

Lectures on

Robotic Planning and Kinematics



Francesco Bullo
Stephen L. Smith

Lectures on Robotic Planning and Kinematics
Francesco Bullo and Stephen L. Smith
Version v0.91(d) (7 Apr 2016).



This work is licensed under the [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0), available at: <https://creativecommons.org/licenses/by-nc-sa/4.0>. A summary (not a substitute) of the license is as follows: You are allowed to

- share, that is, copy and redistribute this work in any medium or format, and
- adapt, that is, remix, transform, and build upon this work,

provided (i) appropriate attribution is given, (ii) the material is not used for commercial purposes, and (iii) the adapted material is distributed under the same license as the original.

Exceptions to this licence are the Figures [3.2](#), [3.3](#), [3.5](#), [3.14](#) and [8.4](#). Permission is granted to reproduce these figures as part of this work in both print and electronic formats, for distribution worldwide in the English language. All other copyrights for these figures belongs to their respective owners.

We are thankful for any feedback information, including suggestions, errors, or comments about teaching or research uses.

To Lily, Marcello and Gabriella

To Julia

Contents

| | |
|--|------------|
| Contents | iv |
| Preface | vii |
| I Planning | 1 |
| 1 Sensor-based Planning | 3 |
| 1.1 Problem setup and modeling assumptions | 3 |
| 1.2 The Bug 0 algorithm | 5 |
| 1.3 The Bug 1 algorithm | 6 |
| 1.4 The Bug 2 algorithm | 11 |
| 1.5 The completeness of the Bug 1 algorithm | 16 |
| 1.6 Appendix: Operations on sets | 19 |
| 1.7 Exercises | 20 |
| 2 Motion Planning via Decomposition and Search | 25 |
| 2.1 Problem setup and modeling assumptions | 25 |
| 2.2 Workspace decomposition | 27 |
| 2.3 Search algorithms over graphs | 34 |
| 2.4 Exercises | 44 |
| 3 Configuration Spaces | 49 |
| 3.1 Problem setup: Multi-body robots in realistic workspaces | 49 |
| 3.2 The configuration space | 51 |
| 3.3 Example configuration spaces | 54 |
| 3.4 Forward and inverse kinematic maps | 59 |
| 3.5 Appendix: Inverse trigonometric problems | 62 |

| | | |
|-----------|--|------------|
| 3.6 | Exercises | 65 |
| 4 | Free Configuration Spaces via Sampling and Collision Detection | 71 |
| 4.1 | The free configuration space | 71 |
| 4.2 | Numerical computation of the free configuration space | 78 |
| 4.3 | Sampling methods | 79 |
| 4.4 | Collision detection methods | 84 |
| 4.5 | Appendix: Runtime of the numerical computation of the free configuration space | 90 |
| 4.6 | Exercises | 92 |
| 5 | Motion Planning via Sampling | 95 |
| 5.1 | Roadmaps | 95 |
| 5.2 | Complete planners on exact roadmaps | 96 |
| 5.3 | General-purpose planners via sampling-based roadmaps | 99 |
| 5.4 | Incremental sampling-based planning | 102 |
| 5.5 | Appendix: Shortest paths in weighted graphs via Dijkstra's algorithm | 105 |
| 5.6 | Exercises | 109 |
| II | Kinematics | 111 |
| 6 | Introduction to Kinematics and Rotation Matrices | 113 |
| 6.1 | Geometric objects and their algebraic representation | 114 |
| 6.2 | Geometric properties of rotations | 116 |
| 6.3 | Representing reference frames | 118 |
| 6.4 | Appendix: A primer on matrix theory | 122 |
| 6.5 | Appendix: The theory of groups | 124 |
| 6.6 | Exercises | 126 |
| 7 | Rotation Matrices | 129 |
| 7.1 | From reference frames to general rotation matrices | 129 |
| 7.2 | Composition of rotations | 132 |
| 7.3 | Parametrization of rotation matrices | 136 |
| 7.4 | Appendix: Properties of skew symmetric matrices | 142 |
| 7.5 | Appendix: Proof of Rodrigues' formula and of its inverse | 143 |
| 7.6 | Exercises | 146 |
| 8 | Displacement Matrices and Inverse Kinematics | 151 |
| 8.1 | Displacements as matrices | 151 |
| 8.2 | Basic and composite displacements | 154 |
| 8.3 | Inverse kinematics on the set of displacements | 157 |
| 8.4 | Exercises | 163 |

| | |
|--|------------|
| 9 Linear and Angular Velocities of a Rigid Body | 167 |
| 9.1 Angular velocity | 168 |
| 9.2 Linear and angular velocities | 171 |
| 9.3 Vehicle motion models and integration | 172 |
| 9.4 Exercises | 175 |
| Bibliography | 177 |

Preface

Topics These lecture notes cover motion planning and kinematics with an emphasis on geometric reasoning, programming and matrix computations. In the context of motion planning, we present sensor-based planning, configuration spaces, decomposition and sampling methods, and advanced planning algorithms. In the context of kinematics, we present reference frames, rotation matrices and their properties, displacement matrices, and kinematic motion models.

This book is a collection of lecture notes and is not meant to provide a comprehensive treatment of any topic in particular. For more comprehensive textbooks and more advanced research monographs, we refer the reader to a rich growing literature, including (Choset et al. 2005; Corke 2011; Craig 2003; Dudek and Jenkin 2010; Jazar 2010; Kelly 2013; LaValle 2006; Mason 2001; Murray et al. 1994; Niku 2010; Selig 1996; Siciliano et al. 2009; Siegwart et al. 2011; Spong et al. 2006; Thrun et al. 2005).

The intended audience and use of these lectures These lecture notes are intended for undergraduate students pursuing a 4-year degree in mechanical, electrical or related engineering disciplines. Computer Science students will find that the chapters on motion planning overlap with their courses on data structures and algorithms. Prerequisites for these lecture notes include a course on programming, a basic course on linear algebra and matrix theory, and basic knowledge of ordinary differential equations.

These lecture notes are intended for use in a course with approximately 30-36 contact hours (e.g., roughly one chapter per week in a 10-week long course). Student homework includes standard textbook exercises as well as an extensive set of programming exercises (focused on motion planning). We envision this textbook to be used with weekly homework assignments, weekly computer laboratory assignments, a midterm exam, and a final exam.

For the benefit of instructors, these lecture notes are supplemented by two documents. First, a solutions manual, including programming solutions, is available upon request free of charge to instructors at accredited institutions. Second, these lecture notes are also available in “slide/landscape” format especially suited for classroom teaching with projectors or screens.

Learning Objectives Course learning outcomes, i.e., skills that students should possess at the end of a 10-week course based upon these lectures, include:

- (i) an ability to apply knowledge of geometry, graph algorithms, and linear algebra to robotic systems,
- (ii) an ability to use a numerical computing and programming environment to solve engineering problems,
- (iii) an ability to formulate, and solve motion planning problems in robotics, and
- (iv) an ability to formulate and solve kinematics problems in robotics.

Implementation via Programming These notes contain numerous algorithms written in pseudocode and numerous programming assignments. We ask the reader to choose a programming language and environment capable of numerical computation and graphical visualization. Typical choices may be Matlab and Python (with its packages `numpy` and `scipy`). Later programming assignments depend upon earlier ones and so readers are advised to complete the programming assignments in the order in which they appear.

Readers should be informed that the purpose of the programming assignments is to develop their understanding of algorithms and their programming ability. For more advanced purposes (e.g., commercial enterprises or research projects), we refer to CGAL for an efficient and reliable algorithm library available as open source software; see (Fabri and Pion 2009; The CGAL Project 2015), and to the “The Motion Strategy Library” and the “Open Motion Planning Library” for motion planning libraries; see (LaValle et al. 2003) and (Şucan et al. 2012) respectively.

Acknowledgments First, we wish to thank Joey W. Durham, who designed and implemented the first version of many exercises and programming assignments, and Patrick Therrien, who prepared a comprehensive set of programming solutions in Matlab and Python. We are also very thankful to all instructors who were early adopters of these lecture notes and provided useful feedback, including Professors Sonia Martínez and Fabio Pasqualetti. Finally, it is our pleasure to thank Anahita Mirtabatabaei, Rush Patel, Giulia Piovan, Cenk Oguz Saglam, Sepehr Seifi and Carlos Torres for having contributed in various ways to the material in these notes and in the assignments.

Santa Barbara, California, USA
Waterloo, Ontario, Canada
6 Jul 2010 — 7 Apr 2016

Francesco Bullo
Stephen L. Smith

Part I

Planning

Sensor-based Planning

In this chapter we begin our investigation into motion planning problems for mobile robots. This chapter focuses on sensor-based motion planning, where a robot acquires information about its surroundings using onboard sensors. We consider the basic task of moving from A to B in an environment with obstacles. Whether or not a robot can succeed at this task will depend on its sensors and capabilities. In this chapter we

- (i) introduce three bug algorithms for sensor-based motion planning,
- (ii) define notions of optimality and completeness for motion planning algorithms, and
- (iii) study the completeness of the three bug algorithms.

This chapter is inspired by the original article (Lumelsky and Stepanov 1987), the treatment (Lumelsky 2006), the first chapter in (Choset et al. 2005) and the lecture slides (Dodds 2006; Hager 2006).

1.1 Problem setup and modeling assumptions

Motion planning is an important and common problem in robotics. In its simplest form, the motion planning problem is: how to move a robot from a “start” location to a “goal” location avoiding obstacles. This problem is sometimes referred to as the “move from A to B ” or the “piano movers problem” (how do you move a complex object like a piano in an environment with lots of obstacles, like a house).

In this first chapter, we consider a sensor-based planning problem for a point moving in the plane \mathbb{R}^2 . In other words, we assume the robot has a sensor and, based on the sensor measurements, it plans its motion from start to goal. To properly describe the motion planning problem, we need to specify: what capacities does the robot have? What information does the robot have?

In this chapter, we make the following assumptions on the robot and its environment. As illustrated in Figure 1.1, we are given

- a workspace W that is a subset of \mathbb{R}^2 or \mathbb{R}^3 , often just a rectangle;

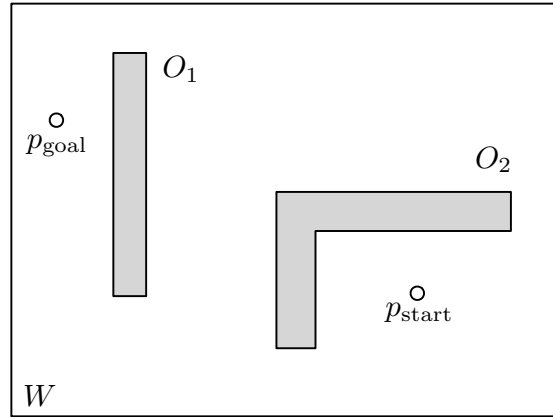


Figure 1.1: The environment for sensor-based planning

- some obstacles O_1, O_2, \dots, O_n ;
- a start point p_{start} and a goal point p_{goal} ; and
- a robot described by a moving point (that is, the robot has zero size).

We define the *free workspace* $W_{\text{free}} = W \setminus (O_1 \cup O_2 \cup \dots \cup O_n)$ as the set of points in W that are outside all obstacles. (Recall the definition of the set $A \setminus B = \{a \in A \mid a \notin B\}$, that is, all points in A that are not in B .)

Robot Assumptions We also make some assumptions on the capabilities and knowledge of the robot. We assume the robot:

- knows the direction towards the goal,
- knows the straight-line distance between itself and the goal,
- does not know anything about the obstacles (number, location, shape, etc),
- has a contact sensor that allows it to locally detect obstacles,
- can move either in a straight line towards the goal or can follow an obstacle boundary (possibly by using its contact sensor), and
- has limited memory in which it can store distances and angles.

We discuss in more detail later what these robot capabilities imply in terms of robot sensors and knowledge.

Our task is to plan the robot motion from the start point to the goal point. This plan is not a precomputed sequence of steps to be executed, but rather a policy to deal with the possible obstacles that the robot may encounter along the way.

Environment Assumptions Finally, we make some assumptions on the workspace and obstacles:

- the workspace is bounded,
- there are only a finite number of obstacles,
- the start and goal points are in the free workspace W_{free} , and
- any straight line drawn in the environment crosses the boundary of each obstacle only a finite number of times. (This assumption is easily satisfied for “normal objects” and we will use it later on to establish the correctness of our algorithms.)

In the next sections we see three different algorithms for planning the robot motion from start to goal, called Bug 0, Bug 1, and Bug 2. Each algorithm has slightly different requirements on the robot’s capabilities and knowledge. As a result, we will also see that they have different performance in finding paths from start to goal.

1.2 The Bug 0 algorithm

Starting from the scenario illustrated in Figure 1.1, suppose the robot heads towards the goal position from the start position. How does the robot handle collisions with obstacles? (Note that the robot sensor is local so that the robot only knows it has hit an obstacle.) We need a strategy to avoid the obstacle and move towards the goal destination.

What follows is our first motion planning algorithm.

The Bug 0 algorithm

```

1: while not at goal :
2:     move towards the goal
3:     if hit an obstacle :
4:         while not able to move towards the goal :
5:             follow the obstacle’s boundary moving to the left

```

Moving to the left means that the robot is just sliding along the obstacle boundary, i.e., circumnavigating the obstacle boundary in a *clockwise* fashion. The moving direction, left or right, is fixed but irrelevant. We only need to designate one preferred direction to turn once the robot hits an obstacle. A right-turning robot will circumnavigate an obstacle in a *counterclockwise* fashion. A right-turning robot follows the same path as a left-turning robot in a reflected world.

As shown in Figure 1.2 we label the point on the obstacle boundary where the robot hits the obstacle as p_{hit} and the point on the obstacle boundary where the robot leaves as p_{leave} .

Note: we are not being very careful clarifying whether the robot moves in discrete steps or in continuous time. For now imagine that the robot can move smoothly, visit all boundary points, take a measurement at each point and store the distance from the closest boundary point.

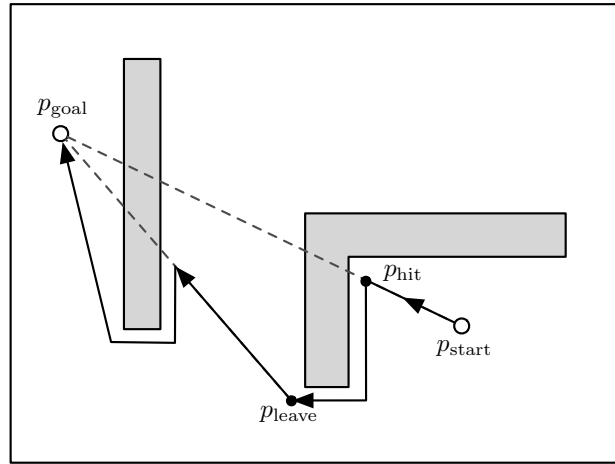


Figure 1.2: A successful execution of the Bug 0 algorithm

The Bug 0 algorithm does not always find a path to the goal Unfortunately, our Bug 0 algorithm does not work properly in the sense that there are situations (workspaces, obstacles, start and goal positions) for which there exists a solution (a path from start to goal) but the Bug 0 Algorithm does not find it. We will talk more later about the correctness of an algorithm and in particular about the notion of *completeness*.

This example in the Figure 1.3 clearly illustrates a periodic loop generated by the Algorithm Bug 0. At the end of each loop there is no complete progress to goal.

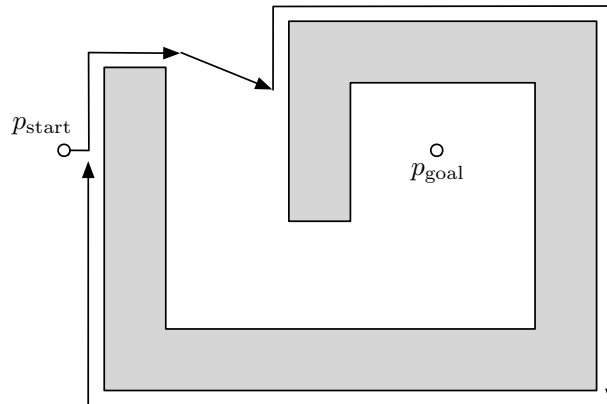


Figure 1.3: An unsuccessful execution of the Bug 0 Algorithm

1.3 The Bug 1 algorithm

The fact that Bug 0 does not always find a path to the goal may not be too surprising: The algorithm is not making use of all of the capabilities of the robot. In particular, the algorithm does not use

any memory, nor does it use the distance to the goal. This observation motivates our second smarter sensor-based algorithm for a more capable bug.

The Bug 1 algorithm

- 1: **while** not at goal :
 - 2: move towards the goal
 - 3: **if** hit an obstacle :
 - 4: circumnavigate it (moving to the left or right is unimportant). While circumnavigating, store in memory the minimum distance from the obstacle boundary to the goal
 - 5: follow the boundary back to the boundary point with minimum distance to the goal
-

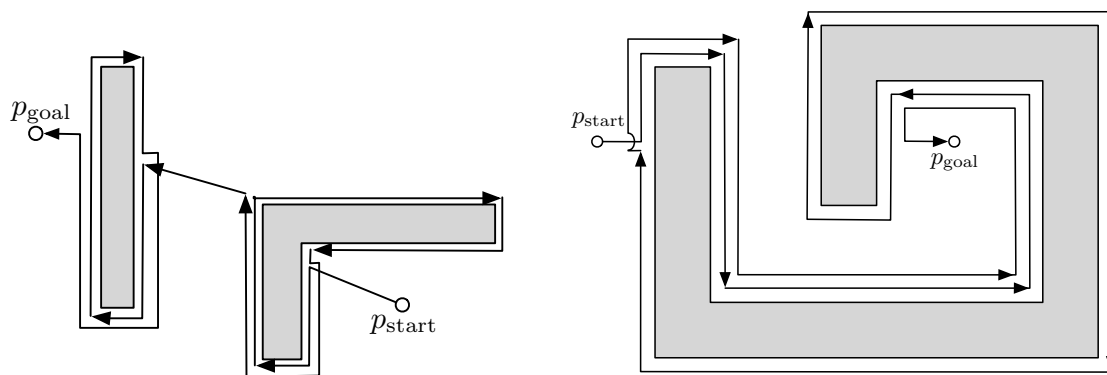


Figure 1.4: Two successful executions of the Bug 1 Algorithm

Note: the only difference between Bug 0 and Bug 1 is the reaction to the obstacle encounter, i.e., the behavior inside the **if** command.

1.3.1 Implementing Bug 1

The Bug 1 algorithm can be implemented as follows. In the simplest version, when the robot hits an obstacle at p_{hit} , it records the distance and direction to the goal. The robot then circumnavigates the obstacle, storing in memory a variable containing the minimum distance from its current position along the obstacle boundary to the goal. Regarding instruction 4, the circumnavigation is complete when the robot returns to the distance and direction it recorded at p_{hit} . The robot then partially circumnavigates the obstacle a second time until its distance to goal matches the minimum distance it has stored in memory.

In a more sophisticated version, the robot would additionally measure the distance it travels while circumnavigating the obstacle and therefore return to the closest point along the boundary using the shorter of the clockwise and counterclockwise paths. An instrument for measuring distance traveled is known as a *linear odometer*. If the robot moves at constant speed, then a clock suffices as a linear odometer: distance traveled is equal to speed times travel time. If the robot's

speed is variable, then one typically uses encoders in the robot wheels to measure the number of wheel rotations.

In summary, the Bug 1 robot must have memory for storing information about p_{hit} and for computing p_{leave} and it benefits from (but does not require) a linear odometer, i.e., an instrument to measure traveled distance.

1.3.2 Flowcharts

Before we proceed any further, it is useful to stop and discuss how to represent algorithms. So far we have adopted the *pseudocode* representation, i.e., a simplified English-like language that is midway between English and computer programming. It is also useful to understand how to represent our algorithms using *flowcharts*. Since flowchart representations can become quite large, they are typically useful for only simple programs.

According to [Wikipedia:pseudocode](#), “pseudocode is compact and informal high-level description of a computer programming algorithm that uses the structural conventions of a programming language, but is intended for human reading rather than machine reading.”

According to [Wikipedia:flowchart](#), “a flowchart is a type of diagram that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows.”

The four constitutive elements of a flowchart A flowchart consists of four symbols shown in Figure 1.5, which can be thought of as a graphical language.

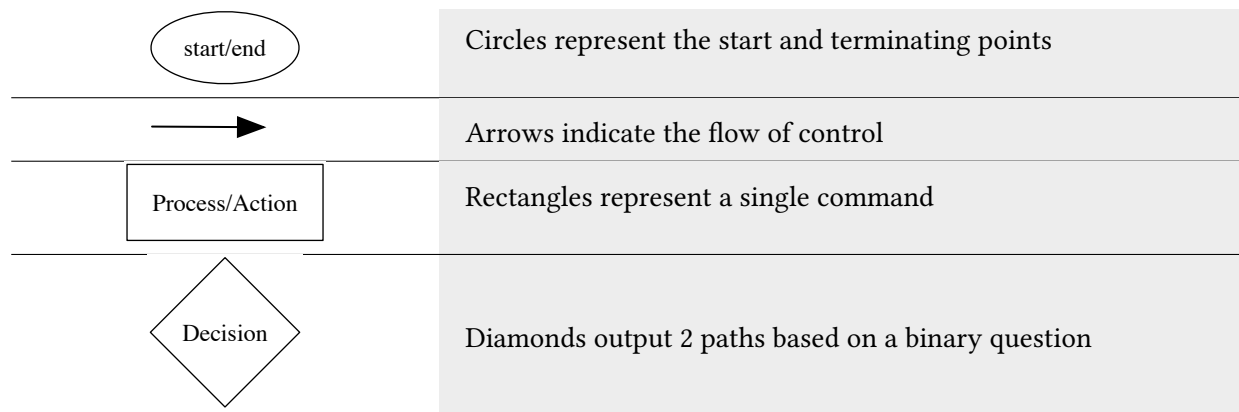


Figure 1.5: The four elements of a flowchart

Note: The dominant convention in drawing flowcharts is to have the flow of control go from top to bottom and left to right.

Flowchart representation for the Bug 1 Algorithm If you examine carefully the Bug 1 flowchart, you can clearly see that the algorithm may *end in failure*. This is indeed possible if the workspace

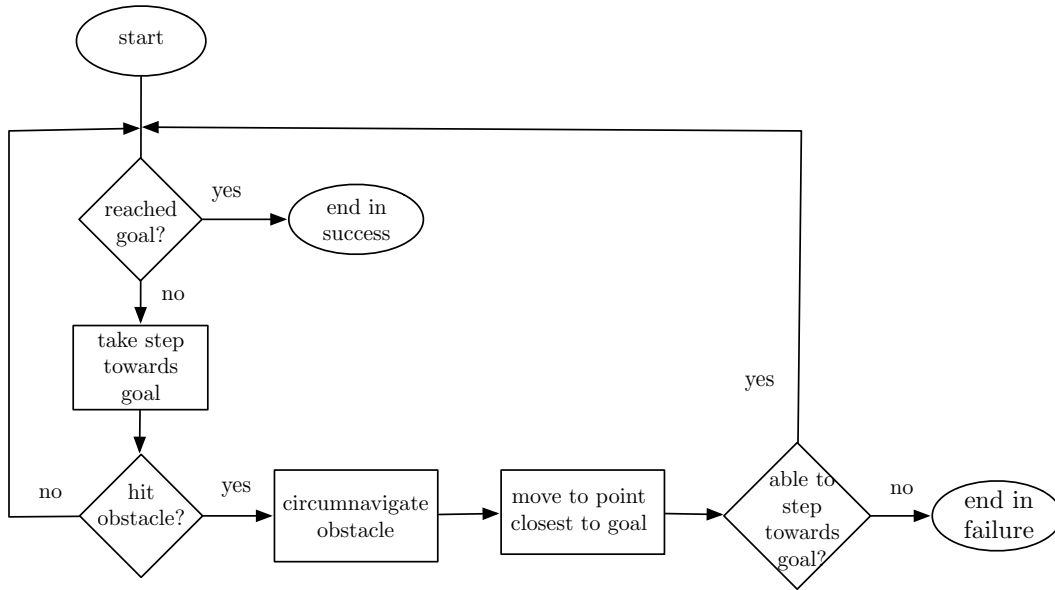


Figure 1.6: Flowchart representation for the Bug 1 Algorithm

is composed of disconnected components (that is, pieces of the free workspace that can not be connected with a path), and the start and goal locations belong to distinct disconnected components as shown in Figure 1.7.

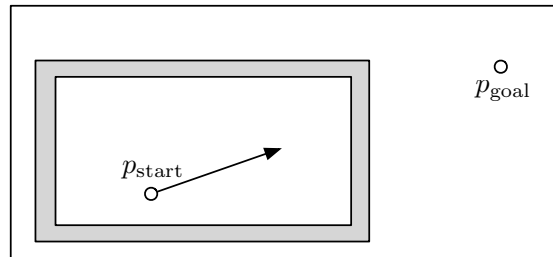


Figure 1.7: Environment for which no path exists from start to goal

1.3.3 The performance of the Bug 1 algorithm

Next, let us begin to rigorously analyze the Bug 1 algorithm. There are 2 desirable properties we wish to establish:

- optimality (with respect to some desirable metric), and
- completeness (i.e., correctness in an appropriate sense).

For now let us begin with the simpler analysis, i.e., the study of optimality. We are interested in seeing how efficient is the algorithm in completing its task in a workspace with arbitrary obstacles.

The question is: what is the length of the path generated by the Bug 1 Algorithm in going from start to goal? While a precise answer is hard to obtain in general, we can ask three more specific questions:

- (i) Will Bug 1 find the shortest path from start to goal?
- (ii) How long will the path found by Bug 1 be? Can we find a lower bound and an upper bound on the path length generated by Bug 1?
- (iii) Is there a workspace where the upper bound is required?

In order to answer these mathematical questions, it is good to have some notation:

$D :=$ length of straight segment from start to goal,
 $\text{perimeter}(O_i) =$ length of perimeter of the i th obstacle.

Theorem 1.1 (Performance of Bug 1). *Consider a workspace with n obstacles and assume that the Bug 1 algorithm finds a path to the goal. Assuming the robot is not equipped with a linear odometer, the following properties hold:*

- (i) Bug 1 does not find the shortest path in general;
- (ii) the path length generated by Bug 1 is lower bounded by D ;
- (iii) the path length generated by Bug 1 is upper bounded by $D + 2 \sum_{i=1}^n \text{perimeter}(O_i)$; and
- (iv) the upper bound is reached in the workspace described in Figure 1.8.

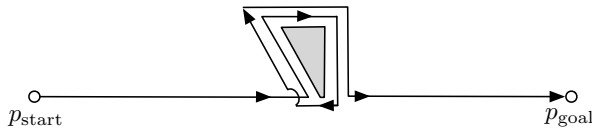


Figure 1.8: An example environment where the upper bound on the performance of Bug 1 is achieved, as stated in Theorem 1.1(iii).

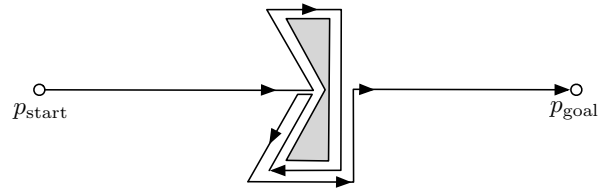


Figure 1.9: An example environment where the upper bound on the performance of Bug 1 is achieved for a robot with a linear odometer.

Note: Assume now that the robot is equipped with a linear odometer. After circumnavigating an obstacle, the robot can therefore move along the shorter of the two paths from hit point to leave point. In this case, the robot will travel at most $1/2$ of the perimeter to return to the leave point and the upper bound on the path length can be strengthened to $D + \frac{3}{2} \sum_{i=1}^n \text{perimeter}(O_i)$. An example environment achieving this bound is shown on the right of Figure 1.9.

1.4 The Bug 2 algorithm

Let us now try to design a new algorithm that generates shorter paths than Bug 1. The perceived problem with Bug 1 is that each obstacle needs to be fully explored before the robot can proceed towards the goal. Can we do better, i.e., can we decide to leave the obstacle without traversing all its boundary?

We use the term *start-goal line* to refer to the unique line that passes through the start point and goal point. The start-goal line is the dashed line intersecting the two obstacles as shown on the left of Figure 1.10. To aid in the design of Bug 2, we begin with a preliminary version.

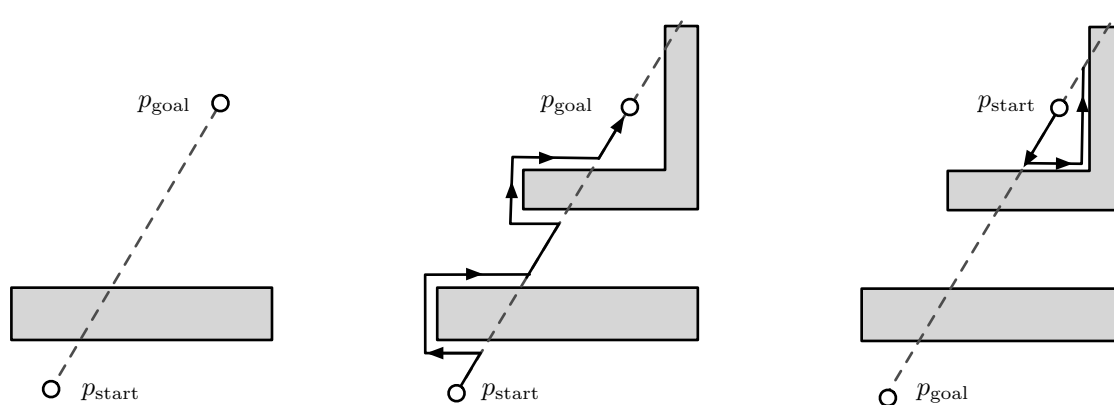


Figure 1.10: The start-goal line, a successful execution of the Bug2.prelim algorithm, and an unsuccessful execution of the Bug 2.prelim algorithm

The Bug 2.prelim algorithm

- 1: **while** not at goal :
 - 2: move towards the goal (along the start-goal line)
 - 3: **if** hit an obstacle :
 - 4: follow the obstacle's boundary (moving either left or right), until you encounter the start-goal line again and are able to move towards the goal
-

The example execution on the right of Figure 1.10 amounts to an undesired periodic cyclic trap again. How do we improve and possibly fix this misbehavior in our algorithm? It turns out that a small fix is sufficient. For convenience we repeat the entire algorithm, but the only difference is the addition of the requirement that the leave point be *closer to the goal* than the hit point!

From the left of Figure 1.11 we see that Bug 2 finds a path to the goal where Bug 2.prelim did not. Let us briefly mention the requirements of Bug 2, although we will compare them more carefully in Section 1.4.3. For Bug 0, we assumed the robot can sense direction towards the goal, and it knows when it has reached the goal. For Bug 1, we assumed the robot can measure and store in memory the distances and directions to goal point that it senses along the boundary. Bug

The Bug 2 algorithm

- 1: **while** not at goal :
 - 2: move towards the goal (along the start-goal line)
 - 3: **if** hit an obstacle :
 - 4: follow the obstacle's boundary (the turn direction is irrelevant), until you encounter the start-goal line again *closer to the goal* and are able to move towards the goal
-

2 can also measure and store in memory the distance and direction to the goal. In particular, it stores distance and direction at the hit point and compares these two quantities with the ones it senses along the boundary.

1.4.1 Monotonic performance and its implications

Here we briefly discuss why our correction to the Bug 2.prelim algorithm is indeed helpful and has a chance to render the algorithm correct. Consider the function of time equal to the distance between the robot and the goal point (this distance is a function of time because the robot is moving). Let us plot this distance function of time along the execution of the Bug 2 algorithm.

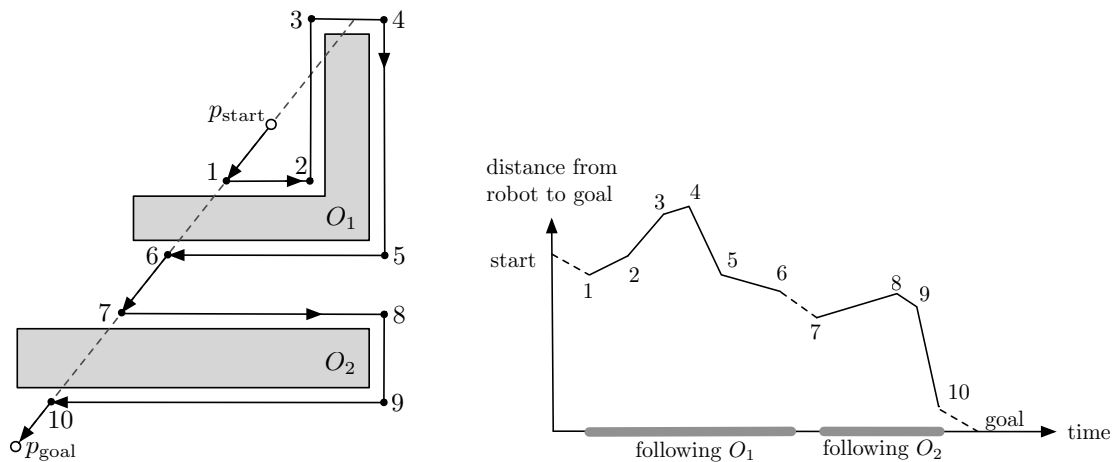


Figure 1.11: The monotonic performance of Bug 2

As Figure 1.11 illustrates, the leave point p_{leave} is closer to the goal than the hit point p_{hit} . Of course, throughout the search phase (while the robot is moving along the boundary to find the optimal leave point) the distance function may not always decrease, but after the search phase is complete, the robot will indeed be closer to the goal.

This discussion establishes that the distance between the robot and the goal is a monotonically decreasing function of time, when the robot is away from any obstacles.

Monotonicity immediately implies that there can be no cycles (and therefore no infinite cycles) in the execution of the algorithm. This lack of cycle is true because (1) the leave point is closer to the goal than the hit point and (2) when the robot moves away from the obstacle the distance

continues to decrease. Therefore, it is impossible for the robot to hit the same obstacle again at the same hit point.

1.4.2 The performance of the Bug 2 algorithm

Let us now analyze the performance of the Bug 2 algorithm. With the usual convention (D is the distance between start and goal and $\text{perimeter}(O_i)$ is the perimeter of the i th obstacle), we have the following results.

Theorem 1.2 (Performance of Bug 2). *Consider a workspace with n obstacles and assume that the Bug 2 algorithm finds a path to the goal. The following properties hold:*

- (i) *Bug 2 does not find the shortest path in general;*
- (ii) *the path length generated by Bug 2 is lower bounded by D ;*
- (iii) *the path length generated by Bug 2 is upper bounded by $D + \sum_{i=1}^n c_i \text{perimeter}(O_i)/2$, where c_i is the number of intersections of the start-goal line with the boundary of obstacle O_i .*

Proof. The first statement is obvious. Regarding the second statement, the lower bound is the same as that for Bug 1 – no surprise here. Regarding the third statement, the upper bound is different from that for Bug 1 and is due to the following fact: each time Bug 2 hits an obstacle (at a hit point), it might need to travel the entire obstacle's perimeter before finding an appropriate leave point. So for each pair of hit point and leave point (2 intersection points), the Bug 2 travels at most the obstacle's perimeter. ■

1.4.3 Comparison between bug algorithms

After introducing the Bug 1 and Bug 2 algorithms, let us compare in terms of path length. We ignore Bug 0 in this discussion because we already established it is not correct via the example in Figure 1.3. Without losing any generality, let us assume both the Bug 1 and Bug 2 algorithms are left-turning.

Example where Bug 2 finds shorter path than Bug 1 Recall that the Bug 2 algorithm was introduced in an attempt to find shorter paths by not fully exploring the boundary of each encountered obstacle. Indeed it appears that Bug 2 works better in our running example, as shown in Figure 1.12.

However, it is not clear that this fact must hold true for any problem (remember that a problem is determined by the workspace, the obstacles, and start and goal positions).

Counterexample where instead Bug 1 is better than Bug 2 Looking at the environment in Figure 1.13, we see that Bug 1 explores the entire perimeter of the obstacle only once and then moves to point l before leaving the obstacle for the goal. Bug 2, on the other hand, takes a much longer path. Whenever a robot implementing Bug 2 encounters a “left finger” in the environment, then note that the robot ends up traveling all the way back near to the start position!

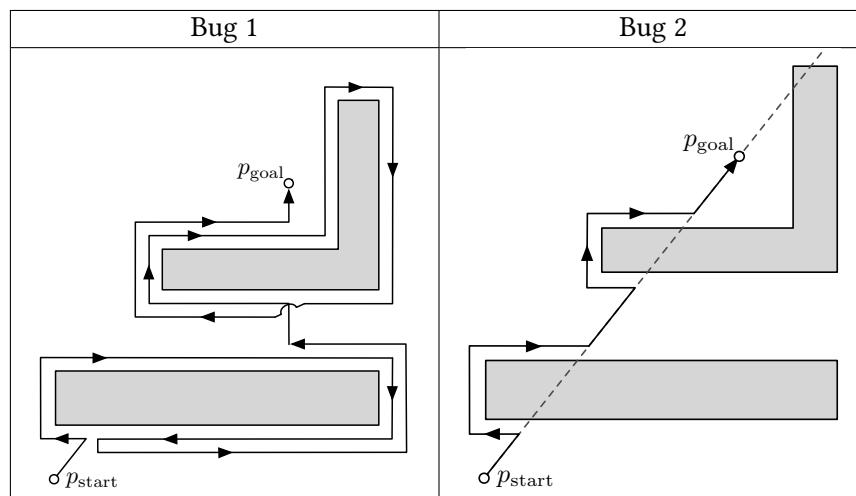


Figure 1.12: Example environment in which Bug 2 is better than Bug 1

Summary of path length Bug 1 performs an *exhaustive search* by examining all possible leave points before committing to the optimal choice. Bug 2 is a greedy algorithm that takes the first-available leave point that is closer to the goal without any specific performance guarantee. While it is impossible to predict which of the two will outperform in an arbitrary environment, we may say that Bug 2 will outperform Bug 1 in many simple environments but Bug 1 has more predictable performance.

Summary of robot capabilities The bug algorithms have slightly different assumptions on the sensors and capabilities needed by the robot. Table 1.1 summarizes these capabilities: direction to

| Sensor/Capability | Bug 0 | Bug 1 | Bug 2 |
|-----------------------------|-------|----------|-------|
| direction to goal | yes | yes | yes |
| distance to goal | no | yes | yes |
| memory | no | yes | yes |
| linear odometry | no | optional | no |
| angular odometry or compass | no | yes | yes |

Table 1.1: Summary of the robot capabilities needed to implement each bug algorithm

goal, distance to goal, memory, linear odometry, and a new capability called angular odometry or a compass. Recall that linear odometry was an optional capability for Bug 1 and it allowed the robot to return to the leave point using the shorter of the two paths along the boundary.

The new capability of “angular odometry or compass” is needed in order for the robot to store the direction to goal in memory. This is required in Bug 1 to determine when circumnavigation is complete, and in Bug 2 to determine when the start-goal line is encountered. The direction to goal must be stored in a known reference frame (for example, as a counterclockwise angle relative

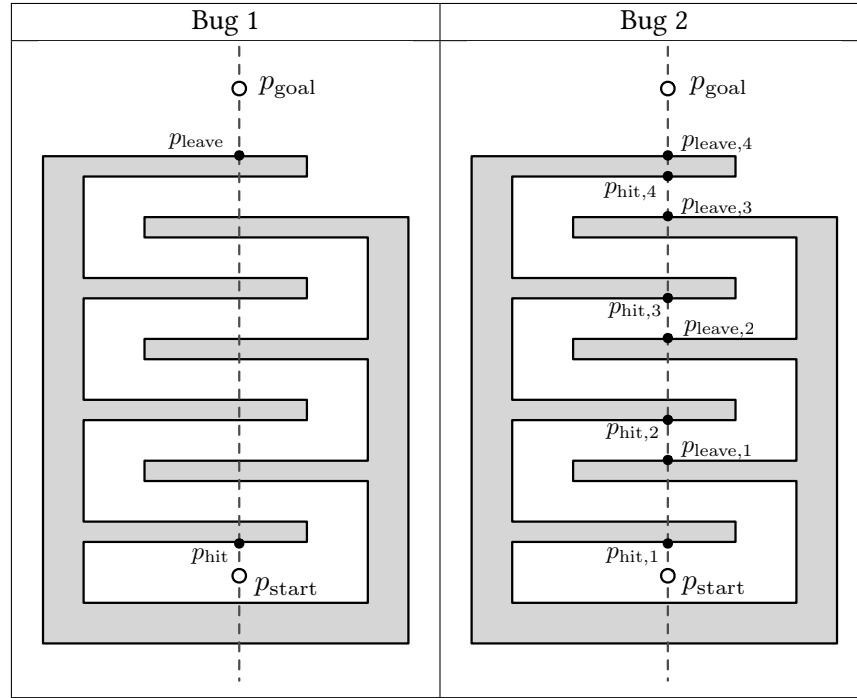


Figure 1.13: Example environment where Bug 1 is better than Bug 2. Bug 1 has one pair of hit and leave points. Bug 2 hits and then leaves the obstacle four times, as shown by the sequence of hit and leave points.

to a fixed x -axis) so that measured directions can be compared to the stored directions. The robot could define a local reference frame based on its heading, but this frame would rotate with the robot. It is not useful to store a direction in such a frame, unless the robot also somehow records the orientation of the frame when the direction was measured.

There are two possible fixes for this problem, illustrated in Figure 1.14. The first option is that the robot has a compass, and can then record the direction to goal relative to a fixed orientation as given by the compass. This is shown as the angle α_1 relative to north on the left of Figure 1.14. The second option is that the robot can use an angular odometer to measure changes in its heading. The robot can use its initial heading at p_{start} as the orientation for its reference frame. The direction to goal can be specified in this initial frame as the sum of two angles, as shown on the right of Figure 1.14: the angle α_2 can be measured with the aid of an angular odometer, and the angle α_3 is the output of the “direction to goal” sensor.

This discussion highlights some subtleties in the assumptions we have made on robot capabilities. While the Bug algorithms seem very simple at first glance, they actually require fairly strong assumptions on the sensing and knowledge of the robot. These assumptions are discussed in more detail by [Taylor and LaValle \(2009\)](#).

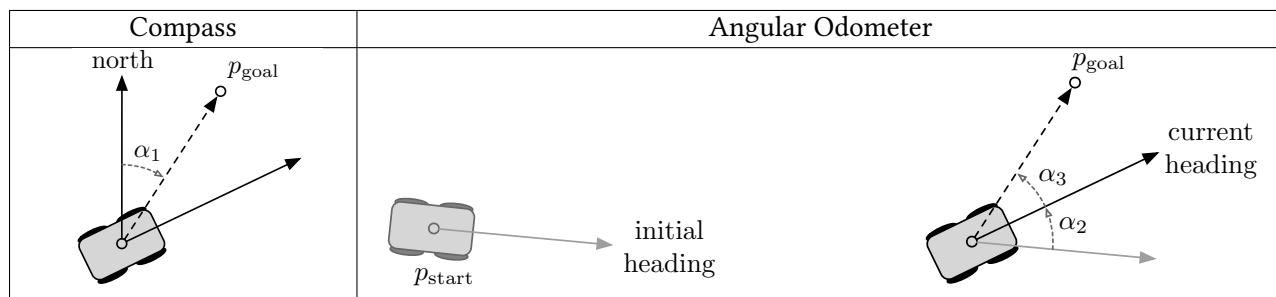


Figure 1.14: Left figure: the robot records the direction to goal as an angle α_1 relative to north. Right figure: the robot measures the direction to goal as $\alpha_2 + \alpha_3$, where α_2 must be measured from angular odometry. The angle α_3 is what is given by the “direction to goal” sensor.

1.5 The completeness of the Bug 1 algorithm

Here we follow up on our previous discussions and formally define the completeness of an algorithm.

Definition 1.3. *An algorithm is complete if, in finite time,*

- (i) *it finds a solution (i.e., a path), if a solution exists, or*
- (ii) *it terminates with a failure decision, if no solution exists.*

Next, we establish that one of our proposed algorithms is indeed complete. This result was originally obtained by [Lumelsky and Stepanov \(1987\)](#).

Theorem 1.4 (Completeness for Bug 1). *The Bug 1 algorithm is complete (under the modeling assumptions stated early in the chapter).*

1.5.1 On the geometry of closed curves

To prove Theorem 1.4, we start by introducing a wonderful and useful geometric result.

Theorem 1.5 (The Jordan Curve Theorem). *Every non-self-intersecting continuous closed curve divides the plane into two connected parts. One part is bounded (called the inside) and the other part is unbounded (called the outside) and the curve is the boundary of both parts.*



Figure 1.15: A closed curve divides the plane into two parts: the inside and the outside.

Next, consider the following. Imagine that the curve describes the boundary of the obstacle. Given a start and goal point outside the curve (obstacle), connect the two points using a straight segment and count the number of intersections between the segment and the boundary of the obstacle.

It is easy to see that this number of intersections must be even (where we regard 0 as an even number). Each time the segment enters the inside of the curve, it must then return to the outside. A few examples are given in Figure 1.16.

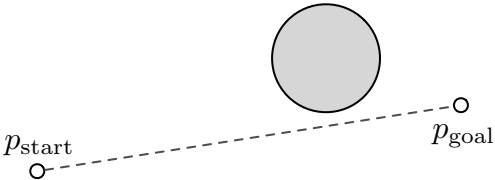
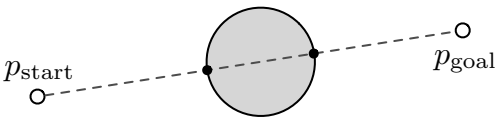
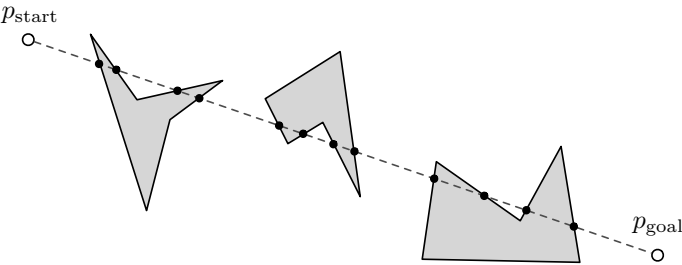
| | |
|---|------------------------------|
|  | 0 intersections |
|  | 2 intersections |
|  | even number of intersections |

Figure 1.16: The number of intersections between a line segment and a closed curve, where the endpoints of the segment lie on the outside of the curve, is even.

1.5.2 Proof of the completeness theorem

Here we prove Theorem 1.4. Recall that the Bug 1 algorithm is presented in pseudocode on page 7 and in flowchart on page 8.

By contradiction, assume Bug 1 does not find a path, even if a path exists Let us prove the theorem by contradiction. That is, we assume that Bug 1 is incomplete and we find a contradiction.

Consider the situation when a path from start to goal does exist, but Bug 1 does not find it. The flowchart description of Bug 1 implies the following statement: if Bug 1 does not find the path, then necessarily Bug 1 will either *terminate in failure in finite time* or *keep cycling forever*.

Bug 1 cannot keep cycling forever Suppose Bug 1 cycles forever. Because there is a finite number of obstacles, the presence of an infinite cycle implies the robot must hit the same obstacle more than once. Now, during the execution of Bug 1, the distance to the goal is a function of time that is monotonically decreasing when the robot is away from any obstacle. Moreover, when the robot hits an obstacle, the distance from p_{leave} to the goal is strictly lower than the distance from p_{hit} to the goal (this fact can be seen geometrically). Therefore, when the robot leaves an obstacle it is closer to the goal than any point on the obstacle. Hence, Bug 1 cannot hit the same obstacle twice.

Bug 1 cannot end in failure, if a path exists If Bug 1 is incomplete and a path actually exists, then the only possible result is that it terminates in failure. According to the Bug 1 flowchart, failure occurs when: the robot visits all the obstacle boundary points reachable from the hit point, moves to the boundary point p_{leave} closest to the goal, and is unable to move towards the goal. Consider now the segment from p_{leave} to the goal point. Because a path exists from start to goal, a path must also exist between p_{leave} and goal. By the Jordan Curve Theorem 1.5, there must be an even number of intersections between this segment and the obstacle boundary. Since p_{leave} is one intersection, there must exist at least another one. Let p_{other} be the intersection point closest to the goal. Now: the point p_{other} has the following properties:

- (i) lies on the obstacle boundary,
- (ii) is reachable from p_{leave} (because it is reachable from the goal), and
- (iii) is closer to the goal than p_{leave} .

These facts are a contradiction with the definition of p_{leave} .

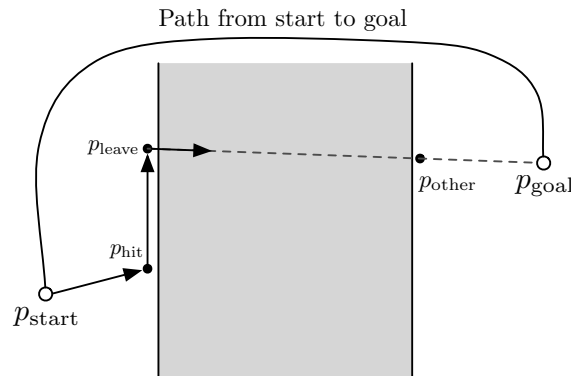


Figure 1.17: An illustration of the two points p_{leave} and p_{other}

1.6 Appendix: Operations on sets

A *set* is a collection of objects. For example, using standard conventions, \mathbb{N} is the set of natural numbers, \mathbb{R} is the set of real numbers, and \mathbb{C} is the set of complex numbers.

If a is a point in A , we write $a \in A$. Sets may be defined in one of two alternative ways. A set is defined by either listing the items, e.g., $A = \{1, 2, 3\}$, or by describing the items via a condition they satisfy, e.g.,

$$A = \{n \in \mathbb{N} \mid n < 4\}.$$

By convention, the empty set is denoted by \emptyset .

The *cardinality* of a set A is the number of elements in A . The set of natural numbers has infinite cardinality. As illustrated in Figure 1.18, the three core set operations are union, intersection, and set-theoretic difference.

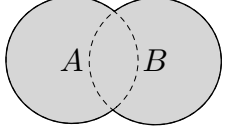
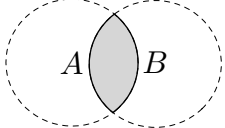
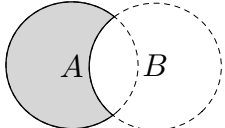
| | |
|---|---|
|  | The <i>union</i> of two sets A and B is the collection of points which are in A or in B (or in both): $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$. |
|  | The <i>intersection</i> of two sets A and B is the set that contains all elements of A that also belong to B : $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$. |
|  | The <i>set-theoretic difference</i> of two sets A and B , also known as the relative complement of B in A is the set of elements in A that are not in B : $A \setminus B = \{a \in A \mid a \notin B\}$. |

Figure 1.18: The three set operations: union, intersection, and set-theoretic difference.

A set A is a *subset* of B , written $A \subset B$, if and only if any member of A is a member of B , that is, $a \in A$ implies $a \in B$. Intervals are subsets of the set of real numbers \mathbb{R} and are denoted as follows: for any $a < b \in \mathbb{R}$, we define $[a, b] = \{x \mid a \leq x \leq b\}$, $]a, b[= \{x \mid a < x < b\}$, $]a, b] = \{x \mid a < x \leq b\}$, $[a, b[= \{x \mid a \leq x < b\}$, $[a, \infty[= \{x \mid a \leq x\}$, and $] - \infty, b] = \{x \mid x \leq b\}$.

The *Cartesian product* of two sets A and B is the set of all possible ordered pairs whose first component is a member of A and the second component is a member of B : $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$. An example is the 2-dimensional plane $\mathbb{R}^2 = \mathbb{R} \times \mathbb{R}$. We denote the unit circle on the plane by \mathbb{S}^1 , the unit sphere in \mathbb{R}^3 by \mathbb{S}^2 . We denote the 2-torus by $\mathbb{T}^2 = \mathbb{S}^1 \times \mathbb{S}^1$.

A set $S \subset \mathbb{R}^d$ is *convex* if for every pair of points p and q within S , every point on the straight line segment that joins them (\overline{pq}) is also within S : If $p \in S$ and $q \in S \Rightarrow \overline{pq} \subset S$, where $\overline{pq} = \{\alpha p + (1 - \alpha)q \mid \alpha \in [0, 1]\}$.

1.7 Exercises

E1.1 On the right-turning Bug 0 algorithm (25 points).

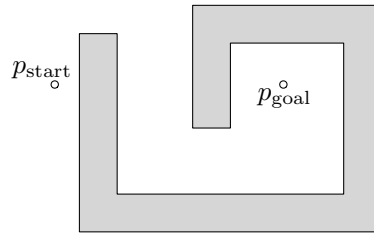
- (i) (10 points) For the following loose pseudo-code implementation of a *right-turning Bug 0 algorithm*, draw a flowchart (as discussed in Section 1.3):
 - 1: **while** not at goal location :
 - 2: move towards goal
 - 3: **if** hit an obstacle :
 - 4: **while** not able to move towards goal :
 - 5: follow obstacle moving to the right until can head towards goal
- (ii) (5 points) Draw an environment for which the right-turning Bug 0 algorithm never reaches the goal location, even though a path from start to goal does exist. Include both start and goal locations along with arrows indicating the path of the robot. Label several notable points along the path with the letters (A, B, C, \dots).
- (iii) (5 points) Sketch a graph of the distance between the robot and the goal location along the path you drew for part (ii). Use the labels from part (ii) to clarify the distance to the goal at the several notable points along the path.
- (iv) (5 points) Draw an environment which contains a path from start to goal, but for which both the left and right turning Bug 0 algorithms fail to find it. Include two sketches, showing the path of the left-turning algorithm and the path of the right-turning algorithm.

E1.2 A switching Bug 0 algorithm (25 points).

- (i) (10 points) Draw a flowchart for the following pseudo-code implementation of the *switching Bug 0 algorithm*:
 - 1: choose an initial direction (either left or right)
 - 2: **while** not at goal :
 - 3: move towards the goal
 - 4: **if** hit an obstacle :
 - 5: **while** not able to move towards goal :
 - 6: follow the obstacle boundary moving in the chosen direction
 - 7: switch the choice of direction
- (ii) (5 points) Draw an environment for which the switching Bug 0 algorithm never reaches the goal location, even though a path from start to goal does exist. Include both start and goal locations along with arrows indicating the path of the robot. Label several notable points along the path with the letters (A, B, C, \dots).
- (iii) (5 points) Sketch a graph of the distance between the robot and the goal location along the path you drew for part (ii). Use the labels from part (ii) to clarify the distance to the goal at the several notable points along the path.

E1.3 The Bug 2 algorithm (25 points).

- (i) (10 points) Draw a flowchart for the Bug 2 algorithm (the correct version, not the first attempt Bug 2.prelim).
- (ii) (5 points) Draw the path of a robot using the Bug 2 algorithm for the environment in the following figure. Label several notable points along the path with the letters (A, B, C, \dots).



- (iii) (5 points) Sketch a graph of the distance between the robot and the goal location along the path you drew for part (ii). Use the labels from part (ii) to clarify the distance to the goal at different points along the path.
- (iv) (5 points) Explain in a few accurate sentences why the Bug 2 algorithm will always reach the goal location when it is possible to do so.

Hint: Consider making a monotonicity argument based on your graph for part (iii).

E1.4 On the completeness of the Bug 2 algorithm (5 points).

Even though the Bug 2 algorithm always finds a path when one exists, it is not *complete*. Do the following:

- (i) state what properties render an algorithm complete,
- (ii) explain why the Bug 2 algorithm is not complete,
- (iii) explain how to revise the pseudocode presented in Section 1.4 so that the revised Bug 2 is complete,
- (iv) provide the flowchart for the revised Bug 2 algorithm.

E1.5 On the Jordan Curve Theorem (15 points).

- (i) (5+5 points) Given a workspace with n circular obstacles and start and goal points in the free workspace, let k denote the number of times that the straight segment from start to goal crosses an obstacle boundary. What is a lower bound and what is an upper bound on k ? For each bound, sketch an example where the bound is achieved.
- (ii) (5 points) What if the obstacles are arbitrary polygons with 5 edges – what are the lower and upper bounds in this case? Sketch an example where the upper bound is achieved.

E1.6 Programming: Lines and segments (30 points).

In this exercise you are asked to begin implementing a number of basic algorithms that perform geometric computations. Consider the following planar geometry functions:

`computeLineThroughTwoPoints` (10 points)

Input: two distinct points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ on the plane.

Output: parameters (a, b, c) defining the line $\{(x, y) \mid ax + by + c = 0\}$ that passes through both p_1 and p_2 . Normalize the parameters so that $a^2 + b^2 = 1$.

`computeDistancePointToLine` (10 points)

Input: a point q and two distinct points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ defining a line.

Output: the distance from q to the line defined by p_1 and p_2 .

`computeDistancePointToSegment` (10 points)

Input: a point q and a segment defined by two distinct points (p_1, p_2) .

Output: the distance from q to the segment with extreme points (p_1, p_2) .

For each function, do the following:

- (i) explain how to implement the function, possibly deriving analytic formulas, and characterize special cases,

- (ii) program the function, including correctness checks on the input data and appropriate error messages, and
- (iii) verify your function is correct on a broad range of test inputs.

Hints: To check two points are distinct, use a tolerance equal to 0.1^8 . Regarding `computeDistancePointToLine`, compute the orthogonal projection of q onto the line. Regarding `computeDistancePointToSegment`, the distance depends on whether or not the orthogonal projection of q onto the line defined by p_1 and p_2 belongs or not to the segment between p_1 and p_2 .

E1.7 **Programming: Polygons (30 points).**

Requires Exercise (E1.6).

A polygon with n vertices is represented as an array with n rows and 2 columns. Consider the following functions:

`computeDistancePointToPolygon` (15 points)

Input: a polygon P and a point q .

Output: the distance from q to the closest point in P , called the distance from q to the polygon.

`computeTangentVectorToPolygon` (15 points)

Input: a polygon P and a point q .

Output: the unit-length vector u tangent at point q to the polygon P in the following sense: (i) if q is closest to a segment of the polygon, then u should be parallel to the segment, (ii) if q is closest to a vertex, then u should be tangent to a circle centered at the vertex that passes through q , and (iii) the tangent should lie in the counter-clockwise direction.

Hint: Determine which segment or vertex of P is closest to q to determine whether to use the segment or vertex tangent case.

For each function, do the following:

- (i) explain how to implement the function, possibly deriving analytic formulas, and characterize special cases,
- (ii) program the function, including correctness checks on the input data and appropriate error messages, and
- (iii) verify your function is correct on a broad range of test inputs.

Programming Note: It is convenient to learn to visualize points, vectors and polygons in your chosen programming environment. To visualize a red square in Matlab (RGB triplet [1,0,0]), one can execute the commands:

```
» mysquare = [0 0; 0 1; 1 1; 1 0];
» fill(mysquare(:,1), mysquare(:,2), [1,0,0]);
» axis([-0.5 1.5, -0.5 1.5]);
```

In Python, the plotting tools are provided in a module called `matplotlib`. We can plot the same red square as follows:

```
» import matplotlib.pyplot as plt
» mysquare = [[0, 0], [0, 1], [1, 1], [1, 0]]
» square = plt.Polygon(mysquare, fc="r")
» plt.gca().add_patch(square)
» plt.axis([-0.5, 1.5, -0.5, 1.5])
» plt.show()
```

E1.8 Programming Project: The Bug 1 algorithm (80 points).

Requires Exercises (E1.6) and (E1.7).

In this programming project, you are asked to implement the Bug 1 algorithm. For your convenience we provide below a detailed pseudocode implementation of a simple Bug algorithm called BugBase; this pseudocode implementation contains all detailed steps required to execute a bug algorithm, but does not have any logic for getting around obstacles.

The BugBase algorithm

Input: Two locations *start* and *goal* in W_{free} , a list of polygonal obstacles *obstaclesList*, and a length *step-size*

Output: A sequence, denoted *path*, of points from *start* to the first obstacle between *start* and *goal* (or from *start* to *goal* if no obstacle lies between them). Successive points are separated by no more than *step-size*.

```

1: current-position = start
2: path = [start]
3: while distance(current-position, goal) > step-size :
4:     find polygon closest to current-position
5:     if distance from current-position to closest polygon < step-size :
6:         return "Failure: There is an obstacle lying between the start and goal" and path
7:     compute new current-position by taking a step of length step-size towards goal
8:     path = [path, current-position]
9: path = [path, goal]
10: return "Success" and path

```

Perform the following tasks and document them in a concise project report:

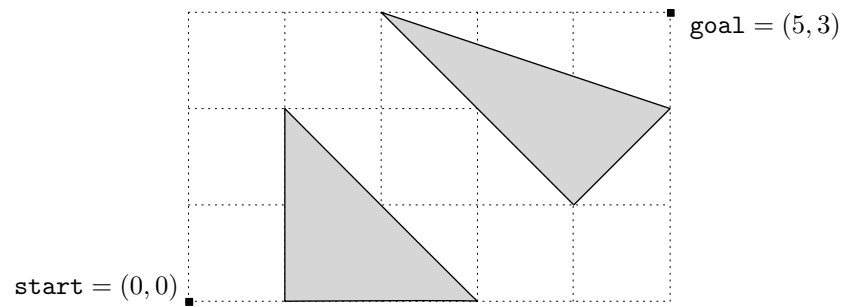
- (i) (15 points) Sketch a flowchart and implement the BugBase algorithm.
- (ii) (5 points) Describe in a paragraph how you will modify BugBase to implement the Bug 1 algorithm. Explain the roles of the geometric functions in Exercises (E1.6) and (E1.7) and what new logic will be needed.
- (iii) (40 points) Implement a fully-functioning version of Bug 1, described as follows:

`computeBug1`

Input: Two locations *start* and *goal* in W_{free} , a list of polygonal obstacles *obstaclesList*, and a length *step-size*

Output: A sequence, denoted *path*, of points from *start* to *goal* or returns an error message if such a path does not exist. Successive points are separated by no more than *step-size* and are computed according to the Bug 1 algorithm.

- (iv) (20 points) Test your program on the following environment:
`start = (0, 0)` and `goal = (5, 3)`
`step-size = 0.1`
`obstaclesList = { {(1, 2), (1, 0), (3, 0)}, {(2, 3), (4, 1), (5, 2)} }`



Include in your report a plot of the path taken by your bug from `start` to `goal`, the total path length and the computing time, and a plot of the distance from the bug's position to the goal as a function of time.

Motion Planning via Decomposition and Search

In this chapter we continue our investigation into motion planning problems. As motivation for our interest in planning problems, let us remind the reader that the ultimate goal in robotics is the design of autonomous robots, that is, the design of robots capable of executing *high-level instructions* without having to be programmed with extremely detailed commands. Moving from A to B is one such simple high-level instruction. In this chapter we

- (i) study techniques for decomposing the continuous robot workspace into convex regions,
- (ii) define roadmaps, which encode the decomposed workspace, and
- (iii) introduce graph algorithms for computing point-to-point paths in roadmaps.

The first part of this chapter on decomposition is inspired by Chapter 13 in (de Berg et al. 2000).

2.1 Problem setup and modeling assumptions

The sensor-based planning problems we studied in the previous chapter are also referred to as *closed-loop* planning problems in the sense that the robot actions were functions of the robot sensors. The loop here is the following: the robot moves, then it senses the environment, and then it decides how to move again.

In this chapter we begin our discussion about open-loop planning. By *open-loop*, as compared with sensor-based and closed-loop, we mean the design of algorithms for robots that do not have sensors, but rather have access to a map of the environment. Open-loop and closed-loop strategies are synonyms for feedforward and feedback control.

As in the previous chapter, we are given

- a workspace that is a subset of \mathbb{R}^2 or \mathbb{R}^3 , often just a rectangle,
- some obstacles, say O_1, \dots, O_n ,

- a start point and a goal point, and
- a robot described by a moving point.

As in the previous chapter, we define the *free workspace* $W_{\text{free}} = W \setminus (O_1 \cup O_2 \cup \dots \cup O_n)$, see Figure 2.1. We continue to postpone more realistic and complex problems where the robot has a shape, size and orientation.

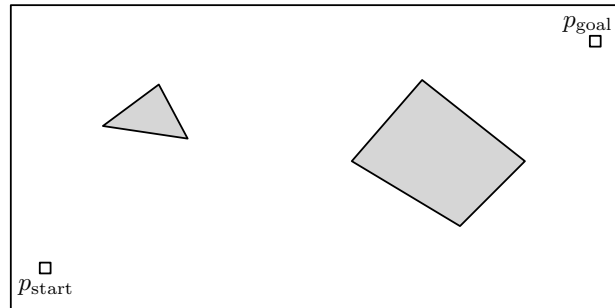


Figure 2.1: An example free workspace with start and goal locations

Our task is to plan the robot motion from the start point to the goal point via a precomputed open-loop sequence of steps to be executed. We want to design a motion plan under the following modeling assumptions.

Robot Assumptions The robot has the following capacities:

- knows the start and goal locations, and
- knows the workspace and obstacles.

World Assumptions The workspace has the following properties:

- the workspace is a bounded polygon,
- there are only a finite number of obstacles that are polygons inside the workspace, and
- the start and goal points are inside the workspace and outside all obstacles.

It is instructive now to compare these new robot and world assumptions with the ones in the previous chapter for sensor-based closed-loop planning. The similarities are the following: (1) the robot is still just a point, it has no size, shape, or orientation, and (2) the robot's motion is omni-directional (i.e., the robot can move in every possible direction). The differences are the following: (3) the robot has no sensors, but rather knowledge of the free workspace, and (4) planning is now a sequence of pre-computed steps, whereas before sensor-based algorithms are a policy on how to deal with possible obstacles encountered along the way.

2.1.1 Polygons

In these notes, a *polygon* is a plane figure composed by a finite chain of segments (called *sides* or *edges*) closing in a loop. The points where the segments meet are called *vertices* (or *corners*). Polygons are always assumed to be simple, i.e., their boundary does not cross itself. Programming-wise, it is convenient to represent a polygon by a counter-clockwise ordered sequence of vertices. (It is our convention not to repeat the first vertex as last.)

As illustrated in Figure 2.1, this chapter assumes that the workspace is a polygon, that each obstacle is a polygon (i.e., a polygonal hole inside the workspace), and that the total number of vertices in workspace and obstacles is n .

2.1.2 Run-time of an algorithm

In this chapter we will be interested in characterizing the amount of time it takes for a motion planning algorithm to run, called its *run-time*, rather than the length of the path it produces. The run-time of an algorithm is given by the number of computer steps required to execute the code. For the purposes of these notes, we assume that addition, multiplication and division of two numbers can be performed in one computer step. Similarly, we assume a single computer step is required to test if one number is equal to, greater than, or less than another number. Clearly, the run-time of an algorithm is a function of the input fed to the algorithm, and in particular its size.

Then, given an input to the algorithm of size n (for example, a polygonal obstacle with n vertices), it is useful to know the run-time as a function of the input size n . Counting the exact number of computation steps can be a tricky endeavor, and so one focuses on determining how the run-time scales with the input size n . That is, if an algorithm takes exactly $5n^2 + 4n + 1$ computer steps, it is customary to drop all constants and all lower-order terms, and simply report the run-time as $O(n^2)$.

Formally, the big- O notation is defined as follows: Given two positive functions $f(n)$ and $g(n)$ where n is an arbitrary natural number, we write $f \in O(g)$, g , if there exist a number n_0 and a positive constant K such that $f(n) \leq Kg(n)$ for all $n \geq n_0$. Thus, if we write that $f \in O(g)$, we are saying that the function f is less than or equal to the function g (up to a multiplicative constant and for large n). For example, in Figure 2.2 the function $f(n)$ grows linearly with n and $g(n)$ grows with the square of n . Thus, we have $f \in O(g)$.

2.2 Workspace decomposition

We start with two useful geometric ideas.

Convexity A set S is *convex* if for any two points p and q in S , the entire segment \overline{pq} is also contained in S . Examples of convex and non-convex sets are drawn in Figure 2.3. (Here we assume that the set S is a subset of the Euclidean space \mathbb{R}^d in some arbitrary dimension d .) For polygons, convexity is related to the *interior angles* at each vertex of the polygon (each vertex of a polygon

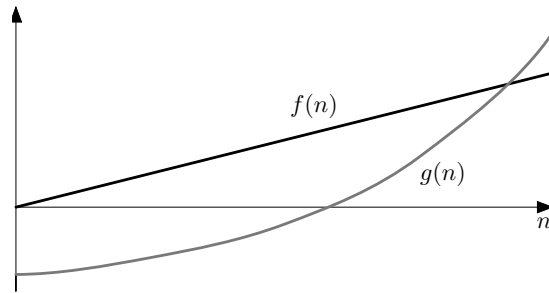


Figure 2.2: The function $f(n)$ grows linearly with n . The function $g(n)$ grows with the square of n .

has an interior and an exterior angle): a polygonal set is convex if and only if each vertex is *convex*, i.e., it has an interior angle less than π . A vertex is instead called *non-convex* if its interior angle is larger than π .

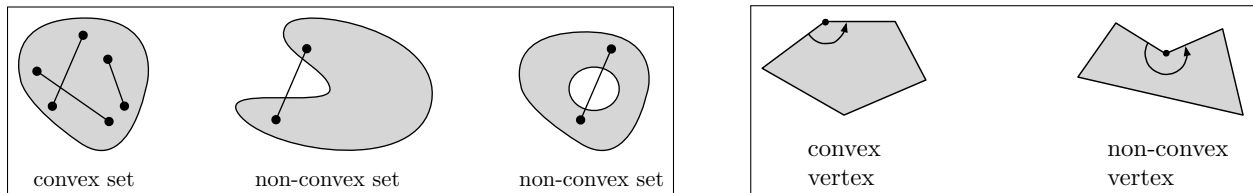


Figure 2.3: Examples of convex and non-convex sets. A convex set cannot have any hole. Polygons have convex and non-convex interior angles.

Planning in non-convex sets via convex decompositions Let us now use the notion of convexity for planning purposes. If the start point and the goal point belong to the same convex set, then the segment connecting the two points is an obstacle-free path. If, instead, the free workspace is not convex, then the following figure and algorithmic ideas provide a simple effective answer.

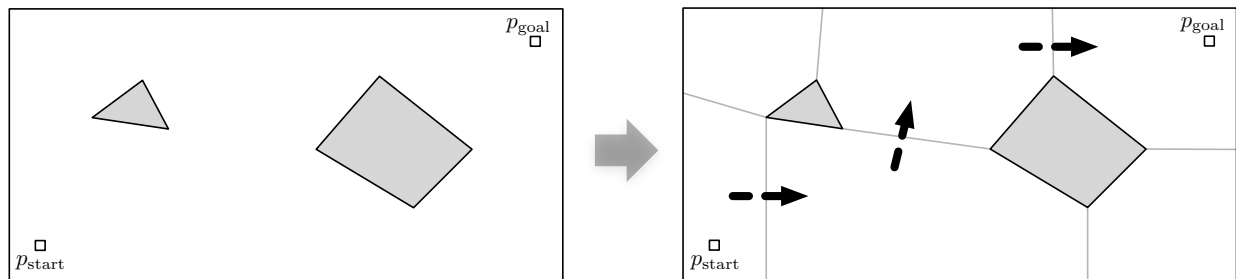


Figure 2.4: Planning through a non-convex environment by moving from a convex subset to another

2.2.1 Decomposition into convex subsets

For complex non-convex environments, we use an algorithm to decompose the free workspace into the union of convex subsets, such as triangles, convex quadrilaterals or trapezoids (quadrilaterals with at least one pair of parallel sides). The following nomenclature is convenient:

- (i) the *triangulation* of a polygon is the decomposition of the polygon into a collection of triangles, and
- (ii) the *trapezoidation* of a polygon is the decomposition of the polygon into a collection of trapezoids. (We allow some trapezoids to have a side of zero length and therefore be triangles.)

It is easy to see that a polygon can be triangulated in multiple ways (e.g., consider the two possible diagonals of a square). In what follows we present an algorithm to trapezoidate, i.e., decompose into trapezoids, a polygon with polygonal holes.

The sweeping trapezoidation algorithm

Input: a polygon possibly with polygonal holes

Output: a set of disjoint trapezoids, whose union equals the polygon

- 1: initialize an empty list \mathcal{T} of trapezoids
 - 2: order all vertices (of the obstacles and of the workspace) horizontally from left to right
 - 3: **for** each vertex selected in a left-to-right sweeping order :
 - 4: extend vertical segments upwards and downwards from the vertex until they intersect an obstacle or the workspace boundary
 - 5: add to \mathcal{T} the new trapezoids, if any, generated by these segment(s)
-

The algorithm is among a class of algorithms studied in computational geometry; feel free to inform yourself about this topic at [Wikipedia:computational geometry](https://en.wikipedia.org/wiki/Computational_geometry). An execution of the algorithm is illustrated in the Figure 2.5. Note that trapezoids T_3 and T_7 are degenerate, i.e., they are triangles.

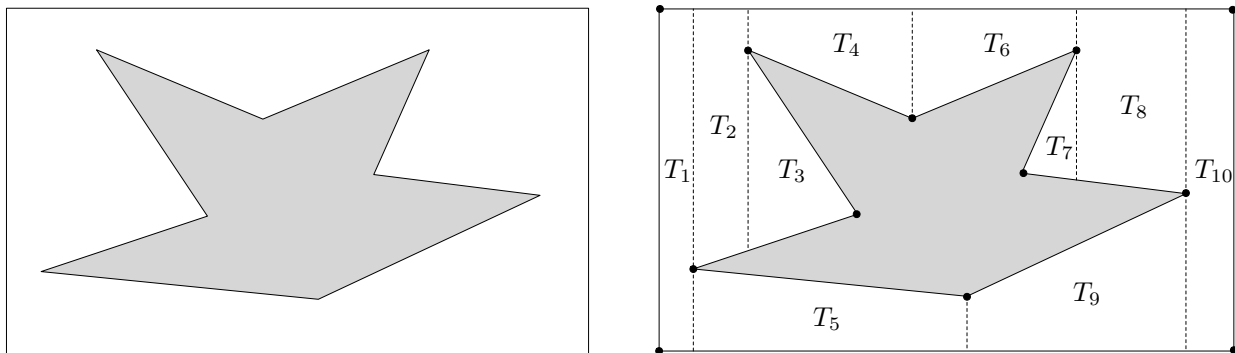


Figure 2.5: A trapezoidation of a workspace into trapezoids T_1, \dots, T_{10}

2.2.2 The sweeping trapezoidation algorithm

To understand in more detail how the sweeping trapezoidation algorithm is implemented, consider a workspace in which the boundary is an axis-aligned rectangle and every obstacle vertex has a unique x -coordinate. As an example, see Figure 2.5. Since all x -coordinates are unique, each line segment s_i describing an obstacle boundary has a *left endpoint* ℓ_i and a *right endpoint* r_i , where the x -coordinate of the left endpoint is smaller than that of the right endpoint. We write this segment as $s_i = [\ell_i, r_i]$.

To visualize the order in which vertices are processed by the algorithm in step 3:, we define a sweeping vertical line L moving from left to right. When the sweeping line L hits an environment vertex, the vertex must connect two segments. This vertex can be categorized into one of six types (see Figure 2.6) as summarized in Table 2.1.

| Vertex Type | Endpoints at Vertex | Convex/Concave | Example |
|-------------|---------------------|----------------|-----------------|
| (i) | both left | convex | p_6 and p_8 |
| (ii) | both left | concave | p_3 |
| (iii) | both right | convex | p_2 and p_4 |
| (iv) | both right | concave | p_7 |
| (v) | one left one right | convex | p_1 |
| (vi) | one left one right | concave | p_5 |

Table 2.1: The six vertex types encountered during the trapezoidal decomposition algorithm

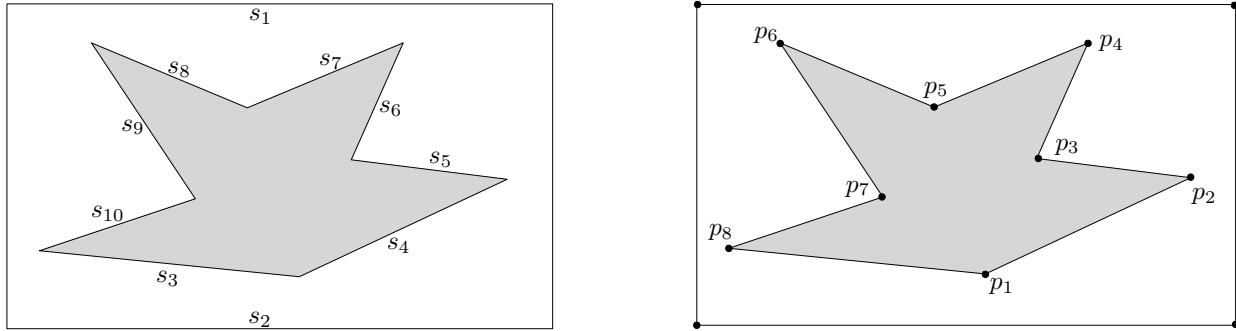


Figure 2.6: The line segments s_1, \dots, s_{10} and the obstacle vertices p_1, \dots, p_8 required by the algorithm

To execute steps 4: and 5: of the sweeping trapezoidation algorithm, we maintain a list \mathcal{S} of the obstacle segments intersected by the sweeping line L . The obstacle segments are stored in decreasing order of their y -coordinates at the intersection point with L . A key property of \mathcal{S} is that it changes only when L hits a new vertex. Thus, when the new vertex v is encountered, steps 4: and 5: update the list of trapezoids \mathcal{T} and the list of obstacle segments \mathcal{S} . The details of Steps 4: and 5: are as follow, and are illustrated in Figure 2.7 for each of the six vertex types of Table 2.1.

- 4.1: determine the type of vertex v , as shown in Table 2.1
- 4.2: update \mathcal{S} by adding obstacle segments starting at v and removing obstacle segments ending at v (i.e., add two segments, remove one segment and add one segment, or remove two segments, depending on vertex type as shown in Figure 2.7)
- 4.3: use \mathcal{S} to extend vertical segments upwards and downwards from v , that is, to find intersection points p_t and p_b above and below v (if any) – more detail on this computation is given in the paragraph below
- 5.1: add to \mathcal{T} zero, one or two new trapezoids depending on vertex type (see Figure 2.7)
- 5.2: update the left endpoints of the obstacle segments in \mathcal{S} above and below the vertex v

The type of v can be determined by checking its convexity and looking at the number of obstacle segments in \mathcal{S} that have v as an endpoint: zero obstacle segments implies v is of type (i) or (ii); one obstacle segment implies v is of type (v) or (vi); and two obstacle segments implies v is of type (ii) or (iv). The point p_t (respectively, p_b) is defined as the point where the vertical segment extended upward (respectively, downward) from v intersects an obstacle. In types (i) and (iii) both p_t and p_b are defined. In types (ii) and (iv) neither are defined as the upward and downward vertical segments immediately enter obstacles. In types (v), and (vi) only one of p_t and p_b is defined.

Instruction 5.2: updates the obstacle segments in \mathcal{S} to facilitate the following computations of trapezoids.

2.2.3 Run-time analysis of trapezoidation algorithm

Given a free workspace (i.e., workspace minus obstacles) with n vertices, the sweeping line is implemented (as in steps 1: and 2:) by sorting the vertices from left to right; that is, in increasing order of their x -coordinates. There are many well-known sorting algorithms including bubble sort, merge sort, quick sort, etc., and the best of these run in $O(n \log(n))$, where n is the number of items to be sorted (Cormen et al. 2001).

Next, for each vertex v in the sorted list, we perform the steps detailed in Figure 2.7. The runtime of these steps is dominated by the time needed to insert new segments into \mathcal{S} and delete old segments from \mathcal{S} . There are exactly two insert/delete operations for each vertex v , one for each segment that has v as an endpoint. If we maintain \mathcal{S} as a sorted array, then to insert/delete a segment, we need to scan through the array. Since there are n segments, the array can contain at most n entries, and insertion or deletion requires $O(n)$ time. We repeat this procedure for each of the n vertices, giving a total runtime of $O(n^2)$.

We can improve the runtime by using a more sophisticated data structure for \mathcal{S} that allows us to insert and delete segments more quickly. In particular, a *binary tree* can be used to maintain the ordered segments in \mathcal{S} . A segment can be inserted/deleted in $O(\log(n))$, instead of $O(n)$ time for the simple array implementation. With a binary tree, the sweeping decomposition algorithm can be implemented with a run-time belonging to $O(n \log(n))$ for a free workspace with n vertices. The details of binary trees are beyond the scope of this book, but can be found in (Cormen et al. 2001) or Chapter 13 in (de Berg et al. 2000).

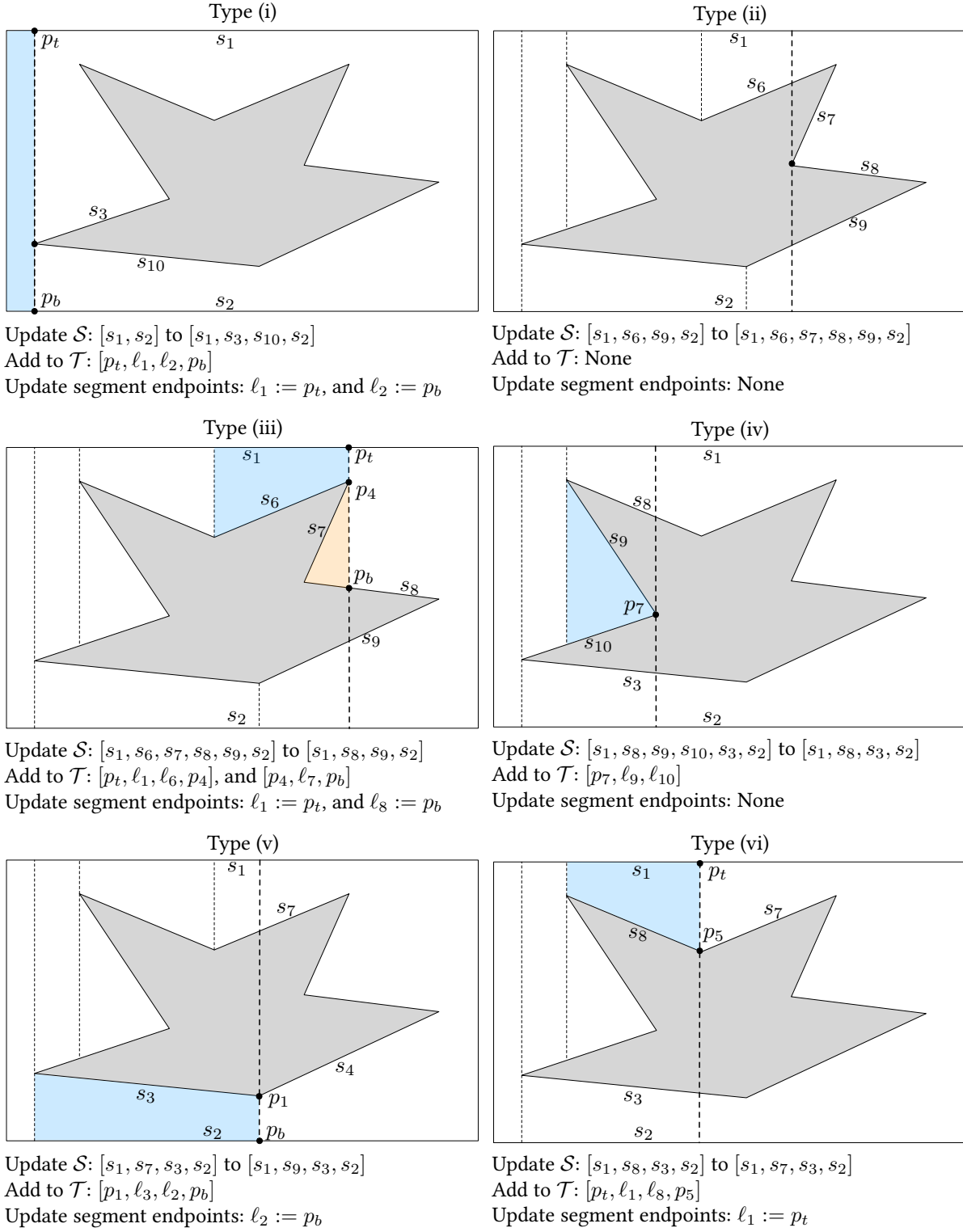


Figure 2.7: Classification of the six types of vertices. For each vertex type, the figure illustrates the actions performed in Step 4: and Step 5: to update list of trapezoids \mathcal{T} , the segment list \mathcal{S} , and the segment endpoints.

2.2.4 Navigation on roadmaps

Before proceeding, let us recall the three key ideas we introduced so far: (1) convexity leads to very simple paths, (2) if the free workspace is not convex, it is easy to navigate between neighboring convex sets, (3) a complex free workspace can be decomposed into convex subsets via, for example, the sweeping trapezoidation algorithm.

The next observation is that the sweeping trapezoidation algorithm (and other decomposition-into-convex-subsets procedures) can easily be modified to additionally provide a list of neighborhood relationships between trapezoids. In other words, we assume that we can compute an easy-to-navigate *roadmap*. The roadmap of a trapezoidation is computed as follows; an example is drawn in Figure 2.8.

The roadmap-from-decomposition algorithm

Input: the trapezoidation of a polygon (possibly with holes)

Output: a roadmap

- 1: label the center of each trapezoid with the symbol \diamond
 - 2: label the midpoint of each vertical separating segment with the symbol \bullet
 - 3: **for** each trapezoid :
 - 4: connect the center to all the midpoints in the trapezoid
 - 5: **return** the roadmap consisting of centers and connections between them through midpoints
-

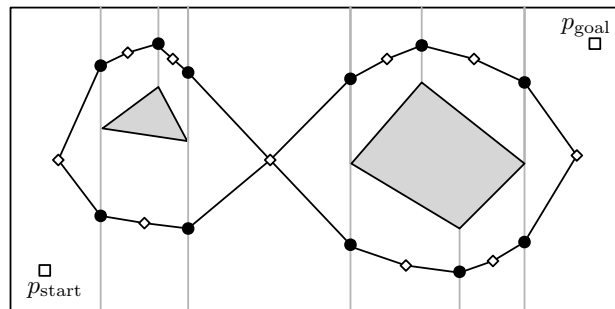


Figure 2.8: An example roadmap for a free workspace

As a result of this algorithm we obtain a *roadmap* specified as follows: (1) a collection of center points (one for each trapezoid), and (2) a collection of paths connecting center points (each path being composed of 2 segments, connecting a center to a midpoint and the same midpoint to a distinct center).

Note: It is not important what precise point we select as center of a trapezoid. We will see in the next section that the roadmap can be represented as a special type of “graph” that is generated from an environment partition and is called the “dual graph of a partition.”

Consider now a motion planning problem in a free workspace W_{free} that has been decomposed into convex subsets and that is now equipped with a roadmap. the planning-via-decomposition+search

algorithm described below provides a solution to this problem by combining various useful concepts.

The planning-via-decomposition+search algorithm

Input: free workspace W_{free} , start point p_{start} and goal point p_{goal}

Output: a path from p_{start} to p_{goal} if it exists, otherwise a failure notice. Either outcome is obtained in finite time.

- 1: compute a decomposition of W_{free} and the corresponding roadmap
 - 2: in the decomposition, find the start trapezoid Δ_{start} containing p_{start} and the goal trapezoid Δ_{goal} containing p_{goal}
 - 3: in the roadmap, search for a path from Δ_{start} to Δ_{goal}
 - 4: **if** no path exists from Δ_{start} to Δ_{goal} :
 - 5: **return** *failure* notice
 - 6: **else**
 - 7: **return** path by concatenating:
 - the segment from p_{start} to the center of Δ_{start} ,
 - the path from the Δ_{start} to Δ_{goal} , and
 - the segment from the center of Δ_{goal} to p_{goal} .
-

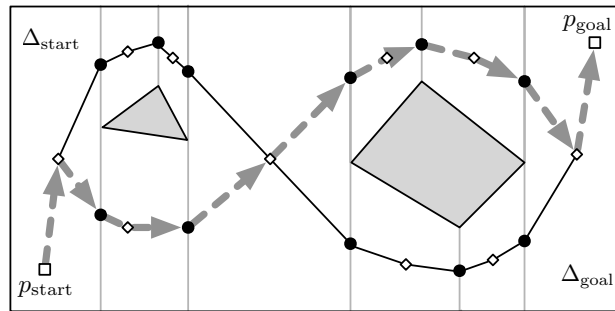


Figure 2.9: Illustration of the planning-via-decomposition+search algorithm

Note: by means of the decomposition, we have transformed a continuous planning problem into a discrete planning problem: a path in the free workspace is now computed by first computing a path in the discrete roadmap.

2.3 Search algorithms over graphs

2.3.1 Graphs

In the previous section and in the last algorithm, we did not specify (1) what mathematical object is the roadmap and (2) how to search it. In short,

(1) a roadmap is a “graph” and (2) paths in roadmaps are computed via “graph search algorithm.”

A *graph* is a pair (V, E) , where V is a set of *nodes* (also called *vertices*) and E is a set of *edges* (also called *links* or *arcs*). Every edge is a pair of nodes. (Note: A graph is not the graph of a function.) Often, the graph is denoted by the letter G , V is called the node set and E is called the edge set. If $\{u, v\}$ is an edge, then u and v are said to be *neighbors*.

(Note: graphs as defined here are sometimes referred to as “unweighted” and “undirected” graphs.)

Figure 2.10 contains three simple graphs and a more complex example with node set $V = \{n_1, \dots, n_{11}\}$ and edge set

$$E = \{e_1 = \{n_1, n_2\}, e_2 = \{n_1, n_3\}, e_3 = \{n_{11}, n_5\}, e_4 = \{n_6, n_7\}, e_5 = \{n_1, n_4\}, e_6 = \{n_1, n_6\}, e_7 = \{n_4, n_{10}\}, e_8 = \{n_4, n_6\}, e_9 = \{n_8, n_{10}\}, e_{10} = \{n_8, n_9\}, e_{11} = \{n_7, n_9\}, e_{12} = \{n_8, n_{11}\}\}.$$

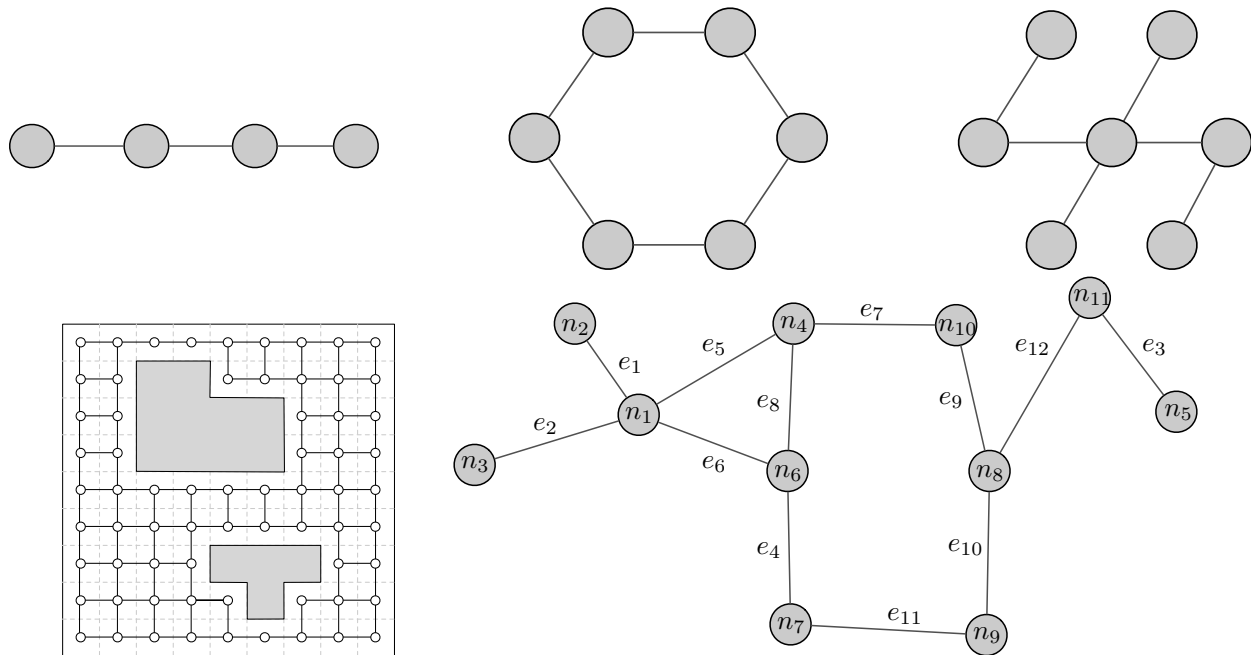


Figure 2.10: Example graphs. The top three graphs are: the *path graph* on 4 nodes, the *circular graph* (also called the *ring graph*) on 6 nodes, and a tree (see definition below).

Graphs are widely used in science and engineering in broad range of applications. Graphs are used to describe wireless communication networks (each node is an antenna and each edge is a wireless link), electric circuits (each node is a circuit node and each edge is either a resistor, a capacitor, an inductor, a voltage source or a current source), power grids (each node is a

power generator and each edge describes the corresponding pair-wise admittance), transportation networks (each node is a location and each edge is a route), social networks (each node is an individual and each edge describes a relationship between individuals), etc. It is a curious historical fact that [Euler \(1741\)](#) was perhaps the earliest scientist to study graphs and did so in the context of path planning problems (other early works include [Kirchhoff \(1847\)](#) and [Cayley \(1857\)](#) on electrical networks and theoretical chemistry, respectively). You are invited to read the wikipedia pages on [Wikipedia:graph](#) and [Wikipedia:graph theory](#).

Now that we know what a graph is, we are interested in defining paths and their properties. We will need the following simple concepts from graph theory.

- (i) A *path* is an ordered sequence of nodes such that from each node there is an edge to the next node in the sequence. For example, in Figure 2.10, a path from node n_1 to node n_5 is given by the sequence of nodes $(n_1, n_4, n_{10}, n_8, n_{11}, n_5)$ or equivalently, by the sequence of edges $(e_5, e_7, e_9, e_{12}, e_3)$. The *length of a path* is the number of edges in the path from *start node* to *end node*.
- (ii) Two nodes in a graph are *path-connected* if there is a path between them. A graph is *connected* if every two nodes are path-connected.
- (iii) If a graph is not connected, it is said to have multiple *connected components*. More precisely, a connected component is a subgraph in which (1) any two nodes are connected to each other and (2) all nodes outside the subgraph are not connected to the subgraph. For example, if a free workspace is disconnected, then the roadmap resulting from any decomposition algorithm will contain multiple connected components.
- (iv) A *shortest path between two nodes* is a path of minimum length between the two nodes. The *distance between two nodes* is the length of a shortest path connecting them, i.e., the minimum number of edges required to go from one node to the other. Note that a shortest path does not need to be unique, e.g., see Figure 2.10 and identify the two distinct shortest paths from node n_8 to node n_6 .
- (v) A *cycle* is a path with at least three distinct nodes and with no repeating nodes, except for the first and last node which are the same. A graph that contains no cycles and is connected is called a *tree*.

Roadmaps as dual graphs Having introduced some graph theory, let us review our definition of the roadmap. Given a decomposition of a workspace into a collection of trapezoids (or more general subsets), the *dual graph* of the decomposition is the graph whose nodes are the trapezoids and whose edges are defined as follows: there exists an edge between any two trapezoids if and only if the two trapezoids share a vertical segment. The *roadmap* generated by the workspace trapezoidation is the dual graph of the decomposition with the additional specifications: to each node of the dual graph (i.e., to each trapezoid) we associate a center location, and to each edge of the dual graph we associate a polygonal path that connects the two centers through the midpoint of the common vertical segment.

2.3.2 The breadth-first search algorithm

Now, let us turn our attention back to the topic of search. Search algorithms are a classic subject in computer science and operations research. Here we present a short description of one algorithm. We refer to (Cormen et al. 2001) for a comprehensive discussion about computationally-efficient algorithms for appropriately-defined data structures.

Problem 2.1 (The shortest-path problem). *Given a graph with a start node and a goal node, find a shortest path from the start node to the goal node.*

The breadth-first search algorithm, also called BFS algorithm, is one of the simplest graph search strategy and is optimal in the sense that it computes shortest paths. The algorithm proceeds in layers. We begin with the start node (Layer 0), and find all of its neighbors (Layer 1). We then find all unvisited neighbors of Layer 1 (Layer 2), and so on, until we reach a layer that has no unvisited neighbors. Each vertex v in Layer $k + 1$ is discovered from a vertex u in Layer k ; we refer u as the “parent” of v . An informal easy-to-understand description of this procedure is shown below. A corresponding execution is shown in Figure 2.11.

- 1: begin with the start node and mark as visited // The start node forms Layer 0
- 2: **for** each unvisited neighbor u of the start node :
- 3: mark u as visited, and set the start node as the parent of u . // The nodes u form Layer 1
- 4: **for** each unvisited neighbor v of the nodes in Layer 1 :
- 5: mark v as visited and record the neighbor from Layer 1 as the parent of v
- 6: repeat the process until you reach a layer that has no unvisited neighbors
- 7: **if** the goal node has been visited :
- 8: follow the parent values back to the start node, and return this sequence of vertices as the shortest path from start to goal
- 9: **else**
- 10: return a failure notice (i.e., the start and goal node are not path-connected)

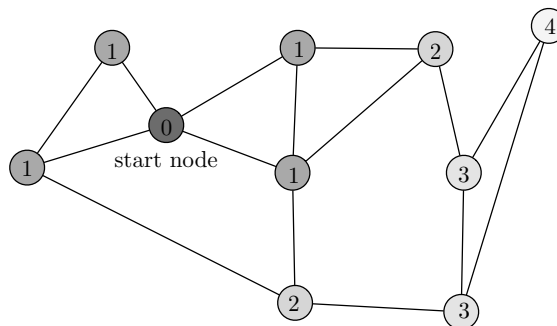


Figure 2.11: A sample execution of the breadth-first search strategy (in an unweighted graph). Each vertex is labelled with (and shaded according to) its layer, which turns out to be equal to the distance between that node and the start node.

To efficiently implement the BFS algorithm we introduce a useful data structure. A *queue* (also called a *first-in-first-out (FIFO) queue*) is a variable-size data container, denoted Q , that supports two operations: 1) the operation $\text{insert}(Q, v)$ inserts an item v into the back of the queue, and 2) the operation $\text{retrieve}(Q)$ returns (and removes) the item that sits at the front of the queue. An example of a queue is shown in Figure 2.12.

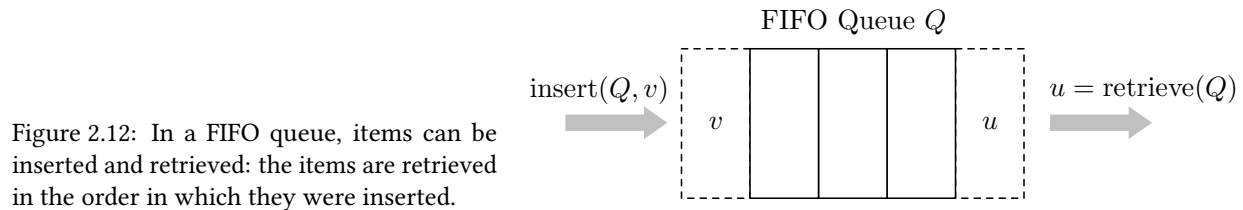


Figure 2.12: In a FIFO queue, items can be inserted and retrieved: the items are retrieved in the order in which they were inserted.

A queue can be implemented such that each insert and retrieve operation runs in $O(1)$ time; that is the runtime of each operation is independent of the number of items in the queue.

Programming Note: In Matlab, a simple way to manipulate a list and implement a FIFO queue is as follows. Define a queue as a row vector by:

```
» queue = [20, 21];
```

Insert the element 22 into the queue by adding it to the end of the vector:

```
» queue = [queue, 22];
```

Retrieve an element from the queue by:

```
» element = queue(1); queue(1) = [];
```

These same three commands in Python are:

```
» queue = [20, 21]
```

```
» queue.append(22)
```

```
» element = queue.pop(0)
```

Note: While these simple queue implementations in Matlab and Python will suffice in many applications, they are not efficient. In particular, the insert and retrieve operations are not constant time. The reason is that deleting the first element of an array (i.e., a vector) is a $O(n)$ operation, where n is the length of the array: After the element at the front of the array is deleted, each remaining element is sequentially shifted one place to the left in memory. To correct this, one typically implements a queue using a fixed-length array along with two additional pieces of data. The first gives the location of the element at the front of the queue, and the second gives the location of the back of the queue. These locations are updated after each insertion and retrieval.

Efficient queue implementations are available in Python using the Queue module, which can be including using the command `import Queue`.

Here is another more-specific pseudocode version of the breadth-first search strategy that lends itself to relatively-immediate implementation. The implementation uses an array parent that contains an entry for each node. The entry $\text{parent}(u)$ records the node that lies immediately

before u on the shortest path from v_{start} to u . We use `NONE` for parent values that have not yet been set, and `SELF` for the start node, whose parent is itself. Notice that the parent values also serve the purpose of marking nodes as visited or unvisited. A vertex u is unvisited if $\text{parent}(u) = \text{NONE}$ and visited otherwise.

The breadth-first search (BFS) algorithm

Input: a graph G , a start node v_{start} and goal node v_{goal}

Output: a path from v_{start} to v_{goal} if it exists, otherwise a failure notice

```

1: create an empty queue  $Q$  and insert( $Q, v_{\text{start}}$ )
2: for each node  $v$  in  $G$  :
3:      $\text{parent}(v) = \text{NONE}$ 
4:  $\text{parent}(v_{\text{start}}) = \text{SELF}$ 
5: while  $Q$  is not empty :
6:      $v = \text{retrieve}(Q)$ 
7:     for each node  $u$  connected to  $v$  by an edge :
8:         if  $\text{parent}(u) == \text{NONE}$  :
9:             set  $\text{parent}(u) = v$  and insert( $Q, u$ )
10:        if  $u == v_{\text{goal}}$  :
11:            run extract-path algorithm to compute the path from start to goal
12:        return success and the path from start to goal
13: return failure notice along with the parent values.
```

Note that the output of this algorithm is not necessarily unique, since the order in which the edges are considered in step 7: of the algorithm is not unique.

At the successful completion of BFS, we can use the parent values to define a set of edges $\{\text{parent}(u), u\}$ for each node u for which $\text{parent}(u) \neq \text{NONE}$. These edges define a *tree* in the graph G . That is, they form a connected graph that does not contain any cycles. An example is shown in Figure 2.13.

The last piece we need is a method to use the parent values to reconstruct the sequence of nodes on the shortest path from v_{start} to v_{goal} . This can be done as follows:

The extract-path algorithm

Input: a goal node v_{goal} , and the parent values

Output: a path from v_{start} to v_{goal}

```

1: create an array  $P = [v_{\text{goal}}]$ 
2: set  $u = v_{\text{goal}}$ 
3: while  $\text{parent}(u) \neq \text{SELF}$  :
4:      $u = \text{parent}(u)$ 
5:     insert  $u$  at the beginning of  $P$ 
6: return  $P$ 
```

Note: The *extract-path* is often implemented by inserting each u at the end of the array P

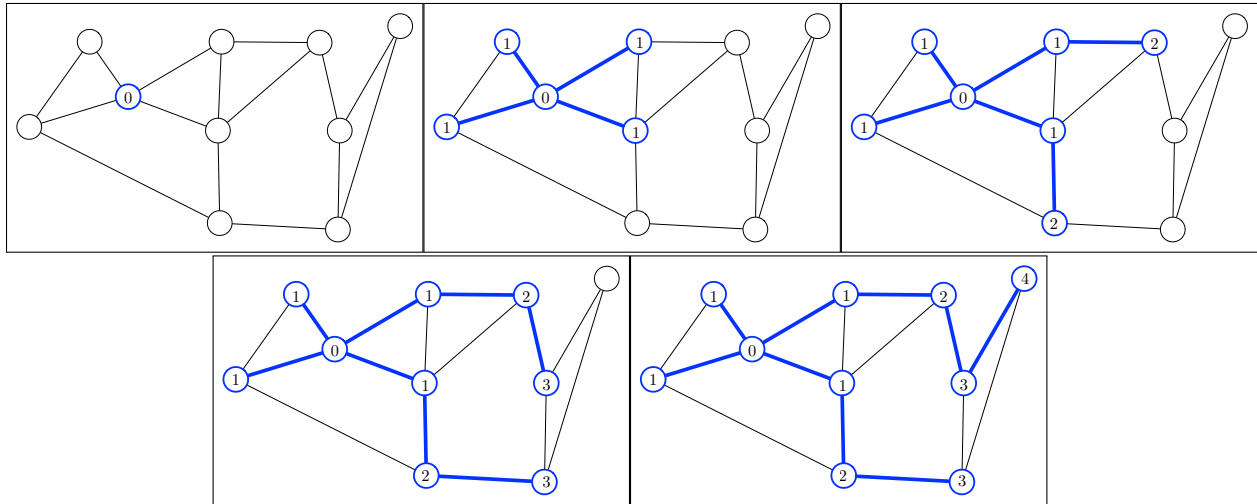


Figure 2.13: Execution of the breadth-first search algorithm. In the leftmost frame, the start node is colored in blue (Layer 0). The subsequent frames show Layers 1 to 4 as computed by BFS. We assume here that there is no goal node and just compute parent values, which are illustrated as blue edges. The blue edges define a subgraph called the *BFS tree*.

(rather than at the beginning), and then reversing the order of P prior returning it.

Note: Problem 2.1 is also referred to as the *single-source single-destination shortest paths problem*. There are efficient algorithms to compute the single-source all-destination shortest paths and the all-sources all-destinations shortest paths.

Note: We refer the reader interested in state-of-the-art implementations of graph manipulation algorithms to the Boost Graph Library (Siek et al. 2007) and its Matlab and Python packages called MatlabBGL and Boost.Python.

2.3.3 Representing a graph

One might wonder, how do we represent a graph that can be used as the input to the BFS algorithm. Recall that a graph $G = (V, E)$ is described by a set of nodes and a set of edges. For simplicity, label the nodes $1, \dots, n$, so that the question quickly reduces to: how do you represent the edge set E ?

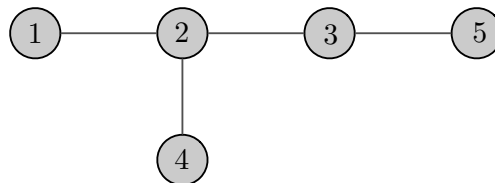


Figure 2.14: A sample graph for demonstrating graph representations

Representation #1 (Adjacency Table/List): The most common way to represent the edge set is

via a *lookup table*, that is, an array whose elements are lists of varying length. The lookup table contains the adjacency information as follows: the i -th entry is a list of all neighbors of node i . For example, for the graph above, the *adjacency table* (commonly called an *adjacency list*) would be

$$\begin{aligned}\text{AdjTable}[1] &= [2] \\ \text{AdjTable}[2] &= [1, 3, 4] \\ \text{AdjTable}[3] &= [2, 5] \\ \text{AdjTable}[4] &= [2] \\ \text{AdjTable}[5] &= [3]\end{aligned}$$

Representation #2 (Adjacency Matrix): In some mathematical and optimization problems, it is often convenient to represent the set of edges via the *adjacency matrix*, i.e., a symmetric matrix whose i, j entry is equal to 1 if the graph contains the edge $\{i, j\}$ and is equal to 0 otherwise. For example, for the graph above, the adjacency matrix would be

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

Representation #3 (Edge List): Finally, the set of edges may be represented as an array, where each entry is an edge in the graph. This representation of edges is called an *edge list*. For example, for the graph below, the edge list would be $\{1, 2\}, \{2, 3\}, \{2, 4\}, \{3, 5\}$.

Programming Note: In Matlab an adjacency table can be implemented using a structure called a cell array. Type `help paren` in Matlab for further information. For example, the adjacency table for the graph in Figure 2.14 would be specified as

```
» AdjTable{1} = [2];
» AdjTable{2} = [1, 3, 4];
» AdjTable{3} = [2, 5];
» AdjTable{4} = [2];
» AdjTable{5} = [3];
```

In Python there are two options to represent an adjacency table. The first is as a list of lists

```
»> AdjTable = [[2], [1, 3, 4], [2, 5], [2], [3]]
```

Note that since Python indexes from zero (i.e., the first entry of a python list is indexed with 0), the neighbors of node number i are contained in `AdjTable[i-1]`.

Alternatively, an adjacency list can implemented as a Python dictionary

```
»> AdjTable = {1: [2], 2: [1, 3, 4], 3: [2, 5], 4: [2], 5: [3]}
```

In this case the neighbors of node i are contained in `AdjTable[i]`.

2.3.4 Runtime of BFS

There are two questions that we will commonly ask when designing an optimization algorithm:

- (i) Is the algorithm complete? and
- (ii) How quickly does it run?

The answer to the first question is that the algorithm is complete: If a path exists in the graph from the start to the goal, then the BFS algorithm will find the shortest such path. We will skip the formal proof of this, but an interested reader can refer to a book on algorithms such as (Cormen et al. 2001) for a complete proof.

To answer the second question, we will determine the runtime of the algorithm as a function of the size of the input. In the case of BFS, the input is a graph G with node set V and edge set E . If we let n and m denote the number of vertices and the number edges in the graph G , respectively, then we can characterize the runtime of BFS as follows.

Theorem 2.2 (Run-time of the BFS algorithm). *Consider a graph $G = (V, E)$ with n vertices and m edges, along with a start and goal node. Then the runtime of the breadth-first-search algorithm is*

- $O(n + m)$ if G is represented as an adjacency table,
- $O(n^2)$ if G is represented as an adjacency matrix, and
- $O(n \cdot m)$ if G is represented as an edge list.

Proof. To analyze the run-time of our pseudocode for BFS, we will break it up into different pieces.

Initialization: First, initializing the queue and inserting v_{start} in step 1: can be done in $O(1)$ time. Initializing the parent values for every node in steps 2: to 4: requires a $O(1)$ time per node, for a total of $O(n)$ time.

Outer while-loop: Notice that this loop runs at most n times: each node is inserted into the queue at most once (when it is first found by the search) and one node is removed from the queue at each iteration of the while-loop. Thus, the algorithm spends $O(1)$ time on step 6: at each iteration of the while-loop, or $O(n)$ in total.

Inner for-loop: Now, let us look at step 7: of BFS. In each iteration of the while-loop this for-loop runs once for each neighbor node v . The time required for an iteration of the for-loop is $O(1)$, since it consists of looking at the value of $\text{parent}(u)$, and then possibly setting $\text{parent}(u)$ to v and inserting u into the queue: all three operations require $O(1)$ time. Notice that for each edge $\{u, v\}$ in the graph, the node v will be listed as a neighbor of u and the node u will be listed as a neighbor of v . Thus, the number of iterations of the for-loop over the entire execution of BFS is at most $2m$, which is $O(m)$.

Extracting the path: The *extract-path* algorithm extracts the path from the parent values in $O(n)$ time, since the while loop runs at most n times.

From this analysis we see that the overall runtime of BFS is $O(n + m)$. However, notice that in this analysis we were implicitly assuming that our graph is represented as an adjacency table,

since at step 7: we did not account for any extra computation in order to determine the neighbors of a given node: that is, we assumed that we had a list of the neighbors of each node.

Adjacency matrix: If instead the graph were represented as an adjacency matrix, then we would require $O(n)$ time to determine the neighbors of a node u : This would be done by searching through each of the n entries in the row of the adjacency matrix corresponding to node u . Thus, BFS runs in $O(n^2)$ when the graph is represented as an adjacency matrix.

Edge list: If an edge list is used, then the run-time is even worse, as the entire edge list must be searched at 7: to determine the neighbors of a node v . The run-time of searching the edge list is $O(m)$, giving a total runtime of $O(n \cdot m)$. ■

Note: If a graph is connected, then $m \geq n - 1$. In addition, for any undirected graph, there can be at most n choose 2 edges, and thus $m \leq n(n - 1)/2$. Thus, the number of edges is $m \in O(n^2)$. From this we see that the best graph representation for BFS is an adjacency table, followed by an adjacency matrix, followed by an edge list.

2.4 Exercises

E2.1 **Convexity (30 points).** Convexity is a key concept, not just in analysis of robotic algorithms but in optimization, control and numerous other fields.

- (i) (6 points) Explain in one paragraph (equal to a few sentences) when a set is convex and why it is useful for motion planning.

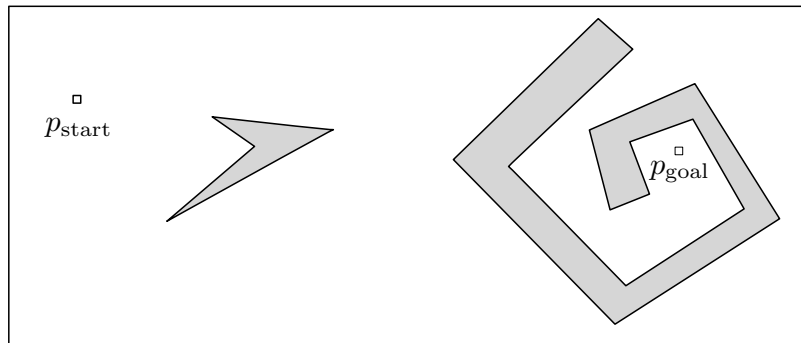
Hint: The question talks about sets, not polygons, so your answer should not refer to internal angles.

- (ii) (3 each points) For each of the following 5 properties, state if the property is true or false and provide a sketch supporting your statement.

- The intersection of any two overlapping convex sets is convex.
- The union of two convex sets is convex.
- Take a set Q and a point p outside Q . If Q is convex, then there exists a unique point in Q , say q , such that the distance between p and q is smaller than the distance between p and any other point in Q .
- Property (ii)c holds if Q is not convex.
- Take a line tangent to a convex set Q (that is, a line that just touches the boundary of Q but does not contain any point in the interior of Q). The line divides Q into two or more components.

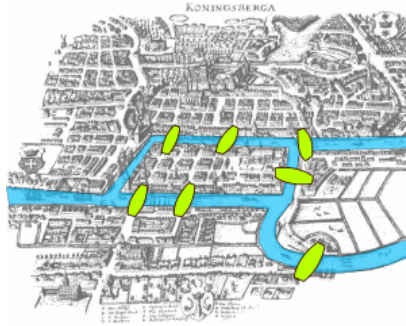
- (iii) (3 each points) Prove whether the statements (ii)a, (ii)c, and (ii)e are true or false.

E2.2 **Partitions and roadmaps (20 points).** For the free workspace in figure, do the following:



- (5 points) Sketch the free workspace and trapezoidate it (using the sweeping trapezoidation algorithm).
- (5 points) Sketch the dual graph for the trapezoidal partition and the roadmap (i.e., the dual graph with centers and with connecting polygonal paths through midpoints).
- (5 points) In the dual graph, draw a Breadth-First-Search tree with start node being the trapezoid containing the start point. Number the nodes in the graph based on the number of hops required to reach each node from the start node.
- (5 points) Sketch the continuous path from start point to goal point that a robot would actually follow in the workspace. The path depends upon the decomposition and upon the breadth-first search.

E2.3 **The seven bridges of Königsberg (15 points).** The city of Königsberg in Prussia (nowadays Kaliningrad in Russia) was founded on both sides of the Pregel River. The following figure (courtesy of Wikipedia) illustrates some main topographical features of this city, including four masses of land connected by seven bridges.



Here is the intriguing question that troubled the citizens of this small town during the 18th century:

Find a walk through the city that crosses each bridge exactly once.

Here are the questions:

- (i) How many vertices and edges are there? Draw a graph to represent the masses of land and the bridges connecting them, carefully label the vertices.

Hint: For this exercise only, we accept graphs with multiple edges connecting the same two nodes. Such graphs are sometimes called *multi-graphs* or *graphs with parallel edges*.

- (ii) Can you find the desired path? If so, please draw it and label the edges sequentially (1, 2, 3, ..., etc). If not, describe informally why not.
- (iii) Assuming that a single additional bridge can be constructed, modify the graph from part (i) and find a walk that crosses each bridge exactly once.

The comprehensive answer to these questions, given by [Euler \(1741\)](#), is today recognized as one of the earliest works on graph theory. (While the questions and the answers themselves are not all that important in general, Euler's contribution was the introduction of graph theory.)

E2.4 Counting triangles and trapezoids (15 points). For all of the following questions, explain your reasoning and provide an answer in terms of the numbers ℓ , n , and k_i .

- (i) (5 points) How many triangles are required to triangulate a polygonal workspace with ℓ vertices (and no obstacle inside)?
- (ii) (5 points) How many triangles are required to triangulate a polygonal workspace with ℓ vertices and one polygonal obstacle inside (assume that the obstacle has k vertices)?
- (iii) (5 points) How many trapezoids are required to trapezoidate a rectangular workspace (with 4 vertices) and n polygonal obstacles (each with k_i vertices) using the sweeping algorithm, in the worst-case?

Hint: Note that there are degenerate situations where the number of trapezoids is lower than for generic vertex positions. We are asking for the worst-case number, that is, for an upper bound.

E2.5 Computing time complexity (18 points). The time complexity of a program is the number steps or operations required for the program to execute. For the following brief programs, determine the order of their time complexity as a function of n , i.e., how the number of required operations scales with n . The order of the time complexity should be given in "big O" notation, that is, for example, $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^n)$, etc.

Hint: Count each addition, multiplication or division as having unit cost.

- (i) (3 points)
 - 1: $a = 0$
 - 2: **for** i from 1 to n :
 - 3: $a = a + 1$

(ii) (3 points)

```
1:  $a = 0$ 
2: for  $i$  from 1 to 10 :
3:      $a = a + 1$ 
```

(iii) (3 points)

```
1:  $a = 0$ 
2: for  $i$  from 1 to  $n$  :
3:      $a = a + 1$ 
4: for  $j$  from 1 to  $n$  :
5:      $a = a + 1$ 
```

(iv) (3 points)

```
1:  $a = 0$ 
2: for  $i$  from 1 to  $n$  :
3:     for  $j$  from 1 to  $n$  :
4:          $a = a + 1$ 
```

(v) (3 points)

```
1:  $a = 0$ 
2: for  $i$  from 1 to  $n$  :
3:     for  $j$  from  $i$  to  $n$  :
4:          $a = a + 1$ 
```

(vi) (3 points)

```
1:  $a = n$ 
2: while  $a > 1$  :
3:      $a = a/2$ 
```

E2.6 **On connected graphs and trees (10 points).** Assume G is a connected graph with n nodes. Show that the following statements are equivalent:

- (i) G is a tree, and
- (ii) G has exactly $n - 1$ edges.

Hint: Prove the (i) \Rightarrow (ii) direction by induction.

E2.7 **The BFS algorithm for disconnected graphs (15 points).**

Starting from the BFS algorithm, provide an algorithm in pseudocode that efficiently performs the following functions:

- (i) verifies whether the graph is connected,
- (ii) calculates how many connected components it has, and
- (iii) computes the number of nodes in each connected component.

E2.8 **Programming: BFS algorithm (30 points).**

Consider the following functions:

`computeBFStree` (15 points)

Input: a graph described by its adjacency table `AdjTable` and a start node `start`

Output: a vector of pointers `parents` describing the BFS tree rooted at `start`

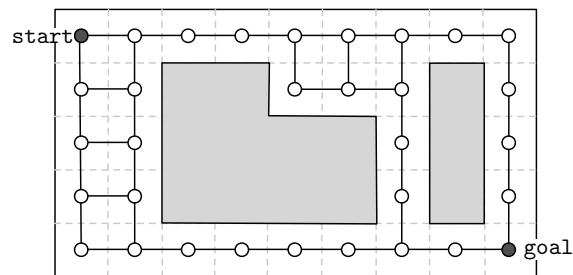
`computeBFSpath` (15 points)

Input: a graph described by its adjacency table `AdjTable`, a start node `start` and a goal node `goal`

Output: a path from `start` to `goal` along the BFS tree rooted at `start`

For each function, do the following:

- (i) explain how to implement the function, possibly deriving analytic formulas, and characterize special cases,
- (ii) program the function, including correctness checks on the input data and appropriate error messages, and
- (iii) verify your function is correct on a broad range of test inputs and, specifically, on the graph and the start/goal problem drawn in figure.



Hints: The function `computeBFSPATH` should invoke the function `computeBFSTree`.

E2.9 **Programming Project: The sweeping trapezoidation algorithm (60 points).**

In this project you will write a function that, given as input a rectangular workspace with multiple polygonal obstacles (each described by a counter-clockwise sequence of vertices), computes the trapezoidal decomposition of the workspace and the associated roadmap graph. For simplicity, assume that the polygonal obstacles do not overlap and are strictly inside the rectangular workspace.

- (i) Write in detailed pseudocode a function that classifies each obstacle vertex in the 6 possible types.
- (ii) Implement the main function:

SweepingTrapezoidation

Input: a rectangle W with polygonal holes P , where P is a list of non-overlapping polygons inside W .

Output: a collection of trapezoids (including the degenerate case of triangles), and a collection of edges (each describing a shared side between two trapezoids).

Configuration Spaces

The previous chapters focused on planning paths for a robot modelled as a point in the plane. In this chapter we consider robots with physical size and robots composed of multiple moving bodies. In particular, we

- (i) describe a robot as a single or multiple interconnected rigid bodies,
- (ii) define the configuration space of a robot,
- (iii) examine numerous example configuration spaces, and
- (iv) discuss forward and inverse kinematic maps that arise in robot motion planning.

This chapter is inspired by [de Berg et al. \(2000, Chapter 13\)](#), [LaValle \(2006, Chapter 4\)](#), and [Hager \(2006\)](#).

3.1 Problem setup: Multi-body robots in realistic workspaces

Recall that we have worked so far with

- a planar workspace $W \subset \mathbb{R}^2$,
- some obstacles O_1, \dots, O_n , and
- a point robot with no shape, size or orientation.

We then defined the free workspace $W_{\text{free}} = W \setminus (O_1 \cup \dots \cup O_n)$ as the set of locations where a point robot is not hitting any obstacle. A feasible motion plan for a point robot is computed as a path in W_{free} . In this chapter we begin to consider robots that are more realistic than the point robot by considering robots. As illustrated in Figure 3.1, path planning algorithms for point robots are not immediately applicable to robots with a shape.

We start with some simple concepts:

- (i) a *rigid body* is a collection of particles whose position relative to one another is fixed,

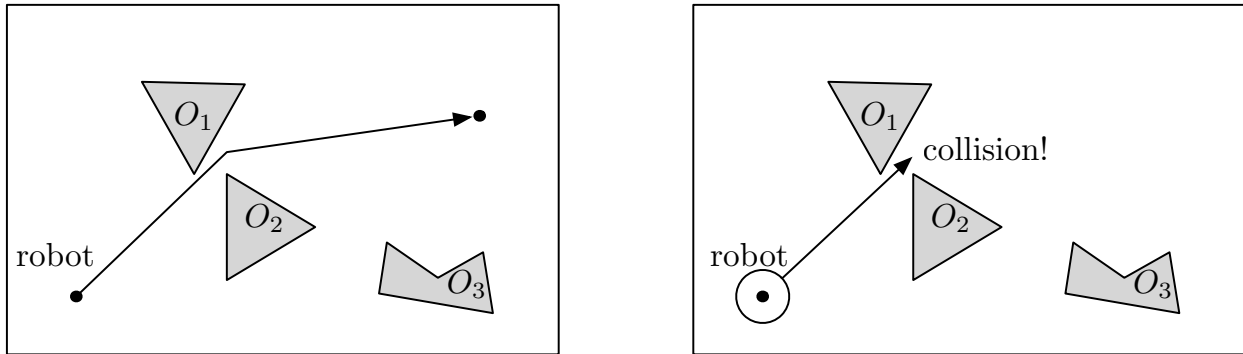
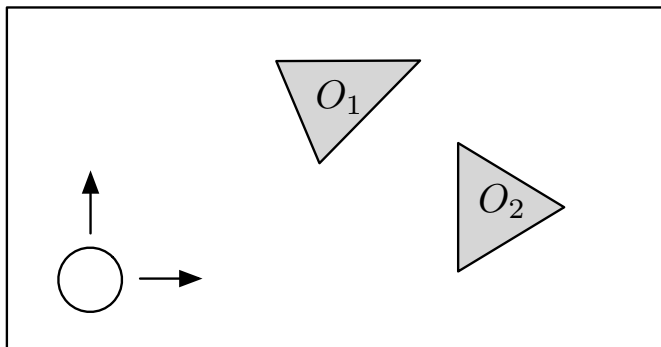


Figure 3.1: A path planned for a point robot (in left figure) will not work for a robot with the shape of a disk.

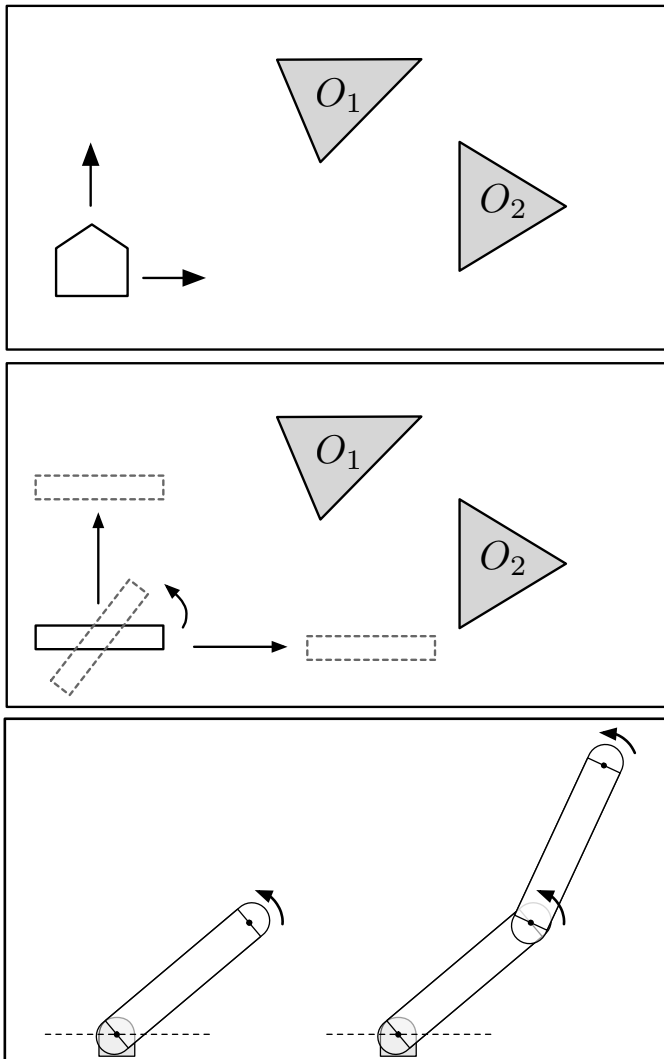
- (ii) a robot is composed of a single rigid body or multiple interconnected rigid bodies,
- (iii) robots are 3-dimensional in nature, but this chapter focuses on planar problems; we postpone 3-dimensional rigid bodies to the later chapters on kinematics and rotation matrices,
- (iv) it is equivalent to specify
 - a) the position of every point belonging to a rigid body, or
 - b) the position of a specific point and the orientation of the rigid body, plus a representation of the shape of the rigid body.

Note: we do not consider flexible or deformable shapes, as in general an infinite number of variables is required to represent such robots.

Example systems Let us discuss a few preliminary examples of more complex robots. We consider only planar systems composed of either a single rigid body or multiple interconnected rigid bodies.



The *disk robot* has the shape of a disk and is characterized by just one parameter, its radius $r > 0$. The disk robot does not have an orientation. Accordingly, the disk robots only motion is translation in the plane: that is, translations in the horizontal and vertical directions.



The *translating polygon robot* has a polygonal shape, e.g., a ship-like shape in the side figure. This robot is assumed to have a fixed orientation, and thus its only motion is translations in the plane.

The *roto-translating polygon robot*, with an arbitrary polygonal shape, is capable of translating in the horizontal and vertical directions as well as rotating.

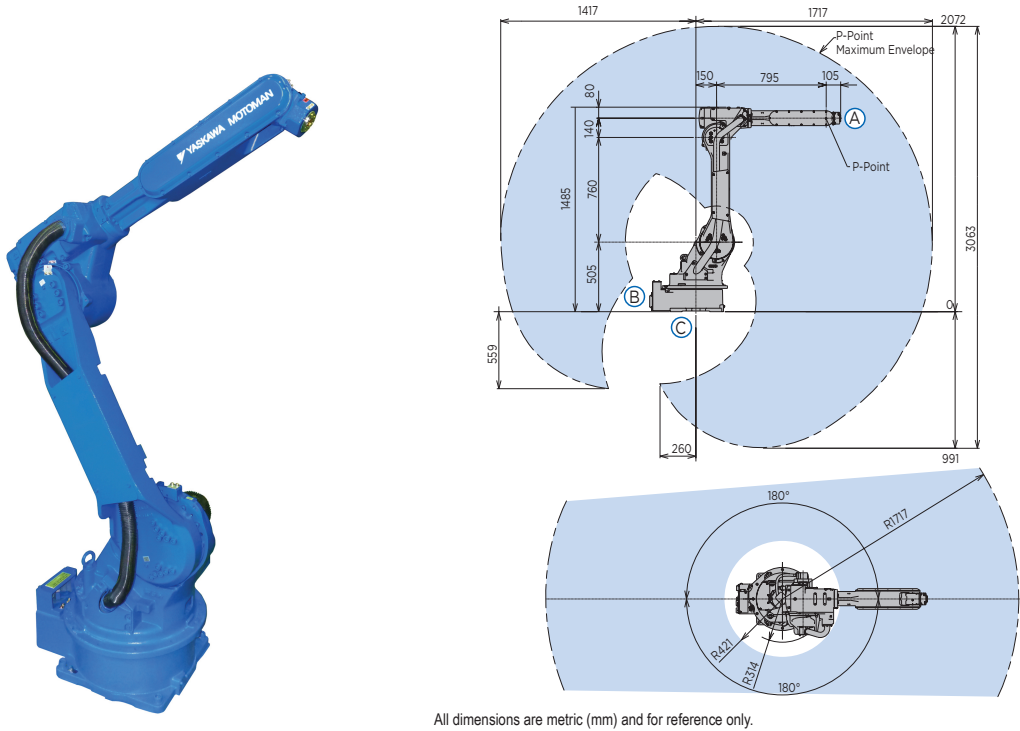
A *multi-link or multi-body robot* is composed of multiple rigid bodies (or links) interconnected. Each link of the robot can rotate and translate in the plane, but these motions are constrained by connections to the other links and to the robot base. The *1-link robot* is described by just a single angle. The *2-link robot* is described by two angles.

Multi-body robots are commonly used in industrial manipulation tasks. An example of an industrial manipulator is shown in Figure 3.2. Robots with higher numbers of degrees of freedom are also of interest, e.g., see Figure 3.3.

3.2 The configuration space

In the rest of this chapter we study how to represent the position of robots composed of rigid bodies assuming no obstacles are present. We postpone the study of obstacles to the next chapter.

- (i) A *configuration* of a robot is a minimal set of variables that specifies the position and orientation of each rigid body composing the robot. The robot configuration is usually denoted by the letter q .



All dimensions are metric (mm) and for reference only.

Figure 3.2: The Motoman® HP20 manipulator is a multi-body robot versatile high-speed industrial robot with a slim base, waist, and arm. Image courtesy of Yaskawa Motoman, <http://www.motoman.com>.

- (ii) The *configuration space* is the set of all possible configurations of a robot. The robot configuration space is usually denoted by the letter Q , so that $q \in Q$.
- (iii) The number of *degrees of freedom* of a robot is the dimension of the configuration space, i.e., the minimum number of variables required to fully specify the position and orientation of each rigid body belonging to the robot.
- (iv) Given that the robot is at configuration q , we know where all points of the robot are. In other words, there is a function $\mathcal{B}(q)$ as shown in Figure 3.4 that specifies the position of each point belonging to the robot at configuration q . The function \mathcal{B} is called the *configuration map*, and it maps each point q configuration space Q to the set of all points $\mathcal{B}(q)$ of the workspace belonging to the robot. Note, one way to represent the position of every point of the robot at a configuration q is to specify the position, orientation, and shape of each rigid body belonging to the robot.

Note: The configuration space should not be confused with the workspace. As shown in Figure 3.4, the workspace is always the 2-dimensional Euclidean space \mathbb{R}^2 or the 3-dimensional space \mathbb{R}^3 where the robot moves. The configuration space instead is a space of variables that describe the position and orientation of each rigid body component of a robot.



Figure 3.3: RoboSimian is an ape-like robot that moves around on four limbs. It was designed and built at NASA's Jet Propulsion Laboratory in Pasadena, California. It will compete in the 2015 DARPA Robotics Challenge Finals. Image courtesy of NASA/JPL-Caltech.

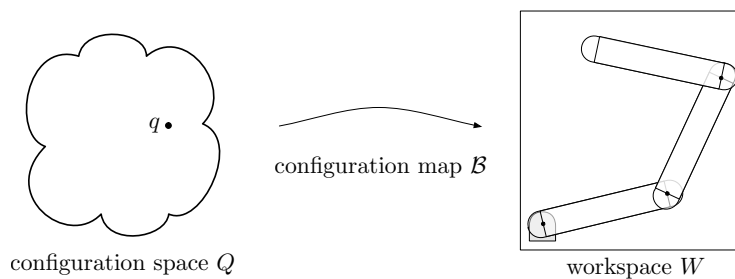


Figure 3.4: Each configuration q determines the position and orientation of each rigid body composing the robot.

To make these notions more concrete, let us now examine the configuration spaces of the robot examples introduced above. For example, we will learn that:

- a robot composed by a single rigid body (moving in the plane) has 3 degrees of freedom (2 due to translation and 1 due to rotation), and
- a robot composed by multiple rigid bodies (moving in the plane) has 3 degrees of freedom for each rigid body minus the number of constraints imposed by the joints.

Several different types of joints are used to interconnect rigid bodies. Several different types of interconnection are shown in Figure 3.5. The most common joint types are *prismatic*, which allows one rigid body to translate relative to another (for example, an telescoping pole that can

extend and retract), and *revolute*, which allows one joint to rotate about a single axis relative to another (for example, the elbow joint of a human arm). The manipulator shown in Figure 8.3 consists of three rigid bodies interconnected by three revolute joints. This is commonly referred to as the RRR configuration (each revolute joint is denoted by an R).

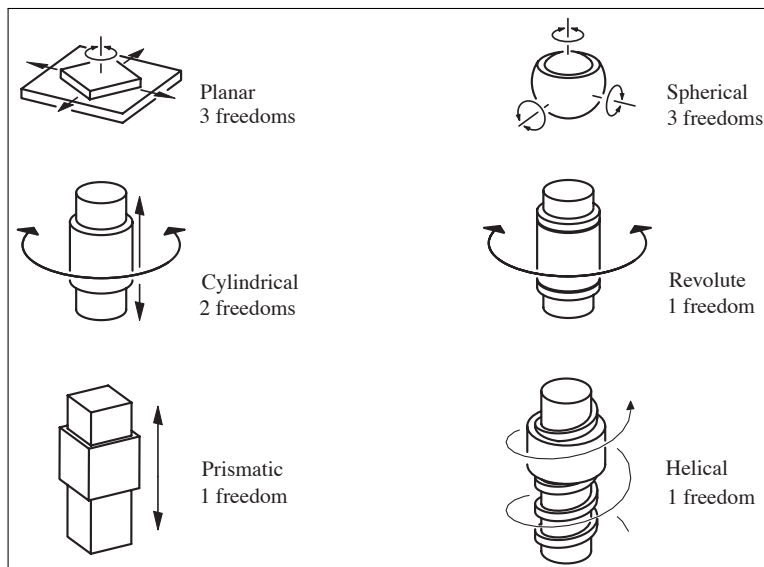


Figure 3.5: Example joints to interconnect rigid bodies. In classic kinematics, joints are called “kinematic pairs.” Each joint constrains the relative motion of the two rigid bodies being interconnected. This image is figure 2.21 in (Mason 2001) © 2001 Massachusetts Institute of Technology; it is used with permission of The MIT Press.

3.3 Example configuration spaces

We now present some examples of configuration spaces for simple robots presented above.

3.3.1 Configuration space of translating planar robots

In this first example we consider the disk robot and the polygonal robot that may only translate and not rotate. (For these two robots, we silently assume the existence of a constraint prohibiting the robot from rotating.) As shown in Figure 3.6, we designate a specific point of the robot to be the reference point (for example, the center of the body).

Given a reference frame in space (we will discuss the concept of reference frame further beginning in Chapter 6), place the robot such that its reference point lies at the origin of the reference frame. Let $\mathcal{B}(0, 0)$ denote the set of points belonging to the robot when the robot is at the reference position $(0, 0)$. Any other placement $\mathcal{B}(q)$ of the robot is specified by two parameters, i.e., by a point $q = (q_x, q_y)$ in the plane \mathbb{R}^2 .

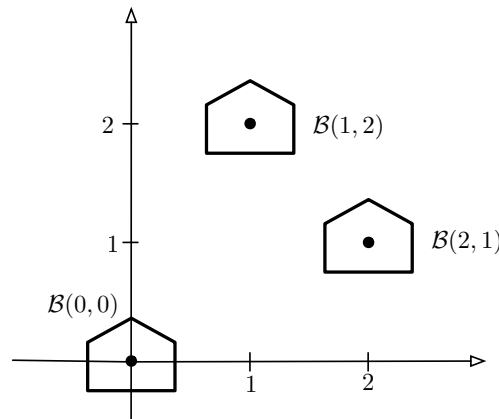


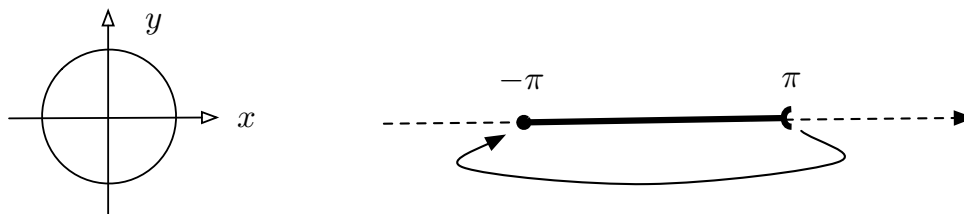
Figure 3.6: The polygonal robot at distinct positions in the workspace

Therefore, the configuration space of the robot is $Q = \mathbb{R}^2$ and a configuration $q = (q_x, q_y) \in Q$ is simply a point in the plane \mathbb{R}^2 . The robot has 2 degrees of freedom since two parameters are required to specify its configuration.

3.3.2 Configuration space of the 1-link robot

The configuration q of the 1-link robot is given by the angle of the link relative to a reference axis (for example, the horizontal axis). The configuration space Q is then the set of angles.

There are two ways we can represent the set of angles. The first is simply as an interval $[-\pi, \pi[$, where the number $-\pi$ is included in the interval and the number π is excluded. In this representation, one must remember that $-\pi$ and π are the same angle as shown in Figure 3.7, which can cause problems when calculating distances between angles.

Figure 3.7: The circle \mathbb{S}^1 and its representation as an interval $[-\pi, \pi[$, with $-\pi$ and π being the same angle. A circle is very different from a segment, when it comes to path planning.

The second way, as shown in Figure 3.7, is to represent the set of angles as the set of points on the unit-radius circle. We define the unit circle as $\mathbb{S}^1 = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}$, where the symbol \mathbb{S}^1 is used because the unit-radius circle is also the unit-radius one-dimensional sphere. The circle naturally captures the fact that $-\pi$ and π are the same angle.

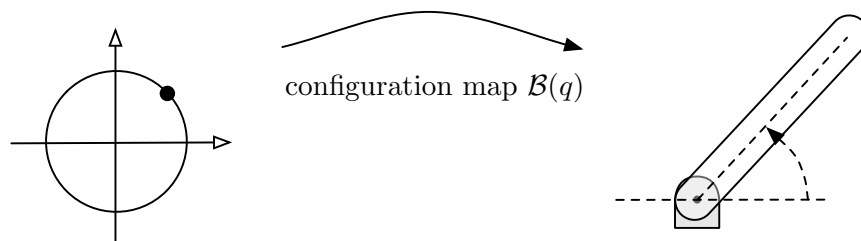


Figure 3.8: The configuration map for the 1-link robot. The point (approximately 45° measured counterclockwise from positive horizontal axis) on the configuration space (left image) determines where the 1-link robot is in the workspace (right image).

In summary, the configuration space of the 1-link robot is the circle $Q = \mathbb{S}^1$, and a configuration q can be thought of either as a point (x, y) on the circle, or as the corresponding angle θ defined by $x = \cos(\theta)$ and $y = \sin(\theta)$; see Figure 3.8. We will usually write the configuration simply as an angle $\theta \in \mathbb{S}^1$. The 1-link robot has 1 degree of freedom since its configuration can be described by a single angle.

3.3.3 Configuration space of roto-translating planar robots

The configuration of a planar object that translates and rotates is

$$q = (x, y, \theta),$$

where (x, y) is the position of the reference point and θ is the angle of the body measured counterclockwise with respect to the positive horizontal axis. The configuration space of a roto-translating polygon is $Q = \mathbb{R}^2 \times \mathbb{S}^1$. The robot has 3 degrees of freedom. Figure 3.9 illustrates the configuration map. (In the later chapters, we will study this configuration space as a matrix group of planar displacements.)

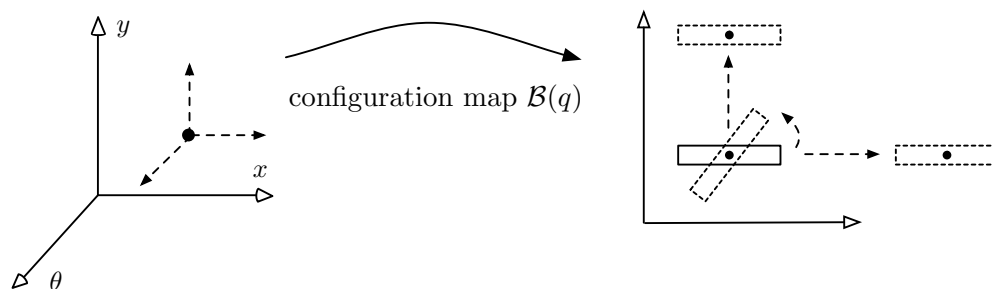


Figure 3.9: The configuration map for a roto-translating planar robot. The point and the arrows on the configuration space (left image) describes where the rigid body is and where it is moving towards in the workspace (right image).

3.3.4 Configuration space of robots moving in 3-dimensions

Robots moving in three dimensions may rotate or roto-translate in 3-dimensional Euclidean space. On the left of Figure 3.10 we illustrate a three-dimensional robot composed of a single rigid body. We postpone a detailed treatment of the configuration space for such robots to the later chapters on kinematics and rotation matrices. For now, let us just mention that a unconstrained single-rigid-body robot has 6 degrees of freedom: 3 degrees of translational freedom and 3 degrees of rotational freedom.

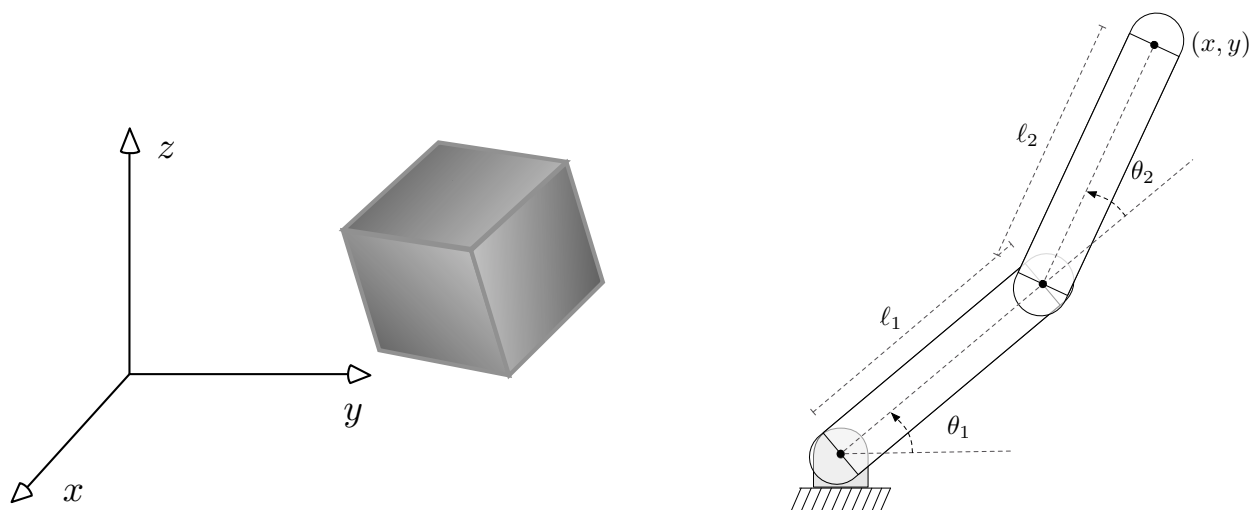


Figure 3.10: A three-dimensional rigid body on the left and a 2-link robot on the right

3.3.5 Configuration space of a multi-link robot

The configuration space of the 2-link robot brings up some interesting issues. As in Figure 3.10, let ℓ_1 and ℓ_2 be the lengths of the first and second link. Let θ_1 denote the angle of the first link measured counterclockwise with respect to the positive horizontal axis, and let θ_2 denote the angle of the second link measured counterclockwise with respect to the first link.

Therefore, the configuration q of the 2-link robot is described by the two angles θ_1 and θ_2 . The configuration space is then $Q = \mathbb{S}^1 \times \mathbb{S}^1$. We will write $(\theta_1, \theta_2) \in \mathbb{S}^1 \times \mathbb{S}^1$ or, slightly less precisely, $(\theta_1, \theta_2) \in [-\pi, \pi[\times [-\pi, \pi[$.

It is useful to define the *2-torus* (or simply *torus*) as the product of two circles: $\mathbb{T}^2 = \mathbb{S}^1 \times \mathbb{S}^1$. The torus can be depicted in two symbolic ways, see Figure 3.11. The left figure illustrates how the torus can be drawn as a square in the plane, but where (1) the vertical on the left is identified with the vertical segment on the right and (2) the horizontal segment on the top is identified with the horizontal segment on the bottom. The right figure is the 2-torus drawn in three dimensions: the shape is often referred to as the doughnut.

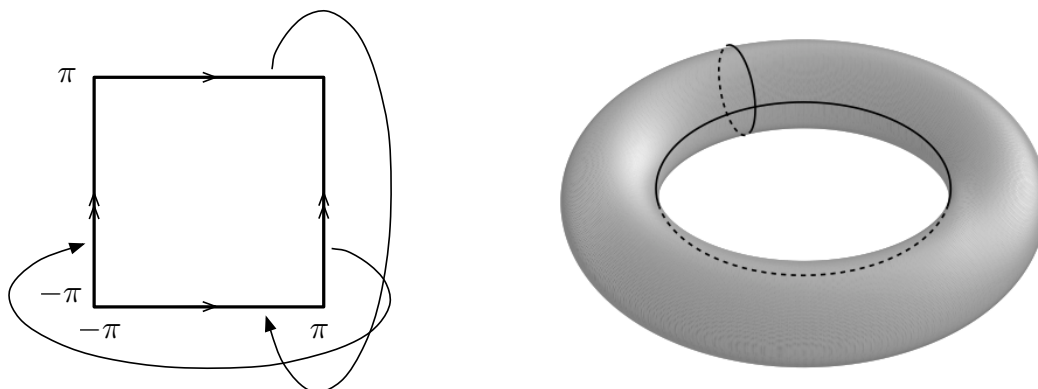


Figure 3.11: The 2-torus \mathbb{T}^2 , sometimes called just the torus. (In the left figure, the arrows in the left and right vertical segments point in the same upward direction meaning that the two segments are to be matched without any twist. For more on this concept, read [Wikipedia:Klein bottle](#).)

Note: The doughnut shape is obtained by (1) preparing a square flat sheet, and (2) gluing together vertical left with vertical right and horizontal top with horizontal bottom segments.

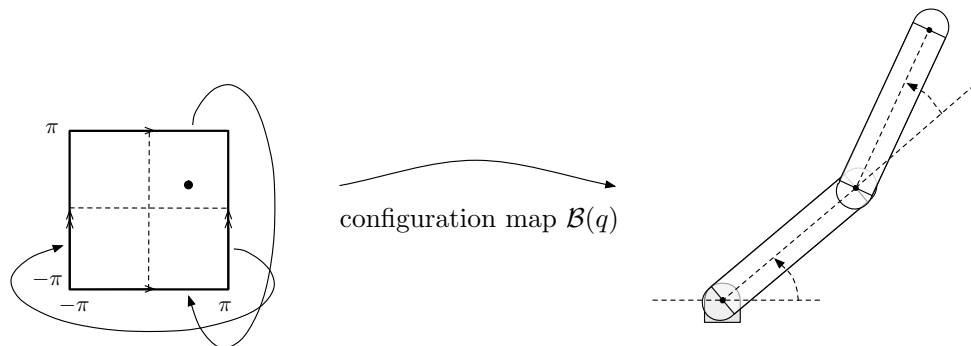


Figure 3.12: The configuration map for the 2-link robot. The point (approximately $(40^\circ, 30^\circ)$) on the configuration space (left image) determines the orientation of both robot links in the workspace (right image).

Note: The 2-sphere (in 3-dimensions) is the set of points in \mathbb{R}^3 at unit distance from the origin. In a formula, $\mathbb{S}^2 = \{(x, y, z) \in \mathbb{R}^3 \mid x^2 + y^2 + z^2 = 1\}$. The 2-torus and the 2-sphere are two sets that can be described by two angles. However, the 2-torus is substantially different from the 2-sphere. In \mathbb{S}^2 each closed path can be deformed continuously to a point because \mathbb{S}^2 has no holes. The 2-torus instead has a hole and it contains closed paths that cannot be continuously deformed to a point. The 2-sphere is usually drawn quite differently from how the 2-torus is drawn, e.g., see Figure 3.13.

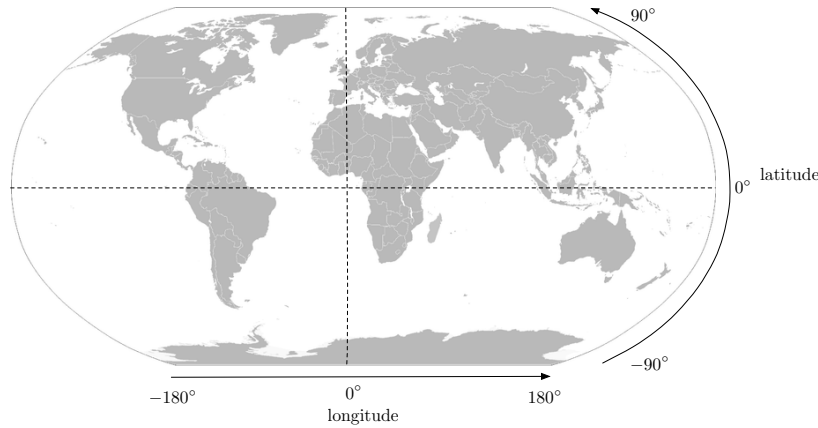


Figure 3.13: A world map as a 2-sphere Image courtesy of Wikipedia.

3.4 Forward and inverse kinematic maps

In this section we discuss how to transform motion planning problems from the workspace W to the configuration space Q via forward and inverse kinematics maps.

Given a motion planning problem in the workspace “move from point $p_{\text{start}} \in W$ to point $p_{\text{goal}} \in W$,” we need to translate this specification into the configuration space, i.e., move from a configuration $q_{\text{start}} \in Q$ to a configuration $q_{\text{goal}} \in Q$.

Rather than presenting a general framework, we discuss a specific example, the 2-link robot, and a specific motivation. Figure 3.14 is the picture of a robotic manipulator commonly used in pick-up and place and assembly applications. Manipulators with this configuration are referred to as “Selective Compliance Assembly Robot Arm,” or SCARA in brief. From [Wikipedia:SCARA](#), “by virtue of the SCARA’s parallel-axis joint layout, the arm is slightly compliant in the X-Y direction but rigid in the Z direction, hence the term: Selective Compliant.” In a vertical view, this manipulator is equivalent to the 2-link robot we just studied. The end position of this manipulator is the position (x, y) (near the far end of the 2-link) is where the *end-effector* is attached: the end-effector is the device with which the manipulator interacts with the environment, e.g., a robot gripper (for pick-up and place applications), a welding head, or a spray painting gun.

In short, we next consider the two following problems:

The forward kinematics problem: compute (x, y) as a function of (θ_1, θ_2) , and

The inverse kinematics problem: compute (θ_1, θ_2) as a function of (x, y) .

Note in the forward kinematics problem we are given the joint angles and our goal is to find the position of the end effector. In contrast, in the inverse kinematics problem we are given a desired end effector position and our goal is to find joint angles that place the end effector at the desired position.



Figure 3.14: The Yamaha© YK500XG is a high-speed SCARA robot with two revolute joints and a vertical prismatic joint. Image courtesy of Yamaha Motor Co., Ltd, <http://global.yamaha-motor.com/business/robot>.

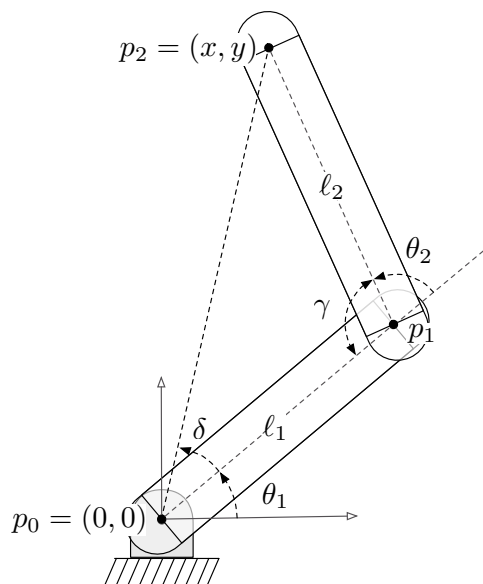


Figure 3.15: Vertical view of a SCARA robot, with end-effector location (x, y) . In the triangle (p_0, p_1, p_2) , define $\gamma \in [0, \pi]$ as the angle opposite the side $\overline{p_0 p_2}$.

3.4.1 Forward and inverse kinematics for the 2-link robot

Consider the 2-link robot with configuration variables (θ_1, θ_2) and end-effector location (x, y) as in Figure 3.15. Because the two joint angles (θ_1, θ_2) are the configuration variables, it should be possible to express (x, y) as a function of (θ_1, θ_2) . Some basic trigonometry leads to

$$\begin{aligned} x &= \ell_1 \cos(\theta_1) + \ell_2 \cos(\theta_1 + \theta_2), \\ y &= \ell_1 \sin(\theta_1) + \ell_2 \sin(\theta_1 + \theta_2). \end{aligned} \quad (3.1)$$

A function of the configuration variables that describes the position of a specific point of the robot body is usually called a *forward kinematics map*. So, equation (3.1) is the forward kinematics map for the 2-link robot.

Next, we are interested in computing what are possible configurations (θ_1, θ_2) when we know the end-effector is at position (x, y) , in other words, we are interested in the inverse function. We start by noting that such a problem does not admit a unique solution, as illustrated in the following Figure 3.16.

We compute the inverse kinematic map as follows. In the triangle defined by (p_0, p_1, p_2) , define $\gamma \in [0, \pi]$ as the angle opposite the side $\overline{p_0 p_2}$, see Figure 3.16. This triangle has sides of length $a = \ell_1$, $b = \ell_2$ and $c = \sqrt{x^2 + y^2}$. For this triangle, the [law of cosines](#) (see [Wikipedia](#)),

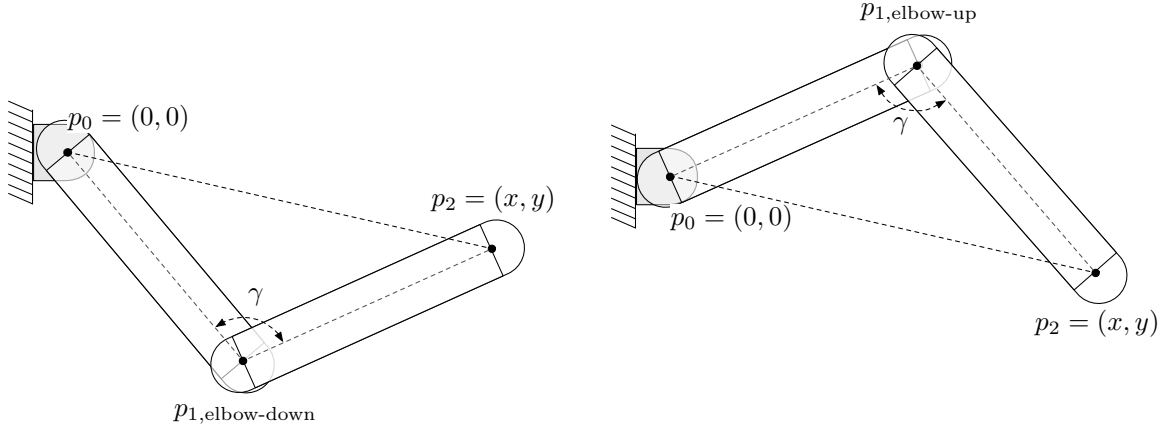


Figure 3.16: Two different solutions to the same inverse kinematics problem (the end-effector locations in the figures are identical): the elbow-down configuration corresponds to $\theta_2 = \pi - \gamma \in [0, \pi]$ and the elbow-up configurations corresponds to $\theta_2 = -\pi + \gamma \in [-\pi, 0]$.

$c^2 = a^2 + b^2 - 2ab \cos \gamma$, leads to

$$x^2 + y^2 = \ell_1^2 + \ell_2^2 - 2\ell_1\ell_2 \cos(\gamma),$$

and, in turn, to

$$\cos(\gamma) = \frac{\ell_1^2 + \ell_2^2 - x^2 - y^2}{2\ell_1\ell_2}. \quad (3.2)$$

According to the discussion in Appendix 3.5, there are either zero, one or two solutions to this equality. The right hand side is in the range $[-1, +1]$ if and only if

$$\begin{aligned} -1 \leq \frac{x^2 + y^2 - \ell_1^2 - \ell_2^2}{2\ell_1\ell_2} \leq 1 &\iff -2\ell_1\ell_2 \leq x^2 + y^2 - \ell_1^2 - \ell_2^2 \leq 2\ell_1\ell_2 \\ &\iff (\ell_1 - \ell_2)^2 \leq x^2 + y^2 \leq (\ell_1 + \ell_2)^2 \\ &\iff |\ell_1 - \ell_2| \leq \sqrt{x^2 + y^2} \leq (\ell_1 + \ell_2). \end{aligned}$$

So, for end-effector positions (x, y) such that $\sqrt{x^2 + y^2}$ is in the range $[|\ell_1 - \ell_2|, \ell_1 + \ell_2]$, there always exist exactly one solution γ in the range $[0, \pi]$ to equation (3.2) equal to

$$\gamma = \arccos\left(\frac{\ell_1^2 + \ell_2^2 - x^2 - y^2}{2\ell_1\ell_2}\right).$$

Finally, it remains to compute θ_2 as a function of γ . From Figure 3.16, it is clear that $\theta_2 = \pi - \gamma$ for the configuration with “robot elbow down” and $\theta_2 = -\pi + \gamma$ for the configuration with “robot elbow up.” We summarize this discussion in the following proposition.

Proposition 3.1 (Inverse kinematics for 2-link robot). *Consider the 2-link robot with configuration variables (θ_1, θ_2) and links lengths (ℓ_1, ℓ_2) as in Figure 3.15. Given a desired end-effector position*

(x, y) such that $|\ell_1 - \ell_2| \leq \sqrt{x^2 + y^2} \leq (\ell_1 + \ell_2)$, there exist two (possibly coincident) solutions for the joint angle θ_2 given by

$$\begin{aligned}\theta_{2,\text{elbow-down}} &= \pi - \arccos\left(\frac{\ell_1^2 + \ell_2^2 - x^2 - y^2}{2\ell_1\ell_2}\right) \in [0, \pi], \\ \theta_{2,\text{elbow-up}} &= -\pi + \arccos\left(\frac{\ell_1^2 + \ell_2^2 - x^2 - y^2}{2\ell_1\ell_2}\right) \in [-\pi, 0].\end{aligned}$$

The two solutions correspond to the diagrams in Figure 3.16.

We leave to the reader in Exercise E3.7 the following tasks: (i) the interpretation of the condition $|\ell_1 - \ell_2| \leq \sqrt{x^2 + y^2} \leq (\ell_1 + \ell_2)$, and (ii) the computation of the other joint angle θ_1 .

3.5 Appendix: Inverse trigonometric problems

We review here some basic identities from trigonometry that are useful in a inverse kinematics problems.

Solutions to basic trigonometric equalities

Given three numbers a , b , and c , we consider the following equations in the variables α , β and γ :

$$\sin(\alpha) = a, \quad \cos(\beta) = b, \quad \text{and} \quad \tan(\gamma) = c.$$

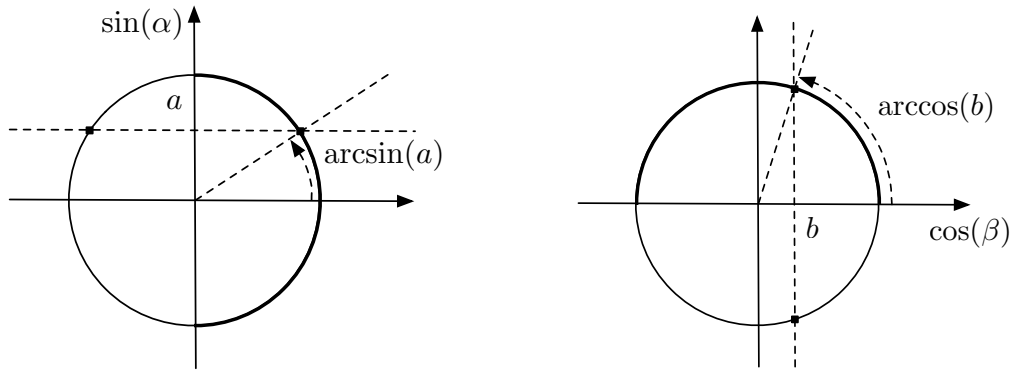


Figure 3.17: Illustrating the equations $\sin(\alpha) = a$ and $\cos(\beta) = b$

Because $\arcsin : [-1, 1] \rightarrow [-\pi/2, \pi/2]$ (that is, the arcsin function computes a single angle always in the interval $[-\pi/2, \pi/2]$), we solve the first equation $\sin(\alpha) = a$ as follows:

$$\begin{aligned}\text{if } |a| > 1, & \text{ no solution,} \\ \text{if } |a| \leq 1, & \text{ two (possibly coincident) solutions: } \alpha_1 = \arcsin(a), \quad \alpha_2 = \pi - \arcsin(a),\end{aligned}\tag{3.3}$$

so that, in particular, $\alpha_1 = \alpha_2 = \pi/2$ for $a = +1$, and $\alpha_1 = \alpha_2 = -\pi/2$ for $a = -1$.

Because $\arccos : [-1, 1] \rightarrow [0, \pi]$ (that is, the arccos function computes a single angle always in the interval $[0, \pi]$), we solve the second equation $\cos(\beta) = b$ as follows:

$$\begin{aligned} &\text{if } |b| > 1, \text{ no solution,} \\ &\text{if } |b| \leq 1, \text{ two (possibly coincident) solutions: } \beta_1 = \arccos(b), \quad \beta_2 = -\arccos(b), \end{aligned} \quad (3.4)$$

so that, in particular, $\beta_1 = \beta_2 = 0$ for $b = +1$, and $\beta_1 = \beta_2 = \pi$ for $b = -1$.

Finally, the third equation $\tan(\gamma) = c$ admits two solutions for any $c \in \mathbb{R}$:

$$\gamma_1 = \text{atan}(c) \quad \text{and} \quad \gamma_2 = \text{atan}(c) + \pi. \quad (3.5)$$

Four-quadrant arctangent function with two arguments

The four-quadrant arctangent function atan_2 computes the inverse tangent map as follows: for any point (x, y) in the plane except for the origin, the value $\text{atan}_2(y, x)$ is the angle between the horizontal positive axis and the point (x, y) measured counterclockwise; see Figure 3.18. So, for

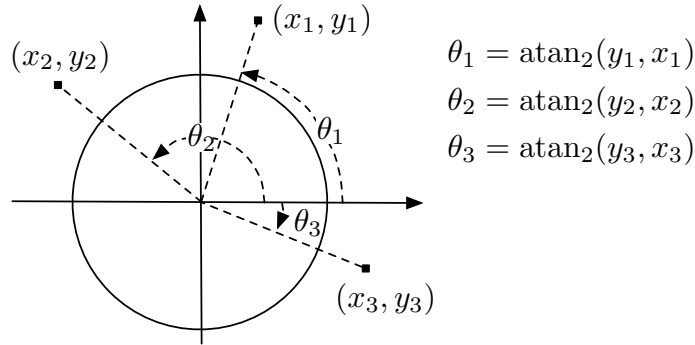


Figure 3.18: The four-quadrant arctangent function

example,

$$\text{atan}_2(0, 1) = 0, \quad \text{atan}_2(1, 0) = \pi/2, \quad \text{atan}_2(0, -1) = \pi, \quad \text{atan}_2(-1, 0) = -\pi/2.$$

Two useful property of the four quadrant arctangent function are:

$$\text{atan}_2(y, x) = \pi + \text{atan}_2(-y, -x), \quad \text{atan}_2(y, x) = -\frac{\pi}{2} + \text{atan}_2(x, -y).$$

Also, it is easy to see that, for all angles θ , we have

$$\theta = \text{atan}_2(\sin(\theta), \cos(\theta)).$$

The order of the arguments is very important as there are two possible conventions: (y, x) or (x, y) . In this class, we adopt the (y, x) convention.

Programming Note: In Matlab, the (y, x) convention is used. This can be seen by typing `help atan2`, which produces

ATAN2 Four quadrant inverse tangent.

ATAN2(Y,X) is the four quadrant arctangent of the real parts of the elements of X and Y. $-\pi \leq \text{ATAN2}(Y,X) \leq \pi$.

In Python, `atan2` is implemented in the `math` module as `math.atan2` and also uses the (y, x) convention. Some texts and the program Mathematica use the (x, y) convention.

Alternative solutions to basic trigonometric equalities

The four-quadrant arctangent function provides an alternative and sometimes easier way of computing solutions to

$$\sin(\alpha) = a, \quad \cos(\beta) = b, \quad \text{and} \quad \tan(\gamma) = c,$$

where we assume $|a| \leq 1$ and $|b| \leq 1$. Specifically, if $\sin(\alpha) = a$, then $\cos(\alpha) = \pm\sqrt{1-a^2}$ and therefore

$$\alpha_1 = \text{atan}_2(a, \sqrt{1-a^2}), \quad \text{and} \quad \alpha_2 = \text{atan}_2(a, -\sqrt{1-a^2}). \quad (3.6)$$

Similarly, if $\cos(\beta) = b$, then $\sin(\beta) = \pm\sqrt{1-b^2}$ and therefore

$$\beta_1 = \text{atan}_2(\sqrt{1-b^2}, b), \quad \text{and} \quad \beta_2 = \text{atan}_2(-\sqrt{1-b^2}, b). \quad (3.7)$$

Finally, $\tan(\gamma) = c$ is equivalent to

$$\gamma_1 = \text{atan}_2(c, 1), \quad \text{and} \quad \gamma_2 = \text{atan}_2(-c, -1).$$

3.6 Exercises

E3.1 Shortest paths and distances on the circle (20 points).

Given two points on the circle θ_1 and θ_2 (represented as values in the interval $[-\pi, \pi]$), the *counter-clockwise distance from θ_1 to θ_2* , denoted by $\text{dist}_{\text{cc}}(\theta_1, \theta_2)$, is the length of the counter-clockwise arc starting at θ_1 and ending at θ_2 . Similarly, the *clockwise distance*, denoted by $\text{dist}_{\text{c}}(\theta_1, \theta_2)$, is the length of the clockwise arc from θ_1 to θ_2 . Finally, the *distance between θ_1 to θ_2* is the smallest of the two counter-clockwise and clockwise distances.

Hint: On the unit circle, the length of an arc is equal to the angle subtended by the arc and is measured in radians.

- (i) (2 points) Compute counter-clockwise and the clockwise distances between $\theta_1 = \pi/3$ and $\theta_2 = -3\pi/4$.
- (ii) (5 points) Provide a formula for the counter-clockwise distance from arbitrary θ_1 to arbitrary θ_2 .

Hint: Recall the “modulo” operator: $\text{mod}(\theta, 2\pi)$ is the remainder of the division of θ by 2π .

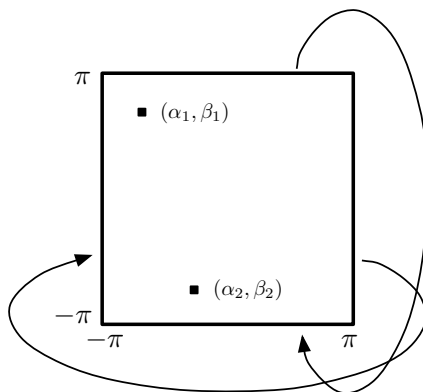
- (iii) (5 points) Provide a formula for the clockwise distance from θ_1 to θ_2 , denoted by $\text{dist}_{\text{c}}(\theta_1, \theta_2)$.
- (iv) (3 points) Define the distance between two angles to be the smallest between their counterclockwise and clockwise distance. Provide a formula for the distance between θ_1 and θ_2 , denoted by $\text{dist}_{\text{circle}}(\theta_1, \theta_2)$.

For further reference, note the following general properties:

| | |
|--|--|
| $\text{dist}_{\text{cc}}(\theta_1, \theta_2) \neq \text{dist}_{\text{cc}}(\theta_2, \theta_1),$ | lack of symmetry of $\text{dist}_{\text{cc}},$ |
| $\text{dist}_{\text{cc}}(\theta_1, \theta_2) = \text{dist}_{\text{c}}(\theta_2, \theta_1),$ | relationship between dist_{cc} and $\text{dist}_{\text{c}},$ and |
| $\text{dist}_{\text{circle}}(\theta_1, \theta_2) = \text{dist}_{\text{circle}}(\theta_2, \theta_1),$ | symmetry of $\text{dist}_{\text{circle}}.$ |

E3.2 Shortest paths and distances on the 2-torus (10 points).

Requires Exercise E3.1. Consider the two points (α_1, β_1) and (α_2, β_2) in \mathbb{T}^2 as depicted in the figure.



- (i) (5 points) Sketch the shortest path between (α_1, β_1) and (α_2, β_2) .
- (ii) (5 points) Define the distance between two points to be the length of the shortest path between them. Provide a formula for the distance between (α_1, β_1) and (α_2, β_2) , denoted by $\text{dist}_{2\text{-torus}}((\alpha_1, \beta_1), (\alpha_2, \beta_2))$.

Hints: Your answer should include a square root. Gain deeper intuition into trajectories on the 2-torus by practicing the classic arcade game Asteroids at <http://www.freeasteroids.org>.

E3.3 Shortest paths and distances on the 2-sphere (10 points).

The 2-sphere \mathbb{S}^2 is the set of points in \mathbb{R}^3 at unit distance from the origin.

- (i) What do shortest paths between points look like?
- (ii) Given two points $p = (p_1, p_2, p_3)$ and $q = (q_1, q_2, q_3)$ in the 2-sphere (so that $p_1^2 + p_2^2 + p_3^2 = 1 = q_1^2 + q_2^2 + q_3^2$), provide a formula for the distance between p and q , denoted by $\text{dist}_{2\text{-circle}}(p, q)$.

E3.4 Programming: Distances on the circle and the 2-torus (10 points).

Requires Exercises E3.1 and E3.2. Write the following programs:

`computeDistanceOnCircle` (5 points)

Input: two angles α and β in the interval $[-\pi, \pi[$

Output: the distance between α and β

`computeDistanceOnTorus` (5 points)

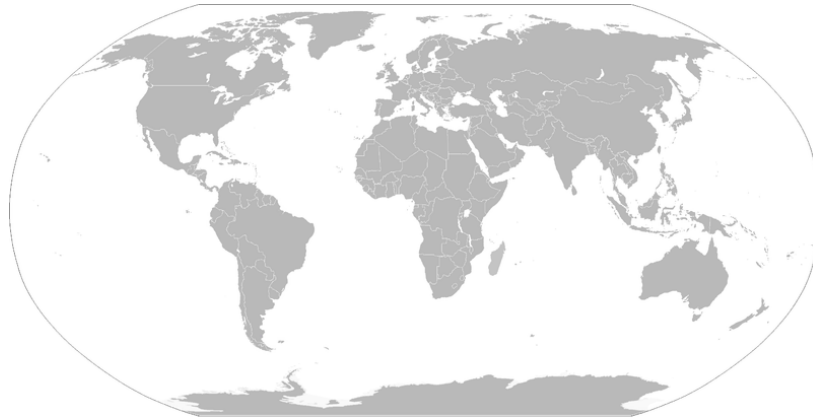
Input: two points on the torus angles (α_1, α_2) and (β_1, β_2)

Output: the distance between (α_1, α_2) and (β_1, β_2)

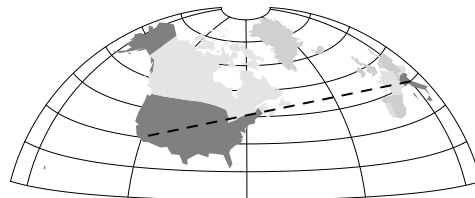
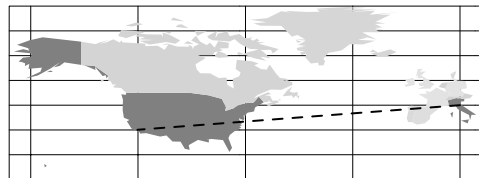
E3.5 Mapping mother Earth onto a flat surface (10 points).

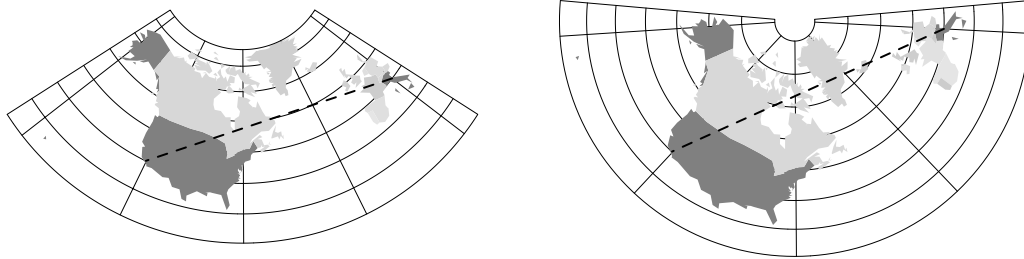
Cartography, the science and practice of drawing maps, has always arisen the curiosity of mankind and played a key role in navigation.

- (i) In the following world map, which points are drawn more than once? In other words, which points in the map correspond to the same physical location on mother Earth? Draw lines explaining the identifications, in the same style as we used for the 2-torus in Figure 3.11.

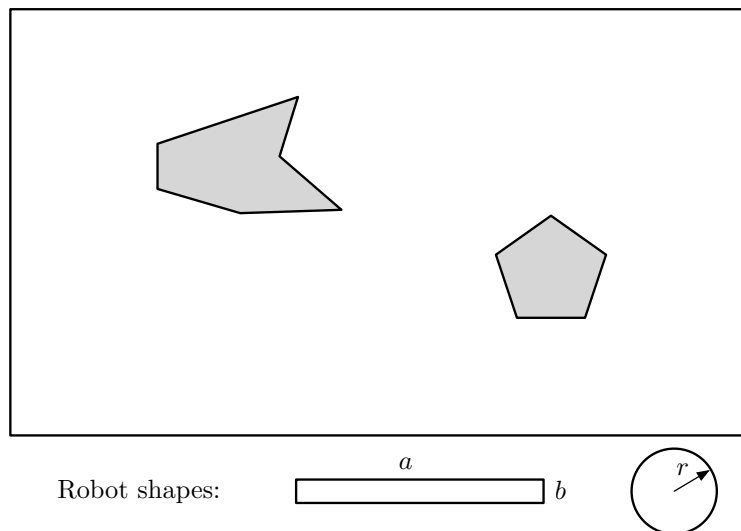


- (ii) The following four figure show different ways of projecting the spherical world onto a flat surface and of computing straight-line paths from Santa Barbara, California, to Venice, Italy. Briefly explain why the four straight segments are not identical and how you would compute the actual length of each of them.





E3.6 **Free configuration spaces (30 points).**

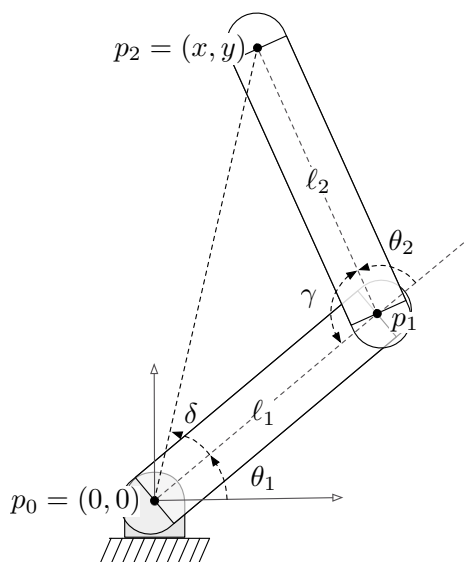


- (i) (10 points) Sketch the free configuration space for the disk robot (include the original obstacles and workspace).

Hint: Consider the scale of the robot and workspace, and check the possibility of movement.

- (ii) (10 points) Sketch the free configuration space for the rectangular robot assuming that the robot cannot rotate (include the original obstacles and workspace).
- (iii) (10 points) While the free configuration space for the rectangular robot still has polygonal obstacles, the same is not true for the disk robot. The triangular and trapezoidal partitioning strategies discussed in Chapter 2 require polygonal obstacles. In a few sentences and a clarifying sketch, propose a solution to partitioning the free configuration space for the disk robot to enable the construction of a roadmap.

E3.7 **Reachable workspace and inverse kinematics for 2-link manipulator (25 points).**



This question deals with the inverse kinematics for the 2-link manipulator with link lengths ℓ_1 and ℓ_2 . We are interested in the following general question: Given the (x, y) location of the end effector, what are the values for the two joint angles (θ_1, θ_2) ? Specifically, perform the following tasks.

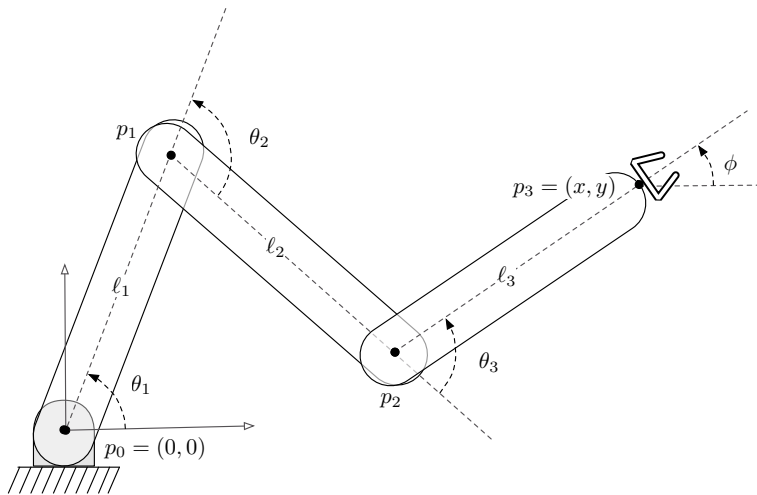
- (i) (5 points) Call the point (x, y) *reachable* if there exists joint angles (θ_1, θ_2) such that the end effector is at (x, y) . The set of reachable points is called the *reachable workspace*. Draw the reachable workspace for three cases: $\ell_1 < \ell_2$, $\ell_1 = \ell_2$ and $\ell_1 > \ell_2$. Are your pictures consistent with the condition given in Proposition 3.1 that $\sqrt{x^2 + y^2}$ is in the range $[|\ell_1 - \ell_2|, \ell_1 + \ell_2]$?
- (ii) (5 points) For the same three cases as in the previous questions, explain how many (θ_1, θ_2) solutions there are for each reachable (x, y) .

Hint: In general, there could be no solutions, one solution, two solutions, or for special cases an infinite number of possible solutions. Also, think carefully about the boundary of the reachable workspace.

- (iii) (15 points) Section 3.4 contains an expression for θ_2 . Using this expression and the results in Appendix 3.5 as a guide, derive an expression for θ_1 .

Hint: Make sure you use the correct order of arguments in your arctangent function

- E3.8 **Kinematic calculations for 3-link manipulator (25 points).** You are given a planar three-link manipulator with a gripper attached at the end of the third link (i.e., the gripper is the end-effector). Assume the three links have unit length (i.e., $\ell_1 = \ell_2 = \ell_3 = 1$).



Let θ_1 be the orientation of the first link, measured counter-clockwise from the horizontal axis. Let θ_2 and θ_3 be the relative orientations of second and third link, respectively, measured counterclockwise.

Let (x, y, ϕ) be the position and orientation of the gripper, where ϕ is measured counter-clockwise relative to the horizontal axis.

- (i) (8 points) Write (x, y, ϕ) as a function of $(\theta_1, \theta_2, \theta_3)$. (This is the forward kinematics map.)
- (ii) (8 points) Given (x, y, ϕ) , write a formula for the point (x_2, y_2) where the second and third link meet.
- (iii) (9 points) Write $(\theta_1, \theta_2, \theta_3)$ as a function of (x, y, ϕ) . (This is the inverse kinematics map.)

Hint: Make use of your answer for (ii).

Free Configuration Spaces via Sampling and Collision Detection

In this chapter we complete the discussion of robot configuration spaces. We discuss the role of obstacles in both workspace and configuration space, the notion of free configuration space, and corresponding computational methods. In particular, we

- (i) represent obstacles and the free space when the robot is composed of a single or multiple rigid bodies with proper shape, position and orientation,
- (ii) compute free configuration spaces via sampling and collision detection,
- (iii) discuss sampling methods, and
- (iv) discuss collision detection methods.

The concepts in this chapter are related to Section 5.3 "Collision Detection" in (LaValle 2006) and Chapter 13 in (de Berg et al. 2000).

4.1 The free configuration space

As in Chapter 3, the configuration of a robot is a minimal set of variables that describes the position of each rigid body component of the robot. Therefore, in the configuration space a robot position is just a single point.

In this section we discuss how to model obstacles in configuration space. The key problem is: what robot configurations correspond to *feasible positions* of the robot, i.e., configurations in Q such that the robot is not in collision with any obstacle in W .

Let us review the basic setup. We are given the workspace W with obstacles O_1, \dots, O_n . Therefore, the free workspace is

$$W_{\text{free}} = W \setminus (O_1 \cup \dots \cup O_n).$$

Second, we are given the robot configuration space Q and the configuration map $\mathcal{B}(q)$, which denotes the set of particles belonging to the robot in the workspace as a function of the robot configuration q .

- (i) The *free configuration space* Q_{free} is the set of configurations q such that all points of the robot are inside W_{free} . Because all points of the robot are given by $\mathcal{B}(q)$, we can write

$$Q_{\text{free}} = \{q \in Q \mid \mathcal{B}(q) \text{ is inside } W_{\text{free}}\}.$$

- (ii) Given an obstacle O in workspace, the corresponding *configuration space obstacle* O_Q is the set of configurations q such that the robot at configuration q is in collision with the obstacle O . In a formula, this is written as

$$O_Q = \{q \in Q \mid \mathcal{B}(q) \text{ overlaps with } O\}.$$

Combining these two notions, if $O_{Q,1}, \dots, O_{Q,n}$ are the configuration space representation of the obstacles O_1, \dots, O_n , the free configuration space can be written as

$$Q_{\text{free}} = \{q \in Q \mid \mathcal{B}(q) \subset W\} \setminus (O_{Q,1} \cup \dots \cup O_{Q,n}).$$

4.1.1 Free configuration space for the disk robot

Consider a planar robot with the shape of a disk of radius r and with the ability to translate only. In the absence of obstacles, the configuration space and the workspace are identical: the configuration of the robot is given by the pair (x, y) that takes value in \mathbb{R}^2 . We now imagine the disk robot is restricted to move inside a rectangle W and to avoid a rectangular obstacle O ; as in Figure 4.1. Easily, the free workspace is the rectangle W minus the obstacle O .

To compute the *free configuration space* and the *obstacles in configuration space* we reason as follows: a disk with radius r is in collision with an obstacle if and only if the disk center is closer to the obstacle than r . Accordingly, we grow or “expand” the obstacle and, correspondingly, to “shrink” the workspace in the following manner: Using this key idea, the *obstacle in configuration*

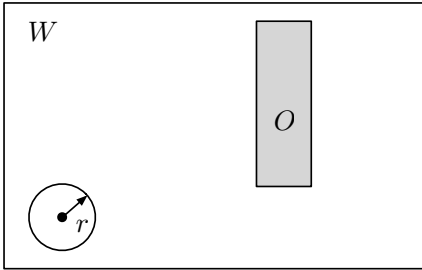


Figure 4.1: An example configuration space

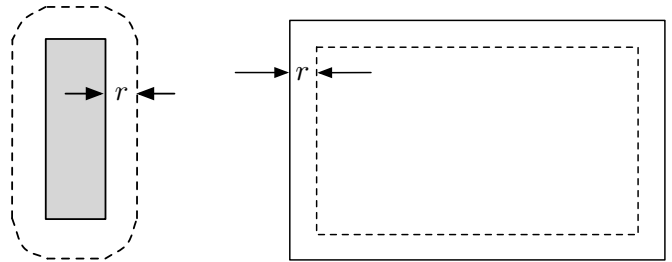


Figure 4.2: Expanding an obstacle and shrinking a workspace

space is the expanded rectangle and the *free configuration space* of the disk robot is described in any of the two completely equivalent forms:

- (i) the set of positions of the disk center such that the disk does not intersect the obstacle and is inside the workspace,
- (ii) the shrunk workspace minus the expanded obstacle.

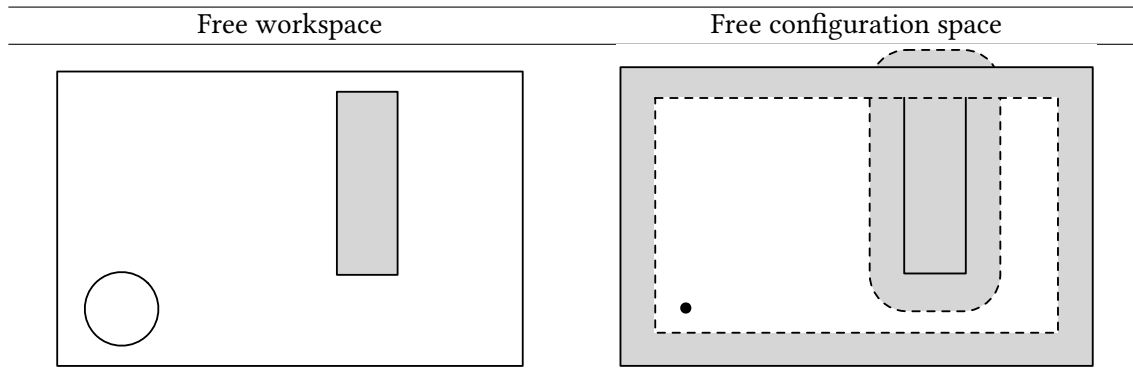


Figure 4.3: It is equivalent to plan feasible paths for the disk robot moving in the free workspace (left figure) or for a point robot moving in the free configuration space (right figure).

Note: The disk robot moving in a workspace with obstacles is now understood as a configuration point moving in the free configuration space. Therefore, motion planning for the disk robot can again be performed via decomposition and search *on the free configuration space*.

Note: one problem with the expansion of polygonal obstacles is that their expansion is no longer a polygon. At the cost of over-estimating the obstacle and therefore over-constraining the robot motion, one can always render the expansion a polygon by “over-expanding” the corners as shown in Figure 4.4.

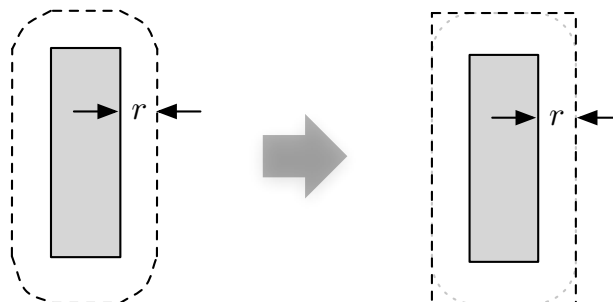


Figure 4.4: Approximating expanded obstacles by a larger polygonal obstacle

4.1.2 Free configuration space for the translating polygonal robot

Suppose our robot is polygonal and convex with shape as shown in Figure 4.5. The robot has a fixed orientation as shown in the figure, and moves by translating in the plane. Now, given any polygonal convex obstacle, how would you determine the configuration space obstacle for this robot? The “obstacle expansion” procedure is now more complicated than in the disk robot example, because now both robot and obstacle are polygonal.



Figure 4.5: A polygonal and convex robot in the plane

There is a simple graphical approach to computing the configuration space obstacle illustrated in Figure 4.6 and described as follows:

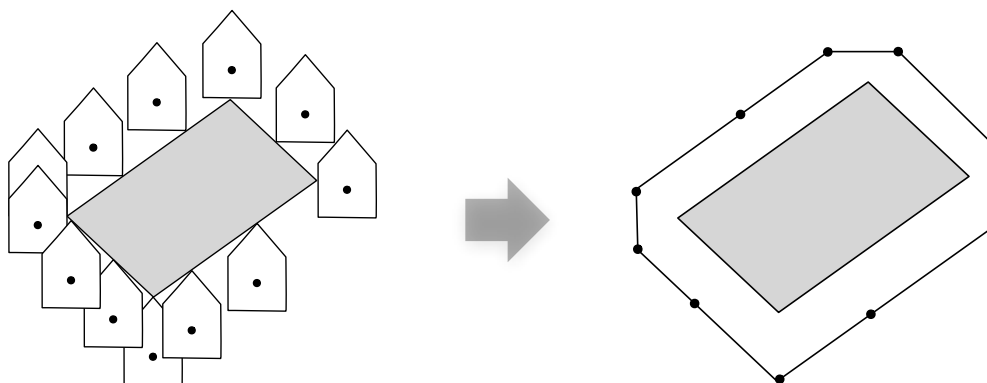


Figure 4.6: The configuration space obstacle for a translating robot

- (i) move the robot to touch the obstacle boundary (recall the robot is not allowed to rotate),
- (ii) slide the robot body along the obstacle boundary, maintaining the contact between the robot boundary and the obstacle boundary,
- (iii) while sliding, store the location of the reference point of the robot: the resulting path encloses a convex polygon equal to the configuration space obstacle.

To turn this graphical procedure into a programmable algorithm, we introduce a useful operation. Given two sets S_1 and S_2 in \mathbb{R}^2 , the *Minkowski difference* $S_1 \ominus S_2$ is defined by

$$S_1 \ominus S_2 = \{p - q \mid p \in S_1 \text{ and } q \in S_2\}.$$

Proposition 4.1. *Assume the robot body, with reference position $\mathcal{B}(0,0)$ and the obstacle O are convex polygons with n and m vertices respectively. Then the resulting configuration space obstacle O_Q is a convex polygon with at most $n + m$ vertices and satisfies*

$$O_Q = O \ominus \mathcal{B}(0,0). \quad (4.1)$$

To see that this formula is correct and to gain intuition into the Minkowski difference, let us reason as follows. Let w be one of the vertices of the body at reference position $\mathcal{B}(0, 0)$. When the body is at position q , the translated vertex is touching the vertex v of the obstacle precisely if $q + w = v$. In other words, q is a configuration of vertex-to-vertex contact precisely when $q = v - w$; we denote this configuration by $q_{v,w}$ and we illustrate the setting in the Figure 4.7. Therefore, body and obstacle are overlapping at all configurations q that are the sum of a point v in the obstacle *minus* a robot point w , as measured at the reference configuration $\mathcal{B}(0, 0)$.

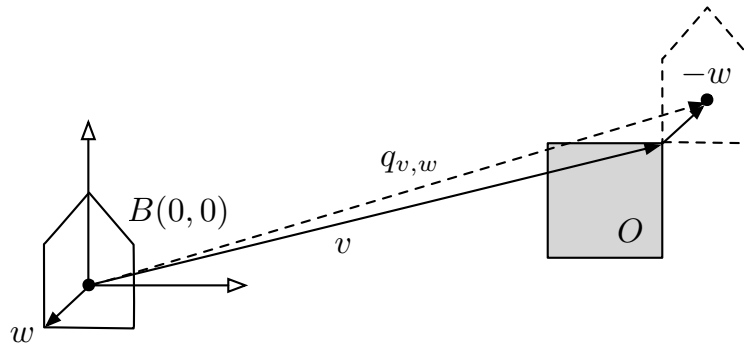


Figure 4.7: Computing configuration space obstacles via Minkowski differences

Given convex polygons O and $\mathcal{B}(0, 0)$, we now present a simple algorithm to compute the Minkowski difference in equation (4.1). First, we define the *convex hull* of a group of points as the minimum-perimeter convex set containing them. The convex hull of a group of points is a convex polygon. Graphically, the convex hull can be obtained by snapping a tight rubber band around all the points (the length of the rubber band is the perimeter of the envelope) as shown in Figure 4.8. Each convex polygon is the convex hull of its vertices.

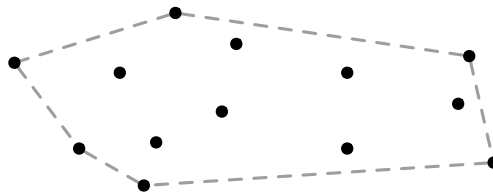


Figure 4.8: Example convex hull of a set of points

Programming Note: The Matlab command `convhull` computes the convex hull in 2 and 3 dimensional spaces. In Python the convex hull of a set of points can be computed using the `scipy` module. Import the module using `from scipy.spatial import ConvexHull`. The function operates on a numpy array of points. Numpy is imported via `import numpy`.

Finally, here is the promised algorithm for computing the configuration space obstacle using the Minkowski difference.

The Minkowski-difference-via-convex-hull algorithm

Input: two convex polygonal subsets P_1 and P_2 of \mathbb{R}^2

Output: the Minkowski difference $P_1 \ominus P_2$

- 1: **for** each vertex v of P_1 :
 - 2: **for** each vertex w of P_2 :
 - 3: compute the difference point $v - w$
 - 4: **return** the convex hull of all difference points
-

To characterize the runtime of this algorithm, suppose that P_1 has n vertices and P_2 has m vertices. The algorithm will compute nm difference points. There are many algorithms for computing the convex hull of a set of points N points, the best of which run in $O(N \log(N))$ time. Therefore, the runtime of the Minkowski-difference-via-convex-hull algorithm is in $O(nm \log(nm))$.

However, recall that by Proposition 4.1, the resulting convex polygon will have at most $n + m$ vertices. Based on this fact, there exists a much more efficient algorithm. The idea is to directly compute the (at most) $n + m$ points, rather than computing nm and then throwing all but $n + m$ away via the convex hull computation. This more efficient algorithm can be implemented to run in $O(n + m)$ time as detailed by [de Berg et al. \(2000\)](#).

To conclude this section, we complete the discussion of the Minkowski difference with an example of its application. That is, a complete motion planning example for the translating polygon. Figure 4.9 shows a workspace containing a translating polygonal robot, along with the configuration space obstacles and one possible path from start to goal.

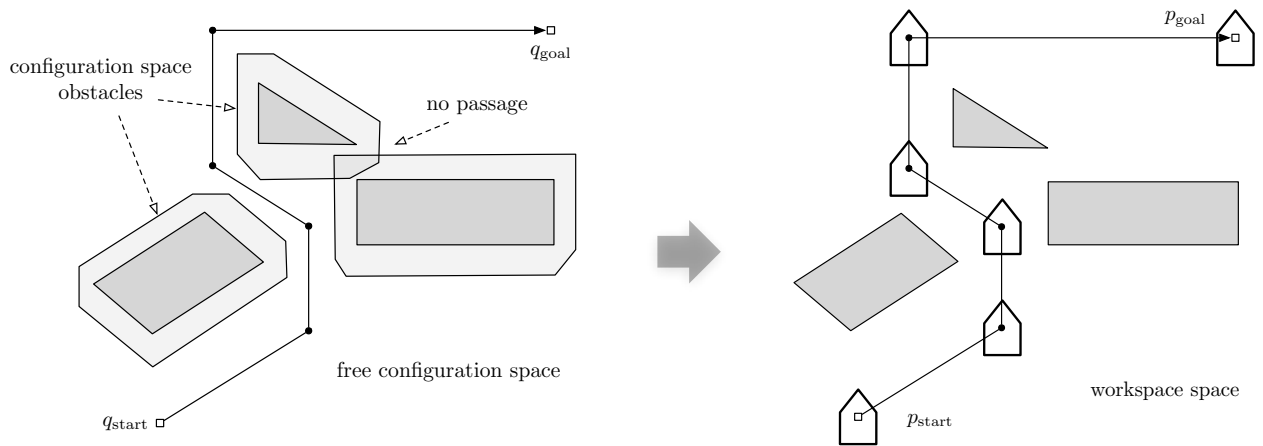


Figure 4.9: Planning the motion of a translating polygonal robot from start to goal

4.1.3 Free configuration space for the 2-link robot

The 2-link robot in Figure 4.10 illustrates how, for general shapes of obstacles and robots and for robots that can rotate, it is complex to compute exactly the free configuration space. Consider the

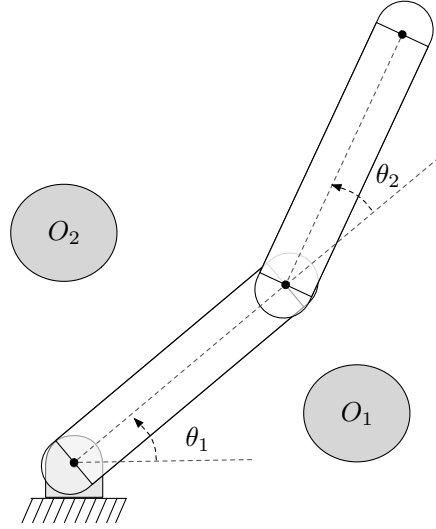


Figure 4.10: The two-link robot in a workspace with obstacles

2-link robot moving in a workspace with two obstacles O_1 and O_2 as depicted in Figure 4.10. Let us think about how to compute the free configuration space examining step-by-step the effect of the obstacles. Our three steps are depicted in the three images in Figure 4.11:

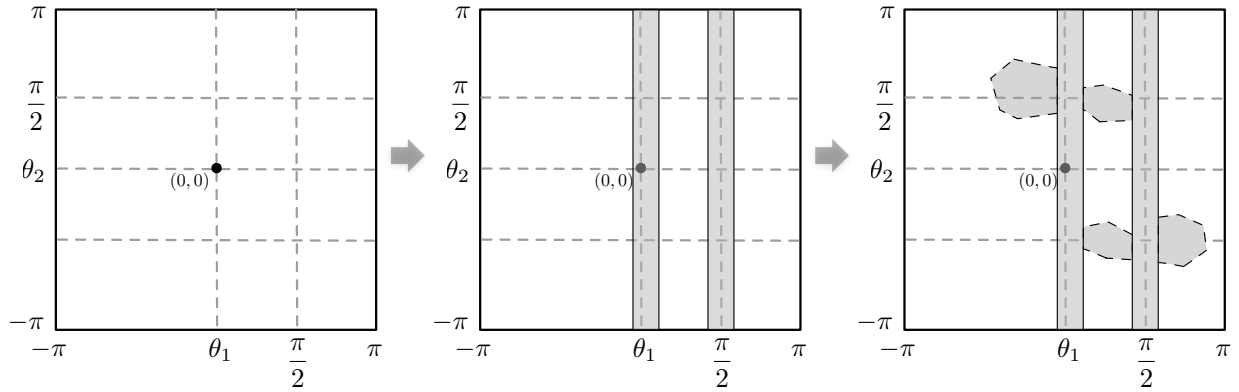


Figure 4.11: Step-by-step graphical approximate computation of the free configuration space for the 2-link robot. The shaded regions are obstacles in configuration space. The dashed boundaries are highly approximate because of the complexity inherent in their computation.

- (i) in the first image we recall that the 2-link configuration space is a 2-torus,

- (ii) in the second image we compute the set of angles θ_1 at which the first link hits one of the two obstacles. This set is the union of two intervals: the obstacle O_1 prohibits the angle θ_1 from taking value in an interval around $\theta_1 = 0$, and, similarly, the obstacle O_2 prohibits the angle θ_1 from taking value in an interval around $\theta_1 = \pi/2$. The two intervals in θ_1 are depicted as two vertical rectangles in the second image,
- (iii) in the third image, we look to compute the effect of the obstacles on the allowable values of θ_2 . However, one can begin to see the difficulty in accurately estimating the free configuration space. Given an angle θ_1 for which the first link is not in collision with O_1 and O_2 , we can approximately compute what θ_2 angles correspond to collision for the second link. This calculation is highly approximate as each choice of θ_1 changes the allowable values of θ_2 . Consequently, this computation is very hard to perform exactly for general shapes of the obstacles and of the robot links.

From this two-link robot example we can see that a more general method is needed to compute free configuration spaces. The following section introduces the idea of sampling.

4.2 Numerical computation of the free configuration space

By now we have studied multiple methods to describe the free configuration space. (1) For the disk robot and the translating polygon, we saw that we can explicitly characterize the free configuration space. (2) For the 2-link and the roto-translating polygon, it is hard to precisely compute the free configurations. In general, unfortunately, it is hard to have an explicit characterization of the obstacles in configuration space and, in turn, to decompose the free configuration space into convex subsets.

Therefore, we study here an alternative numerical method, based on sampling and collision detection.

The sampling & collision-detection algorithm

Input: A number of samples n , the free workspace W_{free} , and the robot configuration map $B(q)$

Output: A set of configurations in the free configuration space

- 1: Initialize `free-configs` = \emptyset
 - 2: Compute a sequence of sample configurations q_1, q_2, \dots, q_n
 - 3: **for** each configuration sample q_i in the sequence :
 - 4: compute the positions of the robot rigid bodies corresponding to the sample, $B(q_i)$
 - 5: detect if the robot collides with the obstacles (i.e., test if $B(q_i) \subset W_{\text{free}}$)
 - 6: **if** robot does not collide with obstacles and is inside the workspace :
 - 7: Add q_i to `free-configs`
 - 8: **return** `free-configs`
-

This numerical approach computes a “cloud representation” of the free configuration space for

robots and obstacles of arbitrary shape. Motivated by this numerical approach, the next two sections study algorithms for sampling and collision detection.

4.3 Sampling methods

A sampling method should have certain properties:

- (i) *Uniformity*: the samples should provide a “good covering” of space. Mathematically, this can be formulated using the notion of dispersion; see below.
- (ii) *Incremental property*: the sequence of samples should provide good coverage at any number n of samples. In other words, it should be possible to increase n continuously and not only in discrete large amounts.
- (iii) *Lattice structure*: given a sample, the location of nearby samples should be computationally easy to determine.

Consider the d -dimensional unit cube $X = [0, 1]^d \subset \mathbb{R}^d$. The *sphere-dispersion* and the *square-dispersion* of a set of points P in the set X are defined by (see also Figure 4.12)

$\text{dispersion}_{\text{sphere}}(P) = \text{radius of the largest empty disk, whose center lies in } X,$

$\text{dispersion}_{\text{square}}(P) = \frac{1}{2} \text{the length of the side of the largest empty square whose center lies in } X.$

Typically, square dispersion is calculated with respect to a fixed coordinate frame. The sides of the square are aligned with the axes of this coordinate frame and cannot be rotated.

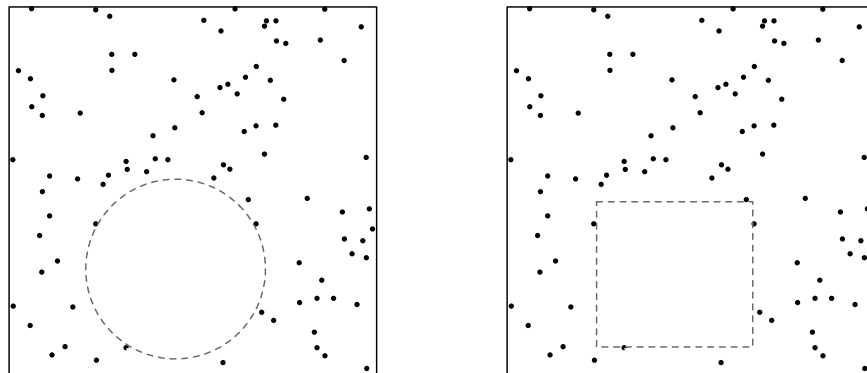


Figure 4.12: A set of points in the unit square with poor, that is, high sphere-dispersion (left figure) and square-dispersion (right figure)

More generally, given a distance metric defining the distance $\text{dist}(x, y)$ between any two points $x, y \in X$, we can define the dispersion of P with respect to the distance metric as

$$\text{dispersion}(P) = \max_{x \in X} \min_{p \in P} \text{dist}(x, p).$$

That is, the dispersion is defined as maximum distance from a point in X to its nearest sample in P . The sphere- and square-dispersion measures are given by the following distance metrics:

(i) sphere dispersion uses the 2-norm:

$$\text{dist}(x, p) = \|x - p\|_2 = \sqrt{(x_1 - p_1)^2 + \cdots + (x_d - p_d)^2},$$

(ii) square dispersion uses the ∞ -norm:

$$\text{dist}(x, p) = \|x - p\|_\infty = \max(|x_1 - p_1|, \dots, |x_d - p_d|),$$

where $x = (x_1, \dots, x_d)$ and $p = (p_1, \dots, p_d)$ are the coordinates of the points x and p .

Note: The sampling methods discussed in this chapter look only at sampling in a d -dimensional cube. These methods can be extended to sampling methods for general configuration spaces such as the sphere and the set of rotations.

Uniform grids There are two ways of defining uniform grids in the unit cube $X = [0, 1]^d$ as shown in Figure 4.13 for $d = 2$. We call them the *center grid* and the *corner grid*. Both grids with n points can be defined if $n = k^d$ for some number k . For $n = k^d$ for some k , the center grid is defined in two steps: (1) along each of the d dimensions, divide the $[0, 1]$ interval into k subintervals of equal length and therefore compute k^d sub-cubes of X , (2) place one grid point at the center of each sub-cube; see the left image in Figure 4.13. This center uniform grid is

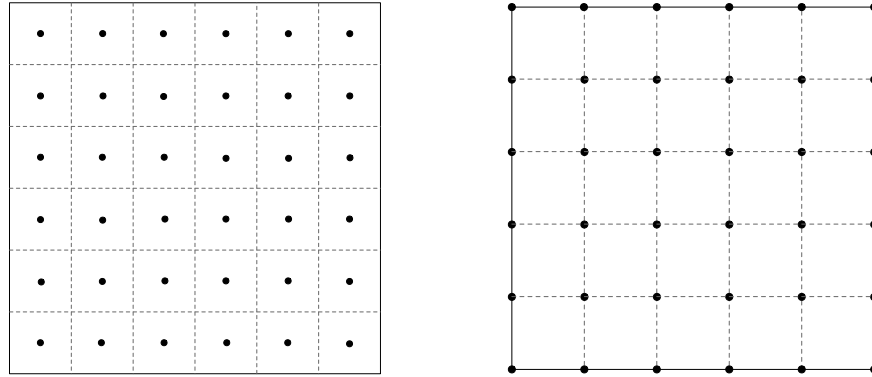


Figure 4.13: Uniform grids with $n = 36$ points in the 2-dimensional cube. Left figure: the center grid, also called the Sukharev grid. Right figure: the corner grid.

sometimes called the *Sukharev grid* and has minimal square-dispersion (Sukharev 1971) equal to

$$\text{dispersion}_{\text{square}}(P_{\text{center grid}}(n, d)) = \frac{1}{2\sqrt[d]{n}}.$$

For the corner grid, divide the $[0, 1]$ interval into $(k - 1)$ subintervals of equal length and therefore compute $(k - 1)^d$ sub-cubes of X , (2) place one grid point at each vertex of each sub-cube; see the right image in Figure 4.13.

Note: There is no uniform grid if n is not equal to k^d for some k : one can set $k := \lfloor \sqrt[d]{n} \rfloor$, place the k points as Figure 4.13 and the other points arbitrarily.

Note: It is possible to design multi-resolution versions of the uniform center grid, i.e., compute a uniform grid with some number of points and then add additional points while keeping the ones from the previous resolution level. This process is illustrated in Figure 4.14. The growth in number of points in multi-resolution uniform grids is exponential: one can see that a step increase in resolution correspond to a jump in number of samples from n to $n3^d$.

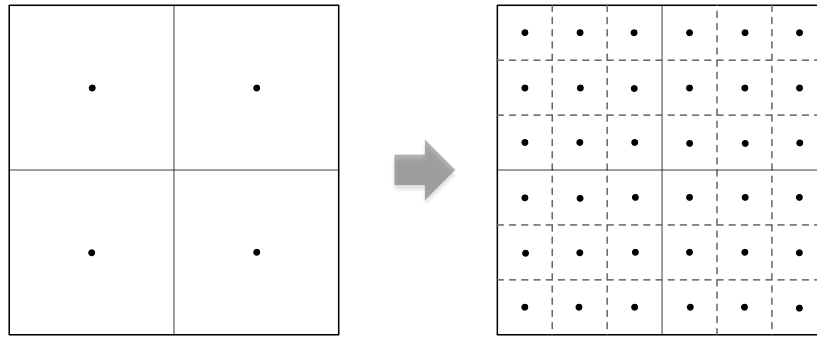


Figure 4.14: Growth in number of points when increasing resolution of uniform grid

Random and pseudo-random sampling Adopting a random number generator is usually a very simple approach to (uniformly or possibly non-uniformly) sample the cube $X = [0, 1]^d$. An example set of 100 randomly generated points is shown in Figure 4.15.

Note: Let P be a set of n points generated independently and uniformly over X . As $n \rightarrow \infty$, the set P has (Deheuvels 1983) square-dispersion of order $O((\ln(n)/n)^{1/d})$. Therefore, randomly-sampled points have asymptotically worse dispersion than center grids.

Programming Note: In Matlab the command `rand(n,d)` returns a vector with n rows and d columns: each row i gives a uniform sample q_i in $[0, 1]^d$.

In Python, uniform samples can be generated using the `random` module or the `numpy` module:

- (i) Import the `random` module using the command `import random`. A random number between 0 and 1 is generated with `random.random()`. A For-loop can then be used to generate n samples, in $[0, 1]^d$. Or,
- (ii) Import the `numpy` module using the command `import numpy`. An array containing n random numbers between 0 and 1 is generated using `numpy.random.sample(n)`. Running this d times, d -dimensional coordinates for each sample are generated.

Deterministic sampling sequences Halton sequences (Halton 1960) are an elegant way of sampling an interval with good uniformity (better than a pseudorandom sequence, though not as

good as the optimal center grid) and with the incremental property (which the center grid does not possess). Each scalar Halton sequence is generated by a prime number. In what follows we provide (1) an example, (2) the exact definition, and (3) a simple algorithm.

By way of an example, the Halton sequence generated by the prime number 2 is given by

$$\frac{1}{2}, \quad \frac{1}{4}, \frac{3}{4}, \quad \frac{1}{8}, \frac{5}{8}, \frac{3}{8}, \frac{7}{8}, \quad \frac{1}{16}, \frac{9}{16}, \quad \dots$$

The horizontal spacing illustrates a shift in accuracy of the sequence. To understand the Halton sequence we require a formal definition (you might want to review the notion of [Wikipedia:Numerical system](#)). Given a number p , one can write any number i in base p as

$$i = e_j p^j + \dots + e_1 p + e_0, \quad \text{where } e_j, \dots, e_0 \in \{0, 1, \dots, p-1\}.$$

The digits of i in base p are then $e_j e_{j-1} \dots e_0$. For example, the number 6 is written in $p = 2$ (i.e., binary) as 110.

Then, the method for attaining the 6th number in the Halton sequence, using $p = 2$, is as follows:

- (i) Write the number i in base p , where p is a prime number:

$$6 \text{ in base } 2 \text{ is } 110,$$

- (ii) add a decimal place and then reverse the digits, including the decimal:

$$110.0 \text{ becomes } 0.011,$$

- (iii) convert this binary number back to base 10 and output this as the i th number in the Halton sequence:

$$0.011 = 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 1 \cdot \frac{1}{8} = \frac{3}{8}.$$

We can write a formula for the i th number base p as

$$\text{"}i\text{th sample of Halton sequence in base } p\text{"} = \frac{e_0}{p} + \frac{e_1}{p^2} + \dots + \frac{e_j}{p^{j+1}}.$$

We can write this procedure as an algorithm to compute a Halton sequence of length N generated by any prime number. Repeating our previous example, let us compute the 6th sample of the Halton sequence in base 2, using the algorithm:

- (i) First we initialize $i_{\text{tmp}} = 6$, $S(6) = 0$, $f = 1/2$,
- (ii) 6 divided by 2 gives a quotient $q = 3$ and remainder $r = 0$. We set $S(6) = 0$, $i_{\text{tmp}} = 3$, $f = 1/4$,

The Halton sequence algorithm

Input: length of the sequence $N \in \mathbb{N}$ and prime number $p \in \mathbb{N}$

Output: an array $S[1 \dots N]$ with the first N samples of the Halton sequence generated by p

```

1: initialize:  $S$  to be an array of  $N$  zeros (i.e.,  $S(i) = 0$  for each  $i$  from 1 to  $N$ )
2: for each  $i$  from 1 to  $N$  :
3:     initialize:  $i_{\text{tmp}} = i$ , and  $f = 1/p$ 
4:     while  $i_{\text{tmp}} > 0$  :
5:         compute the quotient  $q$  and the remainder  $r$  of the division  $i_{\text{tmp}}/p$ 
6:          $S(i) = S(i) + f \cdot r$ 
7:          $i_{\text{tmp}} = q$ 
8:          $f = f/p$ 
9: return  $S$ 

```

(iii) 3 divided by 2 gives a quotient $q = 1$ and remainder $r = 1$. We set $S(6) = 1/4$, $i_{\text{tmp}} = 1$, $f = 1/8$,

(iv) 1 divided by 2 gives a quotient $q = 0$ and remainder $r = 1$. We set $S(6) = 1/4 + 1/8$, $i_{\text{tmp}} = 0$, $f = 1/16$.

For Halton sequences in higher dimensions, select a prime number for each dimension of the problem: usually, the number 2 for the first dimension, the number 3 for the second dimension, and so forth. The i th Halton sample in d -dimension is a point in $[0, 1]^d$ whose components are the i th samples of the d sequences computed with d different prime numbers. For example, in $[0, 1]^2$, using the primes 2 and 3, one has

$$\left(\frac{1}{2}, \frac{1}{3}\right), \left(\frac{1}{4}, \frac{2}{3}\right), \left(\frac{3}{4}, \frac{1}{9}\right), \left(\frac{1}{8}, \frac{4}{9}\right), \left(\frac{5}{8}, \frac{7}{9}\right), \left(\frac{3}{8}, \frac{2}{9}\right), \left(\frac{7}{8}, \frac{5}{9}\right), \left(\frac{1}{16}, \frac{8}{9}\right), \left(\frac{9}{16}, \frac{1}{27}\right), \dots$$

Figure 4.15 shows the first 100 samples points of the Halton sequence in 2-dimension generated by the prime numbers 2 and 3.

Note: It is known that the square-dispersion of a Halton sequence of n samples is $f(d)/\sqrt[n]{n}$, where $f(d)$ is a constant for each dimension d . Thus, the Halton sequence achieves a dispersion similar to that of uniform grids, but has the advantage of allowing for incremental increases in the number of samples.

Comparison of sampling methods We have seen three sampling methods: uniform grids, random sampling, and deterministic sampling via the Halton sequence. The uniform grid is optimal in terms of dispersion. It also possess the lattice property in that for a given grid resolution the neighbors of a particular grid point are predetermined (i.e., they are the samples in adjacent grid squares). However, uniform grids are not incremental, and one must jump from n to $n3^d$ samples to increase the resolution.

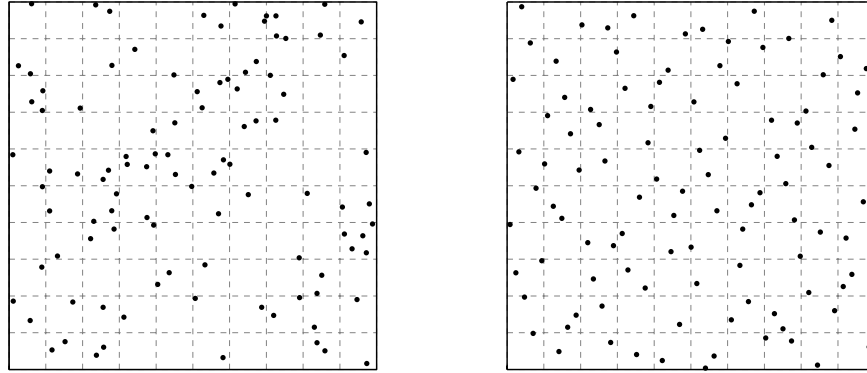


Figure 4.15: Left figure: a pseudorandom set of 100 samples. Right figure: the Halton sequence in 2-dimension generated by the prime numbers 2 and 3.

Random sampling has higher dispersion, but is incremental. It does not, however, possess the lattice structure – the neighbors of a particular sample are not predetermined and must be computed by searching over all other samples.

Halton sequences have the advantage of achieving nearly the same dispersion as uniform grids while also being incremental. The lattice structure of Halton sequences is not as simple as that of the uniform grid. However, given a number of samples n along with the base used for each dimension, the neighbors of a given sample are pre-determined, although this calculation is somewhat more complex.

Table 4.1 summarizes the three methods and their properties.

| Sampling property | Uniform grids | Random sampling | Halton sequences |
|-------------------|----------------------------|---------------------------------------|-------------------------------|
| dispersion | $O(\frac{1}{\sqrt[n]{n}})$ | $O(\frac{\ln^{1/d}(n)}{\sqrt[n]{n}})$ | $O(\frac{f(d)}{\sqrt[n]{n}})$ |
| incremental | no | yes | yes |
| lattice | yes | no | yes (more complex) |

Table 4.1: The three sampling methods and their key properties

4.4 Collision detection methods

We now have several different methods for generating samples q_1, \dots, q_n in the configuration space Q . The next step is to detect whether or not these configurations are in collision with an obstacle or workspace boundary. To do this we require several collision detection methods.

Problem 4.2. *Given two bodies B_1 and B_2 , determine if they collide. (In equivalent set-theoretic words, determine if the intersection between two sets is non-empty.)*

The *distance between two sets A and B* is

$$\text{dist}(A, B) = \inf_{a \in A} \inf_{b \in B} \text{dist}(a, b).$$

If the distance is zero, then we say the bodies are in collision. Of course, it is undesirable to check collision by computing the pairwise distance between any two points. Therefore, it is convenient to devise careful algorithms for collision detection.

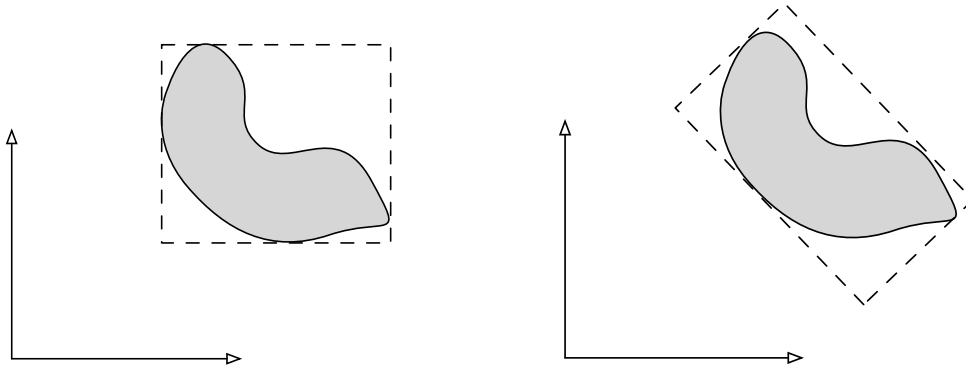


Figure 4.16: Bounding boxes are used to over-approximate sets in collision detection problems. Left figure: Axis-aligned Bounding Box (AABB). Right figure: Oriented Bounding Box (OBB).

Loosely speaking, it is computationally easier to deal with 2-dimensional problems rather than 3-dimensional problems. Also, it is computationally easier to deal with the following shape (in order of complexity):

- (i) bounding spheres, rather than
- (ii) Axis-Aligned Bounding Boxes (AABB), rather than
- (iii) Oriented Bounding Boxes, rather than
- (iv) convex polygons, rather than
- (v) non-convex polygons, rather than
- (vi) arbitrary shapes.

4.4.1 Basic primitive #1: is a point in a convex polygon?

Problem 4.3. *Given a convex polygon and a point, determine if the point is inside the polygon.*

The polygon is defined by a counter-clockwise sequence of vertices, p_1, \dots, p_4 in Figure 4.17. For each side of the polygon, we define the *interior normal* as in Figure 4.18 for the side $\overline{p_1 p_2}$.

Note: a convex polygon can be equivalently represented as either (1) a set of n points (the polygon having those points as vertices) or (2) a set of n half-planes (the polygon being the intersection of the half-planes). These two equivalent representations are called Vertex Description

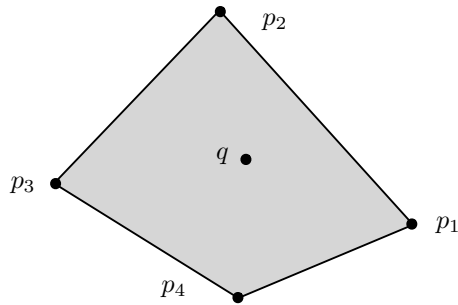


Figure 4.17: Testing if a point lies in a polygon

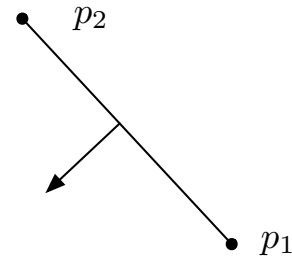


Figure 4.18: Interior normal to a side of the polygon

and Half-plane Description of a convex polygon. With some computational cost, it is possible to convert one representation into the other.

Given a convex polygon with counter-clockwise vertices $\{p_1, \dots, p_n\}$ and a point q , the following conditions are equivalent:

- (i) the point q is in the polygon (possibly on the boundary),
- (ii) for all $i \in \{1, \dots, n\}$, the point q belongs to the half-plane with boundary line passing through the vertices p_i and p_{i+1} and containing the polygon, and
- (iii) for all $i \in \{1, \dots, n\}$, the dot product between the interior normal to the side $\overline{p_i p_{i+1}}$ and the segment $\overline{p_i q}$ is positive or zero.

(Here the convention is that $p_{n+1} = p_1$. A similar set of results can be given to check that the point is strictly inside the polygon.)

4.4.2 Basic primitive #2: do two segments intersect?

Problem 4.4. *Given two segments, determine if they intersect.*

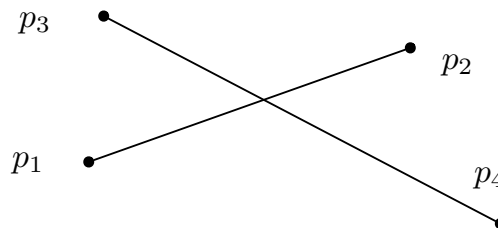


Figure 4.19: Testing for the intersection of two segments

Note: any two lines in the plane are in one of three exclusive configuration: (1) parallel and coincident, (2) parallel and distinct, or (3) intersecting at a single point. In order for two segments to intersect, the two corresponding lines may be coincident (and the two segments need to intersect at least partly) or may intersect at a point (and the point must belong to the two segments).

A segment is described by its two vertices as shown in Figure 4.19. Each point p_a belonging to the segment $\overline{p_1 p_2}$ can be written as

$$p_a = p_1 + s_a(p_2 - p_1), \quad \text{for } s_a \in [0, 1].$$

Similarly, we have

$$p_b = p_3 + s_b(p_4 - p_3), \quad \text{for } s_b \in [0, 1].$$

Assume $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, $p_3 = (x_3, y_3)$, and $p_4 = (x_4, y_4)$. The equality $p_a = p_b$ is equivalent to two linear equations in the two unknowns s_a, s_b :

$$\begin{aligned} x_1 + s_a(x_2 - x_1) &= x_3 + s_b(x_4 - x_3), \\ y_1 + s_a(y_2 - y_1) &= y_3 + s_b(y_4 - y_3). \end{aligned}$$

These two equations can be solved and, for example, one obtains

$$s_a = \frac{(x_4 - x_3)(y_1 - y_3) - (y_4 - y_3)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} =: \frac{\text{num}}{\text{den}}. \quad (4.2)$$

One can show that

- (i) if $\text{num} = \text{den} = 0$, then the two lines are coincident,
- (ii) if $\text{num} \neq 0$ and $\text{den} = 0$, then the two lines are parallel and distinct, and
- (iii) if $\text{num} \neq 0$ and $\text{den} \neq 0$, then the two lines are not parallel and therefore intersect at a single point.

Now, if the two lines intersect at a point, one still needs to check that the intersection point actually belongs to the segment. This fact can be checked by solving for the coefficients s_a and s_b and verifying that they belong to the interval $[0, 1]$.

4.4.3 Basic primitive #3: do two convex polygons intersect?

Problem 4.5. *Given two convex polygons, determine if they intersect.*

There are five possible cases that one must consider, and each is illustrated in Figure 4.20. It is computationally easy to distinguish case (1) (no collision), from cases (2), (3) and (4). To distinguish case (5) from case (1) is a bit more complex.

Based on these five cases we can define an algorithm for determining if two convex polygons intersect.

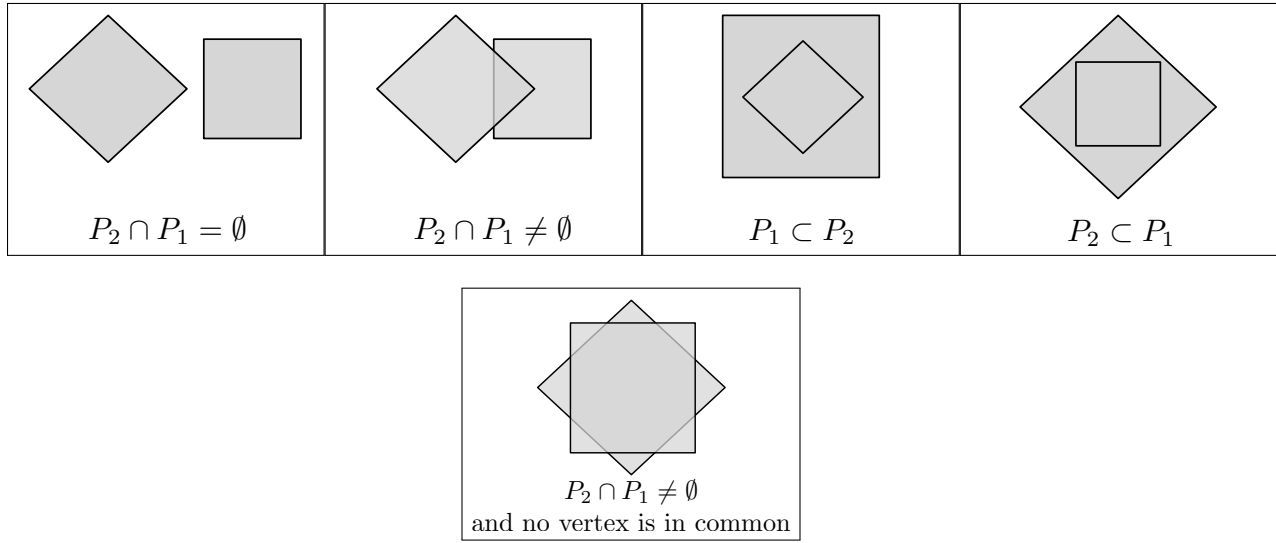


Figure 4.20: The five possible cases for determining if two convex polygons intersect

The polygon-intersection algorithm**Input:** two convex polygons P_1 and P_2 **Output:** collision or no collision

- 1: **if** (any vertex of P_1 belongs to P_2) OR (any vertex of P_2 belongs to P_1) :
- 2: **return** collision
- 3: **if** any edge of P_1 intersects any edge of P_2 :
- 4: **return** collision
- 5: **return** no collision

4.4.4 Extension to non-convex polygons

Collision detection can be extended to non-convex polygons. First, we need to extend primitive #1 to test if a point lies in a non-convex polygon.

Problem 4.6. *Given a non-convex polygon and a point, determine if the point is inside the polygon.*

We solve this problem using an algorithm called *ray shooting*. A *ray* is a line with an endpoint that extends infinitely in one direction. It can be characterized by the endpoint o , and a unit vector v that points in the direction that the line extends. The procedure is as follows.

The algorithm counts the number of times a ray extending from q intersects the boundary of the polygon. By the Jordan Curve Theorem 1.5 and Figure 1.16, if the number of intersections is odd, then q must lie inside the polygon. If there are an even number of intersections, then q must lie outside the polygon. Figure 4.21 shows an example in which the ray intersects three segments, s_1 , s_2 , and s_3 . Since the number of intersections is odd, the point q lies in the polygon.

The ray shooting algorithm requires that we test if a ray and a line segment intersect as shown on the left of Figure 4.22.

Ray-shooting algorithm for point in non-convex polygon

Input: a point q and a non-convex polygon P with n sides s_1, \dots, s_n .

Output: inside or outside

- 1: Choose an arbitrary direction, and define a ray R extending from q in the chosen direction.
 - 2: `intersections = 0`
 - 3: **for** i from 1 to n :
 - 4: **if** segment s_i intersects the ray R :
 - 5: `intersections = intersections + 1`
 - 6: **if** intersections is odd :
 - 7: **return** inside
 - 8: **return** outside
-

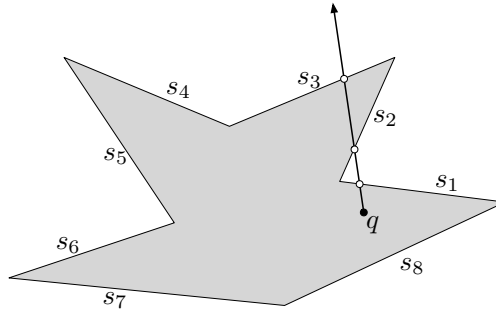


Figure 4.21: The ray shooting algorithm. The point q lies inside the polygon, and thus the ray intersects the boundary an odd number of times.

Problem 4.7. Given a line segment s and a ray R , determine if they intersect.

We can solve this problem by thinking of the ray as a very long segment and then applying primitive #2. The question is how long do we need to make this segment? Notice that the farthest point from o on the line segment s is one of its end points, p_1 or p_2 . Thus, if the ray and segment intersect, then the intersection point is at a distance of at most

$$d = \max\{\|o - p_1\|, \|o - p_2\|\}$$

from the ray endpoint o . Based on this we define a line segment with end points $p_3 = o$ and $p_4 = o + dv$ as shown on the right of Figure 4.22 and test it intersects with the line segment defined by p_1 and p_2 using primitive #2. The segment and ray intersect if and only if the two segments intersect.

Third and finally we can test if two non-convex polygons intersect.

Problem 4.8. Given two non-convex polygons, determine if they intersect.

For this step, we can now use the exact same algorithm as in primitive #3. The two non-convex polygons must be in one of the five cases shown in Figure 4.20 and thus we can apply the

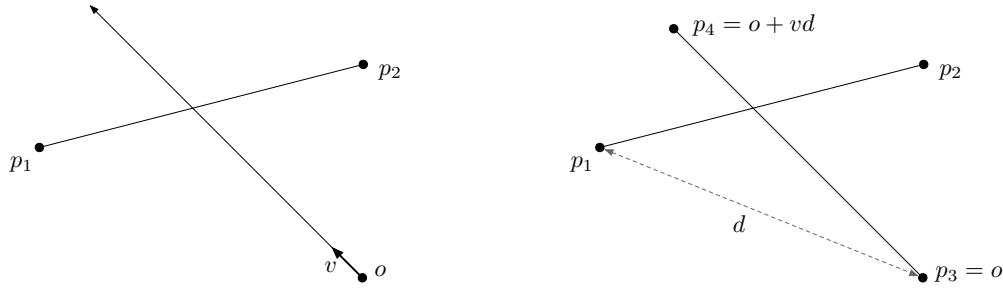


Figure 4.22: Left figure: Intersection of a ray and a segment. Right figure: The ray is converted to a segment with length equal to the distance from o to p_1 .

polygon-intersection algorithm using ray shooting to test if the vertices of one polygon lie in the other.

4.4.5 Final comments on collision detection

Here are some lessons and some final comments:

- (i) collision detection algorithms for simple objects are easy to perform,
- (ii) for complex objects, e.g., arbitrary shapes, a reasonable approach is to use hierarchical approximations and decompositions, described as follows:
 - a) approximate the complex shape by a simple enclosing shape, e.g., a sphere, an AABB, or an OBB,
 - b) if no collision occurs between the two simple enclosing shapes, then return a “no collision” result,
 - c) if a collision is detected between two simple enclosing shapes, then approximate the bodies less conservatively and more accurately, e.g., by decomposing them into the union of multiple simple shapes. One can then check collision between these more accurate decompositions;
- (iii) to detect collisions between moving objects, discretize time and perform a collision detection test for each time step, and
- (iv) in industrial motion planning applications, collision detection is usually performed via a stand-alone black-box subroutine, e.g., see (Pan et al. 2012).

4.5 Appendix: Runtime of the numerical computation of the free configuration space

Let us now look at the runtime of the sampling & collision-detection algorithm. To do this, we first characterize the time complexity of the algorithms that are used for sampling and for collision

detection in the overall algorithm (de Berg et al. 2000)

- (i) each of the sampling methods (uniform grids, random sampling, and pseudo-random sampling) for generating n sample points, runs in $O(n)$ time;
- (ii) checking if a point belongs to convex polygon with n vertices as in basic primitive #1 has complexity $O(n)$;
- (iii) checking if two lines intersect as in basic primitive #2 can be done in $O(1)$ time by simply plugging the line endpoints into the corresponding equations;
- (iv) given two convex polygons, with n and m vertices respectively, our algorithm for checking if they intersect as in basic primitive #3 has a runtime in $O(nm)$. Our algorithm for checking if two non-convex polygons intersect has the same runtime of $O(nm)$. Faster algorithms exist with runtime in $O((n + m) \log(n + m) + I \log(n + m))$, where I is the number of intersections between the two polygons. This faster algorithm works for both convex and non-convex polygons.

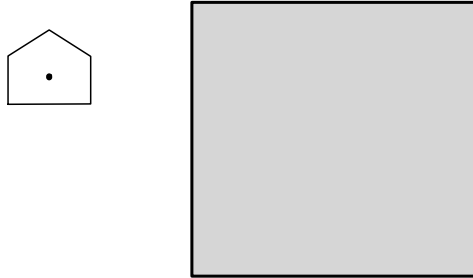
Returning to the sampling & collision detection algorithm, the runtime of this algorithm is dominated by the time to check collisions between the robot body and each obstacle. For each configuration q_i we first compute the polygon describing the robot at that configuration, $B(q_i)$, and then loop over each obstacle polygon O_j , checking for an intersection (i.e., collision) between $B(q_i)$ and O_j . The run time of each collision check is given by the runtime of basic primitive #3, where n is the number of vertices in the robot polygon, and m is the number of vertices in the obstacle polygon above. This must be repeated for each obstacle (or until the first collision is found), and for each sample point.

4.6 Exercises

E4.1 Minkowski difference (15 points).

The Minkowski difference of a convex polygonal obstacle and a translating convex polygonal robot is a useful method for computing configuration space obstacles. In this exercise you will explore how to find the configuration space obstacle for the ship-shaped robot and square obstacle below.

- (i) (5 points) First, sketch the trajectory of the robot's reference point as you slide the robot body around the outside of the square obstacle. This is a graphical approach to sketching the configuration space obstacle.
- (ii) (5 points) Next, in a new drawing of the obstacle use the Minkowski difference method to sketch the configuration space obstacle.
- (iii) (5 points) In a couple of sentences, explain why the Minkowski difference approach in (ii) is well suited for implementation in a computer program. (Note: there do exist fast computational methods for finding the convex hull of a collection of points.)



E4.2 Dispersion (20 points). This question explores how dispersed, or well placed, a set of sample points are over a space. First, consider the unit interval from $[0, 1]$.

- A set of three sample points $\{A, B, C\}$ inside $[0, 1]$ are given. For any point p inside $[0, 1]$, the *distance* from p to the set $\{A, B, C\}$ is the smallest of the three distances $|p - A|$, $|p - B|$, and $|p - C|$.
- The *dispersion* of the set $\{A, B, C\}$ is the largest of the distances from all positions p inside $[0, 1]$ to the sample set $\{A, B, C\}$.

Verify that this definition is equivalent to that for sphere-dispersion and square-dispersion for the case of a 1-dimensional segment. Answer the following question:

- (i) (4 points) What is the optimal placement for 3 sample points that minimizes their dispersion? And what is the corresponding dispersion?

Next, consider the unit cube $[0, 1]^d$ in d -dimensions and compute the dispersion of a *corner uniform grid* (see Section 4.3) with k samples on each dimension of the cube, as a function of k , for the following cases:

- (ii) (4 points) The sphere-dispersion in 2 dimensions ($d = 2$).
- (iii) (4 points) The square-dispersion in 2 dimensions ($d = 2$).
- (iv) (4 points) The sphere-dispersion in arbitrary dimensions.
- (v) (4 points) The square-dispersion in arbitrary dimensions.

E4.3 Programming: Sampling algorithms (40 points).

Consider the unit square $[0, 1]^2$ in the plane. Pick an arbitrary k and do, for number of samples, write formulas for the coordinates of:

- (i) (5 points) write formulas for the $n = k^2$ sample points in the uniform Sukharev center grid,

- (ii) (5 points) write formulas for the $n = k^2$ sample points in the uniform corner grid, where sample points are placed at the corners of each squarelet (instead of at centers),
- (iii) write the following programs (representing a grid with n entries in $[0, 1]^2$ by an array with n rows and 2 columns):

`computeGridSukharev` (10 points)

Input: the number of samples n (assuming $n = k^2$ for some number k)

Output: the uniform Sukharev center grid on $[0, 1]^2$ with $\lfloor \sqrt{n} \rfloor$ samples along each axis.

`computeGridRandom` (10 points)

Input: the number of samples n

Output: a random grid on $[0, 1]^2$ with n uniformly-generated samples

`computeGridHalton` (10 points)

Input: the number of samples n , two prime numbers b_1 and b_2

Output: a Halton sequence of n samples inside $[0, 1]^2$ generated by the two prime numbers b_1 and b_2

For each function, do the following:

- (i) explain how to implement the function, possibly deriving analytic formulas, and characterize special cases,
- (ii) program the function, including correctness checks on the input data and appropriate error messages, and
- (iii) verify your function is correct by plotting the three grids for $n = 100$.

E4.4 **Programming: Collision detection primitives (45 points).** Implement the three collision detection primitives for convex polygons discussed in Subsections 4.4, i.e., write the following programs:

`isPointInConvexPolygon` (15 points)

Input: a point q and a convex polygon P

Output: true or false

`doTwoSegmentsIntersect` (15 points)

Input: two segments described by their respective vertices p_1, p_2 and p_3, p_4

Output: the answer true or false and, if the answer is true, the intersection point

`doTwoConvexPolygonsIntersect` (15 points)

Input: two convex polygons P_1 and P_2

Output: true or false

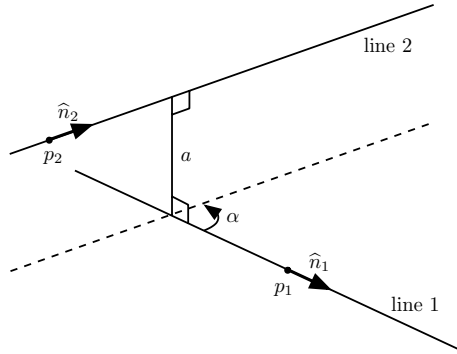
For the segments intersection test, verify that, given the definitions of `num` and `den` in equation (4.2),

- (i) if `num` = `den` = 0, then the two lines are coincident,
- (ii) if `num` \neq 0 and `den` = 0, then the two lines are parallel and distinct, and
- (iii) if `num` \neq 0 and `den` \neq 0, then the two lines are not parallel and therefore intersect at a single point.

For each function, do the following:

- (i) explain how to implement the function, possibly deriving analytic formulas, and characterize special cases; specifically, write a pseudo-code routine to check whether a point is inside a convex polygon,
- (ii) program the function, including correctness checks on the input data and appropriate error messages, and
- (iii) verify your function is correct on a broad range of test inputs.

- E4.5 **Distance between two lines in three dimensions (20 points).** Consider the following idealized collision detection problem between two aircraft moving along two lines. As in Figure, let line 1 pass through a point p_1 with a unit-length vector \hat{n}_1 , and let line 2 pass through a point p_2 with a unit vector \hat{n}_2 . For simplicity, assume that the two lines do not intersect and are not parallel, as in Figure.



- (i) The *mutual perpendicular segment* is the unique shortest segment connecting the two lines (i.e., the segment touches line 1 and 2) and perpendicular to both lines;
- (ii) the *distance between the two lines* is the length a of the mutual perpendicular segment; and
- (iii) the *angle between the two lines* is the positive angle $\alpha \in [0, \pi]$ about the mutual perpendicular segment (measured according to the right-hand-rule, with orientation from line 1 to line 2) that line 1 needs to be rotated by, so that it is parallel to line 2.

Give expressions for a and α as functions of the points p_1, p_2 and the vectors \hat{n}_1, \hat{n}_2 .

Hint: Pay attention to the correct signs

Motion Planning via Sampling

This chapter presents general roadmap-based approaches to motion planning. Specifically, we will discuss:

- (i) general roadmaps and their desirable properties,
- (ii) complete planners based on exact roadmap computation (specifically, we will review decomposition-based roadmaps and will introduce a novel shortest-paths visibility graph),
- (iii) general-purpose planners based on sampling and approximate roadmaps. (Sampling-based Roadmap Methods) For this general-purpose planners we will discuss:
 - connection rules for fixed resolution grid-based roadmaps,
 - connection rules for arbitrary-resolution methods,
 - comparison between sampling-based approximate and exact planners
- (iv) incremental sampling-based planning methods, including:
 - from multi-query to single-query,
 - rapidly-exploring random trees (RRT),
 - the application of receding-horizon incremental planners to sensor-based planning.
- (v) Appendix: shortest-path planning via visibility roadmap and shortest path search via Dijkstra's algorithm.

The concepts and presentation in this chapter are inspired by Chapter 5 in [\(LaValle 2006\)](#).

5.1 Roadmaps

Generalizing what we discussed in Chapter 2, a roadmap is here understood to be a collection of locations in the configuration space along with paths connecting them. With each path, we associate a positive weight that represents a cost for traveling along that path, for example, the

path length or the travel time. More formally, we can think of a roadmap as a *weighted graph* $G = (V, E, w)$, where V is the set of nodes representing robot configurations, E is the set of edges representing paths between nodes, and w is a function that assigns the weight (e.g., path length) to each edge in E .

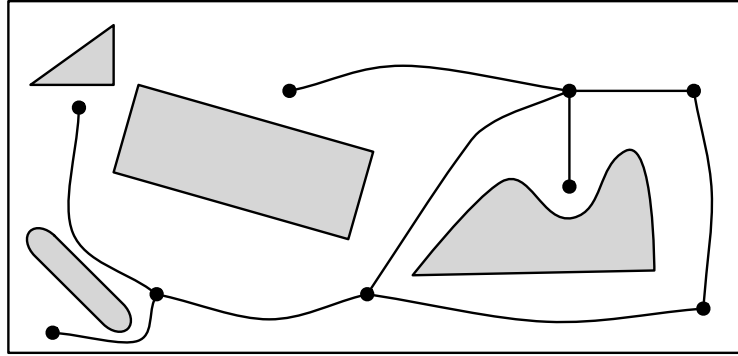


Figure 5.1: An example roadmap. The black dots are the nodes V , the paths connecting nodes are edges E , and the length of the path gives the edge's weight.

A roadmap may have the following properties:

- (i) the roadmap is *accessible* if, for each point q_{start} in Q_{free} , there is an easily computable path from q_{start} to some node in the roadmap,
- (ii) similarly, the roadmap is *departable* if, for each point q_{goal} in Q_{free} , there is an easily computable path from some node in the roadmap to q_{goal} , and
- (iii) the roadmap is *connected* if, as in graph theory, any two locations of the roadmap are connected by a path in the roadmap, and
- (iv) the roadmap is *efficient with factor $\delta \geq 1$* if, for any two locations in the roadmap, say u and v , the path length from u to v along edges of the roadmap is no longer than δ times the length of the shortest path from u to v in the environment.

The notions of accessibility and departability are not fully specified as they depend upon the notion of “easily computable path.” For example, if “easily computable paths” are straight lines from start locations q_{start} to nodes of the roadmap, then the roadmap depicted in Figure 5.1 is not accessible from start locations in the top left corner, behind the triangular obstacle.

5.2 Complete planners on exact roadmaps

Decomposition-based roadmaps In Chapter 2 we studied how to generate roadmaps using decomposition algorithms. For polygonal environments with polygonal obstacles, the sweeping trapezoidation algorithm decomposes the free environment into a collection of pair-wise adjacent

convex subsets; see Figure 5.2. The roadmap computed via the decomposition algorithm is guaranteed to be accessible and departable (via straight segments) and to be connected. It is clear that this roadmap, however, does not contain shortest paths among environment locations.

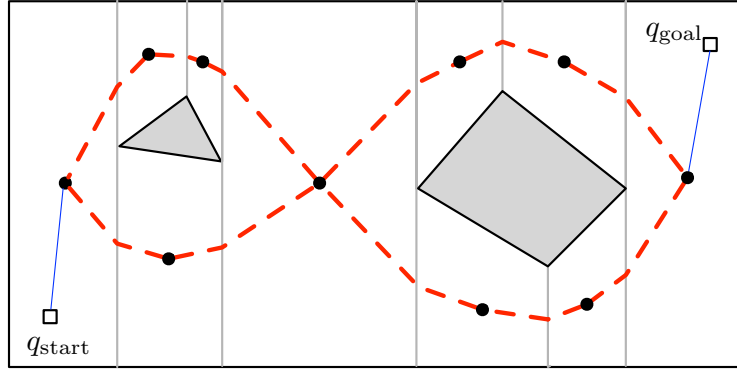


Figure 5.2: An exact roadmap (black points and thick dashed red paths) obtained via the sweeping trapezoidation algorithm. The thin blue paths connect start and goal locations to the roadmap.

Visibility roadmaps Next, we propose a new approach to computing efficient roadmaps in environments with polygonal obstacles. Given a set of polygonal obstacles O_1, \dots, O_n , define the *visibility graph* $G = (V, E, w)$ with node set V , and edge set E , and weights w as follows:

- (i) the nodes V of the visibility graph are all *convex vertices* of the polygons O_1, \dots, O_n , and
- (ii) the edges E of the visibility graph are all pairs of vertices that are *visibly connected*. That is, given two nodes u and v in V , we add the edge $\{u, v\}$ to the edge set E if the straight line segment between u and v is not in collision with any obstacle.
- (iii) the *edge weight* of an edge $\{u, v\}$ is given by the length of the segment connecting u and v .

Note: recall that a vertex of a polygon is a *convex vertex* if its interior angle is strictly smaller than π radians. (A non-convex vertex has an interior angle larger than π .) The non-convex vertices of a polygonal obstacle is not needed in the visibility graph.

Figure 5.3 illustrates a visibility graph whose vertices are the black disks and whose edges are the red dashed segments.

Then, given a start location q_{start} and a goal location q_{goal} , along with a visibility graph G of the obstacles, we connect q_{start} and q_{goal} to all nodes in the visibility graph that are visible from q_{start} (that is, all nodes that can be connected with a collision-free segment). In this graph we can search for a path from the start to goal.

Note: The visibility graph we have defined does not include the start and goal locations. This is done so that the visibility graph represents the configuration space (i.e., the working environment) and not the specific planning problem at hand (which is specified by the start and

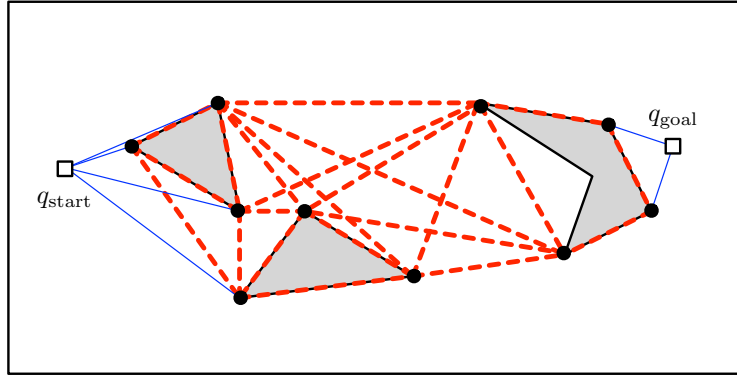


Figure 5.3: A roadmap obtained as the visibility graph generated by three polygonal obstacles. The roadmap edges are drawn in thick dashed red. The thin blue paths connect start and goal locations to the roadmap.

goal locations). One can then use the same visibility graph for multiple start-to-goal problems, rather than re-computing the graph each time.

One can show that the visibility graph has the following properties: if the free environment is connected, then the visibility graph is connected, departable, accessible. We can also characterize the efficiency of the visibility graph in the following theorem.

Theorem 5.1 (Shortest paths through convex polygonal obstacles). *Consider a configuration space with convex polygonal configuration space obstacles. Any shortest path in the free configuration space between q_{start} and q_{goal} consists of only straight line segments. Moreover, each segment endpoint is either the start location q_{start} , the goal location q_{goal} , or an obstacle vertex.*

This theorem states that the shortest path from start to goal is a path in the visibility graph. Hence, the roadmap obtained via the visibility graph is optimally efficient, in the sense that the efficiency factor δ is 1. For completeness we include a proof, which can be skipped without loss of continuity.

Proof. Let P be a shortest path from q_{start} and q_{goal} . First, suppose by way of contradiction that this shortest path does not consist solely of straight line segments. Since the obstacles are polygonal, there is a point x on the path that lies in the interior of the free space Q_{free} with the property that the path P is curved at x , as shown in the left of Figure 5.4. Since x is in the interior of the free space, there is a disc of radius $r > 0$ centered at x that is completely contained in the free space. The path P passes through the center of this disc, and is curved. So, we can shorten the path by replacing it with a straight line segment connecting the point where P enters the disc to the point where it leaves the disc. This contradicts the optimality of P . Thus, any shortest path consists only of straight line segments.

Next, let's consider an endpoint of a segment on P that is not the start or goal. The endpoint cannot lie in interior of Q_{free} . If it did then there would be a disc centered at the endpoint that is completely contained in the free space, and we could replace the subpath of P inside the disc

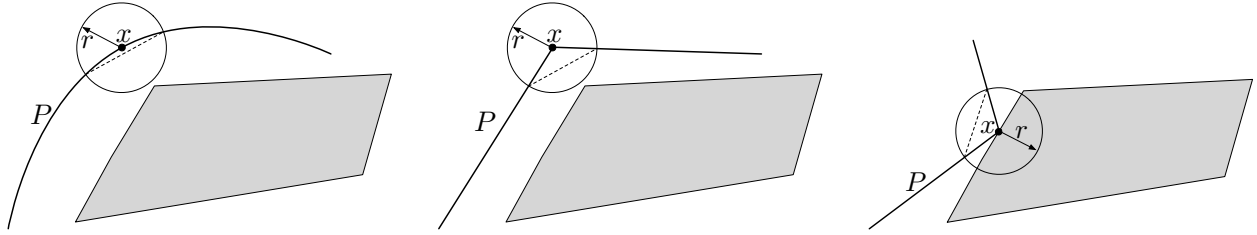


Figure 5.4: The three cases of shortening a suboptimal path for the proof of the optimality of the visibility graph

by a shortest straight line segment as shown in the center of Figure 5.4. Similarly, the endpoint cannot lie on the interior of an edge of an obstacle (i.e., on the boundary of an obstacle, but not at a vertex): if it did, then there would be a disc centered at the endpoint such that half the disc is in the free space, which again implies that we can replace the subpath inside the disc with a straight line segment, as shown on the right of Figure 5.4. The only possibility left is that the endpoint is an obstacle vertex concluding the proof. ■

The last question is how long it takes to compute the visibility graph. The number of nodes in the visibility graph is equal to the total number of obstacle vertices in the environment: $|V| = \sum_{i=1}^n |O_i|$. To compute the edges E , we need to check every pair of nodes $v_i, v_j \in V$ and see if it intersects with any of the $|V|$ obstacle edges. Thus, the graph can be computed in a total runtime of $O(|V|^3)$. A more sophisticated implementation (de Berg et al. 2000) reduces this runtime to $O(|V|^2 \log(|V|))$.

5.3 General-purpose planners via sampling-based roadmaps

Here we propose a general numerical method for motion planning. The method is based on computing a roadmap for the free configuration space Q_{free} via (1) sampling, (2) collision detection, and (3) a so-called connection rule. We have covered techniques for sampling and collision detection in Chapter 4.

A *connection rule* is an algorithm that decides when and how to (try to) compute a path connecting two nodes of a sampled configuration space. The following pseudocode contains a connection rule based on the notion of *neighborhood* of a point.

Once a roadmap is computed, and once a start and goal location are given, the motion planning problem (find a path from start to goal) is solved using the following method. This method is similar to the planning-via-decomposition+search algorithm in Section 2.2.4:

- 1: connect q_{start} and q_{goal} to the roadmap (i.e., compute a larger graph containing also start and goal locations)
- 2: run a graph search algorithm such as the breadth-first search algorithm (or Dijkstra's algorithm, that we present later in this chapter) to find a path from start to goal

An example sampling-based roadmap and path from start to goal is given in Figure 5.5.

The sampling-based roadmap computation algorithm

Input: number of sample points in roadmap $N \in \mathbb{N}$. Requires access to a sampling algorithm, collision detection algorithm, and a notion of neighborhood in Q

Output: a roadmap (V, E) for the free configuration space Q_{free}

```

1: initialize  $(V, E)$  to be the empty graph
   // compute a set of locations  $V$  in  $Q_{\text{free}}$ , via sampling & collision detection
2: while number of nodes in  $V$  is less than  $N$  :
3:     compute a new sample  $q$  in the configuration space  $Q$ 
4:     if the configuration  $q$  is collision-free :
5:         add  $q$  to  $V$ 
   // compute a set of paths  $E$  via the following connection rule
6: for each sampled location  $q$  in  $V$  :
7:     for all other sampled locations  $p$  in a neighborhood of  $q$  :
8:         if the path from  $q$  to  $p$  hits no obstacle :
9:             add the path  $q$  to  $p$  to  $E$ 
10: return  $(V, E)$ 

```

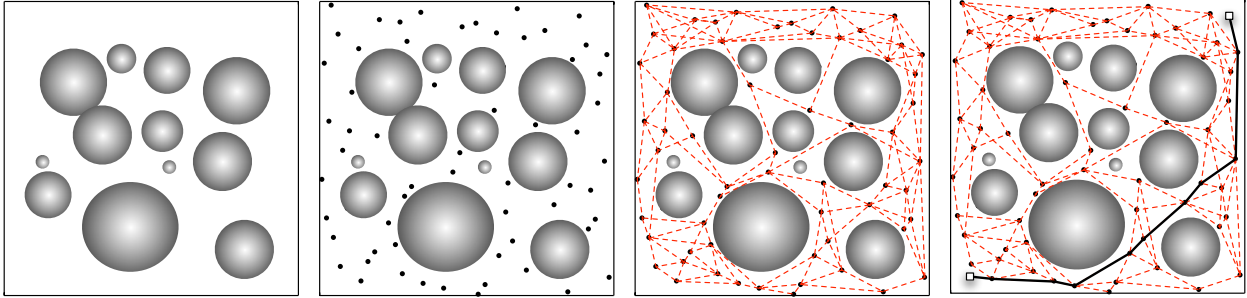


Figure 5.5: The free configuration space Q_{free} , a set of samples in Q_{free} (the first 100 samples of the Halton sequence with base numbers $(2, 3)$), a roadmap in Q_{free} and, finally, a path connecting a start location to a goal location.

5.3.1 Neighborhood functions

In deciding which sample configuration p to try to connect to q in step 7: of the previous algorithm, there are many possible choices. We briefly discuss three choices, each shown in Figure 5.6:

- (i) r -radius rule: fix a radius $r > 0$ and select all locations p within distance r of q ,
- (ii) K -closest rule: select the K closest locations p to q , and
- (iii) component-wise K -closest rule: from each connected component of the current roadmap, select the K closest locations p to q .

Note that the r -radius rule and the K -closest rules are not very different. If the points are sampled using a uniform grid, then radius rule and the K -closest rule are essentially the same: The number of neighbors within a radius r is the exact same for each sample point.

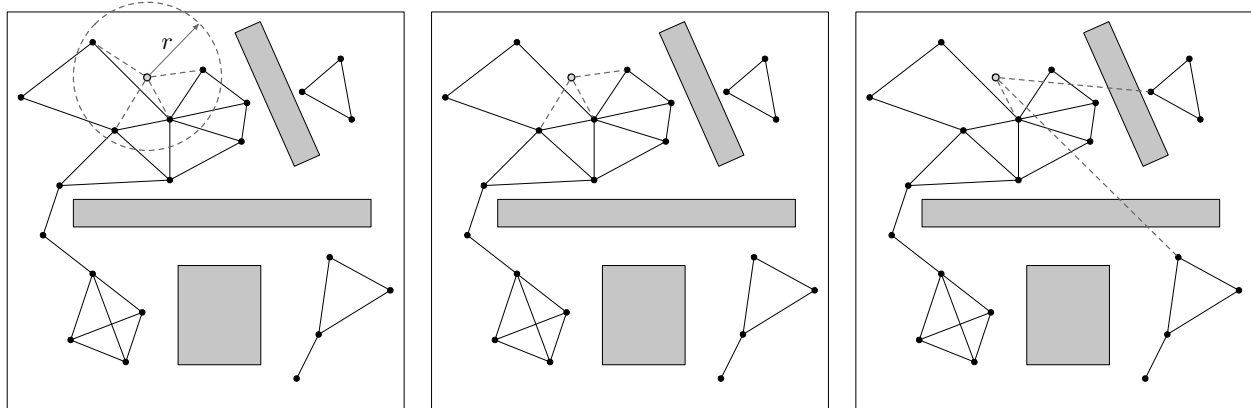


Figure 5.6: Three neighborhood functions. Left figure: r -radius rule. Middle figure: K -closest rule with $K = 3$. Right figure: component-wise K -closest rule with $K = 1$.

Note: in the r -radius rule, one strategy to design the radius r is to select it approximately of the same order as the dispersion of the sequence of sample points.

Distance functions

All neighborhood rules rely upon a notion of distance in the free configuration space Q_{free} . Distances between points in Euclidean spaces (e.g., the line, the plane, the 3-dimensional space) are easy to define and compute. How to properly define the distance between configurations in arbitrary spaces is more complex. Here are some observations and examples.

- Typically, the distance between points is defined as the length of the shortest path connecting two points.
- The exercises E3.1 and E3.2 discuss proper distances on the circle \mathbb{S}^1 and on the 2-torus \mathbb{T}^2 .
- One can also define shortest paths and distances on the 2-sphere \mathbb{S}^2 . A useful notion is that of great circle, i.e., the intersection of the sphere with a plane passing through the sphere center. The shortest path between two points on the 2-sphere is on the great circle through them and the corresponding distance is the angular distance on the great circle.
- For a roto-translating body with configuration space $\mathbb{R}^2 \times \mathbb{S}^1$, the definition of a distance function involves introducing a trade-off between translational and angular distances. A priori, it is not clear how to compare a translational with a rotational displacement.

Comparison between exact and approximate planners

Exact planners have the important feature of being automatically accessible, departable, and connected. The visibility graph is additionally efficient. Unfortunately, exact planners are only applicable to low-dimensional problems with free configuration spaces that are described by simple geometric shapes.

| | Exact Planners | Approximate Planners |
|-----------------------|--------------------|----------------------|
| accessible/departable | yes | not necessarily |
| connected | yes | not necessarily |
| efficient | yes for visibility | not guaranteed |
| applicability | narrow | broad |

Table 5.1: Comparison between exact planners (decomposition-based and visibility roadmaps) and approximate planners (sampling-based roadmaps)

Sampling-based approximate roadmaps have the key feature of being broadly and simply applicable to any problem. Roadmaps are not automatically guaranteed to be connected, efficient and easily accessible/departable. As the resolution N increases, these properties become more easily satisfied, at the cost of increased computational complexity. We summarize this comparison in Table 5.1.

Additional reading

A number of algorithmic improvements based on careful reasoning and numerical heuristics are discussed in Chapter 5 of ([LaValle 2006](#)).

5.4 Incremental sampling-based planning

In this section we adapt the sampling-based roadmap to single-query scenarios in which we are concerned only with computing a path between a single start and goal location, rather than building a re-usable roadmap of the configuration space.

5.4.1 Multiple-query and single-query scenarios

Roadmap-based methods are structured in general as a two-phase computation process consisting of

- (i) a preprocessing phase – given the free configuration space, compute the roadmap, followed by
- (ii) a query phase – given start and goal locations, connect them to the roadmap and search the resulting graph.

This structure is particularly well-suited for multiple-query problems, in which the same roadmap can be utilized multiple times to solve multiple motion planning problems in the same workspace. In this sense, we refer to roadmap-based planning methods as “multiple-query solvers.”

However, there are times when we only wish to make a single query, and thus we do not need to compute a reusable roadmap. In this case we can compute a special roadmap to solve the specific query. The roadmap is

- (i) computed directly as a function of the start location,
- (ii) is just a tree, as cycles do not add new paths from the start location.

Such tree-based roadmaps are “single-query solvers” and have some advantages in computation time when just a single query is needed.

Note: For symmetry reasons that we need not worry about here, one often computes two trees, one originating from the start location and one originating from the goal location.

5.4.2 Incremental tree-roadmap computation

The following is a general algorithm for constructing a single-query tree roadmap.

The incremental tree-roadmap computation method

Input: start location q_{start} , number of sample points in tree roadmap $N \in \mathbb{N}$. Also requires access to a sampling algorithm, collision detection algorithm, and a distance notion on Q

Output: a tree roadmap (V, E) for the free configuration space Q_{free} containing q_{start}

- 1: initialize (V, E) to contain the start location q_{start} and no edges
 - 2: **while** number of nodes in V is less than N :
 - 3: select a node $q_{\text{expansion}}$ from V for expansion
 - 4: choose a collision-free configuration q_{nearby} near $q_{\text{expansion}}$
 - 5: **if** can find a collision-free path from $q_{\text{expansion}}$ to q_{nearby} :
 - 6: add q_{nearby} to V
 - 7: add collision-free path from $q_{\text{expansion}}$ to q_{nearby} to E
 - 8: **return** (V, E)
-

We refer to step 3: as *node selection* and to step 4: as *node expansion*. An illustration of the steps 3: to 7: is given in Figure 5.7.

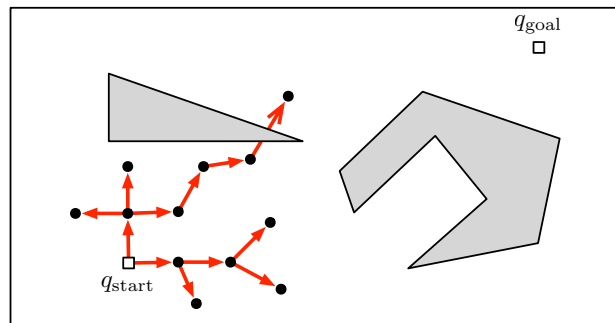


Figure 5.7: Example of incremental tree roadmap computation

The Rapidly-Exploring Random Tree (RRT) algorithm

Step 3: and 4: are not specified in the algorithm above. In what follows we give these two instructions according to a famous planning algorithm:

- 3: choose a random configuration q_{random} in Q and select for expansion the node $q_{\text{expansion}}$ from V that is closest to q_{random}
- 4: select a collision-free configuration q_{nearby} near $q_{\text{expansion}}$ by moving from $q_{\text{expansion}}$ towards q_{random}

Together these two choices push the tree roadmap to grow in random directions and to rapidly explore the free configuration space. In reality, the name “random” is unnecessary and it is completely possible to use deterministic Halton sequences to grow an incremental rapidly-exploring tree.

The following figures illustrate RRTs in an empty environment and in an environment with obstacles.

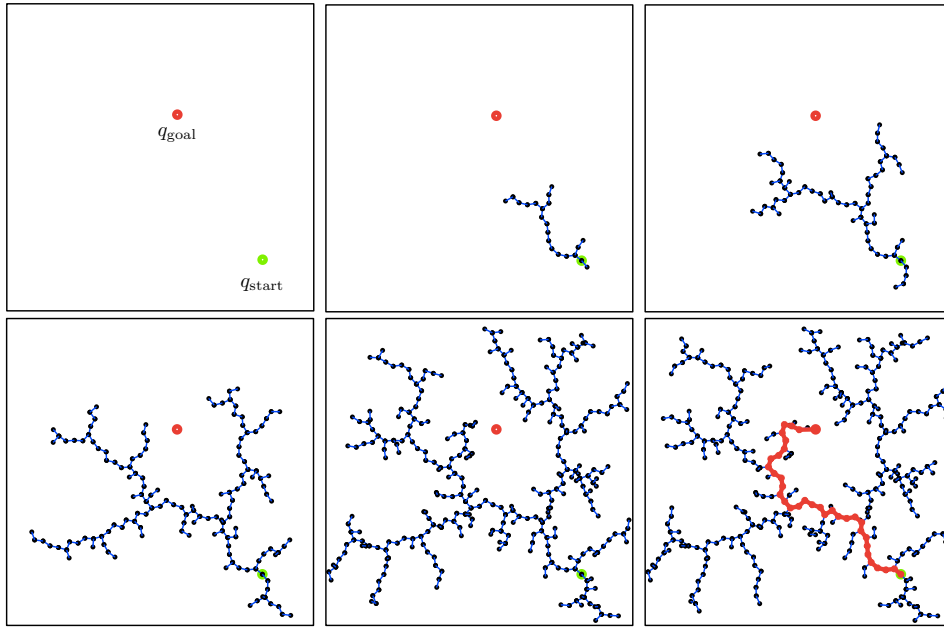


Figure 5.8: RRT in an empty environment

5.4.3 Application to sensor-based planning

We conclude our discussion of incremental planning methods with a brief discussion of their application to sensor-based planning problems.

Note: regarding instruction 1:, the problem of estimating a map of the world surrounding the robot and the robot position in the map is referred to as the Simultaneous Localization and Mapping problem (SLAM).

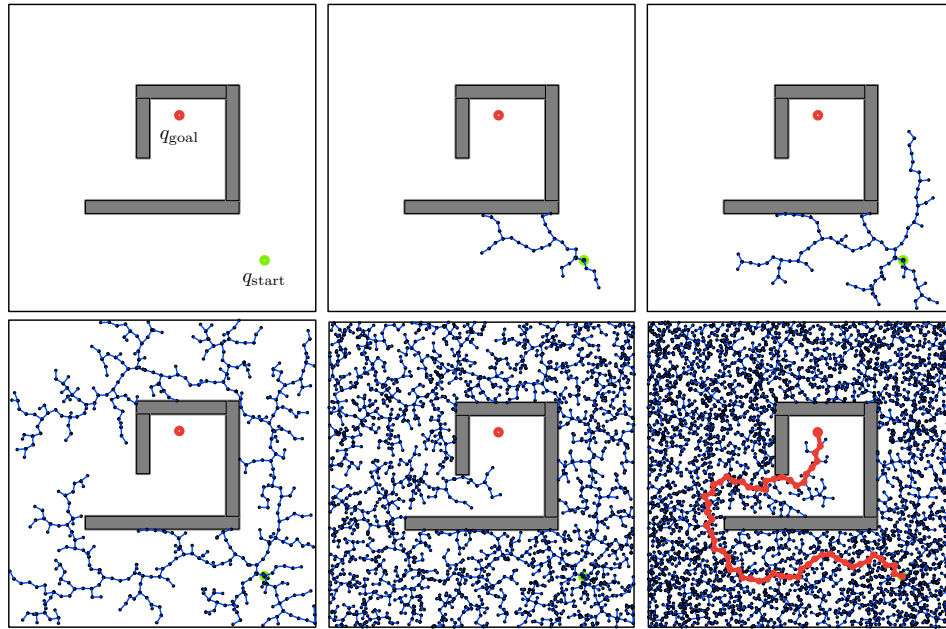


Figure 5.9: RRT in environment with obstacles

The receding-horizon sensor-based planning algorithm

- 1: maintain in memory an estimate of your location and of the workspace map around you
 - 2: **while** robot is not at destination :
 - 3: invoke an incremental (single-query) planner to compute a motion plan from current location towards the goal over a time horizon T
 - 4: move according to the motion plan for a fraction of the time horizon T
 - 5: while moving, sense the environment with all sensors and update your localization and mapping estimate
-

Note: regarding instruction 3:, the motion plan needs not be complete all the way to the goal location; it suffices instead that the robot will be at a location nearer the goal after the plan execution. Also, the plan could have high accuracy for the near future and low accuracy for the far future.

5.5 Appendix: Shortest paths in weighted graphs via Dijkstra's algorithm

A graph is *weighted* if a positive number, called the *weight*, is associated to each edge. In motion planning problems, the edge weight might be a distance between locations, a time required to travel or a cost associated with the travel. (In electrical networks the edge weight may be a resistance or impedance of the edge) The *weight of a path* is the sum of the weights of each edge in the path.

The *minimum-weight path between two nodes*, also called the *shortest path in a weighted graph*, is a path of minimum weight between the two nodes.

Consider the following example problem: how to compute the shortest paths from node n_1 to all other nodes in the following weighted graph.

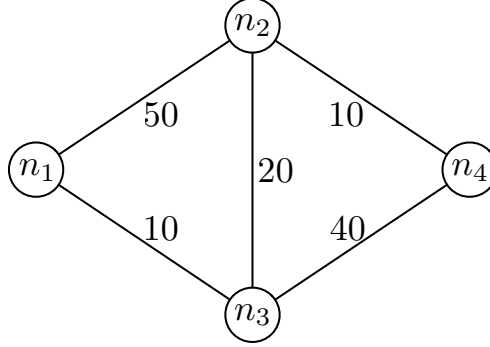


Figure 5.10: Example weighted graph

Let us start with some ideas.

- (i) A shortest path from n_1 to any other node cannot contain any cycle because each edge has positive weight. Additionally, the set of shortest paths from n_1 is a tree.
- (ii) Suppose you know the shortest path from A to B and suppose this path passes through C . Then the path from A to C is also optimal. (This is referred to as the Bellman Principle of Optimality.)
- (iii) We introduce a number dist associated to each node, describing the distance of each node from the start node n_1 . At the beginning, we initialize this distance to $+\infty$ for all nodes other than n_1 , and we initialize $\text{dist}(n_1) := 0$.

We are now ready to present Dijkstra's algorithm that, given a connected weighted graph G of order n and a node v , computes a tree $T_{\text{shortest-paths}}$ with all shortest-paths starting at v :

Informal description: For each node, maintain a weighted distance estimate from the source, denoted by dist . Incrementally construct a tree that contains only shortest paths from the source. Starting with an empty tree, at each round, add to the tree (1) the node outside the tree with smallest dist , and (2) the edge corresponding to the shortest path to this node. The estimates dist are updated as follows: when a node is added to the tree, the estimates of the neighboring outside nodes are updated (see details below). The tree is stored using parent pointers that for each node u record the node immediately before u on the shortest path from the source to u .

As in BFS the parent pointers define all edges in the shortest path tree as $\{\text{parent}(u), u\}$ for each node u for which $\text{parent}(u) \neq \text{NONE}$. Given a goal node v_{goal} we can use the parent

Dijkstra's algorithm

Input: a weighted graph G and a start node v_{start} **Output:** the parent pointer and dist values for each node in the graph G

```

    // Initialization of distance and parent pointer for each node
1: for each node  $v$  in  $G$  :
2:      $\text{dist}(v) = +\infty$ 
3:      $\text{parent}(v) = \text{NONE}$ 
4:  $\text{dist}(v_{\text{start}}) = 0$ 
5:  $Q = \text{set of all nodes in } G$ 
    // Main loop to grow the tree and update distance estimates
6: while  $Q$  is not empty :
7:     find node  $v$  in  $Q$  with smallest  $\text{dist}(v)$ 
8:     remove  $v$  from  $Q$ 
9:     for each node  $w$  connected to  $v$  by an edge :
10:        if  $\text{dist}(w) > \text{dist}(v) + \text{weight}(v, w)$  :
11:             $\text{dist}(w) = \text{dist}(v) + \text{weight}(v, w)$ 
12:             $\text{parent}(w) = v$ 
13: return parent pointers and  $\text{dist}$  values for all nodes  $v$ 

```

pointers to reconstruct the sequence of nodes on the shortest path from v_{start} to v_{goal} using the `extract-path` algorithm from Section 2.3.1.

Note that the output of this algorithm is not necessarily unique, since the choice of node at step 7: in the algorithm may not be unique.

Figure 5.2 shows an execution of the Dijkstra algorithm. When the goal node is known, there is a commonly-used improvement upon Dijkstra's algorithm called the A^* Algorithm. We invite the reader to read more about this simple heuristic improvement [Wikipedia:A* Algorithm](#).

Runtime of Dijkstra's algorithm Given a graph $G = (V, E)$ with n vertices and m edges, the runtime of Dijkstra's algorithm depends on the data structure used to store the distance of each vertex. There are three commonly-used options.

The first option is to store the distances in an array, where $V = \{1, \dots, n\}$ and the entry $\text{dist}[i]$ stores the distance to vertex i . In this case, step 7: requires $O(n)$ runtime each time it is executed. Since the while-loop runs n times, the runtime of Dijkstra's algorithm using an array is $O(n^2)$.

The second option is using a data structure called a *priority queue* in which the minimum element is stored at the top of a binary heap for easy access. In this data structure, the minimum element can be removed from Q in $O(\log n)$ time. However, changing the value of an entry of the queue at step 11: also requires $O(\log n)$ time per operation in order to properly maintain the priority queue. The resulting runtime of Dijkstra's algorithm using a priority queue is $O((n + m) \log n)$.

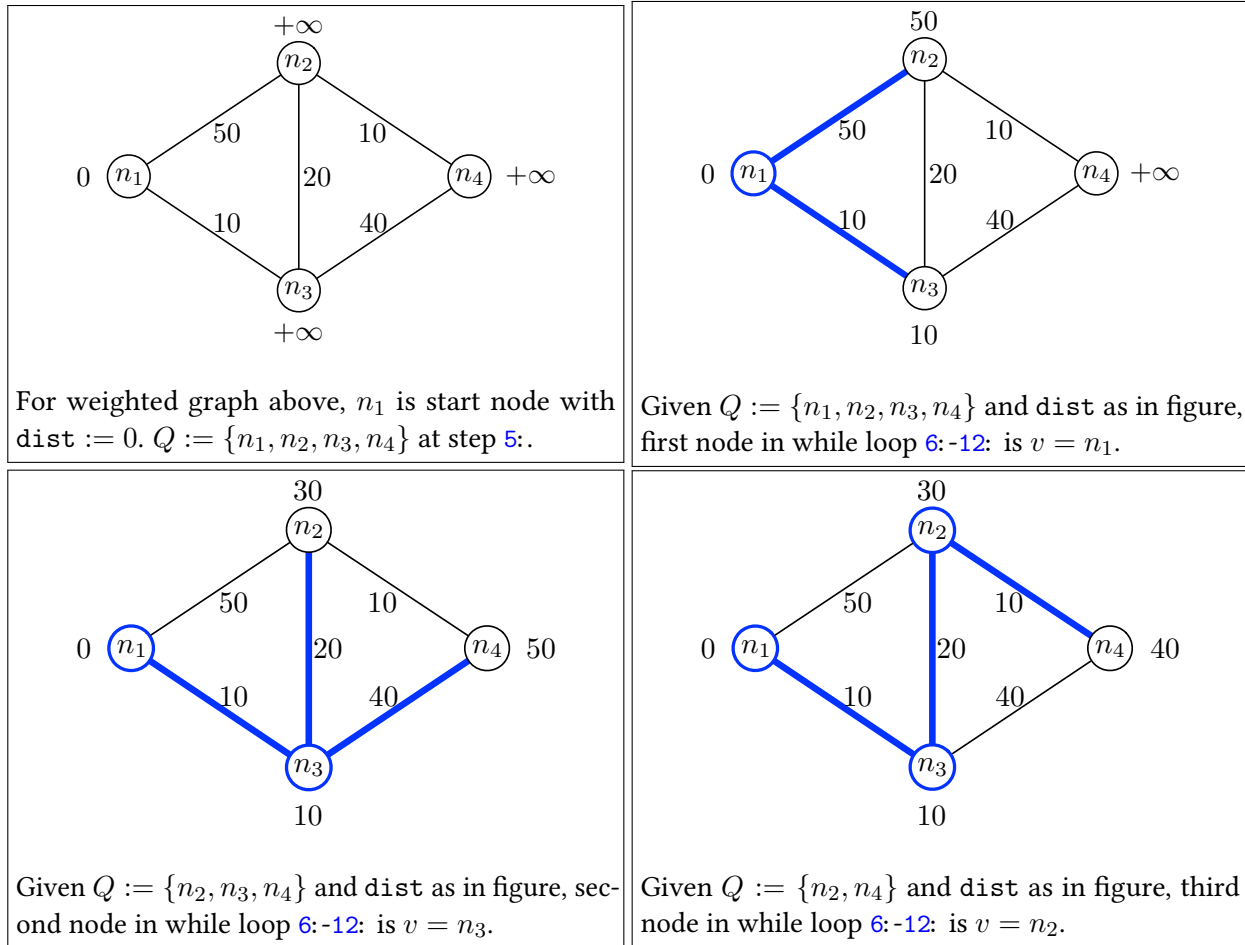


Table 5.2: Example execution of Dijkstra's algorithm. Near each node we plot the value of the corresponding distance estimate dist . Each node drawn in blue is already part of the tree containing all shortest paths.

Finally, there exists a more advanced data structure called a Fibonacci heap for which the runtime of Dijkstra's algorithm becomes $O(n \log n + m)$. However, due to the complexity of the structure, it is not commonly used in practice.

5.6 Exercises

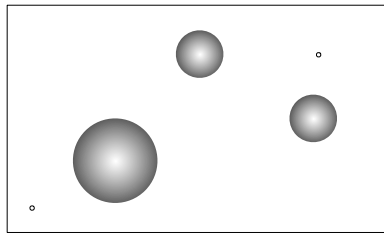
E5.1 **Visibility graph (25 points).** Consider a rectangular workspace with polygonal obstacles, where n is the total number of vertices of the obstacles.

- (i) (5 points) Give an upper bound on the number of edges in the visibility graph as a function of n .
- (ii) (10 points) What is the time complexity of finding the visibility graph in terms of n assuming that all obstacles are convex. Explain your reasoning.

Hint: Note that in order to find the visibility graph, one should check whether each edge intersects or is in the interior of an obstacle.

- (iii) (10 points) What is the time complexity of finding the visibility graph (as a function of n) when the environment can contain both convex and non-convex obstacles. Explain your reasoning.

E5.2 **Shortest paths in environments with circular obstacles (14 points).** Consider an environment with circular obstacles as in the figure below.



- (i) (10 points) What do shortest paths look like?
- (ii) (4 points) In such an environment, can you draw a roadmap that, like the visibility graph in environments with polygonal obstacles, contains all shortest paths?

E5.3 **Properties of roadmaps (16 points).**

- (i) (6 points) Explain the importance of each of the four properties required for a roadmap: accessibility, departability, connectivity, and efficiency.
- (ii) (10 points) In a paragraph, describe which other property of a roadmap you believe is important for path planning purposes.

Part II

Kinematics

Introduction to Kinematics and Rotation Matrices

This chapter begins our discussion of two interconnected topics of kinematics and rotation matrices.

Kinematics is study of motion, independent of the force causing it. In kinematics one is concerned with describing the motion of objects by relating positions to velocities, i.e., via differential equations of the first order. While it is simple to relate the position of a point to its linear velocities, relating the position and orientation of an 3D object to its linear and angular velocities is less obvious. Kinematics is not to be confused with dynamics, where Newton's law is adopted to explain how velocities change in time. We will not discuss dynamics in these notes. In kinematics we study how to define and manipulate:

- (i) body positions and orientations,
- (ii) reference frames and changes of coordinates, and
- (iii) rigid body motions and their composition.

Rotation matrices are a powerful modelling tool with multiple applications. First, the set of rotation matrices is the configuration space or part of the configuration space of any object that rotates; recall our discussion about configuration spaces in Chapter 3. Therefore, by understanding rotation matrices one can then use the motion planning algorithms in the first part of the course to plan the motion of three-dimensional objects such as aerial, space and underwater vehicles. Additionally, rotation matrices can be understood as transformations that rotate objects and play a central role in modeling reference frames in space. In short, rotation matrices are a fundamental technology in numerous disciplines in engineering and science, including for example:

- (i) robotics and aerospace engineering: orientation of robots, aircraft, underwater vehicles, satellites;

- (ii) medical sciences: modelling of eye movement and of the vestibular system, e.g., see (Allison et al. 1996);
- (iii) computer vision: orientation of cameras and objects in the scene; and
- (iv) video games and 3D virtual worlds, see (Carpin et al. 2007; Epic Games, Inc. 2009; Lewis and Jacobson 2002).

6.1 Geometric objects and their algebraic representation

The 3-dimensional Euclidean space contains various useful geometric objects:

- points, for example, the point p in Figure 6.1
- free vectors, that is, objects described by a length and a direction, but without a base point – we do not assume that the free vector is attached to any specific start point,
- vectors fixed at a point, that is, vectors with length direction and base points and, therefore, equivalent to an ordered pair of two points, and
- reference frames; a reference frame is a pair $\Sigma_0 = (O_0, \{x_0, y_0, z_0\})$, where O_0 is a point called the *origin* of the reference frame, and $\{x_0, y_0, z_0\}$ is a triplet of unit-length orthogonal vectors fixed at the origin and satisfying the right-hand rule. (recall that two free vectors are orthogonal when they form an angle of $\pi/2$).

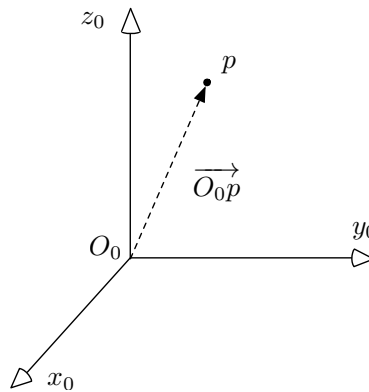


Figure 6.1: Basic geometric objects

6.1.1 Vector products between free vectors

Two different notions of products between free vectors are useful. In what follows v and w are two free vectors:

- (i) *the dot product between free vectors (or scalar product or inner product)* is defined as follows: the scalar quantity $v \cdot w$ is equal to (length of v) \times (length of w) \times (cosine of angle between v and w).
- (ii) *The cross product between free vectors (or outer product)* is defined as follows: the free vector $v \times w$ is equal to the free vector
- a) whose length equal to (length of v) \times (length of w) \times (sine of angle between v and w), and
 - b) whose direction is orthogonal to both u and v as given by the right-hand-rule.

Note: The dot product is a scalar number, whereas the cross product is another free vector.

Note: Both the dot product and the cross product are defined without any reference frame. Using these products, one can define orthogonal and parallel free vectors: Two free vectors are orthogonal if their dot product is zero. Two free vectors are parallel if their cross product is zero.

Note: An *axis* a unit-length vector. The dot product of a vector with an axis is the length of the projection of the vector onto the axis.

6.1.2 Representing vectors and points as arrays

In this and later chapters, it is important to distinguish between vectors and arrays of numbers:

- a free vector and a vector fixed at a point are geometric objects,
- an *array* is an ordered arrangements of numbers or quantities. An array has entries or elements, usually real numbers. There are *column arrays* and *row arrays*, with any number of entries (e.g., two entries or three entries). The transpose of a column array is a row array, of course. A matrix is a double array.

Given a reference frame, we will now review how a free vector can be written as an column array, whose entries are then called the *coordinates* of the free vector. Often one denotes free vectors and arrays with the same symbol, but you should be able to distinguish between them and understand what is meant by the context. By introducing coordinates, we study how to represent and manipulate various geometric objects (points, free vectors, frames, etc) via algebraic equations.

Let p be a point in 3-dimensional space, v be a free vector, and $\Sigma_0 = (O_0, \{x_0, y_0, z_0\})$ be a reference frame, as in Figure 6.1. Let $\vec{p_0p}$ from O_0 to p . The *coordinate representation* of v and p with respect to Σ_0 are

$$v^0 = \begin{bmatrix} v \cdot x_0 \\ v \cdot y_0 \\ v \cdot z_0 \end{bmatrix}, \quad p^0 = (\vec{O_0p})^0 = \begin{bmatrix} \vec{O_0p} \cdot x_0 \\ \vec{O_0p} \cdot y_0 \\ \vec{O_0p} \cdot z_0 \end{bmatrix}.$$

In other words, it is possible to decompose a vector or a point into its components along the three orthogonal axis as

$$v = (v \cdot x_0)x_0 + (v \cdot y_0)y_0 + (v \cdot z_0)z_0, \text{ and } \overrightarrow{O_0p} = (\overrightarrow{O_0p} \cdot x_0)x_0 + (\overrightarrow{O_0p} \cdot y_0)y_0 + (\overrightarrow{O_0p} \cdot z_0)z_0.$$

Given two vectors v and w with coordinates $v^0 = [v_1, v_2, v_3]^T$ and $w^0 = [w_1, w_2, w_3]^T$, it is possible (see Exercises E6.1 and E6.2) to verify that

$$v \cdot w = (v^0)^T w^0 = v_1 w_1 + v_2 w_2 + v_3 w_3,$$

$$(v \times w)^0 = [v_2 w_3 - v_3 w_2, \quad v_3 w_1 - v_1 w_3, \quad v_1 w_2 - v_2 w_1]^T.$$

Already now, interesting questions begin to arise. For example, how would you show that the dot product between two free vectors is independent of reference frame in which the two free vectors are expressed?

Remark 6.1 (Rotations and translations of basic objects). *Although the representations of points and free vectors are identical, remember that points may be translated and free vectors may be rotated. But, it makes no sense to rotate a point or to translate a free vector.*

Remark 6.2 (Geometric and algebraic viewpoints). *In summary, kinematic concepts can be understood using two equivalent languages: a geometric and algebraic viewpoint. The geometric viewpoint is described by:*

- *points and free vectors in space, as basic objects,*
- *properties independent of reference frames, and*
- *concepts and tools from Euclidean geometry.*

The algebraic viewpoint is described by:

- *arrays and matrices, as basic objects,*
- *computations in a specific reference frame, and*
- *concepts and tools from linear algebra and matrix theory.*

The two approaches are equivalent once a reference frame is selected. It is however very useful to understand both approaches and their equivalence. Typically, a geometric understanding is easy to attain and then the geometric intuition leads to the correct algebraic equations.

6.2 Geometric properties of rotations

Let us now begin to study rotations in some detail. For now, let us just understand geometrically what properties a rotation should have. We will later study how to represent rotations algebraically and how to manipulate them.

Geometric properties of rotations

- (i) Rotations are operations on free vectors; they preserve length of vectors and angles between vectors.
- (ii) In three-dimensional space, there are three “independent” basic rotations. In vehicle kinematics, three basic angles are roll, pitch and yaw, as illustrated in Figure 6.2 below.
- (iii) Rotations can be composed and the order of composition is essential.
- (iv) Each rotation admits an inverse rotation.
- (v) Each rotation is a rotation about a rotation axis of a rotation angle. Also, each rotation leaves its own rotation axis unchanged.

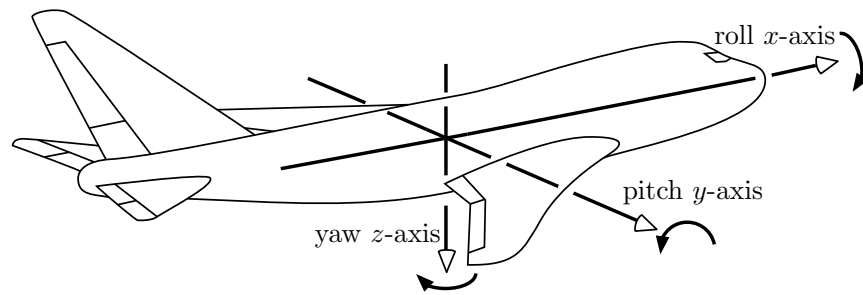


Figure 6.2: The roll-pitch-yaw conventions for aircraft. The yaw axis points downward towards the ground. The roll axis points forward. The pitch axis is selected so that positive pitch corresponds to the front of the aircraft pointing up.

Rotations do not commute, i.e., the order of composition is essential Let us now expand a little bit on the fourth property: the order of composition is essential. Imagine that we perform either one of the following two operations:

- (Composed rotation #1) = rotate about axis x_0 by $\pi/2$ and then about axis z_0 by $\pi/2$, or
- (Composed rotation #2) = rotate about axis z_0 by $\pi/2$ and then about axis x_0 by $\pi/2$.

As illustrated graphically in Figure 6.3, it is easy to see that the composite rotation #1 is *not equal* to the composite rotation #2.

Rotations may not be composed as though they were arrays Imagine we represent a rotation as an array of the form

$$\begin{bmatrix} \text{roll} \\ \text{pitch} \\ \text{yaw} \end{bmatrix}.$$

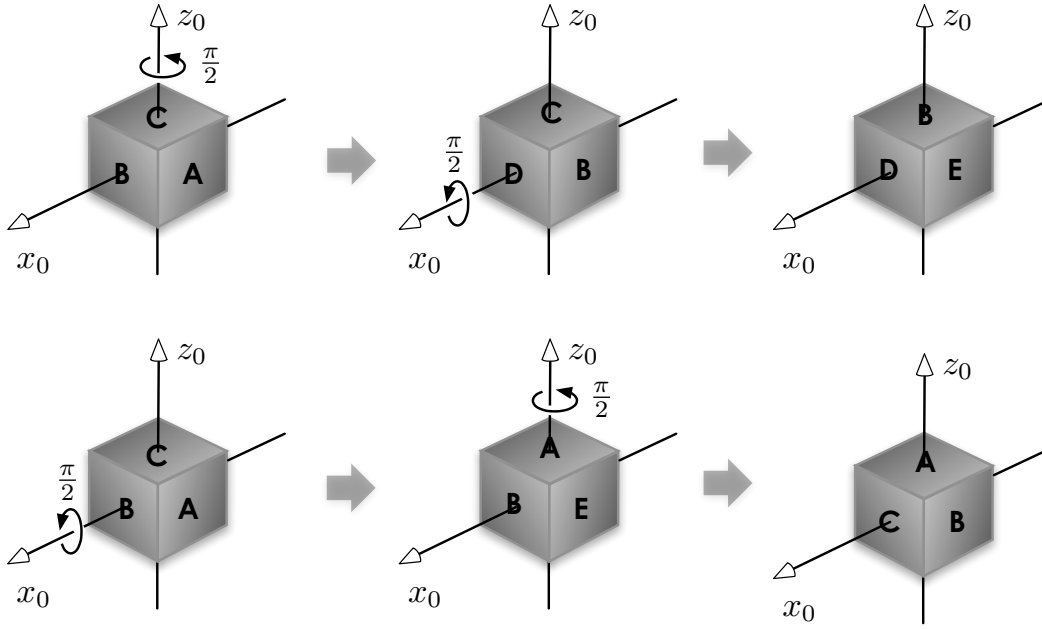


Figure 6.3: Rotations do not commute, i.e., the order of composition is essential.

Imagine now the sequence: +90 pitch, +90 roll, -90 pitch. By rotating a rigid body with one's own hands, readers should convince themselves that the composite rotation equals +90 yaw. However, if we write this fact as an equation between arrays, we find the following contradiction:

$$\begin{bmatrix} 0 \\ +90 \\ 0 \end{bmatrix} + \begin{bmatrix} +90 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ -90 \\ 0 \end{bmatrix} \neq \begin{bmatrix} 0 \\ 0 \\ +90 \end{bmatrix}.$$

The lesson of this simple calculation is as follows: if we adopt column arrays to represent rotations, we may not sum column arrays to compute the composite rotation, i.e., the composition of subsequent rotations. This first attempt at modeling rotations using column arrays is therefore unsuccessful.

6.3 Representing reference frames

Consider two reference frames $\Sigma_0 = (O_0, \{x_0, y_0\})$ and $\Sigma_1 = (O_1, \{x_1, y_1\})$ in the plane with the same origin $O_0 = O_1$. Because the frames share the same origin, Σ_1 is a rotated version of Σ_0 (one can imagine that the rotation is about the direction perpendicular to the figure, coming out of the page). Let us assume that Σ_1 is obtained by rotating Σ_0 counterclockwise by an angle θ . The array representation in the frame Σ_0 of the basis vector x_1 for the frame Σ_1 is

$$x_1^0 = \begin{bmatrix} x_1 \cdot x_0 \\ x_1 \cdot y_0 \end{bmatrix} = \begin{bmatrix} \cos \theta \\ \sin \theta \end{bmatrix}.$$

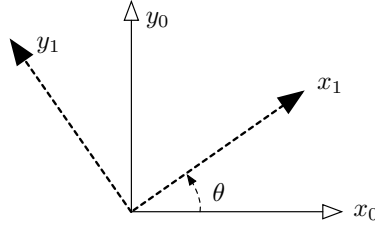


Figure 6.4: Two example reference frames with the same origin

Now, define a 2×2 *reference-frame rotation matrix* (representing Σ_1 with respect to Σ_0) by

$$R_1^0 = [x_1^0 \mid y_1^0] = \begin{bmatrix} x_1 \cdot x_0 & y_1 \cdot x_0 \\ x_1 \cdot y_0 & y_1 \cdot y_0 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}. \quad (6.1)$$

For example, if $\theta = \pi/3$, we easily compute $R_1^0 = \begin{bmatrix} 1/2 & -\sqrt{3}/2 \\ \sqrt{3}/2 & 1/2 \end{bmatrix}$. In summary, we can represent the frame Σ_1 with respect to the frame Σ_0 by either

- (i) $\theta \in [-\pi, \pi[= \mathbb{S}^1$, or
- (ii) R_1^0 a 2×2 matrix with special properties, that we will study in this chapter.

6.3.1 Three-dimensional reference frames

The fundamental advantage of the rotation matrix representation is that it naturally extends to problems in three dimensions. Indeed, let us write down 3-dimensional reference frames and understand 3-dimensional reference-frame rotation matrices. Given two reference frames

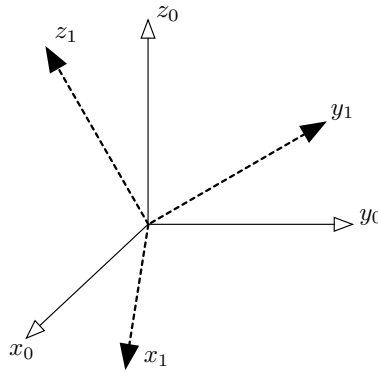


Figure 6.5: Two example reference frames with the same origin in three dimensions

$\Sigma_0 = (O_0, \{x_0, y_0, z_0\})$ and $\Sigma_1 = (O_1, \{x_1, y_1, z_1\})$ with the same origin $O_0 = O_1$ in 3-dimensions,

the reference-frame rotation matrix describing Σ_1 with respect to Σ_0 is defined by

$$R_1^0 = [x_1^0 \mid y_1^0 \mid z_1^0] = \begin{bmatrix} x_1 \cdot x_0 & y_1 \cdot x_0 & z_1 \cdot x_0 \\ x_1 \cdot y_0 & y_1 \cdot y_0 & z_1 \cdot y_0 \\ x_1 \cdot z_0 & y_1 \cdot z_0 & z_1 \cdot z_0 \end{bmatrix}. \quad (6.2)$$

The reference-frame rotation matrix is also sometimes referred to as the “direction cosine matrix,” the reason being that each entry is the dot product between unit-length direction vectors.

6.3.2 Changes of reference frame

Since we just studied how to represent Σ_1 as a function of Σ_0 , let us also define the reference-frame rotation matrix representing Σ_0 with respect to Σ_1 , denoted by R_0^1 . By looking at the entries in equation (6.2), one can immediately establish

$$R_0^1 = (R_1^0)^T.$$

We continue to consider two frames Σ_0 and Σ_1 with the same origin.

Lemma 6.3 (Coordinate transformation). *Given a point p and two reference frames Σ_0 and Σ_1 , the arrays p^0 and p^1 are related through*

$$p^0 = R_1^0 p^1.$$

Proof. First, write the geometric decomposition of vector $\overrightarrow{O_0 p}$ with respect to $\{x_1, y_1, z_1\}$:

$$\overrightarrow{O_0 p} = (\overrightarrow{O_0 p} \cdot x_1)x_1 + (\overrightarrow{O_0 p} \cdot y_1)y_1 + (\overrightarrow{O_0 p} \cdot z_1)z_1.$$

Second, express the left-hand side and the right-hand side with respect to Σ_0 :

$$\begin{aligned} p^0 = (\overrightarrow{O_0 p})^0 &= (\overrightarrow{O_0 p} \cdot x_1)x_1^0 + (\overrightarrow{O_0 p} \cdot y_1)y_1^0 + (\overrightarrow{O_0 p} \cdot z_1)z_1^0 \\ &= [x_1^0 \mid y_1^0 \mid z_1^0] \begin{bmatrix} \overrightarrow{O_0 p} \cdot x_1 \\ \overrightarrow{O_0 p} \cdot y_1 \\ \overrightarrow{O_0 p} \cdot z_1 \end{bmatrix} = R_1^0 p^1. \end{aligned}$$

■

Remark 6.4 (Subscript and superscript convention). *The role of subscript and superscripts in the equation $p^0 = R_1^0 p^1$ is important. In the right-hand side, the subscript matches the superscript. The convention is helpful in remembering correctly how to compose changes of reference frame.*

6.3.3 The three basic rotations

Consider two frames $\Sigma_0 = (O_0, \{x_0, y_0, z_0\})$ and $\Sigma_1 = (O_1, \{x_1, y_1, z_1\})$ with the same origin $O_0 = O_1$. If the two frames share the same z -axis, that is, if $z_0 = z_1$, then there must exist an angle θ such that

$$\begin{cases} x_1 = \cos(\theta)x_0 + \sin(\theta)y_0, \\ y_1 = -\sin(\theta)x_0 + \cos(\theta)y_0, \end{cases}$$

so that, since $z_1^0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$,

$$R_1^0 = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

We illustrate this setup in Figure 6.6. Therefore, this reference-frame rotation matrix is a *basic*

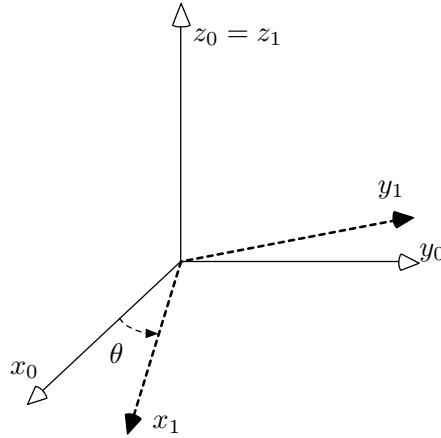


Figure 6.6: The frame Σ_1 obtained by rotating about z_0 by θ

rotation about the z -axis, that we denote by $\text{Rot}_z(\theta)$, that is, a counterclockwise rotation about the z -axis consistent with right-hand rule. Similar reasoning can be given for the other two basis vectors and, in summary, the three basic rotations about the three orthogonal axes are

$$\begin{aligned} \text{Rot}_x(\theta) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}, & \text{Rot}_y(\theta) &= \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}, \\ \text{Rot}_z(\theta) &= \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \end{aligned} \tag{6.3}$$

Every one of the three matrices satisfies the following algebraic properties:

$$\begin{aligned}\text{Rot}_z(0) &= I_3, \\ \text{Rot}_z(\theta) \cdot \text{Rot}_z(\phi) &= \text{Rot}_z(\theta + \phi), \\ (\text{Rot}_z(\theta))^{-1} &= (\text{Rot}_z(\theta))^T = \text{Rot}_z(-\theta),\end{aligned}$$

each of which has a nice geometric interpretation.

6.4 Appendix: A primer on matrix theory

In the next chapters we will rely upon basic knowledge of matrix theory. Matrix theory is a broad field and provides numerous useful tools with applications in most disciplines of engineering and science.

We briefly review here the concepts that we will use.

- (i) We consider arbitrary matrices with real entries $A \in \mathbb{R}^{n \times m}$. We denote the (i, j) entry of A by a_{ij} or equivalently by $(A)_{ij}$. A matrix $A \in \mathbb{R}^{n \times m}$ defines n row arrays with m entries, called the *rows* of A , and m columns arrays with n entries, called the *columns* of A .

If $A \in \mathbb{R}^{d \times d}$ we say A is a square matrix of dimension d . We let I_d denote the identity matrix in d dimensions, that is, $(I_d)_{ij} = 1$ if $i = j$, and $(I_d)_{ij} = 0$ otherwise.

- (ii) Given two matrices A and B with dimensions $n \times d$ and $d \times m$ respectively, the *matrix product* AB is a matrix of dimension $n \times m$ with entries

$$(AB)_{ij} = \sum_{k=1}^d a_{ik} b_{kj}, \quad \text{for } i \in \{1, \dots, n\}, \text{ and } j \in \{1, \dots, m\}.$$

In other words, the (i, j) entry of AB is the scalar product of the i th row of A with the j th column of B .

In matrix multiplication, the order of composition is essential. In other words, for arbitrary matrices A and B , square of the same dimensions, note that $AB \neq BA$.

- (iii) We regard column arrays as matrices. In other words, if B is a column vector with n entries, we write $B \in \mathbb{R}^{n \times 1}$. Therefore, for example, if $A \in \mathbb{R}^{2 \times 2}$ and $B \in \mathbb{R}^{2 \times 1}$, the matrix product is written as

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_{11}b_1 + a_{12}b_2 \\ a_{21}b_1 + a_{22}b_2 \end{bmatrix}.$$

- (iv) For a matrix $A \in \mathbb{R}^{n \times m}$, the *transpose* $A^T \in \mathbb{R}^{m \times n}$ is the matrix whose rows are the columns of A . For matrices A and B of appropriate dimensions, the transpose operation has the following property: $(AB)^T = B^T A^T$.
- (v) A matrix $A \in \mathbb{R}^{d \times d}$ is *invertible* if there exists a matrix, called its *inverse matrix* and denoted by $A^{-1} \in \mathbb{R}^{d \times d}$, with property that $AA^{-1} = A^{-1}A = I_d$.

- (vi) The *trace* of a square matrix A , denoted by $\text{trace}(A)$, is the sum of the diagonal entries of A , that is, $\text{trace}(A) = \sum_{k=1}^d a_{kk}$. For square matrices A and B of the same dimension, the trace operator has the following property:

$$\text{trace}(AB) = \text{trace}(BA).$$

Therefore, if B is invertible, one also has $\text{trace}(B^{-1}AB) = \text{trace}(A)$.

- (vii) The array $v \in \mathbb{R}^d$ and the number λ are an *eigenvector* and *eigenvalue* for the square matrix A if $Av = \lambda v$.
- (viii) A matrix $A \in \mathbb{R}^{n \times m}$ can be written as a *block matrix* of the form

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad (6.4)$$

where $A_{11} \in \mathbb{R}^{a \times c}$, $A_{12} \in \mathbb{R}^{a \times d}$, $A_{21} \in \mathbb{R}^{b \times c}$, and $A_{22} \in \mathbb{R}^{b \times d}$ are each matrices such that $a + b = n$ and $c + d = m$. For example, the matrix

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 8 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix},$$

can be written in block form (6.4) by defining four 2×2 matrices:

$$A_{11} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix}, \quad A_{12} = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix}, \quad A_{21} = \begin{bmatrix} 8 & 9 \\ 13 & 14 \end{bmatrix}, \quad A_{22} = \begin{bmatrix} 11 & 12 \\ 15 & 16 \end{bmatrix}.$$

Multiplication of block matrices is performed using the matrix product of the blocks:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}.$$

Determinants of square matrices We assume the reader recalls the definition of *determinant* of a square matrix in arbitrary dimension. We review here the definition for matrices in two and three dimensions. In two dimensions, the determinant of a matrix is equal to (plus or minus) the area of the parallelogram whose sides are the two columns of the matrix; see Figure 6.7 where c_1 and c_2 are the columns. Specifically,

$$\det [c_1 \quad c_2] = \det \begin{bmatrix} a & c \\ b & d \end{bmatrix} = ad - cb.$$

Note that the determinant can be negative, whereas the area of the parallelogram is always positive.

In three dimensions, the determinant of a matrix is equal to (plus or minus) the volume of a parallelepiped whose sides are the three columns of the matrix; see Figure 6.7. Specifically, if c_1, c_2, c_3 are the three columns of the matrix, we have

$$\det [c_1 \ c_2 \ c_3] = (c_1 \times c_2) \cdot c_3. \quad (6.5)$$

The last product in equation is called the *triple product* of the three column arrays c_1, c_2, c_3 in \mathbb{R}^3 .

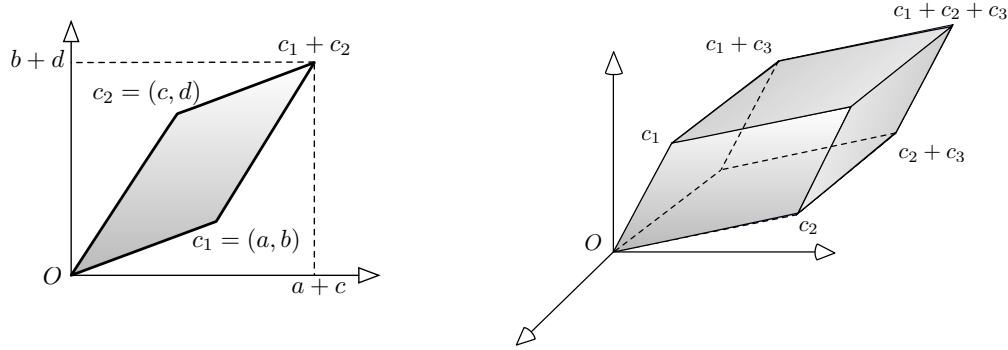


Figure 6.7: The determinant of a 2x2 (resp. 3x3) matrix is related to the area (resp. volume) of an appropriate parallelogram (resp. parallelepiped).

We conclude with some useful properties of the determinant operator.

Lemma 6.5 (Properties of the determinant). *Let A and B be square matrices of the same dimension.*

- (i) A is invertible if and only if $\det(A) \neq 0$,
- (ii) $\det(A^T) = \det(A)$, and
- (iii) $\det(AB) = \det(A) \det(B)$.

6.5 Appendix: The theory of groups

Definition 6.6 (Group). *A group is a set G along with a binary operation \star that takes two elements $a, b \in G$ and generates a new element $a \star b \in G$, satisfying the following properties:*

- (i) $a \star (b \star c) = (a \star b) \star c$ for all $a, b, c \in G$ (associativity),
- (ii) there exists an identity $e \in G$ such that $a \star e = e \star a = a$ for all $a \in G$, and
- (iii) there exists an inverse $a^{-1} \in G$ such that $a \star a^{-1} = a^{-1} \star a = e$ for all $a \in G$.

We write the group as (G, \star) in order to emphasize the operation with respect to which the group is defined.

Here are some examples of groups:

- the integer numbers with the operation of sum,
- the real positive numbers with the operation of product,
- the *set of rotations* in n dimensions, with the operation of “rotation composition.” This set is usually referred to as the group $SO(n)$,
- the *set of rotations and translations*, we will see later how to describe it via matrices, and
- the set of permutations of $\{1, \dots, m\}$, sometimes referred to as the permutation group.

The last example, the permutation group, contains a finite number of elements; all other example sets contain an infinite number of elements.

6.6 Exercises

- E6.1 **Free vectors and arrays (20 points).** In this exercise, we distinguish between free vectors (as in geometric vectors in space) and arrays of numbers (as column lists containing entries). Let us review these two concepts.

A *free vector* in 2-dimensional or 3-dimensional space is a geometric object with a magnitude (or length) and a direction. We do not assume that the free vector is attached to any specific start point.

An *array* is an ordered arrangement of numbers or quantities. An array has entries or elements. There are *column arrays* and *row arrays*, with any number of entries (e.g., two entries or three entries). The transpose of a column array is a row array, of course.

If there is a reference frame in Euclidean space, then a free vector can be written as an column array and the entries in the array are called the coordinates of the vector. Because of this equivalence, it is convenient to denote free vectors and arrays with the same symbol.

For the following questions, assume v and w are free vectors in 2-dimensional space and, simultaneously, column arrays with two entries.

- (i) (5 points) What is the formula definition for $v \cdot w$, the dot product between geometric vectors?
- (ii) (5 points) What is the formula definition for $v^T w$, the matrix product between arrays?
- (iii) (10 points) Using your formulas from (i) and (ii), prove the following equivalence:

$$v \cdot w = v^T w.$$

- E6.2 **Properties of the triple product (20 points).** Given two vectors v and w with coordinates $v^0 = [v_1, v_2, v_3]^T$ and $w^0 = [w_1, w_2, w_3]^T$, show that

$$(v \times w)^0 = [v_2 w_3 - v_3 w_2, \quad v_3 w_1 - v_1 w_3, \quad v_1 w_2 - v_2 w_1]^T.$$

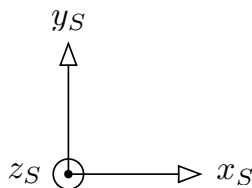
Given three vectors r_1 , r_2 , and r_3 in \mathbb{R}^3 , show that

$$(r_1 \times r_2) \cdot r_3 = (r_3 \times r_1) \cdot r_2 = (r_2 \times r_3) \cdot r_1.$$

Explain why the triple product vanishes whenever any two vectors are parallel.

- E6.3 **Drawing reference frames (20 points).** The frame $\Sigma_S = (O_S, \{x_S, y_S, z_S\})$ is drawn below (a circle surrounding the origin means that the z_S vector is pointing out of the page). Copy the frame in your answer and sketch the frames Σ_M and Σ_N determined by

$$R_M^S = \begin{pmatrix} \frac{1}{2} & \frac{-\sqrt{3}}{2} & 0 \\ \frac{\sqrt{6}}{4} & \frac{\sqrt{2}}{4} & \frac{-\sqrt{2}}{2} \\ \frac{\sqrt{6}}{4} & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{2} \end{pmatrix}, \quad \text{and} \quad R_N^S = \begin{pmatrix} \frac{1}{2} & \frac{-\sqrt{3}}{2} & 0 \\ \frac{\sqrt{6}}{4} & \frac{\sqrt{2}}{4} & \frac{-\sqrt{2}}{2} \\ \frac{\sqrt{6}}{4} & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{2} \end{pmatrix}.$$



- E6.4 **Basic rotations and their products (15 points).** Consider the basic rotations about the three main axes, defined in equation (6.3).

- (i) (5 points) Compute by hand (i.e., do the algebra) the following products:

$$R_1 = \text{Rot}_x(\pi/4) \text{Rot}_y(\pi/3),$$

$$R_2 = \text{Rot}_y(\pi/3) \text{Rot}_x(\pi/4),$$

$$R_3 = \text{Rot}_z(-\pi/2) \text{Rot}_x(\pi/2) \text{Rot}_z(\pi/2).$$

- (ii) (5 points) What is the reason why R_1 is not equal to R_2 ?
(iii) (5 points) Express R_3 as a single basic rotation.

Hint: Convince yourself of your answers to questions (ii) and (iii) as follows: select x , y , and z axes for a rigid body (e.g., a book or your bike) and perform the corresponding rotations.

Rotation Matrices

In this chapter we study the set of rotation matrices and all their properties, including how to compose them and how to parametrize them. Standard references rotation matrices in robotics include (Craig 2003; Mason 2001; Murray et al. 1994; Siciliano et al. 2009; Spong et al. 2006).

7.1 From reference frames to general rotation matrices

We continue our study of the properties of reference-frame rotation matrices introduced in Section 6.3. First, we note two characterizing properties of each reference-frame rotation matrix. We then define rotation matrices independently of reference frame, by just requiring them to satisfy the two characterizing properties.

7.1.1 The two characterizing properties

We present two important properties of reference-frame rotation matrices. We will later show that each matrix that satisfies these properties must represent a rotation.

Lemma 7.1 (Orthonormal columns and rows). *Let R_1^0 be the reference-frame rotation matrix (in either dimension $n \in \{2, 3\}$) representing a frame Σ_1 with respect to another frame Σ_0 . The following facts are true and equivalent:*

- (i) *the columns of R_1^0 form a complete set of orthonormal arrays, that is, orthogonal and unit length, and*
- (ii) $(R_1^0)^T R_1^0 = I_n$.

Proof. By definition of reference frame Σ_1 , the vectors $\{x_1, y_1, z_1\}$ are orthonormal. Therefore, also the three arrays $\{x_1^0, y_1^0, z_1^0\}$ have the properties that $(x_1^0)^T x_1^0 = (y_1^0)^T y_1^0 = (z_1^0)^T z_1^0 = 1$ and $(x_1^0)^T y_1^0 = (y_1^0)^T x_1^0 = (x_1^0)^T z_1^0 = 0$ – because the value of the dot product is independent upon the reference frame. This proves statement (i).

Regarding statement (ii), for $R_1^0 = \begin{bmatrix} x_1^0 & y_1^0 & z_1^0 \end{bmatrix}$, we write

$$(R_1^0)^T = \begin{bmatrix} (x_1^0)^T \\ (y_1^0)^T \\ (z_1^0)^T \end{bmatrix}$$

and compute

$$(R_1^0)^T R_1^0 = \begin{bmatrix} (x_1^0)^T \\ (y_1^0)^T \\ (z_1^0)^T \end{bmatrix} \begin{bmatrix} x_1^0 & y_1^0 & z_1^0 \end{bmatrix} = \begin{bmatrix} (x_1^0)^T x_1^0 & (x_1^0)^T y_1^0 & (x_1^0)^T z_1^0 \\ (y_1^0)^T x_1^0 & (y_1^0)^T y_1^0 & (y_1^0)^T z_1^0 \\ (z_1^0)^T x_1^0 & (z_1^0)^T y_1^0 & (z_1^0)^T z_1^0 \end{bmatrix} = I_3,$$

where we have used the definition of matrix product and property (i). ■

Note: it is true that $R^T R = I_n$ if and only if $R R^T = I_n$. Therefore, it suffices to verify that the columns of R are orthogonal to imply that also its rows are orthogonal.

Lemma 7.2 (Positive determinant). *In dimension $n = 2$ and $n = 3$, we have $\det(R_1^0) = +1$.*

Proof. As we reviewed in equation (6.5) in the appendix to the previous chapter, the determinant of a 3×3 matrix is given by the triple product among its columns:

$$\det R_1^0 = \det \begin{bmatrix} x_1^0 & y_1^0 & z_1^0 \end{bmatrix} = (x_1^0 \times y_1^0) \cdot z_1^0.$$

But we know $x_1^0 \times y_1^0 = z_1^0$ precisely because the reference frame Σ_1 is right-handed and we know $z_1^0 \cdot z_1^0 = +1$. ■

7.1.2 The set of rotation matrices

We are now finally ready to define rotation matrices independently of reference frames.

For $n \in \{2, 3\}$, we define *the set of rotation matrices* by

$$\text{SO}(n) = \{R \in \mathbb{R}^{n \times n} \mid R^T R = I_n \text{ and } \det(R) = +1\}.$$

Here, the abbreviation SO stands for “special orthonormal”, where the word “special” is a reflection of the determinant being positive. The set $\text{SO}(n)$ has some basic properties

(P0) Closedness with respect to matrix product: if R_1 and R_2 belong to $\text{SO}(n)$, then $R_1 R_2$ belongs $\text{SO}(n)$,

(P1) Associativity: if R_1 , R_2 and R_3 belong to $\text{SO}(n)$, then $R_1(R_2 R_3) = (R_1 R_2) R_3$,

(P2) Zero rotation: the “zero rotation” is the identity matrix I_n with the property that $R I_n = I_n R = R$,

(P3) Inverse rotation: for any rotation R , the inverse rotation always exists and is equal to R^T , that is, $R^T R = R R^T = I_n$.

In the Appendix 6.5 later in this chapter we discuss how these four properties turn the set $\text{SO}(n)$ into a so-called group. The proof of these properties is left as Exercise E7.5.

7.1.3 Three roles and uses of rotation matrices

Finally, it is useful to emphasize and clarify the three roles that a rotation matrix can play.

- (i) The reference-frame rotation matrix R_1^0 describes the reference frame Σ_1 with respect to Σ_0 . Specifically, the columns of R_1^0 are the basis vectors of Σ_1 with respect to Σ_0 .
- (ii) The reference-frame rotation matrix R_1^0 is the coordinate transformation from Σ_1 to Σ_0 , as characterized by the equation $p^0 = R_1^0 p^1$.
- (iii) Any rotation matrix $R \in \text{SO}(n)$ can be used to rotate a vector in \mathbb{R}^n . Given a point p , define a new point q as follows: the vector $\overrightarrow{O_0 q}$ is the rotation of the vector $\overrightarrow{O_0 p}$ by an angle θ , that is,

$$q^0 = R p^0.$$

This operation is illustrated in Figure 7.1, where R is of the form

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

for a planar problem.

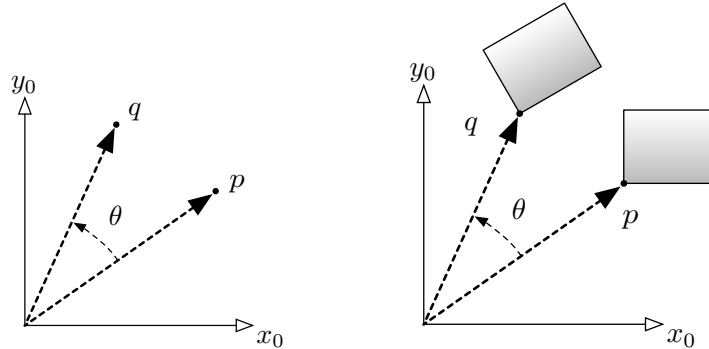


Figure 7.1: Rotating a point and a rigid body about the origin

7.1.4 Geometric properties of rotations

We are now able to verify that our proposed model of rotation matrices enjoy the geometric properties we initially discussed in Section 6.2.

- (i) “Rotations are operations on free vectors; they preserve length of vectors and angles between vectors.” Given an array v , the array Rv is the rotated version. Saying that length of vectors and angles between vectors are preserved is equivalent to the following equalities:

$$v^T w = (Rv)^T (Rw), \quad \text{and} \quad \|v\| = \|Rv\|.$$

We leave the proof of these two equations to Exercise E7.1.

- (ii) “In three-dimensional space, there are three independent basic rotations. In vehicle kinematics, three basic angles are roll, pitch and yaw.” This fact is reflected by the definition in equation (6.3) of the three basic rotations $\text{Rot}_x(\alpha)$, $\text{Rot}_y(\beta)$, and $\text{Rot}_z(\gamma)$. In Section 7.3, we will explain how to express any rotation matrix as a product of appropriate basic rotations.
- (iii) “Rotations can be composed and the order of composition is essential.” The composition of rotations occurs via matrix multiplication: if R_1 and R_2 are rotation matrices, then $R_1 R_2$ and $R_2 R_1$ are rotation matrices. Indeed, matrix multiplication is not commutative in general and, in particular, rotations do not commute, that is,

$$R_1 R_2 \neq R_2 R_1.$$

For example, it is easy to verify that $\text{Rot}_x(\pi/2) \text{Rot}_y(\pi/2) \neq \text{Rot}_y(\pi/2) \text{Rot}_x(\pi/2)$:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \neq \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ -1 & 0 & 0 \end{bmatrix}.$$

We study the composition of rotations in the next section.

- (iv) “Each rotation admits an inverse rotation.” The inverse rotation matrix of R is R^T .
- (v) “Each rotation is a rotation about a rotation axis of a rotation angle.” We postpone a discussion of this property to Section 7.3.3 below.

7.2 Composition of rotations

Let us consider the problem of how to compose rotations. It is easy to see that, to compose rotations, we multiply the corresponding rotation matrices. There are however two problems:

- (i) the order of multiplication matters, and so it is not clear in which order the rotation matrices should be multiplied, and
- (ii) the information “first rotate the body by $R_{[1]}$ and then by $R_{[2]}$ ” is ambiguous in the sense that it is not clear with respect to which frame the second rotation $R_{[2]}$ is expressed.

For example, looking at Figure 7.2, Consider a body with initial reference frame Σ_0 . Assume Σ_0 is a *fixed or inertial reference frame* and rotate the body about the y_0 pitch axis. Because Σ_0 is fixed, we introduce the new body-fixed reference frame Σ_1 , satisfying $y_1 = y_0$ but $z_1 \neq z_0$ and $x_1 \neq x_0$. We call Σ_1 the *successive or body reference frame*. If you are now asked to rotate about the z axis, it is necessary to know whether that is the z_0 or the z_1 axis, i.e., whether the second rotation is expressed in the fixed/inertial frame or in the successive/body frame.

Thus, if we consider the statement

“rotate about y by $\pi/6$, then rotate about z by $\pi/3$,”

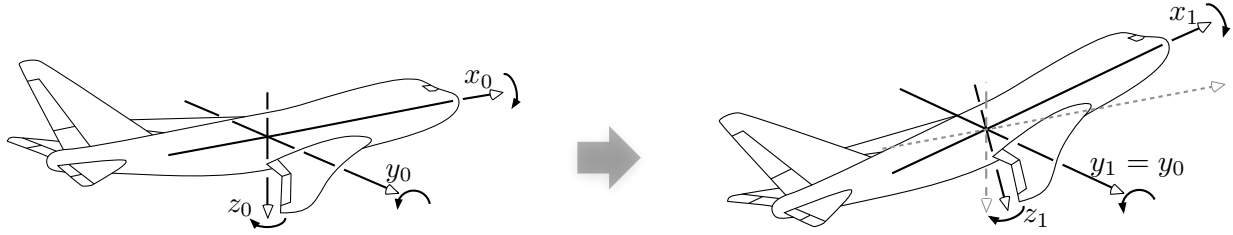


Figure 7.2: An initial fixed reference frame Σ_0 , and the new body-fixed frame Σ_1 obtained by rotating about y_0

we can use the formulas for the basic rotations to compute the two rotation matrices. But, is the composite rotation $\text{Rot}_y(\pi/6) \text{Rot}_z(\pi/3)$ or $\text{Rot}_z(\pi/3) \text{Rot}_y(\pi/6)$?

Problem 7.3 (Composition of rotations). *Given a rigid body and two rotations, represented by the matrices $R_{[1]}$ and $R_{[2]}$, suppose we first rotate the body by $R_{[1]}$ and then by $R_{[2]}$, what information do we need and how do we compute the resulting rotation?*

The following theorem will provide us with the answer to this question.

Theorem 7.4 ("Post-multiplication if successive frame" and "pre-multiplication if fixed frame"). *The composite rotation is*

$$R_{\text{composite}} = \begin{cases} R_{[1]}R_{[2]}, & \text{if } R_{[2]} \text{ is expressed in successive/body frame,} \\ R_{[2]}R_{[1]}, & \text{if } R_{[2]} \text{ is expressed in fixed/inertial frame.} \end{cases}$$

Hence, here is the solution to our example: if the statement is "rotate about y_0 by $\pi/6$, then rotate about z_1 by $\pi/3$," then the composite rotation is $\text{Rot}_y(\pi/6) \text{Rot}_z(\pi/3)$, and instead if the statement is "rotate about y_0 by $\pi/6$, then rotate about z_0 by $\pi/3$," then the composite rotation is $\text{Rot}_z(\pi/3) \text{Rot}_y(\pi/6)$.

To rigorously understand this theorem, we study two problems of independent interest:

- (i) given three reference frames and given the reference-frame rotation matrices R_1^0 and R_2^1 , what is the reference-frame rotation matrix R_2^0 ?
- (ii) given a rotation expressed as a rotation matrix R in reference frame Σ_0 , what is its representation with respect to a second reference frame Σ_1 ?

Problem #1: Composing reference-frame changes *Given three frames, frame-changes matrices R_1^0 and R_2^1 , what is the frame-change matrix R_2^0 ?*

By definition of reference frame rotation matrices, we write

$$p^0 = R_1^0 p^1, \quad p^1 = R_2^1 p^2, \quad \text{and } p^0 = R_2^0 p^2.$$

Composing the first and second equality, we obtain

$$p^0 = R_1^0 p^1 = R_1^0 R_2^1 p^2,$$

and, in turn, we obtain the desired rule:

$$R_2^0 = R_1^0 R_2^1.$$

From these calculations we learn two lessons:

(i) Subscript and superscripts play a critical role, e.g., see the correct relationships:

a) $p^0 = R_1^0 p^1,$

b) $R_2^0 = R_1^0 R_2^1.$

(ii) When the second rotation is expressed with respect to first rotation (which is the case here because we used the matrix R_2^1), then the second rotation is equivalent to a so-called “rotation expressed in a successive frame” and the final composite rotation is obtained by “post-multiplying or right-multiplying the second rotation.”

Problem #2: Representing a rotation in a different frame Given rotation S expressed in frame Σ_0 , what is its representation with respect to Σ_1 ?

Let us write down the facts we know:

- $p^0 = R_1^0 p^1,$
- rotation in Σ_0 is S , hence rotated point $q^0 = S p^0$, and
- $q^1 = R_0^1 q^0.$

Putting these facts together, we calculate

$$q^1 = R_0^1 q^0 = R_0^1 S p^0 = R_0^1 S R_1^0 p^1 = \left((R_1^0)^T S R_1^0 \right) p^1.$$

Hence, the same rotation has two different expressions as a rotation matrix when expressed in different frames:

$$\underbrace{S}_{\text{rotation in frame } \Sigma_0} \longrightarrow \underbrace{(R_1^0)^T S R_1^0}_{\text{rotation in frame } \Sigma_1}.$$

Here is the geometric interpretation of this result. Given that S is the rotation expressed with respect to Σ_0 , you rotate a point p expressed as p^1 with respect to Σ_1 by doing three multiplications: first, express p with respect to Σ_0 as $R_1^0 p^1$; second, rotate the point $R_1^0 p^1$ to obtain $S R_1^0 p^1$, and third, express the resulting point back with respect to Σ^1 as $(R_1^0)^T S R_1^0 p^1$.

Example 7.5. Given Σ_0 , define Σ_1 by $x_1 = y_0$, $y_1 = z_0$, and $z_1 = x_0$. (Verify that Σ_1 is right handed. Note that, because none of the basis axis is unchanged, Σ_1 is not a rotation about a basic axis of Σ_0). Now, consider a rotation S about the vertical axis z_0 which is expressed as $S = \text{Rot}_z(\theta)$

in the Σ_0 frame. The rotation S and the two frames are depicted in Figure 7.3. We now write R_1^0 and show $(R_1^0)^T \text{Rot}_z(\theta) R_1^0 = \text{Rot}_y(\theta)$:

$$\begin{aligned}
 R_1^0 &= \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \\
 (R_1^0)^T \text{Rot}_z(\theta) R_1^0 &= \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^T \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \\
 &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} -\sin(\theta) & 0 & \cos(\theta) \\ \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} = \text{Rot}_y(\theta).
 \end{aligned}$$

This answer makes sense because rotating about the z -axis in the Σ_0 frame should be the same as rotating about the y -axis in the Σ_1 frame — since we defined $y_1 = z_0$.

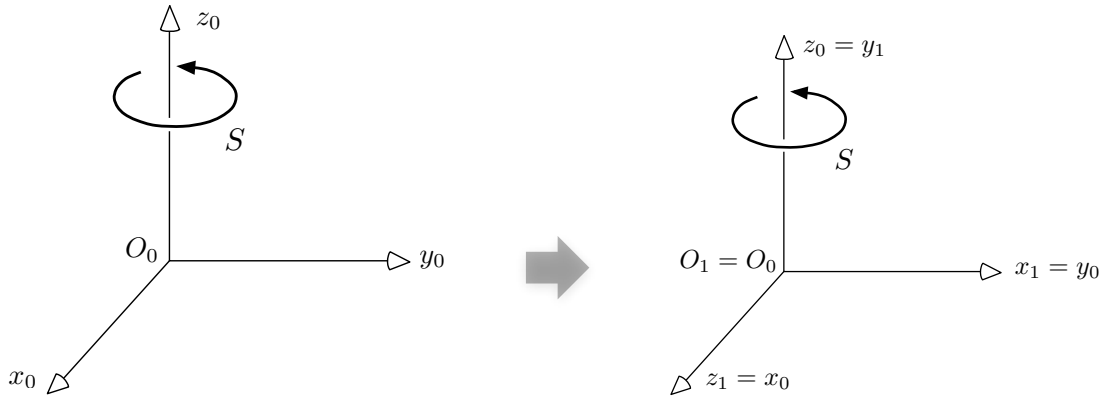


Figure 7.3: A rotation about the axis z_0 and a change of reference frame

Proof of Theorem 7.4 We are now ready to understand Theorem 7.4. We assume the frame Σ_0 is rotated by $R_{[1]}$ into a frame Σ_1 so that $R_{[1]} = R_1^0$.

First, if the rotation matrix $R_{[2]}$ is expressed in the successive/body frame, then we are in a situation analogous to Problem #1: $R_{[1]}$ rotates Σ_0 onto Σ_1 , and $R_{[2]}$ is expressed with respect to Σ_1 and represents a new frame Σ_2 . In this case the composite rotation is $R_1^0 R_2^1 = R_{[1]} R_{[2]}$.

Second, if the rotation matrix $R_{[2]}$ is expressed in the fixed/inertial frame, then we need to change the reference frame it is expressed in. In other words, the second rotation is $R_{[2]}$ when expressed with respect to Σ_0 and we wish to express it with respect to Σ_1 . In this case, the second rotation expressed with respect to Σ_1 is $(R_1^0)^T R_{[2]} R_1^0$. Hence, the composite rotation is

$$R_1^0 \cdot \left((R_1^0)^T R_{[2]} R_1^0 \right) = R_{[2]} R_1^0 = R_{[2]} R_{[1]}.$$

7.3 Parametrization of rotation matrices

We start with a result on the dimension of the set of special orthogonal matrices, i.e., on the degrees of freedom of a rotation.

Theorem 7.6 (Euler rotation theorem). *Any rotation matrix can be described by 3 parameters.*

Proof. A rotation matrix R has 9 entries, but each column has unit length (3 constraints) and the three columns are perpendicular with each other (3 additional constraints). Hence, the degrees of freedom of matrix R are equal to $9 - 3 - 3 = 3$. ■

There are numerous ways to represent R . In what follows we discuss (1) Euler angles, (2) roll-pitch-yaw angles, and (3) the axis-angle parametrization.

7.3.1 Euler angles

The rotation R is represented by three angles $\{\alpha, \beta, \gamma\}$ corresponding to basic rotations about Z-Y-Z axis with respect to successive frames. With $c_\beta = \cos(\beta)$, $s_\beta = \sin(\beta)$:

$$R = \text{Rot}_z(\alpha) \text{Rot}_y(\beta) \text{Rot}_z(\gamma) = \begin{bmatrix} c_\alpha c_\beta c_\gamma - s_\alpha s_\gamma & -c_\alpha c_\beta s_\gamma - s_\alpha c_\gamma & c_\alpha s_\beta \\ s_\alpha c_\beta c_\gamma + c_\alpha s_\gamma & -s_\alpha c_\beta s_\gamma + c_\alpha c_\gamma & s_\alpha s_\beta \\ -s_\beta c_\gamma & s_\beta s_\gamma & c_\beta \end{bmatrix}. \quad (7.1)$$

Equation (7.1) is illustrated in Figure 7.4 with the following convention:

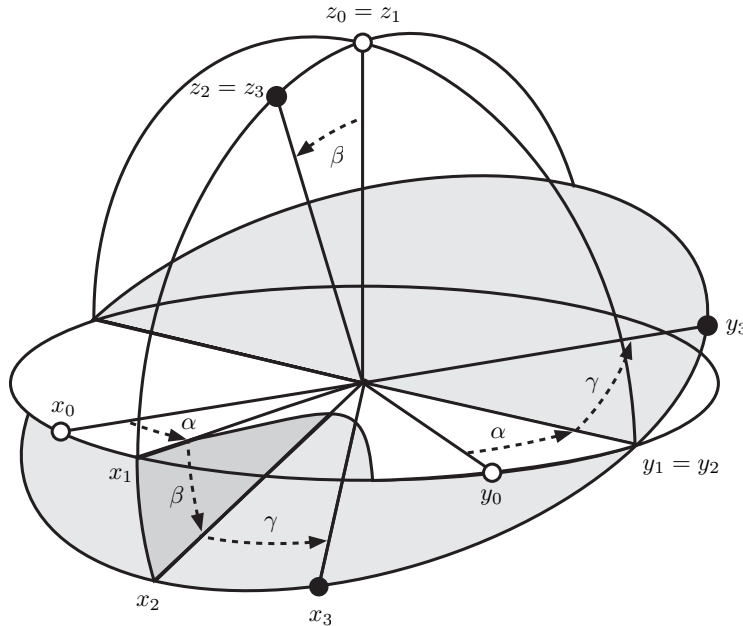


Figure 7.4: The Euler angles defined by the Z-Y-Z convention. Image courtesy of <http://www.easyspin.org>.

- (i) rotate $\{x_0, y_0, z_0\}$ about the z_0 axis by α to obtain $\{x_1, y_1, z_1\}$,
- (ii) rotate $\{x_1, y_1, z_1\}$ about the y_1 axis by β to obtain $\{x_2, y_2, z_2\}$, and
- (iii) rotate $\{x_2, y_2, z_2\}$ about the z_2 axis by γ to obtain the final $\{x_3, y_3, z_3\}$.

Inverse kinematics for Euler angles Equation (7.1) states how to compute a rotation matrix from three Euler angles. Next, we consider the following problem: *Given a matrix*

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix},$$

compute the Euler angles $\{\alpha, \beta, \gamma\}$.

To solve this problem, we carefully consider the 9 scalar equations induced by matrix equation

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = \begin{bmatrix} c_\alpha c_\beta c_\gamma - s_\alpha s_\gamma & -c_\alpha c_\beta s_\gamma - s_\alpha c_\gamma & c_\alpha s_\beta \\ s_\alpha c_\beta c_\gamma + c_\alpha s_\gamma & -s_\alpha c_\beta s_\gamma + c_\alpha c_\gamma & s_\alpha s_\beta \\ -s_\beta c_\gamma & s_\beta s_\gamma & c_\beta \end{bmatrix}.$$

From the $(3, 3)$ entry we single out the equality $\cos(\beta) = r_{33}$. Recall that we discussed this problem and its solution in equation (3.7) via the four-quadrant arctangent function atan_2 Appendix 3.5. Specifically, we computed two solutions, one with $\sin \beta > 0$ and one with $\sin \beta < 0$. Once β is known, it is easy to use again the atan_2 function to compute α from the entries r_{13} and r_{23} and γ from the entries r_{31} and r_{32} . These arguments lead to the following theorem and algorithm.

Theorem 7.7 (Existence and non-uniqueness of Euler angles). *Consider a rotation matrix R with components r_{ij} , $i, j \in \{1, 2, 3\}$ and the equation (7.1) in the variables $\{\alpha, \beta, \gamma\}$:*

- (i) *if $-1 < r_{33} < 1$, then there are two sets of Euler angles $\{\alpha, \beta, \gamma\}$ solving equation (7.1).*
- (ii) *if $r_{33} = \pm 1$, then equation (7.1) admits infinite solutions $\{\alpha, \beta, \gamma\}$.*

Moreover, the following algorithm computes the Euler angles of any rotation matrix.

Example 7.8. In an example in Section 7.2, we considered the frame Σ_1 defined by $x_1 = y_0$, $y_1 = z_0$, and $z_1 = x_0$. (One can verify that Σ_1 is right handed and that Σ_1 is not a rotation about a basic axis of Σ_0). The reference frame rotation matrix is

$$R_1^0 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Euler angles algorithm

Input: a rotation matrix R with components r_{ij} , $i, j \in \{1, 2, 3\}$

Output: Euler angles (α, β, γ) for rotation matrix R , i.e., solutions to (7.1)

- 1: **if** $-1 < r_{33} < 1$:
 - 2: $\beta_1 = \text{atan}_2(\sqrt{1 - r_{33}^2}, r_{33})$, $\alpha_1 = \text{atan}_2(r_{23}, r_{13})$ and $\gamma_1 = \text{atan}_2(r_{32}, -r_{31})$
// $(\alpha_1, \beta_1, \gamma_1)$ is solution with positive $\sin(\beta)$
 - 3: $\beta_2 = \text{atan}_2(-\sqrt{1 - r_{33}^2}, r_{33})$, $\alpha_2 = \text{atan}_2(-r_{23}, -r_{13})$, and $\gamma_2 = \text{atan}_2(-r_{32}, r_{31})$
// $(\alpha_2, \beta_2, \gamma_2)$ is solution with negative $\sin(\beta)$
 - 4: **return** two solutions $(\alpha_1, \beta_1, \gamma_1)$ and $(\alpha_2, \beta_2, \gamma_2)$
 - 5: **else if** $r_{33} = 1$:
 - 6: **return** any $(\alpha, 0, \gamma)$ such that $\alpha + \gamma = \text{atan}_2(r_{21}, r_{11})$
 - 7: **else if** $r_{33} = -1$:
 - 8: **return** any (α, π, γ) such that $\alpha - \gamma = \text{atan}_2(-r_{21}, -r_{11})$
-

We wish to compute the Euler angles for this rotation matrix. The Euler angles algorithm leads to the two following solutions:

$$\begin{aligned}\beta_1 &= \text{atan}_2(1, 0) = \pi/2, \\ \alpha_1 &= \text{atan}_2(0, 1) = 0, \\ \gamma_1 &= \text{atan}_2(1, 0) = \pi/2,\end{aligned}$$

and

$$\begin{aligned}\beta_2 &= \text{atan}_2(-1, 0) = -\pi/2, \\ \alpha_2 &= \text{atan}_2(0, -1) = \pi, \\ \gamma_2 &= \text{atan}_2(-1, 0) = -\pi/2.\end{aligned}$$

From the first set of Euler angles, we know and we indeed verify that

$$R_1^0 = \text{Rot}_y(\pi/2) \text{Rot}_z(\pi/2) \iff \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

From the second set of Euler angles, we know and we indeed verify that

$$\begin{aligned}R_1^0 &= \text{Rot}_z(\pi) \text{Rot}_y(-\pi/2) \text{Rot}_z(-\pi/2) \\ \iff \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} &= \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.\end{aligned}$$

7.3.2 Roll-pitch-yaw angles

Next we consider a second parametrization. The rotation R is represented by three angles $\{\alpha, \beta, \gamma\}$ corresponding to basic rotations about X-Y-Z axis with respect to fixed frame:

$$R = \text{Rot}_z(\gamma) \text{Rot}_y(\beta) \text{Rot}_x(\alpha).$$

These angles are illustrated in Figure 7.5. The remaining analysis is similar to that for the Euler angles. Assume a rotating frame is attached to a vehicle. The axes are set as follows: the x -axis

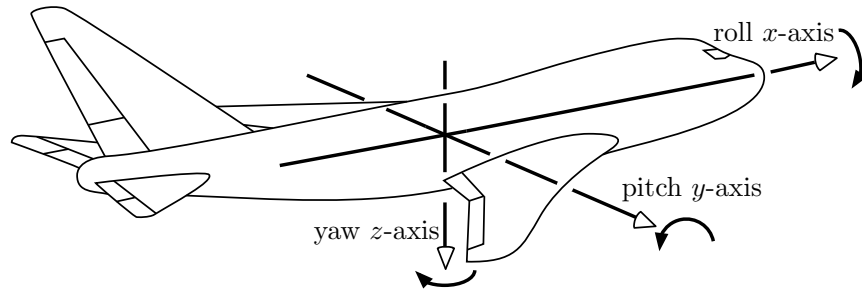


Figure 7.5: The roll-pitch-yaw conventions for aircraft. The yaw axis points downward towards the ground. The roll axis points forward. The pitch axis is selected so that positive pitch corresponds to the front of the aircraft pointing up.

points forward (roll axis in a boat) and the z -axis points downwards (towards earth in an aircraft). Then

- (i) α about the x -axis is the *roll* angle,
- (ii) β about the y -axis is the *pitch* angle, and
- (iii) γ about the z -axis is the *yaw* angle.

We do not discuss here the inverse kinematics for the roll-pitch-yaw angles as we did earlier for the Euler angles. Instead we refer the reader to Exercise E7.4.

7.3.3 Axis-angle parametrization

In this subsection we discuss the axis-angle parametrization. We answer two questions: (i) what is the rotation matrix corresponding to a given rotation axis and angle? and (ii) what are the rotation axis and angle of a given rotation matrix? This second question is well posed because it is true that each rotation is necessarily a rotation about an axis.

Preliminaries: skew symmetric matrices We begin with a necessary detour into the world of three-dimensional skew symmetric matrices.

Definition 7.9 (Skew symmetric matrices). *A square matrix is skew symmetric when it is equal to minus its transpose. The set of three-dimensional skew symmetric matrices has a specific symbol:*

$$\mathfrak{so}(3) = \{S \in \mathbb{R}^{3 \times 3} \mid S^T = -S\}.$$

In skew symmetric matrices the diagonal entries are always zero (why?) and the lower diagonal elements are uniquely determined by the upper diagonal elements. In other words, given an incomplete matrix

$$\begin{bmatrix} \star & a & b \\ \star & \star & c \\ \star & \star & \star \end{bmatrix},$$

there is a unique way of completing it into a skew matrix:

$$\begin{bmatrix} 0 & a & b \\ -a & 0 & c \\ -b & -c & 0 \end{bmatrix}.$$

In other words, 3×3 skew matrices have 3 degree of freedom. Next, we set up an equivalence between $\mathfrak{so}(3)$ and \mathbb{R}^3 . Given three real numbers ω_1, ω_2 , and ω_3 , we define the operation $\widehat{}$ from \mathbb{R}^3 to $\mathfrak{so}(3)$ and its inverse $^\vee$ by

$$\widehat{\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix}} = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix}, \quad \begin{bmatrix} 0 & s_{12} & s_{13} \\ -s_{12} & 0 & s_{23} \\ -s_{13} & -s_{23} & 0 \end{bmatrix}^\vee = \begin{bmatrix} -s_{23} \\ s_{13} \\ -s_{12} \end{bmatrix}. \quad (7.2)$$

A few additional properties of skew symmetric matrices are given in Appendix 7.4.

The axis-angle parametrization and its inverse

We are now ready to consider the following problem: *Given a unit-length rotation axis and angle, what is the corresponding rotation matrix?*

Theorem 7.10 (Rodrigues' formula). *Given a unit-length rotation axis n and angle $\theta \in [0, \pi]$, there exists a unique rotation matrix, denoted by $\text{Rot}_n(\theta)$, representing a rotation about n by an angle θ and it is given by*

$$\text{Rot}_n(\theta) = I_3 + \sin \theta \widehat{n} + (1 - \cos \theta) \widehat{n}^2.$$

It is a simple exercise to check that the matrix $\text{Rot}_n(\theta)$ as defined in the equation (7.10) is indeed a rotation matrix in $\text{SO}(3)$. We postpone the proof of this formula to the Appendix 7.5.

Example 7.11. *For example, consider $\theta = \frac{2\pi}{3}$ and*

$$n = \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Note easily that $\sin(\theta) = \sqrt{3}/2$, $\cos(\theta) = -1/2$,

$$\hat{n} = \frac{1}{\sqrt{3}} \begin{bmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{bmatrix}, \quad \text{and} \quad \hat{n}^2 = \frac{1}{3} \begin{bmatrix} -2 & 1 & 1 \\ 1 & -2 & 1 \\ 1 & 1 & -2 \end{bmatrix}.$$

Therefore, we compute

$$\text{Rot}_n(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \frac{\sqrt{3}}{2} \frac{1}{\sqrt{3}} \begin{bmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{bmatrix} + \frac{3}{2} \frac{1}{3} \begin{bmatrix} -2 & 1 & 1 \\ 1 & -2 & 1 \\ 1 & 1 & -2 \end{bmatrix}.$$

Simplifying the expression leads to

$$\text{Rot}_n(\theta) = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Next, we are also interested in the inverse problem. Given a rotation matrix R , what is the rotation angle θ and the corresponding axis of rotation n ?

Theorem 7.12 (Inverse Rodrigues' formula). *Given an arbitrary rotation matrix R , consider the Rodrigues formula (7.10) in the two variables (θ, n) , where θ is the angle of rotation in $[0, \pi]$ and n is the unit-length axis of rotation:*

- (i) *if $R = I_3$, then there exists an infinite number of solutions defined by $\theta = 0$ and arbitrary axis of rotation n ,*
- (ii) *if $3 > \text{trace}(R) > -1$, then there exists a unique solution defined by*

$$\theta = \arccos((\text{trace}(R) - 1)/2),$$

$$n = \frac{1}{2 \sin(\theta)} (R - R^T)^\vee,$$

- (iii) *if $\text{trace}(R) = -1$, then there exist two solutions defined by $\theta = \pi$ and by the two solutions $n_1 = -n_2$ to $nn^T = \frac{1}{2}(R + I_3)$.*

Example 7.13. As before, we consider the frame Σ_1 defined by $x_1 = y_0$, $y_1 = z_0$, and $z_1 = x_0$, with reference frame rotation matrix:

$$R_1^0 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

We now wish to compute the rotation axis and angle for this rotation matrix. Using the Inverse Rodrigues' formula and equation (7.2), we obtain:

$$\theta = \arccos((\text{trace}(R) - 1)/2) = \arccos(-1/2) = \frac{2\pi}{3},$$

$$n = \frac{1}{2 \sin(\theta)} (R - R^T)^\vee = \frac{1}{2\sqrt{3}/2} \begin{bmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{bmatrix}^\vee = \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

7.4 Appendix: Properties of skew symmetric matrices

Given a vector $\omega = [\omega_1 \ \omega_2 \ \omega_3]^T \in \mathbb{R}^3$ and the corresponding skew symmetric matrix $\hat{\omega}$ is defined as

$$\hat{\omega} = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix}.$$

The following are some key properties of skew symmetric matrices:

- (i) The map from $\omega \in \mathbb{R}^3$ to $\hat{\omega} \in \text{so}(3)$ is linear. That is, given $u, v \in \mathbb{R}^3$ and $\alpha, \beta \in \mathbb{R}$ we have

$$\widehat{(\alpha u + \beta v)} = \alpha \hat{u} + \beta \hat{v}.$$

- (ii) For any vectors $u, v \in \mathbb{R}^3$, we have

$$\hat{u}v = u \times v,$$

where $u \times v$ is the *cross product* of the vectors u and v .

- (iii) If $R \in \text{SO}(3)$ is a rotation matrix and $u \in \mathbb{R}^3$ is a vector, then

$$R\hat{u}R^T = \widehat{Ru}.$$

- (iv) For any vector $u \in \mathbb{R}^3$ and skew symmetric matrix $S \in \text{so}(3)$, we have $u^T Su = 0$.

These four properties can be established as follows:

- (i) Linearity follows directly from the definition of a skew symmetric matrix:

$$\begin{aligned} \widehat{\alpha u + \beta v} &= \begin{bmatrix} 0 & -\alpha u_3 - \beta v_3 & \alpha u_2 + \beta v_2 \\ \alpha u_3 + \beta v_3 & 0 & -\alpha u_1 - \beta v_1 \\ -\alpha u_2 - \beta v_2 & \alpha u_1 + \beta v_1 & 0 \end{bmatrix} \\ &= \alpha \begin{bmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{bmatrix} + \beta \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix} = \alpha \hat{u} + \beta \hat{v}. \end{aligned}$$

(ii) The cross product again follows by expanding out the operation

$$\widehat{uv} = \begin{bmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} u_2v_3 - u_3v_2 \\ u_3v_1 - u_1v_3 \\ u_1v_2 - u_2v_1 \end{bmatrix} = u \times v.$$

(iii) Given a rotation matrix R and any two vectors $u, v \in \mathbb{R}^3$, notice that

$$R(u \times v) = (Ru) \times (Rv). \quad (7.3)$$

That is, rotating the cross product of u and v is equivalent to rotating u and v , and then taking the cross product. Thus, we have

$$\begin{aligned} R\widehat{u}R^Tv &= R(u \times R^Tv) && \text{by property (ii)} \\ &= (Ru) \times (RR^Tv) && \text{by equation (7.3)} \\ &= Ru \times v && \text{since } RR^T = I_3 \\ &= \widehat{Ru}v. \end{aligned}$$

From this we conclude that $R\widehat{u}R^T = \widehat{Ru}$.

(iv) The final property follows from the cross product property as $u^T S u = u^T (S^\vee \times u) = 0$ since the vector $(S^\vee \times u)$ is orthogonal to both S^\vee and u .

7.5 Appendix: Proof of Rodrigues' formula and of its inverse

Proof of Theorem 7.10. The following proof is one of the more complex derivations in the lecture notes.

Next, we consider the problem of rotating a vector about an axis. Given a point p and an origin point O , we want to rotate the vector \overrightarrow{Op} about the rotation axis n by an angle θ to generate a new point q . By rotation axis n we accept any unit-length vector. Our objective is to compute the point q as a function of p , or more specifically, the vector q^0 as a rotation of p^0 . By doing so, we aim to compute the rotation matrix $\text{Rot}_n(\theta)$ that describes a rotation about the axis n by an angle θ . We illustrate the problem and some relevant variables in the Figure 7.6.

First, we decompose the vector p^0 into its component along n and perpendicular to n :

$$p^0 = p_{\parallel}^0 + p_{\perp}^0 = (n \cdot p^0)n - n \times (n \times p^0).$$

From the figure we can write a formula for q :

$$\begin{aligned} q^0 &= p_{\parallel}^0 + \text{Rot}_n(\theta)p_{\perp}^0 \\ &= \left(p^0 - p_{\perp}^0\right) + \left(\sin(\theta)(n \times p^0) + \cos(\theta)(-1)(n \times (n \times p^0))\right) \\ &= p^0 + \sin(\theta)n \times p^0 + (1 - \cos \theta)n \times (n \times p^0). \end{aligned}$$

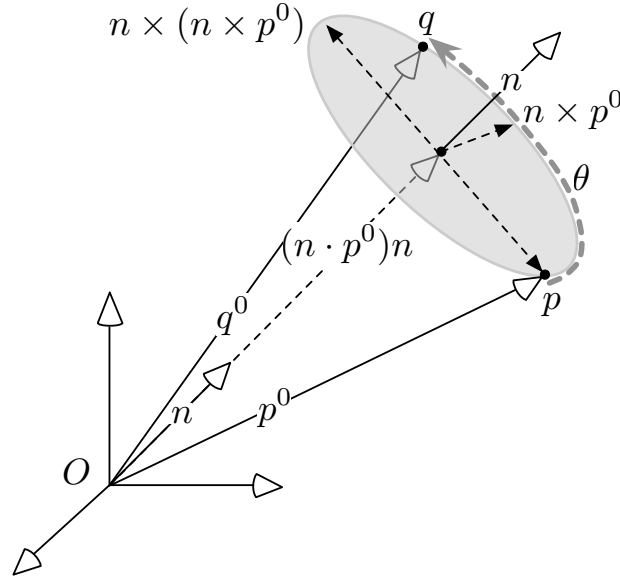


Figure 7.6: The circle through p and q is orthogonal to n and has its center on line through n at the point $(n \cdot p^0)n$.

If we now write matrix products instead of cross products using the skew symmetric property that $\omega \times v = \hat{\omega}v$ where $\hat{\omega} \in \mathfrak{so}(3)$ is a skew symmetric matrix (see Appendix 7.4), the last equation can be rewritten as:

$$q^0 = \underbrace{\left(I_3 + \sin \theta \hat{n} + (1 - \cos \theta) \hat{n}^2 \right)}_{\text{this must be a rotation matrix}} p^0 =: \text{Rot}_n(\theta) p^0.$$

This equality concludes the proof of Theorem 7.10. ■

Next, we aim to prove Theorem 7.12 about the inverse Rodrigues formula. To do so, we present a simple result first.

Lemma 7.14 (Angle of a rotation matrix). *The angle of rotation θ of any $R \in \text{SO}(3)$ satisfies $\text{trace}(R) = 1 + 2 \cos(\theta)$.*

Proof. If R is a basic rotation, then result is obvious. If R is a rotation about an arbitrary axis, we proceed as follows. First, we note that R is equal to $\text{Rot}_z(\theta)$ with respect to some frame. Therefore, we can write R as $S^T \text{Rot}_z(\theta) S$. Second, we compute $\text{trace}(R) = \text{trace}(S^T \text{Rot}_z(\theta) S) = \text{trace}(S S^T \text{Rot}_z(\theta)) = \text{trace}(\text{Rot}_z(\theta)) = 1 + 2 \cos \theta$. ■

Proof of Theorem 7.12. First, it is clear that the rotation axis n can be selected arbitrarily when we have the rotation matrix $R = I_3$ and the rotation angle $\theta = 0$. Second, from the Rodrigues' formula we sum R to its transpose R^T , note that the symmetric terms disappear, and we obtain

$$R - R^T = 2 \sin(\theta) \hat{n}.$$

Now, if $\sin(\theta) \neq 0$, i.e., if $0 < \theta < \pi$, we obtain the formula for n . Third, when $\theta = \pi$, there are two possible choices of rotation axis and the lengthy derivation is postponed to Exercise [E7.8](#). ■

7.6 Exercises

- E7.1 **Rotation matrices and dot products (20 points).** The rotation group $SO(3)$ is the set of all rotation matrices in 3-dimensional space, that is,

$$SO(3) = \{R \in \mathbb{R}^{3 \times 3} \mid RR^T = I_3, \det(R) = +1\}.$$

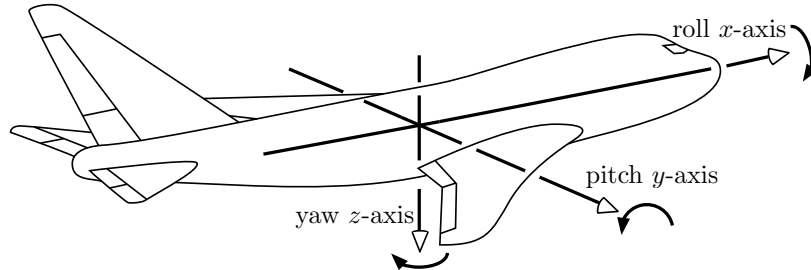
For all 3-dimensional column arrays v, w and rotation matrices R (i.e., elements of $SO(3)$), show:

- (i) (10 points) $v^T w = (Rv)^T (Rw)$,
- (ii) (5 points) $\|v\| = \|Rv\|$, where $\|v\|$ is the norm of the array v ,
- (iii) (5 points) Explain the geometric interpretation of facts (i) and (ii).

Hint: Regarding (i), it is not useful to write R in components, i.e., as $R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$. Instead, use the properties of the dot product between vectors and use the fact that R is a rotation matrix.

- E7.2 **Reasoning about roll-pitch-yaw angles (15 points).**

- (i) (10 points) Using the image below as a guide, find the rotation matrix representing a roll of $\pi/4$, followed by a pitch of $\pi/2$, followed by a yaw of $-\pi/2$. Assume that these rotations are expressed in the frame of the aircraft, i.e., they are expressed in successive frames.



- (ii) (5 points) Assume the plane is initially flying at constant altitude and right-side up. If the plane performs the composite rotations from part (i), then will the plane be pointed more towards space or the ground? Explain your reasoning.

- E7.3 **Computing Euler angles.** Consider the Euler angle representation of a rotation:

$$\begin{aligned} R_{ZYX} &= R_{z,\alpha} R_{y,\beta} R_{z,\gamma} \\ &= \begin{bmatrix} c_\alpha c_\beta c_\gamma - s_\alpha s_\gamma & -c_\alpha c_\beta s_\gamma - s_\alpha c_\gamma & c_\alpha s_\beta \\ s_\alpha c_\beta c_\gamma + c_\alpha s_\gamma & -s_\alpha c_\beta s_\gamma + c_\alpha c_\gamma & s_\alpha s_\beta \\ -s_\beta c_\gamma & s_\beta s_\gamma & c_\beta \end{bmatrix}. \end{aligned}$$

Given an arbitrary rotation matrix

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, \quad (\text{E7.1})$$

verify the expressions for the Euler angles α, β and γ of R given in the algorithm of Theorem 7.7.

- E7.4 **Forward and inverse kinematics of roll-pitch-yaw angles (25 points).** In the Roll-Pitch-Yaw parametrization of rotation matrices, a rotation R is represented by three angles $\{\alpha, \beta, \gamma\}$ corresponding to basic rotations about $X - Y - Z$ axis with respect to a fixed frame:

$$R = \text{Rot}_z(\gamma) \text{Rot}_y(\beta) \text{Rot}_x(\alpha).$$

- (i) (10 points) Compute the components of R as a function of α , β , and γ .

Hint: Adopt the convention $c_\alpha = \cos(\alpha)$, $s_\alpha = \sin(\alpha)$, $c_\beta = \cos(\beta)$, etc, to simplify notation.

- (ii) (15 points) Given a rotation matrix

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix},$$

compute the roll-pitch-yaw angles $\{\alpha, \beta, \gamma\}$ using only atan_2 as inverse trigonometric function.

Hint: Use the definition of arctangent with two arguments from Appendix 3.5.

- E7.5 **Rotation matrices (20 points).** Prove the four properties (P0)-(P4) in Subsection 7.1.2.

- E7.6 **Composing rotation matrices (20 points).** Pick a rigid body and let $\Sigma_{\text{body}} = (O_{\text{body}}, \{x_{\text{body}}, y_{\text{body}}, z_{\text{body}}\})$ be a reference frame attached to it. Assume the spatial reference frame $\Sigma_0 = (O_0, \{x_0, y_0, z_0\})$ is at a fixed location and orientation and let R_{initial} be the initial orientation of the rigid body with respect to the spatial frame. Perform the following subsequent rotations on the body:

- (i) rotate by $\pi/3$ about the x_0 axis,
- (ii) rotate by $\pi/5$ about the z_{body} axis,
- (iii) rotate by $\pi/2$ about the x_{body} axis,
- (iv) rotate by $\pi/4$ about the y_0 axis,
- (v) rotate by $\pi/6$ about the y_{body} axis, and
- (vi) rotate by $\pi/5$ about the z_0 axis.

What is the final orientation of the body? (It is sufficient to write a formula for it and it is not required to perform all the computations.)

- E7.7 **Eigenvectors of rotation matrices (35 points).** Show that for any unit-length vector $n \in \mathbb{R}^3$ and any angle θ :

- (i) (10 points) $n \times n = \hat{n}n = 0$.

- (ii) (15 points) $\text{Rot}_n(\theta) \cdot n = n$.

Hint: You do not need to write down n or $\text{Rot}_n(\theta)$ in components. Rather, recall the axis-angle parametrization of $\text{Rot}_n(\theta)$ and use part (i).

- (iii) (5 points) \hat{n}^2 is a 3×3 symmetric matrix.

- (iv) (5 points) Explain in a few sentences what the property in part (ii) means and why it is useful.

- E7.8 **On the Rodrigues' formula.**

Consider the Rodrigues' formula given in Theorem 7.10. Show that, for arbitrary unit-length rotation axis n and angle θ , the matrix $\text{Rot}_n(\theta) = I_3 + \sin \theta \hat{n} + (1 - \cos \theta) \hat{n}^2$ is a rotation matrix.

- E7.9 **On the inverse Rodrigues' formula.**

Consider the inverse Rodrigues' formula, given in Theorem 7.12. Consider a rotation matrix R with $\text{trace}(R) = -1$. Show that

- (i) if (θ, n) are an angle and axis representation of R , then $\theta = \pi$ and n is a solution to

$$nn^T = \frac{1}{2}(I_3 + R),$$

- (ii) if n^* is a solution, then $-n^*$ is also a solution,
 (iii) there exists at most one index $i \in \{1, 2, 3\}$ such that $R_{ii} = -1$,
 (iv) if $R_{ii} \neq -1$, then denoting the other two indices by j and k so that $\{1, 2, 3\} = \{i, j, k\}$, an axis of rotation is:

$$n_i = \sqrt{\frac{1}{2}(1 + R_{ii})}, \quad n_j = \frac{1}{2n_i}R_{ij}, \quad \text{and} \quad n_k = \frac{1}{2n_i}R_{ik}.$$

E7.10 Programming: Representing 3D rotations (30 points).

The purpose of this exercise is to write functions to translate between axis-angle representation and rotation matrix representation of rotations in three dimensions. Specifically, consider the two functions:

`computeRMfromAA` (10 points)

Input: axis-angle pair (θ, n) , where $\theta \in [0, \pi]$ and n is a unit-length vector

Output: corresponding rotation matrix R .

`computeAAfromRM` (10 points)

Input: a rotation matrix R .

Output: the angle-axis representation of R (one of the two solutions if $\text{trace}(R) = -1$ and an arbitrary rotation axis if $\text{trace}(R) = 3$).

For each function, do the following:

- (i) explain how to implement the function, possibly deriving analytic formulas, and characterize special cases,
- (ii) program the function, including correctness checks on the input data and appropriate error messages, and
- (iii) verify your function is correct on a broad range of test inputs (e.g., for rotation angles equal to 0 radians, π radians, or other in-between values).

E7.11 Angles and planar rotation matrices (20 points).

- (i) (5 points) Assume you are given two numbers a and b satisfying $a^2 + b^2 = 1$. Show that there exists a unique $\theta \in (-\pi, \pi]$ such that $a = \cos(\theta)$ and $b = \sin(\theta)$.

Hint: Draw the unit circle, the horizontal axis and the point (a, b) .

- (ii) (10 points) Assume you are given a 2×2 rotation matrix R , that is, a matrix

$$R = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix}$$

satisfying $R^T R = I_2$ and $\det(R) = +1$. Show that there exists a unique angle $\theta \in (-\pi, \pi]$ such that

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}.$$

Hint: Use the statement in part (i) and recall that, if you write $R = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix}$, then Cramer's rule says

$$\text{that } R^{-1} = \frac{1}{\det(R)} \begin{bmatrix} r_{22} & -r_{12} \\ -r_{21} & r_{11} \end{bmatrix}.$$

(iii) (5 points) If

$$R_1 = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1) \\ \sin(\theta_1) & \cos(\theta_1) \end{bmatrix}, \quad R_2 = \begin{bmatrix} \cos(\theta_2) & -\sin(\theta_2) \\ \sin(\theta_2) & \cos(\theta_2) \end{bmatrix},$$

then what is the angle for the rotation matrix $R_1 R_2$?

E7.12 **Reflection matrices (9 points).** A matrix R with orthonormal columns and a negative determinant is known as a *reflection matrix*.

- (i) (3 points) Show that a reflection matrix R has the properties $v^T w = (Rv)^T (Rw)$ and $\|v\| = \|Rv\|$.
- (ii) (3 points) Show that the product of two reflection matrices is a rotation matrix.
- (iii) (3 points) Explain why such matrices are called reflection matrices. Specifically, what line or what point do they represent a reflection with respect to?

E7.13 **Permutation matrices (16 points).**

- (i) (4 points) The set of permutations of $\{1, \dots, m\}$ is a group in the mathematical sense, as defined in Section 6.5. This set is called the *permutation group* and denote by the symbol P_m .
 - a) What does it mean to combine permutations?
 - b) What is the identity permutation?
 - c) What is the inverse permutation?
- (ii) (4 points) Show that each permutation in P_m can be described by an $m \times m$ matrix with entries equal to $\{0, +1\}$ such that each column and each row contains precisely one entry equal to $+1$.
- (iii) (4 points) Show that each permutation matrix is an orthogonal matrix in m dimensions (that is, P_m is a subset of $O(m)$).
- (iv) (4 points) Show that the group P_m contains a finite number of elements.

Displacement Matrices and Inverse Kinematics

In this chapter we study rigid-body displacements, i.e., motions composing rotations and translations. We will represent displacements by introducing displacement matrices. We will discover that displacement matrices have very similar properties to rotation matrices, as studied in the previous chapter.

8.1 Displacements as matrices

Consider two general frames, fix a frame in space and one with a moving body. Consider also a general point p . Assume the two frames are: $\Sigma_0 = (O_0, \{x_0, y_0, z_0\})$ and $\Sigma_1 = (O_1, \{x_1, y_1, z_1\})$, as depicted in Figure 8.1. To represent Σ_1 with respect to Σ_0 , note that

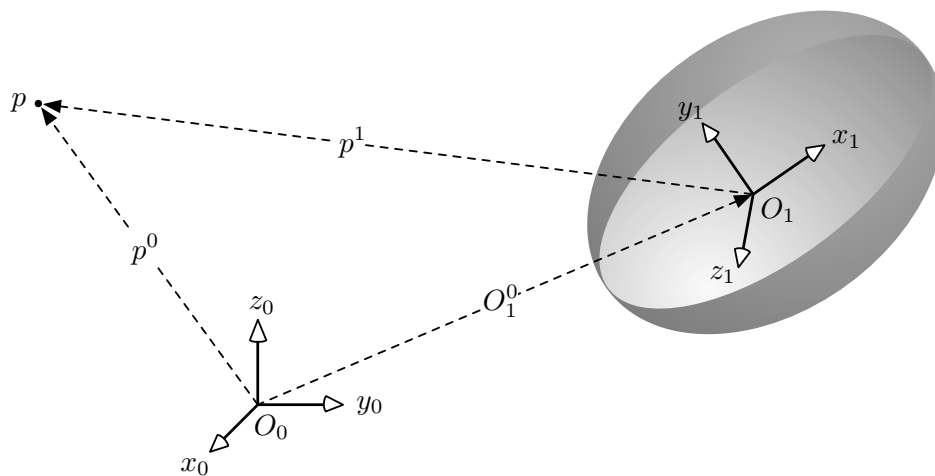


Figure 8.1: Reference frames and points in three-dimensions

(i) origin of Σ_1 with respect to Σ_0 is

$$O_1^0 = (\overrightarrow{O_0 O_1})^0,$$

(ii) the axes of Σ_1 with respect to Σ_0 are given by R_1^0 (as discussed in Chapter 6).

Therefore, the frame Σ_1 is represented by pair (O_1^0, R_1^0) where $O_1^0 \in \mathbb{R}^3$, and $R_1^0 \in \text{SO}(3)$.

8.1.1 Changes of reference frame and composition of displacements

Next, let us compute the changes of reference frame. Because $\overrightarrow{O_0 p} = \overrightarrow{O_0 O_1} + \overrightarrow{O_1 p}$ and $(\overrightarrow{O_1 p})^0 = R_1^0(\overrightarrow{O_1 p})^1$, we obtain:

$$p^0 = R_1^0 p^1 + O_1^0. \quad (8.1)$$

Given three frames, frame-changes (O_1^0, R_1^0) and (O_2^1, R_2^1) , what is the frame-change (O_2^0, R_2^0) ? Let us collect known facts:

$$p^0 = R_1^0 p^1 + O_1^0, \quad p^1 = R_2^1 p^2 + O_2^1, \quad p^0 = R_2^0 p^2 + O_2^0.$$

Substitute the second equality into the first:

$$p^0 = R_1^0(R_2^1 p^2 + O_2^1) + O_1^0 = (R_1^0 R_2^1) p^2 + (R_1^0 O_2^1 + O_1^0).$$

This implies

$$R_2^0 = R_1^0 R_2^1, \quad \text{and} \quad O_2^0 = R_1^0 O_2^1 + O_1^0. \quad (8.2)$$

As usual, note the important role of subscript and superscripts.

8.1.2 Matrix representation of displacements and homogeneous representation of points

To simplify bookkeeping in equations (8.1) and (8.2), we write displacements as *displacement matrices* and points in *homogeneous representation*.

First, instead of using the pair (R_1^0, O_1^0) , we represent the frame Σ_1 with respect to Σ_0 by the single matrix

$$H_1^0 = \begin{bmatrix} R_1^0 & O_1^0 \\ 0_{1 \times 3} & 1 \end{bmatrix},$$

where $0_{1 \times 3}$ is 1×3 -vector of zeros. We call such a matrix a *displacement matrix*. We can now rewrite equation (8.2) as

$$H_2^0 = H_1^0 H_2^1.$$

This result is true because of the multiplication between block matrices:

$$\begin{bmatrix} R_1^0 & O_1^0 \\ 0_{1 \times 3} & 1 \end{bmatrix} \begin{bmatrix} R_2^1 & O_2^1 \\ 0_{1 \times 3} & 1 \end{bmatrix} = \begin{bmatrix} R_1^0 R_2^1 & R_1^0 O_2^1 + O_1^0 \\ 0_{1 \times 3} & 1 \end{bmatrix}.$$

In summary, as for rotation matrices, we note that *displacement composition corresponds to matrix products*.

Second, given a point p , we define the *homogeneous representations* of the point p with respect to Σ_0 by

$$P^0 = \begin{bmatrix} p^0 \\ 1 \end{bmatrix}.$$

With this notion, the changes of reference frame in equation (8.1) is rewritten as the following matrix-vector multiplication:

$$P^0 = H_1^0 P^1. \quad (8.3)$$

We summarize similarities and differences in Table 8.1.

| | Rotations | Displacements |
|--|-----------------------|------------------------|
| how to represent Σ_1 with respect to Σ_0 : | R_1^0 | H_1^0 |
| how to represent a point p : | p^0 | P^0 |
| how to change reference frame: | $p^0 = R_1^0 p^1$ | $P^0 = H_1^0 P^1$ |
| how to compose displacements: | $R_2^0 = R_1^0 R_2^1$ | $H_2^0 = H_1^0 H_2^1$ |
| inverse rotation/displacement: | $R_0^1 = (R_1^0)^T$ | $H_0^1 = (H_1^0)^{-1}$ |

Table 8.1: Comparison between rotations and displacements

Regarding the last property, the inverse reference frame representation is

$$H_0^1 = (H_1^0)^{-1} = \begin{bmatrix} (R_1^0)^T & -(R_1^0)^T O_1^0 \\ 0_{1 \times 3} & 1 \end{bmatrix}.$$

To see that this formula is correct, it suffices to show that

$$\begin{bmatrix} R_1^0 & O_1^0 \\ 0_{1 \times 3} & 1 \end{bmatrix} \begin{bmatrix} (R_1^0)^T & -(R_1^0)^T O_1^0 \\ 0_{1 \times 3} & 1 \end{bmatrix} = I_4.$$

The formula for the inverse frame representation is illustrated in Figure 8.2.

8.1.3 Displacement matrices

The set of *displacement matrices* (also called special Euclidean set) is

$$\text{SE}(3) = \left\{ H \in \mathbb{R}^{4 \times 4} \mid H = \begin{bmatrix} R & d \\ 0 & 1 \end{bmatrix} \text{ where } R \in \text{SO}(3), d \in \mathbb{R}^3 \right\}.$$

Just like the set of rotation matrices $\text{SO}(3)$, the set of displacement matrices $\text{SE}(3)$ is closed with respect to matrix multiplication, that is,

$$H_1 \text{ and } H_2 \in \text{SE}(3) \implies H_1 \cdot H_2 \in \text{SE}(3).$$

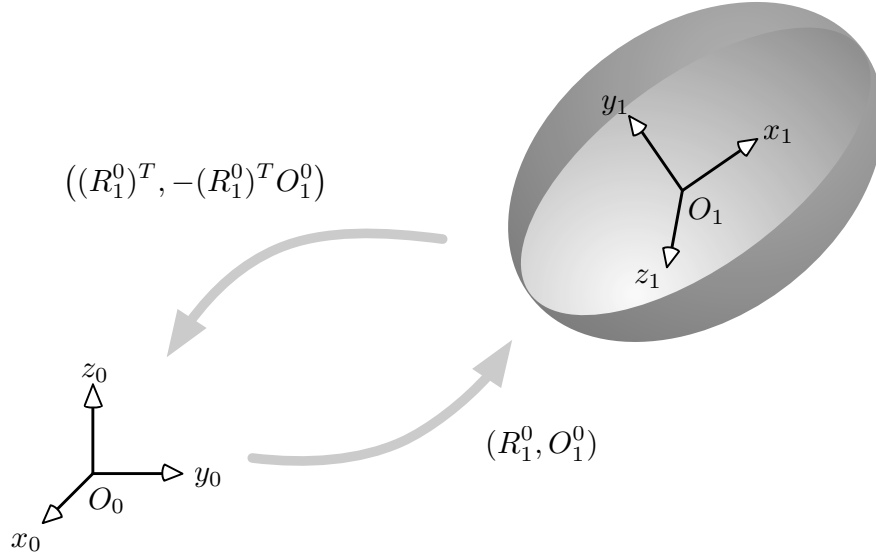


Figure 8.2: Inverse frame representation

Moreover, as with the set $SO(3)$, the order of multiplication of displacements matters:

$$H_1 \cdot H_2 \neq H_2 \cdot H_1.$$

And, finally, as with the set $SO(3)$, the set of displacement matrices $SE(3)$ has the properties of a group, that is

- (i) $H_1(H_2H_3) = (H_1H_2)H_3$ (associativity property),
- (ii) $I_4 \in SE(3)$ is the identity displacement, and
- (iii) for any displacement

$$H = \begin{bmatrix} R & v \\ 0_{1 \times 3} & 1 \end{bmatrix},$$

there always exists the inverse displacement

$$H^{-1} = \begin{bmatrix} R^T & -R^T v \\ 0_{1 \times 3} & 1 \end{bmatrix} \in SE(3).$$

8.2 Basic and composite displacements

Just like we did for rotation matrices, it is useful to emphasize the three roles and uses of displacement matrices:

- (i) H_1^0 describes Σ_1 with respect to Σ_0 , that is, the columns of R_1^0 are the bases vectors of Σ_1 with respect to Σ_0 , and O_1^0 is origin.

- (ii) H_1^0 is the coordinate transformation from Σ_1 to Σ_0 , by using the identity $P^0 = H_1^0 P^1$.
- (iii) Any $H \in \text{SE}(3)$ can be used to *rotate and then translate* a vector. Consider a point p , define a new point q as follows: the vector $\overrightarrow{O_0 q}$ is the rotation of the vector $\overrightarrow{O_0 p}$ by an angle θ about axis n^0 , and followed by the translation by a vector t^0 :

$$q^0 = \text{Rot}_{n^0}(\theta)p^0 + t^0 \quad \Longleftrightarrow \quad Q^0 = H P^0, \text{ where } H = \begin{bmatrix} \text{Rot}_n(\theta) & t^0 \\ 0 & 1 \end{bmatrix}.$$

Note that the operation “rotate and then translate” is not the same as “first translate and then rotate.” In what follows we review basic displacements, and how to compose them.

8.2.1 The six basic displacements

There are six basic displacements: three basic rotations and three basic translations. For $\alpha, \beta, \gamma \in [-\pi, \pi]$, the three basic rotations are

$$\text{Rotd}_x(\alpha) = \begin{bmatrix} \text{Rot}_x(\alpha) & 0_{3 \times 1} \\ 0_{1 \times 3} & 1 \end{bmatrix}, \quad \text{Rotd}_y(\beta) = \begin{bmatrix} \text{Rot}_y(\beta) & 0_{3 \times 1} \\ 0_{1 \times 3} & 1 \end{bmatrix}, \quad \text{and} \quad \text{Rotd}_z(\gamma) = \begin{bmatrix} \text{Rot}_z(\gamma) & 0_{3 \times 1} \\ 0_{1 \times 3} & 1 \end{bmatrix}.$$

Note the choice of symbols $\text{Rot}_x(\alpha)$ is a rotation matrix representing the rotation by α about x , and $\text{Rotd}_x(\alpha)$ is the displacement matrix representing the same rotation.

For $a, b, c \in \mathbb{R}$, the three basic translations are

$$\text{Trans}_x(a) = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \text{Trans}_y(b) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \text{and} \quad \text{Trans}_z(c) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

For example, the matrix $\text{Trans}_x(a)$ is a translation along the x -axis a distance of a .

8.2.2 Composition of displacements

Let H_1^0 represent Σ_1 with respect to Σ_0 and consider a second displacement H of Σ_1

- (i) if H is expressed in current frame = successive frame = Σ_1 , then the composite displacement is obtained by *post-multiplication = right-multiplication*:

$$H_2^0 = H_1^0 H.$$

- (ii) if H is expressed in fixed frame = body frame = Σ_0 , then the composite displacement is obtained by *pre-multiplication = left-multiplication*:

$$H_2^0 = H H_1^0.$$

8.2.3 Composite displacements

In this subsection we study and classify the possible displacements that can be applied to a point in 3-dimensional Euclidean space. Recall that every operation on a point can be regarded as an operation on a rigid body by applying that operation to every point attached to the rigid body.

We assume n and t are two unit-length axes, θ is an angle, d is a distance, and r is a reference point. Given a reference frame $\Sigma_0 = \{O_0, \{x_0, y_0, z_0\}\}$, let r^0 denote the array representation of r with respect to Σ_0 . For simplicity, let n and t denote both the free vectors and their array representations in Σ . Recall that the rotation matrix $\text{Rot}_n(\theta)$ represents the rotation by an angle θ about the n axis passing through the point 0_0 , when expressed in the reference frame Σ_0 .

We describe all possible displacements in Table 8.2. Note that classification in the table is redundant, in the sense that some operations are special cases of others.

| Displacement operation | Displacement matrix in SE(3) |
|--|---|
| (i) Translation by a distance d along an axis t | $\text{Trans}_t(d) = \begin{bmatrix} I_3 & dt \\ 0_{1 \times 3} & 1 \end{bmatrix}$ |
| (ii) Rotation by an angle θ about an axis n passing through the origin | $\text{Rot}_n(\theta) = \begin{bmatrix} \text{Rot}_n(\theta) & 0_{3 \times 1} \\ 0_{1 \times 3} & 1 \end{bmatrix}$ |
| (iii) Rotation by an angle θ about an axis n passing through a point r | $\text{Rot}_{n,r}(\theta) = \begin{bmatrix} \text{Rot}_n(\theta) & (I_3 - \text{Rot}_n(\theta))r^0 \\ 0_{1 \times 3} & 1 \end{bmatrix}$ |
| (iv) Rotation by θ about n passing through the origin, and then translation by d along t | $\text{Trans}_t(d) \text{Rot}_n(\theta) = \begin{bmatrix} \text{Rot}_n(\theta) & dt \\ 0_{1 \times 3} & 1 \end{bmatrix}$ |
| (v) Rotation by θ about n passing through r , and then translation by d along t | $\text{Trans}_t(d) \text{Rot}_{n,r}(\theta) = \begin{bmatrix} \text{Rot}_n(\theta) & (I_3 - \text{Rot}_n(\theta))r^0 + dt \\ 0_{1 \times 3} & 1 \end{bmatrix}$ |
| (vi) Translation by d along t , and then rotation by θ about n passing through the origin | $\text{Rot}_n(\theta) \text{Trans}_t(d) = \begin{bmatrix} \text{Rot}_n(\theta) & d \text{Rot}_n(\theta)t \\ 0_{1 \times 3} & 1 \end{bmatrix}$ |
| (vii) Translation by d along t , and then rotation by θ about n passing through r | $\text{Rot}_{n,r}(\theta) \text{Trans}_t(d) = \begin{bmatrix} \text{Rot}_n(\theta) & d \text{Rot}_n(\theta)t + (I_3 - \text{Rot}_n(\theta))r^0 \\ 0_{1 \times 3} & 1 \end{bmatrix}$ |

Table 8.2: Composite displacements and their corresponding displacement matrices

In what follows we justify the expressions for the displacement operations in the first three rows of the table. The other rows are obtained by combining the results in the first three rows. Let p be an arbitrary point and let q be the point resulting from the displacement of p according to one of the operations in the table. Let p^0 and q^0 be their representation in Σ^0 and let $P^0, Q^0 \in \mathbb{R}^4$ be their homogeneous representations.

(i) Translation by distance d along a vector t :

$$q^0 = p^0 + dt,$$

$$Q^0 = \begin{bmatrix} I_3 & dt \\ 0_{1 \times 3} & 1 \end{bmatrix} P^0.$$

(ii) Rotation by an angle θ about an axis n passing through the origin. Using Rodrigues' formula, we write

$$q^0 = \text{Rot}_n(\theta)p^0,$$

$$Q^0 = \begin{bmatrix} \text{Rot}_n(\theta) & 0_{3 \times 1} \\ 0_{1 \times 3} & 1 \end{bmatrix} P^0.$$

(iii) Rotation by an angle θ about an axis n passing through a point r . We decompose this action as follows.

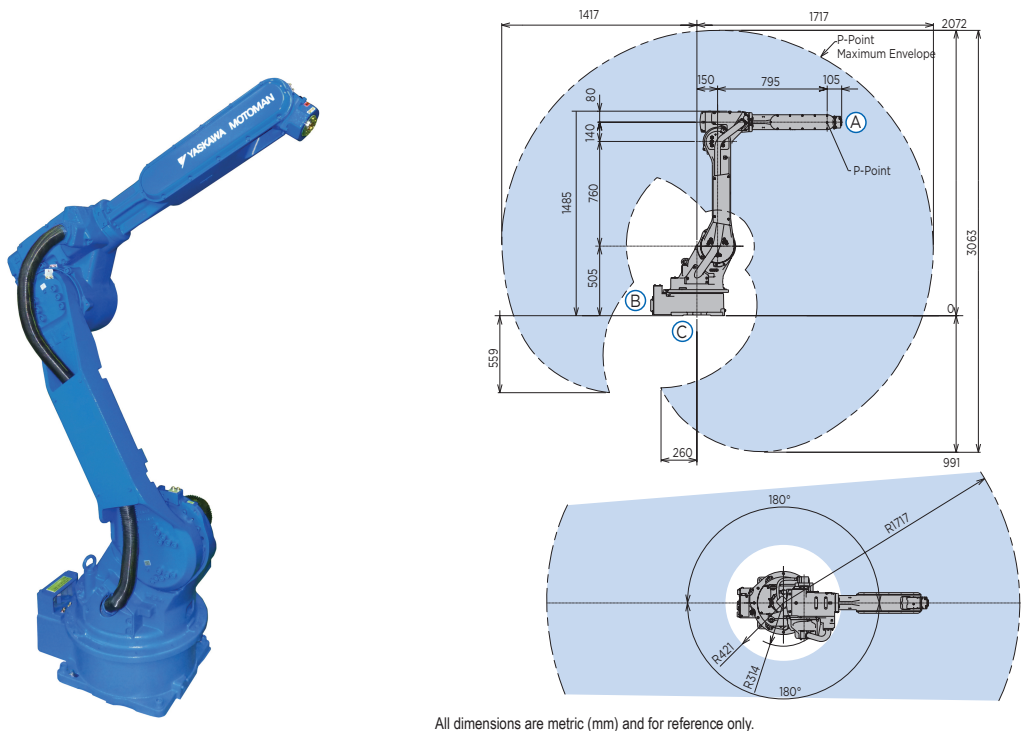
First, we express p in a reference frame Σ_1 with r as origin and with unchanged axes. The frame Σ_1 with respect to Σ_0 has $R_1^0 = I_3$ and $O_1^0 = r^0$. Therefore, we have $p^0 = R_1^0 p^1 + O_1^0 = p^1 + r^0$ and $p^1 = p^0 - r^0$.

Second, we rotate p^1 about the new origin using $\text{Rot}_n(\theta)$ to obtain $q^1 = \text{Rot}_n(\theta)p^1 = \text{Rot}_n(\theta)(p^0 - r^0)$.

Third and final, we express the resulting point back with respect to Σ_0 by using $q^0 = R_1^0 q^1 + O_1^0 = I_3 \text{Rot}_n(\theta)(p^0 - r^0) + r^0 = \text{Rot}_n(\theta)p^0 + (I_3 - \text{Rot}_n(\theta))r^0$.

8.3 Inverse kinematics on the set of displacements

Recall the inverse kinematics problem introduced in Chapter 3. We are given a desired configuration for the end effector of the robot, and our goal is compute joint angle that achieve the desired configuration. In Chapter 3 we studied a solved the inverse kinematics problem for a 2-link planar manipulator arm, with links connected by revolute joints (see Figure 3.15). For the 2-link robot, we could compute the joint angles θ_1 and θ_2 using geometric arguments. However, for more complex robots that have higher-dimensional configuration spaces, such calculations are very challenging. In this section we will see how reference frames and displacement matrices can be used to solve the inverse kinematics problem.



All dimensions are metric (mm) and for reference only.

Figure 8.3: The Motoman© HP20 manipulator is a multi-body robot versatile high-speed industrial robot with a slim base, waist, and arm. Image courtesy of Yaskawa Motoman, <http://www.motoman.com>.

8.3.1 Multi-link robots

Multi-link robots are made for a variety of purposes, from the larger industrial automation robots as shown in Figure 8.3, to smaller assistive robots as shown in Figure 8.4. For many applications, a common design is to have a base consisting of three links connected by three revolute joints. The base is used to position the end effector. The end effector, or gripper, is commonly attached to the base using a spherical wrist. The wrist is used to orient the gripper at its desired position.

8.3.2 Multiple frames and loops

Consider an environment with multiple frames and with corresponding displacements that form a loop, as depicted in Figure 8.5. In general, we know that $H_i^j = \text{frame } \Sigma_i \text{ expressed with respect to } \Sigma_j$.

We consider the following sample problem: assume we know all matrices corresponding to the displacements illustrated in figure, that is, H_1^0 , H_2^1 , H_3^0 , H_4^2 , except for the displacement matrix H_4^3 . How would you compute H_4^3 ?

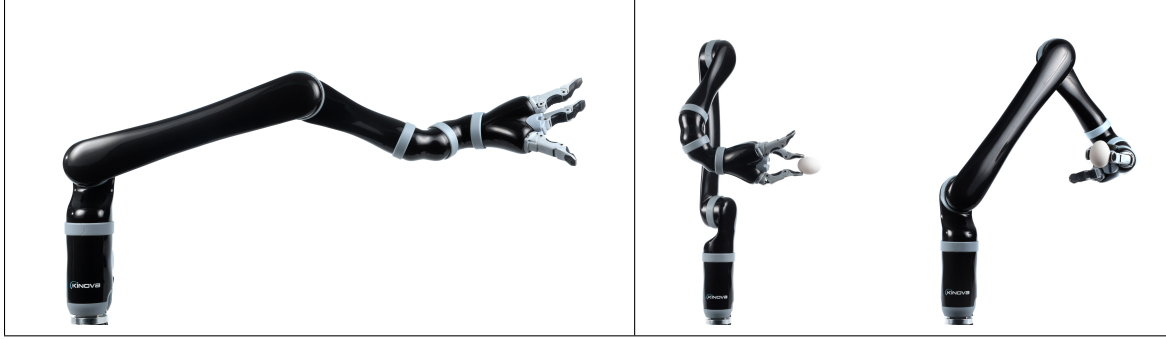


Figure 8.4: The Jaco² by Kinova Robotics is a light-weight robot arm with six degrees of freedom. Images courtesy of Kinova Robotics, <http://kinovarobotics.com/>.

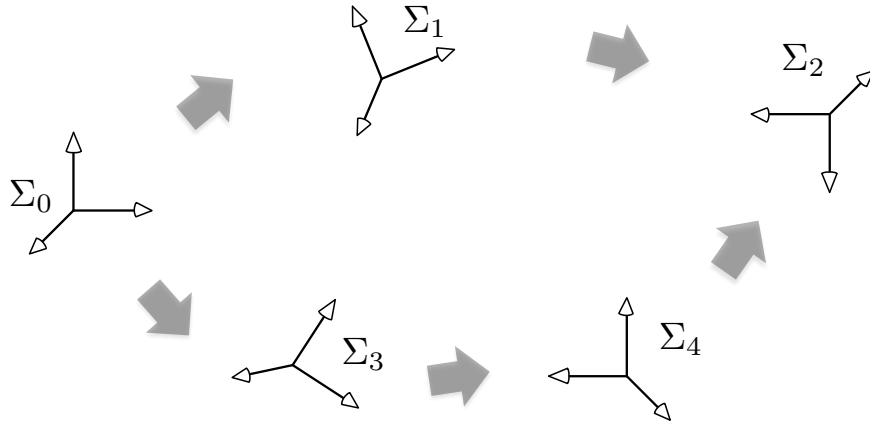


Figure 8.5: Multiple reference frames in a three-dimensional environment

Write known facts and obtain loop relationship

$$\begin{aligned} H_2^0 &= H_1^0 H_2^1, \\ H_2^0 &= H_3^0 H_4^3 H_2^4, \\ \implies H_1^0 H_2^1 &= H_3^0 H_4^3 H_2^4. \end{aligned}$$

Therefore

$$H_4^3 = (H_3^0)^{-1} H_1^0 H_2^1 (H_2^4)^{-1}.$$

8.3.3 The pick-up and place problem

Consider the pick-up and place problem illustrated in Figure 8.6. Our goal is use the robot to grasp the object sitting on the table, which is an instance of inverse kinematics problem.

- The reference frames Σ_{base} , Σ_{gripper} , Σ_{table} , and Σ_{object} describe the placement and orientation of relevant objects in the environment.

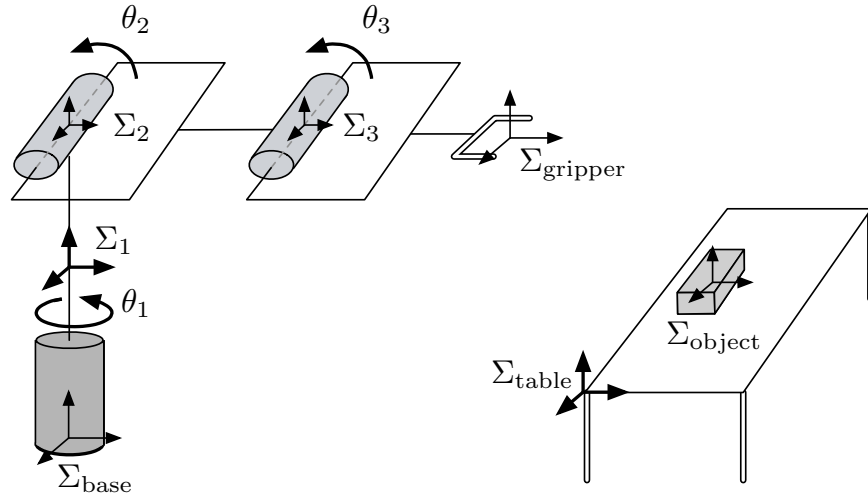


Figure 8.6: Prototypical pick-up and place problem

- Each link of the robot (i.e., each rigid body component of the robot) is described by a reference frame: $\Sigma_1, \Sigma_2, \Sigma_3$.
- As an aside, a systematic methodology to place frames in each link of a robot is the Denavit and Hartenberg (D-H) convention (see [Wikipedia:D-H Parameters](#) for a discussion and a clarifying video). Once frames are placed, we should have an expression for the displacement matrices

$$H_1^{\text{base}}(\theta_1), H_2^1(\theta_2), \text{ and } H_3^2(\theta_3),$$

as a function of the three joint angles.

- Because of the way the gripper works, the gripper and object need to be at a specific, desirable “relative displacement with respect to each other” in order for the correct grasp to take place. Denote this object displacement with respect to the gripper by H_{desired} . Then, the correct grasp takes place when

$$H_{\text{object}}^{\text{gripper}} = H_{\text{desired}}.$$

- Now, we can apply the same ideas as in the previous discussion about multiple frames and loops. First, let us study the “object with respect to gripper” displacement

$$H_{\text{desired}} = H_{\text{object}}^{\text{gripper}} = H_{\text{base}}^{\text{gripper}} H_{\text{table}}^{\text{base}} H_{\text{object}}^{\text{table}} = (H_{\text{gripper}}^{\text{base}})^{-1} H_{\text{table}}^{\text{base}} H_{\text{object}}^{\text{table}}. \quad (8.4)$$

- Second, let us study the “gripper with respect to base” displacement, which depends upon the robot joint angles, i.e., the robot forward kinematics:

$$H_{\text{gripper}}^{\text{base}}(\theta_1, \theta_2, \theta_3) = H_1^{\text{base}}(\theta_1) H_2^1(\theta_2) H_3^2(\theta_3) H_{\text{gripper}}^3. \quad (8.5)$$

- Finally, to place gripper at correct displacement, plug equation (8.5) into equation (8.4) and solve for joint angles = robot inverse kinematics

$$H_1^{\text{base}}(\theta_1)H_2^1(\theta_2)H_3^2(\theta_3)H_{\text{gripper}}^3 = H_{\text{table}}^{\text{base}}H_{\text{object}}^{\text{table}}H_{\text{desired}}^{-1}. \quad (8.6)$$

Solving the equation (8.6) is akin to computing Euler angles for a rotation matrix: trigonometric calculations are typical.

To continue the analysis, we draw the manipulator in Figure 8.7, measure some distances and write down the three relevant displacement matrices as function of the joint angles.

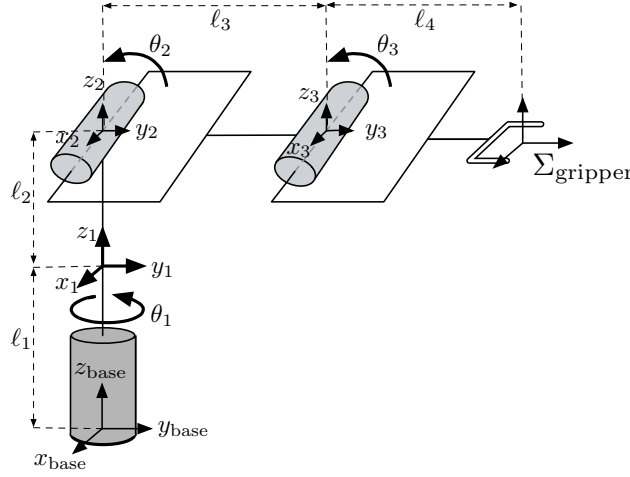


Figure 8.7: Manipulator with three-revolute joints: variables and parameters

$$H_1^{\text{base}}(\theta_1) = \begin{bmatrix} \text{Rot}_z(\theta_1) & \begin{bmatrix} 0 \\ 0 \\ l_1 \end{bmatrix} \\ 0_{1 \times 3} & 1 \end{bmatrix}, \quad H_2^1(\theta_2) = \begin{bmatrix} \text{Rot}_x(\theta_2) & \begin{bmatrix} 0 \\ 0 \\ l_2 \end{bmatrix} \\ 0_{1 \times 3} & 1 \end{bmatrix},$$

$$H_3^2(\theta_3) = \begin{bmatrix} \text{Rot}_x(\theta_3) & \begin{bmatrix} 0 \\ l_3 \\ 0 \end{bmatrix} \\ 0_{1 \times 3} & 1 \end{bmatrix}, \quad \text{and } H_{\text{gripper}}^3 = \begin{bmatrix} I_3 & \begin{bmatrix} 0 \\ l_4 \\ 0 \end{bmatrix} \\ 0_{1 \times 3} & 1 \end{bmatrix}.$$

After some careful bookkeeping (or some symbolic computations on a computer), we obtain

$$H_1^{\text{base}}(\theta_1)H_2^1(\theta_2)H_3^2(\theta_3)H_{\text{gripper}}^3 = \begin{bmatrix} \cos(\theta_1) & -\cos(\theta_2 + \theta_3)\sin(\theta_1) & \sin(\theta_1)\sin(\theta_2 + \theta_3) & -\sin(\theta_1)(\ell_3 \cos(\theta_2) + \ell_4 \cos(\theta_2 + \theta_3)) \\ \sin(\theta_1) & \cos(\theta_1)\cos(\theta_2 + \theta_3) & -\cos(\theta_1)\sin(\theta_2 + \theta_3) & \cos(\theta_1)(\ell_3 \cos(\theta_2) + \ell_4 \cos(\theta_2 + \theta_3)) \\ 0 & \sin(\theta_2 + \theta_3) & \cos(\theta_2 + \theta_3) & \ell_1 + \ell_2 + \ell_3 \sin(\theta_2) + \ell_4 \sin(\theta_2 + \theta_3) \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (8.7)$$

In what follows, we are only interested in setting the location of the end-effector equal to that of the object. In other words, we consider only the three translation components of equation (8.6), and we only aim to find the angles $\theta_1, \theta_2, \theta_3$ such that

$$\begin{bmatrix} -\sin(\theta_1)(\ell_3 \cos(\theta_2) + \ell_4 \cos(\theta_2 + \theta_3)) \\ \cos(\theta_1)(\ell_3 \cos(\theta_2) + \ell_4 \cos(\theta_2 + \theta_3)) \\ \ell_1 + \ell_2 + \ell_3 \sin(\theta_2) + \ell_4 \sin(\theta_2 + \theta_3) \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix},$$

where (v_1, v_2, v_3) is the translation vector of the displacement matrix $H_{\text{table}}^{\text{base}} H_{\text{object}}^{\text{table}} H_{\text{desired}}^{-1}$. This is an inverse kinematics problem.

Suppose we look for solutions where $\theta_2 \in]-\pi/2, \pi/2[$ and $\theta_2 + \theta_3 \in]-\pi/2, \pi/2[$ so that the corresponding cosines are strictly positive. Then we have the following equalities:

$$\begin{aligned} \theta_1 &= \text{atan}_2(\sin \theta_1, \cos \theta_1) = \text{atan}_2(-v_1, v_2), \\ \ell_3 \cos(\theta_2) + \ell_4 \cos(\theta_2 + \theta_3) &= \sqrt{v_1^2 + v_2^2}, \\ \ell_3 \sin(\theta_2) + \ell_4 \sin(\theta_2 + \theta_3) &= v_3 - \ell_1 - \ell_2. \end{aligned}$$

The last two equations are identical to the ones we solved for the 2-link manipulator; see Proposition 3.1.

8.4 Exercises

E8.1 Frame displacements (20 points).

Imagine the frame $\Sigma_S = (O_S, \{x_S, y_S, z_S\})$ is fixed.

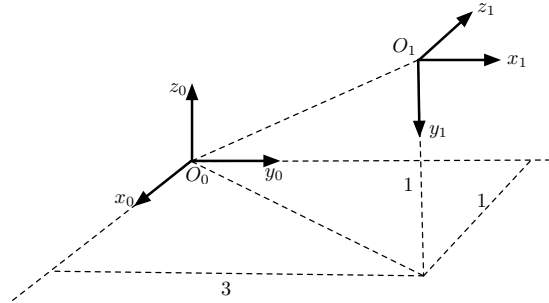
- (i) First, draw the frame Σ_S and then sketch the frames Σ_M and Σ_N determined by

$$H_M^S = \begin{pmatrix} 1/2 & 0 & \sqrt{3}/2 & 1 \\ 0 & 1 & 0 & -1 \\ -\sqrt{3}/2 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad H_S^N = \begin{pmatrix} 0 & 0 & -1 & 0 \\ -1/2 & \sqrt{3}/2 & 0 & -\sqrt{3} \\ \sqrt{3}/2 & 1/2 & 0 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

- (ii) Compute the displacement H_M^N .

E8.2 Change of reference frame (14 points).

- (i) Write the displacement matrix describing the position and orientation of frame 1 relative to frame 0.



- (ii) Suppose that

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is a homogeneous transformation defined in frame 0. Given the homogeneous representation of a point p in frame 0, denoted P^0 , then $Q^0 = AP^0$ is a new point in frame 0, and obtained by translating and scaling p . What is the homogeneous transformation A defined in frame 1?

E8.3 The set of planar displacements (10 points).

Consider the set of planar displacements, i.e., displacements on the horizontal plan described by a rotation about the vertical axis by an angle θ and a horizontal translation described by a vector $[x, y]$. For such a planar displacement, we define the planar displacement matrix

$$H = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & x \\ \sin(\theta) & \cos(\theta) & y \\ 0 & 0 & 1 \end{bmatrix}.$$

We denote the set of 3×3 planar displacement matrices by the symbol $SE(2)$.

- Show that the set of planar displacement matrices $SE(2)$ is closed with respect to matrix multiplication, that is, the product of two planar displacement matrices is again a planar displacement matrix.
- What is the planar displacement matrix describing the zero-rotation zero-translation displacement?
- Compute the inverse matrix H^{-1} and verify it is a planar displacement matrix.

E8.4 Commuting displacements (20 points).

Let v be a unit-length vector. Consider an angle θ and a distance d . Let $\text{Rot}_v(\theta)$ be 4×4 displacement matrix representing the rotation about v by an angle θ and let $\text{Trans}_v(d)$ be the 4×4 displacement matrix representing translation along v by a distance d , so that

$$\text{Rot}_v(\theta) = \begin{bmatrix} \text{Rot}_v(\theta) & 0_{3 \times 1} \\ 0_{1 \times 3} & 1 \end{bmatrix}, \quad \text{and} \quad \text{Trans}_v(d) = \begin{bmatrix} I_3 & dv \\ 0_{1 \times 3} & 1 \end{bmatrix}.$$

In general, displacement matrices do not commute, but when the rotations and translations are about the same vector then they do. Perform the appropriate computations to show that the equality

$$\text{Rot}_v(\theta) \text{Trans}_v(d) = \text{Trans}_v(d) \text{Rot}_v(\theta)$$

is true for all v , θ , and d .

Hint: Think about how to show this commuting relationship using matrix multiplication by blocks and the properties of rotation matrices.

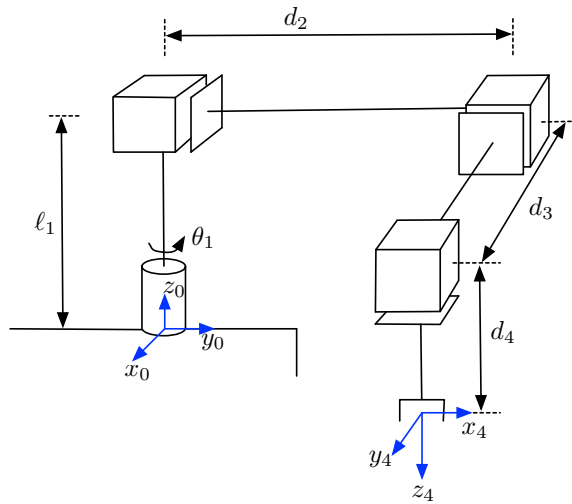
E8.5 Using reference frames (20 points).

Consider three robotic cameras with respective body-fixed reference frames Σ_1 , Σ_2 , and Σ_3 . At some time, camera 1 detects an object of interest and, through an appropriate computer-vision algorithm, estimates the object to be located at coordinates $(-2, -1, 5)$, as expressed with respect to Σ_1 . At that same time, assume the other camera frames satisfy

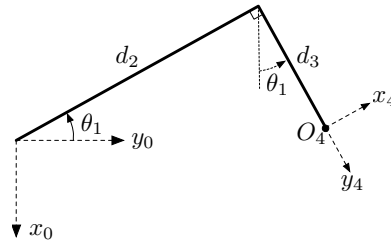
$$H_2^1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \text{and} \quad H_2^3 = \begin{bmatrix} 0 & 0 & -1 & 0 \\ -0.24 & -0.97 & 0 & 0 \\ -0.97 & -0.24 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

What is the position of the object in reference frames Σ_2 and Σ_3 ?

E8.6 Kinematics of a manipulator (20 points). Consider the manipulator shown below consisting of four links connected by four joints. The first joint is revolute and allows the robot to rotate about the z_0 axis. The next three joints (shown as “boxes”) are prismatic joints that can extend and retract. Assume d_2 , d_3 and d_4 can take any value between 1 and 2 and that in the shown configuration, $\theta_1 = 0$. The distance ℓ_1 is fixed, and assume it is equal to 1.



- (i) What is the configuration space of this robot, and how many degrees of freedom does this robot have?
- (ii) Determine the displacement matrix that defines frame 4 relative to frame 0. You should be able to do this by inspection with the aid of following top view of the manipulator.



- (iii) Suppose that we would like to grasp an object located at position

$$p^0 = \begin{bmatrix} -2 \\ 1.5 \\ 0 \end{bmatrix}$$

in frame 0. Find one configuration of the robot that places the origin of frame 4 at p^0 .

Hint: The figure from part (ii) may be helpful.

- (iv) What is R_0^4 (the orientation of frame 4 relative to frame 0) when the robot is grasping this object?

Linear and Angular Velocities of a Rigid Body

In this chapter we study the motion of rigid bodies and therefore of robots. We aim to understand the notion of linear and angular velocity of a rigid body. Being able to model linear and angular velocities is a stepping stone to analyzing, simulating and designing the motion of a rigid body.

As in the previous chapter and as illustrated in Figure 9.1, we consider two general frames:

- $\Sigma_0 = (O_0, \{x_0, y_0, z_0\})$ is fixed in space, i.e., it does not move at all. This frame is called the *spatial frame*, and
- $\Sigma_1 = (O_1, \{x_1, y_1, z_1\})$ is fixed with the rigid body and it is called the *body frame*.

Moreover, we let p denote a general point, possibly moving in space. We observe that this moving point satisfies the simple equation

$$\dot{p}^0(t) = v^0(t), \quad (9.1)$$

where v^0 is the *linear velocity* of the point p expressed with respect to the frame Σ_0 . Here, we use the standard notation $\dot{p}^0 := \frac{dp^0}{dt}$ to denote the time derivative of p^0 . It is worth emphasizing that, in equation (9.1), both the point and its linear velocity are expressed with respect to a fixed frame Σ_0 .

Next, here are the questions we answer in this chapter:

- how do we describe the velocity of a rotating body or rotating frame?
- what are linear and angular velocities of a moving rigid body?
- what is the velocity of a time-varying rotation matrix $R(t)$?
- assuming we understand a notion of angular velocity, in what frame is the angular velocity expressed?

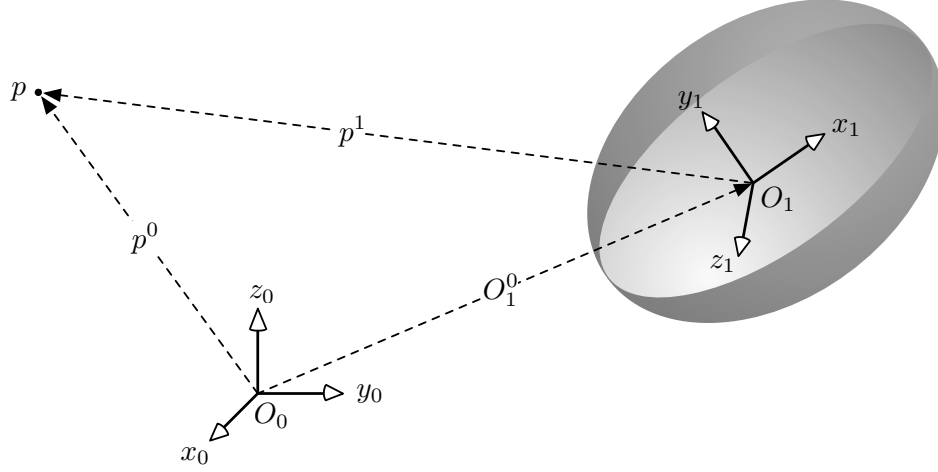


Figure 9.1: Reference frames and points in three-dimensions

9.1 Angular velocity

We first perform some insightful calculations on rotation matrices and then explain their meaning as the body angular velocity in spatial and body frame, respectively.

Time derivative of rotation matrices

Recall the set of skew symmetric matrices $\mathfrak{so}(3)$ and the operation $\hat{\cdot}$ from \mathbb{R}^3 to $\mathfrak{so}(3)$. In addition, given a matrix $R(t)$ with entries $r_{ij}(t)$, the derivative of $R(t)$ with respect to time is a matrix $\dot{R}(t)$ with entries $\dot{r}_{ij}(t)$.

Lemma 9.1 (The time-derivative of a rotation matrix). *Given a time-dependent rotation matrix $R(t)$ (i.e., a curve of rotations or a trajectory in the set of rotation matrices), there exist time-dependent vectors $\Omega_{\text{left}}(t) \in \mathbb{R}^3$ and $\Omega_{\text{right}}(t) \in \mathbb{R}^3$ such that*

$$\dot{R}(t) = \hat{\Omega}_{\text{left}}(t)R(t) = R(t)\hat{\Omega}_{\text{right}}(t).$$

Equivalently, we can write $\hat{\Omega}_{\text{left}}(t) := \dot{R}(t)R^T(t)$ and $\hat{\Omega}_{\text{right}}(t) := R^T(t)\dot{R}(t)$.

Proof. We reason along the following lines. To begin, we consider the equality $R(t)R^T(t) = I_3$ and compute its time derivative (dropping the time argument for simplicity)

$$\dot{R}R^T + R\dot{R}^T = 0_{3 \times 3},$$

where $0_{3 \times 3}$ is the 3×3 matrix of zeros. If we define $S(t) = \dot{R}(t)R^T(t)$, then the previous equation implies that $S(t) + S^T(t) = 0_{3 \times 3}$ and, in turn, that $S(t) \in \mathfrak{so}(3)$ for all times t . Therefore, there must exist a time-dependent vector $\Omega_{\text{left}}(t) \in \mathbb{R}^3$ satisfying

$$\hat{\Omega}_{\text{left}}(t) = S(t) = \dot{R}(t)R^T(t).$$

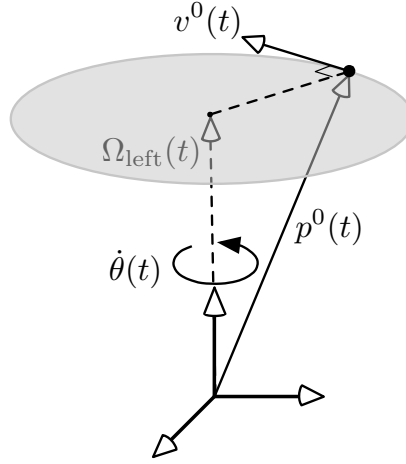


Figure 9.2: The angular velocity $\Omega_{\text{left}}(t)$ and instantaneous (or tangential) velocity $v^0(t)$ of a point $p^0(t)$

Post-multiplying both sides of the latter equation by $R(t)$ and using the fact that $R(t)R^T(t) = I_3$ we obtain

$$\dot{R}(t) = \hat{\Omega}_{\text{left}}(t)R(t).$$

To obtain the alternative result, we perform the same calculations starting from the equality $R(t)^T R(t) = I_3$. ■

The derivative of a rotation matrix with respect to time is related to the angular velocity of a rigid body (or, equivalently, of a frame fixed with the body). The previous lemma tells us that differentiating a rotation matrix is equivalent to multiplication by a skew symmetric matrix. Notice however that there are two ways to compute this derivative. Let us immediately clarify the meaning of the left angular velocity Ω_{left} .

Lemma 9.2 (Relationship between the “classic angular velocity” and the time derivative of a rotation matrix). *Consider a rigid body possessing a point O fixed in time and rotating about an axis n with a scalar angular speed $\dot{\theta}(t)$. Take two reference frames Σ_0 and Σ_1 coincident at time $t = 0$ and with origin fixed at O . Assume Σ_0 is fixed in space and Σ_1 is fixed with the body. Let $R_1^0(t)$ represent Σ_1 with respect to Σ_0 so that $\dot{R}_1^0(t) = \hat{\Omega}_{\text{left}}(t)R_1^0(t)$. Then*

$$\Omega_{\text{left}}(t) = \dot{\theta}(t)n^0.$$

Proof. We reason along the following lines. To begin, take a point p attached to frame Σ^1 and note that $\frac{d}{dt}p^1 = 0$ by construction. From the classic notion of angular velocity as illustrated in Figure 9.2, we know

$$\frac{d}{dt}p^0(t) = v^0(t) = (\dot{\theta}(t)n^0) \times p^0(t).$$

Next, write $p^0(t) = R_1^0(t)p^1$ and differentiate with respect to time to obtain

$$\frac{d}{dt}p^0(t) = \dot{R}_1^0(t)p^1 = \widehat{\Omega}_{\text{left}}(t)R_1^0(t)p^1 = \Omega_{\text{left}}(t) \times p^0(t),$$

where we have used the property that for a vector ω and its skew-symmetric matrix $\widehat{\omega}$ we have that the cross product with a vector v satisfies $\omega \times v = \widehat{\omega}v$ (see Appendix 7.4).

Equating the last two formulas we obtain $(\dot{\theta}(t)n^0) \times p^0(t) = \Omega_{\text{left}}(t) \times p^0(t)$ and, noting that $p^0(t)$ is arbitrary, we establish the desired result. ■

Thus, we see that Ω_{left} is a vector pointing along the axis of rotation and with length equal to the angular speed: that is, it is the angular velocity of the rigid body.

Angular velocity in spatial and body frame

Now that we have more properly the meaning of the left angular velocity Ω_{left} , we are ready to give it a more explanatory name and symbol. In previous example, $\dot{R}_1^0(t) = \widehat{\Omega}_{\text{left}}(t)R_1^0(t)$ clearly implies the following two facts:

- (i) Ω_{left} is angular velocity of Σ_1 with respect to Σ_0 , and
- (ii) Ω_{left} is expressed in frame Σ_0 .

As a consequence of these two facts we introduce the following notion.

Definition 9.3 (Angular velocity in spatial frame). *Given a rotating frame $\Sigma_1(t)$ (with fixed origin), its angular velocity with respect to a fixed spatial frame Σ_0 expressed in Σ_0 is*

$$\widehat{\Omega}_{0,1}^0(t) := \dot{R}_1^0(t)(R_1^0(t))^T.$$

Next, we need to properly explain the meaning of the right angular velocity Ω_{right} . First, from Appendix 7.4 a useful property of the skew symmetric operator $\widehat{\cdot}$ is that $R\widehat{u}R^T = \widehat{Ru}$, or equivalently

$$(\widehat{Ru})R = R\widehat{u}.$$

Starting from angular velocity in spatial frame, $\dot{R}_1^0(t) = \widehat{\Omega}_{0,1}^0(t)R_1^0(t)$, we define the *angular velocity in body frame* $\Omega_{0,1}^1$ by requiring it to satisfy the standard change of reference frame relation:

$$\Omega_{0,1}^0(t) = R_1^0(t)\Omega_{0,1}^1(t).$$

Finally, we put it all together and compute, dropping the time argument for simplicity,

$$\dot{R}_1^0 = (\widehat{R_1^0\Omega_{0,1}^1})R_1^0 = R_1^0\widehat{\Omega_{0,1}^1}(R_1^0)^T R_1^0 = R_1^0\widehat{\Omega_{0,1}^1}.$$

This derivation implies $\Omega_{0,1}^1(t) = \Omega_{\text{right}}(t)$ and the following definition.

Definition 9.4 (Angular velocity in body frame). *Given a rotating frame $\Sigma_1(t)$ (with fixed origin), its angular velocity with respect to a fixed spatial frame Σ_0 expressed in Σ_1 is*

$$\widehat{\Omega_{0,1}^1}(t) := (R_1^0(t))^T \dot{R}_1^0(t).$$

Finally, it is interesting to consider a point p attached to body frame Σ_1 , so that $\dot{p}^1 = 0$, and compute

$$\frac{d}{dt}p^0(t) = \Omega_{0,1}^0(t) \times p^0(t). \quad (9.2)$$

9.2 Linear and angular velocities

We now consider bodies that are not only rotating, but also translating. In other words, the origin of the body-fixed frame Σ_1 is moving with respect to the fixed spatial frame Σ_0 . In what follows, we drop the time argument for simplicity.

Lemma 9.5 (Linear and angular velocity). *Recall that we represent the location and orientation of Σ_1 with respect to Σ_0 by the displacement matrix*

$$H_1^0 = \begin{bmatrix} R_1^0 & O_1^0 \\ 0 & 1 \end{bmatrix}.$$

Recall the notions of angular velocity in spatial and body frames $\Omega_{0,1}^0$ and $\Omega_{0,1}^1$. Define the linear velocity of the body-frame origin by \dot{O}_1^0 . With these premises, we write the body velocity as a 4×4 matrix

$$\dot{H}_1^0 = \begin{bmatrix} \widehat{\Omega_{0,1}^0} & v_1^0 \\ 0_{1 \times 3} & 0 \end{bmatrix} H_1^0, \quad (9.3)$$

where $v_1^0 = \dot{O}_1^0 - \Omega_{0,1}^0 \times O_1^0$. Similarly, in the body frame,

$$\dot{H}_1^0 = H_1^0 \begin{bmatrix} \widehat{\Omega_{0,1}^1} & v_1^1 \\ 0_{1 \times 3} & 0 \end{bmatrix}, \quad (9.4)$$

where $v_1^1 = R_1^0 \dot{O}_1^0$.

Proof. It is a simple calculation to compute $\dot{H}_1^0(H_1^0)^{-1}$ and show that the equalities (9.3) and (9.4) holds true. ■

Finally, it is interesting to consider a point p attached to body frame Σ_1 , so that $\dot{p}^1 = 0$, and compute

$$\frac{d}{dt}P^0 = \dot{H}_1^0 P^1 = \begin{bmatrix} \widehat{\Omega_{0,1}^0} & v_1^0 \\ 0_{1 \times 3} & 0 \end{bmatrix} H_1^0 P^1 = \begin{bmatrix} \widehat{\Omega_{0,1}^0} & v_1^0 \\ 0_{1 \times 3} & 0 \end{bmatrix} P^0,$$

so that

$$\frac{d}{dt}p^0(t) = \Omega_{0,1}^0 \times p^0 + v_1^0 = \dot{O}_1^0 + \Omega_{0,1}^0 \times (p^0 - O_1^0).$$

This relationship is the velocity equation for a point fixed with the body and generalizes equation (9.2).

9.2.1 Basic velocities

Recall that we defined the three basic rotations in Section 6.3.3 and the six basic displacements in Section 8.2.1. We are now in a position to present the basic “independent” velocities of a rigid body.

First, let us consider the rotation of a rigid body and neglect its position. For a time-varying rotation matrix, the set of angular velocities (in either spatial or body frame) is the set of skew symmetric matrices $\mathfrak{so}(3)$.

Second, from the results in Lemma 9.5, we know that the set of linear and angular velocities of a rigid body is given by the following set of matrices:

$$\mathfrak{se}(3) = \left\{ \begin{bmatrix} S & v \\ 0_{3 \times 1} & 0 \end{bmatrix} \mid S \in \mathfrak{so}(3), v \in \mathbb{R}^3 \right\}.$$

Accordingly, there are 6 basic velocity “vectors” (just like 6 basic displacements): three infinitesimal rotations about the three axes and the infinitesimal translations along the three axes.

Next, it is convenient to introduce the operation $\widehat{\cdot} : \mathbb{R}^6 \rightarrow \mathfrak{se}(3)$ by

$$\widehat{\begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \\ v_x \\ v_y \\ v_z \end{bmatrix}} = \begin{bmatrix} 0 & -\omega_z & \omega_y & v_x \\ \omega_z & 0 & -\omega_x & v_y \\ -\omega_y & \omega_x & 0 & v_z \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

With this notation, the 6 basic velocities are then $\widehat{e}_1, \dots, \widehat{e}_6$, where

$$e_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad e_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad e_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad e_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad e_5 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad \text{and} \quad e_6 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

9.3 Vehicle motion models and integration

In this section we write down the differential equations that describe the motion of some simple robotic vehicle; these differential equations are usually referred to as the equations of motion of

the vehicle. Additionally, we study how to forward integrate these equations of motion so as to be able to simulate the vehicle motion and possibly solve motion planning problems for them.

Vehicle motion models Typical vehicles determine their motion by controlling their velocity in the body frame. Indeed, satellite thrusters, boat propellers and aircraft propellers are attached to the vehicle body. Let us present two examples of such systems.

First, consider a simplified model of fully controlled satellite described by

$$\dot{R}(t) = R(t)(\omega_1(t)\hat{e}_1 + \omega_2(t)\hat{e}_2 + \omega_3(t)\hat{e}_3),$$

where $\omega_1, \omega_2, \omega_3$ are the angular velocity controls. Second, consider a simplified model of an aircraft described by

$$\dot{H} = H(v_0(t)\hat{e}_4 + \omega_1(t)\hat{e}_1 + \omega_2(t)\hat{e}_2 + \omega_3(t)\hat{e}_3),$$

where v_0 is the forward speed and $\omega_1, \omega_2, \omega_3$ are the angular velocity controls (possibly about the roll, pitch and yaw axis respectively); see Figure 9.3.

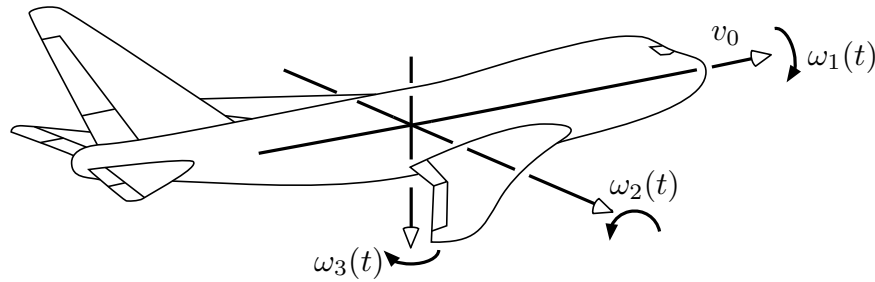


Figure 9.3: Angular velocities of a three-dimensional body

In what follows we aim to study how to numerically integrate the equations of motion just described. First, consider $\dot{R}(t) = R(t)\hat{\omega}$ with constant angular velocity $\omega \neq 0$. The rotation $R(t + \delta t)$ at time $t + \delta t$ is

$$\begin{aligned} R(t + \delta t) &= R(t) \text{Rot}_{\frac{\omega}{\|\omega\|}}(\delta t \|\omega\|) \\ &= R(t) \left(I_3 + \frac{\sin((\delta t)\|\omega\|)}{\|\omega\|} \hat{\omega} + \frac{1 - \cos((\delta t)\|\omega\|)}{\|\omega\|^2} \hat{\omega}^2 \right). \end{aligned}$$

Second, consider $\dot{H}(t) = H(t)\widehat{(\omega, v)}$ with constant angular and linear velocity. At time $t + \delta t$, we have

$$H(t + \delta t) = H(t) \begin{bmatrix} R & d \\ 1 & 0 \end{bmatrix},$$

where

$$R = I_3 + \frac{\sin((\delta t)\|\omega\|)}{\|\omega\|}\hat{\omega} + \frac{1 - \cos((\delta t)\|\omega\|)}{\|\omega\|^2}\hat{\omega}^2,$$

$$d = \left(I_3 + \frac{1 - \cos((\delta t)\|\omega\|)}{\|\omega\|^2}\hat{\omega} + \frac{\|\omega\|(\delta t) - \sin((\delta t)\|\omega\|)}{\|\omega\|^3}\hat{\omega}^2 \right)v.$$

9.4 Exercises

E9.1 A curve in 3D space (20 points).

Let $\text{Rotd}_n(\theta)$ denote the rotation by θ about an axis n and let $\text{Trans}_t(d)$ be the translation displacement by d along t . Define

$$v = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \text{and} \quad p = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}.$$

For each instant of time t , define $q(t)$ by

$$\begin{bmatrix} q(t) \\ 1 \end{bmatrix} = \text{Rotd}_v(t\pi) \text{Trans}_v(t) \begin{bmatrix} p \\ 1 \end{bmatrix}.$$

Do the following:

- (i) (5 points) write out the matrices $\text{Rotd}_v(t\pi)$ and $\text{Trans}_v(t)$,
- (ii) (10 points) find equations for the three components of $q(t)$, and
- (iii) (5 points) sketch the resulting curve for t in the interval $[0, 10]$.

E9.2 Frame changes (10 points).

Given an inertial fixed frame Σ_S , sketch the position and orientation of the frame Σ_M at time $t = 1$ if

$$H_M^S = \begin{pmatrix} \cos(\pi t/3) & 0 & \sin(\pi t/3) & 1 \\ 0 & 1 & 0 & -2t \\ -\sin(\pi t/3) & 0 & \cos(\pi t/3) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Next, assume a point p has components $p^S = (1, 2, 1)$ with respect to Σ_S . Compute

- (i) the components $p^M(t)$ of p with respect to Σ_M , and
- (ii) the velocity $\dot{p}^M(t)$ of p with respect to Σ_M .

Bibliography

- R. S. Allison, M. Eizenman, and B. S. K. Cheung. Combined head and eye tracking system for dynamic testing of the vestibular system. *IEEE Transactions on Biomedical Engineering*, 43(11): 1073–1082, 1996.
- S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper. USARSim: a robot simulator for research and education. In *IEEE Int. Conf. on Robotics and Automation*, pages 1400–1405, Roma, Italy, Apr. 2007.
- A. Cayley. On the Theory of Analytic Forms Called Trees. *Philosophical Magazine*, 13:19–30, 1857.
- H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, 2005. ISBN 0-262-03327-5.
- P. Corke. *Robotics, Vision And Control: Fundamental Algorithms In MATLAB*. Springer, 2011. ISBN 3642201431.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2 edition, 2001. ISBN 0262032937.
- J. J. Craig. *Introduction to Robotics: Mechanics and Control*. Prentice Hall, 3 edition, 2003. ISBN 0201543613.
- M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2 edition, 2000. ISBN 3540656200.
- P. Deheuvels. Strong bounds for multidimensional spacings. *Probability Theory and Related Fields*, 64(4):411–424, 1983.
- Z. Dodds. The bug algorithms. Lecture Slides for CS 154, Harvey Mudd College, 2006.

- G. Dudek and M. Jenkin. *Computational Principles of Mobile Robotics*. Cambridge University Press, 2 edition, 2010. ISBN 0521692121.
- Epic Games, Inc. Unreal Development Kit. <http://www.udk.com>, 2009.
- L. Euler. Solutio Problematis ad Geometriam Situs Pertinentis. *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, 8:128–140, 1741. Also in *Opera Omnia* (1), Vol. 7, 1-10.
- A. Fabri and S. Pion. CGAL: The computational geometry algorithms library. In *ACM Int. Conf. on Advances in Geographic Information Systems*, pages 538–539, Seattle, Washington, Nov. 2009.
- G. D. Hager. Algorithms for Sensor-Based Robotics. Lecture Slides for CS 336, Johns Hopkins University, 2006.
- J. H. Halton. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik*, 2:84–90, 1960.
- R. N. Jazar. *Theory of Applied Robotics: Kinematics, Dynamics, and Control*. Springer, 2 edition, 2010. ISBN 1441917497.
- A. Kelly. *Mobile Robotics: Mathematics, Models, and Methods*. Cambridge University Press, 2013. ISBN 110703115X.
- G. Kirchhoff. Über die Auflösung der Gleichungen, auf welche man bei der Untersuchung der linearen Verteilung galvanischer Ströme geführt wird. *Annalen der Physik und Chemie*, 148 (12):497–508, 1847.
- S. LaValle et al. The Motion Strategy Library, July 2003. URL <http://msl.cs.uiuc.edu/msl>. Version 2.0.
- S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006. ISBN 0521862051. Available at <http://planning.cs.uiuc.edu>.
- M. Lewis and J. Jacobson. Game engines in scientific research. *Communications of the ACM*, 45(1): 27–31, 2002.
- V. Lumelsky. *Sensing, Intelligence, Motion: How Robots and Humans Move in an Unstructured World*. John Wiley & Sons, 2006. ISBN 0471707406.
- 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.
- M. T. Mason. *Mechanics of Robotic Manipulation*. MIT Press, 2001. ISBN 0262133962.
- R. M. Murray, Z. X. Li, and S. S. Sastry. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, 1994. ISBN 0849379814.

- S. B. Niku. *Introduction to Robotics: Analysis, Control, Applications*. John Wiley & Sons, 2 edition, 2010. ISBN 0470604468.
- J. Pan, S. Chitta, and D. Manocha. FCL: a general purpose library for collision and proximity queries. In *IEEE Int. Conf. on Robotics and Automation*, pages 3859–3866, Saint Paul, MN, USA, May 2012.
- J. M. Selig. *Geometrical Methods in Robotics*. Springer, 1996. ISBN 0387947280.
- B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics: Modelling, Planning and Control*. Advanced Textbooks in Control and Signal Processing. Springer, 2009. ISBN 1846286417.
- R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, 2 edition, 2011. ISBN 0262015358.
- J. G. Siek, L.-Q. Lee, and A. Lumsdaine. Boost Graph Library. <http://www.boost.org>, July 2007. Version 1.34.1.
- M. W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. John Wiley & Sons, 3 edition, 2006. ISBN 0-471-64990-2.
- I. A. Şucan, M. Moll, and L. E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012. <http://ompl.kavrakilab.org>.
- A. G. Sukharev. Optimal strategies of the search for an extremum. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 11(4):910–924, 1971. Translated from Russian, *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*.
- K. Taylor and S. M. LaValle. I-bug: An intensity-based bug algorithm. In *IEEE Int. Conf. on Robotics and Automation*, pages 3981–3986, Kobe, Japan, May 2009.
- The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 2015. URL <http://doc.cgal.org>.
- S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005. ISBN 0262201623.