussian, then all possible derived information states will be Gaussian!

The continuous-time case is similar.

This means the information space can be parameterized by mean and covariance.

The information state computations are called the Kalman filter.

11.8 Multiple Decision Makers

11.8.1 Information Spaces for Everyone

Common state space, but one information space per decision maker.

Strategies

Mention for which conditions Nash equilibria exist, etc.

11.8.2 Extended Form Games

11.8.3 Examples

Give battleship game-like example.

Team theory? Limited communication, but a common goal.

Literature

Information spaces – where have they come from?

- **Stochastic control theory**
Due to disturbances in prediction and measurements, there is imperfect state information.
- **Differential/dynamic game theory**
Modeling unknown state information that results from the choices made by other players.
- **Robotics**
Uncertainty in configuration or state due to sensing limitations.

Related robotics work:

- Preimage Planning (Lozano-Perez, Mason, Taylor, Erdmann 1984)
- Error Detection and Recovery (Donald, 1987)
- Sensorless Manipulation (Erdmann, Mason, 1986; Mason, Goldberg, 1990; Akella, Huang, Lynch, Mason, 1997)
- Perceptual Kinematic Maps (Herve, Cucka, Sharma, 1990)
- Perceptual Equivalence Classes and Information Invariants (Donald, Jennings, 1991; Donald, 1995)
- Pursuit-Evasion (Parsons, 1977; Suzuki, Yamashita, 1992; LaValle, Lin, Guibas, Latombe, Motwani, 1997; Simov, Slutzki, LaValle, 2000)
- Probabilistic Robot Navigation (Simmons, Koenig, 1995)
- Bayesian Localization (Thrun, 1998)

Stochastic control.

POMDPs in AI.

Manipulation planning literature in motion planning.

Information invariants (Donald)

Pebbles and mazes (Blum, Kozen, related papers)

Exercises

1. Derive forward and backwards projections for the discrete case. (this will be several exercises, depending on the different cases; some hints will be given too)

2. Need a simple example that involves showing that part of the info space is not reachable. Also, it can be unidirectional.
3. At the end of Section 11.3.2, it is mentioned that an equivalent DFA can be constructed from an NFA.
 - (a) Give an explicit DFA that accepts the same set of strings as the NFA in Figure 11.5.b.
 - (b) Express the problem of determining whether the NFA in Figure 11.5.b accepts any strings as a planning problem using Formulation 2.2.1.
4. *A problem that generalizes Figure 11.4 to a “plus” or “square” shape.*
5. Show that the information space is not connected for Example 11.4. Give an example of an information state that cannot be reached from the initial information state. Can you characterize all of the connected components?

Chapter 12

Planning in the Information Space

Chapter Status



What does this mean? Check <http://msl.cs.uiuc.edu/planning/status.html> for information on the latest version.

Cover this somewhere:

S. Blind, C. McCullough, S. Akella, and J. Ponce, "Manipulating Parts with an Array of Pins: A Method and a Machine," *International Journal of Robotics Research*, Vol. 20, No. 10, pp. 808-818, October 2001.

S. Akella, W. H. Huang, K. M. Lynch, and M. T. Mason, "Parts Feeding on a Conveyor with a One Joint Robot," *Algorithmica* (Special issue on Robotics), Vol. 26, No. 3/4, pp. 313-344, March/April 2000.

12.1 Information Spaces over Sets of Environments

12.1.1 Maze Searching

Cover old Blum and Kozen-style maze searching. Really interesting stuff!

Give some very simple mazes (e.g., 3x3), to clearly show the information spaces.

Explain how building a perfect map explores the information space.

Explain that BK are collapsing the information space.

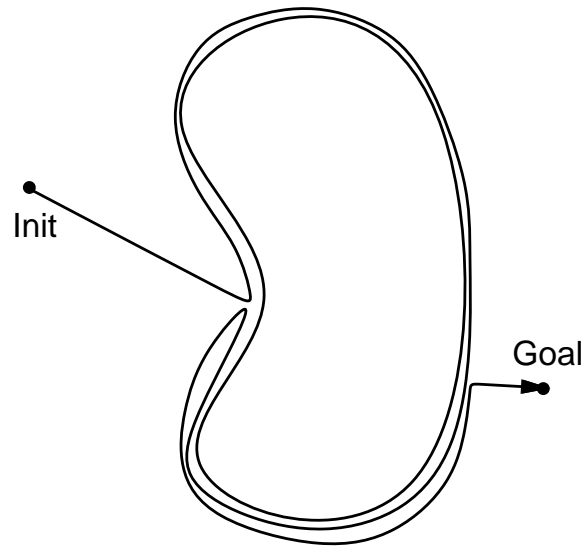
Give the BK exploration algorithm. Lower left corner markers, green-eyed automaton, etc. The automaton requires logarithmic space, which is much less than that required to hold a map!

12.1.2 Bug Algorithms

This section addresses a motion strategy problem that deals with uncertainty with sensing. The Bug algorithms make the following assumptions:

- The robot is a point in a 2D world.
- The obstacles are unknown and nonconvex.
- An initial and goal positions are defined.
- The robot is equipped with a short-range sensor that can detect an obstacle boundary from a very short distance. This allows the robot to execute a trajectory that follows the obstacle boundary.
- The robot has a sensor that allows it to always know the direction and Euclidean distance to the goal.

Bug 1 This robot moves in the direction of the goal until an obstacle is encountered. A canonical direction is followed (clockwise) until the location of the initial encounter is reached. The robot then follows the boundary to reach the point along the boundary that is closest to the goal. At this location, the robot moves directly toward the goal. If another obstacle is encountered, the same procedure is applied.

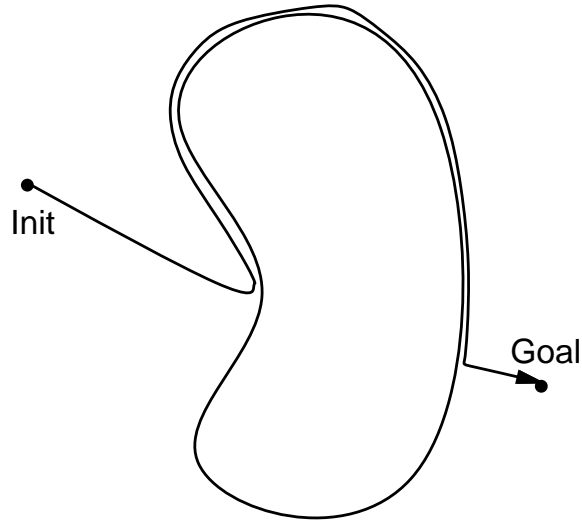


The worst case performance, L , is

$$L \leq d + \frac{3}{2} \sum_{i=1}^N p_i$$

in which d is the Euclidean distance from the initial position to the goal position, p_i is the perimeter of the i^{th} obstacle, and N is the number of obstacles.

Bug 2 In this algorithm, the robot always attempts to move along the line of sight toward the goal. If an obstacle is encountered, a canonical direction is followed until the line of sight is encountered.



The worst case performance, L , is

$$L \leq d + \frac{1}{2} \sum_{i=1}^N n_i p_i$$

in which n_i is the number of times the i^{th} obstacle crosses the line segment between the initial position and goal position.

12.1.3 Gap Navigation Trees

Optimal navigation can be performed with only minimal sensing information and no metric measurements. The representation corresponds to collapsed information states.

12.2 Localization and Map Building

This will be more estimation-oriented

12.3 Manipulation with Minimal Information

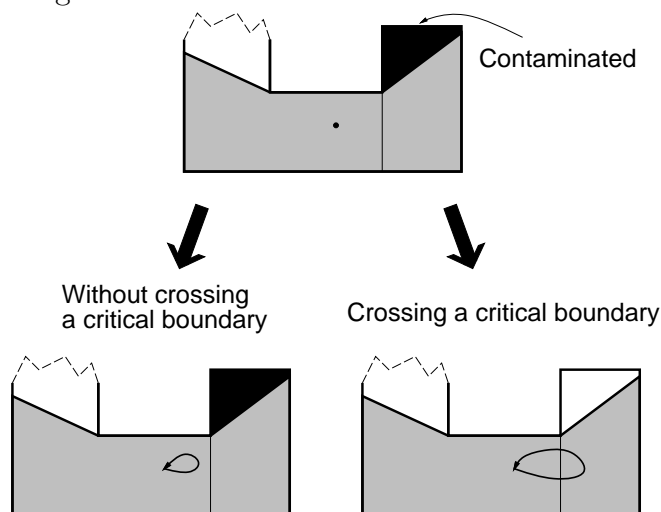
12.4 Visibility-Based Pursuit-Evasion

This section addresses another motion strategy problem that deals with uncertainty in sensing. The model is:

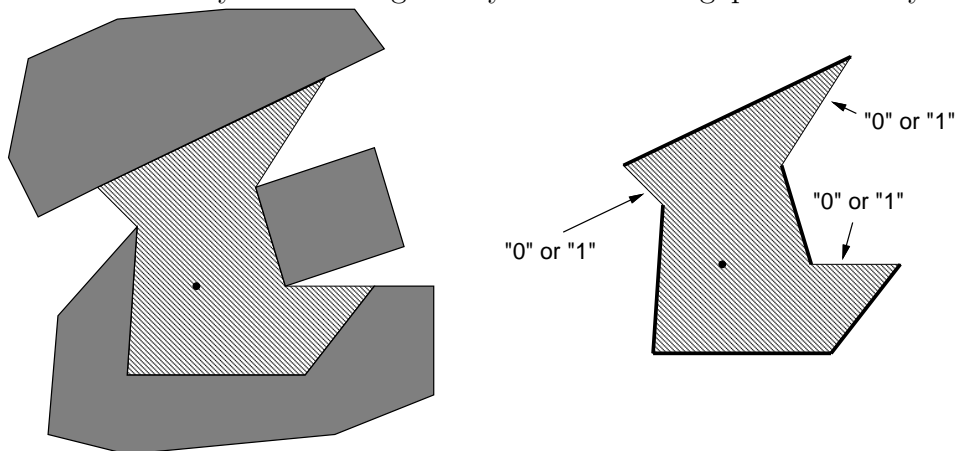
- The robot is a point (the “pursuer”) that moves in a 2D world that is bounded by a simple polygon (it is simply-connected).
- The world contains a point “evader” that can move arbitrarily fast.
- The task is to move the pursuer along a path that guarantees that the evader will eventually be seen using line-of-sight visibility in the 2D world.

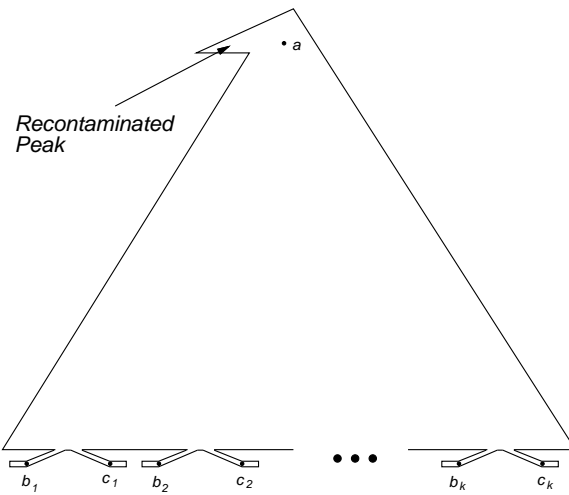
The problem can be formulated as a search in an *information space*, in which each information state is of the form (q, S) . The information state represents the position of the pursuer, q , and the set, S , of places where the evader could be hiding.

The key idea in developing a complete algorithm that will construct a solution if one exists is to partition the world into cells, such that inside of each cell there are no critical changes in information.



A finite graph search can be performed over these cells, cells might generally be visited multiple times. As the pursuer moves from cell to cell, the information state is maintained by maintaining binary labels on the gaps in visibility.





12.5 Preimage Planning

12.6 Algorithms for Solving POMDPs

12.7 Dynamic Programming on Information Spaces

Literature

POMDPs in AI.

Exercises

1. show how different bug algorithms explore some examples
2. make the gap navigation tree for a given example

Part IV

**Planning Under Differential
Constraints**

Chapter 13

Differential Models

Chapter Status



What does this mean? Check
<http://msl.cs.uiuc.edu/planning/status.html>
for information on the latest version.

13.1 Motivation

In the models and methods studied so far, it has been assumed that a path can easily be obtained between any two configurations if there are no collisions. For example, the randomized roadmap approach assumed that two nearby configurations could be connected by a “straight line” in the configuration space. The constraints on the path are global in the sense that the restrictions are on the set of allowable configurations.

For the next few chapters, local constraints will be introduced. One of the simplest examples is a car-like robot. Imagine a trying to automate the motions of a typical automobile that has a limited steering angle. Consider the difficulty of moving a car sideways, while the rear wheels are always pointing forward. It would certainly make parallel parking easy if it was possible to simply turn all four wheels toward the curb. The orientation limits of the wheels, however, prohibit this motion. At any configuration, there are constraints on the velocity of the car. In other words, it is permitted only to move along certain directions to ensure that the wheels roll.

Although the motion is constrained in this way, most of us are experienced with making very complex driving maneuvers to parallel park a car. We would generally like to have algorithms that can maneuver a car-like robot and a variety of other nonholonomic systems while avoiding collisions. This will be the subject of nonholonomic planning.

13.2 Representing Differential Constraints

Implicit velocity constraints

Suppose that X represents an n -dimensional manifold that serves as the *state space*. Let $x \in X$ represent a state. It will often be the case that $X = \mathcal{C}$; however, a state could include additional information. It will be assumed that X is differentiable at every point. To enable this formally, one must generally characterize the X by using multiple coordinate systems, each of which covers a subset of X [719]. We avoid these technicalities in the concepts that follow because they are not critical for understanding the material.

Consider a moving point, $x \in X$. Let \dot{x} denote the *velocity vector*,

$$\dot{x} = \begin{bmatrix} \frac{dx_1}{dt} & \frac{dx_2}{dt} & \cdots & \frac{dx_n}{dt} \end{bmatrix}.$$

Let \dot{x}_i denote dx_i/dt . At most places in this chapter where differentiation occurs, it can be imagined that $X = \mathbb{R}^n$. Recall that any manifold of interest can be considered as a rectangular region in \mathbb{R}^n with identification of some boundaries. Multiple coordinate systems are generally used to ensure differentiability properties across these identifications. Imagining that $X = \mathbb{R}^n$ will be reasonable except at the identification points. For example, if $X = \mathbb{R}^2 \times S^1$, then special care must be given if $\theta = 0$. Motions in the negative θ direction will actually cause θ to increase because of the identification.

Suppose that a classical path planning problem has been defined, resulting in $X = \mathcal{C}$, and that a collision-free path, τ has been computed. Recall that τ was defined as $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$. Although it did not matter before, suppose now that $[0, 1]$ represents an interval of time. At time $t = 0$ the state is $x = q_{init}$, and at time $t = 1$, the state is $x = q_{goal}$. The velocity vector is $\dot{x} = d\tau/dt$.

Up to now, there have been no constraints placed on \dot{x} , which means that any velocity vector is possible. Suppose that the velocity magnitude is bounded, $\|\dot{x}\| \leq 1$. Does this make the classical path planning problem more difficult? It does not because any path $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$ can be converted into another path, τ' which satisfies the bound by lengthening the time interval. For example, suppose s denotes the maximum speed (velocity magnitude) along τ . A new path, $\tau' : [0, s] \rightarrow \mathcal{C}_{free}$, can be defined by $\tau'(t) = \tau(t/s)$. For τ' , the velocity will be bounded by one for all time.

Suppose now that a constraint such as $\dot{x}_1 \leq 0$ is added. This implies that for any path, the variable x_1 must be monotonically nonincreasing. For example, consider path planning for a rigid robot in the plane, yielding $X = \mathbb{R}^2 \times S^1$. Suppose that constraint $\dot{\theta} \leq 0$ is imposed. This implies that the robot is only capable of clockwise rotations!

In general, we allow constraints of the implicit form $h_i(\dot{x}, x) = 0$ to be imposed. Thus, the constrained velocity can depend on the state, x . Inequality constraints

of the form $h_i(\dot{x}, x) < 0$ and $h_i(\dot{x}, x) \leq 0$ are also permitted. Each constraint restricts the set of allowable velocities at any state $x \in X$.

The state transition equation

Although the implicit constraints are general, it is often difficult to work directly with them. A similar difficulty exists with plotting solutions to an implicit function of the form $f(x, y) = 0$, in comparison to plotting the function $y = f(x)$. It turns out for our problem that the implicit constraints can be converted into a convenient form if it is possible to solve for \dot{x} .¹ This will yield a direct expression for the set of allowable velocities.

For example, suppose $X = \mathbb{R}^2 \times S^1$ and let (x, y, θ) denote a state. Consider the constraints $2\dot{x} - y = 0$ and $\dot{\theta} - 1 \leq 0$.² By simple manipulation, we can write $\dot{x} = \frac{1}{2}y$. What should be done with \dot{y} and $\dot{\theta}$? It turns out that new variables need to be introduced to parameterize the set of solutions. This occurs because the set of implicit equations is generally underconstrained (i.e., there is an infinite number of solutions). By introducing $u_1 \in \mathbb{R}$ and $u_2 \in \mathbb{R}$, we can write $\dot{y} = u_1$ and $\dot{\theta} = u_2$ such that $u_2 \leq 1$. The restriction on u_2 comes from the implicit equation $\dot{\theta} - 1 \leq 0$. Note that there is no restriction on u_1 .

By solving for \dot{x} and introducing extra variables, the resulting form can be considered as a control system representation in which the extra variables represent *inputs*. The input is selected by the user, and could correspond, for example, to the steering angle of a car. Suppose f is a vector-valued function, $f : X \times U \rightarrow \mathbb{R}^n$, in which X is an n -dimensional state space, and U is an m -dimensional *input space*.

The *state transition equation* indicates how the state will change over time, given a current state and current input.

$$\dot{x} = f(x, u). \quad (13.1)$$

For a given state, $x \in X$ and a given input $u \in U$, the state transition equation yields a velocity. Simple examples of the state transition equation will be given in Section 13.3.

Two different representations of differential constraints have been introduced. The implicit form is the most general; however, it is difficult to use in many cases. The state transition equation represents a parametric form that directly characterizes the set of allowable velocities at every point in X . The parametric form is also useful for numerical integration, which enables the construction of an incremental simulator.

¹Jacobian-based conditions for this are given by the implicit function theorem in calculus.

²Be careful of notation collision. A general state vector is denoted as x ; however for some particular instances, we also use the standard (x, y) to denote a point in the plane.

An Incremental Simulator

By performing integration over time, the state transition equation can be used to determine the state after some fixed amount of time, Δt has passed. For example, if we know $x(t)$ and inputs $u(t')$ over the interval $t' \in [t, t + \Delta t]$, then the state, $x(t + \Delta t)$ can be determined as

$$x(t + \Delta t) = x(t) + \int_t^{t+\Delta t} f(x(t'), u(t')) dt'$$

The integral above cannot be evaluated directly because $x(t')$ appears in the integrand, but is unknown for time $t' > t$.

Several numerical techniques exist for numerically approximating the solution. Using the fact that

$$f(x, u) = \dot{x} = \frac{dx}{dt} \approx \frac{\Delta x}{\Delta t} = \frac{x(t + \Delta t) - x(t)}{\Delta t},$$

one can solve for $x(t + \Delta t)$ to yield the classic Euler integration method,

$$x(t + \Delta t) \approx x(t) + \Delta t f(x(t), u(t)).$$

For many applications, too much numerical error introduced by Euler integration. Runge-Kutta integration provides an improvement that is based on higher-order Taylor series expansion of the solution. One useful form of Runge-Kutta integration is the fourth-order approximation,

$$x(t + \Delta t) \approx x(t) + \frac{\Delta t}{6}(w_1 + 2w_2 + 2w_3 + w_4),$$

in which

$$w_1 = f(x(t), u(t)),$$

$$w_2 = f\left(x(t) + \frac{\Delta t}{2} w_1, u(t)\right),$$

$$w_3 = f\left(x(t) + \frac{\Delta t}{2} w_2, u(t)\right),$$

and

$$w_4 = f(x(t) + \Delta t w_3, u(t)).$$

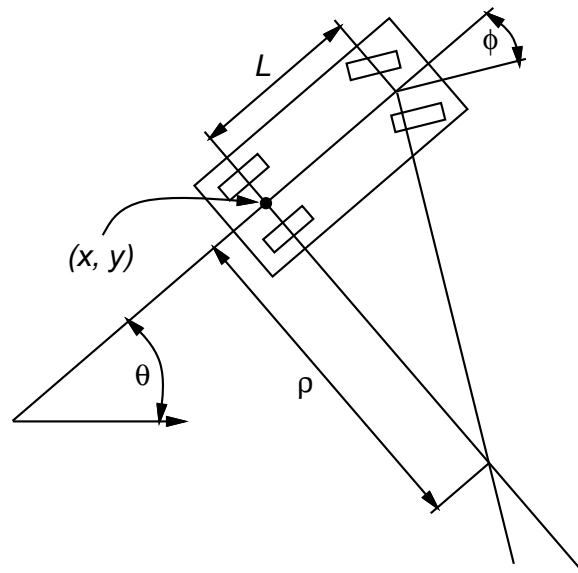
For some problems, a state transition equation might not be available; however, it is still possible to compute any future state, given a current state and an input. This might occur, for example, in a complex software system that simulates the dynamics of a automobile, or a collection of parts that bounce around on a table. In this situation, we simply define the existence of an *incremental simulator*, which serves as a “black box” that produces a future state, given any current state and input. Euler and Runge-Kutta integration may be viewed as techniques that convert a state transition equation into an incremental simulator.

13.3 Kinematics for Wheeled Systems

Several interesting state transition equations can be defined to model the motions of objects that move by rolling wheels. For all of these examples, the state space, X , is equivalent to the configuration space, \mathcal{C} .

13.3.1 A Simple Car

A simple example is the car-like robot. It is assumed that the car can translate and rotate, resulting in $\mathcal{C} = \mathbb{R}^2 \times S^1$. Assume that the state space is defined as $X = \mathcal{C}$. For convenience, let each state be denoted by (x, y, θ) . Let s and ϕ denote two scalar inputs, which represent the speed of the car and the steering angle, respectively. The picture below indicates several parameters associated with the car.



The distance between the front and rear axles is represented as L . The steering angle is denoted by ϕ . The configuration is given by (x, y, θ) . When the steering angle is ϕ , the car will roll in a circular motion, in which the radius of the circle is ρ . Note that ρ can be determined from the intersection of the two axes as shown (the angle between these axes is ϕ).

The task is to represent the motion of the car as a set of equations of the form

$$\dot{x} = f_1(x, y, \theta, s, \phi)$$

$$\dot{y} = f_2(x, y, \theta, s, \phi)$$

$$\dot{\theta} = f_3(x, y, \theta, s, \phi).$$

In a small time interval, the car must move in the direction that the rear wheels are pointing. This implies that $\frac{dy}{dx} = \tan \theta$. Since $\frac{dy}{dx} = \frac{\dot{y}}{\dot{x}}$ and $\tan \theta = \frac{\sin \theta}{\cos \theta}$, this motion constraint can be written as an implicit constraint:

$$-\dot{x} \sin \theta + \dot{y} \cos \theta = 0. \quad (13.2)$$

The equation above is satisfied if $\dot{x} = \cos \theta$ and $\dot{y} = \sin \theta$. Furthermore, any scalar multiple of this solution is also a solution, which corresponds directly to the speed of the car. Thus, the first two scalar components of the state transition equation are $\dot{x} = s \cos \theta$ and $\dot{y} = s \sin \theta$.

The next task is to derive the equation for $\dot{\theta}$. Let p denote the distance traveled by the car. Then $\dot{p} = s$, which is the speed. As shown in the figure above, ρ represents the radius of a circle that will be traversed by the center of the rear axle, when the steering angle is fixed. Note that $dp = \rho d\theta$. From simple trigonometry, $\rho = \frac{L}{\tan \phi}$, which implies

$$d\theta = \frac{\tan \phi}{L} dp.$$

Dividing by dt and using the fact that $\dot{p} = s$ yields

$$\dot{\theta} = \frac{s}{L} \tan \phi.$$

Thus, the state transition equation for the car-like robot is

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} s \cos \theta \\ s \sin \theta \\ \frac{s}{L} \tan \phi \end{pmatrix}$$

Most vehicles with steering have a limited steering angle, ϕ_{max} such that $0 < \phi_{max} < \frac{\pi}{2}$.

The speed of the car is usually bounded. If there are only two possible speeds (forward or reverse), $s \in \{-1, 1\}$, then the model is referred to as the *Reeds-Shepp* car [648, 730]. If the only possible speed is $s = 1$, then the model is referred to as the *Dubins* car [214].

13.3.2 A Continuous-Steering Car

In the previous model, the steering angle, ϕ , was an input, which implies that one can instantaneously move the front wheels. In many applications, this assumption is unrealistic. In the path traced out in the plane by the center of the rear axle of the car, there is a curvature discontinuity will occur when the steering angle is changed discontinuously. To make a car model that only generates smooth paths, the steering angle can be added as a state variable. The input is the angular velocity, ω , of the steering angle.

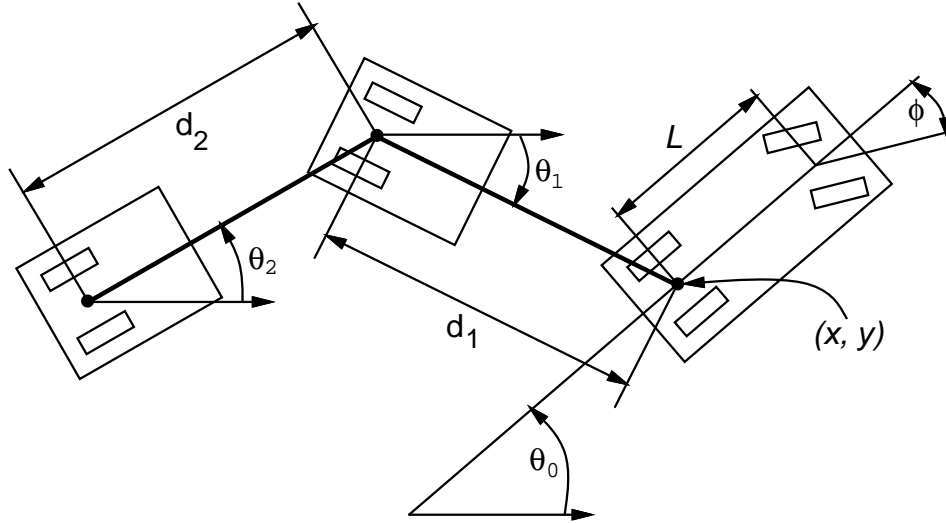
The result is a four-dimensional state space, in which each state is represented as (x, y, ϕ, θ) . This yields the following state transition equation:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\phi} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} s \cos \theta \\ s \sin \theta \\ \omega \\ \frac{s}{L} \tan \phi, \end{pmatrix}$$

in which there are two inputs, s and ω . This model was considered in [673].

13.3.3 A Car Pulling Trailers

The continuous-steering car can be extended to allow one or more single-axle trailers to be pulled. For k trailers, the state is represented as $(x, y, \phi, \theta_0, \theta_1, \dots, \theta_k)$.



The state transition equation is

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\phi} \\ \dot{\theta}_0 \\ \vdots \\ \dot{\theta}_i \\ \vdots \end{pmatrix} = \begin{pmatrix} s \cos \theta \\ s \sin \theta \\ \omega \\ \frac{s}{L} \tan \phi \\ \vdots \\ \frac{s}{d_i} \left(\prod_{j=1}^{i-1} \cos(\theta_{j-1} - \theta_j) \right) \sin(\theta_{i-1} - \theta_i) \\ \vdots \end{pmatrix},$$

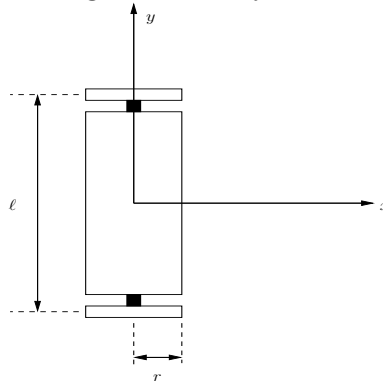
in which θ_0 is the orientation of the car, θ_i is the orientation of the i^{th} trailer, and d_i is the distance from the i^{th} trailer wheel axle to the hitch point. This model was considered in [573].

13.3.4 A Differential Drive

The differential drive model is very common in mobile robotics. It consists of a single axle, which connects two independently-controlled wheels. Each wheel is driven by its own motor, and it free to rotate without affecting the other wheel. Each state is represented as (x, y, θ) . The state transition equation is

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \frac{r}{2}(u_l + u_r) \cos \theta \\ \frac{r}{2}(u_l + u_r) \sin \theta \\ \frac{r}{\ell}(u_r - u_l) \end{pmatrix}, \quad (13.3)$$

in which r is the wheel radius, ℓ is the axle length, u_r is the angular velocity of the right wheel, and u_l is the angular velocity of the left wheel.



If $u_l = u_r = 1$, the differential drive rolls forward. If $u_l = u_r = -1$, the differential drive rolls in the opposite direction. If $u_l = -u_r$, the differential drive performs a rotation.

13.4 Rigid-Body Dynamics

so far, this is only a point mass...

For problems that involve dynamics, constraints will exist on accelerations, in addition to velocities and configurations. Accelerations may appear problematic because they represent second-order derivatives, which cannot appear in the state transition equation (13.1). To overcome this problem a state space will be defined that allows the equations of motion to be converted into the form $\dot{x} = f(x, u)$. Usually, the dimension of this state space is twice the dimension of the configuration space.

The state space For a broad class of problems, equations of motion that involve dynamics can be expressed as $\ddot{q} = g(\dot{q}, q)$, for some measurable function g . Suppose a problem is defined on an n -dimensional configuration space, \mathcal{C} . Define a $2n$ -dimensional state vector $x = [q \ \dot{q}]$. In other words, x represents both configuration and velocity,

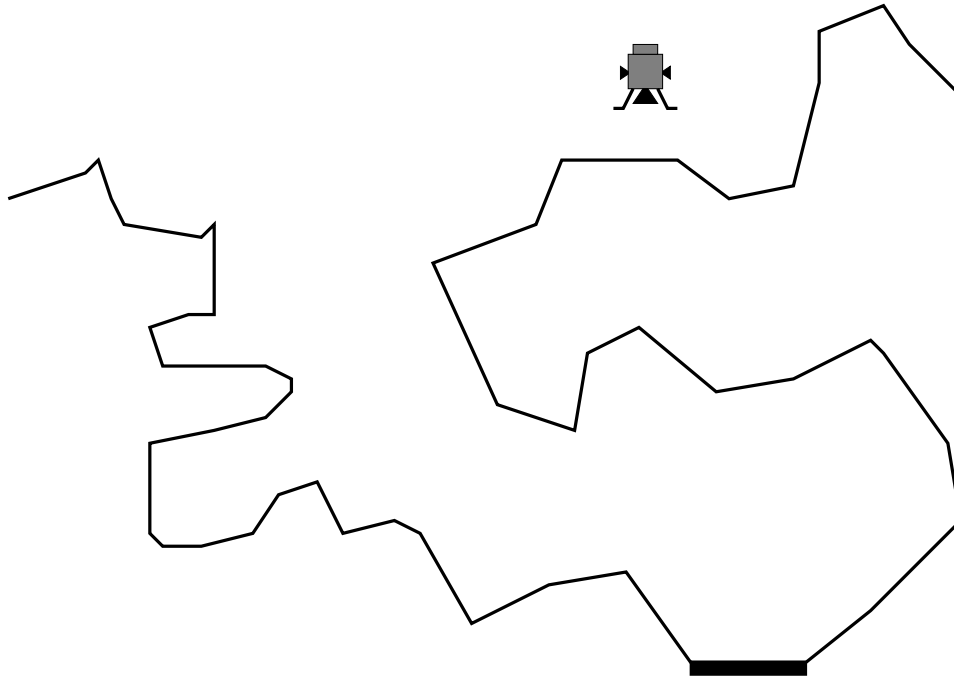
$$x = [q_1 \ q_2 \ \cdots \ q_n \ \dot{q}_1 \ \dot{q}_2 \ \cdots \ \dot{q}_n].$$

Let X denote the $2n$ -dimensional state space, which is the set of all state vectors.

The goal is to construct a state transition equation of the form $\dot{x} = f(x, u)$. Given the definition of the state vector, note that $\dot{x}_i = x_{n+i}$ if $i \leq n$. This immediately defines half of the components of the state transition equation. The other half is defined using $\ddot{q} = g(\dot{q}, q)$. This is obtained by simply substituting each of the \ddot{q} , \dot{q} , and q variables by their state space equivalents.

Example: Lunar lander A simple example that illustrates the concepts is given. The same principles can be applied to obtain equations of motion of the

form $\dot{x} = f(x, y)$ for state spaces that represent the configuration and velocity of rigid and articulated bodies.



The lander is modeled as a point with mass, m , in a 2D world. It is not allowed to rotate, implying that $\mathcal{C} = \mathbb{R}^2$. There are three thrusters on the lander: Thruster One (right side), Thruster Two (bottom), and Thruster Three (left side). The activation of each thruster is considered as a binary switch. Let u_i denote a binary-valued action that can activate the i^{th} thruster. If $u_i = 1$, the thruster fires, if $u_i = 0$, then the thruster is dormant. Each of the two lateral thrusters provides a force F_s when activated. The upward thruster, mounted to the bottom of the lander, provides a force F_u when activated. Let g denote the acceleration of gravity.

From simple Newtonian mechanics, $\sum F = ma$, in which $\sum F$ denotes the vector sum of the forces, m denotes the mass of the lander, and a denote the acceleration, \ddot{q} . The q_1 -component (x-direction) yields

$$m\ddot{q}_1 = u_1F_s - u_3F_s,$$

and the q_2 -component (y-direction) yields

$$m\ddot{q}_2 = u_2F_u - mg$$

The constraints above can be written in the form $f(q, \dot{q}, \ddot{q}) = 0$ (actually, the equations are simple enough to obtain $f(\ddot{q}) = 0$).

The lunar lander model can be transformed into a four-dimensional state space in which $x = [q_1 \ q_2 \ \dot{q}_1 \ \dot{q}_2]$. By replacing \ddot{q}_1 and \ddot{q}_2 with \dot{x}_3 and \dot{x}_4 , respectively, the Newtonian equations of motion can be written as

$$\dot{x}_3 = \frac{F_s}{m}(u_1 - u_3)$$

$$\dot{x}_4 = \frac{u_2 F_u}{m} - g$$

Since $\dot{x}_1 = x_3$ and $\dot{x}_2 = x_4$, the state transition equation becomes

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{pmatrix} = \begin{pmatrix} x_3 \\ x_4 \\ \frac{F_s}{m}(u_1 - u_3) \\ \frac{u_2 F_u}{m} - g \end{pmatrix},$$

which is in the desired form $\dot{x} = f(x, u)$.

13.5 Multiple-Body Dynamics

13.6 More Examples

This section includes other examples of state transition equations.

The nonholonomic integrator Here is a simple nonholonomic system that might be useful for experimentation. Let $X = \mathbb{R}^3$, and let the set of inputs, $U = \mathbb{R}^2$. The state transition equation for the nonholonomic integrator is

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{pmatrix} = \begin{pmatrix} u_1 \\ u_2 \\ x_1 u_2 - x_2 u_1 \end{pmatrix}.$$

Chapter 14

Nonholonomic System Theory

Chapter Status



What does this mean? Check

<http://msl.cs.uiuc.edu/planning/status.html>
for information on the latest version.

This chapter deals with the analysis of problems that involve differential constraints. One fundamental result is the Frobenius theorem, which allows one to determine whether the state transition equation represents a system is actually nonholonomic. In some cases, it may be possible to integrate the state transition equation, resulting in a problem that can be described without differential models. Another result is Chow's theorem, which indicates whether a system is controllable. Intuitively, this means that the differential constraints can be completely overcome by generating arbitrarily short maneuvers. The car-like robot enjoys the controllability property, which enables it to move itself sideways by performing parallel parking maneuvers.

14.1 Vector Fields and Distributions

A special form of the state transition equation Most of the concepts in this chapter are developed under the assumption that the state transition equation, $\dot{x} = f(x, u)$ has the following form:

$$\dot{x} = \alpha^1(x)u_1 + \alpha^2(x)u_2 + \cdots + \alpha^m(x)u_m, \quad (14.1)$$

in which each $\alpha^i(x)$ is a vector-valued function of x , and m is the dimension of U (or the number of inputs). The α^i functions can also be arranged in an $n \times m$ matrix,

$$A(x) = [\alpha^1(x) \quad \alpha^2(x) \quad \cdots \quad \alpha^m(x)]. \quad (14.2)$$

It will usually be assumed that $m < n$.

In this case, the state transition equation can be expressed as

$$\dot{x} = A(x)u. \quad (14.3)$$

For the rest of the chapter, it will be assumed that the matrix $A(x)$ is nonsingular. In other words, the rows of $A(x)$ are linearly independent for all x . To determine whether $A(x)$ is nonsingular, one must find at least one $m \times m$ cofactor (or signed submatrix) of $A(x)$ which has a nonzero determinant.

Vector fields A vector field, \vec{V} , on a manifold X , is a function that associates with each $x \in X$, a vector, $\vec{V}(x)$. The *velocity field* is a special vector field that will be used extensively. Each vector $\vec{V}(x)$ in a velocity field represents the infinitesimal change in state with respect to time,

$$\dot{x} = \left[\frac{dx_1}{dt} \quad \frac{dx_2}{dt} \quad \cdots \quad \frac{dx_n}{dt} \right], \quad (14.4)$$

evaluated at the point $x \in X$.

Note that for a fixed u , any state transition equation, $\dot{x} = f(x, u)$ defines a vector field because \dot{x} is expressed as a function of x .

Distributions Each input $u \in U$ can be used to define a vector field. It will be convenient to define the set of all vector fields that can be generated using inputs. Assume that a state transition equation of the form in (14.1) is given for a state space X , and an input space $U = \mathbb{R}^m$. The set of all vector fields that can be generated using inputs $u \in U$ is called the *distribution*, and is denoted by $\Delta(X)$ or Δ .

The distribution can be considered as a vector space. Note that each α^i can be interpreted as a vector field. Any vector field in Δ can be expressed as a linear combination of the α^i functions, which serve as a basis of the vector space. Consider the effect of inputs of the form

$$[1 \ 0 \ 0 \ \cdots \ 0 \ 0 \ 0 \ \cdots \ 0]$$

$$[0 \ 1 \ 0 \ \cdots \ 0 \ 0 \ 0 \ \cdots \ 0]$$

$$\vdots$$

$$[0 \ 0 \ 0 \ \cdots \ 0 \ 1 \ 0 \ \cdots \ 0]$$

$$\vdots$$

$$[0 \ 0 \ 0 \ \cdots \ 0 \ 0 \ 0 \ \cdots \ 1].$$

If $u_i = 1$, and $u_j = 0$ for $j \neq i$, then the state transition equation yields $\dot{x} = \alpha^i(x)$. Thus, each input in this form can be used to generate a basis vector field. The dimension of the distribution the number of vector fields in its basis (in other words, the maximum number of linearly-independent vector fields that can be generated).

In terms of basis vector fields, a distribution is expressed as

$$\Delta = \text{span}\{\alpha^1(x), \alpha^2(x), \dots, \alpha^n(x)\} \quad (14.5)$$

Example: Differential Drive The state transition equation (13.3) for the differential drive can be expressed in the form of (14.1) as follows:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} (\frac{r}{2} \cos \theta)u_l + (\frac{r}{2} \cos \theta)u_r \\ (\frac{r}{2} \sin \theta)u_l + (\frac{r}{2} \sin \theta)u_r \\ (-\frac{r}{\ell})u_l + (\frac{r}{\ell})u_r \end{pmatrix} = \begin{pmatrix} \frac{r}{2} \cos \theta & \frac{r}{2} \cos \theta \\ \frac{r}{2} \sin \theta & \frac{r}{2} \sin \theta \\ -\frac{r}{\ell} & \frac{r}{\ell} \end{pmatrix} \begin{pmatrix} u_l \\ u_r \end{pmatrix} = A(x, y, \theta) \begin{pmatrix} u_l \\ u_r \end{pmatrix}. \quad (14.6)$$

The matrix $A(x, y, \theta)$ is nonsingular because any of the three 2×2 cofactors of $A(x, y, \theta)$ has a nonzero determinant for all states.

To simplify the characterization of the distribution, a linear transformation will be performed on the inputs. Let $u_1 = u_l + u_r$ and $u_2 = u_r - u_l$. Intuitively, u_1 means “go straight” and u_2 means “rotate”. Note that the original u_l and u_r can be easily recovered from u_1 and u_2 . For additional simplicity, assume that $\ell = 2$ and $r = 2$. The state transition equation becomes

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos \theta & 0 \\ \sin \theta & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}. \quad (14.7)$$

Using input $u = [1 \ 0]$, the vector field $\vec{V} = [\cos \theta \ \sin \theta \ 0]$ is obtained. Using $u = [0 \ 1]$, the vector field $\vec{W} = [0 \ 0 \ 1]$ is obtained. Any other vector field that can be generated using inputs can be constructed as a linear combination of \vec{V} and \vec{W} . The distribution Δ has dimension two, and is expressed as $\text{span}\{\vec{V}, \vec{W}\}$.

14.2 The Lie Bracket

The Lie bracket attempts to generate velocities that are not directly permitted by the state transition equation. For the car-like robot, it will produce a vector field that can move the car sideways (it is achieved through combinations of vector fields, and therefore does not violate the nonholonomic constraint). This operation

is called the *Lie bracket* (pronounced as “Lee”), and for given vector fields \vec{V} and \vec{W} , it is denoted by $[\vec{V}, \vec{W}]$. The Lie bracket is computed by

$$[\vec{V}, \vec{W}] = D\vec{W} \cdot \vec{V} - D\vec{V} \cdot \vec{W} \quad (14.8)$$

in which \cdot denotes a matrix-vector multiplication,

$$D\vec{V} = \begin{pmatrix} \frac{\partial V_1}{\partial x_1} & \frac{\partial V_1}{\partial x_2} & \cdots & \frac{\partial V_1}{\partial x_n} \\ \frac{\partial V_2}{\partial x_1} & \frac{\partial V_2}{\partial x_2} & \cdots & \frac{\partial V_2}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial V_n}{\partial x_1} & \frac{\partial V_n}{\partial x_2} & \cdots & \frac{\partial V_n}{\partial x_n} \end{pmatrix}, \quad (14.9)$$

and

$$D\vec{W} = \begin{pmatrix} \frac{\partial W_1}{\partial x_1} & \frac{\partial W_1}{\partial x_2} & \cdots & \frac{\partial W_1}{\partial x_n} \\ \frac{\partial W_2}{\partial x_1} & \frac{\partial W_2}{\partial x_2} & \cdots & \frac{\partial W_2}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial W_n}{\partial x_1} & \frac{\partial W_n}{\partial x_2} & \cdots & \frac{\partial W_n}{\partial x_n} \end{pmatrix}. \quad (14.10)$$

In the expressions above, V_i and W_i denote the i^{th} components of \vec{V} and \vec{W} , respectively.

To compute the Lie bracket it is often convenient to directly use the expression for each component of the new vector field. This is obtained by performing the multiplication indicated above. The i^{th} component of the Lie bracket is given by

$$\sum_{j=1}^n \left(V_j \frac{\partial W_i}{\partial x_j} - W_j \frac{\partial V_i}{\partial x_j} \right). \quad (14.11)$$

Two well-known properties of the Lie bracket are:

1. (skew-symmetry) $[\vec{V}, \vec{W}] = -[\vec{W}, \vec{V}]$ for any two vector fields, \vec{V} and \vec{W}
2. (Jacobi identity) $[[\vec{V}, \vec{W}], \vec{U}] + [[\vec{W}, \vec{U}], \vec{V}] + [[\vec{U}, \vec{V}], \vec{W}] = 0$

It can be shown using Taylor series expansions that the Lie bracket $[\vec{V}, \vec{W}]$ can be approximated by performing a sequence of four integrations. From a point, $x \in X$, the Lie bracket yields a motion in the direction obtained after performing

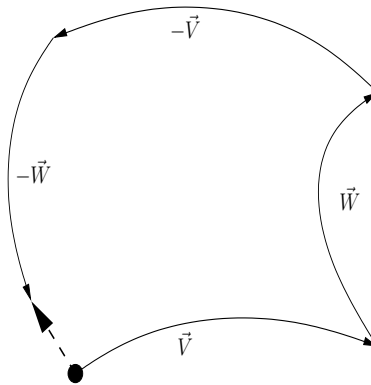


Figure 14.1: The velocity obtained by the Lie bracket can be approximated as a sequence of four motions.

1. Motion along \vec{V} for time $\Delta t/4$
2. Motion along \vec{W} for time $\Delta t/4$
3. Motion along $-\vec{V}$ for time $\Delta t/4$
4. Motion along $-\vec{W}$ for time $\Delta t/4$

The direction from x to the resulting state after performing the four motions represents the direction given by the Lie bracket, as shown in Figure 14.1 by the dashed arrow.

14.3 Integrability and Controllability

The Lie bracket can be used to generate vector fields that potentially lie outside of Δ . There are two theorems that express useful system properties that can be inferred using the vector fields generated by Lie brackets.

The Control Lie Algebra (CLA) For a given state transition equation of the form (14.1), consider the set of all vector fields that can be generated by taking Lie brackets, $[\alpha^i(x), \alpha^j(x)]$, of vector fields $\alpha^i(x)$ and $\alpha^j(x)$ for $i \neq j$. Next, consider taking Lie brackets of the new vector fields with each other, and with the original vector fields. This process can be repeated indefinitely by iteratively applying the Lie bracket operations to new vector fields. The resulting set of vector fields can be considered as a kind of algebraic closure with respect to the Lie bracket operation. Let the *control Lie algebra*, $CLA(\Delta)$, denote the set of all vector fields that are obtained by this process.

In general, $CLA(\Delta)$ can be considered as a vector space, in which the basis elements are the vector fields $\alpha^1(x), \dots, \alpha^m(x)$, and all new, linearly-independent vector fields that were generated from the Lie bracket operations.

Finding the basis of $CLA(\Delta)$ is generally a tedious process. There are several systematic approaches for generating the basis, of which one of the most common is called the Phillip-Hall basis. This basis automatically eliminates any vector fields from the Lie bracket calculations that could be obtained by skew symmetry or the Jacobi identity.

Each Lie bracket has the opportunity to generate a vector field that is linearly-independent; however, it is not guaranteed to generate one. In fact, all Lie bracket operations may fail to generate a vector field that is independent of the original vector fields. Consider for example, the case in which the original vector fields, α^i , are all constant. All Lie brackets will be zero.

Integrability In some cases, it is possible that the differential constraints are integrable. This implies that it can be expressed purely as a function of x and u , and not of \dot{x} . In the case of an integrable state transition equation, the motions are actually restricted to a lower-dimensional subset of X , which is a global constraint as opposed to a local constraint.

As a simple example, suppose that $X = \mathbb{R}^2$, and a state transition equation is:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} -y \\ x \end{pmatrix} u_1 \quad (14.12)$$

Suppose that an initial state $(a, 0)$ is given for some $a \in (0, \infty)$. By selecting an input $u_1 \in (\infty, 0)$, integration of the state transition equation over time will yield a counterclockwise path along a circle of radius a , centered at the origin. If $u_1 < 0$, then a clockwise motion along the circle is generated. Note that starting from any initial state, there is no way for the state, $(x(t), y(t))$ to leave a circle centered at the origin. Thus, the state transition equation simply represents a global constraint that the set of states is constrained to a circle.

In general, it is very difficult to determine whether a given state transition can be integrated to remove all differential constraints. The Frobenius theorem gives an interesting condition that may be applied to determine whether the state transition equation is integrable.

Theorem 1 (Frobenius) *The state transition equation is integrable if and only if all vector fields that can be obtained by Lie bracket operations are contained in Δ .*

Intuitively, if the Lie bracket operation is unable to produce any new (linearly-independent) vector fields that lie outside of Δ , then the state transition equation can be integrated. Thus, the equation is not needed, and the problem can be reformulated without using \dot{x} . This is, however, a theoretical result; it may be a difficult or impossible task in general to integrate the state transition equation in practice.

The Frobenius theorem can also be expressed in terms of dimensions. If $\dim(CLA(\Delta)) = \dim(\Delta)$, then the state transition equation is integrable. Note that the dimension of $CLA(\Delta)$ can never be greater than n .

If the state transition equation is not integrable, then it is called *nonholonomic*. These equations are of greatest interest.

Controllability In addition to integrability, another important property of a state transition equation is controllability. Intuitively, controllability implies that the robot is able to overcome its differential constraints by using Lie brackets to compose new motions. The controllability concepts assume that there are no obstacles.

Two kinds of controllability will be considered. A point, x' , is *reachable* from x , if there exists an input that can be applied to bring the state from x to x' . Let $R(x)$ denote the set of all points reachable from x . A system is *locally controllable* if for all $x \in X$, $R(x)$ contains an open set that contains x . This implies that any state can be reached from any other state.

Let $R(x, \Delta t)$ denote the set of all points reachable in time Δt . A system is *small-time controllable* if for all $x \in X$ and any Δt , then $R(x, \Delta t)$ contains an open set that contains x .

The Dubins car is an example of a system that is locally controllable, but not small-time controllable. If there are no obstacles, it is possible to bring the car to any desired configuration from any initial configuration. This implies that the car is locally controllable. Suppose one would like to move the car to a position that would be obtained by the Reeds-Shepp car by moving a small amount in reverse. Because the Dubins car must drive forward to reach this configuration, it could require time larger than some small Δt . Hence, the Dubins care is not small-time controllable.

However, a substantial amount of time might be required to drive the care
Chow's theorem is used to determine small-time controllability.

Theorem 2 (Chow) *A system is small-time controllable if and only if the dimension of $CLA(\Delta)$ is n , the dimension of X .*

Example of integrability and controllability As an example of controllability and integrability, recall the differential drive model. From the differential drive example in Section 14.1, the original vector fields are $\alpha^1(x) = [\cos \theta \ \sin \theta \ 0]$ and $\alpha^2(x) = [0 \ 0 \ 1]$.

Let \vec{V} denote $\alpha^1(x)$, and let \vec{W} denote $\alpha^2(x)$. To determine integrability and controllability, the first step is to compute the Lie bracket, $\vec{Z} = [\vec{V}, \vec{W}]$. The components are

$$Z_1 = V_1 \frac{\partial W_1}{\partial x} - W_1 \frac{\partial V_1}{\partial x} + V_2 \frac{\partial W_1}{\partial y} - W_2 \frac{\partial V_1}{\partial y} + V_3 \frac{\partial W_1}{\partial \theta} - W_3 \frac{\partial V_1}{\partial \theta} = \sin \theta, \quad (14.13)$$

$$Z_2 = V_1 \frac{\partial W_2}{\partial x} - W_1 \frac{\partial V_2}{\partial x} + V_2 \frac{\partial W_2}{\partial y} - W_2 \frac{\partial V_2}{\partial y} + V_3 \frac{\partial W_2}{\partial \theta} - W_3 \frac{\partial V_2}{\partial \theta} = -\cos \theta, \quad (14.14)$$

and

$$Z_3 = V_1 \frac{\partial Y_3}{\partial x} - W_1 \frac{\partial V_2}{\partial x} + V_2 \frac{\partial Y_3}{\partial y} - W_2 \frac{\partial V_2}{\partial y} + V_3 \frac{\partial Y_3}{\partial \theta} - W_3 \frac{\partial V_2}{\partial \theta} = 0. \quad (14.15)$$

The resulting vector field is $\vec{Z} = [\sin \theta \quad -\cos \theta \quad 0]$.

We immediately observe that \vec{Z} is linear independent from \vec{V} and \vec{W} . This can be seen by noting that the determinant of the matrix


$$\begin{pmatrix} \cos \theta & \sin \theta & 0 \\ 0 & 0 & 1 \\ \sin \theta & -\cos \theta & 0 \end{pmatrix} \quad (14.16)$$

is nonzero for all (x, y, θ) . This implies that the dimension of $CLA(\Delta) = 3$. Using the Frobenius theorem, it can be inferred that the state transition equation is not integrable, and the system is nonholonomic. From Chow's theorem, it is known that the system is small-time controllable.

A nice interpretation of the result can be constructed by using the motions depicted in Figure 14.1. Suppose the initial state is $(0, 0, 0)$. The Lie bracket at this state is $[0 \quad -1 \quad 0]$, which can be constructed by four motions: 1) apply u_1 , which translates the drive along the X axis; 2) apply u_2 , which rotates the drive counterclockwise; 3) apply $-u_1$, which translates the drive back towards the Y axis, but the motion is at a downward angle due to the rotation; 4) apply $-u_2$, which rotates the drive back into its original orientation. The net effect of these four motions moves the differential drive downward along the Y axis, which is precisely the direction $[0 \quad -1 \quad 0]$ given by the Lie bracket!

Chapter 15

Planning Under Differential Constraints

| Chapter Status | |
|---|---|
|  | What does this mean? Check http://msl.cs.uiuc.edu/planning/status.html for information on the latest version. |



This chapter presents several alternative planning methods. For each method, it is assumed that a state transition equation or incremental simulator has been defined over a state space. The state could represent configuration or both configuration and velocity.

15.1 Problem formulations

Nonholonomic planning

 Kinodynamic planning

 Brief summary of complexity analysis

15.2 Steering Methods

CBHD formulas, flat systems, etc.

15.2.1 Geodesic curve families

Need to include Balkcom-Mason curves, Reeds-Shepp, Dubins, etc.

A common theme for many planning approaches is to divide the problem into two phases. In the first phase, a holonomic planning method is used by producing a collision-free path that ignores the nonholonomic constraints. In the second phase, an iterative method attempts to replace portions of the holonomic

path with portions that satisfy the nonholonomic constraints, yet still avoid obstacles. In general, this will lead to an incomplete algorithm because there is no guarantee that the original path provides a correct starting point for obtaining a nonholonomic solution. However, it typically leads to a fast planning algorithm.

In this section, we describe this approach for the case of a car-like robot. Assume that a fast holonomic planning method has been selected for the first phase. Suppose that a path, $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$ has been computed. The path can be iteratively improved as follows. Randomly select two real numbers $\alpha_1 \in [0, 1]$ and $\alpha_2 \in [0, 1]$. Assuming $\alpha_2 > \alpha_1$ (if not, then swap them), attempt to replace the portion of τ from $\tau(\alpha_1)$ to $\tau(\alpha_2)$ with a path segment that satisfies the nonholonomic constraints. This implies that τ is broken into three segments, $\tau_1 : [0, \alpha_1] \rightarrow \mathcal{C}_{free}$, $\tau_2 : [\alpha_1, \alpha_2] \rightarrow \mathcal{C}_{free}$, and $\tau_3 : [\alpha_2, 1] \rightarrow \mathcal{C}_{free}$. Note that $\tau_1(\alpha_1) = \tau_2(\alpha_1)$ and $\tau_2(\alpha_2) = \tau_3(\alpha_2)$. The portions τ_1 and τ_3 remain fixed, but τ_2 is replaced with a new path, $\tau' : [\alpha_1, \alpha_2] \rightarrow \mathcal{C}_{free}$, that satisfies the nonholonomic constraints. Note that τ' must also avoid collisions, $\tau'(\alpha_1) = \tau_1(\alpha_1)$, and $\tau'(\alpha_2) = \tau_3(\alpha_2)$. This procedure can be iterated multiple times until eventually, the original path is completely transformed into a nonholonomic path. Note that $\alpha_1 = 0$ and $\alpha_2 = 1$ must each have nonzero probability of being chosen in each iteration. In many iterations, the path substitution will fail; in this case, the previous path is retained.

To make this and related approaches succeed, a fast technique is needed that constructs a nonholonomic path between any two configurations. Although this might appear as difficult as the original nonholonomic planning problem, it is assumed that the obstacles are ignored. In general, this is referred to as the *steering problem*, which has received a considerable amount of attention in recent years, particularly for car-like robots that pull trailers. For the case of a simple car-like robot with a limited steering angle, there are some analytical solutions to the problem of finding the shortest path between two configurations. In 1957, Dubins showed that for a car that can only go forward, the optimal path will take one of the six following forms:

$$\{LRL, LSL, LSR, RLR, RSR, RSL\}.$$

Each sequence of labels indicates the type of path. For example, “LRL” indicates a path that consists of a sharp left turn, immediately followed by a sharp right turn, immediately followed by a sharp left turn. Above, “S” denotes a straight segment. For a given pair of configurations, one can simply attempt to connect them using all six path types. The one with the shortest path length among the six choices is known to be the minimum-length path out of all possible paths. This path provides a nice substitution for τ_2 , as described above.

For the case of a car-like robot that can move forward or backwards, Reeds and Shepp showed in 1990 that the optimal path between two configurations will take one of 48 different forms. Although this situation is more complicated, the same general strategy can be applied as for the case of a forward-only car.

15.2.2 Series Methods

15.3 Sampling-Based Planning Methods

Currently much of this section is pasted from a recent paper by Peng Cheng and Steve LaValle

Problem \mathcal{P} is transformed into a multistage decision problem, which is called a *discretized problem* $\tilde{\mathcal{P}}$. At each stage there is a simpler motion planning problem, which is solved by a local planner. After the discretization, we expect that in most cases, an exact solution to \mathcal{P} will no longer exist. Therefore, we assume that when $\tilde{\mathcal{P}}$ is given, a *solution tolerance*, $\epsilon_s \in [0, \infty)$, is specified.

Note that: 1) control set $\tilde{U}(x)$ could be state dependent, which means that sets of available controls for different states are different; 2) since controls are designed by the local planner, control set \tilde{U} also depends on the local planner. For example, if the local planner can only return piecewise constant controls, \tilde{U} is only a subset of the control space. In some sense, the local planner introduces a discretization on the control space of \mathcal{P} .

The discretization process partitions the time line into intervals. Any control $\tilde{u} \in \tilde{U}$ is applied over some time interval. Let $\bar{t} : \tilde{U} \rightarrow [0, \infty)$ give the duration of any $\tilde{u} \in \tilde{U}$. A control $\tilde{u} \in \tilde{U}$ is defined as a piecewise continuous function from $[0, \bar{t}(u)]$ into U ; thus, it is not limited to a constant control. If $\bar{t}(\tilde{u}) = \bar{t}(\tilde{u}')$, for all $\tilde{u}, \tilde{u}' \in \tilde{U}$, it is called a discretization with a *fixed control sampling rate*; otherwise, it is called a discretization with a *varying control sampling rate*. The range of the length of time intervals is denoted by an interval

$$\mathcal{D} = [\inf_{\tilde{u} \in \tilde{U}} \bar{t}(u), \sup_{\tilde{u} \in \tilde{U}} \bar{t}(u)]. \quad (15.1)$$

We assume that $\inf_{\tilde{u} \in \tilde{U}} \bar{t}(\tilde{u}) > 0$ and $\sup_{\tilde{u} \in \tilde{U}} \bar{t}(\tilde{u}) < \infty$. Depending on the problem and discretization, \tilde{U} may or may not be finite.

Classification of discretizations Based on whether the sampling rate is fixed and whether the control set is finite, there are four types of discretized motion planning problems:

- **FF: Fixed control sampling rate, a finite set of controls**

The first type has a finite set of controls. In [54, 203, 343, 463], for every motion planning problem at every stage, the local planner chooses a control in \tilde{U} and applies it on the system for a fixed period of time.

A special case of this type is considered in [203, 212], in which a constant acceleration is applied on the system for a fixed period of time. For general systems, a non-constant control is necessary to maintain the constant acceleration. Thus, a non-trivial local planner needs to be assumed to provide

the constant acceleration. In these planners, along each degree of freedom, there are only a finite number of accelerations at each stage, which makes \tilde{U} finite. The main difference between this case and more general systems is that the reachability graph in this case is always finite because the state space is compact, and the fixed control sampling rate is carefully chosen to ensure that the velocity bound is an integer multiple of the product of the acceleration and the sampling rate.

This case often appears in the definition of the system; however, the control space is usually discretized before a motion planning algorithm is employed. In this paper, we also consider the case in which continuous methods are used to select controls, which results in the FI case.

- **VF: Varying control sampling rate, a finite set of controls**

Problems of this type were considered in [524, 544, 669, 698]. At each stage, a non-trivial local planner drives the system from the current state to a finite number of adjacent neighboring states on a grid, resulting in a finite control set for each state on the grid. However, each control might last for different amount of time and \tilde{U} are state dependent.

- **VI: Varying control sampling rate, an infinite set of controls**

Problems of this type were considered in [164, 261, 381]. At every stage, the local planner may drive the system from the current state to a possibly infinite number of states. Each control designed by the local planner may last for a different amount of time and $\tilde{U}(x)$ might vary from state to state.

15.3.1 An Incremental Search Framework

The reachability graph *Currently much of this is pasted from a recent paper by Peng Cheng and Steve LaValle*

The reachability graph describes the connectivity between reachable states from x_{init} and is fixed for a given $\tilde{\mathcal{P}}$. It is *not* something that is constructed by an algorithm; it simply exists once $\tilde{\mathcal{P}}$ is defined. The reachability graph will serve in this paper as an important frame of reference for comparing the search graph generated by an algorithm.

Before defining the reachability graph, we define the set of *reachable states at stage k* , denoted as R_k , by induction. First, $R_0 = \{x_{init}\}$. At stage k :

$$R_k = \{x \mid x = \tilde{f}(x', u), x' \in R_{k-1}, u \in \tilde{U}_{vf}(x')\}. \quad (15.2)$$

The set of reachable states from x_{init} is

$$\mathcal{R}_\infty = \bigcup_{k=0}^{\infty} R_k. \quad (15.3)$$

For a given $\tilde{\mathcal{P}}$, the *reachability graph*, $\mathcal{G}\langle\mathcal{N}, \mathcal{E}\rangle$, in which \mathcal{N} and \mathcal{E} are the sets of nodes and directed edges of \mathcal{G} , respectively. Every node corresponds to a unique reachable state, which implies a bijection between \mathcal{N} and \mathcal{R}_∞ . Every node n in \mathcal{G} is associated with a state $x(n) \in X$, and every edge $e \in \mathcal{E}$ is associated with a control $u(e)$. The same notation will also be used for the search graph defined in the next section with minor modification. An edge $e \in \mathcal{E}$ from node n_s to node n_e exists if there is a control $u(e) \in \tilde{U}_{vf}(x(n_s))$ such that $x(n_e) = \tilde{f}(x(n_s), u)$. We say that \mathcal{G} is *cyclic* if it contains a directed cycle.

If \mathcal{N} is finite, \mathcal{G} will be called *finite*; otherwise, \mathcal{G} will be called *infinite*. Note that if \tilde{U} is infinite, then \mathcal{G} might be infinite. This occurs for problems of FI and VI (as defined in Section ??). Intuitively, when \tilde{U} is finite, as in problems of FF and VF, it might seem that \mathcal{G} would finite. An interesting exception appears in [73], which provides conditions for \mathcal{R}_∞ to be dense for a class of discrete-time chained-form systems with quantized control sets, i.e., finite or with values on regular meshes in \mathbb{R}^m for some positive integer m .

General algorithm description An iterative procedure is generally defined in which each iteration attempts to add a new edge and corresponding trajectory segment to a search graph. The steps are briefly enumerated here, and then further explanation follows:

1. **Initialization:** Let $G\langle N, E\rangle$ represent a directed *search graph*, for which the node set, N contains a node for x_{init} and possibly other states in X_{free} , and the edge set, E , is empty.
2. **Select Node:** Choose a node $n_{cur} \in N$ for expansion.
3. **Generate Trajectory Segment:** Use a local planning method to generate a trajectory from $x(n_{cur})$ to some state x_{new} by applying some control u_{new} .
4. **Update Search Graph:** Determine whether an edge will be added to E . If so, then n_{cur} will be the starting node, and one of several possibilities exist for the ending node: 1) the ending node is selected from a node already in N , 2) a new node, n_{new} with associated state x_{new} is added to N , or 3) n_{new} is added as in the previous case, but other nodes in N may be deleted, and their associated edges are associated with n_{new} .
5. **Check for Solution:** Determine whether G encodes a solution to $\tilde{\mathcal{P}}$.
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.

Initialization In a single-tree approach, such as the planner of Barraquand and Latombe [54], only one node, $n(x_{init})$ exists in N . In a bidirectional approach, such as RRTEExtExt [464], $n(x_{goal})$ may also be included in N . One could initially

place thousands of nodes in N , as in the case of initializing a probabilistic roadmap [387, ?] with uniform random samples from X_{free} .

Select node This step is critical to the global search behavior of the algorithm; it is similar in some ways to the search queue prioritization in classical AI search. If dynamic programming is used, as in [54], then Step 2 selects a node with untried controls that has the trajectory shortest distance to x_{init} . Other possibilities are depth-first, breadth-first, or A^* [622]. In the case of an RRT [145, 466], a state, x_{rand} is generated at random in X , and then the nearest node (with respect to a metric on X) to x_{rand} is returned. Numerous other possibilities exist based on other algorithms (e.g., [164, 212, 261, 343]).

Generate trajectory segment Step 3 is implemented by a *local planner*, which may be considered as a separate component that produces a control $u_{new} \in \tilde{U}$ that evolves the system from x_{cur} to some state x_{new} .

Some local planners may attempt to reach another predetermined node, say n_{tar} [261, 524, 669, 698]. We refer to these as *connecting* local planners. These local planners may either: 1) succeed in exactly reaching $x(n_{tar})$, 2) yield a trajectory that ends within a specified distance bound from $x(n_{tar})$, or 3) may fail to reach sufficiently close to $x(n_{tar})$, in which case another node must be selected in Step 2. For the second condition, suppose that the user specifies a tolerance, $\epsilon_l > 0$, and requires that the local planner must achieve $\|x_{new} - x_{tar}\| \leq \epsilon_l$ to report success in reaching n_{tar} . If a connecting local planner is permitted to succeed under condition 2, then it is called *approximate*; otherwise, it is called *exact* if it only succeeds under condition 1.

To be consistent with the definition of \mathcal{G} in Section 15.3.1 even when approximate local planners with tolerance ϵ_l are used, we model an approximate local planner as an exact local planner plus a sampling process. Given the initial state x_i and goal state x_g , a state x_s in the ϵ_l neighborhood of x_g is first sampled, and then a control is designed by the associated exact local planner to connect x_i and x_s . With this model, \mathcal{G} is built using only exact local planners, ensuring that there will be no discontinuities, and $\tilde{U}(x)$ will be precisely the controls associated with edges in \mathcal{G} .

Since the algorithm must run for a countable number of iterations, the local planner actually uses a fixed, countable set of controls \tilde{U}_s . Control set \tilde{U}_s is generally obtained by sampling.

As mentioned in Section ??, \tilde{U} for state x depends on the local planner. If an exact connecting local planner is used, \tilde{U} for state x consists of controls, which drive the system to reachable states from x . If we assume that the connecting local planner will return a unique solution given an initial and goal state, then sampling in \tilde{U} and X is equivalent; that is, for every sampled state, if it is reachable from x , a control in \tilde{U} is sampled. Therefore, dispersion of the state space sampling could be used to characterize the control space sampling. Note that the above sampling

procedure could also be applied when an approximate connecting local planner is used. The difference is that there exists control errors between sampled controls and controls in \tilde{U} since these controls only drive the system from initial states to the neighborhood of goal states.

When a non-connecting local planner is used, the sampling in \tilde{U} normally has two steps. The first sampling is in the interval \mathcal{D} . Each sampled point corresponds to a set of controls with the same duration in \tilde{U} , on which the second sampling procedure happens. The quality with which the sampled set \tilde{U}_s approximates \tilde{U} may be characterized by dispersion [579], which may be defined as follows for two sets, A and B , such that $A \subseteq B$ and B is a subset of a normed space. The *dispersion* of A with respect to B is

$$\sup_{b \in B} \inf_{a \in A} \|a - b\|, \quad (15.4)$$

in which $\|\cdot\|$ is the norm. Therefore, the first sampling procedure is characterized by $\epsilon_t \in [0, \infty)$, which denotes dispersion of the set of duration of controls in \tilde{U}_s with respect to \mathcal{D} . The second sampling procedure is uniformly characterized by $\epsilon_u \in [0, \infty)$, that is, for every sampling set on which the second sampling procedure happens, dispersion of the sampled control set with respect to this sampling set is always ϵ_u . Control set \tilde{U}_s could be either provided as a parameter to the algorithm or calculated by the algorithm given ϵ_u and ϵ_t . If \tilde{U}_s is given, ϵ_u and ϵ_t could be also calculated once they are given.

Update search graph As indicated in Step 4, there are several possible alternatives to updating the graph, depending on the particular algorithm. The simplest case is to simply add an edge associated with u_{new} connecting n_{cur} to a new node associated with x_{new} to G . If the local planner exactly reaches some existing node, n_{tar} , then the edge is added from n_{cur} to n_{tar} since $x_{new} = x(n_{tar})$; otherwise, the systematic search requirement would be violated. If an approximate connecting local planner is used, then the behavior is the same, except that the edge is added from n_{cur} to n_{tar} whenever $\|x_{new} - x(n_{tar})\| \leq \epsilon_l$.

An additional complication is caused by state space discretization. Some planners try to avoid generating nodes that are too close to each other. For example, in the method of Barraquand and Latombe [54], the space is partitioned into a tiling of rectangular cells. As the algorithm runs, a new node is introduced in the search graph only when it lies in an unvisited cell. This ensures that each cell will contain no more than one node, which prevents the algorithm from generating a countably infinite number of states in the search graph.

The resolution of this state space discretization and many other schemes can be expressed in terms of the dispersion of the search graph nodes with respect to X_{free} . Let $\epsilon_d \in [0, \infty)$ denote a bound on the dispersion due to state space discretization. If u_{new} leads to a violation-free trajectory, then the algorithm must add x_{new} to N if the nearest vertex in N is at least ϵ_d from x_{new} . If x_{new}

is within distance ϵ_d of some other nodes in N , then one of two behaviors may occur, depending on the particular algorithm: 1) x_{new} is discarded, or 2) x_{new} is inserted, but all existing nodes within distance ϵ_d of x_{new} are deleted.

The parameter ϵ_d may be used to specify the size of the cells. For example, under an ℓ^∞ metric, ϵ_d directly gives the maximal cell width. In addition to using predefined partitions, other schemes may be possible. For example, a new node may be inserted into the search graph only if there exist no other nodes within distance ϵ_d .

Check for solution Step 5 must determine whether a solution in the sense defined in ?? exists within the graph. A candidate solution can be constructed from any connected sequence, (e_1, \dots, e_k) of edges (path) in G that starts with $n(x_{init})$. Starting at x_{init} , the control $u(e_i)$ is applied over its specified duration, for each i from 1 to k . The motion equation, f , is integrated during this process, and it can be determined whether the resulting trajectory is violation free and terminates within ϵ_s of x_{goal} .

The determination of which sequences to check for solutions depends on the particular algorithm. In many cases, the number of new candidate solutions that appear in one iteration may be small. In this case, all of them could be checked. In other algorithms, heuristics may be used to prune the consideration of too many candidate solutions. For example, a connection tolerance could be given and solution checking happens only when the distance between states in two subgraphs is less than the tolerance. For the purposes of resolution completeness analysis, we assume that no such pruning is performed.

15.3.2 Tree-Based Dynamic Programming

The forward dynamic programming (FDP) method is similar to an RRT in that it grows a tree from x_{init} . The key difference is that FDP uses dynamic programming to decide how to incrementally expand the tree, as opposed to nearest-neighbors of random samples. FDP performs a systematic exploration over fine-resolution grid that is placed over the state space. This limits its applicability to low-dimensional state spaces (up to 3 or 4 dimensions).

The configuration space, X , is divided into a rectangular grid (typically there are a hundred grid points per axis). Each element of the grid is called a *cell*, which designates a rectangular subset of X . One of three different labels can be applied to each cell:

- OBST: The cell contains points in X_{obs} .
- FREE: The cell has not yet been visited by the algorithm, and it lies entirely in X_{free} .
- VISITED: The cell has been visited, and it lies entirely in X_{free} .

Initially, all cells are labeled either FREE or OBST by using an collision detection algorithm.

Let Q represent a priority queue in which the elements are configurations, sorted in increasing order according to L , which represents the cost accumulated along the path constructed so far from x_{init} to x . This cost can be assigned in many different ways. It could simply represent the time (number of Δt steps), or could count the number of times a car changes directions.

The algorithm proceeds as follows:

FORWARD_DYNAMIC_PROGRAMMING(x_{init}, x_{goal})

```

1   $Q.insert(x_{init}, L);$ 
2   $G.init(x_{init});$ 
3  while  $Q \neq \emptyset$  and FREE( $x_{goal}$ )
4       $x_{cur} \rightarrow Q.pop();$ 
5      for each  $x \in \text{NBHD}(x_{cur})$ 
6          if FREE( $x$ )
7               $Q.insert(x, L);$ 
8               $G.add\_vertex(x);$ 
9               $G.add\_edge(x_{cur}, x);$ 
10             Label cell that contains  $x$  as VISITED;
11  Return  $G;$ 

```

The algorithm iterative grows a tree, G , which it rooted at x_{init} . The NBHD function tries the possible inputs, and returns a set of configurations that can be reached in time Δt . For each of these configurations, if the cell that contains it is FREE, then G is extended. At any given time, there is at most one vertex per cell. The algorithm terminates when the cell that contains the goal has been reached.

15.3.3 RDT-Based Methods

This is very incomplete...

The RRT planning method can be easily adapted to the case of nonholonomic planning. All references to configurations are replaced by references to states; this is merely a change of names. The only important difference between holonomic planning and nonholonomic planning with an RRT occurs in the EXTEND procedure. For holonomic planning, the function NEW_CONFIG generated a configuration that lies on the line segment that connects q to q_{near} . For nonholonomic planning, motions must be generated by applying inputs. The NEW_CONFIG function is replaced by NEW_STATE, which attempts to apply all of the inputs in U , and selects the input that generates an x_{new} that is closest to x_{near} with respect to the metric ρ . If U is infinite, then it can be approximated with a finite set of inputs.

15.3.4 Other Sampling-Based Methods

Hsu, Kindel, Latombe, Rock

Sampling-based roadmap approaches

15.4 Gradient-Based Optimization Techniques

The importance of gap reduction

15.5 Optimal Feedback Strategies

15.5.1 Problem Definition

15.5.2 Exact Solutions for Linear Systems

15.5.3 Functional Dynamic Programming

Bibliography

- [1] R. Abgrall. Numerical discretization of the first-order Hamilton-Jacobi Equation on triangular meshes. *Comm. Pure Appl. Math.*, 49(12):1339–1373, December 1996.
- [2] A. Abrams and R. Ghrist. Finding topology in a factory: Configuration spaces. *The American Mathematics Monthly*, 109:140–150, February 2002.
- [3] C. C. Adams. *The Knot Book: An elementary introduction to the mathematical theory of knots*. W. H. Freeman, New York, 1994.
- [4] M. D. Adams, H. Hu, and P. J. Roberts. Towards a real-time architecture for obstacle avoidance and path planning in mobile robots. In *IEEE Int. Conf. Robot. & Autom.*, pages 584–589, May 1990.
- [5] P. K. Agarwal, N. Amenta, B. Aronov, and M. Sharir. Largest placements and motion planning of a convex polygon. In J.-P. Laumond and M. Overmars, editors, *Robotics: The Algorithmic Perspective*. A.K. Peters, Wellesley, MA, 1996.
- [6] P. K. Agarwal, J.-C. Latombe, R. Motwani, and P. Raghavan. Nonholonomic path planning for pushing a disk among obstacles. In *IEEE Int. Conf. Robot. & Autom.*, 1997.
- [7] P. K. Agarwal, P. Raghavan, and H. Tamaki. Motion planning for a steering constrained robot through moderate obstacles. In *Proc. ACM Symposium on Computational Geometry*, 1995.
- [8] S. Akella, W. H. Huang, K. M. Lynch, and M. T. Mason. Sensorless parts feeding with a one joint robot. In J.-P. Laumond and M. Overmars, editors, *Algorithms for Robotic Motion and Manipulation*, pages 229–237. A K Peters, Wellesley, MA, 1997.
- [9] R. Alami, J. P. Laumond, and T. Siméon. Two manipulation planning algorithms. In *Proc. Workshop on Algorithmic Foundations of Robotics (WAFR 94)*, 1994.

- [10] R. Alami, T. Siméon, and J. P. Laumond. A geometrical approach to planning manipulation tasks. In *5th Int. Symp. Robot. Res.*, pages 113–119, 1989.
- [11] H. Alt, R. Fleischer, M. Kaufmann, K. Mehlhorn, S. Naher, S. Schirra, and C. Uhrig. Approximate motion planning and the complexity of the boundary of the union of simple geometric figures. In *Proc. ACM Symposium on Computational Geometry*, pages 281–289, 1990.
- [12] E. Altman and A. Shwartz. Markov decision problems and state-action frequencies. *SIAM J. Control & Optimization*, 29(4):786–809, July 1991.
- [13] N. M. Amato, O. B. Bayazit, L. K. Dale, C. Jones, and D. Vallejo. OBPRM: An obstacle-based PRM for 3D workspaces. In *Proceedings of the Workshop on Algorithmic Foundations of Robotics*, pages 155–168, 1998.
- [14] N. M. Amato, K. A. Dill, and G. Song. Using motion planning to map protein folding landscapes and analyze folding kinetics of known native structures. In *Proceedings of the 6th ACM International Conference on Computational Molecular Biology (RECOMB)*, pages 2–11, 2002.
- [15] N. M. Amato and G. Song. Using motion planning to study protein folding pathways. *Journal of Computational Biology*, 9(2):149–168, 2002.
- [16] N. M. Amato and Y. Wu. A randomized roadmap method for path and manipulation planning. In *IEEE Int. Conf. Robot. & Autom.*, pages 113–120, 1996.
- [17] N.M. Amato, O.B. Bayazit, L.K. Dale, C. Jones, and D. Vallejo. Choosing good distance metrics and local planners for probabilistic roadmap methods. In *IEEE Int. Conf. Robot. & Autom.*, pages 630–637, 1998.
- [18] N.M. Amato, O.B. Bayazit, L.K. Dale, C. Jones, and D. Vallejo. Choosing good distance metrics and local planners for probabilistic roadmap methods. *IEEE Trans. Robot. & Autom.*, 16(4):442–447, Aug 2000.
- [19] B. D. Anderson and J. B. Moore. *Optimal Control: Linear-Quadratic Methods*. Prentice-Hall, Englewood Cliffs, NJ, 1990.
- [20] J. Angeles. *Fundamentals of robotics mechanical systems: Theory, methods, and algorithms*. Springer-Verlag, Berlin, 2003.
- [21] M. D. Ardema and J. M. Skowronski. Dynamic game applied to coordination control of two arm robotic system. In R. P. Hämmäläinen and H. K. Ehtamo, editors, *Differential Games - Developments in Modelling and Computation*, pages 118–130. Springer-Verlag, Berlin, 1991.

- [22] E. M. Arkin and R. Hassin. Approximation algorithms for the geometric covering traveling salesman problem. *Discrete Applied Mathematics*, 55:194–218, 1994.
- [23] M. A. Armstrong. *Basic Topology*. Springer-Verlag, New York, NY, 1983.
- [24] D. S. Arnon. Geometric reasoning with logic and algebra. *Artif. Intell.*, 37(1-3):37–60, 1988.
- [25] B. Aronov and M. Sharir. On translational motion planning of a convex polyhedron in 3-space. *SIAM J. Comput.*, 26(6):1875–1803, December 1997.
- [26] J. Arvo. Fast random rotation matrices. In D. Kirk, editor, *Graphics Gems III*, pages 117–120. Academic Press, 1992.
- [27] S. Arya and D. M. Mount. Algorithm for fast vector quantization. In *IEEE Data Compression Conference*, pages 381–390, March 1993.
- [28] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.
- [29] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions (revised version). In *Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, January 1994.
- [30] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45:891–923, 1998.
- [31] R. B. Ash. *Information Theory*. Dover Publications, New York, NY, 1990.
- [32] A. Atramentov and S. M. LaValle. Efficient nearest neighbor searching for motion planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 632–637, 2002.
- [33] F. Avnaim, J. D. Boissonat, and B. Faverjon. A practical exact planning algorithm for polygonal objects amidst polygonal obstacles. In *IEEE Int. Conf. Robot. & Autom.*, pages 1656–1660, 1988.
- [34] J. Bañon. Implementation and extension of the ladder algorithm. In *IEEE Int. Conf. Robot. & Autom.*, pages 1548–1553, 1990.
- [35] T. Başar. Game theory and H^∞ -optimal control: The continuous-time case. In R. P. Hämmäläinen and H. K. Ehtamo, editors, *Differential Games - Developments in Modelling and Computation*, pages 171–186. Springer-Verlag, Berlin, 1991.

- [36] T. Başar and P. R. Kumar. On worst case design strategies. *Comput. Math. Applic.*, 13(1-3):239–245, 1987.
- [37] T. Başar and G. J. Olsder. *Dynamic Noncooperative Game Theory*. Academic Press, London, 1982.
- [38] B. Baginski. The Z^3 method for fast path planning in dynamic environments. In *Proceedings of IASTED Conference on Applications of Control and Robotics*, pages 47–52, 1996.
- [39] B. Baginski. *Motion Planning for Manipulators with Many Degrees of Freedom - The BB-Method*. PhD thesis, Technische Universität München, 1998.
- [40] M. J. Bailey. Tele-manufacturing: Rapid prototyping on the internet. *IEEE Computer Graphics & Applications*, 15(6):20–26, November 1995.
- [41] D. Balkcom and M. Mason. Geometric construction of time optimal trajectories for differential drive robots. In *Preprints of the 4th Workshop on Algorithmic Foundations of Robotics*, Dartmouth College, Hanover, NH, 2000.
- [42] J. E. Banta, Y. Zhen, X. Z. Wang, G. Zhang, M. T. Smith, and M. A. Abidi. A "best-next-view" algorithm for three-dimensional scene reconstruction using range images. In *Proc. SPIE vol. 2588*, pages 418–29, 1995.
- [43] M. Barbehenn, P. C. Chen, and S. Hutchinson. An efficient hybrid planner in changing environments. In *IEEE Int. Conf. Robot. & Autom.*, pages 2755–2760, 1994.
- [44] M. Barbehenn and S. Hutchinson. Efficient search and hierarchical motion planning by dynamically maintaining single-source shortest paths trees. *IEEE Trans. Robot. & Autom.*, 11(2):198–214, April 1995.
- [45] J. Bares, M. Hebert, T. Kanade, E. Krotkov, T. Mitchell, R. Simmons, and W. Whittaker. AMBLER an autonomous rover for planetary exploration. *IEEE Computer*, 22(6):18–26, Jun 1989.
- [46] S. Barnett. *Matrices in Control Theory*. Van Nostrand Reinhold Company, New York, NY, 1971.
- [47] J. Barraquand and P. Ferbach. A penalty function method for constrained motion planning. In *IEEE Int. Conf. Robot. & Autom.*, pages 1235–1242, 1994.
- [48] J. Barraquand and P. Ferbach. Motion planning with uncertainty: The information space approach. In *IEEE Int. Conf. Robot. & Autom.*, pages 1341–1348, 1995.

- [49] J. Barraquand, L. Kavraki, J.-C. Latombe, T.-Y. Li, R. Motwani, and P. Raghavan. A random sampling scheme for robot path planning. In G. Giralt and G. Hirzinger, editors, *Proc. of the 7th International Symposium on Robotics Research*, pages 249–264. Springer, New York, NY, 1996.
- [50] J. Barraquand, B. Langlois, and J. C. Latombe. Numerical potential field techniques for robot path planning. *IEEE Trans. Syst., Man, Cybern.*, 22(2):224–241, 1992.
- [51] J. Barraquand and J.-C. Latombe. A Monte-Carlo algorithm for path planning with many degrees of freedom. In *IEEE Int. Conf. Robot. & Autom.*, pages 1712–1717, 1990.
- [52] J. Barraquand and J.-C. Latombe. Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles. In *IEEE Int. Conf. Robot. & Autom.*, pages 2328–2335, 1991.
- [53] J. Barraquand and J.-C. Latombe. Robot motion planning: A distributed representation approach. *Int. J. Robot. Res.*, 10(6):628–649, December 1991.
- [54] J. Barraquand and J.-C. Latombe. Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles. *Algorithmica*, 10:121–155, 1993.
- [55] A. G. Barto, R. S. Sutton, and C. J. C. H. Watkins. Learning and sequential decision making. In M. Gabriel and J.W. Moore, editors, *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pages 539–602. MIT Press, 1990.
- [56] S. Basu, R. Pollack, and M.-F. Roy. Computing roadmaps of semi-algebraic sets on a variety. Submitted for publication, 1998.
- [57] S. Basu, R. Pollack, and M. F. Roy. Computing roadmaps of semi-algebraic sets on a variety. *Journal of the American Society of Mathematics*, 3(1):55–82, 1999.
- [58] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in Real Algebraic Geometry*. Springer-Verlag, Berlin, 2003.
- [59] K. Basye, T. Dean, J. Kirman, and M. Lejter. A decision-theoretic approach to planning, perception, and control. *IEEE Expert*, 7(4):58–65, August 1992.
- [60] M. Baykal-Gursoy. A sample-path approach to stochastic games. In *IEEE Conf. Decision & Control*, pages 180–185, Tampa, FL, December 1989.
- [61] C. Becker, H. González-Baños, J.-C. Latombe, and C. Tomasi. An intelligent observer. In *Preprints of International Symposium on Experimental Robotics*, pages 94–99, 1995.

- [62] K. E. Bekris, B. Y. Chen, A. Ladd, E. Plakue, and L. E. Kavraki. Multiple query probabilistic roadmap planning using single query primitives. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, 2003.
- [63] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.
- [64] R. E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, Princeton, NJ, 1962.
- [65] M. A. Bender, A. Fernandez, D. Ron A. Sahai, and S. Vadhan. The power of a pebble: Exploring and mapping directed graphs. In *Proc. Annual Symposium on Foundations of Computer Science*, 1998.
- [66] J. Bernard, J. Shannan, and M. Vanderploeg. Vehicle rollover on smooth surfaces. In *Proc. SAE Passenger Car Meeting and Exposition*, Dearborn, Michigan, 1989.
- [67] D. Bertsekas. *Dynamic Programming and Optimal Control: Volume I*. Athena Scientific, Belmont, MA, USA, 2000.
- [68] D. Bertsekas. *Dynamic Programming and Optimal Control: Volume II*. Athena Scientific, Belmont, MA, USA, 2000.
- [69] D. P. Bertsekas. Convergence in discretization procedures in dynamic programming. *IEEE Trans. Autom. Control*, 20(3):415–419, June 1975.
- [70] D. P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Englewood Cliffs, NJ, 1987.
- [71] J. T. Betts. Survey of numerical methods for trajectory optimization. *J. of Guidance, Control, and Dynamics*, 21(2):193–207, March-April 1998.
- [72] W. F. Bialas. Cooperative n -person Stackelberg games. In *IEEE Conf. Decision & Control*, pages 2439–2444, Tampa, FL, December 1989.
- [73] A. Bicchi, A. Marigo, and B. Piccoli. On the reachability of quantized control systems. *IEEE Trans. on Automatic Control*, 47(4):546–563, April 2002.
- [74] Z. Bien and J. Lee. A minimum-time trajectory planning method for two robots. *IEEE Trans. Robot. & Autom.*, 8(3):414–418, June 1992.
- [75] D. Bienstock and P. Seymour. Monotonicity in graph searching. *J. Algorithms*, 12:239–245, 1991.
- [76] D. Blackwell and M. A. Girshik. *Theory of Games and Statistical Decisions*. Dover Publications, New York, NY, 1979.

- [77] M. Blum and D. Kozen. On the power of the compass (or, why mazes are easier to search than graphs). In *Proc. Annual Symposium on Foundations of Computer Science*, pages 132–142, 1978.
- [78] J. E. Bobrow. Optimal robot path planning using the minimum-time criterion. *IEEE Trans. Robot. & Autom.*, 4(4):443–450, August 1988.
- [79] J. E. Bobrow, S. Dubowsky, and J. S. Gibson. Time-optimal control of robotic manipulators along specified paths. *Int. J. Robot. Res.*, 4(3):3–17, 1985.
- [80] R. Bohlin. Path planning in practice; lazy evaluation on a multi-resolution grid. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, 2001.
- [81] R. Bohlin and L. Kavraki. Path planning using Lazy PRM. In *IEEE Int. Conf. Robot. & Autom.*, 2000.
- [82] R. Bohlin and L. Kavraki. A randomized algorithm for robot path planning based on lazy evaluation. In S. Rajasekaran, P. Pardalos, J. Reif, and J. Rolim, editors, *Handbook on Randomized Computation*. Kluwer Academic, 2001.
- [83] J.-D. Boissonat and M. Yvinec. *Algorithmic Geometry*. Cambridge Press, 1998.
- [84] J.-D. Boissonat, A. Cérézo, and J. Leblond. Shortest paths of bounded curvature in the plane. *J. Intelligent and Robotic Systems*, 11:5–20, 1994.
- [85] J. D. Boissonat and S. Lazard. A polynomial-time algorithm for computing a shortest path of bounded curvature amidst moderate obstacles. In *Proc. ACM Symposium on Computational Geometry*, pages 242–251, 1996.
- [86] V. Boor, N. H. Overmars, and A. F. van der Stappen. The gaussian sampling strategy for probabilistic roadmap planners. In *IEEE Int. Conf. Robot. & Autom.*, pages 1018–1023, 1999.
- [87] J. Borenstein and Y. Koren. Real-time obstacle avoidance for fast mobile robots. *IEEE Trans. Syst., Man, Cybern.*, 19(5):1179–1187, 1989.
- [88] B. Bouilly, T. Siméon, and R. Alami. A numerical technique for planning motion strategies of a mobile robot in presence of uncertainty. In *IEEE Int. Conf. Robot. & Autom.*, pages 1327–1332, 1995.
- [89] H. I. Bozma and J. S. Duncan. Integration of vision modules: A game-theoretic approach. In *Proc. IEEE Conf. on Comp. Vision and Patt. Recog.*, pages 501–507, June 1991.

- [90] H. I. Bozma and J. S. Duncan. Modular system for image analysis using a game-theoretic framework. *Image and Vision Computing*, 10(6):431–443, 1992.
- [91] H. I. Bozma and J. S. Duncan. A game-theoretic approach to integration of modules. *IEEE Trans. Pattern Anal. Machine Intell.*, 16(11):1074–1086, November 1994.
- [92] R. I. Brafman, J.-C. Latombe, and Y. Shoham. Towards knowledge-level analysis of motion planning. In *Proc. of the Eleventh National Conference on Artificial Intelligence*, pages 670–675, 1993.
- [93] M. S. Branicky, V. S. Borkar, and S. K. Mitter. Percentile objective criteria in limiting average markov control problems. In *IEEE Conf. Decision & Control*, pages 4228–4234, Lake Buena Vista, FL, December 1994.
- [94] M. S. Branicky and M. M. Curtiss. Nonlinear and hybrid control via rrts. In *Proc. Intl. Symp. on Mathematical Theory of Networks and Systems*, 2002.
- [95] M. S. Branicky, S. M. LaValle, K. Olson, and L. Yang. Quasi-randomized path planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 1481–1487, 2001.
- [96] F. Brenti. q -Eulerian polynomials arising from Coxeter groups. *European J. Combin.*, 15:417–441, 1994.
- [97] M. Bridson and A. Haefliger. *Metric Spaces of Non-Positive Curvature*. Springer-Verlag, Berlin, 1999.
- [98] A. Briggs. An efficient algorithm for one-step compliant motion planning with uncertainty. In *5th ACM Symp. Comp. Geom.*, 1989.
- [99] A. J. Briggs and B. R. Donald. Robust geometric algorithms for sensor planning. In J.-P. Laumond and M. Overmars, editors, *Proc. 2nd Workshop on Algorithmic Foundations of Robotics*. A.K. Peters, Wellesley, MA, 1996.
- [100] R. A. Brooks. Solving the find-path problem by good representation of free space. *IEEE Trans. Syst., Man, Cybern.*, 13(3):190–197, 1983.
- [101] R. A. Brooks and T. Lozano-perez. A subdivision algorithm in configuration space for findpath with rotation. In *ICAI*, pages 799–806, 1983.
- [102] R. A. Brooks and T. Lozano-Pérez. A subdivision algorithm in configuration space for findpath with rotation. *IEEE Trans. Syst., Man, Cybern.*, SMC-15(2):224–233, 1985.
- [103] R. C. Brost. Automatic grasp planning in the presence of uncertainty. *Int. J. Robot. Res.*, 7(1):3–17, 1988.

- [104] R. C. Brost. *Analysis and Planning of Planar Manipulation Tasks*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1991.
- [105] R. C. Brost and A. D. Christiansen. Probabilistic analysis of manipulation tasks: A research agenda. In *IEEE Int. Conf. Robot. & Autom.*, volume 3, pages 549–556, 1993.
- [106] R. C. Brost and A. D. Christiansen. Probabilistic analysis of manipulation tasks: A computational framework. Technical Report SAND92-2033, Sandia National Laboratories, Albuquerque, NM, January 1994.
- [107] R. C. Brost and A. D. Christiansen. Probabilistic analysis of manipulation tasks: A computational framework. *Int. J. Robot. Res.*, 15(1):1–23, February 1996.
- [108] M. Brown. Optimal robot path planning via state space networks. Master's thesis, Princeton University, 1984.
- [109] J. Bruce and M. Veloso. Real-time randomized path planning for robot navigation. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, 2002.
- [110] A. E. Bryson and Y.-C. Ho. *Applied Optimal Control*. Hemisphere Publishing Corp., New York, NY, 1975.
- [111] S. J. Buckley. Fast motion planning for multiple moving robots. In *IEEE Int. Conf. Robot. & Autom.*, pages 322–326, 1989.
- [112] F. Bullo. Series expansions for the evolution of mechanical control systems. *SIAM J. Control and Optimization*, 40(1):166–190, 2001.
- [113] F. Bullo and K. M. Lynch. Kinematic controllability for decoupled trajectory planning in underactuated mechanical systems. *IEEE Trans. on Robotics and Automation*, 17(4):402–412, 2001.
- [114] J. W. Burdick. *Kinematic Analysis and Design of Redundant Manipulators*. PhD thesis, Stanford University, 1988.
- [115] W. Burgard, A. B. Cremers, D. Fox, D. Hahnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. The interactive museum tour-guide robot. In *Proc. Am. Assoc. Artif. Intell.*, pages 11–18, 1998.
- [116] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek. Sequential composition of dynamically dexterous robot behaviors. *Int. J. Robot. Res.*, 18(6):534–555, 1999.
- [117] L. G. Bushnell, D. M. Tilbury, and S. S. Sastry. Steering three-input non-holonomic systems: the fire truck example. *Int. J. Robot. Res.*, 14(4):366–381, 1995.

- [118] S. Cameron. A comparison of two fast algorithms for computing the distance between convex polyhedra. *IEEE Trans. Robot. & Autom.*, 13(6):915–920, December 1997.
- [119] J. Canny. Constructing roadmaps of semi-algebraic sets I. *Artif. Intell.*, 37:203–222, 1988.
- [120] J. Canny. Computing roadmaps of general semi-algebraic sets. *Computer Journal*, 36(5):504–514, 1995.
- [121] J. Canny, A. Rege, and J. Reif. An exact algorithm for kinodynamic planning in the plane. *Discrete and Computational Geometry*, 6:461–484, 1991.
- [122] J. Canny and J. Reif. New lower bound techniques for robot motion planning problems. In *Proc. IEEE Conf. on Foundations of Computer Science*, pages 49–60, 1987.
- [123] J. F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, MA, 1988.
- [124] J. F. Canny. On computability of fine motion plans. In *IEEE Int. Conf. Robot. & Autom.*, pages 177–182, 1989.
- [125] J. F. Canny and K. Y. Goldberg. "RISC" industrial robots: Recent results and current trends. In *IEEE Int. Conf. Robot. & Autom.*, pages 1951–1958, 1994.
- [126] S. Carpin and E. Pagello. On parallel rrts for multi-robot systems. In *Proc. of the 8th Conference of the Italian Association for Artificial Intelligence*, pages 834–841, 2002.
- [127] S. Carpin and G. Pillonetto. Robot motion planning using adaptive random walks. In *IEEE Int. Conf. Robot. & Autom.*, pages 3809–3814, 2003.
- [128] B. Mc Carragher and H. Asada. A discrete event approach to the control of robotic assembly tasks. In *Proc. of IEEE International Conference on Robotics and Automation*, volume 1, pages 331–336, 1993.
- [129] A. S. Çela and Y. Hamam. Optimal motion planning of a multiple-robot system based on decomposition coordination. *IEEE Trans. Robot. & Autom.*, 8(5):585–596, October 1992.
- [130] D. Challou, D. Boley, M. Gini, and V. Kumar. A parallel formulation of informed randomized search for robot motion planning problems. In *IEEE Int. Conf. Robot. & Autom.*, pages 709–714, 1995.

- [131] C. Chang, M. J. Chung, and B. H. Lee. Collision avoidance of two robot manipulators by minimum delay time. *IEEE Trans. Syst., Man, Cybern.*, 24(3):517–522, 1994.
- [132] H. Chang and T. Y. Li. Assembly maintainability study with motion planning. In *IEEE Int. Conf. Robot. & Autom.*, pages 1012–1019, 1995.
- [133] S.-C. Chang and D.-T. Liao. Scheduling flexible flow shops with no setup effects. *IEEE Trans. Robot. & Autom.*, 10(2):99–111, 1994.
- [134] F. Chaumette, P. Rives, and B. Espiau. Positioning of a robot with respect to an object, tracking it and estimating its velocity by visual servoing. In *IEEE Int. Conf. Robot. & Autom.*, pages 2249–2253, 1991.
- [135] B. Chazelle. A theorem on polygon cutting with applications. In *Proc. 23rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 339–349, 1982.
- [136] B. Chazelle. Approximation and decomposition of shapes. In J. T. Schwartz and C. K. Yap, editors, *Algorithmic and Geometric Aspects of Robotics*, pages 145–185. Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.
- [137] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6(5):485–524, 1991.
- [138] B. Chazelle. *The Discrepancy Method*. Cambridge University Press, 2000.
- [139] C.-T. Chen. *Linear System Theory and Design*. Holt, Rinehart, and Winston, New York, NY, 1984.
- [140] P.C. Chen and Y.K. Hwang. Sandros: A motion planner with performance proportional to task difficulty. In *Proc. of IEEE Int. Conf. Robotics and Automation*, pages 2346–2353, Nice, France, 1992.
- [141] Z. Chen and C.-M. Huang. Terrain exploration of a sensor-based robot moving among unknown obstacles of polygonal shape. *Robotica*, 12(1):33–44, 1994.
- [142] P. Cheng, E. Frazzoli, and S. M. LaValle. Exploiting group symmetries to improve precision in kinodynamic and nonholonomic planning. In *IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems*, 2003.
- [143] P. Cheng, E. Frazzoli, and S. M. LaValle. Improving the performance of sampling-based planners by using a symmetry-exploiting gap reduction algorithm. In *Proc. IEEE International Conference on Robotics and Automation*, 2004. Under review.

- [144] P. Cheng and S. M. LaValle. Reducing metric sensitivity in randomized trajectory design. In *Proc. IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems*, pages 43–48, 2001.
- [145] P. Cheng and S. M. LaValle. Resolution complete rapidly-exploring random trees. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 267–272, 2002.
- [146] P. Cheng, Z. Shen, and S. M. LaValle. Using randomization to find and optimize feasible trajectories for nonlinear systems. In *Proc. Annual Allerton Conference on Communications, Control, Computing*, pages 926–935, 2000.
- [147] P. Cheng, Z. Shen, and S. M. LaValle. RRT-based trajectory design for autonomous automobiles and spacecraft. *Archives of Control Sciences*, 11(3-4):167–194, 2001.
- [148] M. Cherif. Kinodynamic motion planning for all-terrain wheeled vehicles. In *IEEE Int. Conf. Robot. & Autom.*, 1999.
- [149] F. L. Chernousko, N. N. Bolotnik, and V. G. Gradetsky. *Manipulation Robots*. CRC Press, Ann Arbor, MI, 1994.
- [150] E. Cheung and V. Lumelsky. Motion planning for robot arm manipulators with proximity sensing. In *IEEE Int. Conf. Robot. & Autom.*, pages 740–745, 1988.
- [151] L. P. Chew and K. Kedem. A convex polygon among polygonal obstacles: Placement and high-clearance motion. *Comput. Geom. Theory Appl.*, 3:59–89, 1993.
- [152] Y. P. Chien. An algorithmic approach to the trajectory planning for multiple robots. In *IEEE Southeastern Symp. on Systems Theory*, pages 303–307, 1990.
- [153] W.-P. Chin and S. Ntafos. Optimum watchman routes. *Information Processing Letters*, 28:39–44, 1988.
- [154] G. Chirikjian, A. Pamecha, and I. Ebert-Uphoff. Evaluating efficiency of self-reconfiguration in a class of modular robots. *Journal of Robotic Systems*, 13(5):717–338, 1996.
- [155] G. S. Chirikjian and A. B. Kyatkin. *Engineering Applications of Noncommutative Harmonic Analysis*. CRC Press, Boca Raton, 2001.
- [156] H. Chitsaz, J. M. O’Kane, and S. M. LaValle. Pareto-optimal coordination of two translating polygonal robots on an acyclic roadmap. In *Proc. IEEE International Conference on Robotics and Automation*, 2004. Under review.

- [157] J. Choi, J. Sellen, and C. K. Yap. Precision-sensitive euclidean shortest path in 3-space. *Journal of Computer and System Sciences*, 31:288–301, 1985.
- [158] J. Choi, J. Sellen, and C. K. Yap. Approximate euclidean shortest path in 3-space. In *Proc. ACM Symposium on Computational Geometry*, pages 41–48, 1994.
- [159] J. Choi, J. Sellen, and C. K. Yap. Precision-sensitive euclidean shortest path in 3-space. In *Proc. ACM Symposium on Computational Geometry*, pages 350–359, 1995.
- [160] H. Choset. Coverage of known spaces: The boustrophedon cellupdar decomposition. *Autonomous Robots*, 9:247–253, 2000.
- [161] H. Choset. Coverage for robotics - A survey of recent results. *Annals of Mathematics and Artificial Intelligence*, 31:113–126, 2001.
- [162] H. Choset and J. Burdick. Sensor based planning, part I: The generalized Voronoi graph. In *IEEE Int. Conf. Robot. & Autom.*, pages 1649–1655, 1995.
- [163] H. Choset and P. Pignon. Cover path planning: The boustrophedon decomposition. In *Int. Conf. on Field and Service Robotics*, Canberra, Australia, December 1996.
- [164] P. Choudhury and K. Lynch. Trajectory planning for second-order under-actuated mechanical systems in presence of obstacles. In *Proceedings of the Workshop on Algorithmic Foundations of Robotics*, 2002.
- [165] K.-C. Chu. Team decision theory and information structures in optimal control problems-part II. *IEEE Trans. Autom. Control*, 17(1):22–28, February 1972.
- [166] C. K. Chui and G. Chen. *Kalman Filtering*. Springer-Verlag, Berlin, 1991.
- [167] D. E. Clark, G. Jones, P. Willett P. W. Kenny, and Glen. Pharmacophoric pattern matching in files of three-dimensional chemical structures: Comparison of conformational searching algorithms for flexible searching. *J. Chem. Inf. Comput. Sci.*, 34:197–206, 1994.
- [168] G. E. Collins. *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1975.
- [169] G. E. Collins. Quantifier elimination by cylindrical algebraic decomposition—twenty years of progress. In B. F. Caviness and J. R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 8–23. Springer-Verlag, Berlin, 1998.

- [170] C. Connolly and R. Grupen. The application of harmonic potential functions to robotics. *J. Robotic Systems*, 10(7):931–946, 1993.
- [171] C. Connolly, R. Grupen, and K. Souccar. A Hamiltonian framework for kinodynamic planning. In *Proc. of the IEEE International Conf. on Robotics and Automation (ICRA '95)*, Nagoya, Japan, 1995.
- [172] C. I. Connolly, J. B. Burns, and R. Weiss. Path planning using laplace's equation. In *IEEE Int. Conf. Robot. & Autom.*, pages 2102–2106, May 1990.
- [173] C. I. Conolly. The determination of next best views. In *IEEE Int. Conf. Robot. & Autom.*, pages 432–435, 1985.
- [174] J. H. Conway and N. J. A. Sloane. *Sphere Packings, Lattices, and Groups*. Springer-Verlag, Berlin, 1999.
- [175] H. W. Corley. Some multiple objective dynamic programs. *IEEE Trans. Autom. Control*, 30(12):1221–1222, December 1985.
- [176] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *An Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
- [177] J. Cortés. *Motion Planning Algorithms for General Closed-Chain Mechanisms*. PhD thesis, Institut National Polytechnique de Toulouse, Toulouse, France, 2003.
- [178] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms*. Springer-Verlag, Berlin, 1992.
- [179] H. S. M. Coxeter. *Regular Polytopes*. Dover Publications, New York, NY, 1973.
- [180] J. J. Craig. *Introduction to Robotics*. Addison-Wesley, Reading, MA, 1989.
- [181] D. Crass, I. Suzuki, and M. Yamashita. Searching for a mobile intruder in a corridor – the open edge variant of the polygon search problem. *Int. J. Comput. Geom. & Appl.*, 5(4):397–412, 1995.
- [182] J. C. Culberson. Sokoban is pspace-complete. In *Proceedings of the International Conference on Fun with Algorithms (FUN98)*, pages 65–76, Waterloo, Ontario, Canada, June 1998. Carleton-Scientific.
- [183] L. M. M. Custodio, J. J. S. Sentieiro, and C. F. G. Bispo. Production planning and scheduling using a fuzzy decision system. *IEEE Trans. Robot. & Autom.*, 10(2):160–168, 1994.
- [184] M. R. Cutkosky. *Robotic Grasping and Fine Manipulation*. Kluwer Academic Publishers, Boston, MA, 1985.

- [185] C. S. Czerwinski and P. B. Luh. Scheduling products with bills of materials using an improved lagrangian relaxation technique. *IEEE Trans. Robot. & Autom.*, 10(2):99–111, 1994.
- [186] A. Datta, C. A. Hipke, and S. Schuierer. Competitive searching in polygons—beyond generalized streets. In J. Staples, P. Eades, N. Katoh, and A. Moffat, editors, *Algorithms and Computation, ISAAC '95*, pages 32–41. Springer-Verlag, Berlin, 1995.
- [187] J. Davenport and J. Heintz. Real quantifier elimination of doubly exponential. *J. Symb. Comput.*, 5:29–35, 1988.
- [188] S. Davies. Multidimensional triangulation and interpolation for reinforcement learning. In *Advances in neural information processing systems*, 1996.
- [189] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, 1997.
- [190] J. Gil de Lamadrid and J. Zimmerman. Avoidance of obstacles with unknown trajectories: locally optimal paths and path complexity, part I. *Robotica*, 11:299–308, 1993.
- [191] M. de Rougement and J. F. Diaz-Frias. A theory of robust planning. In *IEEE Int. Conf. Robot. & Autom.*, pages 2453–2458, Nice, France, May 1992.
- [192] M. J. de Smith. *Distance and Path: The Development, Interpretation and Application of Distance Measurement in Mapping and Modelling*. PhD thesis, University College, University of London, London, 2003.
- [193] T. Dean, L. P. Kaebbling, J. Kirman, and A. Nicholson. Deliberation scheduling for time-critical sequential decision making. In *Uncertainty in Artificial Intelligence*, pages 309–316, Washington, D.C., July 1993.
- [194] T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1989.
- [195] T. Dean, K. Kanazawa, and J. Shewchuk. Prediction, observation, and estimation in planning and control. In *IEEE Symp. on Intelligent Control*, pages 645–650, 1990.
- [196] T. L. Dean and M. P. Wellman. *Planning and Control*. Morgan Kaufman, San Mateo, CA, 1991.
- [197] T. L. DeFazio and D. E. Whitney. Simplified generation of all mechanical assembly sequences. *IEEE Trans. Robot. & Autom.*, 3(6):640–658, 1987.

- [198] M. H. DeGroot. *Optimal Statistical Decisions*. McGraw Hill, New York, NY, 1970.
- [199] D. F. Delchamps. Stabilizing a linear system with quantized output record. *IEEE Trans. Autom. Control*, 35(8):916–926, 1990.
- [200] X. Deng, T. Kameda, and C. Papadimitriou. How to learn an unknown environment I: The rectilinear case. Available from "http://www.cs.berkeley.edu/~christos/", 1997.
- [201] O. Devillers, V. Dujmovic, H. Everett, X. Goaoc, S. Lazard, H.-S. Na, and S. Petitjean. A linear bound on the expected number of visibility events. In *ACM Sympos. Comput. Geom.*, 2002.
- [202] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [203] B. Donald and P. Xavier. Provably good approximation algorithms for optimal kinodynamic planning for cartesian robots and open chain manipulators. *Algorithmica*, 14(6):480–530, 1995.
- [204] B. Donald and P. Xavier. Provably good approximation algorithms for optimal kinodynamic planning: Robots with decoupled dynamics bounds. *Algorithmica*, 14(6):443–479, 1995.
- [205] B. R. Donald. Motion planning with six degrees of freedom. Technical Report AI-TR-791, Artificial Intelligence Lab., Massachusetts Institute of Technology, 1984.
- [206] B. R. Donald. *Error Detection and Recovery for Robot Motion Planning with Uncertainty*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1987.
- [207] B. R. Donald. A search algorithm for motion planning with six degrees of freedom. *Artif. Intell.*, 31:295–353, 1987.
- [208] B. R. Donald. A geometric approach to error detection and recovery for robot motion planning with uncertainty. *Artif. Intell.*, 37:223–271, 1988.
- [209] B. R. Donald. Planning multi-step error detection and recovery strategies. *Int. J. Robot. Res.*, 9(1):3–60, 1990.
- [210] B. R. Donald. On information invariants in robotics. *Artif. Intell.*, 72:217–304, 1995.
- [211] B. R. Donald and J. Jennings. Sensor interpretation and task-directed planning using perceptual equivalence classes. In *IEEE Int. Conf. Robot. & Autom.*, pages 190–197, Sacramento, CA, April 1991.

- [212] B. R. Donald, P. G. Xavier, J. Canny, and J. Reif. Kinodynamic planning. *Journal of the ACM*, 40:1048–66, November 1993.
- [213] K.-F. Böhringer B. R. Donald and N. C. MacDonald. Upper and lower bounds for programmable vector fields with applications to mems and vibratory plate parts feeders. In J.-P. Laumond and M. Overmars, editors, *Algorithms for Robotic Motion and Manipulation*. A K Peters, Wellesley, MA, 1997.
- [214] L. E. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79:497–516, 1957.
- [215] G. Dudek and C. Zhang. Vision-based robot localization without explicit object models. In *IEEE Int. Conf. Robot. & Autom.*, pages 76–82, 1996.
- [216] I. Duleba. *Algorithms of Motion Planning for Nonholonomic Robots*. Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław, Poland, 1998.
- [217] H. F. Durrant-Whyte. *Integration, Coordination and Control of Multi-Sensor Planning Systems*. Kluwer Academic Publishers, Boston, MA, 1988.
- [218] H. F. Durrant-Whyte. Uncertain geometry in robotics. *IEEE Trans. Robot. & Autom.*, 4(1):23–31, February 1988.
- [219] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, Berlin, 1987.
- [220] A. Efrat, L. J. Guibas, D. C. Lin, J. S. B. Mitchell, and T. M. Murali. Sweeping simple polygons with a chain of guards. In *Proc. ACM-SIAM Sympos. Discrete Algorithms*, 2000.
- [221] S. Ehmann and M.C. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. In *Proc. Eurographics 2001*, 2001.
- [222] D. Einav and M. R. Fehling. Computationally-optimal real-resource strategies for independent, uninterruptible methods. In *Uncertainty in Artificial Intelligence 6*, pages 145–158. North-Holland, Amsterdam, 1991.
- [223] T. F. Elbert. *Estimation and Control of Systems*. Van Nostrand Reinhold, New York, NY, 1984.
- [224] A. Elfes. Using occupancy grids for mobile robot perception and navigation. *IEEE Computer*, 22(6):46–57, June 1989.

- [225] I. Z. Emiris and B. Mourrain. Computer algebra methods for studying and computing molecular conformations. Technical report, Institut National De Recherche En Informatique Et En Automatique, Sophia-Antipolis, France, 1997.
- [226] M. Erdmann. Using backprojections for fine motion planning with uncertainty. *Int. J. Robot. Res.*, 5(1):19–45, 1986.
- [227] M. Erdmann. Randomization in robot tasks. *Int. J. Robot. Res.*, 11(5):399–436, October 1992.
- [228] M. Erdmann. Randomization for robot tasks: Using dynamic programming in the space of knowledge states. *Algorithmica*, 10:248–291, 1993.
- [229] M. Erdmann. On a representation of friction in configuration space. *Int. J. Robot. Res.*, 13(3):240–271, 1994.
- [230] M. Erdmann. Understanding action and sensing by designing action-based sensors. *Int. J. Robot. Res.*, 14(5):483–509, 1995.
- [231] M. Erdmann. An exploration of nonprehensile two-palm manipulation using two zebra robots. In J.-P. Laumond and M. Overmars, editors, *Algorithms for Robotic Motion and Manipulation*, pages 239–254. A K Peters, Wellesley, MA, 1997.
- [232] M. Erdmann and T. Lozano-Perez. On multiple moving objects. In *IEEE Int. Conf. Robot. & Autom.*, pages 1419–1424, 1986.
- [233] M. Erdmann and T. Lozano-Pérez. On multiple moving objects. *Algorithmica*, 2:477–521, 1987.
- [234] M. Erdmann, M. T. Mason, and Jr. G. Vaněček. Mechanical parts orienting: The case of a polyedron on a table. *Algorithmica*, 10:206–247, 1993.
- [235] M. A. Erdmann. On motion planning with uncertainty. Master’s thesis, Massachusetts Institute of Technology, Cambridge, MA, August 1984.
- [236] M. A. Erdmann. *On Probabilistic Strategies for Robot Tasks*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1989.
- [237] M. A. Erdmann and M. T. Mason. An exploration of sensorless manipulation. *IEEE Trans. Robot. & Autom.*, 4(4):369–379, August 1988.
- [238] B. Espiau, F. Chaumette, and P. Rives. A new approach to visual servoing in robotics. *IEEE Trans. Robot. & Autom.*, 8(3):313–326, June 1992.
- [239] J.M. Esposito and V. Kumar. Efficient dynamic simulation of robotic systems with hierarchy. In *IEEE Int. Conf. Robot. & Autom.*, pages 2818–2823, 2001.

- [240] N. Faiz and S. K. Agrawal. Trajectory planning of robots with dynamics and inequalities. In *IEEE Int. Conf. Robot. & Autom.*, pages 3977–3983, 2000.
- [241] B. Faverjon. Obstacle avoidance using an octree in the configuration space of a manipulator. In *IEEE Int. Conf. Robot. & Autom.*, pages 504–512, 1984.
- [242] B. Faverjon. Hierarchical object models for efficient anti-collision algorithms. In *IEEE Int. Conf. Robot. & Autom.*, pages 333–340, 1989.
- [243] B. Faverjon and P. Tournassoud. A local based method for path planning of manipulators with a high number of degrees of freedom. In *IEEE Int. Conf. Robot. & Autom.*, pages 1152–1159, 1987.
- [244] J. T. Feddema and O. R. Mitchell. Vision-guided servoing with feature-based trajectory generation. *IEEE Trans. Robot. & Autom.*, 5(5):691–700, October 1989.
- [245] P. Ferbach. A method of progressive constraints for nonholonomic motion planning. In *IEEE Int. Conf. Robot. & Autom.*, pages 2949–2955, 1996.
- [246] R. Fierro, A. Das, V. Kumar, and J. P. Ostrowski. Hybrid control of formations of robots. In *IEEE Int. Conf. Robot. & Autom.*, pages 157–162, 2001.
- [247] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving. *Artif. Intell.*, 2:189–208, 1971.
- [248] J. A. Filar, D. Krass, and K. W. Ross. Percentile objective criteria in limiting average markov control problems. In *IEEE Conf. Decision & Control*, pages 1273–1276, Tampa, FL, December 1989.
- [249] P. W. Finn, D. Halperin, L. E. Kavraki, J.-C. Latombe, R. Motwani, C. Shelton, and S. Venkatasubramanian. Geometric manipulation of flexible ligands. In *Applied Computational Geometry (Lecture Notes in Computer Science, 1148)*, pages 67–78. Springer-Verlag, Berlin, 1996.
- [250] R. J. Firby. An investigation into reactive planning in complex domains. In *Proc. Am. Assoc. Artif. Intell.*, 1987.
- [251] G. F. Fishman. *Monte Carlo: Concepts, Algorithms, and Applications*. Springer-Verlag, Berlin, 1996.
- [252] M. Fliess, J. Levine, P. Martin, and P. Rouchon. Flatness and defect of nonlinear systems. *International Journal of Control*, 61(6):1327–1361, 1993.

- [253] G. B. Folland. *Real Analysis: Modern Techniques and Their Applications*. Wiley, New York, 1984.
- [254] S. Fortune and G. Wilfong. Planning constrained motion. In *STOCS*, pages 445–459, 1988.
- [255] A. Fox and S. Hutchinson. Exploiting visual constraints in the synthesis of uncertainty-tolerant motion plans. *IEEE Trans. Robot. & Autom.*, 1(11):56–71, February 1995.
- [256] D. Fox, W. Burgard, S. Thrun, and A. B. Cremers. Position estimation for mobile robots in dynamic environments. In *Proc. Am. Assoc. Artif. Intell.*, 1998.
- [257] T. Fraichard. Dynamic trajectory planning with dynamic constraints: A ‘state-time space’ approach. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 1393–1400, 1993.
- [258] Th. Fraichard and C. Laugier. Kinodynamic planning in a structured and time-varying 2d workspace. In *IEEE Int. Conf. Robot. & Autom.*, pages 2: 1500–1505, 1992.
- [259] Th. Fraichard and A. Scheuer. Car-like robots and moving obstacles. In *IEEE Int. Conf. Robot. & Autom.*, pages 64–69, 1994.
- [260] E. Frazzoli, M. A. Dahleh, and E. Feron. Robust hybrid control for autonomous vehicles motion planning. Technical Report LIDS-P-2468, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, 1999.
- [261] E. Frazzoli, M. A. Dahleh, and E. Feron. Real-time motion planning for agile autonomous vehicles. *AIAA Journal of Guidance and Control*, 25(1):116–129, 2002.
- [262] E. Freund and H. Hoyer. Path finding in multi robot systems including obstacle avoidance. *Int. J. Robot. Res.*, 7(1):42–70, February 1988.
- [263] J. H. Friedman, J. L. Bentley, and R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.
- [264] K. S. Fu, R. C. Gonzalez, and C. S. G. Lee. *Robotics: Control, Sensing, Vision, and Intelligence*. McGraw-Hill, New York, 1987.
- [265] K. Fujimura. On motion planning amidst transient obstacles. In *IEEE Int. Conf. Robot. & Autom.*, pages 1488–1493, 1992.

- [266] K. Fujimura and H. Samet. A hierarchical strategy for path planning among moving obstacles. Technical Report CAR-TR-237, Center for Automation Research, University of Maryland, November 1986.
- [267] K. Fujimura and H. Samet. Planning a time-minimal motion among moving obstacles. *Algorithmica*, 10:41–63, 1993.
- [268] Y. Gabriely and E. Rimon. Spanning-tree based coverage of continuous areas by a mobile robot. Technical report, Dept. of Mechanical Engineering, Technion, Israel Institute of Technology, December 1999.
- [269] Y. Gabriely and E. Rimon. Spanning-tree based coverage of continuous areas by a mobile robot. In *IEEE Int. Conf. Robot. & Autom.*, pages 1927–1933, 2001.
- [270] Y. Gabriely and E. Rimon. Competitive on-line coverage of grid environments by a mobile robot. *Computational Geometry: Theory and Applications*, 24:197–224, April 2003.
- [271] H. Geering, L. Guzzella, S. A. R. Hepner, and C. H. Onder. Time-optimal motions of robots in assembly tasks. *IEEE Trans. Automat. Contr.*, AC-31(6):512–518, June 1986.
- [272] R. Geraerts and M. Overmars. Sampling techniques for probabilistic roadmap planners. In *Proc. of International Conference on Intelligent Autonomous Systems*, 2004.
- [273] R. Geraerts and M. H. Overmars. A comparative study of probabilistic roadmap planners. In *Proc. Workshop on the Algorithmic Foundations of Robotics*, December 2002.
- [274] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. In press, 2004.
- [275] A. K. Ghose, M. E. Logan, A. M. Treasurywala, H. Wang, R. C. Wahl, B. E. Tomczuk, M. R. Gowravaram, E. P. Jaeger, and J. J. Wendoloski. Determination of pharmacophoric geometry for collagenase inhibitors using a novel computational method and its verification using molecular dynamics, NMR, and X-ray crystallography. *J. Am. Chem. Soc.*, 117:4671–4682, 1995.
- [276] S. K. Ghosh and D. M. Mount. An output sensitive algorithm for computing visibility graphs. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 11–19, 1987.
- [277] R. Ghrist. Shape complexes for metamorphic robot systems. In *Proc. Workshop on the Algorithmic Foundations of Robotics*, December 2002.

- [278] E. G. Gilbert and D. W. Johnson. Distance functions and their application to robot path planning in the presence of obstacles. *IEEE Trans. Robot. & Autom.*, 1(1):21–30, March 1985.
- [279] E. G. Gilbert, D. W. Johnson, and S. S. Keerth. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE J. of Robot. & Autom.*, RA-4(2):193–203, Apr 1988.
- [280] T. N. Gillespie. *Fundamentals of Vehicle Dynamics*. Society of Automotive Engineers, Warrendale, PA, 1992.
- [281] B. Glavina. Solving findpath by combination of goal-directed and randomized search. In *IEEE Int. Conf. Robot. & Autom.*, pages 1718–1723, May 1990.
- [282] B. Glavina. *Planung kollisionsfreier Bewegungen für Manipulatoren durch Kombination von zielgerichteter Suche und zufallsgesteuerter Zwischenziel-erzeugung*. PhD thesis, Technische Universität München, 1991.
- [283] P. J. Gmytrasiewicz, E. H. Durfee, and D. K. Wehe. A decision-theoretic approach to coordinating multi-agent interactions. In *Proc. Int. Joint Conf. on Artif. Intell.*, pages 62–68, 1991.
- [284] K. Y. Goldberg. *Stochastic Plans for Robotic Manipulation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, August 1990.
- [285] K. Y. Goldberg. Orienting polygonal parts without sensors. *Algorithmica*, 10:201–225, 1993.
- [286] K. Y. Goldberg and M. T. Mason. Bayesian grasping. In *IEEE Int. Conf. Robot. & Autom.*, 1990.
- [287] G. H. Golub and C. F. Van Loan. *Matrix Computations (3rd ed)*. Johns Hopkins University Press, Baltimore, MD, 1996.
- [288] J. Gonzalez, A. Reina, and A. Ollero. Map buliding for a mobile robot equipped with a 2D laser rangefinder. In *IEEE Int. Conf. Robot. & Autom.*, pages 1904–1909, 1994.
- [289] R. Gonzalez and E. Rofman. On deterministic control problems: an approximation procedure for the optimal cost, parts i, ii. *SIAM J. Control Optimization*, 23:242–285, 1985.
- [290] H. H. González-Baños, L. Guibas, J.-C. Latombe, S. M. LaValle, D. Lin, R. Motwani, and C. Tomasi. Motion planning with visibility constraints: Building autonomous observers. In Y. Shirai and S. Hirose, editors, *Proc. Eighth Int'l Symp. on Robotics Research*, pages 95–101. Springer-Verlag, Berlin, 1998.

- [291] H. H. Gonzalez-Banos, C.-Y. Lee, and J.-C. Latombe. Real-time combinatorial tracking of a target moving unpredictably among obstacles. In *IEEE Int. Conf. Robot. & Autom.*, 2002.
- [292] J. E. Goodman and J. O’Roarke. *Handbook of Discrete and Computational Geometry*. CRC Press, New York, 1997.
- [293] S. Gottschalk, M. C. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *SIGGRAPH ’96 Proc.*, 1996.
- [294] S. Gottschlich, C. Ramos, and D. Lyons. Assembly and task planning: A taxonomy. *IEEE Robotics and Automation Magazine*, 1(3):4–12, 1994.
- [295] S. N. Gottschlich and A. C. Kak. AMP-CAD: Automatic assembly motion planning using CAD models of parts. *Robotics and Autonomous Systems*, 13:245–289, 1994.
- [296] V. E. Gough and S. G. Whitehall. Universal tyre test machine. In *Proc. 9th Int. Tech. Congress F.I.S.I.T.A.*, May 2000.
- [297] L. Grüne. An adaptive grid scheme for the discrete Hamilton-Jacobi-Bellman equation. *Numerische Mathematik*, 75:319–337, 1997.
- [298] R. Guernut, A. Ouldamar, and B. Vachon. Visual target tracking by a mobile robot in a cluttered environment. *Systems Analysis Modeling Simulation*, 18-19:225–228, 1995.
- [299] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *AMC Trans. Graphics*, 4(2):74–123, 1985.
- [300] L. J. Guibas, D. Hsu, and L. Zhang. H-Walk: Hierarchical distance computation for moving convex bodies. In *Proc. ACM Symposium on Computational Geometry*, pages 265–273, 1999.
- [301] L. J. Guibas, J.-C. Latombe, S. M. LaValle, D. Lin, and R. Motwani. Visibility-based pursuit-evasion in a polygonal environment. *International Journal of Computational Geometry and Applications*, 9(5):471–494, 1999.
- [302] L. J. Guibas, R. Motwani, and P. Raghavan. The robot localization problem. In K. Goldberg, D. Halperin, J.-C. Latombe, and R. Wilson, editors, *Proc. 1st Workshop on Algorithmic Foundations of Robotics*, pages 269–282. A.K. Peters, Wellesley, MA, 1995.
- [303] K. Gupta and X. Zhu. Practical motion planning for many degrees of freedom: A novel approach within sequential framework. *Journal of Robotics Systems*, 2(12):105–118, 1995.

- [304] J.-S. Gutman and C. Schlegel. Amos: Comparison of scan matching approaches for self-localizing in indoor environments. In *1st Euromicro Workshop on Advanced Mobile Robots (Eurobot)*, 1996.
- [305] G. Hager and H. G. Durrant-Whyte. Information and multi-sensor coordination. In J. F. Lemmer and L. N. Kanal, editors, *Uncertainty in Artificial Intelligence*, pages 381–393. Elsevier, New York, NY, 1988.
- [306] G. D. Hager. *Task-Directed Sensor Fusion and Planning*. Kluwer Academic Publishers, Boston, MA, 1990.
- [307] O. Hájek. *Pursuit Games*. Academic Press, New York, 1975.
- [308] K. Haji-Ghassemi. On differential games of fixed duration with phase coordinate restrictions on one player. *SIAM J. Control & Optimization*, 28(3):624–652, May 1990.
- [309] D. Halperin, J.-C. Latombe, and R. H. Wilson. A general framework for assembly planning: the motion space approach. In *Proc. ACM Symposium on Computational Geometry*, pages 9–18, 1998.
- [310] D. Halperin and M. Sharir. A near-quadratic algorithm for planning the motion of a polygon in a polygonal environment. *Discrete Comput. Geom.*, 16:121–134, 1986.
- [311] J. H. Halton. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numer. Math.*, 2:84–90, 1960.
- [312] J. M. Hammersley. Monte-Carlo methods for solving multivariable problems. *Ann. New York Acad. Sci.*, 86:844–874, 1960.
- [313] L. Han and N. M. Amato. A kinematics-based probabilistic roadmap method for closed chain systems. In B. R. Donald, K. M. Lynch, and D. Rus, editors, *Algorithmic and Computational Robotics: New Directions*, pages 233–246. A K Peters, Wellesley, MA, 2001.
- [314] B. Hannaford. Resolution-first scanning of multi-dimensional spaces. *CVGIP: Graphical Models and Image Processing*, 55:359–369, 1993.
- [315] J. C. Harsanyi. Games with incomplete information played by Bayesian players. *Management Science*, 14(3):159–182, November 1967.
- [316] R. S. Hartenberg and J. Denavit. A kinematic notation for lower pair mechanisms based on matrices. *J. Applied Mechanics*, 77:215–221, 1955.
- [317] A. Hatcher. *Algebraic Topology*. Cambridge University Press, 2002.

- [318] G. Heinzinger, P. Jacobs, J. Canny, and B. Paden. Time-optimal trajectories for a robotic manipulator: A provably good approximation algorithm. In *IEEE Int. Conf. Robot. & Autom.*, pages 150–155, Cincinnati, OH, 1990.
- [319] J. A. Hendler and J. C. Sanborn. Planning and reaction in dynamic domains. In *Proc. DARPA Workshop on Knowledge-Based Planning Systems*, 1987.
- [320] O. Hernández-Lerma and J. B. Lasserre. Error bounds for rolling horizon policies in discrete-time markov control processes. *IEEE Trans. Autom. Control*, 35:1118–1124, 1990.
- [321] J. Hershberger and S. Suri. Efficient computation of Euclidean shortest paths in the plane. In *Proc. 34th Annual IEEE Sympos. Found. Comput. Sci.*, pages 508–517, 1995.
- [322] J. Hervé, P. Cucka, and R. Sharma. Qualitative visual control of a robot manipulator. In *Proc. of the DARPA Workshop on Image Understanding*, pages 895–908, 1991.
- [323] F. J. Hickernell. Lattice rules: How well do they measure up? In P. Bickel, editor, *Random and Quasi-Random Point Sets*, pages 109–166. Springer-Verlag, Berlin, 1998.
- [324] F. J. Hickernell, H. S. Hong, P. L’Ecuyer, and C. Lemieux. Extensible lattice sequences for quasi-monte carlo quadrature. *SIAM Journal on Scientific Computing*, 22:1117–1138, 2000.
- [325] K. W. Hipel, K. J. Radford, and L. Fang. Multiple participant-multiple criteria decision making. *IEEE Trans. Syst., Man, Cybern.*, 23(4):1184–1189, 1993.
- [326] H. Hirukawa and Y. Papegay. Motion planning of objects in contact by the silhouette algorithm. In *IEEE Int. Conf. Robot. & Autom.*, pages 722–729, April 2000.
- [327] Y.-C. Ho and K.-C. Chu. Team decision theory and information structures in optimal control problems-part I. In *IEEE Trans. Autom. Control*, pages 15–22, 1972.
- [328] J. G. Hocking and G. S. Young. *Topology*. Dover Publications, New York, NY, 1988.
- [329] C. Hoffman, J. Hopcroft, and M. Karasick. Towards implementing robust geometric computations. In *Fourth Symp. on Computational Geometry*, 1988.
- [330] R. L. Hoffman. Automated assembly in a csg domain. In *IEEE Int. Conf. Robot. & Autom.*, pages 210–215, 1989.

- [331] C. M. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann, San Mateo, CA, 1989.
- [332] C. Holleman and L. E. Kavraki. A framework for using the workspace medial axis in PRM planners. In *IEEE Int. Conf. Robot. & Autom.*, pages 1408–1413, 2000.
- [333] J. Hollerbach. Dynamic scaling of manipulator trajectories. Technical report, MIT A.I. Lab Memo 700, 1983.
- [334] J. Hollerbach. Dynamic scaling of manipulator trajectories. In *Proc. Amer. Contro. Conf.*, pages 752–756, 1983.
- [335] L. S. Homem de Mello and A. C. Sanderson. A correct and complete algorithm for the generation of mechanical assembly sequences. *IEEE Trans. Robot. & Autom.*, 7(2):228–240, 1991.
- [336] L. S. Homem de Mello and A. C. Sanderson. Representations of mechanical assembly sequences. *IEEE Trans. Robot. & Autom.*, 7(2):211–227, 1991.
- [337] J. Hopcroft, D. Joseph, and S. Whitesides. Movement problems for 2-dimensional linkages. In J. T. Schwartz, M. Sharir, and J. Hopcroft, editors, *Planning, Geometry, and Complexity of Robot Motion*, pages 282–329. Ablex Publishing Corporation, Norwood, NJ, 1987.
- [338] J. E. Hopcroft, J. T. Schwartz, and M. Sharir. On the complexity of motion planning for multiple independent objects: PSPACE-hardness of the “wareouseman’s problem”. *Int. J. Robot. Res.*, 3(4):76–88, 1984.
- [339] J. E. Hopcroft, J. D. Ullman, and R. Motwani. *Introduction to Automata Theory, Languages, and Automation*. Addison-Wesley, Reading, MA, 2000.
- [340] T. Horsch, F. Schwarz, and H. Tolle. Motion planning for many degrees of freedom : Random reflections at c-space obstacles. In *IEEE Int. Conf. Robot. & Autom.*, pages 3318–3323, San Diego, CA, April 1994.
- [341] D. Hsu, T. Jiang, J. Reif, and Z. Sun. The bridge test for sampling narrow passages with probabilistic roadmap planners. In *IEEE Int. Conf. Robot. & Autom.*, 2003.
- [342] D. Hsu, L. E. Kavraki, J.-C. Latombe, R. Motwani, and S. Sorkin. On finding narrow passages with probabilistic roadmap planners. In et al. P. Agarwal, editor, *Robotics: The Algorithmic Perspective*, pages 141–154. A.K. Peters, Wellesley, MA, 1998.
- [343] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock. Randomized kinodynamic motion planning with moving obstacles. In *The Fourth International Workshop on Algorithmic Foundations of Robotics*. 2000.

- [344] D. Hsu, J.-C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. *Int. J. Comput. Geom. & Appl.*, 4:495–512, 1999.
- [345] H. Hu and M. Brady. A Bayesian approach to real-time obstacle avoidance for a mobile robot. *Autonomous Robots*, 1(1):69–92, 1994.
- [346] H. Hu, M. Brady, and P. Probert. Coping with uncertainty in control and planning for a mobile robot. In *IEEE/RSJ Int. Workshop on Intelligent Robots and Systems*, pages 1025–1030, Osaka, Japan, November 1991.
- [347] Y.-R. Hu and A. A. Goldenberg. Dynamic control of multiple coordinated redundant robots. *IEEE Trans. Syst., Man, Cybern.*, 22(3):568–574, May/June 1992.
- [348] L. Huang and Y. Aloimonos. Relative depth from motion using normal flow: An active and purposive solution. Technical Report CAR-TR-535, Center for Automation Research, University of Maryland, 1991.
- [349] W. Huang. Optimal line-sweep-based decompositions for coverage algorithms. In *IEEE Int. Conf. Robot. & Autom.*, 2001.
- [350] D. F. Huber, O. Carmichael, and M. Hebert. 3D map reconstruction from range data. In *IEEE Int. Conf. Robot. & Autom.*, pages 891–897, 2000.
- [351] T. W. Hungerford. *Algebra*. Springer-Verlag, Berlin, 1984.
- [352] S. Hutchinson, G. D. Hager, and P. I. Corke. A tutorial on visual servo control. *IEEE Trans. Robot. & Autom.*, 12(5):651–670, October 1996.
- [353] S. A. Hutchinson and A. C. Kak. Applying uncertainty reasoning to planning sensing strategies in a robot work cell with multi-sensor capabilities. In *IEEE Int. Conf. Robot. & Autom.*, pages 129–134, 1989.
- [354] D. P. Huttenlocher, G. A. Klanderman, and W. J. Rucklidge. Comparing images using the Hausdorff distance. *IEEE Trans. Pattern Anal. Machine Intell.*, 15(9):850–863, 1993.
- [355] D. P. Huttenlocher, J. J. Noh, and W. J. Rucklidge. Tracking non-rigid objects in complex scenes. In *Proc. Int. Conf. on Computer Vision*, pages 93–101, 1993.
- [356] Y. K. Hwang and N. Ahuja. Gross motion planning—A survey. *ACM Computing Surveys*, 24(3):219–291, September 1992.
- [357] Y. K. Hwang and N. Ahuja. A potential field approach to path planning. *IEEE Trans. Robot. & Autom.*, 8(1):23–32, February 1992.

- [358] C. Icking, G. Rote, E. Welzl, and C.-K. Yap. Shortest paths for line segments. *Algorithmica*, 10:182–200, 1992.
- [359] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.
- [360] R. Isaacs. *Differential Games*. Wiley, New York, NY, 1965.
- [361] A. Isidori. *Nonlinear Control Systems*. Springer-Verlag, Berlin, 1989.
- [362] P. Isto. Constructing probabilistic roadmaps with powerful local planning and path optimization. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 2323–2328, 2002.
- [363] j. Cortés, T. Simeon, and J.-P. Laumond. A random loop generator for planning the motions of closed kinematic chains using prm methods. In *IEEE Int. Conf. Robot. & Autom.*, 2002.
- [364] H. Jacob, S. Feder, and J. Slotine. Real-time path planning using harmonic potential functions in dynamic environment. In *IEEE Int. Conf. Robot. & Autom.*, pages 874–881, 1997.
- [365] P. Jacobs and J. Canny. Planning smooth paths for mobile robots. In *IEEE Int. Conf. Robot. & Autom.*, pages 2–7, 1989.
- [366] P. Jacobs, J. P. Laumond, and M. Taix. Efficient motion planners for non-holonomic mobile robots. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 1229–1235, 1991.
- [367] W. Jang and Z. Bien. Feature-based visual servoing of an eye-in-hand robot with improved tracking performance. In *IEEE Int. Conf. Robot. & Autom.*, pages 2254–3360, 1991.
- [368] S. T. Jones. Solving problems involving variable terrain, part I: A general algorithm. *BYTE*, 5(2), 1980.
- [369] D. Jordan and M. Steiner. Configuration spaces of mechanical linkages. *Discrete and Computational Geometry*, 22:297–315, 1999.
- [370] D. A. Joseph and W. H. Plantiga. On the complexity of reachability and motion planning questions. In *Proc. ACM Symp. on Comp. Geom.*, pages 62–66, 1985.
- [371] S. Kagami, J. Kuffner, K. Nishiwaki, and K. Okada M. Inaba. Humanoid arm motion planning using stereo vision and rrt search. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, 2003.

- [372] M. Kahn and A. W. Marshall. Methods for reducing sample size in monte-carlo computations. *Oper. Res.*, 1:263–278, 1953.
- [373] M. E. Kahn and B. E. Roth. The near minimum-time control of open-loop articulated kinematic chains. *Trans. ASME J. Dyn. Sys., Meas., & Contr.*, 93(3):164–172, 1971.
- [374] K. Kakusho, T. Kitahashi, K. Kondo, and J.-C. Latombe. Continuous purposive sensing and motion for 2d map building. In *Proc. IEEE Int. Conf. on Syst., Man, & Cybern.*, pages 1472–1477, 1995.
- [375] M. Kallmann, A. Aubel, T. Abaci, and D. Thalmann. Planning collision-free reaching motions for interactive object manipulation and grasping. *Eurographics*, 22(3), 2003.
- [376] M. H. Kalos and P. A. Whitlock. *Monte Carlo Methods*. Wiley, New York, NY, 1986.
- [377] S. Kambhampati and L. S. Davis. Multiresolution path planning for mobile robots. *IEEE J. of Robot. & Autom.*, 2(3):135–145, 1986.
- [378] I. Kamon and E. Rivlin. Sensory-based motion planning with global proofs. *IEEE Trans. Robot. & Autom.*, 13(6):814–822, December 1997.
- [379] I. Kamon, E. Rivlin, and E. Rimón. Range-sensor based navigation in three dimensions. In *IEEE Int. Conf. Robot. & Autom.*, 1999.
- [380] K. Kant and S. W. Zucker. Toward efficient trajectory planning: The path-velocity decomposition. *Int. J. Robot. Res.*, 5(3):72–89, 1986.
- [381] T. Karatas and F. Bullo. Randomized searches and nonlinear programming in trajectory planning. In *IEEE Conference on Decision and Control*, 2001.
- [382] L. Kauffman. *Knots and Applications*. World Scientific, River Edge, NJ, 1995.
- [383] L. Kavraki and J.-C. Latombe. Randomized preprocessing of configuration space for path planning. In *IEEE Int. Conf. Robot. & Autom.*, pages 2138–2139, 1994.
- [384] L. E. Kavraki. *Random Networks in Configuration Space for Fast Path Planning*. PhD thesis, Stanford University, 1994.
- [385] L. E. Kavraki. Computation of configuration-space obstacles using the Fast Fourier Transform. *IEEE Trans. Robot. & Autom.*, 11(3):408–413, 1995.
- [386] L. E. Kavraki. Geometry and the discovery of new ligands. In J.-P. Laumond and M. Overmars, editors, *Algorithms for Robotic Motion and Manipulation*, pages 435–445. A K Peters, Wellesley, MA, 1997.

- [387] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans. Robot. & Autom.*, 12(4):566–580, June 1996.
- [388] Y. Ke and J. O’Roarke. Lower bounds on moving a ladder in two and three dimensions. *Discrete Comput. Geom.*, 3:197–217, 1988.
- [389] K. Kedem, R. Livne, J. Pach, and M. Sharir. On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. *Discrete and Computational Geometry*, 1:59–71, 1986.
- [390] J. M. Keil. Polygon decomposition. In J. R. Sack and J. Urrutia, editors, *Handbook on Computational Geometry*. Elsevier Science Publishing, 2000.
- [391] J. F. Kenney and E. S. Keeping. *Mathematics of Statistics, Part 2, 2nd ed.* Van Nostrand, Princeton, NJ, 1951.
- [392] H. K. Khalil. *Nonlinear systems*. Macmillan Publishing, New York, NY, 2002.
- [393] O. Khatib. *Commande dynamique dans l’espace opérationnel des robots manipulateurs en présence d’obstacles*. PhD thesis, Ecole Nationale de la Statistique et de l’Administration Economique, France, 1980.
- [394] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *Int. J. Robot. Res.*, 5(1):90–98, 1986.
- [395] J. Kim and J. P. Ostrowski. Motion planning of aerial robot using rapidly-exploring random trees with dynamic constraints. In *IEEE Int. Conf. Robot. & Autom.*, 2003.
- [396] K. H. Kim and F. W. Roush. *Team Theory*. Ellis Horwood Limited, Chichester, England, 1987.
- [397] R. Kimmel, N. Kiryati, and A. M. Bruckstein. Multivalued distance maps for motion planning on surfaces with moving obstacles. *IEEE Trans. Robot. & Autom.*, 14(3):427–435, June 1998.
- [398] R. Kindel, D. Hsu, J.-C. Latombe, and S. Rock. Kinodynamic motion planning amidst moving obstacles. In *IEEE Int. Conf. Robot. & Autom.*, 2000.
- [399] C. L. Kinsey. *Topology of Surfaces*. Springer-Verlag, Berlin, 1993.
- [400] J. Kirman, K. Basye, and T. Dean. Sensor abstraction for control of navigation. In *IEEE Int. Conf. Robot. & Autom.*, pages 2812–2817, 1991.
- [401] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *ACM Symposium on Theory of Computing*, pages 599–608, May 1997.

- [402] D. Koditschek. Robot planning and control via potential functions. In O. Khatib, J. J. Craig, and T. Lozano-Pérez, editors, *The Robotics Review 1*. MIT Press, Cambridge, MA, 1989.
- [403] D. E. Koditschek. An approach to autonomous robot assembly. *Robotica*, 12:137–155, 1994.
- [404] Y. Koga, K. Kondo, J. Kuffner, and J.-C. Latombe. Planning motions with intentions. *Computer Graphics (SIGGRAPH'94)*, pages 395–408, 1994.
- [405] Y. Koga and J.-C. Latombe. On multi-arm manipulation planning. In *Proc. IEEE Int. Conf. on Rob. and Autom.*, pages 945–952, 1994.
- [406] K. Kondo. Motion planning with six degrees of freedom by multistrategic bidirectional heuristic free-space enumeration. *IEEE Trans. Robot. & Autom.*, 7(3):267–277, 1991.
- [407] P. Konkimalla. Efficient computation of optimal navigation functions for nonholonomic planning. Master's thesis, Iowa State University, Ames, IA, 1999.
- [408] R. Korf. Artificial intelligence search algorithms. In *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.
- [409] R. E. Korf. Search: A survey of recent results. In H. E. Shrobe, editor, *Exploring Artificial Intelligence: Survey Talks from the National Conference on Artificial Intelligence*. Moran Kaufmann, San Mateo, 1988.
- [410] K. Kotay, D. Rus, M. Vora, and C. McGray. The self-reconfiguring robotic molecule: Design and control algorithms. In P.K. Agarwal, L. Kavraki, and M. Mason, editors, *Robotics: The Algorithmic Perspective*. AK Peters, Natick, MA, 1998.
- [411] V. S. Kouikoglou and Y. A. Phillis. Discrete event modeling and optimization of unreliable production lines with random rates. *IEEE Trans. Robot. & Autom.*, 10(2):153–159, 1994.
- [412] V. G. Kountouris and H. E. Stephanou. Dynamic modularization and synchronization for intelligent robot coordination: The concept of functional time-dependency. In *IEEE Int. Conf. Robot. & Autom.*, pages 508–513, Sacramento, CA, April 1991.
- [413] K. Kozłowski, P. Dutkiewicz, and W. Wroblewski. *Modeling and Control of Robots (in Polish)*. Wydawnictwo Naukowe PWN, Warsaw, Poland, 2003.
- [414] B. H. Krogh. A generalized potential field approach to obstacle avoidance control. In *Proceedings of SME Conference on Robotics Research*, August 1984.

- [415] E. Krotkov and R. Bajcsy. Active vision for reliable ranging: Cooperating focus, stereo, and vergence. *Intl. J. of Computer Vision*, 11(2):187–203, October 1993.
- [416] E. Kruse, R. Gutschke, and F. M. Wahl. Efficient, iterative, sensor based 3-d map building using rating functions in configuration space. In *IEEE Int. Conf. Robot. & Autom.*, pages 1067–1072, 1996.
- [417] N. V. Krylov. *Controlled diffusion processes*. Springer-Verlag, Berlin, 1980.
- [418] J. J. Kuffner. *Autonomous Agents for Real-time Animation*. PhD thesis, Stanford University, 1999.
- [419] J. J. Kuffner and S. M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 995–1001, 2000.
- [420] P. R. Kumar and P. Varaiya. *Stochastic Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1986.
- [421] H. J. Kushner. Numerical methods for continuous control problems in continuous time. *SIAM Journal on Control and Optimization*, 28:999–1048, 1990.
- [422] K. N. Kutulakos, C. R. Dyer, and V. J. Lumelsky. Provable strategies for vision-guided exploration in three dimensions. In *IEEE Int. Conf. Robot. & Autom.*, pages 1365–1371, 1994.
- [423] H. Kwakernaak and R. Sivan. *Linear Optimal Control Systems*. Wiley, New York, NY, 1972.
- [424] A. Ladd and L. Kavraki. Motion planning for knot untangling. In *Proc. of the Workshop on the Algorithmic Foundations of Robotics*, Nice, France, December 2002.
- [425] G. Laffieriere and H. J. Sussman. Motion planning for controllable systems without drift. In *IEEE Int. Conf. Robot. & Autom.*, 1991.
- [426] G. Laffieriere and H. J. Sussman. A differential geometric approach to motion planning. Available from "<http://www.mth.pdx.edu/~gerardo/papers/>", 1998.
- [427] S. Lahiri. Coalitional fairness and distortion of utilities. *IEEE Trans. Syst., Man, Cybern.*, 21(5):1295–1298, Sep/Oct 1991.
- [428] F. Lamiriaux, D. Bonnafous, and C. Van Geem. Path optimization for non-holonomic systems: Application to reactive obstacle avoidance and path planning. In *IEEE Int. Conf. Robot. & Autom.*, pages 3099–3104, 2002.

- [429] F. Lamiroux and J.-P. Laumond. On the expected complexity of random path planning. In *IEEE Int. Conf. Robot. & Autom.*, pages 3306–3311, 1996.
- [430] A. S. Lapaugh. Recontamination does not help to search a graph. *Journal of the ACM*, 40(2):224–245, April 1993.
- [431] R. E. Larson. A survey of dynamic programming computational procedures. *IEEE Trans. Autom. Control*, 12(6):767–774, December 1967.
- [432] R. E. Larson. *State Increment Dynamic Programming*. Elsevier, New York, NY, 1968.
- [433] R. E. Larson and J. L. Casti. *Principles of Dynamic Programming, Part II*. Dekker, New York, NY, 1982.
- [434] R. E. Larson and W. G. Keckler. Optimum adaptive control in an unknown environment. *IEEE Trans. Autom. Control*, 13(4):438–439, August 1968.
- [435] A. Lasota and M. C. Mackey. *Chaos, Fractals, and Noise: Stochastic Aspects of Dynamics, 2nd Edition*. Springer-Verlag, Berlin, 1995.
- [436] J.-C. Latombe. A fast path planner for a car-like indoor mobile robot. In *Proc. Am. Assoc. Artif. Intell.*, pages 659–665, 1991.
- [437] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, 1991.
- [438] J.-C. Latombe, A. Lazanas, and S. Shekhar. Robot motion planning with uncertainty in control and sensing. *Artif. Intell.*, 52:1–47, 1991.
- [439] J.-P. Laumond. Feasible trajectories for mobile robots with kinematic and environment constraints. In *ECAI*, 1986.
- [440] J.-P. Laumond. Trajectories for mobile robots with kinematic and environment constraints. In *Proc. of International Conference on Intelligent Autonomous Systems*, pages 346–354, 1986.
- [441] J.-P. Laumond. Finding collision-free smooth trajectories for a nonholonomic mobile robot. In *Proc. Int. Joint Conf. on Artif. Intell.*, pages 1120–1123, 1987.
- [442] J.-P. Laumond. Singularities and topological aspects in nonholonomic motion planning. In Z. Li and J. F. Canny, editors, *Nonholonomic Motion Planning*, pages 149–200. Kluwer Academic Publishers, Boston, MA, 1993.
- [443] J.-P. Laumond, P. E. Jacobs, M. Taix, and R. M. Murray. A motion planner for nonholonomic mobile robots. *IEEE Trans. Robot. & Autom.*, 10(5):577–593, October 1994.

- [444] J. P. Laumond, S. Sekhavat, and F. Lamiroux. Guidelines in nonholonomic motion planning for mobile robots. In J.-P. Laumond, editor, *Robot Motion Planning and Control*, pages 1–53. Springer-Verlag, Berlin, 1998.
- [445] J. P. Laumond and T. Siméon. Motion planning for a two degrees of freedom mobile robot with towing. Technical Report 89-148, Laboratoire d'Analyse et d'Architecture des Systemes / Centre National de la Recherche Scientifique, Toulouse, France, 1989.
- [446] J. P. Laumond, T. Siméon, R. Chatila, and G. Giralt. Trajectory planning and motion control of mobile robots. In *Proceedings of IUTAM/IFAC Symposium*, pages 351–366, 1988.
- [447] J. P. Laumond and P. Souères. Metric induced by the shortest paths for a car-like robot. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 1299–1303, 1993.
- [448] S. M. LaValle. *A Game-Theoretic Framework for Robot Motion Planning*. PhD thesis, University of Illinois, Urbana, IL, July 1995.
- [449] S. M. LaValle. Rapidly-exploring random trees: A new tool for path planning. TR 98-11, Computer Science Dept., Iowa State University, Oct. 1998.
- [450] S. M. LaValle. Robot motion planning: A game-theoretic foundation. *Algorithmica*, 26(3):430–465, 2000.
- [451] S. M. LaValle. From dynamic programming to RRTs: Algorithmic design of feasible trajectories. In A. Bicchi, H. I. Christensen, and D. Prattichizzo, editors, *Control Problems in Robotics*, pages 19–37. Springer-Verlag, Berlin, 2002.
- [452] S. M. LaValle and M. S. Branicky. On the relationship between classical grid search and probabilistic roadmaps. In J.-D. Boissonat, J. Burdick, K. Y. Goldberg, and S. A. Hutchinson, editors, *Algorithmic Foundations of Robotics*. Springer-Verlag, Berlin, 2003. To appear.
- [453] S. M. LaValle, M. S. Branicky, and S. R. Lindemann. On the relationship between classical grid search and probabilistic roadmaps. *International Journal of Robotics Research (to appear)*, 24, 2004.
- [454] S. M. LaValle, P. Finn, L. Kavraki, and J.-C. Latombe. Efficient database screening for rational drug design using pharmacophore-constrained conformational search. In *Proc. RECOMB (Annual International Conference on Molecular Biology)*, pages 250–259, 1999.
- [455] S. M. LaValle, P. Finn, L. Kavraki, and J.-C. Latombe. A randomized kinematics-based approach to pharmacophore-constrained conformational

- search and database screening. *J. Computational Chemistry*, 21(9):731–747, 2000.
- [456] S. M. LaValle, H. H. González-Baños, C. Becker, and J.-C. Latombe. Motion strategies for maintaining visibility of a moving target. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 731–736, 1997.
- [457] S. M. LaValle and J. Hinrichsen. Visibility-based pursuit-evasion: The case of curved environments. *IEEE Transactions on Robotics and Automation*, 17(2):196–201, April 2001.
- [458] S. M. LaValle and S. A. Hutchinson. Game theory as a unifying structure for a variety of robot tasks. In *Proc. IEEE Int'l Symp. on Intelligent Control*, pages 429–434, August 1993.
- [459] S. M. LaValle and S. A. Hutchinson. Optimal motion planning for multiple robots having independent goals. In *Proc. IEEE Int'l Conf. Robot. & Autom.*, pages 2847–2852, April 1996.
- [460] S. M. LaValle and S. A. Hutchinson. An objective-based framework for motion planning under sensing and control uncertainties. *International Journal of Robotics Research*, 17(1):19–42, January 1998.
- [461] S. M. LaValle and S. A. Hutchinson. Optimal motion planning for multiple robots having independent goals. *IEEE Trans. on Robotics and Automation*, 14(6):912–925, December 1998.
- [462] S. M. LaValle and P. Konkimalla. Algorithms for computing numerical optimal feedback motion strategies. *International Journal of Robotics Research*, 20(9):729–752, September 2001.
- [463] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 473–479, 1999.
- [464] S. M. LaValle and J. J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In *Workshop on the Algorithmic Foundations of Robotics*, 2000.
- [465] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. *International Journal of Robotics Research*, 20(5):378–400, May 2001.
- [466] S. M. LaValle and J. J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In B. R. Donald, K. M. Lynch, and D. Rus, editors, *Algorithmic and Computational Robotics: New Directions*, pages 293–308. A K Peters, Wellesley, MA, 2001.

- [467] S. M. LaValle, D. Lin, L. J. Guibas, J.-C. Latombe, and R. Motwani. Finding an unpredictable target in a workspace with obstacles. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 737–742, 1997.
- [468] S. M. LaValle and R. Sharma. On motion planning in changing, partially-predictable environments. *International Journal of Robotics Research*, 16(6):775–805, December 1997.
- [469] S. M. LaValle, B. Simov, and G. Slutzki. An algorithm for searching a polygonal region with a flashlight. In *Proc. ACM Annual Symposium on Computational Geometry*, 2000.
- [470] S. M. LaValle, B. Simov, and G. Slutzki. An algorithm for searching a polygonal region with a flashlight. *International Journal of Computational Geometry and Applications*, 12(1-2):87–113, 2002.
- [471] S. Lavallée, J. Troccaz, L. Gaborit, A. L. Benabid P. Cinquin, and D. Hoffman. Image-guided operating robot: A clinical application in stereotactic neurosurgery. In R. H. Taylor, S. Lavallée, G. C. Burdea, and R. Mösges, editors, *Computer-Integrated Surgery*, pages 343–351. Mit Press, Cambridge, MA, 1996.
- [472] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, New York, NY, 1976.
- [473] A. R. Leach and I. D. Kuntz. Conformational analysis of flexible ligands in macromolecular receptor sites. *J. Computational Chemistry*, 13(6):730–748, 1992.
- [474] D. T. Lee and R. L. Drysdale. Generalization of voronoi diagrams in the plane. *SIAM J. Computing*, 10:73–87, 1981.
- [475] J.-H. Lee, S. Y. Shin, and K.-Y. Chwa. Visibility-based pursuit-evasions in a polygonal room with a door. In *Proc. ACM Symp. on Comp. Geom.*, 1999.
- [476] M. H. Lee. *Intelligent Robotics*. Halsted Press, New York, 1989.
- [477] T. T. Lee and C. L. Shih. An approach for robot dynamic motion planning with control torques and obstacles constraints. In *Conf. of IEEE Industrial Electronics Society*, pages 433–438, Pacific Grove, CA, November 1990.
- [478] D. H. Lehmer. Mathematical methods in large-scale computing units. In *Proc. 2nd Sympos. on Large-Scale Digital Computing Machinery*, pages 141–146. Harvard University Press, 1951.

- [479] J. Lengyel, M. Reichert, B. R. Donald, and D. P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. *Computer Graphics*, 24(4):327–335, August 1990.
- [480] D. Leven and M. Sharir. An efficient and simple motion planning algorithm for a ladder moving in a 2-dimensional space amidst polygonal barriers. *J. Algorithms*, 8:192–215, 1987.
- [481] D. Leven and M. Sharir. Planning a purely translational motion for a convex object in two-dimensional space using generalized voronoi diagrams. *Discrete Comput. Geom.*, 2:9–31, 1987.
- [482] P. Leven and S. Hutchinson. Real-time motion planning in changing environments. In *Proc. International Symposium on Robotics Research*, 2000.
- [483] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D scanning of large statues. In *SIG-GRAPH*, pages 24–28, 2000.
- [484] A. D. Lewis and R. M. Murray. Configuration controllability of simple mechanical control systems. *SIAM Journal on Control and Optimization*, 35(3):766–790, 1997.
- [485] F. Leymarie and M. D. Levine. Tracking deformable objects in the plane using an active contour model. *IEEE Trans. Pattern Anal. Machine Intell.*, 15(6):617–631, 1993.
- [486] T.-Y. Li and Y.-C. Shie. An incremental learning approach to motion planning with roadmap management. In *IEEE Int. Conf. Robot. & Autom.*, 2002.
- [487] Z. Li and J. F. Canny. Robot motion planning with nonholonomic constraints. Technical report, Electronics Research Laboratory, University of California, February 1989.
- [488] Z. Li and J. F. Canny. *Nonholonomic Motion Planning*. Kluwer Academic Publishers, Boston, MA, 1993.
- [489] Z. Li, J. F. Canny, and S. S. Sastry. On motion planning for dextrous manipulation, part i: The problem formulation. In *IEEE Int. Conf. Robot. & Autom.*, pages 775–780, 1989.
- [490] J.-M. Lien, S. L. Thomas, and N. M. Amato. A general framework for sampling on the medial axis of the free space. In *IEEE Int. Conf. Robot. & Autom.*, 2003.

- [491] C.-F. Lin and W.-H. Tsai. Motion planning for multiple robots with multi-mode operations via disjunctive graphs. *Robotica*, 9:393–408, 1990.
- [492] M. C. Lin and J. F. Canny. Efficient algorithms for incremental distance computation. In *IEEE Int. Conf. Robot. & Autom.*, 1991.
- [493] M. C. Lin, D. Manocha, J. Cohen, and S. Gottschalk. Collision detection: Algorithms and applications. In J.-P. Laumond and M. Overmars, editors, *Algorithms for Robotic Motion and Manipulation*, pages 129–142. A K Peters, Wellesley, MA, 1997.
- [494] Z. Lin, V. Zeman, and R. V. Patel. On-line robot trajectory planning for catching a moving object. In *IEEE Int. Conf. Robot. & Autom.*, pages 1726–1731, 1989.
- [495] S. R. Lindemann and S. M. LaValle. Incremental low-discrepancy lattice methods for motion planning. In *Proc. IEEE International Conference on Robotics and Automation*, pages 2920–2927, 2003.
- [496] S. R. Lindemann and S. M. LaValle. Current issues in sampling-based motion planning. In P. Dario and R. Chatila, editors, *Proc. Eighth Int'l Symp. on Robotics Research*. Springer-Verlag, Berlin, 2004. To appear.
- [497] S. R. Lindemann and S. M. LaValle. Incrementally reducing dispersion by increasing Voronoi bias in RRTs. In *Proc. IEEE International Conference on Robotics and Automation*, 2004. Under review.
- [498] S. R. Lindemann and S. M. LaValle. Steps toward derandomizing RRTs. In *IEEE Fourth International Workshop on Robot Motion and Control*, 2004. Under review.
- [499] A. Lingas. The power of non-rectilinear holes. In *Proc. 9th Internat. Collog. Automata Lang. Program.: Lecture Notes in Computer Science 140*, pages 369–383. Springer-Verlag, 1982.
- [500] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal of Applied Mathematics*, 36:177–189, 1979.
- [501] C. K. Liu and Z. Popovic. Synthesis of complex dynamic character motion from simple animations. In *SIGGRAPH*, 2002.
- [502] J.-S. Liu, L.-S. Wang, and L.-S. Tsai. A nonlinear programming approach to nonholonomic motion planning with obstacle avoidance. In *IEEE Int. Conf. Robot. & Autom.*, pages 70–75, 1994.
- [503] T. Lozano-Pérez. Automatic planning of manipulator transfer movements. *IEEE Trans. Syst., Man, Cybern.*, 11(10):681–698, 1981.

- [504] T. Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Trans. on Comput.*, C-32(2):108–120, 1983.
- [505] T. Lozano-Pérez. A simple motion-planning algorithm for general robot manipulators. *IEEE J. of Robot. & Autom.*, RA-3(3):224–238, Jun 1987.
- [506] T. Lozano-Pérez, M. T. Mason, and R. H. Taylor. Automatic synthesis of fine-motion strategies for robots. *Int. J. Robot. Res.*, 3(1):3–24, 1984.
- [507] T. Lozano-Pérez and M. A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, 1979.
- [508] F. Lu and E. E. Milios. Optimal global pose estimation for consistent sensor data registration. In *IEEE Int. Conf. Robot. & Autom.*, 1995.
- [509] L. Lu and S. Akella. Folding cartons with fixtures: A motion planning approach. *IEEE Trans. Robot. & Autom.*, 16(4):346–356, Aug 2000.
- [510] P. Lu and J. M. Hanson. Entry guidance for the X-33 vehicle. *J. Spacecraft and Rockets*, 35(3):342–349, 1998.
- [511] S. H. Lu and P. R. Kumar. Distributed scheduling based on due dates and buffer priorities. *IEEE Trans. Autom. Control*, 36(12):1406–1416, 1991.
- [512] A. De Luca, G. Oriolo, and C. Samson. Feedback control of a nonholonomic car-like robot. In J.-P. Laumond, editor, *Robot Motion Planning and Control*, pages 171–253. Springer-Verlag, Berlin, 1998.
- [513] J. Luh and C. S. Lin. Optimum path planning for mechanical manipulators. *J. Dyn. Sys. Meas. Contr.*, 102:142–151, 1981.
- [514] J. Luh and M. Walker. Minimum-time along the path for a mechanical arm. In *Proc. IEEE Conf. Decision and Contr.*, pages 755–759, 1977.
- [515] V. Lumelsky and T. Skewis. Incorporating range sensing in the robot navigation function. *IEEE Trans. Syst., Man, Cybern.*, 20(5):1058–1069, 1990.
- [516] V. Lumelsky and S. Tiwari. An algorithm for maze searching with azimuth input. In *IEEE Int. Conf. Robot. & Autom.*, pages 111–116, 1994.
- [517] V. J. Lumelsky and T. Skewis. A paradigm for incorporating vision in the robot navigation function. In *IEEE Int. Conf. Robot. & Autom.*, pages 734–739, 1988.
- [518] V. J. Lumelsky and A. A. Stepanov. Path planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2:403–430, 1987.

- [519] K. M. Lynch. Controllability of a planar body with unilateral thrusters. *IEEE Trans. on Automatic Control*, 44(6):1206–1211, 1999.
- [520] K. M. Lynch and M. T. Mason. Pulling by pushing, slip with infinite friction, and perfectly rough surfaces. In *IEEE Int. Conf. Robot. & Autom.*, pages 1:745–751, Atlanta, GA, May 1993.
- [521] K. M. Lynch and M. T. Mason. Pulling by pushing, slip with infinite friction, and perfectly rough surfaces. *Int. J. Robot. Res.*, 14(2):174–183, 1995.
- [522] K. M. Lynch and M. T. Mason. Stable pushing: Mechanics, controllability, and planning. In *Algorithmic Foundations of Robotics*. A. K. Peters, Boston, 1995.
- [523] K. M. Lynch and M. T. Mason. Stable pushing: Mechanics, controllability, and planning. *Int. J. Robot. Res.*, 15(6):533–556, 1996.
- [524] K. M. Lynch, N. Shiroma, H. Arai, and K. Tanie. Collision free trajectory planning for a 3-dof robot with a passive joint. *Int. J. Robot. Res.*, 19(12):1171–1184, 2000.
- [525] N. Mahadevamurty, T.-C. Tsao, and S. Hutchinson. Multi-rate analysis and design of visual feedback digital servo control systems. *Trans. ASME J. Dyn. Sys., Meas., & Contr.*, pages 45–55, March 1994.
- [526] I. M. Makarov, T. M. Vinogradskaya, A. A. Rubchinsky, and V. B. Sokolov. *The Theory of Choice and Decision Making*. Mir Publishers, Moscow, 1987.
- [527] F. Makedon and I. H. Sudborough. Minimizing width in linear layouts. In *Proc. 10th ICALP, Lecture Notes in Computer Science 154*, pages 478–490. Springer-Verlag, 1983.
- [528] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for on-line problems. In *Proc. 20th Annu. ACM Sympos. Theory Comput.*, pages 322–333, 1988.
- [529] D. Manocha and J. Canny. Real time inverse kinematics of general 6R manipulators. In *IEEE Int. Conf. Robot. & Autom.*, pages 383–389, Nice, May 1992.
- [530] T. M. Mansell. A method for planning given uncertain and incomplete information. In *Uncertainty in Artificial Intelligence*, pages 350–358, Washington, D.C., July 1993.
- [531] E. Marchand and F. Chaumette. Controlled camera motions for scene reconstruction and exploration. In *Proc. IEEE Conf. on Comp. Vision and Patt. Recog.*, pages 169–176, 1996.

- [532] A. Marigo, B. Piccoli, and A. Bicchi. Reachability analysis for a class of quantized control systems. In *Proc. IEEE Conf. on Decision and Control*, 2000.
- [533] M. T. Mason. Compliance and force control for computer controlled manipulators. In M. Brady *et al.*, editor, *Robot Motion: Planning and Control*, pages 373–404. MIT Press, Cambridge, MA, 1982.
- [534] M. T. Mason. Automatic planning of fine motions: Correctness and completeness. In *Proc. IEEE Int. Conf. Robot. & Autom.*, pages 492–503, 1984.
- [535] M. T. Mason. Mechanics and planning of manipulator pushing operations. *Int. J. Robot. Res.*, 5(3):53–71, 1986.
- [536] M. T. Mason. *Mechanics of Robotic Manipulation*. MIT Press, Boston, 2001.
- [537] A. Massoud. Robot navigation using the vector potential approach. In *IEEE Int. Conf. Robot. & Autom.*, pages 1:805–811, 1993.
- [538] J. Matousek. *Geometric Discrepancy*. Springer-Verlag, Berlin, 1999.
- [539] J. Matousek and J. Nešetřil. *Invitation to Discrete Mathematics*. Oxford Press, 1998.
- [540] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensional equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, January 1998.
- [541] L. Matthies and A. Elfes. Integration of sonar and stereo range data using a grid-based representation. In *IEEE Int. Conf. Robot. & Autom.*, pages 727–733, 1988.
- [542] J. Maver and R. Bajcsy. Occlusions as a guide for planning the next view. *IEEE Trans. Pattern Anal. Machine Intell.*, 15(5):417–433, May 1993.
- [543] E. Mazer, J. M. Ahuactzin, and P. Bessière. The Ariadne’s clew algorithm. *J. Artificial Intell. Res.*, 9:295–316, November 1998.
- [544] E. Mazer, G. Talbi, J. M. Ahuactzin, and P. Bessière. The Ariadne’s clew algorithm. In *Proc. Int. Conf. of Society of Adaptive Behavior*, Honolulu, 1992.
- [545] G. P. McCormick and K. Ritter. Alternative proofs of the convergence properties of the conjugate-gradient method. *JOTA*, 13(5):497–518, 1975.
- [546] S. McMillan, P. Sadayappan, and D. E. Orin. Parallel dynamic simulation of multiple manipulator systems: Temporal versus spatial methods. *IEEE Trans. Syst., Man, Cybern.*, 24(7):982–990, July 1994.

- [547] N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson, and C. H. Papadimitriou. The complexity of searching a graph. *Journal of the ACM*, 35(1):18–44, January 1988.
- [548] E. B. Meier and A. W. Bryson. An efficient algorithm for time optimal control of a two-link manipulator. In *Proc. AIAA conf. Guidance Control*, Monterey, CA, 1987.
- [549] A. C. Meng. Dynamic motion replanning for unexpected obstacles. In *IEEE Int. Conf. Robot. & Autom.*, pages 1848–1849, 1988.
- [550] J.-P. Merlet. *Parallel Robots*. Kluwer Academic Publishers, 2000.
- [551] N. C. Metropolis and S. M. Ulam. The Monte-Carlo method. *J. Amer. Stat. Assoc.*, 44:335–341, 1949.
- [552] P. Milgram and F. Kishino. A taxonomy of mixed reality visual displays. E77-D(12):1321–1329, December 1994.
- [553] R. J. Milgram and J. C. Trinkle. The geometry of configuration spaces for closed kinematic chains with spherical joints. *Homology, Homotopy, and Applications*.
- [554] D. A. Miller and S. W. Zucker. Copositive-plus lemke algorithm solves polymatrix games. *Oper. Res. Lett.*, 10:285–290, 1991.
- [555] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Separators for sphere-packings and nearest neighbor graphs. *Journal of the ACM*, 44(1):1–29, January 1997.
- [556] B. Mirtich. V-Clip: Fast and robust polyhedral collision detection. Technical Report TR97-05, Mitsubishi Electronics Research Laboratory, 1997.
- [557] B. Mirtich, Y. Zhuang, K. Goldberg, J. Craig, R. Zanutta, B. Carlisle, and J. Canny. Estimating pose statistics for robotic part feeders. In *IEEE Int. Conf. Robot. & Autom.*, pages 1140–1146, 1996.
- [558] B. Mishra. *Algorithmic Algebra*. Springer-Verlag, New York, NY, 1993.
- [559] B. Mishra. Computational real algebraic geometry. In J. E. Goodman and J. O’Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 537–556. CRC Press, New York, 1997.
- [560] J. S. B. Mitchell. *Planning Shortest Paths*. PhD thesis, Stanford University, 1986.
- [561] J. S. B. Mitchell. An algorithmic approach to some problems in terrain navigation. In D. Kapur and J. L. Mundy, editors, *Geometric Reasoning*, pages 171–201. MIT Press, Cambridge, MA, 1988.

- [562] J. S. B. Mitchell. Shortest paths among obstacles in the plane. *Int. J. Comput. Geom. & Appl.*, 6(3):309–332, 1996.
- [563] J. S. B. Mitchell. Shortest paths and networks. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 445–466. CRC Press, New York, 1997.
- [564] R. A. Mitchell. Error estimates arising from certain pseudorandom sequences in a quasi-random search method. *Mathematics of Computation*, 55(191):289–297, 1990.
- [565] J. Miura and Y. Shirai. Planning of vision and motion for a mobile robot using a probabilistic model of uncertainty. In *IEEE/RSJ Int. Workshop on Intelligent Robots and Systems*, pages 403–408, Osaka, Japan, May 1991.
- [566] J. Miura and Y. Shirai. Visual-motion planning with uncertainty. In *IEEE Int. Conf. Robot. & Autom.*, pages 1772–1777, Nice, France, May 1992.
- [567] B. Monien and I. H. Sudborough. Min cut is NP-complete for edge weighted graphs. *Theoretical Computer Science*, 58:209–229, 1988.
- [568] M. E. Mortenson. *Geometric Modeling*. Wiley, New York, NY, 1985.
- [569] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- [570] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1994.
- [571] R. M. Murray. Nilpotent bases for a class of nonintegrable distributions with applications to trajectory generation for nonholonomic systems. *Math. Contr. Sig. Syst.*, 7:58–75, 1994.
- [572] R. M. Murray, M. Rathinam, and W. M. Sluis. Differential flatness of mechanical control systems. In *Proc. ASME International Congress and Exposition*, 1995.
- [573] R. M. Murray and S. Sastry. Nonholonomic motion planning: Steering using sinusoids. *Trans. Automatic Control*, 38(5):700–716, 1993.
- [574] T. Nagata, K. Honda, and Y. Teramoto. Multirobot plan generation in a continuous domain: Planning by use of plan graph and avoiding collisions among robots. *IEEE Trans. Robot. & Autom.*, 4(1):2–13, February 1988.
- [575] Y. Nakamura. *Advanced Robotics: Redundancy and Optimization*. Addison-Wesley, Reading, MA, 1991.

- [576] Y. Nakamura and R. Mukherjee. Nonholonomic path planning of space robots. In *IEEE Int. Conf. Robot. & Autom.*, pages 1050–1055, 1989.
- [577] B. K. Natarajan. The complexity of fine motion planning. *Int. J. Robot. Res.*, 7(2):36–42, 1988.
- [578] W. S. Newman and N. Hogan. High speed robot control and obstacle avoidance using dynamic potential functions. In *IEEE Int. Conf. Robot. & Autom.*, pages 14–24, 1987.
- [579] H. Niederreiter. *Random Number Generation and Quasi-Monte-Carlo Methods*. Society for Industrial and Applied Mathematics, Philadelphia, USA, 1992.
- [580] H. Niederreiter and L. Kuipers. *Uniform Distribution of Sequences*. John Wiley and Sons, New York, 1974.
- [581] H. Niederreiter and C. P. Xing. Nets, (t,s)-sequences, and algebraic geometry. In P. Hellekalek and G. Larcher, editors, *Random and Quasi-Random Point Sets, Lecture Notes in Statistics, Vol. 138*, pages 267–302. Springer-Verlag, Berlin, 1998.
- [582] N. J. Nilsson. A mobile automaton: An application of artificial intelligence techniques. In *1st International Conference on Artificial Intelligence*, pages 509–520, 1969.
- [583] N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.
- [584] N. J. Nilsson. Shakey the robot. Technical Report TR 223, SRI International, 1984.
- [585] H. Noroborio and J. Hashime. A feasible collision-free and deadlock-free path-planning algorithm in a certain workspace where multiple robots move flexibly. In *IEEE/RSJ Int. Workshop on Intelligent Robots and Systems*, pages 1074–1079, 1991.
- [586] S. Ntafos. Watchman routes under limited visibility. *Computational Geometry: Theory and Applications*, 1:149–170, 1992.
- [587] P. A. O'Donnell and T. Lozano-Pérez. Deadlock-free and collision-free coordination of two robot manipulators. In *IEEE Int. Conf. Robot. & Autom.*, pages 484–489, 1989.
- [588] C. Ó. Dúnlaing. Motion planning with inertial constraints. *Algorithmica*, 2(4):431–475, 1987.

- [589] C. Ó. Dúnlaing, M. Sharir, and C. K. Yap. Retraction: A new approach to motion planning. In J. T. Schwartz, M. Sharir, and J. Hopcroft, editors, *Planning, Geometry, and Complexity of Robot Motion*, pages 193–213. Ablex Publishing Corporation, Norwood, NJ, 1987.
- [590] C. O’Dunlaing and C. K. Yap. A retraction method for planning the motion of a disc. *Journal of Algorithms*, 6:104–111, 1982.
- [591] G. J. Olsder and G. P. Papavassilopoulos. A Markov chain game with dynamic information. *J. Optimization Theory & Appl.*, 59(3):467–486, December 1988.
- [592] B. O’Neill. *Elementary Differential Geometry*. Academic Press, New York, NY, 1966.
- [593] B. J. Oommen, N. Andrade, and S. S. Iyengar. Trajectory planning of robot manipulators in noisy work spaces using stochastic automata. *Int. J. Robot. Res.*, 10(2):135–148, April 1991.
- [594] J. B. Oommen, S. S. Iyengar, N. S. V. Rao, and R. L. Kashyap. Robot navigation in unknown terrains using learned visibility graphs. part I: The disjoint convex obstacle case. *IEEE J. of Robot. & Autom.*, 3(6):672–681, 1987.
- [595] J. O’Roarke. Visibility. In J. E. Goodman and J. O’Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 467–479. CRC Press, New York, 1997.
- [596] J. O’Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, New York, NY, 1987.
- [597] J. O’Rourke. Combinatorics of visibility and illumination: 37 open problems. Technical Report 050, Department of Computer Science, Smith College, July 1996.
- [598] M. Overmars. A random approach to motion planning. Technical report, Dept. Computer Science, Utrecht University, Utrecht, The Netherlands, October 1992.
- [599] M. H. Overmars and J. van Leeuwen. Dynamic multidimensional data structures based on Quad- and K-D trees. *Acta Informatica*, 17:267–285, 1982.
- [600] A. B. Owen. Monte Carlo variance of scrambled equidistribution quadrature. *SIAM J. Numer. Anal.*, 34(5):1884–1910, 1997.
- [601] G. Owen. *Game Theory*. Academic Press, New York, NY, 1982.

- [602] B. Paden, A. Mees, and M. Fisher. Path planning using a jacobian-based freespace generation algorithm. In *IEEE Int. Conf. Robot. & Autom.*, pages 1732–1737, 1989.
- [603] L. A. Page and A. C. Sanderson. Robot motion planning for sensor-based control with uncertainties. In *IEEE Int. Conf. Robot. & Autom.*, pages 1333–1340, 1995.
- [604] D. W. Paglieroni. Distance transforms: Properties and machine vision. *54(1):56–74*, 1992.
- [605] S. Pandya and S. A. Hutchinson. A case-based approach to robot motion planning. In *Proc. IEEE Int. Conf. on Syst., Man, & Cybern.*, pages 492–497, Chicago, October 1992.
- [606] C. H. Papadimitriou. An algorithm for shortest-path planning in three dimensions. *Information Processing Letters*, 20(5):259–263, 1985.
- [607] C. H. Papadimitriou. Games against nature. *Journal of Computer and System Sciences*, 31:288–301, 1985.
- [608] C. H. Papadimitriou and J. N. Tsitsiklis. The complexity of markov decision processes. *Math. of Oper. Res.*, 12(3):441–450, August 1987.
- [609] N. P. Papanikolopoulos and P. K. Khosla. Shared and traded telerobotic visual control. In *IEEE Int. Conf. Robot. & Autom.*, pages 878–883, 1992.
- [610] N. P. Papanikolopoulos, P. K. Khosla, and T. Kanade. Visual tracking of a moving target by a camera mounted on a robot: A combination of control and vision. *IEEE Trans. Robot. & Autom.*, 9(1):14–35, February 1993.
- [611] A. Papantonopoulou. *Algebra: Pure and Applied*. Prentice Hall, Englewood Cliffs, NJ, 2002.
- [612] G. P. Papavassilopoulos. Cooperative outcomes of dynamic stochastic Nash games. In *IEEE Conf. Decision & Control*, pages 186–191, Tampa, FL, December 1989.
- [613] G. P. Papavassilopoulos. Learning algorithms for repeated bimatrix Nash games with incomplete information. *J. Optimization Theory & Appl.*, 62(3):467–488, September 1989.
- [614] S.-M. Park, J.-H. Lee, and K.-Y. Chwa. Visibility-based pursuit-evasion in a polygonal region by a searcher. Technical Report CS/TR-2001-161, KAIST, Dept. of Computer Science, Korea, January 2001.

- [615] L. E. Parker. Cooperative motion control for multi-target observation. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 1591–1598, 1998.
- [616] T. D. Parsons. Pursuit-evasion in a graph. In Y. Alavi and D. R. Lick, editors, *Theory and Application of Graphs*, pages 426–441. Springer-Verlag, Berlin, 1976.
- [617] T. Parthasarathy and M. Stern. Markov games: A survey. In *Differential Games and Control Theory II*, pages 1–46. Marcel Dekker, New York, 1977.
- [618] R. P. Paul. *Robot Manipulators: Mathematics, Programming, and Control*. MIT Press, Cambridge, MA, 1981.
- [619] R. P. Paul and B. Shimano. Compliance and control. In *Proc. of the Joint American Automatic Control Conference*, pages 1694–1699, 1976.
- [620] E. Paulos and J. F. Canny. Ubiquitous tele-embodiment: Applications and implications. *International Journal of Human-Computer Studies*, 46(6):861–877, 1997.
- [621] E. Paulos and J. F. Canny. PRoP: Personal roving presence. In *ACM SIGCHI Conference on Human Factors in Computing Systems*, 1998.
- [622] J. Pearl. *Heuristics*. Addison-Wesley, Reading, MA, 1984.
- [623] J. R. Perkins, C. Humes Jr., and P. R. Kumar. Distributed scheduling of flexible manufacturing systems: Stability and performance. *IEEE Trans. Robot. & Autom.*, 10(2):133–141, 1994.
- [624] J. Pertin-Troccaz. Grasping: A state of the art. In O. Khatib, J. J. Craig, and T. Lozano-Pérez, editors, *The Robotics Review 1*. MIT Press, Cambridge, MA, 1989.
- [625] S. Petitjean, D. Kriegman, and J. Ponce. Computing exact aspect graphs of curved objects: algebraic surfaces. *Int. J. Comput. Vis.*, 9:231–255, Dec 1992.
- [626] L. A. Petrosjan. *Differential Games of Pursuit*. Singapore, River Edge, NJ, 1993.
- [627] J. Pettré, J.-P. Laumond, and T. Siméon. A 2-stages locomotion planner for digital actors. In *Eurographics/SIGGRAPH Symposium on Computer Animation*, pages 258–264, 2003.
- [628] L. Piegl. On NURBS: A survey. *IEEE Trans. Comp. Graph. & Appl.*, 11(1):55–71, Jan 1991.

- [629] C. Pisula, K. Hoff, M. Lin, and D. Manoch. Randomized path planning for a rigid body based on hardware accelerated Voronoi sampling. In *Proc. Workshop on Algorithmic Foundation of Robotics*, 2000.
- [630] R. Pito. A solution to the next best view problem for automated cad model acquisition of free-form objects using range cameras. Technical Report 95-23, GRASP Lab, University of Pennsylvania, May 1995.
- [631] R. Pito. A sensor based solution to the next best view problem. In *Int. Conf. Pattern Recognition*, 1996.
- [632] M. Pittarelli. Decisions with probabilities over finite product spaces. *IEEE Trans. Syst., Man, Cybern.*, 21(5):1238–1242, Sep/Oct 1991.
- [633] M. Pocchiola and G. Vegter. The visibility complex. *Int. J. Comput. Geom. & Appl.*, 6(3):279–308, 1996.
- [634] I. Pohl. Bi-directional and heuristic search in path problems. Technical report, Stanford Linear Accelerator Center, 1969.
- [635] I. Pohl. Bi-directional search. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, pages 127–140. American Elsevier, New York, 1971.
- [636] S. Premvuti and S. Yuta. Consideration on the cooperation of multiple autonomous mobile robots. In *IEEE Int. Workshop on Intelligent Robots and Systems*, pages 59–63, 1990.
- [637] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, Berlin, 1985.
- [638] A. Quaid and A. Rizzi. Robust and efficient motion planning for a planar robot using hybrid control. In *IEEE Int. Conf. Robot. & Autom.*, pages 4021–4026, 2000.
- [639] S. Quinlan. Efficient distance computation between nonconvex objects. In *IEEE Int. Conf. Robot. & Autom.*, pages 3324–3329, 1994.
- [640] S. Quinlan and O. Khatib. Elastic bands: Connecting path planning and control. In *IEEE Int. Conf. Robot. & Autom.*, pages 802–807, 1993.
- [641] M. Rabin. Transaction protection by beacons. *J. Computation Systems Science*, 27(2):256–267, 1983.
- [642] M. H. Raibert and J. J. Craig. Hybrid position/force control of manipulators. *Trans. ASME J. Dyn. Sys., Meas., & Contr.*, 102, 1981.
- [643] S. Rajko and S. M. LaValle. A pursuit-evasion bug algorithm. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 1954–1960, 2001.

- [644] B. S. Y. Rao, H. F. Durrant-Whyte, and J. S. Sheen. A fully decentralized multi-sensor system for tracking and surveillance. *Int. J. Robot. Res.*, 12(1):20–44, February 1993.
- [645] N. S. V. Rao. Robot navigation in unknown generalized polygonal terrains using vision sensors. *IEEE Trans. Syst., Man, Cybern.*, 25(6):947–962, 1995.
- [646] N. S. V. Rao, S. S. Iyengar, J. B. Oommen, and R. L. Kashyap. On terrain model acquisition by a point robot amidst polyhedral obstacles. *IEEE J. of Robot. & Autom.*, 4:450–455, 1988.
- [647] S. Ratering and M. Gini. Robot navigation in a known environment with unknown moving obstacles. In *IEEE Int. Conf. Robot. & Autom.*, pages 25–30, 1993.
- [648] J. A. Reeds and L. A. Shepp. Optimal paths for a car that goes both forwards and backwards. *Pacific J. Math.*, 145(2):367–393, 1990.
- [649] J. M. Rehg and T. Kanade. Visual tracking of high DOF articulated structures: an application to human hand tracking. In J.-O. Eklundh, editor, *Lecture Notes in Computer Science, Vol 801, Computer Vision - ECCV '94*, pages 35–44. Springer-Verlag, Berlin, 1994.
- [650] J. Reif and H. Wang. Non-uniform discretization approximations for kinodynamic motion planning. In J.-P. Laumond and M. Overmars, editors, *Algorithms for Robotic Motion and Manipulation*, pages 97–112. A K Peters, Wellesley, MA, 1997.
- [651] J. H. Reif. Complexity of the mover’s problem and generalizations. In *Proc. of IEEE Symp. on Foundat. of Comp. Sci.*, pages 421–427, 1979.
- [652] J. H. Reif and M. Sharir. Motion planning in the presence of moving obstacles. In *Proc. of IEEE Symp. on Foundat. of Comp. Sci.*, pages 144–154, 1985.
- [653] J. H. Reif and M. Sharir. Motion planning in the presence of moving obstacles. *J. Assoc. Comput. Machin.*, 41:764–790, 1994.
- [654] J. H. Reif and S. R. Tate. Continuous alternation: The complexity of pursuit in continuous domains. *Algorithmica*, 10:157–181, 1993.
- [655] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1977.
- [656] E. Rimon and J. Canny. Construction of C-space roadmaps using local sensory data – what should the sensors look for? In *IEEE Int. Conf. Robot. & Autom.*, pages 117–124, 1994.

- [657] E. Rimon and D. E. Koditschek. The construction of analytic diffeomorphisms for exact robot navigation on star worlds. In *IEEE Int. Conf. Robot. & Autom.*, pages 21–26, May 1989.
- [658] E. Rimon and D. E. Koditschek. Exact robot navigation in geometrically complicated but topologically simple spaces. In *IEEE Int. Conf. Robot. & Autom.*, pages 1937–1942, May 1990.
- [659] E. Rimon and D. E. Koditschek. Exact robot navigation using artificial potential fields. *IEEE Trans. Robot. & Autom.*, 8(5):501–518, October 1992.
- [660] H. Rohnert. Shortest paths in the plane with convex polygonal obstacles. *Information Processing Letters*, 23:71–76, 1986.
- [661] R. M. Rosenberg. *Analytical Dynamics of Discrete Systems*. Plenum Press, New York, 1977.
- [662] J. J. Rotman. *Introduction to Algebraic Topology*. Springer-Verlag, Berlin, 1988.
- [663] N. C. Rowe and R. F. Richbourg. A new method for optimal path planning through nonhomogeneous free space. Technical Report NPS52-87-003, Naval Postgraduate School, 1987.
- [664] H. L. Royden. *Real Analysis*. MacMillan, New York, NY, 1988.
- [665] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd Edition*. Pearson Education, Inc., Upper Saddle River, NJ, 2003.
- [666] S. Sachs, S. Rajko, and S. M. LaValle. Visibility-based pursuit-evasion in an unknown planar environment. *To appear in International Journal of Robotics Research*, 2004.
- [667] S. R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *IEEE Trans. Syst., Man, Cybern.*, 21(3):660–674, May/June 1991.
- [668] H. Sagan. *Space-Filling Curves*. Springer-Verlag, Berlin, 1994.
- [669] G. Sahar and J. M. Hollerbach. Planning minimum-time trajectories for robot arms. *Int. J. Robot. Res.*, 5(3):97–140, 1986.
- [670] G. Sánchez and J.-C. Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *Int. Symp. Robotics Research*, 2001.
- [671] L. Sander. Fractal growth processes. *Nature*, 322:789–793, 1986.
- [672] Y. Sawaragi, H. Nakayama, and T. Tanino. *Theory of Multiobjective Optimization*. Academic Press, New York, NY, 1985.

- [673] A. Scheuer and Ch. Laugier. Planning sub-optimal and continuous-curvature paths for car-like robots. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 25–31, 1998.
- [674] G. K. Schmidt and K. Azarm. Mobile robot navigation in a dynamic world using an unsteady diffusion equation strategy. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 642–647, 1992.
- [675] J. T. Schwartz and M. Sharir. On the piano movers' problem: I. The case of a two-dimensional rigid polygonal body moving amidst polygonal barriers. *Communications on Pure and Applied Mathematics*, 36:345–398, 1983.
- [676] J. T. Schwartz and M. Sharir. On the piano movers' problem: II. General techniques for computing topological properties of algebraic manifolds. *Communications on Pure and Applied Mathematics*, 36:345–398, 1983.
- [677] J. T. Schwartz and M. Sharir. On the piano movers' problem: III. Coordinating the motion of several independent bodies. *Int. J. Robot. Res.*, 2(3):97–140, 1983.
- [678] J. T. Schwartz and M. Sharir. A survey of motion planning and related geometric algorithms. *Artif. Intell.*, 37:157 – 169, 1988.
- [679] J. T. Schwartz, M. Sharir, and J. Hopcroft. *Planning, Geometry, and Complexity of Robot Motion*. Ablex Publishing Corporation, Norwood, NJ, 1987.
- [680] S. Sekhavat, P. Svestka, J.-P. Laumond, and M. H. Overmars. Multilevel path planning for nonholonomic robots using semiholonomic subsystems. *Int. J. Robot. Res.*, 17:840–857, 1998.
- [681] N. F. Sepetov, V. Krchnak, M. Stankova, S. Wade, K. S. Lam, and M. Lebl. Library of libraries: Approach to synthetic combinatorial library design and screening of "pharmacophore" motifs. *Proc. Natl. Acad. Sci. USA*, 92:5426–5430, June 1995.
- [682] I. K. Sethi and R. Jain. Finding trajectories of feature points in a monocular image sequence. *IEEE Trans. Pattern Anal. Machine Intell.*, 9(1):56–73, 1987.
- [683] J. A. Sethian. *Level set methods : Evolving interfaces in geometry, fluid mechanics, computer vision, and materials science*. Cambridge University Press, 1996.
- [684] U. Shaked and C. E. de Souza. Continuous-time tracking problems in an h_∞ setting: A game theory approach. *IEEE Trans. Autom. Control*, 40(5):841–852, May 1995.

- [685] Y. Shan and Y. Koren. Obstacle accommodation motion planning. *IEEE Trans. Robot. & Autom.*, 1(11):36–49, February 1995.
- [686] M. Sharir. Algorithmic motion planning. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 733–754. CRC Press, New York, 1997.
- [687] M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, 1995.
- [688] R. Sharma. Locally efficient path planning in an uncertain, dynamic environment using a probabilistic model. *IEEE Trans. Robot. & Autom.*, 8(1):105–110, February 1992.
- [689] R. Sharma. A probabilistic framework for dynamic motion planning in partially known environments. In *IEEE Int. Conf. Robot. & Autom.*, pages 2459–2464, Nice, France, May 1992.
- [690] R. Sharma and Y. Aloimonos. Visual motion analysis under interceptive behavior. In *Proc. IEEE Conf. on Comp. Vision and Patt. Recog.*, Urbana, IL, June 1992.
- [691] R. Sharma, J.-Y. Hervé, and P. Cucka. Dynamic robot manipulation using visual tracking. In *IEEE Int. Conf. Robot. & Autom.*, pages 1844–1849, 1992.
- [692] R. Sharma, S. M. LaValle, and S. A. Hutchinson. Optimizing robot motion strategies for assembly with stochastic models of the assembly process. *IEEE Trans. on Robotics and Automation*, 12(2):160–174, April 1996.
- [693] R. Sharma, D. M. Mount, and Y. Aloimonos. Probabilistic analysis of some navigation strategies in a dynamic environment. *IEEE Trans. Syst., Man, Cybern.*, 23(5):1465–1474, September 1993.
- [694] T. Shermer. Recent results in art galleries. *Proc. IEEE*, 80(9):1384–1399, September 1992.
- [695] T. Shibata and T. Fukuda. Coordinative behavior by genetic algorithm and fuzzy in evolutionary multi-agent system. In *IEEE Int. Conf. Robot. & Autom.*, pages 1:760–765, 1993.
- [696] C. L. Shih, T.-T. Lee, and W. A. Gruver. A unified approach for robot motion planning with moving polyhedral obstacles. *IEEE Trans. Syst., Man, Cybern.*, 20:903–915, 1990.
- [697] Z. Shiller. Dynamic motion planning of autonomous vehicles. *IEEE Trans. Robot. & Autom.*, 7(2), April 1991.

- [698] Z. Shiller and S. Dubowsky. On the optimal control of robotic manipulators with actuator and end-effector constraints. In *ICRA*, pages 614–620, 1985.
- [699] Z. Shiller and S. Dubowsky. Global time-optimal motions of robotic manipulators in the presence of obstacles. In *ICRA*, 1988.
- [700] Z. Shiller and S. Dubowsky. On computing time-optimal motions of robotic manipulators in the presence of obstacles. *IEEE Trans. on Robotics and Automation*, 7(6), Dec 1991.
- [701] K. G. Shin and N. D. McKay. Minimum-time control of robot manipulators with geometric path constraints. *IEEE Trans. Autom. Control*, 30(6):531–541, 1985.
- [702] K. G. Shin and Q. Zheng. Minimum-time collision-free trajectory planning for dual-robot systems. *IEEE Trans. Robot. & Autom.*, 8(5):641–644, October 1992.
- [703] P. Shirley. Discrepancy as a quality measure for sample distributions. In *Proc. Eurographics*, pages 183–194, Vienna, 1991. Elsevier Science.
- [704] A. M. Shkel and V. J. Lumelsky. Incorporating body dynamics into sensor-based motion planning: The maximum turn strategy. *IEEE Trans. Robot. & Autom.*, 13(6):873–880, December 1997.
- [705] T. Simeon, J.-P. Laumond., and C. Nissoux. Visibility based probabilistic roadmaps for motion planning. *Advanced Robotics Journal*, 14(6), 2000.
- [706] T. Simeon, S. Leroy, and J.-P. Laumond. Path coordination for multiple mobile robots: a resolution complete algorithm. *IEEE Trans. Robot. & Autom.*, 18(1), February 2002.
- [707] B. Simov, S. M. LaValle, and G. Slutzki. A complete pursuit-evasion algorithm for two pursuers using beam detection. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 618–623, 2002.
- [708] B. Simov, G. Slutzki, and S. M. LaValle. Pursuit-evasion using beam detection. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, 2000.
- [709] B. Simov, G. Slutzki, and S. M. LaValle. Clearing a polygon with two 1-searchers. *Submitted to International Journal of Computational Geometry and Applications*, 2003.
- [710] K. Singh and K. Fujimura. Map making by cooperating mobile robots. In *IEEE Int. Conf. Robot. & Autom.*, pages 254–258, 1993.
- [711] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, Boston, 1997.

- [712] I. H. Sloan and S. Joe. *Lattice Methods for Multiple Integration*. Oxford Science Publications, Englewood Cliffs, NJ, 1990.
- [713] R. C. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainty. *Int. J. Robot. Res.*, 5(4):56–68, 1986.
- [714] R. C. Smith, M. Self, and P. Cheeseman. A stochastic map for uncertain spatial relationships. In *Fourth International Symposium on Robotics Research*, pages 421–429, 1987.
- [715] G. Song and N. M. Amato. Using motion planning to study protein folding pathways. *Journal of Computational Biology*, 9(2):149–168, 2002.
- [716] P. Souères and J. P. Laumond. Shortest paths synthesis for a car-like robot. In *IEEE Transactions on Automatic Control*, pages 672–688, 1996.
- [717] R. Spence and S. A. Hutchinson. Dealing with unexpected moving obstacles by integrating potential field planning with inverse dynamics control. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 1485–1490, 1992.
- [718] R. Spence and S. A. Hutchinson. An integrated architecture for robot motion planning and control in the presence of obstacles with unknown trajectories. *IEEE Trans. Syst., Man, Cybern.*, 25(1):100–110, 1995.
- [719] M. Spivak. *Differential Geometry*. Publish or Perish, Inc., 1979.
- [720] M. W. Spong and M. Vidyasagar. *Robot Dynamics and Control*. Wiley, New York, NY, 1989.
- [721] R. L Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6:579–589, 1991.
- [722] W. Stadler. Fundamentals of multicriteria optimization. In W. Stadler, editor, *Multicriteria Optimization in Engineering and in the Sciences*, pages 1–25. Plenum Press, New York, NY, 1988.
- [723] A. Stentz. Optimal and efficient path planning for partially-known environments. In *IEEE Int. Conf. Robot. & Autom.*, pages 3310–3317, 1994.
- [724] D. Stewart. A platform with six degrees of freedom. In *Institution of Mechanical Engineers, Proceedings 1965-66, 180 Part 1*, pages 371–386, 1966.
- [725] H. K. Struemper. *Motion Control for Nonholonomic Systems on Matrix Lie Groups*. PhD thesis, University of Maryland, College Park, MD, 1997.
- [726] M. R. Stytz. Distributed virtual environments. *IEEE Computer Graphics & Applications*, 16(3):19–31, May 1996.

- [727] S.-H. Suh and K. G. Shin. A variational dynamic programming approach to robot-path planning with a distance-safety criterion. *IEEE Trans. Robot. & Autom.*, 4(3):334–349, June 1988.
- [728] A. G. Sukharev. Optimal strategies of the search for an extremum. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 11(4), 1971. Translated from Russian, *Zh. Vychisl. Mat. i Mat. Fiz.*, 11, 4, 910-924, 1971.
- [729] S. Sundar and Z. Shiller. Optimal obstacle avoidance based on the Hamilton-Jacobi-Bellman equation. *IEEE Trans. Robot. & Autom.*, 13(2):305–310, April 1997.
- [730] H. Sussman and G. Tang. Shortest paths for the Reeds-Shepp car: A worked out example of the use of geometric techniques in nonlinear optimal control. Technical Report SYNCON 91-10, Dept. of Mathematics, Rutgers University, 1991.
- [731] K. Sutner and W. Maass. Motion planning among time dependent obstacles. *Acta Informatica*, 26:93–122, 1988.
- [732] R. S. Sutton. Planning by incremental dynamic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 353–357. Morgan Kaufmann, 1991.
- [733] I. Suzuki and M. Yamashita. Searching for a mobile intruder in a polygonal region. *SIAM J. Computing*, 21(5):863–888, October 1992.
- [734] P. Svestka and M. H. Overmars. Coordinated motion planning for multiple car-like robots using probabilistic roadmaps. In *IEEE Int. Conf. Robot. & Autom.*, pages 1631–1636, 1995.
- [735] M. Taix. *Planification de Mouvement pour Robot Mobile Non-Holonome*. PhD thesis, Laboratoire d'Analyse et d'Architecture des Systemes, Toulouse, France, January 1991.
- [736] O. Takahashi and R. J. Schilling. Motion planning in a plane using generalized Voronoi polygons. *IEEE Trans. Robot. & Autom.*, 5(2):143–150, 1989.
- [737] H. Takeda, C. Facchinetti, and J.-C. Latombe. Planning the motions of a mobile robot in a sensory uncertainty field. *IEEE Trans. Pattern Anal. Machine Intell.*, 16(10):1002–1017, October 1994.
- [738] H. Takeda and J.-C. Latombe. Sensory uncertainty field for mobile robot navigation. In *IEEE Int. Conf. Robot. & Autom.*, pages 2465–2472, Nice, France, May 1992.

- [739] R. Talluri and J. K. Aggarwal. Mobile robot self-location using model-image feature correspondence. *IEEE Trans. Robot. & Autom.*, 12(1):63–77, February 1996.
- [740] J. Tan and Ning Xi. Hybrid system design for singularityless task level robot controllers. In *IEEE Int. Conf. Robot. & Autom.*, pages 3007–3012, 2000.
- [741] X. Tan. Searching a simple polygon by a k-searcher. unpublished manuscript, 2000.
- [742] R. H. Taylor, M. T. Mason, and K. Y. Goldberg. Sensor-based manipulation planning as a game with nature. In *Fourth International Symposium on Robotics Research*, pages 421–429, 1987.
- [743] D. Terzopoulos and D. Metaxas. Dynamic 3D models with local and global deformations. *IEEE Trans. Pattern Anal. Machine Intell.*, 13(7):703–714, 1991.
- [744] S. Tezuka. *Uniform Random Numbers: Theory and Practice*. Kluwer Academic Press, Boston, 1995.
- [745] S. Tezuka. Quasi-monte carlo: The discrepancy between theory and practice. In K.-T. Fang, F. J. Hickernell, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2000*, pages 124–140. Springer-Verlag, Berlin, 2002.
- [746] R. B. Tilove. Local obstacle avoidance for mobile robots based on the method of artificial potentials. In *IEEE Int. Conf. Robot. & Autom.*, pages 566–571, May 1990.
- [747] G. J. Toussaint, T. Başar, and F. Bullo. Motion planning for nonlinear underactuated vehicles using hinfinity techniques. Coordinated Science Lab, University of Illinois, September 2000.
- [748] B. Tovar, S. M. LaValle, and R. Murrieta. Locally-optimal navigation in multiply-connected environments without geometric maps. In *IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems*, 2003.
- [749] B. Tovar, S. M. LaValle, and R. Murrieta. Optimal navigation and object finding without geometric maps or localization. In *Proc. IEEE International Conference on Robotics and Automation*, pages 464–470, 2003.
- [750] J. F. Traub and A. G. Werschulz. *Complexity and Information*. Cambridge University Press, Cambridge, 1998.
- [751] B. Triggs and C. Laugier. Automatic camera placement for robot vision tasks. In *IEEE Int. Conf. Robot. & Autom.*, pages 1732–1737, 1995.

- [752] J. C. Trinkle and D. C. Zeng. Prediction of the quasistatic planar motion of a contacted rigid body. *IEEE Trans. Robot. & Autom.*, 11(2):229–246, April 1995.
- [753] C. Tsai. Multiple robot coordination and programming. In *IEEE Int. Conf. Robot. & Autom.*, pages 978–985, Sacramento, CA, April 1991.
- [754] G. Turk and M. Levoy. Zippered polygon meshes from range images. pages 311–318, 1994.
- [755] S. Udupa. *Collision Detection and Avoidance in Computer Controlled Manipulators*. PhD thesis, Dept. of Electrical Engineering, California Institute of Technology, 1977.
- [756] C. Urmson and R. Simmons. Approaches for heuristically biasing rrt growth. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, 2003.
- [757] D. Vallejo, C. Jones, and N. Amato. An adaptive framework for "single shot" motion planning. Texas A&M, October 1999.
- [758] C. van Delft. Approximate solutions for large-scale piecewise deterministic control systems arising in manufacturing flow control models. *IEEE Trans. Robot. & Autom.*, 10(2):142–152, 1994.
- [759] J. G. van der Corput. Verteilungsfunktionen I. *Akad. Wetensch.*, 38:813–821, 1935.
- [760] D. Vanderpooten. Multiobjective programming: Basic concepts and approaches. In R. Slowinski and J. Teghem, editors, *Stochastic vs. Fuzzy Approaches to Multiobjective Mathematical Programming under Uncertainty*, pages 7–22. Kluwer Academic Publishers, Boston, MA, 1990.
- [761] M. Vendittelli and J.-P. Laumond. Visible positions for a car-like robot amidst obstacles. In J.-P. Laumond and M. Overmars, editors, *Algorithms for Robotic Motion and Manipulation*, pages 213–228. A K Peters, Wellesley, MA, 1997.
- [762] M. W. Walker and J. Dionise. A world model based approach to manipulator control. In *IEEE Int. Conf. Robot. & Autom.*, pages 453–460, 1990.
- [763] F.-Y. Wang and P. J. A. Lever. A cell mapping method for general optimum trajectory planning of multiple robotic arms. *Robots and Autonomous Systems*, 12:15–27, 1994.
- [764] X. Wang and F. J. Hickernell. Randomized Halton sequences. *Math. Comp. Modelling*, 32:887–899, 2000.

- [765] X. Wang and F. J. Hickernell. An historical overview of lattice point sets. In K.-T. Fang, F. J. Hickernell, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2000*, pages 158–167. Springer-Verlag, Berlin, 2002.
- [766] F. W. Warner. *Foundations of Differentiable Manifolds and Lie Groups*. Springer-Verlag, New York, 1983.
- [767] C. W. Warren. Multiple robot path coordination using artificial potential fields. In *IEEE Int. Conf. Robot. & Autom.*, pages 500–505, 1990.
- [768] H. Weyl. Über die Gleichverteilung von Zahlen mod Eins. *Math. Ann.*, 77:313–352, 1916.
- [769] P. Whaite and F. P. Ferrie. From uncertainty to visual exploration. In *Proc. Int. Conf. on Computer Vision*, pages 690–697, 1990.
- [770] D. Whitney. Force feedback control of manipulator fine motions. *Trans. ASME J. of Dyn. Sys., Meas., & Contr.*, 99:91–97, 1977.
- [771] D. E. Whitney. Historical perspectives and the state of the art in robot force control. In *IEEE Int. Conf. Robot. & Autom.*, pages 262–268, 1985.
- [772] H. Whitney. Local properties of analytic varieties. In S. Cairns, editor, *Differential and Combinatorial Topology*, pages 205–244. Princeton University Press, Princeton, USA, 1965.
- [773] G. Wilfong. Motion planning in the presence of movable obstacles. In *Proc. ACM Symposium on Computational Geometry*, pages 279–288, June 1988.
- [774] S. A. Wilmarth, N. M. Amato, and P. F. Stiller. MAPRM: A probabilistic roadmap planner with sampling on the medial axis of the free space. In *IEEE Int. Conf. Robot. & Autom.*, pages 1024–1031, 1999.
- [775] R. H. Wilson. *On Geometric Assembly Planning*. PhD thesis, Stanford University, March 1992.
- [776] R. H. Wilson and J.-C. Latombe. Geometric reasoning about mechanical assembly. *Artificial Intelligence*, 71(2):371–396, 1994.
- [777] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, Reading, MA, 1992.
- [778] L. Wixson. Viewpoint selection for visual search. In *Proc. IEEE Conf. on Comp. Vision and Patt. Recog.*, pages 800–805, 1994.
- [779] C. Wren, A. Azarbayejani, T. Darell, and A. Pentland. Pfindex: Real-time tracking of the human body. Technical Report 353, M.I.T. Media Laboratory Perceptual Computing Section, 1995.

- [780] J. Yakey, S. M. LaValle, and L. E. Kavraki. Randomized path planning for linkages with closed kinematic chains. *IEEE Transactions on Robotics and Automation*, 17(6):951–958, December 2001.
- [781] J. H. Yakey. Randomized path planning for linkages with closed kinematic chains. Master’s thesis, Iowa State University, Ames, IA, 1999.
- [782] M. Yamashita, H. Unemoto, I. Suzuki, and T. Kameda. Searching for mobile intruders in a polygonal region by a group of mobile searchers. Technical Report TR-96-07-01, Dept. of Electrical Engineering and Computer Science, University of Wisconsin - Milwaukee, July 1996.
- [783] L. Yang and S. M. LaValle. A framework for planning feedback motion strategies based on a random neighborhood graph. In *Proc. IEEE Int’l Conf. on Robotics and Automation*, pages 544–549, 2000.
- [784] L. Yang and S. M. LaValle. An improved random neighborhood graph approach. In *Proc. IEEE Int’l Conf. on Robotics and Automation*, pages 254–259, 2002.
- [785] L. Yang and S. M. LaValle. The sampling-based neighborhood graph: A framework for planning and executing feedback motion strategies. *To appear in IEEE Transactions on Robotics and Automation*, 2004.
- [786] Y. Yavin and M. Pachtter. *Pursuit-Evasion Differential Games*. Pergamon Press, Oxford, England, 1987.
- [787] A. Yershova and S. M. LaValle. Deterministic sampling methods for spheres and $SO(3)$. In *Proc. IEEE International Conference on Robotics and Automation*, 2004. Under review.
- [788] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321, 1993.
- [789] M. Yim. *Locomotion with a Unit-Modular Reconfigurable Robot*. PhD thesis, Stanford Univ., December 1994. Stanford Technical Report STAN-CS-94-1536.
- [790] J. Yong. On differential evasion games. *SIAM J. Control & Optimization*, 26(1):1–22, January 1988.
- [791] J. Yong. On differential pursuit games. *SIAM J. Control & Optimization*, 26(2):478–495, March 1988.
- [792] J. Yong. A zero-sum differential game in a finite duration with switching strategies. *SIAM J. Control & Optimization*, 28(5):1234–1250, September 1990.

- [793] S. Yu and M. Berthod. A game strategy approach for image labeling. *Computer Vision and Image Understanding*, 61(1):32–37, 1995.
- [794] Y. Yu and K. Gupta. On sensor-based roadmap: A framework for motion planning for a manipulator arm in unknown environments. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 1919–1924, 1998.
- [795] S. C. Zaharakis and A. Guez. Time optimal robot navigation via the slack set method. *IEEE Trans. Syst., Man, Cybern.*, 20(6):1396–1407, 1990.
- [796] L. S. Zaremba. Differential games reducible to optimal control problems. In *IEEE Conf. Decision & Control*, pages 2449–2450, Tampa, FL, December 1989.
- [797] M. Zefran and V. Kumar. Optimal control of systems with unilateral constraints. In *IEEE Int. Conf. Robot. & Autom.*, 1995.
- [798] D. B. Zhang, L. Van Gool, and A. Oosterlinck. Stochastic predictive control of robot tracking systems with dynamic visual feedback. In *IEEE Int. Conf. Robot. & Autom.*, pages 610–615, 1990.
- [799] Y. F. Zheng, J. Y. S. Luh, and P. F. Jia. Integrating two industrial robots into a coordinated system. *Computers in Industry*, 12:285–298, 1989.
- [800] Q. Zhu. Hidden Markov model for dynamic obstacle avoidance of mobile robot navigation. *IEEE Trans. Robot. & Autom.*, 7(3):390–397, June 1991.
- [801] S. Zions. Multiple criteria mathematical programming: An overview and several approaches. In P. Serafini, editor, *Mathematics of Multi-Objective Optimization*, pages 227–273. Springer-Verlag, Berlin, 1985.
- [802] C. P. E. Zollikofer and M. S. Ponce de León. Tools for rapid prototyping in the biosciences. *IEEE Computer Graphics & Applications*, 15(6):48–55, November 1995.