Flying Autonomous Robots

CSE 360/460 Introduction to Mobile Robotics

Author: David Saldaña

Institute: Lehigh University

Date: November 10, 2024

Contents

Feedback and comments are very welcome! Please e-mail `saldana@lehigh.edu`.

Notation

In the literature, there are many ways to write mathematical symbols to represent robotic system. In order to keep a consistent notation along the course, these notes follow a traditional mathematical notation in engineering:

- **Scalar values** are denoted by a lowercase letter. Examples: height h, weight w.
- **Vectors** are n-dimensional variables that are denoted by a lowercase letter in bold. For example, a point on a plane $\boldsymbol{p}$. The point has x- and y-coordinates, denoted by scalar variables, so we can express the vector as:

  $$\boldsymbol{p} = \begin{bmatrix} x \\ y \end{bmatrix}.$$

  The denote the transpose with the super-index $\top$. So the transposed of the point $\boldsymbol{p}$ will be:

  $$\boldsymbol{p}^\top = \begin{bmatrix} x\, y \end{bmatrix}.$$

- **Matrices** are $n \times m$-variables, denoted by a capital letter in bold. Fr example, a 3×3 rotation matrix is denoted by

  $$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix},$$

  where the elements of the matrix, r_{ij}, are scalar values.
- **Functions:** Depending on the range of the function, the notation will be simular to vectors and matrices. For example, a trajectory function maps time to a three dimensional space, so we will denote the function simular to a vector, in this case, the trajectory is denoted by $\boldsymbol{\gamma}(t)$.

Chapter 1 Introduction

Concepts

- ❏ *Mobile Robot*
- ❏ *Perception*
- ❏ *Actuation*
- ❏ *Computation*

In an era marked by remarkable technological advancements, we find ourselves surrounded by an array of autonomous systems that are reshaping our world. From self-driving cars navigating the streets to cleaning robots maintaining our homes, from robot servers enhancing the dining experience to automated logistics in warehouses, and from autonomous drones soaring the skies for various applications - these advancements are not just futuristic concepts but present-day realities.

This book is dedicated to two primary objectives: Firstly, to build a foundational understanding of the concepts and technologies that underpin these advanced machines. And secondly, to critically analyze current solutions, identifying opportunities for innovation and improvement. Our exploration is not just about comprehending what exists but also about envisioning and creating what could be.

Our journey into the world of mobile robotics begins with a simple yet powerful mathematical abstraction: envisioning a robot as a point on a plane. This abstraction is our stepping stone into the complex realm of robotics. From here, we expand our scope to include rigid bodies and explore diverse models that describe their motion and control. This progression from simplicity to complexity is designed to provide a comprehensive understanding of the principles of robot dynamics and control.

Before delving deeper into these topics, it's crucial to familiarize ourselves with the main components that constitute a robot. Understanding these components is akin to understanding the building blocks of life - it provides insight into how robots function, interact with their environment, and accomplish their tasks. This overview will not only set the stage for our subsequent discussions but also enrich your perspective on the role and capabilities of robots in our rapidly evolving technological landscape.

In this book, we are going to study autonomous electromechanical machines that can move in an environment. Our fist definition is the following.

Definition 1.1. Mobile Robot

A mobile robot *is an autonomous machine capable of perceiving its environment, making decisions, and executing actions.* ♣

According to this definition, we identify three key components essential for understanding mobile robots. As illustrated in Figure 1.1, these main components are perception, computation,

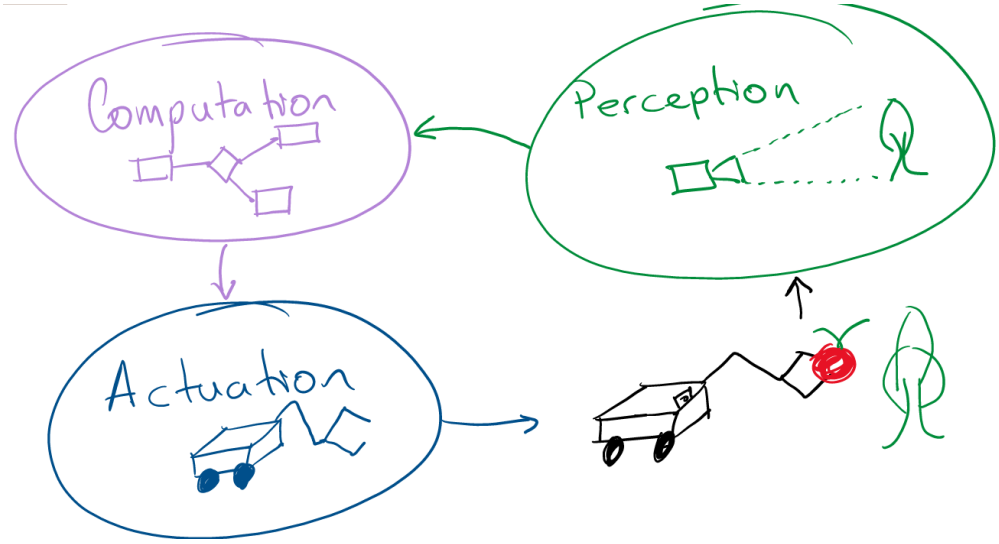


Figure 1.1: Main components of a mobile robot.

and actuation.

1.1 Perception

Consider the task of a robot picking an apple from a tree. The initial step involves recognizing the tree and subsequently pinpointing the apple. This process is governed by the robot's perception system, which is an assembly of various sensors.

Definition 1.2. Sensor

A sensor is an instrument that equips a robot with the ability to 'perceive' or measure aspects of its environment. It captures physical variables, converting them into data that the robot can process.

♣

The initial output from a sensor is often referred to as raw data, representing unprocessed numerical values. For instance, a thermometer might give a mere number, which must then be converted into a familiar unit, such as Celsius or Fahrenheit, for meaningful interpretation. Further, this data requires semantic analysis for instance, determining whether the environment is hot or cold from the thermometer's reading.

A similar principle applies to a camera sensor. Here, incoming light activates numerous semiconductors, generating raw data. However, both autonomous robots and humans seek more than raw numbers; we seek to extract meaningful information from the imagery, such as the presence and identification of humans, cars, houses, pets, or other objects within a scene. This conversion of raw data into actionable information, often through either human or algorithmic interpretation, constitutes the essence of a perception system.

A perception system thus includes all the sensors and the interpretative software that processes their outputs. In the realm of mobile robotics, some commonly employed sensors are cameras, radars, odometers, laser scanners (LIDAR), and inertial measurement units (IMU).

While this list is extensive, we will explore many of these sensors in detail throughout this book. Part of our journey will also involve defining these sensors and discovering additional ones that can be applied in mobile robotics.

1.2 Actuation

Robots are designed to perform a variety of physical tasks, such as grasping objects, navigating through buildings, and moving or manipulating obstacles. To accomplish these tasks, they are equipped with actuators.

Definition 1.3. Actuator

An actuator serves as the functional component in a robot, enabling it to interact with and exert force upon its environment.

♣

Consider a robotic arm, which typically comprises multiple servo motors enabling object manipulation. However, this arm alone lacks the capacity to autonomously traverse an environment. Mobile robots, therefore, incorporate at least one actuator specifically designed to facilitate movement within their operating spaces.

Wheels are the most prevalent form of actuator in ground-based robots, providing a reliable and efficient means of locomotion. Aerial and aquatic robots often employ propellers to navigate through their respective environments, while aircraft may use jet propulsion for movement. This book will explore a range of actuators, broadening our understanding of how robots interact with and adapt to different environments. Identifying and learning about various types of actuators forms an integral part of our exploratory exercises.

1.3 Computation

At the heart of an autonomous robot's functionality lies the intricate world of control algorithms the unseen yet pivotal "magic" that orchestrates a robot's actions. These sophisticated algorithms are the bridge between perception and action. They intelligently process the data gathered by the perception system, integrate it with pre-existing knowledge (a-priori information), and determine the most effective commands for the actuators.

Our journey in this book involves a deep dive into these fundamental algorithms that govern the control of mobile robots. We'll start by setting specific objectives, such as navigating to a designated location within an environment. Then, we'll explore the design and implementation of algorithms capable of accomplishing these goals.

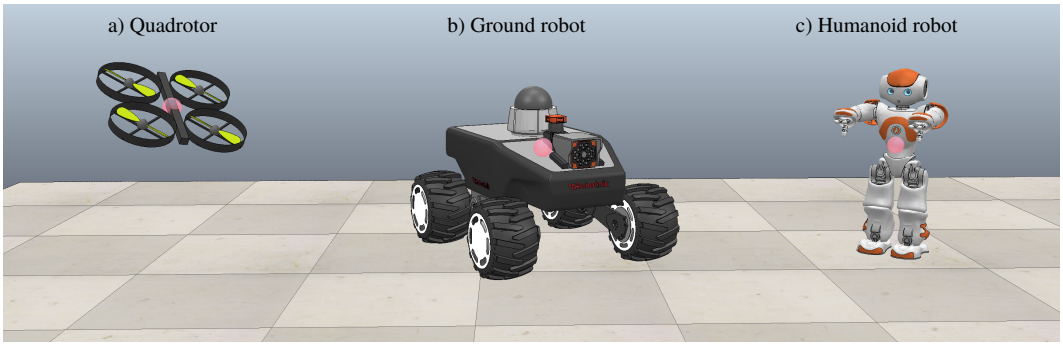


Figure 1.2: Examples of mobile robots.

1.4 Types of Mobile robots

Mobile robots exhibit a remarkable diversity, with various categorization approaches based on their operating environments, applications, and other factors. One fundamental way to classify them is by their medium of movement: ground, aerial, aquatic, or hybrid.

Ground robots, particularly wheeled types, are prevalent due to their high load-carrying capacity. However, the world of ground robots is rich and varied, featuring mechanisms that emulate the sliding of snakes, the hopping of kangaroos, or the bipedal walking of humans. This diversity leads to a range of sub-categories in ground robots, each distinguished by its unique locomotion method.

Aerial robots, such as drones and fixed-wing planes, offer the advantage of speed and high-altitude navigation, allowing them to bypass obstacles efficiently. However, their operational energy demands often limit their flight duration and payload capacity. Drones, often characterized by a quadcopter design, and fixed-wing planes, which maneuver by tilting their wings, represent two common forms of aerial robots. Fixed-wing vehicles are generally more energy-efficient than propeller-only vehicles (like the difference between airplanes and helicopters), but they are constrained in their agility and ability to execute sharp turns.

Aquatic robots operate either on water surfaces, akin to ships, or underwater, like submarines. Their propulsion often mirrors that of aerial vehicles, utilizing propellers to push water rather than air.

Figure 1.2 showcases three distinct examples of mobile robots: a) a drone, b) a wheeled robot, and c) a humanoid robot. Despite their vastly different locomotion systems, a key focus of this book will be to explore methodologies for abstracting and controlling the critical variables of these varied forms of mobile robots.

1.5 Robot modeling and the point-robot

In mathematics, geometrical elements have been deeply studied, and their properties are well defined. Specifically, in Calculus is studied how functions change in time. We can define the trajectory of an object as a function that changes in time, and once we have the function, we

can apply all the methods from calculus to understand the phenomenon. Our work is to use fundamental theory to abstract, model, and control mobile robots. The real world has infinite variables that we can model, but we do not need to study all of them for most practical applications. For example, if we want to control the velocity of a car, we know that if we push the gas, the engine is going to cause gasoline to explode, and move the pistons; the motion is then transmitted to the wheels using a complex gear system. In this process, there are many variables involved, like the quality of the gasoline, how much explodes, the energy lost for the friction in the gears, and so on. But the problem is much simpler, and we don't need to model the full engine. We can focus on what is essential, we need to know the location of the car, and how that location changes in time. Then, all the complexity of the vehicle can be abstracted into a few variables.

In this book, we will start with many simplifications that allow us to control robots under certain circumstances or in a perfect world, and step by step, we will make it more realistic. Let's start with the robots in Figure 1.2; although all of them have different ways to move, all of them are mobile robots. A simple problem, like moving from point A to point B, will depend on many variables. We know that classical robots are composed of links and joints. So if we move a single point, the whole body will be connected to it. Think about moving a chair by pulling only one leg - the entire chair will move as well. So the problem of moving the chair can be transformed into just moving its leg. In order to move the leg, you might use a single finger to push it, so the problem is even simpler because we only need to move the point of contact between the finger and the leg. Finally, our motion problem is transformed into just moving a single point. As illustrated in Figure 1.2 by the pink dot (actually looks like a sphere, but we made it bigger for easier visualization), we will focus on moving that point.

1.6 Further reading

A better introduction to actuators, sensors and mobile robots can be found in (Siegwart et al., 2011). A specific book about robot motion, without entering into specific details is (Choset et al., 2005). A more advanced book for understanding mechanical systems and robot control is (Lynch and Park, 2017). For the final part for this book, localization and mapping, we will use (Thrun et al., 2005). Different from previous books, the following chapters will present a gentle introduction to the fundamental concepts of mobile robotics by focusing on the point-robot.

Chapter 1 Exercise

1. For the following sensors: thermometers, cameras, radars, odometers, laser scanners (LIDAR), and inertial measurement units (IMU), what is the physical variable that they measure? How do they work? What kind of computer data type can be used to store their values?
2. Different from the sensors described in this chapter, list five additional sensors that can be used in a mobile robot.
3. Different from the actuators described in this chapter, list five additional actuators that can be used in a mobile robot.
4. Find three different types of wheels that can be used in a robot. Each type should have a different property that impacts the way a robot moves.
5. Find other ways to categorize robots based on their application.
6. Find three aerial robots, three ground robots, and three water robots. What point would you choose to represent it, and why? (By the end of this book, you might change your answers.)

Chapter 2 The Point-Robot

Concepts

- ❏ *Point-robot*
- ❏ *Robot motion*
- ❏ *Velocity (First order ODE)*
- ❏ *Proportional control*
- ❏ *Position control*
- ❏ *Open- and closed-loop control*

In the field of mobile robotics, the concept of ***location*** emerges as a critical aspect. Unlike stationary robots, such as those in industrial settings, mobile robots operate in diverse environments where understanding their position and destination is vital for successful task execution. This chapter delves into the significance of location in mobile robotics, setting it apart from static robotic systems where position does not change.

The cornerstone of mobile robotics is effective navigation. To illustrate, consider the task of package delivery: a mobile robot must accurately identify both the pick-up and delivery locations. This process necessitates the establishment of a ***reference frame***, a fundamental step in spatial orientation. A prominent landmark can serve as the origin of this frame, allowing the robot to orient itself and navigate within the mapped environment. The movement patterns of these robots can be compared to those of cars, navigating streets within defined parameters.

While a mobile robot's physical characteristics such as color, weight, size, and joint configuration play a role, our primary focus here is on understanding and determining ***location***. To simplify complex real-world scenarios, we initially conceptualize the robot as an infinitesimal point within a reference frame, stripping away attributes like mass, orientation, and external forces like gravity and friction. This abstraction allows us to concentrate on the essence of navigation and positioning. For instance, the Global Positioning System (GPS) used in cars provides a real-world application of this concept. It offers a location estimation within a global coordinate framework, much like how a car's position is depicted as a mere point on a map application. This chapter will explore how such simplifications are not just theoretical constructs but practical tools for understanding and designing mobile robotic systems, emphasizing the central role of location in their functionality.

On the ground or any planar surface, we can abstract the environment as a plane, represented by the ***planar or two-dimensional Euclidean space*** $\mathbb{R}^2$. We denote the location of the robot by $\boldsymbol{p} \in \mathbb{R}^2$ and its $xy-$coordinates by

$$\boldsymbol{p} = \begin{bmatrix} x \\ y \end{bmatrix}. \tag{2.1}$$

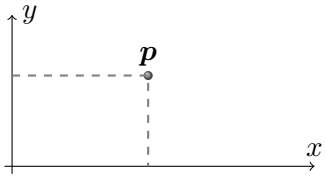


Figure 2.1: Point-robot in 2-D.

The location of the robot in the 2-D coordinate system is illustrated in Figure 2.1. In the case of a flying drone or an airplane, the vehicle can move upwards and downwards. So we need

to take into account an additional variable which is the altitude. Our abstraction is now the ***3-dimensional space*** $\mathbb{R}^3$ and the robot location has three components,

$$\boldsymbol{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}. \tag{2.2}$$

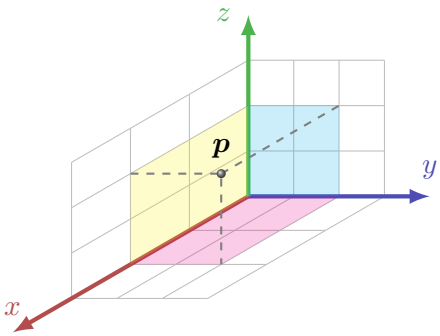


Figure 2.2: Robot point in 3-D.

The location of the point-robot in the 3-D coordinate system is illustrated in Figure 2.2. Even complicated robotic systems tend to be simplified to point robots for convenience. For instance, a drone in the sky monitoring a land field does not need to worry about obstacles. If it moves at the same altitude, it can be modeled as a point on a plane. We will study how to control a drone as a moving point. Later we will discuss other spaces like the configuration space and the task space. For now, let's assume that the robot is a point in an open space without obstacles.

Definition 2.1. Point-robot

A point-robot *is mass-less and infinitesimally small point, represented by its location at a point in the Euclidean space* $\mathbb{R}^d$ *of dimension* $d \in \{1, 2, 3\}$. *We denote its location at time* t *by* $\boldsymbol{p}(t)$.

In physics, the notion of a particle bears resemblance to our concept of a point-robot, with a key distinction: our point-robot has the capability to control its motion. This aspect of controllable movement will be elaborated upon in the subsequent sections.

2.1 Robot motion

A mobile robot is equipped with actuators that allow it to move in the environment; its motion is described by the way it changes its location in time. In this way, our location vector is now a ***parametric function*** $\boldsymbol{p}(t)$ that depends on time $t \in \mathbb{R}_{\geq 0}$. Here, time is a continuously increasing positive variable that can influence the robot's location.

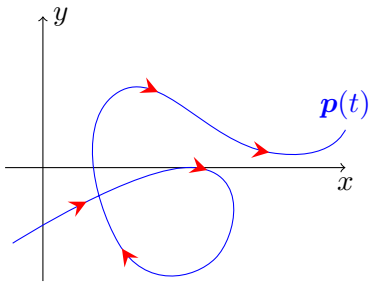


Figure 2.3: Robot trajectory described by a parametric function.

The concept of ***motion*** is related to how the robot alters its location in time. The chance of the location in time is called ***velocity***. In the same way, the change of the velocity in time is called ***acceleration***. We will start studying the robot's motion by controlling its velocity. We denote the derivative of the location using Newton's notation, which places a dot over the mathematical

symbol of the location to represent its derivative with respect to time,

$$\dot{\boldsymbol{p}}(t) = \frac{d\boldsymbol{p}(t)}{dt}. \tag{2.3}$$

Assuming that the robot can move in any direction, we can control the robot's velocity

$$\dot{\boldsymbol{p}}(t) = \boldsymbol{u}(t). \tag{2.4}$$

Let's highlight that the velocity is a vector that can be decomposed into two parts: a unitarian vector that describes the direction of motion, and its magnitude, a scalar value that describes how fast the robot is moving. The magnitude of the velocity is called ***speed***.

Control: A ***control policy*** is a function $\boldsymbol{f}$ that uses the available information of the robot to determine the control input of the robot $\boldsymbol{u}$. The information can be time, position, velocity, acceleration, obstacles, or even information from external observers (other robots or a base station). For instance, we can define a control policy (time-variant) that uses the location of the robot:

$$\boldsymbol{u}(t) = \boldsymbol{f}(t, \boldsymbol{p}). \tag{2.5}$$

Since a robot is represented as a point that can move in the space described by an ordinary differential equation (***ODE***) that is determined by the control policy. Making (2.4) equal to (2.5), we get the equation that describes the robot motion beased on its control policy $\boldsymbol{f}$,

$$\dot{\boldsymbol{p}}(t) = \boldsymbol{f}(t, \boldsymbol{p}). \tag{2.6}$$

Given the initial position of the robot at an arbitrary time, $\boldsymbol{p}(t_0)$, and the control policy $\boldsymbol{f}$, we can obtain the location of the robot at a time t_f. The solution for $\boldsymbol{p}$ can be obtained by solving the ordinary differential equation (ODE) in (2.6). However, in some cases, we might not know the function $\boldsymbol{f}$ for each time step and each robot location, the most common situation is where the robot only knows $\boldsymbol{f}$ at the current time step and location. The challenge in control theory is determining a control policy $\boldsymbol{f}$ that uses the available information in order to make the robot to achieve its ***goal***. In our case, a goal is related to motion, so we might want to drive the robot to a specific point; follow a specific trajectory; follow a moving object; or any desired behavior that involves motion.

We know that we can move our robot by applying a control input $\boldsymbol{u}(t)$; so we control the output $\boldsymbol{p}(t)$ based on the input $\boldsymbol{u}(t)$. We illustrate the control sequence using the block diagram in Figure 2.7. Our Controller, represented by a green block, defines what we want to do with the robot. It uses our desired trajectory (like a circle or a straight line), and the time to create a control function $\boldsymbol{u}(t)$ that goes to the robot, illustrated by the blue block. For now, we will assume that there are no disturbances, but we will deal with them later.

2.2 Moving the point-robot

We can define control policies that depend only on time. We illustrate some cases in the following examples.

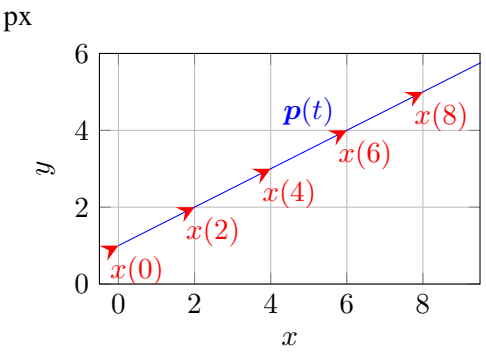


Figure 2.4: Straight-line motion

Example 2.1: Straight-line motion A point robot is at location $\boldsymbol{p}(0) = [0, 1]^\top$ and its control input is

$$\boldsymbol{u}(t) = \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

What is the trajectory of the robot $\boldsymbol{p}(t)$?

Since we know that $\dot{\boldsymbol{p}} = \boldsymbol{u}$, we can obtain $\boldsymbol{p}(t)$ by solving the differential equation $\dot{\boldsymbol{p}}(t) = [2, 1]^\top$ with initial condition $\boldsymbol{p}(0) = [0, 1]^\top$. The first-order differential equation can be solved by using integration

$$\int \dot{\boldsymbol{p}}\, dt = \begin{bmatrix} 2t \\ t \end{bmatrix} + \mathbf{c}.$$

Using the initial condition, we obtain that $\mathbf{c} = [0, 1]^\top$. See Figure 1 for illustration of the initial location, depicted as a black disk. We can observe that the linear velocity keeps constant as it is imposed by the control input.

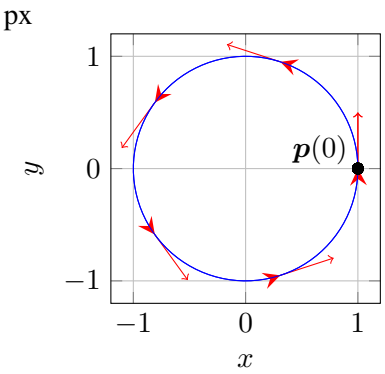


Figure 2.5: Circular motion

Example 2.2: circular motion A point robot starts at location $\boldsymbol{p}(0) = [1, 0]^\top$ and moves based on the control policy $\boldsymbol{u} = [-\sin(t), \cos(t)]^\top$. What is the trajectory of the robot $\boldsymbol{p}(t)$?

Similar to Example 1, we can integrate $\dot{\boldsymbol{p}}(t)$ to obtain the equation of a circle

$$\int \dot{\boldsymbol{p}}(t)\, dt = \begin{bmatrix} \cos t \\ \sin t \end{bmatrix}.$$

The illustration of the circle is in Figure 2.5. In this case, we plot the control input $\boldsymbol{u}(t)$ at some points in time. We can see that the derivative of $\boldsymbol{p}(t)$ is always tangent to the circle.

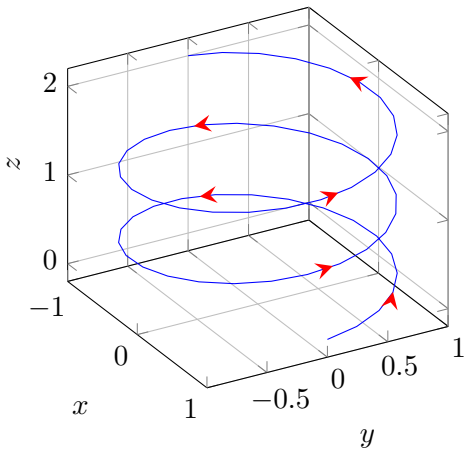


Figure 2.6: Helicoidal motion

Example 2.3: Helicoidal motion Now we want to move a point in 3-D, we have an idea of the trajectory curve to follow, and then we compute its derivative. The best part of the point robot is that we can completely model it analytically and play with analytic functions directly.

In Examples 1 and 2, we had the control input, and we wanted to know where the robot went. In this case, we know that the robot should follow a helix curve, so we will start with the equations of the helix, as shown in Figure 2.6,

$$\boldsymbol{h}(t) = \begin{bmatrix} \cos t \\ \sin t \\ \frac{2}{5\pi} t \end{bmatrix}. \tag{2.7}$$

In this case, the control input is $\boldsymbol{u} = [\cos t, \sin t, 2/(5\pi)]^\top$, making the robot moving in helix curve. However, the helix might be translated depending on the initial location of the robot. For example, if the robot starts at location $\boldsymbol{p}(0) = [0, 0, 2]^\top$. the resultant trajectory would be,

$$\boldsymbol{p}(t) = \boldsymbol{h}(t) + [-1, 0, 2]^\top.$$

Verifying this resultant trajectory will be left as an exercise. In the next chapter, we will learn how to obtain analytic functions to pass through a set of waypoints.

Open-loop control

In an ideal or controlled environment, such as within a computer simulation, we might expect a robot to execute tasks precisely as programmed. However, the complexity of the real world, with its myriad of internal and external factors, often complicates this expectation. From an internal perspective, robots are constructed using mechanical components that, despite best efforts, cannot be manufactured flawlessly. Motors, for example, may exhibit minor fabrication and calibration discrepancies, introducing inherent inaccuracies. Externally, the robot must contend with environmental challenges like uneven terrains, wind forces, or interactions with other entities, all of which can be collectively referred to as disturbances, as depicted in Figure 2.7.

The control input, $\boldsymbol{u}(t)$, this necessitates accounting for a small error term in the motion equation to acknowledge these disturbances. The primary concern here is the cumulative nature of these errors; they tend to aggregate over time, leading to increasingly significant deviations from the intended path the longer the robot operates. In the scenarios described, the control strategy is open-loop, relying solely on predetermined velocity inputs without considering the actual motion or environmental feedback. While this approach might seem overly simplistic and idealistic, it offers a straightforward solution that can be quite effective in controlled situations or when the robot's operation time is brief and within well-understood environments.

To mitigate the limitations of open-loop control, incorporating ***feedback*** from the robot's sensory and perception systems presents a viable alternative. This approach enables the formu-

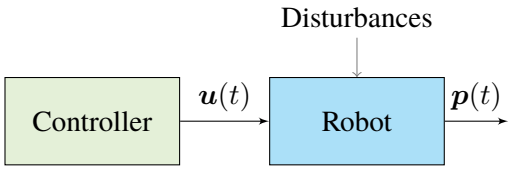


Figure 2.7: Block diagram of a robot controller.

lation of a control policy that not only adheres to the desired trajectory but also adapts based on real-time sensory feedback. This adaptive strategy, known as ***feedback control***, offers a more resilient and accurate means of navigating the complexities of real-world environments, which we will explore in greater detail in the subsequent section.

2.3 Position control: Moving to a point

Lets study the case where the control policy does not only depend on time, but also the location of the robot.

2.3.1 Moving to the origin

For instance, the following control policy only deepens on the location of the robot

$$\boldsymbol{u}(t) = -\boldsymbol{p}(t). \tag{2.8}$$

Computing the magnitude of the velocity vector [1], also called ***speed***, we get that the it is proportional to the distance to the origin of the coordinate system.

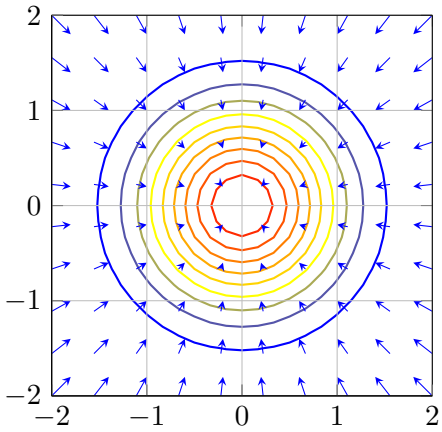


Figure 2.8: Control policy for location control. Any location on the plane will define a velocity vector that will drive the robot to the origin.

In Figure 2.8, we illustrate the velocity vector by the arrows pointing to the origin. The speed is represented by the level-curve of the speed. In this way, we can intuitevely see that if a robot follows the control policy in (2.8), it will move straight in direction to the origin and slow down when approaching it. The final result is independent of the initial condition $\boldsymbol{p}(0)$.

In this equation, the speed is equal to the distance to the origin, but we can make it faster. Lets multiple the location by a proportional constant $K > 0$ to the equation, so we get

$$\boldsymbol{u}(t) = -K\boldsymbol{p}(t). \tag{2.9}$$

In this way, we can either speed up or slow down the robot's motion towards the origin by setting the constant parameter K. We can also make K a diagonal matrix if we want different convergence rates for x and y, but let's keep it simple for now.

Intuitively, we know that (2.9) drives the robot to the origin, but let's verify it by solving the differential equation. Since K is a scalar value, we can solve the first-order ordinary differential equation for x and y independently[2], and the result is

$$\boldsymbol{p}(t) = e^{-Kt}\boldsymbol{p}(0).$$

From this trajectory, we can see that the robot converges exponentially to the origin, and its convergence rate is given by K, which matches our intuition.

[1] Recalling that the magnitud of the vector $\dot{\boldsymbol{p}}$ is $\|\dot{\boldsymbol{p}}\| = \sqrt{x^2 + y^2}$ for the 2-D case, and $\|\dot{\boldsymbol{p}}\| = \sqrt{x^2 + y^2 + z^2}$ for the 3-D case.

[2] The procedure to solve this differential equation can be found in any calculus book, also called exponential solution.

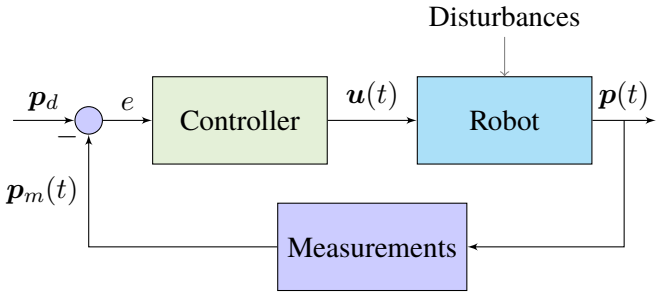


Figure 2.9: Block diagram of closed-loop control

This method works great and fast, but we need to consider that sometimes our robot has some limitations related to the hardware. For example, motors have a maximum speed that they can rotate. The hardware configuration can also limit the maximum velocity that a robot can move in a specific direction. We will talk about ***kinematic constraints*** and ***saturation*** that in later chapters.

2.3.2 Proportional control

Now, we can try to move the robot to a point different from the origin. Lets say that we want to drive the robot towards a desired point $\boldsymbol{p}_d$. In this case, we can just translate the origin to $\boldsymbol{p}_d$. Lets define a new location

$$\hat{\boldsymbol{p}}(t) := \boldsymbol{p}(t) - \boldsymbol{p}_d, \tag{2.10}$$

in this way, when the robot is in $\hat{\boldsymbol{p}}(t) = \mathbf{0}$, the robot in the original frame will be $\boldsymbol{p}(t) = \boldsymbol{p}_d$. By computing the derivative of (2.10) and (2.4), we can get that

$$\hat{\boldsymbol{u}}(t) = \boldsymbol{u}(t).$$

So the control input is the same after translating the origin. This means that we can use the same control policy to drive the robot to the origin in (2.9), but using $\hat{\boldsymbol{p}}(t)$. By replacing $\hat{\boldsymbol{p}}(t)$ from (2.10) into (2.9), we get what is called in the literature as a

Proportional controller or P-controller:

$$\boldsymbol{u}(t) = K(\boldsymbol{p}_d - \boldsymbol{p}(t)). \tag{2.11}$$

In this way, the controller drives the robot from any initial point $\boldsymbol{p}(0)$ to a desired point $\boldsymbol{p}_d$. We can verify it by solving the differential equation to obtain

$$\boldsymbol{p}(t) = e^{-Kt}(\boldsymbol{p}(0) - \boldsymbol{p}_d) + \boldsymbol{p}_d. \tag{2.12}$$

Example 2.4: P-controller A point robot starts at location $\boldsymbol{p}(0) = [4, 2]$ and wants to go to the point $\boldsymbol{p}_d = [1, 1.5]$. The robots knows its location at each time step. What is a suitable control policy for this objective?

Since we know the desired location $\boldsymbol{p}_d$ and the location of the robot at each time step $\boldsymbol{p}(t)$, so we can apply the P-controller of (2.11) as an effective control policy.

2.4 Coding the point-robot

The best part of the the point robot is its simplicity, and as we will study in this section, it is also simple to simulate and interact with. For simulated or actual robots, we basically convert the block diagram from Figure 2.9 into code.

We start having the initial location of the robot $\boldsymbol{p}(0)$, defining its control policy $\boldsymbol{u}(t)$, and assuming that we can obtain the location of the robot using a perfect sensor $\boldsymbol{y}(t) = \boldsymbol{p}(t)$. Then, we can write a simulator in a few lines of code, see Code 2.1. We basically convert the block diagram from Figure 2.9 into a programming language. See Code 2.1 for the implementation in Python. The full python script can be found here[3].

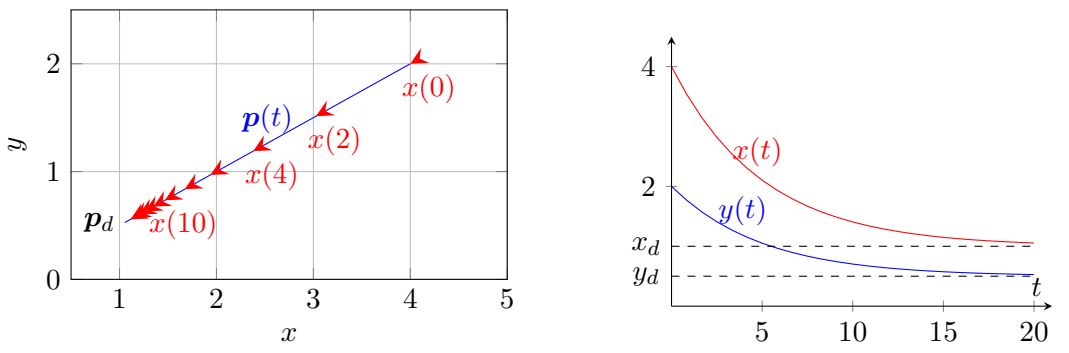
```
4 x = array([0.,0.])            # Initial condition
5 for t in time:
6    y = sense(x)               # Simulate sensors
7    u = control(t, y)          # Control policy
8    x = simulate(Δt, x, u)     # Dynamics
```

Code fragment 2.1: Simulating a point robot. Implementation of the block diagram of Figure 2.9.

In the simulation program, our robot starts at an arbitrary location, defined in Line 4. Then we have a simulation loop that runs for a finite number of time steps (Lines 5-8). Inside the loop, we can see the same blocks of the block diagram, sensing, control, and either simulation or robot. The essential parts to build a simulator are the following.

State: In this chapter, the state vector represents the robot's position. This vector is the focal point of our simulation and analysis, defining the key variable that we aim to understand and

[3]GitHub: https://github.com/dsaldana/CSE360-MobileRobotics/blob/master/point-robot-simulator.ipynb



(a) Motion on the plane

(b) Convergence of the functions $x(t)$ and $y(t)$.

Figure 2.10: Point robot moving towards the point $\boldsymbol{p}_d$.

manipulate.

Time: Our simulation will be executed in a time interval $[0, t_f]$. Since computers do not work in continuous space, what we do is discretizing the time interval. We create a finite number of time steps where the difference between a pair of subsequent time steps is (Δt). So the time steps will be $t = 0,\ \Delta t,\ 2\Delta t,\ ..., t_f$, simply $t_{i+1} = t_i + \Delta t$ for every $i = 1, 2, ...$ We can define a very small value for Δt such that the simulation seems continuous. However, too small values will require higher computational resources and slow down the simulation.

In our implementation, we can give an example and defining a final time $t_f = 6.0$ and $\Delta t = 0.1$. Then we will iterate through the time steps $t = 0, 0.1, 0.2, ..., 6.0$. The code to create an array of values for the time steps is in Code 2.2.

```
1 tf = 10       # Final time
2 Δt = 0.1   # Time step
3 time = linspace(0, tf, tf / Δt + 1)  # Time interval
```

Code fragment 2.2: Simulation sttings for a finite time interval.

Sense: Our robot can observe the world and itself using sensors. In this case, we assume that the robot can sense its location, so at any time step, the sensor reading $\boldsymbol{y}(t)$ measures the robot location $\boldsymbol{y}(t) = \boldsymbol{p}(t)$. Therefore, in the implementation (Line 6 of Code 2.1), the function `sense` receives `x` and returns `x`. This function might seem silly, but it is because the code is general. Later we can add noise to the measurements or reduce the number of variables that we can observe.

Control policy: The control policy is defined in the `control` function (Line 7 of Code 2.1). Here is where the action is! We can define any control policy that we want in this function. An example of a control policy is the following.

```
1 def control(t, y):
2    ux = -sin(t)
3    uy = cos(t)
4    return array([ux, uy])
```

Code fragment 2.3: Control policy to make the robot moving in circles.

Simulation by integration: If we know the robot location $\boldsymbol{p}(t)$, control policy $\boldsymbol{u}(t)$ and the time interval for the time step $[t, t + \Delta t]$. Then we can know the location at the next time step $\boldsymbol{p}(t + \Delta t)$ by integrating, analytically or numerically. Analytical integration works for some mathematical functions, but not all the functions can be integrated. In contrast, numerical integration can be applied to any function. We can use a numerical method like integration by rectangles (Euler method) or the Runge-Kutta method (Mathews et al., 2004). On the one hand,

the integration by rectangles is straightforward and runs fast, but it is not accurate. On the other hand, the Runge-Kutta method requires multiple iterations that make it slower, but it offers high accuracy. Code 2.4 shows the implementation of the rectangular integration. As we can see, the implementation is very straightforward is the computation is very fast since it only has a couple of arithmetic operations. Line 2 is the implementation of Equation 2.4, where we define that the control input is equal to the robot velocity.

```
1 def simulate(Δt, x, u):
2    dx = array(u)  # The control input is equal to the velocity of the robot
3    x += Δt * dx # Euler integration
4 return x
```

Code fragment 2.4: Simulation by rectangular integration.

Robot visualization: Visualizing the behavior of the robot is important because we can analyze how the robot is executing the control policy. We can track the x and y coordinates in the variables `x`, or we can also plot the point robot on the plain. Plotting a point is very simple, we can just call the command `plot([x[1]], [x[2]], 'r.');`, and that is it. Later we will make more elaborated drawings for the robots.

External simulator: The advantage of using our simulator is that we control everything, and we can tailor it based on our needs. However, tailoring takes time to develop and test. Many robot simulators are available online and ready to use. Some of the most popular simulators are Gazebo (Koenig and Howard, 2013), player-stage, CoppeliaSim, Mujoco, and Webots. Gazebo is one of the most popular simulators because it is supported and maintained by the Open Source Robotics Foundation (OSRF). It is widely used in research labs, but it not the most user-friendly. In this course, we will use our point-robot simulator to understand and develop the robotics techniques from scratch, and then we will test them in the simulator CoppeliaSim.

Communication with an actual robot: For real robots, we don't need to worry about integrating or drawing a robot. We need to change the lines 6 and 8 of the Code 2.1. Although we do not know the variable `x` with the robot's actual location, we can use the sensors. So in line 6, the function sense will read from the sensors. For line 8, we do not need to integrate the model kinematics model of the robot; we need to send the command to the robot.

2.5 Further reading

Other authors study the motion of point in detail. We recommend the book (O'Reilly, 2020), they call it as *kinematics of a particle*. Another book with implementations in Python is (Lynch, 2018).

Chapter 2 Exercise

1. For a point robot in 2-D, find a control policy $\boldsymbol{u}$ to make the robot move in an ellipse shape. The ellipse's major axis is equal to 4m, and the minor axis is equal to 2m. The major axis of the ellipse is aligned with the x-axis of the world frame. The center of the ellipse is in the point $[3, 2]^\top$. The ellipse is illustrated in Figure 2.13.

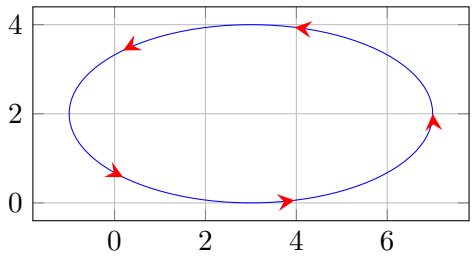


Figure 2.11: Ellipse

   Implement and run the control policy in the simulator.

2. Similar to Exercise 1, write a control policy to make the robot move in an ellipse shape, but now, the major axis of the ellipse is rotated thirty degrees with respect to the world coordinate frame.

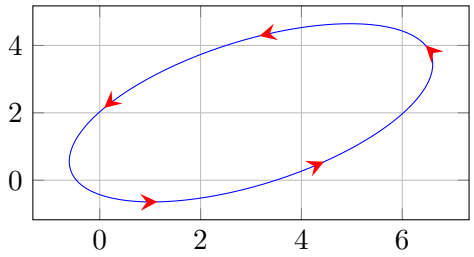


Figure 2.12: Rotated ellipse.

   Implement the policy in the simulator.

3. Write a control policy to follow the shape in the following figure.

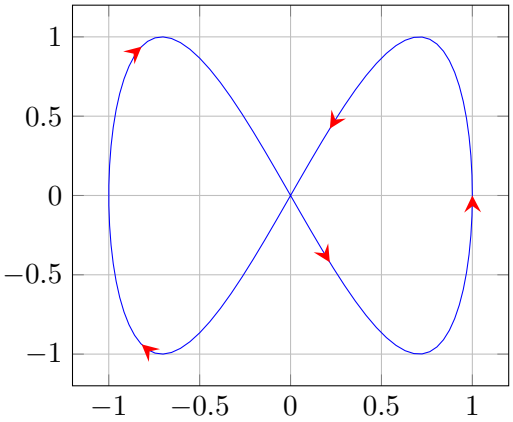


Figure 2.13: Rotated eight curve.

   Implement the policy in the simulator.

4. Simulate wind in the environment, and run your control policy for circular motion.

   We can include the wind in the dynamics of the robot, especifically in the function `simulate` in Code fragment 2.4.

   - Add a constant wind $w = [0.1, 0.1]^\top$.
   - Add a random wind with mean zero in x and y direction, and standard deviation of 0.1.

   Plot the path that the robot follows in both cases.

5. Implement the position controller from Equation 2.11. Test your robot moving from an initial location $[1, 2]$ to a desired location $[3, 3]$.

6. Write a control policy that uses the position controller to make the robot move to different locations to approximate the path of the following figure:

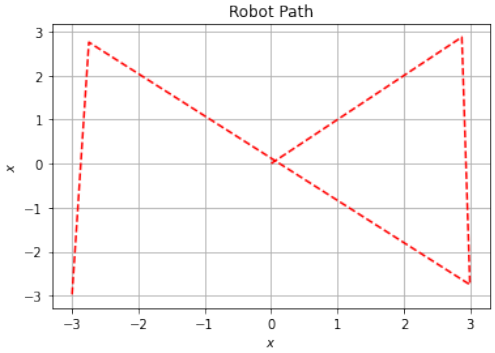


Figure 2.14: Robot path using position control.

7. **Billiard robot:** Design a robot that can move and bounce like a billiard ball within a rectangular table of 2×1 m^2. The initial velocity is random $\boldsymbol{u}(0) = \text{Random}()$. Write the controller and plot the result after five bounces.

8. Extend the 2D simulator to 3D.

Chapter 3 Learning to Fly (Part 1): Translational Dynamics

In this Chapter, we discuss the dynamics principles using the classical Newton Equations. We focus on the main forces that influence the translational motion of the vehicle, starting with the gravity. We start analyzing ***a point robot with mass*** m, without considering a body, links, or orientation. Similar to our previous chapter, the location of the robot at time t is denoted by $\boldsymbol{p}(t)$. Its velocity and acceleration are denoted by $\dot{\boldsymbol{p}}(t)$ and $\ddot{\boldsymbol{p}}(t)$ respectively. We remove the time index (t) to simplify the notation.

Definition 3.1. Newton Equation

In general, the motion of a rigid body (also known as our robot) can be described by the Newton Equation:

$$\boldsymbol{f} = m\,\ddot{\boldsymbol{p}}, \tag{3.1}$$

where $\boldsymbol{f}$ are all the forces that affect the vehicle. ♣

The force generated by the robot is denoted by $\boldsymbol{f}_r$. The gravity force is denoted by $\boldsymbol{f}_g = -m\,g\hat{\boldsymbol{z}}$, where g is the gravity constant, and $\hat{\boldsymbol{z}}$ is a unit vector in the z axis, i.e., $\hat{\boldsymbol{z}} = [0, 0, 1]\top$. So if the robot is in midair without considering the air resistance, the forces affecting the robot are $\boldsymbol{f} = \boldsymbol{f}_g + \boldsymbol{f}_r$. Then, we obtain,

$$\boldsymbol{f}_r - m\,g\hat{\boldsymbol{z}} = m\ddot{\boldsymbol{p}}. \tag{3.2}$$

Free fall: The first force that influences the vehicle is gravity. It is a passive force making it always tend to move downward. If the robot does not generate any force $\boldsymbol{f}_r = \boldsymbol{0}$, all the force $\boldsymbol{f}$ will be generated by gravity. So the equation for free fall would be $-m\,g\hat{\boldsymbol{z}} = m\ddot{\boldsymbol{p}}$, which can be rewritten as $\ddot{\boldsymbol{p}} = -\,g\hat{\boldsymbol{z}}$ after canceling the mass $m > 0$ on both sides. This differential equation does not describe change in the x and y coordinates, i.e., $\ddot{x} = 0$ and $\ddot{y} = 0$. The acceleration in the z-axis is negative, $\ddot{z} = -g$, until there is a force that compensates for the gravity force and cancels it out.

3.1 Mechanisms to Compensate Gravity

To compensate for gravity or generate lift, various types of actuators are employed in different contexts. Some common actuators are:

1. **Rotors (Propellers):** Commonly used in drones, helicopters, and other rotorcraft, rotors create lift by rotating blades that push air downwards, utilizing aerodynamic principles.
2. **Jet Engines:** Utilized in airplanes and jets, these engines expel gas at high speed to produce thrust, lifting the vehicle into the air by the principle of action and reaction (Newton's third

law).

3. **Rocket Engines:** Used in spacecraft, these engines produce thrust by expelling exhaust gases at high speed, enabling lift and propulsion in the vacuum of space.
4. **Gas Balloons (Lighter-than-air):** Filled with a gas less dense than air (like helium or hot air), these balloons rise due to buoyant force, as explained by Archimedes' principle.
5. **Airfoils (Wings):** Used in airplanes and gliders, airfoils generate lift by creating a pressure difference between their upper and lower surfaces as they move through the air.
6. **Flapping Wings:** Mimicking the flight of birds or insects, flapping wing actuators are used in ornithopters and some bio-inspired robots, generating lift through the oscillatory motion of wings.
7. **Electromagnetic Fields (Maglev):** Utilizing magnetic fields to generate force, these actuators are used in some advanced and experimental propulsion systems, such as maglev (magnetic levitation) trains.

In this Chapter, we will focus on two of the most practical and common: Rotors (Propellers), and Gas Balloons (Lighter-than-air).

3.1.1 Rotors and Thrust

Based on the Newton Equation, we need to generate a force $\boldsymbol{f}_r$ to compensate for gravity and be able to generate translational motion. Later in Part 2, we will also explore how this applies to rotational motion. A prevalent actuator used in aerial robots for generating this necessary force, commonly referred to as thrust, is the rotor. Rotors are ingenious devices that exploit the principles of aerodynamics in conjunction with Newton's third law of motion *the principle that for every action, there is an equal and opposite reaction*. By exerting a force on the air, which in turn causes an equal and opposite force to be exerted on the rotor, creating the necessary thrust for flight. This fundamental physical principle is integral to the design and operation of all types of rotorcraft.

One of the most widely recognized configurations employing rotors is the quadrotor. As the name suggests, this vehicle utilizes four rotors, typically arranged in a square layout. This configuration not only provides the needed upward thrust to counteract gravity but also enables a high degree of control over the vehicle's movement in three-dimensional space.

Definition 3.2. Rotor

A rotor comprises a motor and aerodynamically designed blades. These blades are shaped much like airfoils, a design that allows for differential air pressure across the blade surfaces when rotated at high speeds. A vehicle can have multiple rotors, in the case of a quadrotor $n = 4$, we can index them using $i = 1, ..., n$.

A rotor i generates a force

$$f_i = \frac{k_M}{k_F}\omega_i^2, \tag{3.3}$$

where k_M and k_F are motor coefficients that relate the angular velocity of the motor ω_i to the thrust and torque generated by the air drag. Those coefficients can be obtained experimentally using a device called Thrust Stand. Thrust is a vector, its magnitude is described by (3.3) *and its direction depends on the orientation of the rotor, denoted by* $\hat{\boldsymbol{f}}_i$. ♣

Different arrangements of rotors define the forces that the vehicle can generate to translate and rotate. The rotors can be attached to the vehicle frame rigidly, or attached to actuated joints such as servos that make the orientation of the thrust on command. The total thrust of the vehicle is

$$\boldsymbol{f}_r = \sum_{i=1}^{n} \boldsymbol{f}_i. \tag{3.4}$$

3.1.2 Gas Balloons (Lighter-than-air)

A balloon generates vertical force through buoyancy, leveraging the difference in density between the gas inside the balloon and the surrounding air. This principle allows balloons to float and maneuver vertically in the atmosphere, making them suitable for a variety of applications, from recreational hot air ballooning to scientific research and meteorological data collection.

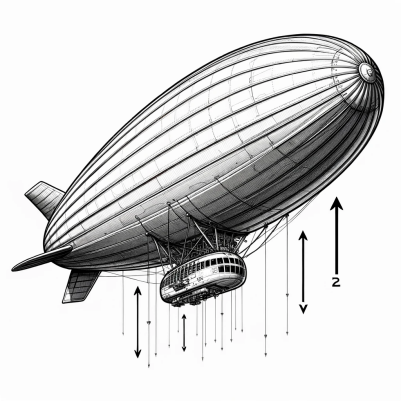


Figure 3.1: A blimp or Lighter-than-air (LTA) vehicle.

A balloon typically consists of a flexible, airtight envelope filled with a gas lighter than air, such as helium, hydrogen, or hot air. The material of the envelope needs to be lightweight yet strong enough to contain the gas and withstand external pressures.

Buoyancy Principle: According to Archimedes' principle, an object immersed in a fluid (in this case, air) is buoyed up by a force equal to the weight of the fluid it displaces. A balloon generates vertical force through this principle of buoyancy.

For a balloon to rise, the combined weight of the gas inside the balloon and the balloon material itself must be less than the weight of the air it displaces. This difference in weight creates an upward force the lift. The greater the difference between the weight of the displaced air and the weight of the balloon, the stronger the upward force.

Definition 3.3. Balloon

A balloon generates vertical force on the vehicle described by

$$\boldsymbol{f}_b = f_b\, \hat{\boldsymbol{z}}, \tag{3.5}$$

where f_b is the buoyancy force. ♣

Passive Balloon: In a passive balloon, buoyancy is typically determined by the initial conditions of the flight the type of gas used (like helium or hot air) and the volume of the balloon. The main advantage of the passive balloon is the ability to compensate for the gravity force. The Newton dynamics can include the bouyancy force,

$$\boldsymbol{f}_r + f_b\,\hat{\boldsymbol{z}} - m\,g\hat{\boldsymbol{z}} = m\ddot{\boldsymbol{p}}. \tag{3.6}$$

If we find a balloon that generates a buoyancy $f_b = m\,g$, then, we are passively compensating for the gravity, so there is no waste of anergy on maintaining altitude and the motors of the robot $\boldsymbol{f}_r$ can be used to only for motion purposes. The is the reason why blimps can stay in air for hours while quadrotors can only fly for few minutes.

Active Balloon for Buoyancy Control: It is possible to control the buoyancy of the balloon. To ascend, a balloon must displace a volume of air that weighs more than the balloon and its payload. This can be achieved by either decreasing the balloon's weight (e.g., releasing ballast) or increasing the volume of displaced air (e.g., heating the air in a hot air balloon). To descend, this process is reversed the balloon's weight is increased, or the volume of displaced air is decreased.

3.2 Hovering

To maintain a robot hovering, we need to generate a force $\boldsymbol{f}_r$ that compensates the gravity force, and maintains the robot at a desired location.

3.2.1 Altitude control

To introduce the concept of control, we will start defining a vehicle with a single rotor.

Definition 3.4. Up-down rotorcraft

An Up-down rotorcraft *is a vehicle with a single rotor pointing upwards. The total thrust of the vehicle is* $\boldsymbol{f}_r = \boldsymbol{f}_1$. *Since the rotor is vertical, the thrust that it can generate is* $\boldsymbol{f}_1 = f_1\hat{\boldsymbol{z}} = [0, 0, f_1]^\top$. *The control input is* $u = f_1$. *The vehicle is illustrated in Fig. 3.2.* ♣

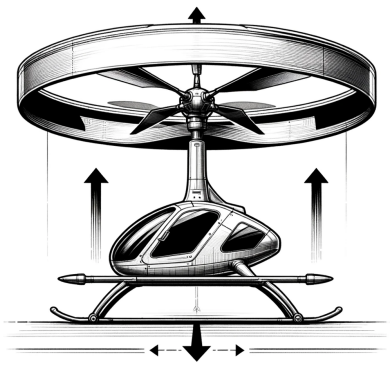


Figure 3.2: Up-down rotorcraft: A monorotor that can only move vertically. This vehicle does not exist, it is only for illustration purposes.

Vertical dynamics: Since we are only interested in the z-axis, the Equation (3.2) becomes uni-dimensional,

$$u - m\,g = m\ddot{z}. \tag{3.7}$$

PD Control: Our objective is to drive the robot to a desired height z_d, assuming that the vehicle does not change its orientation. To achieve this, we will use a proportional-derivative (PD) controller.

PD-controller for altitude control:

$$u = k_p(z_d - z) + k_d(\dot{z}_d - \dot{z}) + mg \tag{3.8}$$

where the desired velocity is zero to maintain altitude, i.e., $\dot{z}_d = 0$, since we do not want the robot to move, just to maintain its altitude.

We can verify that this controller works by analyzing the motion of the vehicle, described by dynamics equation (3.2). Replacing (3.8) in (3.2) and dividing both sides by the mass, we obtain,

$$\ddot{z} = \bar{k}_p(z_d - z) - \bar{k}_d\,\dot{z}_d,$$

where the coefficients are relabeled1 to $\bar{k}_p = k_p/m$ and $\bar{k}_d = k_d/m$. The solution of the second-order differential equation above is the exponential. So the vehicle will be driven to the desired height in exponential time.

The previous controller is stable and converge very fast, however, it requires an accurate knowledge of the mass of the vehicle. If the mass is not accurate, there might be a small term after the subtraction that generates an undesired acceleration. In a practical scenario, the error will be compensated by the proportional term, but the vehicle will not converge to the desired height. In contrast, there might be an offset.

PID Control: In the proportional integral derivative control, the integral term is the key for eliminating the steady-state error in the system. It integrates the error over time, accumulating the past errors. This persistent, cumulative action helps to adjust the control action until the error is nullified. This is particularly vital where precision and accuracy are essential, and even small steady-state errors can significantly impact performance.

PID-controller for altitude control:

$$u = k_p\,e_z + k_d\,\dot{e}_z + k_i \int_0^t e_z\,dt + mg, \tag{3.9}$$

where $e_z = z_d - z$.

In practice, we can remove the mg term and the integral error can accumulate until compensating for the mass error. However, ensuring stability with a PID controller in a mobile robotics context often involves a combination of careful parameter tuning, robust design practices. The stability of the system is highly dependent on the tuning of the PID parameters (proportional gain, integral gain, and derivative gain). Incorrect tuning can lead to instability, such as oscillations or divergent behavior.

3.2.2 Position control for hovering

The monorotor can only move in the direction of the rotor, in the case of the up-down rotorcraft, only in the z-axis. If we want to move in the three coordinates and also rotate in roll, pitch and yaw, we need at least six rotors. An interesting vehicle that can generate any force and torque is the following.

Definition 3.5. Omnicopter

The Omnicopter (Brescianini and DAndrea, 2018) is a multi-rotor vehicle with eight rotors. The rotors are oriented in different directions in a way that it can generate any forces in any direction. So the control input is

$$\boldsymbol{f}_r = \boldsymbol{u}. \tag{3.10}$$

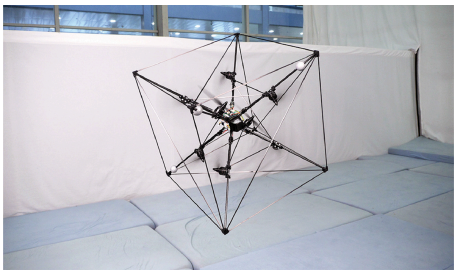


Figure 3.3: Omnicopter (Brescianini and DAndrea, 2018). Video here.

Later we will study how to make the map from individual forces to the total force. For now, lets just assume we can generate any force.

Translational dynamics: Since we have a vehicle that can generate any force, the Newton dynamics in (3.2) becomes,

$$\boldsymbol{u} - m\, g\hat{\boldsymbol{z}} = m\ddot{\boldsymbol{p}}. \tag{3.11}$$

PID Control: Our goal is to drive the robot to the desired location $\boldsymbol{p}_d$. Similar to the unidimensional case in Equation (3.9), we can apply the Proportional Integral Controller, but this time, in a vector form to include x- and y-axis. Since we the robot should stay on the desired location, the desired velocity is zero, i.e., $\dot{\boldsymbol{p}}_d = \boldsymbol{0}$. The controller is the following

PID-controller for hovering: Our controller drive the error $\boldsymbol{e} = \boldsymbol{p}_d - \boldsymbol{p}$ to zero when time goes to infinity.

$$\boldsymbol{u}(t) = \mathbf{K}_p\, \boldsymbol{e} + \mathbf{K}_d\, \dot{\boldsymbol{e}} + \int_0^t \boldsymbol{e}\, dt + m\, g\, \hat{\boldsymbol{z}}. \tag{3.12}$$

3.3 Coding a point robot to compensate gravity

In this section, we extend the simulator framework introduced in the previous chapter. This simulator retains the same functional architecture from Figure 2.9, encompassing modules for perception, control, and motion/simulation. However, a notable enhancement in this iteration is the incorporation of positional and velocity considerations for the robot. This is an important addition, as we are now exerting influence over the robot's acceleration.

State: To effectively manage this, we introduce an expanded ***state vector*** to include the velocity as part of the state, i.e., $\boldsymbol{x} = [\boldsymbol{p}, \dot{\boldsymbol{p}}]^\top$. Here, $\boldsymbol{p}$ represents the robot's position, and $\dot{\boldsymbol{p}}$ signifies its velocity. This enhanced state vector is pivotal for accurately modeling and controlling the robot's interactions with gravitational forces. By accounting for both position and velocity, we can apply more nuanced and effective control strategies, enabling the robot to navigate and perform tasks in environments where gravity plays a significant role. The implementation in Code 2.1 remains the same, except by Line 4, where we include the full state vector $\boldsymbol{x} = [x, y, z, \dot{x}, \dot{y}, \dot{z}]^\top$. Note we also included the z coordinate since we will work on objects that can change their height, different from the planar case in previous chapter. Then, Line 4 is the following,

```
4   x = array([0., 0., 0., 0., 0., 0.])        # Initial condition
```

Simulation: To understand the dynamics of the system, we need to describe how the state is changing with respect to time. In other words, analyzing the derivative of the state vector $\dot{\boldsymbol{x}} = [\dot{\boldsymbol{p}}, \ddot{\boldsymbol{p}}]^\top$. Assuming no air drag or external factors, the system does not have influence over the velocity $\dot{\boldsymbol{p}}$, but the acceleration will depend on the control input and the gravity. Following the Newton dynamics, we take the Equation (3.11), and divide it by the mass. Then, we can write the derivative of the extended state vector as

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{\boldsymbol{p}} \\ \frac{\boldsymbol{u}}{m} - g\hat{\boldsymbol{z}} \end{bmatrix} \tag{3.13}$$

Line 3 and 6 implements the equation of the dynamics, and Line 9 performs the integration. In this simulation, we take into account the collision with the floor. We assume that the collision is plastic, meaning that all the kinetic energy, related to the velocity, is absorbed by the impact. Line 13 stops the robot at the floor, and Line 14 sets its velocity to zero.

```
1  def simulate(Δt, x, u, m=1, g=9.8):
2     # Control input affects the acceleration
3     dx = array([x[3], x[4], x[5], u[0]/m, u[1]/m, u[2]/m])
4
5     # Include the gravity in the acceleration in z
6     dx[5] += - g
7
```

```
8    # Euler integration
9    x += Δt * dx
10
11   # Simulate the floor (Plastic collision)
12   if x[2] < 0:
13      x[2] = 0 # z = 0
14      x[5] = 0 # vz = 0
15   return x
```

Code fragment 3.1: Simulation function that includes gravity and collision with the floor.

Control policy: The following code defines a control policy that do not generate any force. To compensate gravity and drive the robot to a desired height, we need to generate a force in the z-axis as described earlier, so the solution is implementing Equation (3.8) in Line 5. See Exercise 2.

```
1  def control(t, y):
2     ux = 0
3     uy = 0
4     uz = 0  ### WRITE YOUR CONTROL POLICY HERE TO MAKE THE ROBOT FLY
5     return [ux, uy, uz]
```

Code fragment 3.2: Control policy with zero input.

Chapter 3 Exercise

1. Imagine that now you want to simulate an LTA vehicle based on a helium balloon. Modify the `simulate` function to include the buoyancy force. How do you make the vehicle to go up or down? If it goes down, what is the difference with free fall?
2. Given a desired height $z^d = 15$, implement in the simulation a control policy (see Equation (3.8)) that drives the robot from an initial location $\boldsymbol{p} = [0, 0, 10]^\top$ to the desired location $\boldsymbol{p}^d = [0, 0, z^d]^\top$.
3. Solve the previous exercise but now, your controller considers a mass $\hat{m} = 1$. However, the actual mass is $m = 0.8$. How can the robot compensate for the unknown value of the mass? Can the robot maintain the desired altitude?
4. Modify the dynamics of the vehicle to make it bounce on the floor like a ball.

Chapter 4 Learning to Fly (Part 2): Rotational Dynamics in 2D

Concepts

- ❏ *Robot orientation*
- ❏ *Arrow-robot*
- ❏ *Euler Equation*
- ❏ *Rotational dynamics*

Controlling Flying vehicles involves more than compensating gravity and translating in three dimensions, it is essential to control the rotation of the vehicle. Any rigid body can rotate with respect to x-, y- and z-axes. Before we combine all rotations, we study a simple robot, called the arrow robot in 2D, which different from the point robot, this type of robot actually exists. This robot only needs to control its angle with respect to the z-axis, also called yaw angle.

If the robot is equipped with a sensor, such as a camera, we might want to utilize it for environmental scanning or targeting specific objects. This necessitates considering the camera's orientation, which, in turn, depends on the robot's orientation, as the camera is attached to the robot. In this chapter, we operate under the assumption that all sensors are rigidly mounted to the robot. This premise simplifies our study to focus primarily on the robot's orientation and movement.

Building on the knowledge acquired in the previous chapter about controlling the robot's altitude, we will proceed under the assumption that the robot is capable of maintaining a constant altitude, allowing it to navigate solely within the xy-plane. Concurrently, the robot will possess the ability to rotate around the z-axis during its motion.

4.1 Arrow Robot in 2D

The design of a robot's chassis and sensors typically incorporates a specific orientation, often referred to as the 'front' of the robot. Building upon this concept, we can enhance our point robot model to include orientation, initially in a two-dimensional (2-D) space and subsequently extending to three dimensions (3-D).

Delving into the study of 2-D motion is particularly pertinent. This is because, in many practical scenarios, even in a 3-D environment, a robot's movement can be effectively constrained to a two-dimensional plane. A common example of this is a robot operating at a constant altitude, where its vertical position remains fixed while it navigates horizontally. Understanding 2-D motion lays a solid foundation for comprehending more complex 3-D movements and is an essential step in the progression towards mastering the dynamics of robotic orientation and movement in three-dimensional space.

Definition 4.1. 2-D arrow-robot

An arrow-robot *is mass-less and infinitesimally small point, represented by its location* $\boldsymbol{p} = [x, y]^\top \in \mathbb{R}^2$ *and its orientation* $\theta \in [-\pi, \pi]$.

♣

Based on the definition, we need three variables to represent the configuration of the arrow on the plane. Any rigid body in 2-D, similar to a coin on a table, its configuration is determined by its position on the table $\boldsymbol{p}$ and the angle of orientation θ. In the case of the 2-D arrow-robot (see Fig. 4.1), we define its configuration by the vector

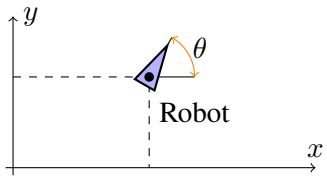


Figure 4.1: Arrow-robot in 2-D.

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{p} \\ \theta \end{bmatrix}.$$

4.2 Velocity control

4.2.1 Full velocity control of the 2-D arrow-robot

Similar to our previous chapter, we can assume that our control input is the translational velocity $\dot{\boldsymbol{p}}$ and rotational velocity $\dot{\theta}$. Then, the control input is can be defined by the linear and angular control inputs

$$\dot{\boldsymbol{p}} = \boldsymbol{u}_l \text{ and } \dot{\theta} = u_\omega. \tag{4.1}$$

The position controller in (2.11), we can be used the proportional controller for the linear translation. Then, we use

$$\boldsymbol{u}_l = \mathbf{K}(\boldsymbol{p}_d - \boldsymbol{p}),$$

where $\mathbf{K} = [k_{ij}]$ is a diagonal matrix with the constant gains in its entries. Since every variable is controlled independently, the ith gain k_{ii} is associated with the ith element of the configuration vector $\boldsymbol{x}$. For example, if we increase the gain k_{22}, the robot will converge faster in the y-coordinate.

Angle problem One might think that the angle can also be solved using a p-controller like $u_\omega = k_\omega(\theta_d - \theta)$, and it actually works if the angle changes only between $-\pi$ and π. When the robot completes a cycle, the angle jumps from $-\pi$ and π and bice-versa creating instability in the system. To fix that problem, we can use the following equation instead,

$$u_\omega = k_\omega \arctan\left(\frac{\sin(\theta_d - \theta)}{\cos(\theta_d - \theta)}\right). \tag{4.2}$$

For implementation purpuses, there is an ambiguiety witht he arctangent since it can return the same angle for two different inputs. To solve this issue, we recommend to use the function `arctan2` instead the normal arctangent function.

4.2.2 Unicycle robot (underactuated robot)

A unicycle robot is a robot whose locomotion mechanism is vehicle that touches the ground with only one wheel. The unicycle robot can control the linear velocity to the front v, and the angular velocity ω. When the robot generates an input (v, ω), the motion is described by

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}. \tag{4.3}$$

Figure 4.2: Unicycle vehicle (From Wikimedia Unicycle).

Unfortunately, we cannot control the three variables $(\dot{x}, \dot{y}, \dot{\theta})$ independently, since we only have two inputs. Then we have to choose a subset of two variables that we want to control. We could control $\dot{x}$ and $\dot{\theta}$, but then the robot would be moving ramdonly on the y-axis. Then, we can choose to define the desired position $\boldsymbol{p}$ where we want to go, similar to the point robot.

4.2.3 Position control for unicycle robot

Our problem is: *Given a unicycle robot and a goal position* $\boldsymbol{p}_d = [x_d, y_d]^\top$*, design a control input* (v, ω) *that drives the robot to the goal position.* For this problem, we can use the angular velocity ω to make the from of the robot point towards the goal, and the linear velocity v to approach the goal,

$$v = k_v \|\boldsymbol{p}_d - \boldsymbol{p}\|, \tag{4.4}$$

$$\omega = k_\omega(\alpha - \theta), \tag{4.5}$$

where $\alpha = \arctan\left(\frac{y_d - y}{x_d - x}\right)$. The difference $(\alpha - \theta)$ can bring problems since the result angle is not between $-\pi$ and π.

4.3 Rotational Dynamics on the plane

The forces generated by the propellers not only generate a force on the vehicle, but it also generate a torque. A torque will influence the way the vehicle rotates

Rotational Dynamics is modeled by the Euler equation that describes how the angular acceleration $\ddot{\theta}$ changes based on the total torque acting on the vehicle, τ. I is the moment of inertia about the center of mass. Then,

$$\ddot{\theta}\, I = \tau. \tag{4.6}$$

It's important to note that this equation is simplified and assume a planar motion. In real-world applications, additional factors such as friction, air resistance, and non-uniform mass distribution might need to be considered.

Moment of Inertia: The moment of inertia, often symbolized as I, is a fundamental concept in physics and engineering, particularly in the fields of mechanics and dynamics. It quantifies an object's resistance to changes in its rotational motion about a specific axis. The moment of inertia is dependent not only on the mass of an object but also on how that mass is distributed with respect to the axis of rotation. Different shapes and mass distributions result in different moments of inertia, even if the objects have the same total mass.

Torque: Torque is a measure of the force causing an object to rotate about an axis. It is a vector quantity, with both magnitude and direction. Torque, denoted by τ , is the product of the force applied f and the lever arm distance r from the axis of rotation to the point where the force is applied. In 2D is,

$$\tau = r\, f\, \sin(\beta),$$

where β is the angle between the force vector and the lever arm. The standard unit of torque in the International System of Units (SI) is the Newton-meter (Nm). In rotational dynamics, torque plays a role analogous to force in linear dynamics. It is the primary factor in determining the angular acceleration of a body around an axis.

4.3.1 Flying

Now that we understand the main concepts of the rotational dynamics, we can put it in practice using our first vehicle that only has two actuators and moves on the plane (See Fig. 4.3).

Definition 4.2. Twin-prop vehicle 2D

A twin-prop vehicle in 2D is a simplified model of a vehicle that consists of a central body and two propellers located symmetrically on either side of this body. Each propeller $i = 1, 2$ rotates to generate a force f_i. Its motion is confined on the xy-plane. The total force is

$$\boldsymbol{f} = [f_1 + f_2, 0]^\top. \tag{4.7}$$

The total torque it can genrate is,

$$\tau = r\, f_1\, \sin(90) + r\, f_2\, \sin(-90) = r(f_1 - f_2). \tag{4.8}$$

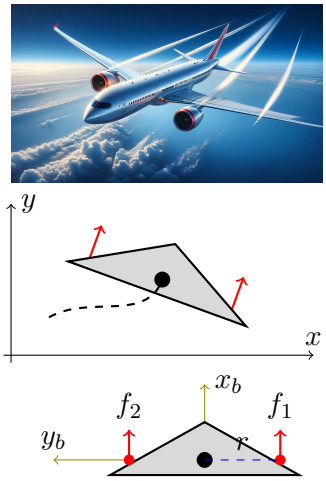


Figure 4.3: Twin-prop airplane. The forces of the two motors on the sides are represented by the red vectors

Chapter 5 Obstacles and Trajectories

Concepts

- ❏ *Geometric obstacles*
- ❏ *Time parametrized functions*
- ❏ *Straight-line trajectory*
- ❏ *Polynomial trajectory*
- ❏ *Piecewise trajectory*
- ❏ *Spline trajectory*

Mobile robots need to move around the environment in order to perform tasks.In applications such as patrolling and surveillance, robots need to move around to watch an area. Other applications like package delivery and object transportation require moving from point A to point B. Although the applications are different, motion and trajectories are always involved in the task. This chapter studies some ways to move a point by designing and following trajectories. We start with the major difficulty of driving a robot to its destination, moving around obstacles.

5.1 Obstacles

Our previous chapter assumed that our robot was in an open space, but there are always impediments to free motion in real scenarios. The environment creates some constraints for the robot, e.g., walls, fences, trees, and doors. In the case of a ground robot, a lake or a river can impede the robot's motion. Inside the environment, there might be multiple objects, e.g., cars, boxes, humans, and chairs. In general, we can call any of these elements that constrain o limit the robot's motion as obstacles. Moving towards an object or an obstacle not only limits its motion but can also damage the object or the robot. The obstacles can be static or dynamic. When they are static (e.g., a tree), we define a trajectory to avoid them. In the case of dynamic obstacles, they can either move by themselves (e.g., humans or cars) or be moved (e.g., ball or a door) For now, we can assume that all obstacles are static, and we will try to avoid them.

We define an *obstacle* as a connected subset of the Euclidean space. Obstacles can have many shapes, and they can be complicated to model, so we can start with geometric abstraction.

Definition 5.1. Geometric obstacle

A geometric obstacle *is a connected subset of the Euclidean space, $\mathcal{O} \in \mathbb{R}^d$. It can be described by a geometric figure in 2-D (e.g. triangle, rectangle, circle, or polygon) or in 3-D (e.g. sphere, ellipsoid, cylinder, cuboid, prism, polyhedron).* ♣

The advantage of the geometric obstacles over highly detailed approximated objects is that they can be represented with a few parameters. We illustrate the abstraction of a chair into a geometric obstacle in Figure 5.1. In the 3-D case, we can use a sphere that only requires the three coordinates of the center point and the radius; a cuboid is represented by 8 points, and a polyhedron is represented by multiple points, as can be seen in the figure. As we can

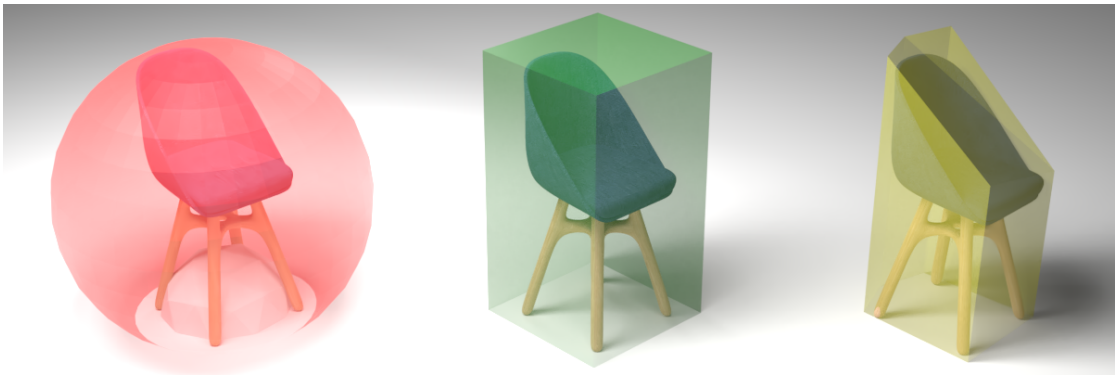


Figure 5.1: Representations of a chair as a geometric obstacle. From left to right: sphere, cuboid and polyhedron.

see, more points offer accurate details, but it requires additional variables for its representation. Geometric obstacles with few variables, like the sphere, are straightforward to represent and fast to compute for planning algorithms, but they lack detail. For the examples of the Figure 5.1, a small robot would be able to pass between the legs, but after abstracting the robot by the geometric obstacle in the figures, the robot would be colliding with the geometric obstacle. An alternative solution is decomposing the obstacle into multiple geometric obstacles or simply using a complex polyhedron with more vertex points. So we can find a trade-off between the number of variables that we want to use to represent the obstacle and the approximation accuracy. We want to move our point-robot without colliding with objects. So we can define a collision.

Definition 5.2. Collision

We say that a point-robot $\boldsymbol{p}(t)$ collides *with an obstacle* $\mathcal{O}$*, if at any time step* $t_c \in [t_0, t_f]$*, the robot touches the obstacle, i.e.,* $\boldsymbol{p}(t_c) \in \mathcal{O}$.

♣

In 2-D, we can also represent geometrical objects. In the chair of Figure 5.1, seeing the objects from the top, we would have a circle, a square, and a polygon, respectively.

5.2 Designing point-to-point trajectories

To avoid obstacles, we will design trajectories that can lead the robot to its goal and avoid obstacles.

Definition 5.3. Trajectory

A trajectory *is a continuous function* $\boldsymbol{\gamma}$*, that maps time to the Euclidean space* $\mathbb{R}^d$ *of dimension* $d \in \{1, 2, 3\}$*. The time interval starts at time* t_0 *and ends at time* t_f*, i.e.,* $\boldsymbol{\gamma} : [t_0, t_f] \to \mathbb{R}^d$.

♣

Different from our previous chapter, here, the notion of time is very important, we want to set the location of the robot at every instant of time. Lets start with the most important information that we need to provide, the time to finish the trajectory. Lets start going from point $\mathbf{p}_0$ to point $\mathbf{p}_f$, but different from our previous chapter, we also want to start at time $t = 0$ and

end at time $t = t_f$. We know that in the Euclidean space, the closest path between two points is the straight-line. Using the general parametric equation of a line, our trajectory can be described by the equation

$$\boldsymbol{\gamma}(t) = \mathbf{a}_1 t + \mathbf{a}_0, \tag{5.1}$$

where $\mathbf{a}_0$ and $\mathbf{a}_1$ are the coefficients that define the straight-line. We can obtain the coefficients by evaluating (5.1) at time $t = 0$, obtaining $\mathbf{p}_0 = \mathbf{a}_0$, and at time $t = t_f$, obtaining $\mathbf{p}_1 = \mathbf{a}_1(t_f) + \mathbf{a}_0$. Based on these two resultant equations, we can solve it to obtain the coefficients $\mathbf{a}_1$ and $\mathbf{a}_0$.

Definition 5.4. Straight-line trajectory

A straight-line trajectory *is a function that describes a segment of a straight line in the Euclidean space that starts at a point* $\mathbf{p}_0$ *at* $t = 0$ *and ends at a point* $\mathbf{p}_f$ *at time* $t = t_f$. *The equation that describes the trajectory is:*

$$\boldsymbol{\gamma}(t) = \mathbf{a}_1 t + \mathbf{a}_0, \tag{5.2}$$

where $\mathbf{a}_1 = \frac{\mathbf{p}_f - \mathbf{p}_0}{t_f}$ *and* $\mathbf{a}_0 = \mathbf{p}_0$, *for* $t \in [0, t_f]$.

The coefficients $\mathbf{a}_1$ and $\mathbf{a}_0$ can be obtained by evaluating the initial and the final conditions, $\boldsymbol{\gamma}(0) = \mathbf{p}_0$ and $\boldsymbol{\gamma}(t_f) = \mathbf{p}_f$ respectively. This gives us the two equations

$$\mathbf{p}_0 = \mathbf{a}_1(0) + \mathbf{a}_0, \tag{5.3}$$

$$\mathbf{p}_f = \mathbf{a}_1(t_f) + \mathbf{a}_0. \tag{5.4}$$

The first equation gives us the constant $\mathbf{a}_0$, and we can obtain $\mathbf{a}_1$ from the second equation.

Example 5.1: Straight-line trajectory Design a straight-line trajectory that starts at point $\mathbf{p}_0 = [0, 0]^\top$, and ends at point $\mathbf{p}_f = [8, 4]^\top$.

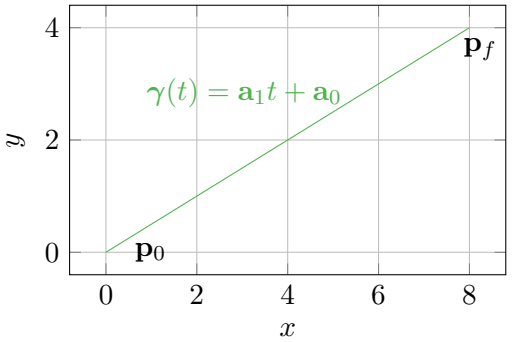


Figure 5.2: Straight-line trajectory.

Figure 5.2 illustrates the example of the straight-line trajectory. The straight-line trajectory's advantage is that it describes the shortest path between two points in the Euclidean space. The straight line is not always the shortest path in other spaces. For example, the earth is approximated by a plane, but its topology is more similar to a sphere. That is the reason why airplane paths look counter-intuitive because they do not follow a straight line on the map.

The straight-line trajectory seems like the optimal solution to move from point to point in our scenario. However, there are two main issues. *i)* Although the Euclidean space can be used as a modeling tool for many applications, robots do not always move in the Euclidean space, as we discussed with the airplane example. *ii)* The environment is not always open, and there are obstacles (e.g., humans, objects, walls, or other robots). We will continue studying obstacles,

represent them, and design trajectories to avoid them.

5.3 Parametric curves and polynomial trajectories

In Section 2.2, we learned about generating control inputs in open-loop. In that case, we went from a control input to compute the resultant trajectory. Now, we want to define the robot's trajectory, making the point that the robot follows it. We can design the trajectory based on known trajectory functions (line, circle, ellipse, sinusoidal, polynomial), or we can also be creative and combine functions.

Example 5.2: Avoiding an obstacle Design a trajectory that starts at location $[0, 0]^\top$ and ends at time $t_f = 8$ on the location $[8, 8]^\top$. Additionally, we want to avoid a rectangular obstacle in the middle as illustrated in Figure 5.3.

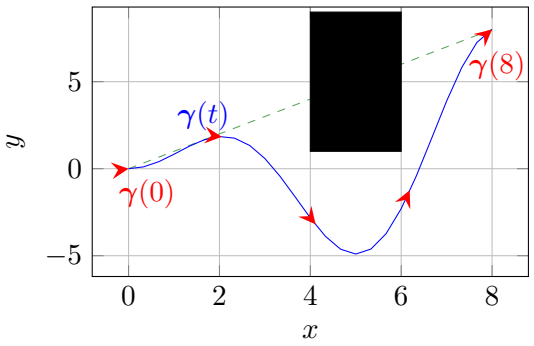


Figure 5.3: Trajectory to avoid an obstacle based on a sinusoidal function (5.5).

In this case, we cannot use a straight-line, denoted by a dashed line in the figure, because it would go through the obstacle (and this is not a course about making tunnels). Instead, we can be creative and design trajectory function that goes around the obstacle. For example, the trajectory function

$$\boldsymbol{\gamma}(t) = \begin{bmatrix} t \\ t\sin(5\pi t/16) \end{bmatrix}, \tag{5.5}$$

describes a curve that allows the robot to avoid the obstacle (as it shown in Figure 5.3). Although this is a solution to the problem, choosing a function and tuning parameters for every scenario can be very tedious. In this scenario, the sinusoidal function worked well, but if we want a different shape, like a straight line, or if we want to control the trajectory's speed, adapting the sinusoidal function will become more and more difficult. Fortunately, polynomials offer a powerful tool for this problem. The first-order polynomial defines a straight line, which we described in Definition 5.2. The coefficients $\mathbf{a}_1$ and $\mathbf{a}_2$ can be obtained using the initial and the final points $\boldsymbol{p}_0$ and $\boldsymbol{p}_f$. Using third-degree polynomials, we can set the initial and final velocities $\dot{\mathbf{p}}_0$ and $\dot{\mathbf{p}}_f$.

Definition 5.5. Cubic polynomial trajectory

A cubic-polynomial trajectory*, also called third-order polynomial trajectory, is a function that describes a segment of a line in the Euclidean space that, starting at time $t = 0$, at a point $\mathbf{p}_0$ with velocity $\dot{\mathbf{p}}_0$ and ends at time $t = t_f$ at a point $\mathbf{p}_f$ with velocity $\dot{\mathbf{p}}_f$. The*

equation that describes the trajectory is:

$$\boldsymbol{\gamma}(t) = \mathbf{a}_0 + \mathbf{a}_1 t + \mathbf{a}_2 t^2 + \mathbf{a}_3 t^3 \tag{5.6}$$

where

$$\begin{aligned} \mathbf{a}_0 &= \mathbf{p}_0 \\ \mathbf{a}_1 &= \dot{\mathbf{p}}_0 \\ \mathbf{a}_2 &= \frac{3\boldsymbol{p}_f - 3\boldsymbol{p}_0 - 2\dot{\boldsymbol{p}}_0\, t_f - \dot{\boldsymbol{p}}_f\, t_f}{t_f^2} \\ \mathbf{a}_3 &= \frac{2\boldsymbol{p}_0 + (\dot{\boldsymbol{p}}_0 + \dot{\boldsymbol{p}}_f) t_f - 2\boldsymbol{p}_f}{t_f^3} \end{aligned}$$

for $t \in [0, t_f]$.

A compact way to write (5.6) is using a matrix form

$$\boldsymbol{\gamma}(t) = \mathbf{A}\, \mathbf{t}(t), \tag{5.7}$$

where $\mathbf{A} = [\mathbf{a}_0\ \mathbf{a}_1\ \mathbf{a}_2\ \mathbf{a}_3]$ and the time vector is composed by the polynomial coefficients $\mathbf{t}(t) = [1, t, t^2, t^3]^\top$.

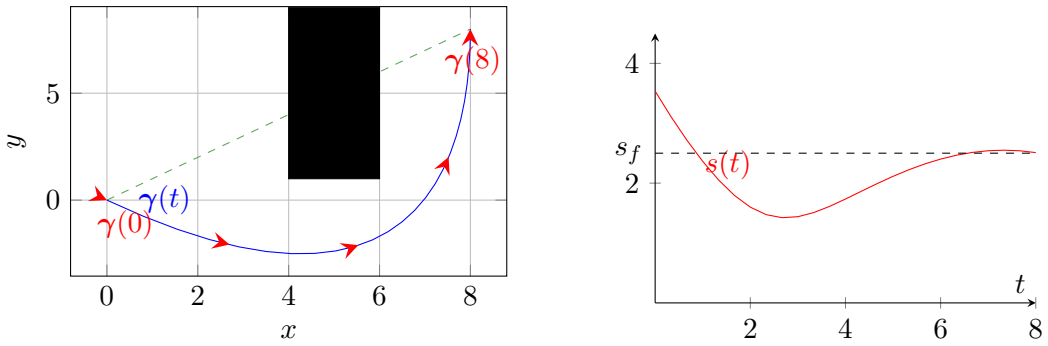
Hint: The cubic polynomial in (5.6) is written in a vector form. For the implementation, you can calculate the polynomial for the x-coordinates and the y-coordinates independently.

Derivation: The coefficients in (5.6) are obtained by just replacing the start and end conditions ($\mathbf{p}_0$, $\dot{\mathbf{p}}_0$, $\mathbf{p}_f$ and $\dot{\mathbf{p}}_f$), in the polynomial trajectory function $\boldsymbol{\gamma}(t)$ and its derivative $\dot{\boldsymbol{\gamma}}(t) = 3\mathbf{a}_3 t^2 + 2\mathbf{a}_2 t + \mathbf{a}_1$. ve us a system of 4 linear equations and four unknown coefficients. Then we can solve the linear system to obtain the same solution as in the definition.

Example 5.3: Avoiding an obstacle using a cubic polynomial Similar to our previous example, we want a trajectory to avoid the obstacle, but in this case, we want the trajectory function to be a cubic polynomial.

Based on the problem, we have the time, initial and final points, (t_f, $\mathbf{p}_0$, $\mathbf{p}_f$), so we have the freedom to choose the initial and final velocity vectors $\dot{\mathbf{p}}_0$ and $\dot{\mathbf{p}}_f$. The trivial case would be choosing $\dot{\mathbf{p}}_0 = \mathbf{0}$ and $\dot{\mathbf{p}}_f = \mathbf{0}$, but the final trajectory would be the line as the dashed line in Figure 5.4a, as that is the shortest path between the two points. We can try different velocities to see how the trajectory starts to generate a curve. The blue trajectory in the figure is generated with velocities $\dot{\mathbf{p}}_0 = [2.5, -2.5]^\top$ and $\dot{\mathbf{p}}_f = [0, 2.5]^\top$.

Although this example seems like it solves the problem, it requires a jump in the speed from zero to high speed, and in the end, from a high speed to zero. Unfortunately, this braking system that suddenly stops the vehicle is very unrealistic for actual robots. Additionally, what if we want to avoid multiple obstacles? We can solve both problems by adding more points to visit!



(a) Polynomial trajectory

(b) Speed of the trajectory.

Figure 5.4: Speed of the trajectory

5.4 Designing trajectories based on waypoints

Robot tasks involve moving to multiple places, so we need to design a trajectory that smoothly passes through various points. Let's take a look at Figure 5.5 and try to find a trajectory that moves from the initial point to its goal destination.

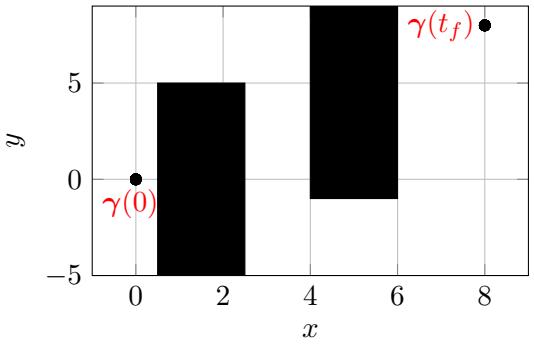


Figure 5.5: Environment with obstacles. The objective is to design a trajectory $\gamma(t)$ that reaches the goal without colliding with the obstacles.

Following the famous paradigm *Divide and Conquer*, we can divide the problem of finding a trajectory into sub-problems of trajectories that connect pairs of points. So the robot passes through n points into a trajectory composed of parts that can be connected. Since we already know how to connect pairs o points by different types of functions and polynomials, we can divide the problem into pairs of connected points. Our robot starts at point $\boldsymbol{p}_0$ and wants to move through n waypoints $\boldsymbol{p}_1, ..., \boldsymbol{p}_n$. We denote the set that contains the starting point and the points to visit by $\mathcal{P} = \{\boldsymbol{p}_0, ..., \boldsymbol{p}_n\}$. For each point $i = 0, 1, 2, ...n$, we can also define the time when the robot should visit it, denoted by t_i. In this way, each waypoint is defined by a tuple $(t_i, \boldsymbol{p}_i)$, and the set of waypoints is denoted by $\mathcal{W} = \{(0, \boldsymbol{p}_0), (t_1, \boldsymbol{p}_1), ..., (t_n, \boldsymbol{p}_n)\}$. Our goal is to find a trajectory $\boldsymbol{\gamma}(t)$ that passes through all the points in the specified times.

In this way, the trajectory function $\boldsymbol{\gamma}(t)$ will be divided into n sub-functions. We call this a piecewise trajectory function with the form

$$\boldsymbol{\gamma}(t) = \begin{cases} \boldsymbol{\gamma}_{0,1}(t), & \text{if } t \in [t_0, t_1], \\ \boldsymbol{\gamma}_{1,2}(t - t_1), & \text{if } t \in [t_1, t_2], \\ \vdots & \\ \boldsymbol{\gamma}_{n-1,n}(t - t_{n-1}), & \text{if } t \in [t_{n-1}, t_n]. \end{cases} \tag{5.8}$$

We can combine any function that we want, for example, a semi-circle with a parabola, but the

important requirement of each pair of subsequent functions is that they have to be **continuous**.

Let's start with the simplest case, connecting the points by multiple straight lines, also called polylines.

Definition 5.6. Polyline trajectory

A polyline trajectory *is a continuous piecewise function that passes through the set of waypoints* $\mathcal{W} = \{(t_0, \boldsymbol{p}_0), ..., (t_n, s\boldsymbol{p}_n)\}$ *using* n *straight-line trajectories,*

$$\boldsymbol{\gamma}(t) = \begin{cases} \boldsymbol{\gamma}_{0,1}(t), & \text{if } t \in [t_0, t_1], \\ \boldsymbol{\gamma}_{1,2}(t - t_1), & \text{if } t \in [t_1, t_2], \\ \vdots & \\ \boldsymbol{\gamma}_{n-1,n}(t - t_{n-1}), & \text{if } t \in [t_{n-1}, t_n]. \end{cases} \tag{5.9}$$

Where each function $\boldsymbol{\gamma}_{i-1,i}(t)$ *for all* $i = 1, ..., n$*, is a straight-line trajectory from Definition 5.2* (5.2).

♣

Example 5.4: Avoiding multiple obstacles using a polyline trajectory. Design a polynomial trajectory for robot that starts at point $[0, 0]^\top$ and ends at point $[8, 8]^\top$ while avoiding collisions with obstacles. The first obstacle has its button left corner at the point $(4, -1)$, with width equals to 2.0 and height equals to 10.0. The second obstacle has its button left corner at the point $(0.5, -5.0)$, with width equals to 3.0 and height equals to 5.5. The environment is illustrated in Figure 5.6.

The easiest solution might be moving upward until we reach the end of the obstacle, then we continue applying the same strategy to continue rightward, downward, rightward, and then upward. The procedure is similar to drawing the solution of a maze using a pencil and a rule. This gives us an intuitive way to find the minimum number of waypoints. We will define the waypoints manually, but later we will learn how to write algorithms to find them automatically; the topic is called planning.

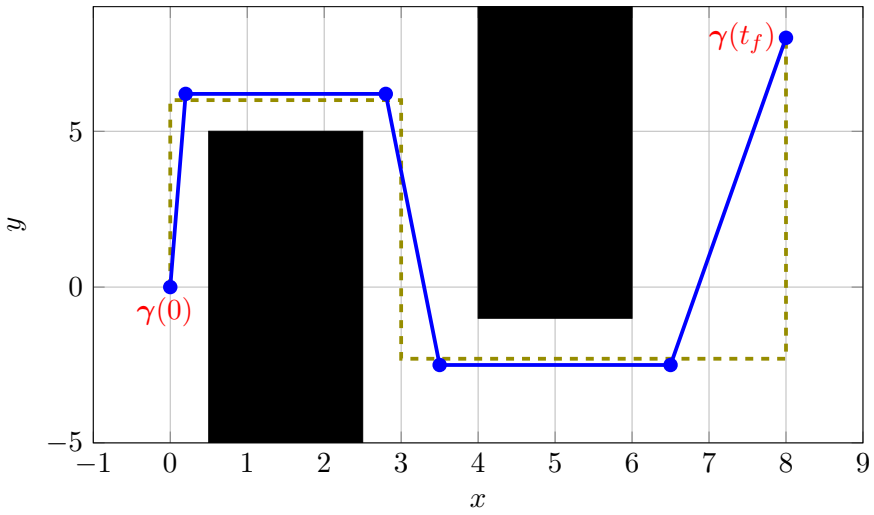


Figure 5.6: Polyline trajectory

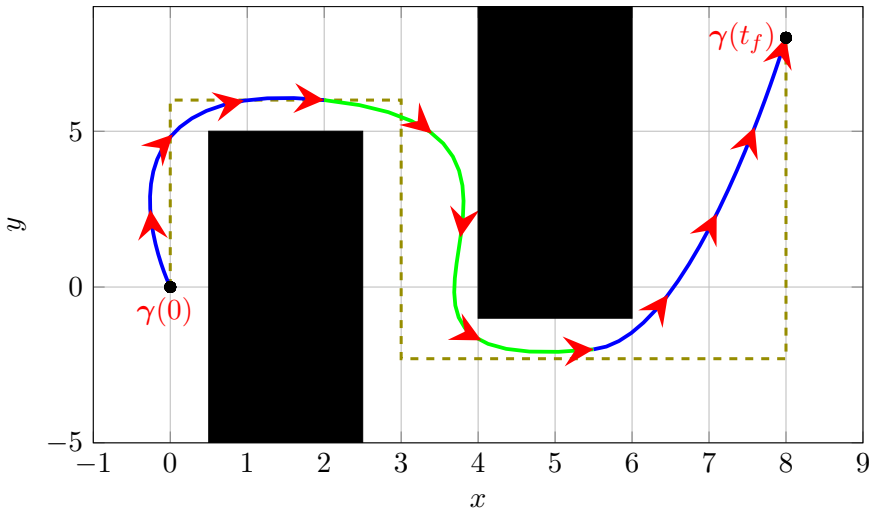


Figure 5.7: Spline trajectory.

Following our simple strategy, we can obtain the points to visit $\mathcal{P} = \{(0,0),(0,6.2),(3,6.2),(3,-2.5),(8,-2.5),(8,8)\}$. For the time, we can simple do $t_i = i$ for $i = 1,...,n$. The trajectory after applying the equation of the polyline trajectory is represented in Figure 5.7 by the solid blue line.

This approach's main advantage is its simplicity since it is easy to define intuitive points that solve the "maze". However, this approach has some disadvantages: It requires sharp changes in the direction of motion and the speed after every waypoint. If we compute the derivative of the trajectory, we can see that the velocity is not continuous, even though the trajectory is. Mechanical systems have difficulties operating aggressively or with instantaneous changes; the desire is to design smooth trajectories.

Let's make a smooth trajectory, similar to the third-degree polynomial trajectory; we can compose smooth trajectories that visits multiple waypoints.

Definition 5.7. Spline trajectory

A spline trajectory *is a continuous and smooth piecewise function that passes the set of waypoints* $\mathcal{W} = \{(t_0, \boldsymbol{p}_0, \dot{\boldsymbol{p}}_0), ..., (t_n, \boldsymbol{p}_n, \dot{\boldsymbol{p}}_n)\}$ *using* n *straight-line trajectories,*

$$\boldsymbol{\gamma}(t) = \begin{cases} \boldsymbol{\gamma}_{0,1}(t), & \text{if } t \in [t_0, t_1], \\ \boldsymbol{\gamma}_{1,2}(t - t_1), & \text{if } t \in [t_1, t_2], \\ \vdots & \\ \boldsymbol{\gamma}_{n-1,n}(t - t_{n-1}), & \text{if } t \in [t_{n-1}, t_n]. \end{cases} \tag{5.10}$$

Where each function $\boldsymbol{\gamma}_{i-1,i}(t)$ *for all* $i = 1, ..., n$, *is a cubic trajectory from Definition 5.3.* ♣

5.5 Following a trajectory

Similar to the position control, we can track a moving point using a P-Controller. However, we need to add a term called ***feed-forward***. Our controller is

Trajectory follower:

$$\boldsymbol{u}(t) = K(\boldsymbol{p}_d(t) - \boldsymbol{p}(t)) + \dot{\boldsymbol{p}}_d(t). \tag{5.11}$$

where $\boldsymbol{p}_d(t) = \boldsymbol{\gamma}(t)$, and $\dot{\boldsymbol{p}}_d(t) = \dot{\boldsymbol{\gamma}}(t)$.

Different from (2.11), the reference point in this controller changes with time. Without the feed-forward term, the robot would slow down when reaching the reference point and then speed up to when the error gets more prominent due to the moving point. In our controller, when the robot reaches the reference point, it will continue following the velocity of the trajectory. In the case of any disturbance causing a larger error in location, it will be reduced exponentially due to the proportional factor.

Bibliography

Brescianini, D. and DAndrea, R. (2018). An omni-directional multirotor vehicle. *Mechatronics*, 55:76–93.

Choset, H. M., Hutchinson, S., Lynch, K. M., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). *Principles of robot motion: theory, algorithms, and implementation*. MIT press.

Koenig, N. and Howard, A. (2013). Gazebo-3d multiple robot simulator with dynamics (2003). *URL: http://gazebosim. org*, 3.

Lynch, K. M. and Park, F. C. (2017). *Modern Robotics*. Cambridge University Press.

Lynch, S. (2018). *Dynamical Systems with Applications using Python*. Springer.

Mathews, J. H., Fink, K. D., et al. (2004). *Numerical methods using MATLAB*, volume 4. Pearson prentice hall Upper Saddle River, NJ.

O'Reilly, O. M. (2020). *Intermediate Dynamics for Engineers: Newton-Euler and Lagrangian Mechanics*. Cambridge University Press.

Siegwart, R., Nourbakhsh, I. R., and Scaramuzza, D. (2011). *Introduction to autonomous mobile robots*. MIT press.

Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic robotics*. MIT press.