INTRODUCTION TO AUTONOMOUS ROBOTS

Nikolaus Correll
University of Colorado at Boulder

LibreTexts™

University of Colorado at Boulder
Introduction to Autonomous Robots

Nikolaus Correll

The LibreTexts libraries are Powered by NICE CXone Expert and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

TABLE OF CONTENTS

Licensing

*A detailed breakdown of this resource's licensing can be found in **Back Matter/Detailed Licensing**.*

CHAPTER OVERVIEW

1: Introduction

Robotics celebrated its 50th birthday in 2011, dating back to the first commercial robot in 1961 (the Unimate). In a “Tonight Show” from the time, this robot did amazing things: it opens a bottle of beer, pours it, puts a golf ball into the hole, and even conducts an orchestra. This robot does all what we expect a good robot to do: it is dexterous, it is accurate, and even creative. Since this robot’s appearance on the Tonight show, more than 50 years have passed — so how incredible must be the capabilities of today’s robots and what must they be able to do?

Interestingly, we just recently learned doing all the things demonstrated by Unimate autonomously. Unimate indeed did what was shown on TV, but all motions have been preprogrammed and the environment has been carefully staged. Only the advent of cheap and powerful sensors and computation has recently enabled robots to detect an object by themselves, plan motions to it and grasp it. Yet, robotics is still far away from doing these tasks with human-like performance.

This book introduces you to the computational fundamentals of autonomous robots. Robots are autonomous when they make decisions in response to their environment vs. simply following a pre-programmed set of motions. They achieve this using techniques from signal processing, control theory, and artificial intelligence, among others. These techniques are coupled with the mechanics, the sensors, and the actuators of the robot. Designing a robot therefore requires a deep understanding of both algorithms and its interfaces to the physical world.

The goals of this introductory chapter are to introduce the kind of problems roboticists deal with and how they solve it.

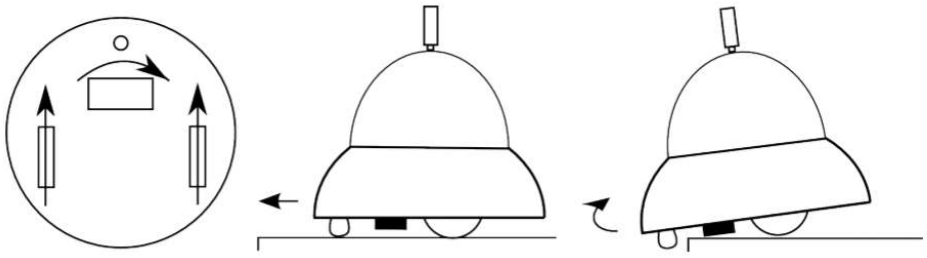


Figure **1.1**: A wind-up toy that does not fall off the table using purely mechanical control. A fly-wheel that turns orthogonal to the robot’s motion induces a right turn as soon as it hits the ground once the front caster wheel goes off the edge.

1.1: Intelligence and Embodiment

Our notion of “intelligent behavior” is strongly biased by our understanding of the brain and how computers work: intelligence is located in our heads. In fact, however, a lot of behavior that looks intelligent can be achieved by very simple means. For example, mechanical wind-up toys can avoid falling off an edge simply by using a fly-wheel that rotates at a right angle to their direction of motion and a caster wheel. Once the caster wheel loses contact with the ground—that is the robot has reached the edge—the fly-wheel kicks in and pulls the robot to the right (Figure 1.1).

A robot vacuum cleaner might solve the same problem very differently: it employs infrared sensors that are pointed downwards to detect edges such as stairs and then issues a command to make an avoiding turn. Once electronics are on-board, this is a much more efficient, albeit much more complex, approach.

Whereas the above examples provide different approaches to implement intelligent behaviors, similar trade-offs exist for robotic planning. For example, ants can find the shortest path between their nest and a food source by simply choosing the trail that already has more pheromones, the chemicals ants communicate with, on it. As shorter paths have ants not only moving faster towards the food, but also returning faster, their pheromone trails build up quicker (Figure 1.2). But ants are not stuck to this solution. Every now and then, ants give the longer path another shot, eventually finding new food sources

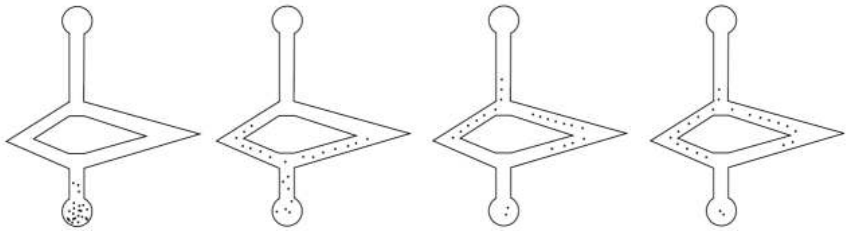


Figure **1.1.1**: *Ants finding the shortest path from their nest (bottom) to a food source (top). From left to right: The ants initially have equal preference for the left and the right branch, both going back and forth. As ants return faster on the shorter branch there will be more pheromones present on the short branch once a new ant arrives from the nest.*

What looks like intelligent behavior at the swarm level, is essentially achieved by a pheromone sensor that occasionally fails. A modern industrial robot would solve the problem completely different: it would first acquire some representation of the environment in the form of a map populated with obstacles, and then plan a path using an algorithm. Which solution to achieve a certain desired behavior is best depends on the resources that are available to the designer. We will now study a more elaborate problem for which many, more or less efficient, solutions exist.

1.2: A Roboticists' Problem

Imagine the following scenario. You are a robot in a maze-like environment such as a cluttered warehouse, hospital or office building. There is a chest full of gold coins hidden somewhere inside. Unfortunately, you don't have a map of the maze. In case you find the chest, you may only take a couple of coins at a time, and bring them to the exit door where your car is parked.

Query

Think about a strategy that will allow you to harvest as many coins in the shortest time as possible. Think about the cognitive and perception capabilities you would make use of. Now discuss alternative strategies, if you would not have these capabilities, i.e., what if you were blind, had no memory?

These are exactly the same problems a robot would have. A robot is a mobile machine that has sensors and computation, which allows it to reason about its environment. Current robots are far from the capabilities that humans have, therefore it makes a lot of sense to think about what strategies you would employ to solve a problem, if you were lacking important perception or computational capabilities.

Before we move forward to discuss potential strategies for robots with impeded sensory systems, let's quickly consider an optimal strategy. You will need to explore the maze without entering any branch twice. You can use a technique known as depth-first search to do this, but will need to be able to not only map the environment, but also localize in the environment, e.g., by recognizing places and dead-reckoning on the map. Once you have found the gold, you will need to plan the shortest path back to the exit, which you can then use to go back and forth until all the gold is harvested.

1.3: Ratslife

Ratslife is a miniature robot maze competition developed by Olivier Michel from Cyberbotics S.A. The Ratslife environment can easily be created from LEGO bricks, card board or wood and the game can be played with any two mobile robots, preferably ones with the ability to identify markers in the environment. These include simple differential-wheel educational platforms with onboard cameras or even a smart-phone driven robot. Figure 1.3 shows a simple sample environment that can be constructed from craft materials and can be used to teach the practical aspects of mobile robots for competitions.

In RatsLife, two miniature robots compete on searching for four “feeders” that are hidden in a maze. Once a robot reaches a feeder, it receives “energy” to go on for another 60s, and the feeder becomes temporarily unavailable. After a short while, the feeder becomes available again. The feeders can be either controlled by a referee who also takes care of time-keeping or constructed as part of a simple curriculum on electronics or mechatronics.

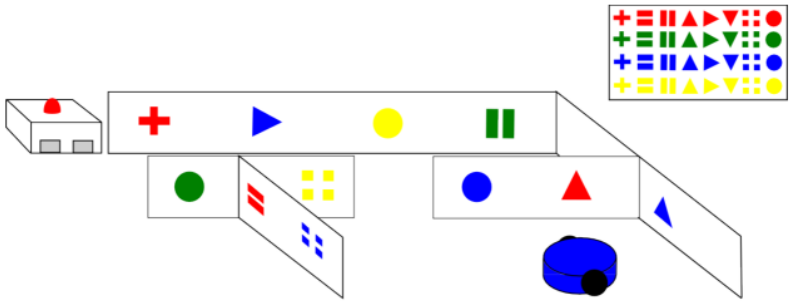


Figure **1.3.1**: A simple maze made from cardboard, wood or Lego bricks with one or more charging stations. Locations in the maze are marked with unique markers that can be recognized by a simple robot.

It should be clear by now, how YOU would solve these tasks using your abilities, and you should have also thought about fall-back strategies in case some of your sensors are unavailable. Here are some possible algorithms for a robot, ordered after the capabilities that it provides:

- Imagine you have a robot that can only drive (actuation) and bounce off a wall. The resulting random walk will eventually let the robot reach a feeder. As the allowed time to do so is limited, it is likely that the robot’s energy will soon deplete.
- Now imagine a robot that has a sensor that gives it the ability to estimate its distance from a wall. This could be a whisker, an infrared distance sensor, an ultra-sound distance sensor, or a laser range finder. The robot could now use this sensor to keep following a wall to its right. Using this strategy for solving the maze, it will eventually explore the entire maze except for islands inside of it.
- Finally, think about a robot that could identify simple patterns using vision, has distance sensors to avoid walls, and an “odometer” to keep track of its wheel rotations. Using these capabilities, a potential winning strategy would be to explore the environment, identify markers in the environment using vision and use them to create a map of all feeder locations, calculate the shortest path from feeder to feeder and keep going back and forth between them. Strategy-wise, it might make sense to wait just in front of the feeder and approach it only shortly before the robot runs out of power.

1.4: Challenges of Mobile Autonomous Robots

Being able to stitch sensor information together to map the environment just by counting your own steps and orienting yourself by using distinct features of the environment is known as Simultaneous Localization and Mapping (SLAM). The key challenge here is that the length of the steps you take are uncertain (a wheeled robot might slip or have slightly differently sized wheels, e.g.) and it is not possible to recognize places with 100% accuracy (not even for a human). In order to be able to implement something like the last algorithm on a real robot, we will therefore need to understand

- How does a robot move? How does rotation of its wheels affect its position and speed in the world?
- How do we have to control the wheel-speed in order to reach a desired position?
- What sensors exist for a robot to perceive its own status and its environment?
- How can we extract structured information from a vast amount of sensor data?
- How can we localize in the world?
- How can error be represented and how can we reason in the face of uncertainty?

In order to answer these questions, we will rely on trigonometry, linear algebra, and probability theory. Specific concepts that will be used throughout this book are basic trigonometry, matrix notation, Bayes' formula, and the concept of probability distributions. You will see that robotics is actually a great vehicle to add meaning to these concepts!

1.5: Challenges of Autonomous Manipulation

Think about the last time you worked with your hands. This includes typing on your keyboard, writing on a piece of paper, sewing a button onto a shirt, and using a hammer or a screwdriver. You will notice that these activities require a wide range of dexterity, that is the ability to manipulate objects with precision, a wide range of forces, and a wide range of sensorial capabilities. You will also notice that some tasks go beyond your capabilities, such as putting yarn through a hole in fabric, grasping a screw, or driving a nail into a piece of wood, but can be easily solved with the right tool.

So far, robotic hands are far from reaching the dexterity of a human hand. Yet, with the right tool (called “end-effector” in robotics speech) some tasks can be solved even better, that is faster and more precisely, than by humans. As for solving a mobile robotics problem, manipulation problems require you to think about the right mix of reasoning and mechanism design. For example, grasping tiny parts might be impossible with tweezers, but really easy when using a sucking mechanism. Or, picking up a test tube that is hardly visible with the robots’ sensors can be picked up almost blindly when using a funnellike mechanism at your end-effector. Unfortunately, these tricks will most likely limit the versatility of your robot, requiring you to think about the problem and the users’s need as a whole.

1.6: Exercises

Take-Home Lessons

- How to best solve a problem is a function of the available sensing, actuation, computation and communication abilities of the available platform. Usually, there exist trade-offs that allow you to solve a problem using a minimal set of resources, but compromise performance such as speed, accuracy or reliability.
- Robotics problems are different from problems in pure Artificial Intelligence, that do not deal with unreliable sensing or actuation.
- The unreliability of sensors, actuators and communication links require a probabilistic notion of the system and reason with uncertainty.

Exercises

1. What kind of sensors do you need to solve the “Ratslife” game? Think both about trivial and close-to-optimal approaches.
2. What devices in your home could be considered robots? Why and why not?
3. Which industries have been recently revolutionized by robotics? Into which industries were robots introduced first?
4. What sensors are you using when grasping an object? Enumerate them all. Which ones are absolutely necessary for good performance?
5. Think about robots vacuuming your floor or mowing your lawn. Do they use any planning? Why or why not?
6. What kind of sensors would you need in a car that drives completely autonomously? Think first about the kind of information that the car needs to be aware of and then discuss possible sensors that could capture this information.
7. Implement a simple line-following using a robot of your choice. How does the thickness of the line affect the sensor placement on the robot? How does its curvature affect the robot’s speed?
8. Implement a maze solving algorithm that uses simple wallfollowing using a robot of your choice. How does the sensor geometry affect the robot’s performance? What are the parameters that you find yourself tuning?

CHAPTER OVERVIEW

2: Locomotion and Manipulation

Autonomous robots are systems that sense, actuate, compute, and communicate. Actuation, the focus of this chapter, is the ability of the robot to move and to manipulate the world. Specifically, we differentiate between locomotion as the ability of the robot to move and manipulation as the ability to move objects in the environment of the robot. Both activities are closely related: during locomotion the robot uses its motors to exert forces on its environment (ground, water or air) to move itself; during manipulation it uses motors to exert forces on objects to move them relative to the environment. This might not even require different motors. Insects are good examples for this: both can use their 6 legs not only for locomotion, but also for picking up and manipulating objects. The goals of this chapter are

- introduce the concepts of locomotion, manipulation and their duality
- explain static vs. dynamic stability
- introduce “degrees-of-freedom”
- and introduce forward kinematics of static arms.

2.1: Locomotion and Manipulation Examples

Locomotion includes very different concepts of motion including rolling, walking, running, jumping, sliding (undulatory locomotion), crawling, climbing, swimming, and flying. They are drastically different in terms of energy consumption, kinematics, stability, and capabilities required by the robot that implements them. Yet, the above definitions are loose and ambiguous: for example, “swimming” can be done using many different forms of propulsion systems. Similarly, a sliding motion on the ground might result into swimming with only few modifications.

The way in which the individual parts of a robot can move with respect to each other and the environment is called the kinematics of the robot. Kinematics are only concerned with the position and speed (first derivative of position) of those parts, but not its dynamics, which include acceleration (second derivative of position) and jerk (third derivative of position).

Commercially, the most dominant form of locomotion is rolling. This is due to the fact that rolling provides by far the most efficient energy-speed ratio (Figure 2.1), making the invention of the wheel one of the greatest technological breakthroughs in history. Consequently, humans have modified their environment to have smooth surfaces of large extent such as the road network, but also warehouse and residential floors. In contrast, evolution has not evolved a single animal with wheel-like actuators.

Query

Can you find examples of robots from the above categories? Identify the different types of actuators that are used in them.

Due to the dominance of rolling robots, the electric motor is among the most popular actuators. Except for the stepper motor, which uses large electromagnets to rotate an internal spindle by a few degrees every time, the physics of the electrical motor requires it to revolve at very high speeds (multiple thousand rotations per minute). Therefore, motors are almost always used in conjunction with gears to reduce the speed and increase the torque, that is the force that the motor can exert to rotate an axis. In order to be able to measure the number of revolutions and the axis’ position, motors are also often combined with rotary encoders. Motors that combine an electric motor with a gear-box, encoder, and controller to move toward desired position are known as servo motors, and are popular among hobbyists. Another popular class of actuator, in particular for legged robots, are linear actuators, that might exist in electric, pneumatic or hydraulic form. Finally, there exist a wide array of specialty actuators such as Shape-Memory Alloys, Electroactive Polymers or Piezo-elements, which often allow for extreme miniaturization, but do not provide attractive energyto-force ratios and are difficult to control.

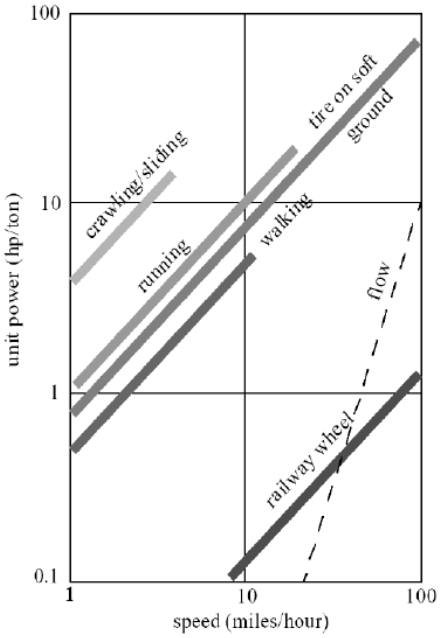


Figure *PageIndex*1: Power consumption vs. speed for various means of locomotion. From Todd (1985).

Most actuators (and mechanisms) capable of locomotion can also be used for manipulation with only minor modifications. Most industrial manipulators consist of a chain of rotary actuators that are connected by links. Most industrial robots have six or more independently rotating axes. We will see why further down below. Modern industrial manipulators have the ability to not only control the position of each of its joints, but precisely control the torque and force at each individual joint, making the arm arbitrary compliant, which is the inverse of stiffness in a mechanical sense. For dexterous manipulation a robot does not only need an arm, but also a gripper or hand. Grasping is a hard problem on its own and deserves its own chapter.

2.2: Static and Dynamic Stability

A fundamental difference between locomotion mechanisms is whether they are statically or dynamically stable. A statically stable mechanism will not fall even when all of its joints freeze (Figure 2.2.1, left). A dynamically stable robot instead requires constant motion to prevent it from falling. Technically, stability requires the robot to keep its center of mass to fall within the polygon spanned by its ground-contact points. For example a quadruped robot's feet span a rectangle. Once such a robot lifts one of its feet, this rectangle becomes a triangle. If the projection of the center of mass of the robot along the direction of gravity is outside of this triangle, the robot will fall. A dynamically stable robot can overcome this problem by changing its configuration so rapidly that a fall is prevented. An example of a purely dynamically stable robot is an inverted pendulum on a cart (Figure 2.2.1, middle). Such a robot has no statically stable configurations and needs to keep moving all the time to keep the pendulum upright. While dynamic stability is desirable for high-speed, agile motions, robots should be designed so that they can easily switch into a statically stable configuration (Figure 2.2.1, right).

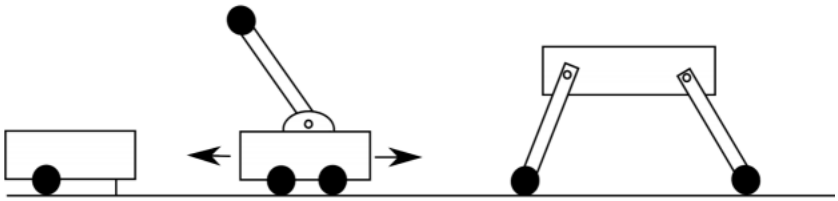


Figure **2.2.1**: *From left to right: statically stable robot. Dynamically stable inverted pendulum robot. Static and dynamically stable robot (depending on configuration).*

An example of a robot that has both statically and dynamically stable configurations is a quadruped ("four legs") runner. Unlike walking, a running robot will always have two legs in the air and alternate between them faster than the robot could fall in either direction. Although statically stable walking is possible with only 4 legs, most animals (and robots) require 6 legs for statically stable walking and use dynamically stable gaits (such as galloping) when they have four legs. Six legs allow the animal to move three legs at a time while the three other legs maintain a stable pose.

2.3: Degrees-of-Freedom

The concept of degrees-of-freedom, often abbreviated as DOF, is important for defining the possible positions and orientations a robot can reach. An object in the physical world can have up to six degrees of freedom, namely forward/backward, sideways, and up/down as well as rotations around those axes. These rotations are known as pitch, yaw and roll and are illustrated in Figure 2.3.1.

How many of those directions a robot can move in depends on the configuration of its actuators and the constraints the robot has with the environment. These relationships are not always intuitive and require more rigorous mathematical treatment (Chapter 3). The goal of this section is to introduce the degrees of freedom of standard mechanisms that are recurrent in robot design such as wheels or simple arms. For wheeled platforms, the degrees-of-freedom are defined by the types of wheels used and their orientation. Common wheel types are listed in Table 2.3.1.

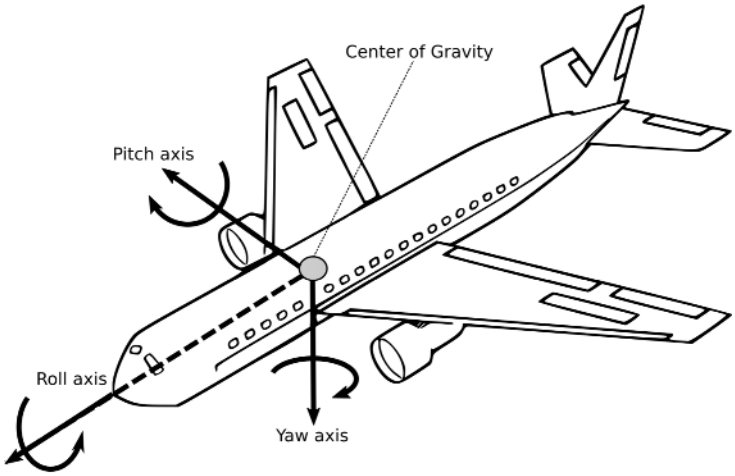


Figure **2.3.1**: *Pitch, yaw and roll around the principal axis of an airplane.*

Only robots that use exclusively wheels with three degrees-offreedom (3-DOF wheels) will be able to freely move on a plane. This is because the pose of a robot on a plane is fully given by its position (two values) and its orientation (one value). Robots that don't have wheels with three degrees of freedom will have kinematic constraints that prevent them from reaching every possible point at every possible orientation. For example, a bicycle wheel can only roll into one direction and turn on the spot. Moving the bicycle wheel orthogonal to its direction of rolling is not possible, unless it is forcefully dragged ("skidding"), which requires more involved treatment not covered in this book. On the other hand, not having three degrees of freedom does not mean that not all poses in the plane can be reached. A good analogue are figures on a chess-board. For example, a knight can reach every cell on a

chess-board but might require multiple moves to do so. This is similar to a car, which can parallel park using back-and-forth motions. Instead, a bishop can only reach either black or white fields on the board.

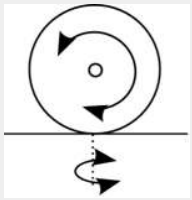
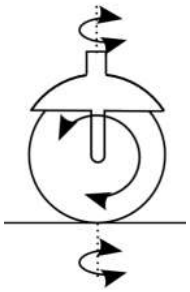
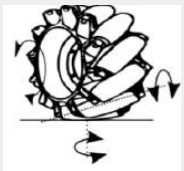
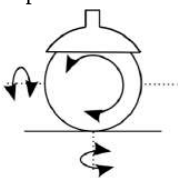
Wheel Type	Example	Degrees-of-Freedom
Standard 	Front-wheel of a wheelbarrow	Two • Rotation around the wheel axle • Rotation around its contact point with the ground
Caster wheel 	Office chair	Three • Rotation around the wheel axle • Rotation around its contact point with the ground • Rotation around the caster axis
Swedish wheel 	Standard wheel with non-actuated rollers around its circumference	Three • Rotation around the wheel axle • Rotation around its contact point with the ground • Rotation around the roller axles
Spherical wheel 	Ball Bearing	Three • Rotation in any direction • Rotation around its contact point

Table 2.3.1: Different types of wheels and their degrees of freedom. Adopted from Siegwart et al. (2011),

Similar reasoning applies to aerial and underwater robots. Here, the position of the robot is affected by the position and orientation of thrusters, either in the form of jets or propellers, mounted on the robot. Things become complicated quickly, however, as the dynamics of the system are subject to fluidand aerodynamic effects, which also change as a function of size of the robot. This book will not go into the details of flying and swimming robots, but the general principles of localization and planning will be applicable to them as well.

Query

Think about possible wheel, propeller and thruster configurations. Don't limit yourself to robots, but consider also street and aerial vehicles and be creative — if you can think about a setup that makes sense, i.e., allows for reasonable mobility — somebody will already have built it and analyzed it. What are the advantages and disadvantages of each?

For manipulating arms, degrees of freedom usually refer to the positions and orientations, i.e., rotations around the primary axes, the end-effector can reach. As a rule of thumb, each joint usually adds a degree of freedom unless they are redundant, that is,

moving in the same direction. Figure 2.4 shows a series of manipulators operating in a plane. By this, the degrees of freedom of the end-effector are limited to moving up and down, sideways, and rotating around its pivot point. As a plane only has those three degrees of freedom, adding additional joints cannot increase the degrees of freedom unless they allow the robot to also move in and out of the plane.

An exact definition of the number of degrees of freedom is tricky and requires deriving analytical expressions for the endeffector position and orientation, which will be subject to Chapter 3.

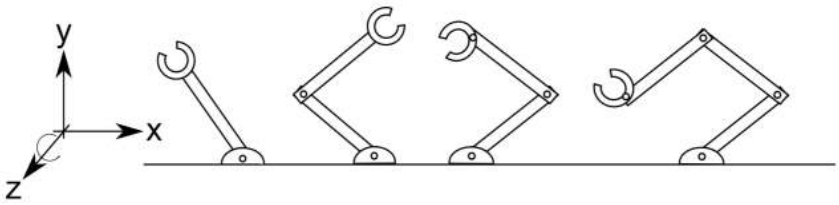


Figure **2.3.2**: From left to right: Manipulators with one, two, three and three DOF. The degrees of freedom of moving in a plane are the position of the end-effector with respect to its height and displacement with respect to the base, as well as its orientation.

Choosing the “right” kinematics is a trade-off between mechanical complexity, maneuverability, achievable precision, cost, and ease of control. The very popular differential-wheel drive consisting of two independently controlled wheels that share a common axis such as on the iRobot Roomba is cheap, highly maneuverable and easy to control, but makes it hard to drive in a straight line. This requires both motors to turn at the exact same speed and both wheels to have the exact same diameter, which is hard to achieve in practice. This problem is solved well by car-like steering mechanisms, but they have poor maneuverability and are difficult to control (think parallel-parking).

2.4: Exercises

Take-home lessons

- In order to do planning for a robot, you need to understand how its control parameters map to actions in the physical world.
- The kinematics of a robot are fully defined by the position and orientation of its wheels, joints and links no matter whether it swims, flies, crawls or drives.
- Many robotic systems cannot be fully understood by considering kinematics alone, but require you to model their dynamics as well. This book will be limited to modeling kinematics, which is sufficient for low-speed, mobile robots and arms.

Exercises

1. What are the degrees of freedom of a lawnmower with four standard wheels? Why are you still able to mow your entire lawn?
2. Is a car statically or dynamically stable? What about a Segway?
3. What are the degrees of freedom of an office chair with all caster-wheels?
4. What are the maximum degrees of freedom for objects driving on the plane?
5. What are the maximum degrees of freedom for objects that can freely move in the world?
6. Calculate the degrees of freedom of a differential wheels robot with a front caster wheel. What happens when you add a second caster wheel?
7. Calculate the degrees of freedom of a standard car. How can you still reach every point on the plane?
8. A steering wheel allows you to change the yaw of your car. Can you also change its pitch and its roll?

CHAPTER OVERVIEW

3: Forward and Inverse Kinematics

In order to plan a robot's movements, we have to understand the relationship between the actuators that we can control and the robot's resulting position in the environment. For static arms, this is rather straightforward: if we know the position/angle of each joint, we can calculate the position of its end-effectors using trigonometry. This process is known as forward kinematics. If we want to calculate the position each joint needs to be at, we need to invert this relationship. This is known as inverse kinematics. For mobile robots, this process is usually more involved, as speeds need to be integrated, which we refer to as *odometry*.

The goals of this chapter are:

- introduce coordinate systems and their transformations
- to introduce the forward kinematics of simple arms and mobile robots
- understand the concept of holonomy
- show how solutions for the inverse kinematics for both static and mobile robots can be derived
- provide an intuition on the relationship between inverse kinematics and path-planning

3.1: Coordinate Systems and Frames of Reference

Every robot assumes a position in the real world that can be described by its position (x, y and z) and orientation (pitch, yaw and roll) along the three major axes of a Cartesian Coordinate system (See also Section 2.3, “Degrees of freedom”).

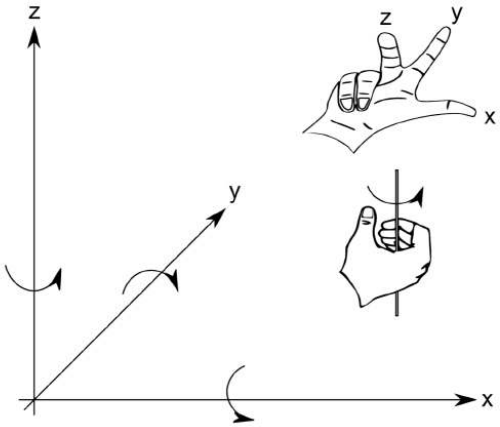


Figure **3.1.1**: A coordinate system indicating the direction of the coordinate axes and rotation around them. These directions have been derived using the right-hand rules.

Such a coordinate system is shown in Figure 3.1.1. Note that the directions and orientations of the coordinate axes are arbitrary. This book uses the “right hand rules”, which are illustrated in Figure 3.1.1 to determine axes labels and directions throughout. Pitch, yaw, and roll, are also known as bank, attitude, and heading in other communities. This makes sense, considering the colloquial use of the word “heading”, which corresponds to a rotation around the z-axis of a vehicle driving on the x-y plane.

Defining all three position axes and orientations might be cumbersome. What level of detail we care about, where the origin of this coordinate system is, and even what kind of coordinate system we choose, depends on the specific application. For example, a simple mobile robot would typically require a representation with respect to a room, a building, or the earth’s coordinate system (given by the longitude and latitude of each point on the earth), whereas a static manipulator usually has the origin of its coordinate system at its base. More complicated systems, such as mobile manipulators or multi-legged robots, make life much easier by defining multiple coordinate systems, e.g. one for each leg and one that describes the position of the robot in the world frame. These local coordinate systems are known as Frames of Reference. An example of two nested coordinate systems is shown in Figure 3.1.2. In this example, a robot located at the origin of x' , y' and z' might plan its motions in its own reference frame, which can then be expressed in the coordinate system x, y and z by performing a translation and a rotation as we will later see.

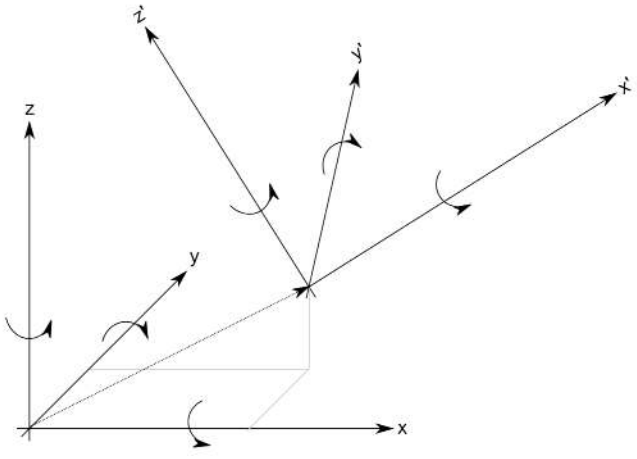


Figure **3.1.2**: Two nested coordinate systems (frames of reference).

Depending on its degrees-of-freedom, that is the number of independent translations and rotations a robot can achieve in Cartesian space, it is also customary to ignore components of position and orientation that remain constant. For example a simple floor-

cleaning robot's pose might be completely defined by its x and y coordinates in a room as well as its orientation, i.e. its rotation around the z-axis.

3.1.1. Matrix notation

Given some kind of fixed coordinate system, we can describe the position of a robot's end-effector by a 3x1 position vector. As there can be many coordinate systems defined on a robot and the environment, we identify the coordinate system a point relates to by a preceeding super-script, e.g., ${}^{A}P$ to indicate that point P is in coordinate system {A}. Each point consists of three elements ${}^{A}P = [p_x p_y p_z]^T$.

More formally, ${}^{A}P$ is a linear combination of the three basis vectors that span A:

$$ {}^{A}\mathbf{P} = p_x \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + p_y \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + p_z \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{3.1.1} $$

As we know, not only the position of the robot is important, but also its orientation. In order to describe the orientation of a point, we will attach a coordinate system to it. Let $\hat{X}_B$, $\hat{Y}_B$ and $\hat{Z}_B$ be unit vectors that correspond to the principal axes of a coordinate system {B}. When expressed in coordinate system {A}, they are denoted ${}^{A}\hat{X}_B$, ${}^{A}\hat{Y}_B$ and ${}^{A}\hat{Z}_B$. In order to express a vector that is given in one coordinate system in another, we need to project each of its components to the unit vectors that span the target coordinate system. For example considering only the axis ${}^{A}\hat{X}_B$

$$ {}^{A}\hat{X}_B = \left(\hat{X}_B \cdot \hat{X}_A, \hat{X}_B \cdot \hat{Y}_A, \hat{X}_B \cdot \hat{Z}_A\right)^T \tag{3.1.2} $$

consists of the projections of $\hat{X}_B$ onto $\hat{X}_A$, $\hat{Y}_A$ and $\hat{Z}_A$. Here, $| \cdot |$ denotes the scalar product (also known as dot or inner product). Note that all vectors in (3.1.2) are unit vectors, i.e. their length is one. By definition of the scalar product, $A \cdot B = ||A|| \, ||B|| \cos \alpha = \cos \alpha$, indeed reduces the projection of $\hat{X}_B$ onto the unit vectors of {A}. This projection is illustrated in Figure 3.1.3.

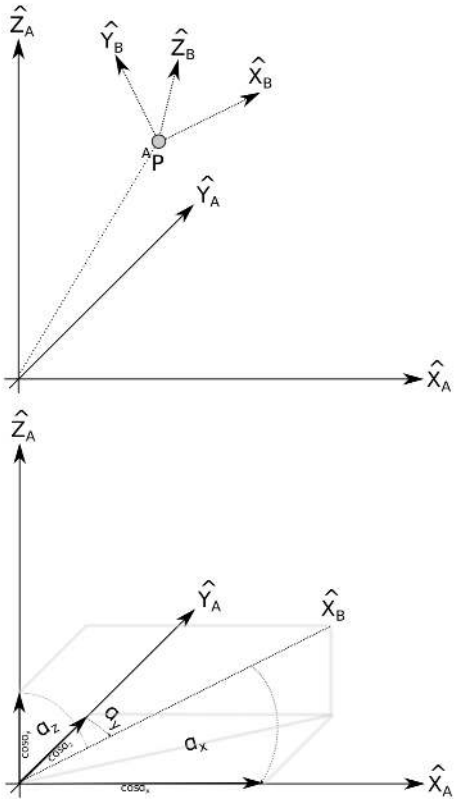


Figure **3.1.3**: Top: A coordinate system {B} with position given by ${}^{A}P$ and orientation given by $\hat{X}_B$, $\hat{Y}_B$, and $\hat{Z}_B$. Bottom: The projection of the unit vector $\hat{X}_B$ onto the unit vectors that span coordinate system {A} after moving {B} into the origin of {A}. As all vectors are unit vectors, $A \cdot B = ||A|| \, ||B|| \cos \alpha = \cos \alpha$.

We can now do this for all three vectors that span coordinate system {B} and stack these three vectors together into a 3x3 matrix to obtain the rotation matrix

$$ {}^{A}_{B}\mathbf{R} = \begin{bmatrix} {}^{A}\hat{X}_{B} & {}^{A}\hat{Y}_{B} & {}^{A}\hat{Z}_{B} \end{bmatrix} \tag{3.1.3} $$

which describes {B} relative to {A}. It is important to note that all columns in ${}_{B}^{A}\mathrm{R}$ are unit vectors, so that the rotation matrix is orthonormal. This is important as this allows us to easily obtain the inverse of ${}_{B}^{A}\mathrm{R}$ as ${}_{B}^{A}\mathrm{R}^{T}$ or ${}_{A}^{B}\mathrm{R} = {}_{B}^{A}\mathrm{R}^{T}$.

Why the unit vectors of a coordinate system {B} expressed in coordinate system {A} actually make up a rotation matrix, can be easily seen when re-arranging Equation 3.1.1 in matrix form

$$ {}^{A}\mathbf{P} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \tag{3.1.4} $$

where the rotation matrix is nothing but the identity as both points already are in the same coordinate system.

We have now established how to express the orientation of a coordinate system using a rotation matrix. Usually, coordinate systems don't lie on top of each other, but are also displaced from each other. Together, position and orientation is known as a frame, which is a set of four vectors, one for the position and three for the orientation, and we can write

$$ \{B\} = \left\{ {}^{A}_{B}\mathbf{R}, {}^{A}\mathbf{P} \right\} \tag{3.1.5} $$

to describe the coordinate frame {B} with respect to {A} using a vector ${}^{A}\mathrm{P}$ and a rotation matrix ${}_{B}^{A}\mathrm{R}$. Robots usually have many such frames defined along their bodies.

3.1.2. Mapping from one frame to another

Having introduced the concept of frames, we need the ability to map coordinates in one frame to coordinates in another frame. For example, let's consider frame {B} having the same orientation as frame {A} and sitting at location ${}^{A}\mathrm{P}$ in space. As the orientation of both frames is the same, we can express a point ${}^{B}\mathrm{Q}$ in frame {A} as

$$ {}^{A}\mathbf{Q} = {}^{B}\mathbf{Q} + {}^{A}\mathbf{P} \tag{3.1.6} $$

Actually, adding two vectors that are in different reference frames, i.e., ${}^{B}\mathrm{Q} + {}^{A}\mathrm{P}$, is only possible if both of them have the same orientation. We can, however, convert from one reference frame to the other using the rotation matrix:

$$ {}^{A}\mathbf{P} = {}^{A}_{B}\mathbf{R}\,{}^{B}P \tag{3.1.7} $$

and therefore solve the mapping problem regardless of the orientation of {A} to {B}:

$$ {}^{A}\mathbf{Q} = {}^{A}_{B}\mathbf{R}\,{}^{B}Q + {}^{A}\mathbf{P} \tag{3.1.8} $$

Using this notation, we can see that leading subscripts cancel the leading superscripts of the following vector/rotation matrix. Whereas we have now a solution to transfer a point from one frame of reference to another by combining a rotation and a translation, it would be more appealing to write something like that:

_{}^{A}\textrm{Q}=_{B}^{A} \textrm{T}^{B}Q

In order to do this, we need to introduce a 4x1 position vector such that

$$ \left[{}^{A}Q \right] = \left[\begin{array}{ccc|c} & {}^{A}_{B}R & & {}^{A}P \\ \hline 0 & 0 & 0 & 1 \end{array} \right] \begin{bmatrix} {}^{B}Q \\ 1 \end{bmatrix} $$

and ${}_{B}^{A}\mathrm{T}$ is a 4x4 matrix. Note that the added '1's and [0001] do not affect the other entries in the matrix during matrix multiplication. A 4x4 matrix of this form is called a *homogenous transform*.

The inverse of an homogeneous transform can be constructed by inverting rotation and translation part independently, leading to

$$\left[\begin{array}{ccc|c} {}^A_B R & & & {}^A P \\ \hline 0 & 0 & 0 & 1 \end{array}\right]^{-1} = \left[\begin{array}{ccc|c} {}^A_B R^T & & & -{}^A_B R^T {}^A P \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

We have now established a convenient notation to convert points from one coordinate system to another. There are many possible ways this can be done, in particular how rotation can be represented (see below), but all can be converted from one into the other.

3.1.3. Transformation arithmetic

Transformations can be combined: consider for example an arm with two links, reference frame {A} at the base, {B} at its first joint, and {C} at its end-effector. Given the transforms ${}_C^B\mathrm{T}$ and ${}_B^A\mathrm{T}$, we can write

$$^A\mathbf{P} = {}^A_B\mathbf{T}\,{}^B_C\mathbf{T}\,{}^C\mathbf{P} = {}^A_C\mathbf{T}\,{}^C\mathbf{P} \tag{3.1.9}$$

to convert a point in the reference frame of the end-effector to that of its base. As this works for rotation and translation operators independently, we can construct ${}_C^A\mathrm{T}$ as

$${}^A_C T = \left[\begin{array}{ccc|c} {}^A_B R\,{}^B_C R & & & {}^A_B R\,{}^B P_C + {}^A P_B \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

where ${}^A\mathrm{P}_B$ and ${}^B\mathrm{P}_C$ are the translations from {A} to {B} and from {B} to {C}, respectively.

3.1.4. Other representations for orientation

So far, we have represented orientation by a 3x3 matrix whose column vectors are orthogononal unit vectors describing the orientation of a coordinate system. Orientation is therefore represented with nine different values. We chose this representation mainly because it is the most intuitive to explain and is derived from simple geometry.

In fact, three values are sufficient to describe orientation. This becomes clear when considering that orthogonality (dot product of all columns is zero) and vector length (each vector must have length 1) impose six constraints on the nine values in the rotation matrix. Indeed, an orientation can be represented about a rotation by certain angles around the x, the y, and the z-axis of the reference coordinate system. This is known as the X − Y − Z fixed angle notation. Mathematically, this can be represented by a rotation matrix of the form

$${}^A_B\mathrm{R}_{XYZ(\gamma,\beta,\alpha)} = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix} \tag{3.1.10}$$

While the X −Y −Z fixed angles approach expresses a coordinate frame using rotations with respect to the original coordinate frame, say {A}, another possible description is to start with a coordinate frame {B} that is coincident with frame {A}, then rotate around the Z-axis with angle α, then the Y-axis with angle β and finally around the X-axis with angle γ. This representation is called Z-Y-X Euler angles. As the coordinate axis do not necessarily need to be different, there are twelve possible valid combinations of sub-sequent rotations:

XYX, XZX, YXY, YZY, ZXZ, ZYZ, XYZ, XZY, YZX, YXZ, ZXY and ZYX

There are only twelve, as sub-sequent rotations around the same axis are not valid. Such rotations would not add any information, but are equivalent to a rotation by the sum of both angles.

It is important to know about the subtle differences between the different available transformations as there is no “right” or “wrong”, but different manufacturers and fields use different conventions. There is only one caveat: each of the rotation matrices can look like subsequent rotations around the same axis for certain values of angles. For example, this happens for the XYZ rotation matrix if the angle of rotation around the Y-axis is 90° . These cases are known as a *singularity*.

Among these, the preferred representation for computational and stability reasons are Quaternions. A quaternion is a 4-tuple that extends the complex numbers with very general applications in mathematics and representing orientation and rotation in particular. The basic idea is that each rotation can be represented as a rotation around a single axis (a vector in space) by a specific angle. Given such an axis $K = [k_x k_y k_z]\ T$ and an angle θ, one can calculate the so-called Euler parameters or unit quaternion:

$$\epsilon_1 = k_x sin\frac{\theta}{2} \tag{3.1.11}$$

$$\epsilon_2 = k_y sin\frac{\theta}{2} \tag{3.1.12}$$

$$\epsilon_3 = k_z sin\frac{\theta}{2} \tag{3.1.13}$$

$$\epsilon_4 = coz\frac{\theta}{2} \tag{3.1.14}$$

These four quantities are constrained by the relationship

$$\epsilon_1^2 + \epsilon_2^2 + \epsilon_3^2 + \epsilon_4^2 = 1 \tag{3.1.15}$$

which might be visualized by a point on a unit hyper-sphere. Analogous to rotation matrices, two quaternions ϵ_i and ϵ'_i can be multiplied using the following equation

$$\begin{pmatrix} \epsilon_4 & \epsilon_1 & \epsilon_2 & \epsilon_3 \\ -\epsilon_1 & \epsilon_4 & -\epsilon_3 & \epsilon_2 \\ -\epsilon_2 & \epsilon_3 & \epsilon_4 & -\epsilon_1^c \\ -\epsilon_3 & -\epsilon_2 & \epsilon_1 & \epsilon_4 \end{pmatrix} \begin{pmatrix} \epsilon_4' \\ \epsilon_1' \\ \epsilon_2' \\ \epsilon_3' \end{pmatrix} \tag{3.1.16}$$

Unlike multiplying two rotation matrices, which requires 27 multiplications and 18 additions, multiplying two quaternions only requires 16 multiplications and 12 additions, making the operation computationally more efficient. In addition, the quaternion representation does not suffer from singularities for specific joint angles, making the approach computationally more robust.

Why any rotation can be expressed by a single vector can be seen when considering the properties of orthonomal rotation matrices. They have three Eigenvalues $\lambda = 1$ and a complex pair $\lambda 1,2 = \cos\theta \pm i \sin\theta$. Eigenvalues and Eigenvectors are defined as $Rv = \lambda v$. For the case of $\lambda = 1$, the corresponding Eigenvector v is unchanged by rotation. This is only possible if v is the actual axis of rotation. The angle of rotation is now given by θ, which can be inferred from the complex pair.

3.2: Forward kinematics of selected Mechanisms

Now that we have introduced the notion of local coordinate frames, we are interested in how to calculate the pose and speed of these coordinate frames as a function of the robot's actuators. We will first consider simple mechanisms where we can determine the relationship between actuators and the pose of various frames on the robot both in the position and speed domain. We will then consider the special class of non-holonomous mechanisms using a series of wheeled robots, for which the forward kinematics can only be calculated in the speed domain.

3.2.1. Forward kinematics of a simple arm

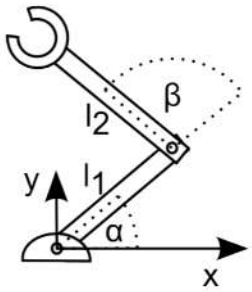


Figure **3.2.1**: *A simple 2-DOF arm.*

Consider a robot arm made out of two links and two joints that is mounted to a table. Let the length of the first link be l_1 and the length of the second link be l_2. You could specify the position of the link closer to the table by the angle α and the angle of the second link relative to the first link using the angle β. Suitable conventions and coordinate systems are shown in Figure 3.2.1.

We can now calculate the position of the joint between the first and the second link using simple trigonometry:

$$x_1 = \cos\alpha l_1 \tag{3.2.1}$$

$$y_1 = \sin\alpha l_1 \tag{3.2.2}$$

Similarly, the position of the end-effector is given by

$$x_2 = \cos(\alpha+\beta) l_2 + x_1 \tag{3.2.3}$$

$$y_2 = \sin(\alpha+\beta) l_2 + y_1 \tag{3.2.4}$$

or together, the position of the end-effector (x, y) is given by

$$x = \cos(\alpha+\beta) l_2 + \cos\alpha l_1 \tag{3.2.5}$$

$$y = \sin(\alpha+\beta) l_2 + \sin\alpha l_1 \tag{3.2.6}$$

The above equations are the kinematic equations of this robot as they relate its control parameters α and β to the position of its end-effector given in the local coordinate system spanned by x and y with the origin at the robot's base. Note that both α and β shown in the figure are positive: Both links rotate around the z-axis. Using the right-hand rule, the direction of positive angles is defined to be counter-clockwise.

The *configuration space*,i.e., the set of angles each actuator can be set to, of this robot is given by $-\pi/2 < \alpha < \pi/2$ as it is not supposed to run into the table, and $-\pi < \beta < \pi$. The configuration space is given with respect to the robot's joints and allows us to calculate the *workspace* of the robot, i.e., the physical space it can move to, using the forward kinematic equations. This terminology will be identical for mobile robots. An example of configuration and work-space for both a manipulator and a mobile robot is shown in Figure 3.2.2.

The orientation of the arm's end-effector is given by α+β. We can now write down a transformation that includes a rotation around the z-axis

$$\begin{pmatrix} cos_{\alpha\beta} & -sin_{\alpha\beta} & 0 & \cos_{\alpha\beta} l_2 + \cos\alpha l_1 \\ sin_{\alpha\beta} & cos_{\alpha\beta} & 0 & \sin_{\alpha\beta} l_2 + \sin\alpha l_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.2.7}$$

The notation *sinαβ* and *cosαβ* are short-hand for *sin(α + β)* and *cos(α + β)*, respectively.

This transformation now allows us to translate from the robot's base to the robot's end-effector as a function of the actuator positions α and β. This transformation will be helpful if we want to calculate suitable joint angles in order to reach a certain pose or if we want to convert measurements taken relative to the end-effector back into the base's coordinate system.

3.2.2. Forward Kinematics of a Differential Wheels Robot

Whereas the pose of a robotic manipulator is uniquely defined by its joint angles—which can be made available using encoders in almost real-time—this is not the case for a mobile robot. Here, the encoder values simply refer to wheel orientation and need to be integrated over time, which will be a huge source of uncertainty as we will later see. What complicates matters is that for so-called *non-holonomic* systems, it is not sufficient to simply measure the distance that each wheel traveled, but also when each movement was executed.

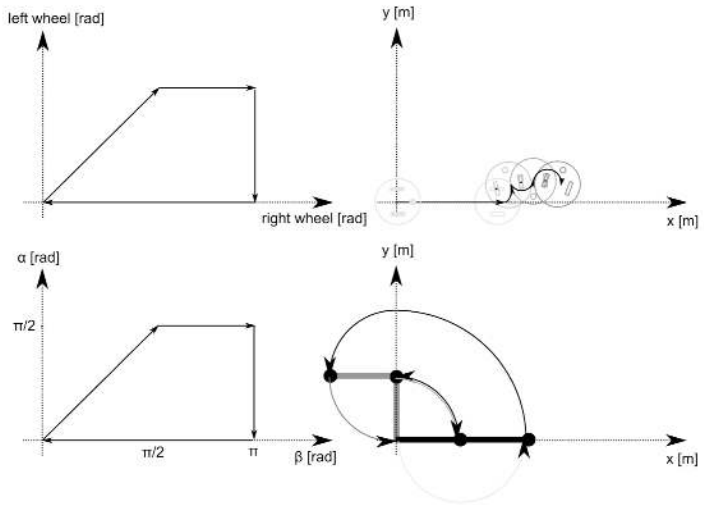


Figure **3.2.2**: Configuration space (left) and workspace (right) for a non-holonomic mobile robot (top) and a holonomic manipulator (bottom). Closed trajectories in configuration space result in closed trajectories in the workspace if the robot's kinematics is holonomic.

A system is non-holonomic when closed trajectories in its configuration space (reminder: the configuration space of a two-link robotic arm is spanned by the possible values of each angle) may not have it return to its original state. A simple arm is holonomic, as each joint position corresponds to a unique position in space. Going through whatever trajectory that comes back to the starting point in configuration space, will put the robot at the exact same position. A train on a track is holonomic: moving its wheels backwards by the same amount they have been moving forward brings the train to the exact same position in space. A car and a differential-wheel robot are non-holonomic vehicles: performing a straight line and then a right-turn leads to the same amount of wheel rotation than doing a right turn first and then going in a straight line; getting the robot to its initial position requires not only to rewind both wheels by the same amount, but also getting their relative speeds right. The configuration and corresponding workspace trajectories for a non-holonomic and a holonomic robot are shown in Figure 3.2.2. Here, a robot first moves on a straight line (both wheels turn an equal amount). Then the left wheel remains fixed and only the right wheel turns forward. Then the right wheel remain fixed and the left wheel turns backward. Finally, the right wheel turns backward, arriving at the initial encoder values (zero). Yet, the robot does not return to its origin. Performing a similar trajectory in the configuration space of a two-link manipulator instead, let the robot return to its initial position.

It should be clear by now that for a mobile robot, not only traveled distance per wheel matters, but also the speed of each wheel as a function of time. Instead, this information was not required to uniquely determine the pose of a manipulating arm. Let's introduce

the following conventions. We will establish a world coordinate system {I}, which is known as the inertial frame by convention (Figure 3.6). We establish a coordinate system {R} on the robot and express the robot's speed ${}^{R}\xi$ as a vector

$$ {}^{R}\xi = [{}^{R}x, {}^{R}y, {}^{R}\theta]^{T} $$

Here R_x and R_y correspond to the speed along the x and y directions in {R}, whereas R_θ corresponds to the rotation around the imaginary z-axis, that you can imagine to be sticking out of the ground. We denote speeds with dots over the variable name, as speed is simply the derivative of distance. Think about the robot's position in {R}. It is always zero, as the coordinate system is fixed on the robot. Therefore, velocities are the only interesting quantities in this coordinate system and we need to understand how velocities in {R} map to positions in {I}, which we denote by ${}^{I}\xi = [{}^{I}x, {}^{I}y, {}^{I}\theta]^{T}$. These coordinate systems are shown in Figure 3.2.3.

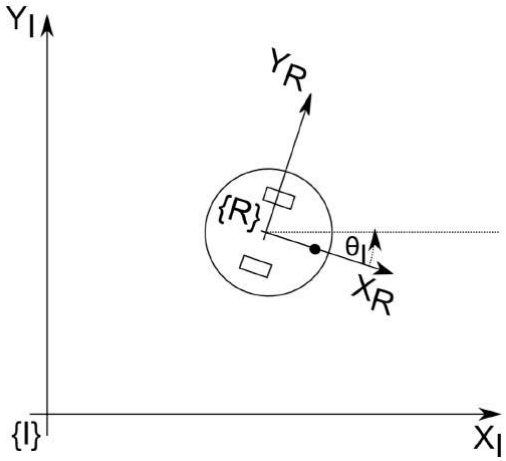


Figure **3.2.3**: Mobile robot with local coordinate system {R} and world frame {I}. The arrows indicate the positive direction of position and orientation vectors.

Notice that the positioning of the coordinate frames and their orientation are arbitrary. Here, we choose to place the coordinate system in the center of the robot's axle and align ${}^{R}x$ with its default driving direction.

In order to calculate the robot's position in the inertial frame, we need to first find out, how speed in the robot coordinate frame maps to speed in the inertial frame. This can be done again by employing trigonometry. There is only one complication: a movement into the robot's x-axis might lead to movement along both the x-axis and the y-axis of the world coordinate frame. By looking at the figure above, we can derive the following components to x_I . First,

$$ x_{I,x} = cos(\theta) x_R \tag{3.2.8} $$

There is also a component of motion coming from $\dot{y}_R$ (ignoring the kinematic constraints for now, see below). For negative θ, as in Figure 3.2.3, a move along y_R would let the robot move into positive X_I direction. The projection from $\dot{y}R$ is therefore given by

$$ x_{I,y} = -sin(\theta) y_R \tag{3.2.9} $$

We can now write

$$ x_{I,y} = cos(\theta) x_R - sin(\theta) y_R \tag{3.2.10} $$

Similar reasoning leads to

$$ y_I = sin(\theta) x_R + cos(\theta) y_R \tag{3.2.11} $$

and

$$ \theta_I = \theta_R \tag{3.2.12} $$

which is the case because both robot's and world coordinate system share the same z-axis in this example. We can now conveniently write

$$ \xi_I = {}^{I}_{R}\mathrm{T}(\theta) \xi_R \tag{3.2.13} $$

with

$$ {}^{I}_{R}\mathrm{T}(\theta) = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.2.14} $$

We are now left with the problem of how to calculate the speed $\dot{\xi}_R$ in robot coordinates. For this, we make use of the *kinematic constraints* of the robotic wheels. For a standard wheel, the kinematic constraints are that every rotation of the wheel leads to strictly forward or backward motion and does not allow side-way motion or sliding. We can therefore calculate the forward speed of a wheel x using its rotational speed φ (assuming the encoder value/angle is expressed as φ) and radius r by

$$ x = \phi r \tag{3.2.15} $$

This becomes apparent when considering that the circumference of a wheel with radius r is 2πr. The distance a wheel rolls when turned by the angle φ (in radians) is therefore x = φr, see also Figure 3.2.4, right. Taking the derivative of this expression on both sides leads to the above expression.

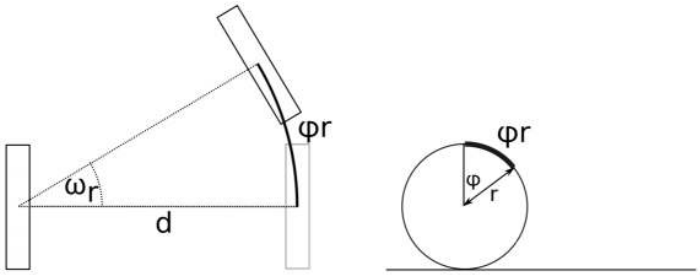


Figure **3.2.4**: Left: Differential wheel robot pivoting around its left wheel. Right: A wheel with radius r moves by φr when rotated by φ degrees.

How each of the two wheels in our example contributes to the speed of the robot's center—where its coordinate system is anchored—requires the following trick: we calculate the contribution of each individual wheel while assuming all other wheels remaining un-actuated. In this example, the distance traveled by the center point is exactly half of that traveled by each individual wheel, assuming the non-actuated wheel rotating around its ground contact point (Figure 3.2.4). We can therefore write

$$ x_R = \frac{r\phi_l}{2} + \frac{r\phi_r}{2} \tag{3.2.16} $$

given the speeds φ_l and φ_r of the left and the right wheel, respectively.

Note

Think about how the robot's speed along its y-axis is affected by the wheel-speed given the coordinate system in the drawing above. Think about the kinematic constraints that the standard wheels impose.

Hard to believe at first, but the speed of the robot along its y-axis is always zero. This is because the constraints of the standard wheel tell us that the robot can never slide. We are now left with calculating the rotation of the robot around its zaxis. That there is such a thing can be immediately seen when imaging the robot's wheels spinning in opposite directions. We will again consider each wheel independently. Assuming the left wheel to be non-actuated, spinning the right wheel forwards will lead to counter-clockwise rotation. Given an axle diameter (distance between the robot's wheels) d, we can now write

$$ \omega_r d = \phi_r r \tag{3.2.17} $$

with ω_r the angle of rotation around the left wheel (Figure 3.2.4, right). Taking the derivative on both sides yields speeds and we can write

$$ \omega_r = \frac{\phi_r r}{d} \tag{3.2.18} $$

Adding the rotation speeds up (with the one around the right wheel being negative based on the right-hand grip rule), leads to

$$ \omega_r = \frac{\phi_r r}{d} - \frac{\phi_l r}{d} \tag{3.2.19} $$

Putting it all together, we can write

$$\begin{pmatrix} x_I \\ y_I \\ \theta \end{pmatrix} = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{r\phi_l}{2} + \frac{r\phi_r}{2} \\ 0 \\ \frac{\phi_r r}{d} - \frac{\phi_l r}{d} \end{pmatrix} \tag{3.2.20}$$

From Forward Kinematics to Odometry

Equation 3.2.20 only provides us with the relationship between the robot's wheel-speed and its speed in the inertial frame. Calculating its actual pose in the inertial frame is known as odometry. Technically, it requires integrating (3.2.20) from 0 to the current time T. As this is not possible, but for very special cases, one can approximate the robot's pose by summing up speeds over discrete time intervals, or more precisely

$$\begin{pmatrix} x_I(T) \\ y_I(T) \\ \theta(T) \end{pmatrix} = \int_0^T \begin{pmatrix} x_I(t) \\ y_I(t) \\ \theta(t) \end{pmatrix} dt \approx \sum_{k=0}^{k=T} \begin{pmatrix} \Delta x_I(k) \\ \Delta y_I(k) \\ \Delta\theta(k) \end{pmatrix} \Delta t \tag{3.2.21}$$

which can be calculated incrementally as

$$x_I(k+1) = x_I(k) + \Delta x(k) \tag{3.2.22}$$

using $\Delta x(k) \approx x_I(t)$ and similar expressions for y_I and θ. Note that (3.2.22) is just an approximation. The larger Δt becomes, the more inaccurate this approximation becomes as the robot's speed might change during the interval.

Note

Don't let the notion of an integral worry you! As robots' computers are fundamentally discrete, integrals usually turn into sums, which are nothing than for-loops.

3.2.3. Forward kinematics of Car-like steering

Differential wheel drives are very popular in mobile robotics as they are very easy to build, maintain, and control. Although not holonomic, a differential drive can approximate the function of a fully holonomic robot by first driving on the spot to achieve the desired heading and then driving straight. Drawbacks of the differential drive are its reliance on a caster wheel, which performs poorly at high speeds, and difficulties in driving straight lines as this requires both motors to drive at the exact same speed.

These drawbacks are mitigated by car-like mechanisms, which are driven by a single motor and can steer their front wheels. This mechanism is known as "Ackermann steering". Ackermann steering should not be confused with "turntable" steering where the front wheels are fixed on an axis with central pivot point. Instead, each wheel has its own pivot point and the system is constrained in such a way that all wheels of the car drive on circles with a common center point, avoiding skid. As the Ackermann mechanism lets all wheels drive on circles with a common center point, its kinematics can be approximated by those of a tricycle with rear-wheel drive, or even simpler by a bicycle. This is shown in Figure 3.2.5.

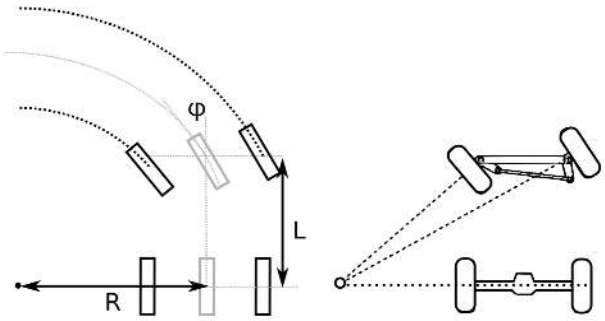


Figure **3.2.5**: Left: Kinematics of car-like steering and the equivalent bicycle model. Right: Mechanism of an Ackermann vehicle.

Let the car have the shape of a box with length L between rear and front axis. Let the center point of the common circle described by all wheels be distance R from the car's longitudinal center line. Then, the steering angle φ is given by

$$\tan\phi = \frac{L}{R} \tag{3.2.23}$$

The angles of the left and the right wheel, φ_l and φ_r can be calculated using the fact that all wheels of the car rotate around circles with a common center point. With the distance between the two front wheels l, we can write

$$\frac{L}{R - l/2} = \tan(\pi/2 - \phi_r) \tag{3.2.24}$$

$$\frac{L}{R + l/2} = \tan(\pi/2 - \phi_l) \tag{3.2.25}$$

This is important, as it allows us to calculate the resulting wheel angles resulting from a specific angle φ and test whether they are within the constraints of the actual vehicle. Assuming the wheelspeed to be ω and the wheel radius r, we can calculate the speeds in the robot's coordinate frame to

$$x_r = \omega_r \tag{3.2.26}$$

$$y_r = 0 \tag{3.2.27}$$

$$\theta_r = \frac{\omega r \tan\phi}{L} \tag{3.2.28}$$

using (3.2.23) to calculate the circle radius R.

3.3: Forward Kinematics using the Denavit-Hartenberg scheme

So far, we have considered the forward kinematics of wheeled mechanisms and simple arms and derived relationships between actuator parameters and velocities using basic trigonometry. In the specific case of multi-link arms, we can also think about the forward kinematics as a chain of homogenous transformations with respect to a coordinate system mounted at the base of a manipulator or a fixed position in the room. Deriving these transformations can be confusing and can be facilitated by following a "recipe" such as conceived by Denavit and Hartenberg. The so-called Denavit-Hartenberg (DH) scheme has evolved as quasi-standard and can easily be automatized, i.e., applied to a 3D model of a robotic arm, e.g.

A manipulating arm consists of links that are connected by joints. Joints can be either rotational or prismatic, i.e., change their length and thus providing additional degrees of freedom. Knowing the length of all rigid links, the position of the manipulators end-effector is fully described by its joint angles and joint offset (for prismatic joints).

In order to use the DH-convention, we first need to define a coordinate system at each joint. We chose the z-axis to be the axis of rotation for a hinge joint and the axis of translation for a prismatic joint. We can now find the common normal between the z-axes of two subsequent joints, i.e., a line that is orthogonal to each z-axis and intersects both. With the direction of the x-axis at the base arbitrary, subsequent x-axis are chosen such that they lie on the common normal shared between two joints. Whereas the direction of the z-axis is given by the positive direction of rotation (right-hand rule), the x-axis points away from the previous joint. This allows defining the y-axis using the right-hand rule. Note that these rules, in particular the requirement that x-axes lie along the commom normal, might result in coordinate systems with their origins outside the joint.

The transformation between two joints is then fully described by the following four parameters:

1. The length r of the common normal between the z-axes of two joints i and i − 1 (link length).
2. The angle α between the z-axes of the two joints with respect to the common normal (link twist), i.e., the angle between the old and the new z-axis, measured about the common normal.
3. The distance d between the joint axes (link offset), i.e., the offset along the previous z-axis to the common normal.
4. The rotation θ around the common axis along which the link offset is measured (joint angle), i.e., the angle from the old x-axis to the new x-axis, about the previous z-axis.

Two of the D-H parameters describe the link between the joints, and the other two describe the link's connection to a neighboring link. Depending on the link/joint type, these numbers are fixed or can be controlled. For example, in a revolute joint θ is the varying joint angle, while all other quantities are fixed. Similarly, for a prismatic joint d is the joint variable. An example of two revolute joints is shown in Figure 3.3.1.

The coordinate transform from one link (i−1) to another (i) can now be constructed using the following matrix:

$$ll^{n}_{n-1}T = \left(\begin{array}{ccc|c} \cos\theta_n & -\sin\theta_n\cos\alpha_n & \sin\theta_n\sin\alpha_n & r_n\cos\theta_n \\ \sin\theta_n & \cos\theta_n\cos\alpha_n & -\cos\theta_n\sin\alpha_n & r_n\sin\theta_n \\ 0 & \sin\alpha_n & \cos\alpha_n & d_n \\ \hline 0 & 0 & 0 & 1 \end{array}\right)$$

$$= \left(\begin{array}{ccc|c} & R & & t \\ \hline 0 & 0 & 0 & 1 \end{array}\right)$$

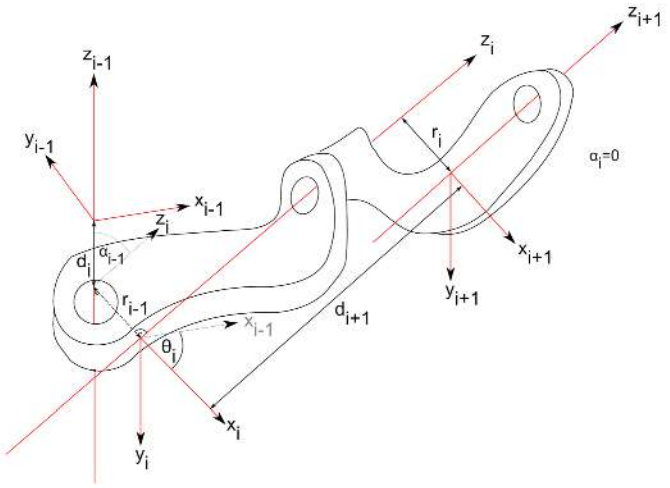


Figure **3.3.1**: Example of selected Denavit-Hartenberg parameters for three revolute joints. The z-axes of joint i and i + 1 are parallel.

with the rotation matrix R and the translation vector t. This matrix can be constructed by a series of rotations and translations, one for each DH parameter:

$$ {}^{n}_{n-1}\mathbf{T} = T'_z(d_n)R'_z(\theta_n)T_x(r_n)R_x(\alpha_n) \tag{3.3.1} $$

with

$$ T'_z(d_n) = \left(\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_n \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \quad R'_z(\theta_n) = \left(\begin{array}{ccc|c} \cos\theta_n & -\sin\theta_n & 0 & 0 \\ \sin\theta_n & \cos\theta_n & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) $$

and

$$ T_x(r_n) = \left(\begin{array}{ccc|c} 1 & 0 & 0 & r_n \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \quad R_x(\alpha_n) = \left(\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_n & -\sin\alpha_n & 0 \\ 0 & \sin\alpha_n & \cos\alpha_n & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) $$

These are a translation of d_n along the previous z-axis ($T'_z(d_n)$), a rotation of θ_n about the previous z-axis ($R'_z(\theta_n)$), a translation of r_n along the new x-axis ($T_x(r_n)$)and a rotation of αn around the new x-axis ($R_x(\alpha_n)$).

Like for any homogeneous transfrom, the inverse ${}_{n-1}{}^{n}T^{-1}n$ is given by

$$ {}^{n-1}_{n}T = \left(\begin{array}{ccc|c} & R^{-1} & & -R^{-1}T \\ \hline 0 & 0 & 0 & 1 \end{array}\right) $$

with the inverse of R simply being its transpose.

3.4: Inverse Kinematics of Selected Mechanisms

The forward kinematics of a system are given by a transformation matrix from the base of a manipulator (or a corner of the room) to the end-effector of a manipulator (or a mobile robot). As such, they are an exact description of the pose of the robot. In order to find the joint angles that lead to the desired pose, we will need to solve these equations for joint angles as a function of the desired pose. For a mobile robot, we can do this only for velocities in the local coordinate system, and need more sophisticated methods to calculate appropriate trajectories for the robot.

3.4.1. Solvability

As the resulting equations are heavily non-linear, it makes sense to briefly think about whether we can solve them at all for specific parameters before trying. Here, the workspace of a robot becomes important. The workspace is the sub-space that can be reached by the robot in any orientation. Clearly, there will be no solutions for the inverse kinematic problem outside of the workspace of the robot.

A second question to ask is how many solutions we actually expect and what it means to have multiple solutions geometrically. Multiple solutions to achieve a desired pose correspond to multiple ways in which a robot can reach a target. For example a three-link arm that wants to reach a point that can be reached without fully extending all links (leading to a single solution), can do this by folding its links in a concave and a convex fashion. How many solutions there are for a given mechanism and pose quickly becomes non-intuitive. For example a 6-DOF arm can reach certain points with up to 16 different conformations.

3.4.2. Inverse Kinematics of a Simple Manipulator Arm

We will now look at the kinematics of a 2-link arm that we introduced earlier. We need to solve the equations determining the robot's forward kinematics by solving for α and β. This is tricky, however, as we have to deal with complicated trigonometric expressions.

To get an intuition, assume there to be only one link, l1. Solving (3.2.1) for α yields actually two solutions [$\cos^{-1}$ x1/l1 , – $\cos^{-1}$ x1/l1] , as cosine is symmetric for positive and negative values. Indeed, for any possible position on the *x−axis* ranging from *−l1 to l1*, there exist two solutions. One with the arm above the table, one with the arm within the table. (At the extremes of the workspace, both solutions are the same.)

Solving for both degrees of freedom actually yields eight solutions, of which only two are feasible:

$$\alpha \rightarrow \cos^{-1}\left(\frac{x^2y+y^3-\sqrt{4x^4-x^6+4x^2y^2-2x^4y^2-x^2y^4}}{2(x^2+y^2)}\right) \tag{3.4.1}$$

and

$$\beta \rightarrow -\cos^{-1}(1/2(-2)x^2+y^2)) \tag{3.4.2}$$

What will drastically simplify this problem, is to not only specificy the desired position, but also the orientation of the end-effector. In this case, a desired pose can be specified by

$$\begin{pmatrix} cos\phi & -sin\phi & 0 & x \\ sin\phi & cos\phi & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.4.3}$$

A solution can now be found by simply equating the individual entries of the transformation (3.2.7) with those of the desired pose. Specifically, we can observe

$$cos\phi = cos(\alpha+\beta) \tag{3.4.4}$$

$$x = \cos_{\alpha\beta}\, l_2 + \cos_{\alpha}\, l_1$$

$$y = \sin_{\alpha\beta}\, l_2 + \sin_{\alpha}\, l_1$$

These can be reduced to

\phi =\alpha +\beta

$$\cos\alpha = \frac{\cos_{\alpha\beta} l_2 - x}{l_1} = \frac{\cos_{\phi} l_2 - x}{l_1}$$

$$\sin\alpha = \frac{\sin\alpha\beta l_2 - y}{l_1} = \frac{\sin_{\phi} l_2 - y}{l_1}$$

Providing the orientation of the robot in addition to the desired position therefore allows solving for α and β just as a function of x, y and φ.

As such solutions quickly become unhandy with more dimensions, however, you can calculate a numerical solution using an approach that we will later see is very similar to path planning in mobile robotics. One way to do this is to plot the distance of the end-effector from the desired solution in configuration space. To do this, you need to solve the forward kinematics for every point in configuration space and use the Euclidian distance to the desired target as height. In our example this would be

$$f_{x,y}(\alpha,\beta) = \sqrt{(\sin(\alpha+\beta)+\sin(\alpha)-y)^2 + (\cos(\alpha+\beta)+\cos(a)-x^2)} \tag{3.4.5}$$

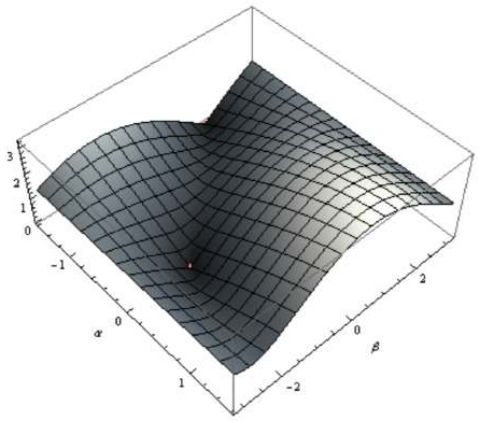


Figure **3.4.1**: Distance to (x = 1, y = 1) over the configuration space of a two-arm manipulator. Minima corresponds to exact inverse kinematic solutions.

This is plotted for α = [−π/2, π/2] and β = [−π, π] and x = 1, y = 1 in Figure 3.4.1. This function has a minima, in this case zero, for values of α and β that bring the manipulator to (1, 1). These values are (α → 0, b → −π 2) and (α → −π 2 , b → π 2).

You can now think about inverse kinematics as a path finding problem from anywhere in the configuration space to the nearest minima. A formal approach to doing this will be discussed in Section 3.5. How to find shortest paths in space, that is finding the shortest route for a robot to get from A to B will be a subject of chapter 4.

3.4.3. Inverse Kinematics of Mobile Robots

As there is no unique relationship between the amount of rotation of a robot's individual wheels and its position in space, but for simple holonomic platforms such as a robot on a track, we will treat the inverse kinematics problem at first only for the velocities of the local robot coordinate frame.

Let's first establish how to calculate the necessary speed of the robot's center given a desired speed ξI in world coordinates. We can transform the expression ξI = T(θ)ξR by multiplying both sides with the inverse of T(θ):

$$T^{-1}(\theta)\xi_I = T^{-1}(\theta)T(\theta)\xi_R \tag{3.4.6}$$

which leads to ξR = T^{-1} (θ)ξ_I. Here

$$T^{-1} = \begin{pmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.4.7}$$

which can be determined by actually performing the matrix inversion or by deriving the trigonometric relationships from the drawing. Similarly, we can now solve

$$\begin{pmatrix} x_R \\ y_R \\ \theta \end{pmatrix} = \begin{pmatrix} \frac{r\phi_l}{2} + \frac{r\phi_r}{2} \\ 0 \\ \frac{\phi_r r}{d} - \frac{\phi_l r}{d} \end{pmatrix} \tag{3.4.8}$$

for φ_l , φ_r

$$\phi_l = (2x_R - \theta d)/2r \tag{3.4.9}$$

$$\phi_r = (2x_R + \theta d)/2r \tag{3.4.10}$$

allowing us to calculate the robot's wheelspeed as a function of a desired x_R and θ, which can be calculated using (3.4.6).

Note that this approach does not allow us to deal with $y_R \neq 0$, which might result from a desired speed in the inertial frame. Non-zero values for translation in y-direction are simply ignored by the inverse kinematic solution, and driving toward a specific point either requires feedback control (Section 3.5.2) or path planning (Chapter 4).

3.5: Inverse Kinematics using Feedback-Control

Solving the inverse kinematic problem for non-holonomic mobile robots require us to find a sequence of actuation commands. One way of doing this is to employ feedback control. In a nutshell, feedback control uses the error between actual and desired position to calculate a trajectory piece that drives the robot a little closer to its desired pose. The process is then repeated until the error is marginally small. This approach can not only be used for mobile robots, but also for manipulator arms with kinematics that are too complicated to solve analytically.

3.5.1. Feedback control for mobile robots

Assume the robot's position given by x_r, y_r and θ_r and the desired pose as x_g, y_g and θ_g with the subscript g indicating "goal". We can now calculate the error in the desired pose by

$$\rho = \sqrt{(x_r - x_g)^2 + (y_r - y_g)^2} \tag{3.5.1}$$

$$\alpha = \tan^{-1} \frac{y_g - y_r}{x_g - x_r} - \theta_r$$

$$\eta = \theta_g - \theta_r$$

which is illustrated in Figure 3.5.1. These errors can be turned directly into robot's speeds, for example using a simple proportional controller with gains p1, p2 and p3:

$$x = p_1 \rho \tag{3.5.2}$$

$$\theta = p_2 \alpha + p_3 \eta \tag{3.5.3}$$

which will let the robot drive in a curve until it reaches the desired pose.

3.5.2. Inverse Jacobian Technique

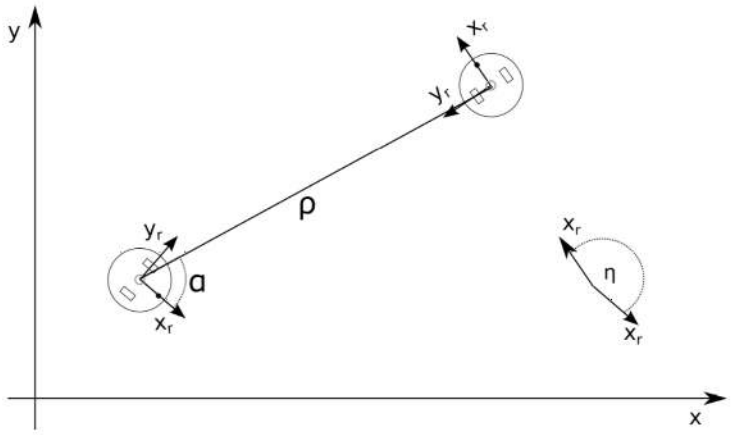


Figure **3.5.1**: Difference in desired and actual pose as a function of distance ρ, bearing α and heading η.

The two-link arm (Figure 3.2.1) involved only two free parameters, but was already pretty complex to solve analytically if the end-effector pose was not specified. One can imagine that things become very hard with more degrees of freedom or more complex geometries. (Mechanisms in which some axes intersect are simpler to solve than others, for example.) Fortunately, there are simple numerical techniques that work reasonably well. One of them known is as Inverse Jacobian technique:

As we can easily calculate the resulting pose for every possible joint angle combination using the forward kinematic equations, we can calculate the error between desired and actual pose. This error actually provides us with a direction that the end-effector needs to move. As we only need to move tiny bits at a time and can then re-calculate the error, this is an attractive method to generate a trajectory that moves the arm to where we want it go and thereby solving the inverse kinematics problem.

In order to do this, we need an expression that relates the desired speed of the robot's end-effector, i.e., the direction in which we want to move, to the speed at which we need to change our joints. Let the translational speed of a robot be given by

$$v = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{3.5.4}$$

As the robot can potentially not only translate, but also rotate, we also need to specify its angular velocity. Let these velocities be given as a vector

$$\omega = \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \tag{3.5.5}$$

This notation is also called a *velocity screw*. We can now write translational and rotational velocities in a 6x1 vector as $(v\ \omega)^T$. Let the joint angles/positions be $j = (j_1, \ldots, j_n)$.

Given a relationship between end-effector velocities j and joint velocities J, we can write

$$(v \quad \omega)^T = J(j_1, ..., j_n)^T \tag{3.5.6}$$

with n the number of joints. J is also known as the Jacobian matrix and contains all partial derivatives that relate every joint angles to every velocities. In practice, J looks like

$$\begin{pmatrix} v \\ \omega \end{pmatrix} = \begin{pmatrix} \frac{\partial x}{\partial j_1} & \cdots & \frac{\partial x}{\partial j_n} \\ \vdots & & \vdots \\ \frac{\partial \omega_z}{\partial j_1} & \cdots & \frac{\partial \omega_z}{\partial j_n} \end{pmatrix} (j_1, ..., j_n)^T \tag{3.5.7}$$

This notation is important as it tells us how small changes in the joint space will affect the end-effector's position in cartesian space. Better yet, the forward kinematics of a mechanism can always be calculated, as well as their analytical derivatives, allowing us to calculate numerical values for the entries of matrix J for every possible joint angle/position.

It would now be desirable to just invert J in order to calculate the necessary joint speeds for every desired end-effector speeds. Unfortunately, J is only invertible if there are exactly 6 independent joints, so that J is quadratic and has full rank. If this is not the case, we can use the pseudo-inverse instead:

$$J^+ = \frac{J^T}{JJ^T} = J^T(JJ^T)^{-1} \tag{3.5.8}$$

As you can see, J^T cancels from the equation leaving 1/J, while being applicable to non-quadratic matrices.

This solution might or might not be numerically stable, depending on the current joint values. If the inverse of J is mathematically not feasible, we speak of a singularity of the mechanism. This happens for example when two joint axes line up, therefore effectively removing a degree of freedom from the mechanism, or at the boundary of the workspace. Mathematically speaking the rank of the Jacobian is smaller than six.

We can now write a simple feedback controller that drives our error e as the difference between desired and actual position to zero:

$$\Delta j = -J^+ e \tag{3.5.9}$$

That is, we move each joint a tiny bit into the direction that minimizes e. It can be easily seen that the joint speeds are only zero if e has become zero. A problem arises, however, when the end-effector has to go through a singularity to get to its goal. Then, the solution to J^+ "explodes" and joint speeds go to infinity. In order to work around this, we can introduce damping to the controller.

This can be achieved by not only minimizing the error, but also the joint velocities. Thus, the minimization problem becomes

$$\| J\Delta j - e \| + \lambda^2 \| \Delta j \|^2 \tag{3.5.10}$$

where λ is some constant. One can show that the resulting controller that achieves this has the form

$$\Delta j = (J^T J + \lambda^2 I)^{-1} J^+ e \tag{3.5.11}$$

This is known as the *Damped Least-Squares* method. Problems with this approach are local minima and singularities of the mechanism, which might render this solution infeasible.

3.6: Exercises

Take-home lessons

- Forward kinematics are equivalent to finding a coordinate transform from a world coordinate system into a coordinate system on the robot. Such a transform is a combination of a (3x1) translation vector and a (3x3) rotation matrix that consists of the unit vectors of the robot coordinate system. Both translation and rotation can be combined into a 4x4 homogeneous transform matrix.
- Forward and Inverse Kinematics of a mobile robot are performed with respect to the speed of the robot and not its position.
- For calculating the effect of each wheel on the speed of the robot, you need to consider the contribution of each wheel independently.
- Calculating the inverse kinematics analytically becomes quickly infeasible. You can then plan in configuration space of the robot using path-planning techniques.
- The inverse kinematics of a robot involves solving the equations for the forward kinematics for the joint angles. This process is often cumbersome if not impossible for complicated mechanisms.
- A simple numerical solution is provided by taking all partial derivatives of the forward kinematics in order to get an easily invertible expression that relates joint speeds to end-effector speeds. The inverse kinematics problem can then be formulated as feedback control problem, which will move the end-effector towards its desired pose using small steps. Problems with this approach are local minima and singularities of the mechanism, which might render this solution infeasible.

Exercises

Coordinate systems

1. a) Write out the entries of a rotation matrix ${}_B^A R$ assuming basis vectors X_A, Y_A, Z_A, and X_B, Y_B, Z_B. b) Write out the entries of rotation matrix ${}_B^A R$.
2. Assume two coordinate systems that are co-located in the same origin, but rotated around the z-axis by the angle α. Derive the rotation matrix from one coordinate system into the other and verify that each entry of this matrix is indeed the scalar product of each basis vector of one coordinate system with every other basis vector in the second coordinate system.
3. Consider two coordinate systems {B} and {C}, whose orientation is given by the rotation matrix ${}_B^C R$ and have distance ${}^B P$. Provide the homogenous transform ${}_B^C T$ and its inverse ${}_C^B T$.
4. Consider the frame {B} that is defined with respect to frame {A} as $\{B\} = \{ {}_B^A R, {}^A P\}$. Provide a homogeneous transform from {A} to {B}.

Forward and inverse kinematics

1. Consider a differential wheel robot with a broken motor, i.e., one of the wheels cannot be actuated anymore. Derive the forward kinematics of this platform. Assume the right motor is broken.
2. Consider a tri-cycle with two independent standard wheels in the rear and the stearable, driven front-wheel. Choose a suitable coordinate system and use φ as the steering wheel angle and wheel-speed ω. Provide forward and inverse kinematics.

CHAPTER OVERVIEW

4: Path Planning

Path-planning is an important primitive for autonomous mobile robots that lets robots find the shortest—or otherwise optimal—path between two points. Optimal paths could be paths that minimize the amount of turning, the amount of braking or whatever a specific application requires. Algorithms to find a shortest path are important not only in robotics, but also in network routing, video games and understanding protein folding.

Path-planning requires a map of the environment and the robot to be aware of its location with respect to the map. We will assume for now that the robot is able to localize itself, is equipped with a map, and capable of avoiding temporary obstacles on its way. How to create a map, how to localize a robot, and how to deal with uncertain position information will be major foci of the reminder of this book. The goals of this chapter are to

- introduce suitable map representations,
- explain basic path-planning algorithms ranging from Dijkstra, to A*, D* and RRT,
- introduce variations of the path-planning problem, such as coverage path planning.

4.1: Map Representations

In order to plan a path, we somehow need to represent the environment in the computer. We differentiate between two complementary approaches: discrete and continuous approximations. In a discrete approximation, a map is sub-divided into chunks of equal (e.g., a grid or hexagonal map) or differing sizes (e.g., rooms in a building). The latter maps are also known as *topological maps*. Discrete maps lend themselves well to a graph representation. Here, every chunk of the map corresponds to a vertex (also known as “node”), which are connected by edges, if a robot can navigate from one vertex to the other. For example a road-map is a topological map, with intersections as vertices and roads as edges, labeled with their length (Figure 4.2.1). Computationally, a graph might be stored as an adjacency or incidence list/matrix. A continuous approximation requires the definition of inner (obstacles) and outer boundaries, typically in the form of a polygon, whereas paths can be encoded as sequences of points defined by real numbers. Despite the memory advantages of a continuous representation, discrete maps are the dominant representation in robotics.

For mapping obstacles, the most common map is the occupancy grid map. In a grid map, the environment is discretized into squares of arbitrary resolution, e.g. 1cm x 1cm, on which obstacles are marked. In a probabilistic occupancy grid, grid cells can also be marked with the probability that they contain an obstacle. This is particularly important when the position of the robot that senses an obstacle is uncertain. Disadvantages of grid maps are their large memory requirements as well as computational time to traverse data structures with large numbers of vertices. A solution to this is storing the grid map as *k-d tree*. A k-d tree recursively breaks the environment into k pieces. For k = 4, an area is broken into four pieces. Each of these pieces is again broken into four pieces and so on, until the desired resolution is reached. These pieces can easily be stored in a graph with each vertex having four children, which are the four pieces the vertex is broken into, or is a leaf of the tree. What makes this data structure attractive is that not all vertices need to be broken down to the smallest possible resolution. Instead only areas, which contain obstacles need to be further broken down. A grid map containing obstacles and the corresponding k-d tree, here a quadtree, are shown in Figure 4.1.1. There is no silver bullet, and each application might require a different solution that could be a combination of different map types.

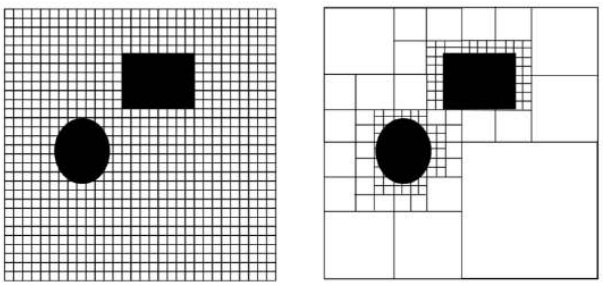


Figure **4.1.1**: A grid map and its corresponding quadtree (k-d tree).

There exist also every possible combination of discrete and continuous representation. For example, roadmaps for GPS systems are stored as topological maps that store the GPS coordinates of every vertex, but might also contain overlays of aerial and street photography or even 3D point clouds stored in a 8-d tree, also known as a *Octree*. These different maps are then used at different stages of the path planning stage.

4.2: Path-Planning Algorithms

The problem to find a “shortest” path from one vertex to another through a connected graph is of interest in multiple domains, most prominently in the internet, where it is used to find an optimal route for a data packet. The term “shortest” refers here to the minimum cumulative edge cost, which could be physical distance (in a robotic application), delay (in a networking application) or any other metric that is important for a specific application. An example graph with arbitrary edgelengths is shown in Figure 4.2.1.

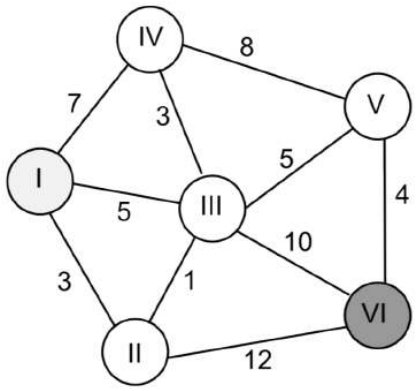


Figure **4.2.1**: A generic path planning problem from vertex I to vertex VI. The shortest path is I-II-III-V-VI with length 13.

4.2.1. Robot embodiment

In order to deal with the physical embodiment of the robot, which complicates the path-planning process, the robot is reduced to a point-mass and all the obstacles in the environment are grown by half of the longest extension of the robot from its center. This representation is known as configuration space as it reduces the representation of the robot to its x and y coordinates in the plane. An example is shown in Figure 4.2.2. The configuration space can now either be used as a basis for a grid map or a continuous representation.

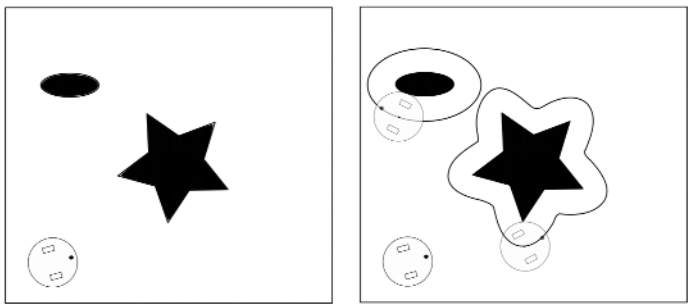


Figure **4.2.2**: A map with obstacles and its representation in configuration space, which can be obtained by growing each obstacle by the robot’s extension.

4.2.2. Dijkstra’s algorithm

One of the earliest and simplest algorithms is Dijkstra’s algorithm (Dijkstra 1959). Starting from the initial vertex where the path should start, the algorithm marks all direct neighbors of the initial vertex with the cost to get there. It then proceeds from the vertex with the lowest cost to all of its adjacent vertices and marks them with the cost to get to them via itself if this cost is lower. Once all neighbors of a vertex have been checked, the algorithm proceeds to the vertex with the next lowest cost. Once the algorithm reaches the goal vertex, it terminates and the robot can follow the edges pointing towards the lowest edge cost.

In Figure 4.2.1, Dijkstra would first mark nodes II, III and IV with cost 3, 5 and 7 respectively. It would then continue to explore all edges of node II, which so far has the lowest cost. This would lead to the discovery that node III can actually be reached in $3+1 < 5$ steps, and node III would be relabeled with cost 4. In order to completely evaluate node II, Dijkstra needs to evaluate the remaining edge before moving on and label node VI with $3 + 12 = 15$.

The node with the lowest cost is now node III (4). We can now relabel node VI with 14, which is smaller than 15, and label node V with 4 + 5 = 9, whereas node IV remains at 4 + 3 = 7. Although we have already found two paths to the goal, one of which better than the other, we cannot stop as there still exist nodes with unexplored edges and overall cost lower than 14. Indeed, continuing to explore from node V leads to a shortest path I-II-III-V-VI of cost 13, with no remaining nodes to explore.

As Dijkstra would not stop until there is no node with lower cost than the current cost to the goal, we can be sure that a shortest path will be found if it exists. We can say that the algorithm is complete.

As Dijkstra will always explore nodes with the least overall cost first, the environment is explored comparably to a wave front originating from the start vertex, eventually arriving at the goal. This is of course highly inefficient in particular if Dijkstra is exploring nodes away from the goal. This can be visualized by adding a couple of nodes to the left of node I in Figure 4.2.1. Dijkstra will explore all of these nodes until their cost exceeds the lowest found for the goal. This can also be seen when observing Dijkstra's algorithm on a grid, as shown in Figure 4.2.3.

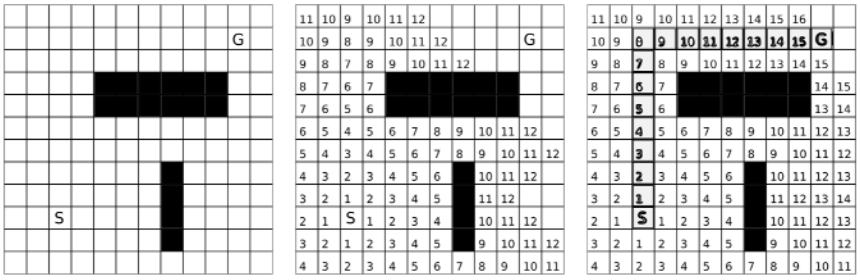


Figure 4.2.3: *Dijkstra's algorithm finding a shortest path from 'S' to 'G' assuming the robot can only travel laterally (not diagonally) with cost one per grid cell. Note the few number of cells that remain unexplored once the shortest path (grey) is found, as Dijkstra would always consider a cell with the lowest path cost first.*

4.2.3. A*

Instead of exploring in all directions, knowledge of an approximate direction of the goal could help avoiding exploring nodes that are obviously wrong to a human observer. Such special knowledge that such an observer has can be encoded using a heuristic function, a fancier word for a "rule of thumb". For example, we could give priority to nodes that have a lower estimated distance to the goal than others. For this, we would mark every node not only with the actual distance that it took us to get there (as in Dijkstra's algorithm), but also with the estimated cost "as the crow flies", for example by calculating the Euclidean distance or the Manhattan distance between the vertex we are looking at and the goal vertex. This algorithm is known as A* (Hart, Nilsson & Raphael 1968). Depending on the environment, A* might accomplish search much faster than Dijkstra's algorithm, and performs the same in the worst case. This is illustrated in Figure 4.2.4 using the Manhattan distance metric, which does not allow for diagonal movements.

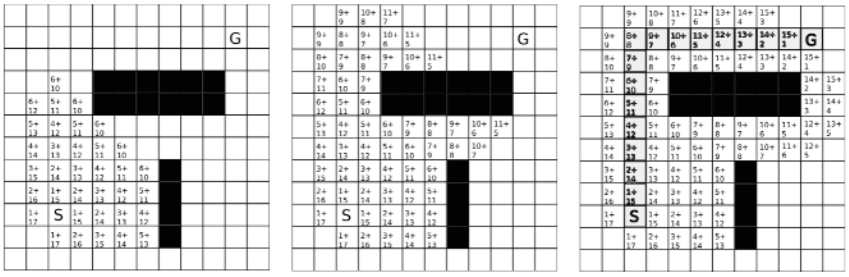


Figure 4.2.4: Finding a shortest path from 'S' to 'G' assuming the robot can only travel laterally (not diagonally) with cost one per grid cell using the A* algorithm. Much like Dijkstra, A* evaluates only the cell with the lowest cost, but takes an estimate of the remaining distance into account.

An extension of A* that addresses the problem of expensive re-planning when obstacles appear in the path of the robot, is known as D* (Stentz 1994). Unlike A*, D* starts from the goal vertex and has the ability to change the costs of parts of the path that include an obstacle. This allows D* to replan around an obstacle while maintaining most of the already calculated path.

A* and D* become computationally expensive when either the search space is large, e.g., due to a fine-grain resolution required for the task, or the dimensions of the search problem are high, e.g. when planning for an arm with multiple degrees of freedom. Solutions to these problems are provided by samplingbased path planning algorithms that are described further below.

4.3: Sampling-based Path Planning

The previous sections have introduced a series of complete algorithms for the path planning problem, i.e. they will find a solution eventually if it exists. Complete solutions are often unfeasible, however, when the possible state space is large. This is the case for robots with multiple degrees of freedom such as arms. In practice, most algorithms are only resolution complete, i.e., only complete if the resolution is fine-grained enough, as the state-space needs to be somewhat discretized for them to operate (e.g., into a grid) and some solutions might be missed as a function of the resolution of the discretization.

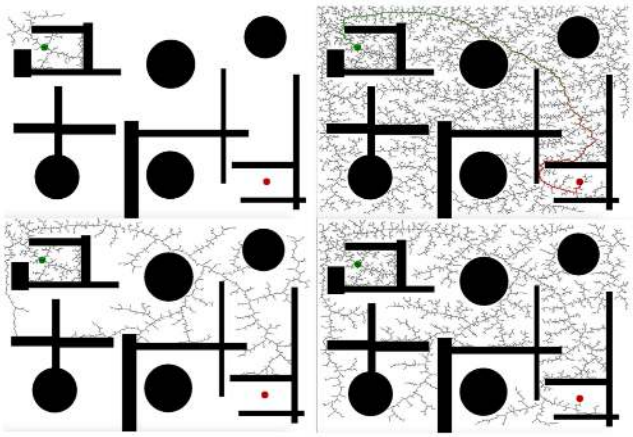


Figure **4.3.1**: Counterclockwise: Random exploration of a 2D search space by randomly sampling points and connecting them to the graph until a feasible path between start and goal is found.

Instead of evaluating all possible solutions or using a noncomplete Jacobian-based inverse kinematic solution, samplingbased planners create possible paths by randomly adding points to a tree until some solution is found or time expires. As the probability to find a path approaches one when time goes to infinity, sampling-based path planners are probabilistic complete. Prominent examples of sampling-based planners are Rapidlyexploring Random Trees (RRT)(LaValle 1998) and Probabilistic Roadmaps(PRM) (Kavraki, Svestka, Latombe & Overmars 1996). An example execution of RRT for an unknown goal, thereby reducing the path-planning problem to a search problem is shown in Figure 4.3.1.

This example illustrates well how a sampling-based planner can quickly explore a large portion of space and refines a solution as time goes on. Whereas RRT can be understood as growing a single tree from a robot's starting point until one of its branches hits a goal, probabilistic road-maps create a tree by randomly sampling points in the state-space, testing whether they are collision-free, connecting them with neighboring points using paths that reflect the kinematics of a robot, and then using classical graph shortest path algorithms to find shortest paths on the resulting structure. The advantage of this approach is clearly that such a probabilistic roadmap has to be created only once (assuming the environment is not changing) and can then be used for multiple queries. PRMs are therefore a multi-query path-planning algorithm. In contrast, RRT's are known as single-query path-planning algorithms.

In practice, the boundary between the different historic algorithms has become very diffuse, and single-query and multiquery variants of both RRT and PRM exist. It is important to note that there is no silver bullet algorithm/heuristic and even their parameter-sets are highly problem-specific. We will therefore limit our discussion on useful heuristics that are common to sampling-based planners.

4.3.1. Basic Algorithm

Let X be a d-dimensional state-space. This can either be the robot's state given in terms of translation and rotations (6 dimensions), a subset thereof, or the joint space with one dimension per possible joint angle. Let $G \subset X$ be a d-ball (ddimensional sphere) in the state-space that is considered to be the goal, and t the allowed time. A tree planner proceeds as follows:

```
Tree=Init(X,start);
WHILE ElapsedTime() < t AND NoGoalFound(G) DO
    newpoint = StateToExpandFrom(Tree);
    newsegment = CreatePathToTree(newpoint);
```

```
IF ChooseToAdd(newsegment) THEN
    Tree=Insert(Tree,newsegment);
ENDIF
ENDWHILE
return Tree
```

This process can be repeated on the resulting tree as long as time allows. This is known as an *AnyTime* algorithm. Given a suitable distance metric, the cost-to-goal can be stored at each node of the tree (much easier if growing the tree from the goal to start), which allows retrieving the shortest path easily. There are four key points in this algorithm:

1. Finding the next point to add to the tree (or discard) (StateToExpandFrom).
2. Finding out where and how to connect this point to the tree taking into account the robot kinematics (CreatePathToTree).
3. Testing whether this path is suitable, i.e., collision-free.
4. Finding the next point to add.

A prominent method is to pick a random point in the statespace and connect it to the closest existing point in the tree or to the goal. This requires searching all nodes in the tree and calculating their distance to the candidate point. Other approaches put preferences on nodes with fewer out-degrees (those which do not yet have very many connections) and choose a new point within its vicinity. Both approaches make it likely to quickly explore the entire state-space.

If there are constraints imposed on the robot's path, for example the robot needs to hold a cup and therefore is not supposed to rotate its wrist, this dimension can simply be taken out of the state-space.

Once a possible path is found, this space can be reduced to the ellipsoid that bounds the maximal path-length. This ellipsoid can be constructed by mounting a wire of the maximum path length between start and goal and pushing it outward with a pen. All the area that can be reached with the pen constrained by the wire can contain a point that can possibly lead to a shorter path. This approach is particularly effective when running multiple copies of the same planner in parallel and exchanging the shortest paths once they are found (Otte & Correll 2013).

4.3.2. Connecting Points to the Tree

A new point is classically connected to the closest point already in the tree or to the goal. This can be done by calculating the distance to all points already in the tree. This does not necessarily generate the shortest path, however. A recent improvement has been made by RRT*, which connects the point to the tree in a way that minimizes the overall path length. This can be done by considering all points in the tree within a d-ball (on a 2D map, d = 2, i.e. a circle) from of fixed radius from the point to add and finding the point that minimizes the overall path length to the start.

Adding a point to the tree is also a good time to take into account the specific kinematics of a robot, for example a car. Here, a local planner can be used to generate a suitable trajectory that takes into account the orientation of the vehicle at each point in the tree.

4.3.3. Collision Checking

Efficient algorithms for testing collisions deserve a dedicated section. While the problem is intuitive in configuration-space planning in 2D (the robot reduces to a point) and can be solved using a simple point-in-polygon test, the problem is more involved for manipulators that are subject to self-collision.

As collision checking takes up to 90% of the execution time in the path-planning problem, a successful method to increase computational speed is "lazy collision evaluation". Instead of checking every point for a possible collision, the algorithm first finds a suitable path. Only then, it checks every segment of this path for collisions. In case some segments are in collision, they are deleted and the algorithm goes on, but keeps the segments of the successful path that were collision-free.

4.4: Path Smoothing

As paths are randomly sampled, they will be most likely shaky and not optimal. For exampe, a grid-map will generate a series of sharp turns and a sampling-based approach will return zig-zag random paths. Results can be drastically improved by running an additional algorithm that smoothes the path. One way of doing this is to connect points of the path using splines, curves or even trajectory snippets that are known to be feasible for a specific platform. Alternatively, one can also use a model of the actual platform and use a feedback controller such as described in Section 3.5.1 for mobile robots and Section 3.5.2 for arms, sample a series of points in front of the robot, and generate a trajectory that the robot can actually drive. When combined with dynamics, this approach is known as *model-predictive control*. Care needs to be taken, however, that the resulting paths are indeed collision free.

4.5: Planning at different length-scales

In practice, no one map representation and planning algorithm might be sufficient. To plan a route for a car, for example, might involve a coarse search over the street network such as performed by your car's navigation system, but not involve planning which lane to actually choose. Planning lanes and how to navigate round-abouts and intersections will then involve another layer of discrete planning. How to actually move the robot within a lane and avoid local obstacles, might then be best done with a sampling-based planning algorithm. Finally, trajectories need to be turned into wheel speeds and turn angles, possibly using some form of feedback control. This hierarchy is depicted in Figure 4.7. Here, downward-pointing arrays indicate input that one planning layer provides to the one below. Upward-pointing arrows instead indicate exceptions that cannot be handled by the lower levels. For example, a feedback controller cannot handle obstacles well, requiring the samplingbased planning layer to come up with a new trajectory. Should the entire road be blocked, this planner would need to hand-off control the lane-based planner. A similar case can be made for manipulating robots, which also need to combine multiple different representations and controllers to plan and execute trajectories efficiently.

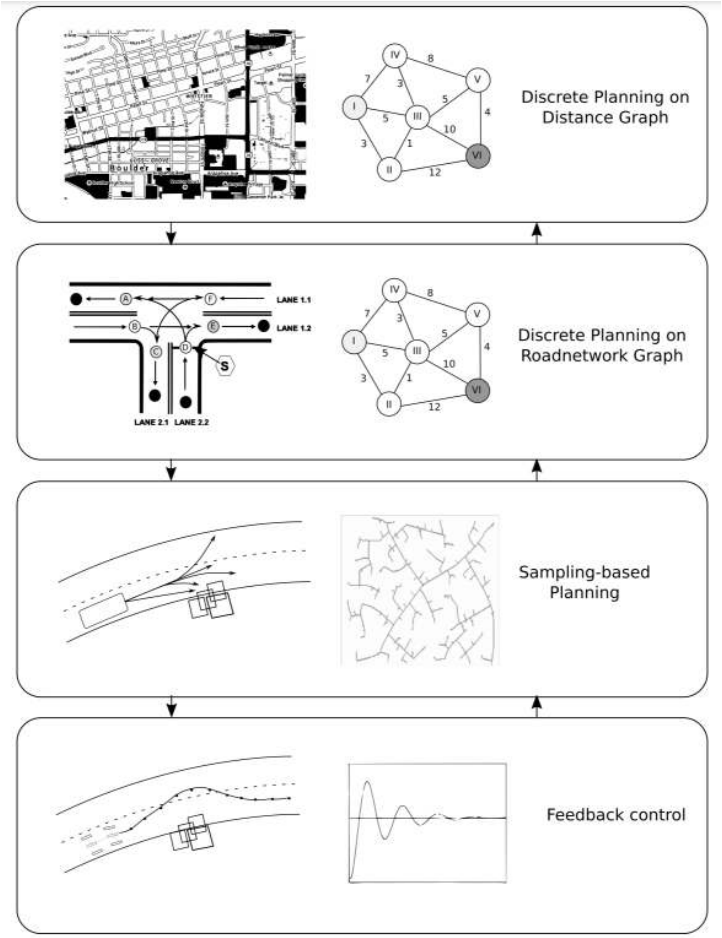


Figure **4.5.1**: Path-planning across different length-scales, requiring a variety of map representations and planning paradigms. Arrows indicate information passed between layers.

Note that this representation does not include a reasoning level that encodes traffic rules and common sense. While some of these might be encoded using cost-functions, such as maximizing distance from obstacles or insuring smooth riding, other more complex behaviors such as adapting driving in the presence of cyclists or properties of the ground need to be implemented in an additional vertical layer that has access to all planning layers.

4.6: Other Path-Planning Applications

Once the environment has been discretized into a graph, we can employ other algorithms from graph theory to plan desirable robot trajectories. For example, floor coverage can be achieved by performing a depth-first search (DFS) or a breadthfirst-search (BFS) on a graph where each vertex has the size of the coverage tool of the robot. “Coverage” is not only interesting for cleaning a floor: the same algorithms can be used to perform an exhaustive search of a configuration space, such as in the example shown in Figure 3.10, where we plotted the error of a manipulator arm in reaching a desired position over its configuration space. Finding a minimum in this plot using an exhaustive search solves the inverse kinematics problem. Similarly, the same algorithm can be used to systematically follow all links on a website till a desired depth (or actually retrieving the entire world-wide web).

Doing a DFS or a BFS might generate efficient coverage paths, but they are far from optimal as many vertices might be visited twice. A path that connects all vertices in a graph but passes every vertex only once is known as a Hamiltonian Path. A Hamiltonian path that returns to its starting vertex is known as a Hamiltonian Cycle. This problem is also known as the Traveling Salesman Problem (TSP), in which a route needs to be calculated that visits every city on his tour only once and is known to be NP Complete.

4.7: Exercises

Summary and Outlook

Path-planning is an ongoing research problem. Finding collision free paths for mechanisms with high degrees of freedom such as multiple arms operating in a common space, multi-robot systems, or systems involving dynamics (and therefore adding the derivatives of the state variables to the planning problem) might take unacceptably long to solve.

Although sampling-based path planners can drastically speed up the time to find some solution, they are not optimal and struggle with specific situations such as narrow passages. There is no “silver bullet” algorithm for solving all path-planning problems and heuristics that lead to massive speed-up in one scenario might be detrimental in others. Also, algorithmic parameters are mostly ad-hoc and correctly tuning them to a specific environment might drastically increase performance.

Take-home lessons

- The first step in path-planning is choosing a map representation that is appropriate to the application.
- This allows the application of generic shortest path finding algorithms, which have applications in a large variety of domains, not limited to robotics.
- A sampling-based planning algorithm finds paths by sampling random points in the environment. Heuristics are used to maximize the exploration of space and bias the direction of search. This makes these algorithms fast, but neither optimal nor complete.
- As the resulting paths are random, multiple trials might lead to totally different results.
- There is no one-size-fits-all algorithm for a path-planning algorithm and care must be taken to select the right paradigm (single-query vs. multi-query), heuristics, and parameters.

Exercises

1. How does the computational complexity of Dijkstra’s algorithm change when moving from 2D to 3D search spaces?
2. A* uses a “heuristic” to bias the search in the expected direction of the goal. Why can it only use a heuristic, not the actual length?
3. Assuming points are sampled uniformly at random in a randomized planning algorithm. Calculate the limiting behaviour of the following ratio (number of points in tree)/(number of points sampled) as the number of points sampled goes to infinity. Assume the total area A_{total} and the area of free space A_{free} within are known.
4. Assuming a kd-tree is used as a nearest-neighbour data structure, and points are sampled uniformly at random, calculate the run-time of inserting a point into a tree of size N. Use “big-Oh” notation, e.g. O(N).
5. What other practical runtime concerns must one consider besides computational complexity alone when doing sampling based motion planning? Can you suggest ways to deal with these other concerns?

CHAPTER OVERVIEW

5: Sensors

Robots are systems that sense, actuate, and compute. So far, we have studied the basic physical principles of actuation, i.e., locomotion and manipulation, and computation, i.e., inverse kinematics and path-planning. We now need to understand the basic principles of robotic sensors that provide the data-basis for computation. The goals of this chapter are

- provide an overview of sensors commonly used on robotic systems
- outline the physical principles that are responsible for uncertainty in sensor-based reasoning

---

5.1: Robotic Sensors

The development of sensors is classically driven by industries other than robotics. These include submarines, automatically opening doors, safety devices for industry, servos for remote-controlled toys, and more recently the cell-phone, automobiles and gaming consoles. These industries are mostly responsible for making “exotic” sensors available at low cost by identifying mass-market applications, e.g., accelerometers and gyroscopes now being used in mass-market smart phones or the 3D depth sensor “Kinect” as part of its XBox gaming console.

Query

Recap the sensors that you are interacting with daily. What sensors do you have in your phone, in your house and kitchen, or in your car?

As we will see later on, sensors are hard to classify by their application. In fact, most problems benefit from every possible source of information that they can obtain. For example, localization can be achieved by counting encoder increments, but also by measuring acceleration, or using vision. All of these approaches differ drastically in their precision and the kind of data that they provide, but none of them is able to completely solve the localization problem on its own. This chapter is therefore organized by the physical quantities (and derivatives thereof), a sensor is measuring, instead of the higher level state estimates it can contribute to.

Query

Think about the kind of data that you could obtain from an encoder, an accelerometer, or a vision sensor on a nonholonomic robot. What are the fundamental differences?

Although an encoder is able to measure position, it is used in this function only on robotic arms. If the robot is nonholonomic, closed tours in its configuration space, i.e., robot motions that return the encoder values to their initial position, do not necessarily drive the robot back to its starting point. In those robots, encoders are therefore mainly useful to measure speed. An accelerometer instead, by definition, measures the derivative of speed. Vision, finally, allows to calculate the absolute position (or the integral of speed) if the environment is equipped with unique features. An additional fundamental difference between those three sensors is the amount and kind of data they provide. An accelerometer samples real-valued quantities that are digitized with some precision. An odometer instead delivers discrete values that correspond to encoder increments. Finally, a vision sensor delivers an array of digitized real-valued quantities (namely colors). Although the information content of this sensor exceeds that of the other sensors by far, cherry-picking the information that are really useful remains a hard, and largely unsolved, problem.

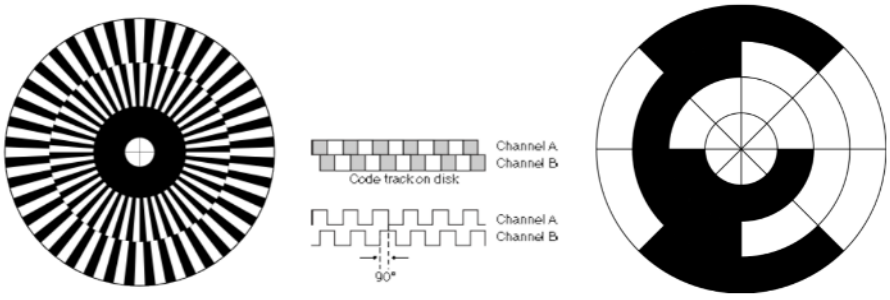


Figure **5.1.1**: From left to right: encoder pattern used in a quadrature encoder, resulting sensor signal (forward motion), absolute encoder pattern (gray coding).

5.2: Proprioception of Robot Kinematics and Internal forces

Proprioception refers to the perception of internal states of a robot. This is different from *exteroception*, which describes sensing of anything outside of the robot. Proprioception includes awareness of the robot's joint angles, its speeds, as well torques and forces.

The main internal sensor is therefore the encoder. Encoders can be used for sensing joint position and speed, as well as — when used together with a spring — a simple force sensor. There are both incremental and absolute encoders. The latter are mostly used in industrial applications, but are not common in mobile robotics. There are magnetical and optical encoders, which both rely on a magnetic or optical beacon turning together with the motor and being sensed by an appropriate sensor that counts every pass-through. The most common encoder in robotics is the optical quadrature encoder. It relies on a pattern rotating with the motor and an optical sensor that can register black/white transitions. Whereas those patterns can be precision manufactured, simple encoders can be made by simply laser-cutting a pattern such as shown in Figure 5.1.1 and reading it with a light sensor.

While a single sensor would be sufficient to detect rotation and its speed, it does not allow for determining the direction of motion. Quadrature encoders therefore have two sensors, A and B, that register an interleaving pattern with distance of a quarter phase. If A leads B, for example, the disk is rotating in a clockwise direction. If B leads A, then the disk is rotating in a counter-clockwise direction. It is also possible to create absolute encoders, an example of which is shown in Figure 5.1, right. This pattern encodes 3-bits, encoding 8 different segments on a disc. Notice that the pattern is arranged in such a way that there is only one bit changing from one segment to the other. This is known as "Gray code". The function of linear encoders is analogous, both for incremental and absolute encoders.

If combined with a spring, such as in a series elastic actuator, rotary and linear encoders can be used as simple force/torque sensors using Hooke's law (F = kx, force equals distance times spring constant). Whereas the series elastic actuator is the most illustrative examples, most load cells operate on the premise of measuring displacements within materials of known properties. Here, measuring changes in resistance or capacitance might be easier choices.

Other means of measuring the actual force at the end-effector or joint torques is measuring the actual current consumed at each joint. Knowing a mechanism's pose allows to calculate the resulting forces and torques across the mechanism as well as the currents required for empty loading conditions. Derivations of these then correspond to additional forces that can hence be calculated.

Finally, there are other means of proprioception, such as simple sensors that can detect when a robot gets picked up, e.g.

5.3: Sensors Using Light

The small form factor and low price of light-sensitive semiconductors have led to a proliferation of light-based sensing relying on a multitude of physical effects. These include reflection, phase shift, and time of flight.

5.3.1. Reflection

Reflection is one of the principles that is easiest to exploit: the closer an object is, the more it reflects light shined at it. This allows to easily measure distance to objects that reflect light well and are not too far away. In order to make these sensors as independent from an object's color (but unfortunately not totally independent), infrared is most commonly chosen. A distance sensor is made from two components: an emitter and a receiver. They work by emitting an infrared signal and then measuring the strength of the reflected signal. A typical response is shown in Figure 5.3.1. The values obtained at an analog-digital converter correspond to the voltage at the infrared receiver and are saturated for low distances (flat line), and quadratically fall off thereafter.

Figure **5.3.1**: Typical response of an infrared distance sensor as a function of distance. Units are left dimensionless intentionally.

When using more than one infrared sensor/emitter pair, e.g., using a camera and a projector not only allows to measure the distance of many points at once, but also to assess the structure of the environment by calculating its impact on the deformation of patterns. For example a straight line becomes a curve when projected onto a round surface. This approach is known as structured light and illustrated in Figure 5.3.2. Thanks to the continuously increasing efficiency of computational systems, a light-weight version of such an approach has become feasible to be implemented at small scale and low cost at around 2010, and emerged as a novel standard in robotic sensing.

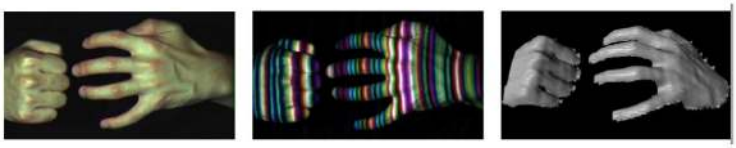


Figure **5.3.2**: From left to right: two complex physical objects, a pattern of colored straight lines and their deformation when hitting the surfaces, reconstructed 3D shape. From (Zhang et al. 2002).

Instead of using line patterns, infrared-based depth image sensors use a speckle pattern (a collection of randomly distributed dots with varying distances), and two computer vision concepts: *depth from focus* and *depth from stereo*. When using a lens with a narrow focal depth, objects that are closer or farther away appear blurred (you can easily observe this on professional portrait photos, which often use this effect for aesthetic purposes). Measuring the "bluriness" of a scene (for known camera parameters) therefore allows for an initial estimate of depth. Depth from stereo instead works by measuring the disparity of the same object appearing in two images taken by cameras that are a known distance apart. Being able to identify the same object in both frames allows to calculate this disparity, and from there the distance of the object. (The farther the object is away, the smaller is the disparity.) This is where the speckle pattern comes in handy, which simply requires to search for blobs with similar size that are close to each other.

5.3.2. Phase shift

As you can see in Figure 5.3.1, reflection can only be precise if distances are short. Instead of measuring the strength (aka amplitude) of the reflected signal, laser distance sensors measure the phase difference of the reflected wave. In order to do this, the emitted light is modulated with a wave-length that exceeds the maximum distance the scanner can measure. If you would use visible light and do this much slower, you would see a light that keeps getting brighter, then getting darker, briefly turns off and then starts getting brighter again. Thus, if you would plot the amplitude, i.e. its brightness, of the emitted signal vs. time you would see a wave that has zero-crossings when the light is dark. As light travels with the speed of light, this wave propagates through space with a constant distance (the wavelength) between its zero crossings. When it gets reflected, the same wave travels back (or at least parts of it that get scattered right back). For example, modern laser scanners emit signals with a frequency of 5 MHz (turning off 5 million times in one second). Together with the speed of light of approximately 300,000km/s, this leads to a wavelength of 60m and makes such a laser scanner useful up to 30m.

When the laser is now at a distance that corresponds exactly to one half the wave-length, the reflected signal it measures will be dark at the exact same time its emitted wave goes through a zero-crossing. Going closer to the obstacle results in an offset that can be measured. As the emitter knows the shape of the wave it emitted, it can calculate the phase difference between emitted and received signal. Knowing the wave-length it can now calculate the distance. As this process is independent from ambient light (unless it has the exact same frequency as the laser being used), the estimates can be very precise. This is in contrast to a sensor that uses signal strength. As the signal strength decays at least quadratically, small errors, e.g. due to fluctuations in the power supply that drives the emitting light, noise in the analog-digital converter, or simply differences in the reflecting surface have drastic impact on the accuracy and precision (see below for a more formal definition of this term).

As the laser distance measurement process is fast, such lasers can be combined with rotating mirrors to sweep larger areas, known as Laser Range Scanners. Such systems have been combined into packages consisting of up to 64 scanning lasers, providing a depth map around a car while driving, e.g. It is also possible to modulate projected images with a phase-changing signal, which is the operational principle of early “time-offlight” cameras, which is not an accurate description of their operation, however.

5.3.3. Time-of-flight

The most precise distance measurements light can provide is by measuring its time of flight. This can be done by counting the time a signal from the emitter becomes visible in the receiver. As light travels very fast (3,000,000,000m/s), this requires highspeed electronics that can measure time periods smaller than nano-seconds in order to achieve centimeter accuracy. In practice this is done by combining the receiver with a very fast (electronic) shutter that operates at the same frequency with which light is emitted. As this timing is known, one can infer the time light must have been traveling by measuring the quantity of photons that have made it back from the reflective surface within one shutter period. Considering a concrete example, light travels 15m in 50ns. Therefore, it will take a pulse 50ns to return from an object at a distance of 7.5m. If the camera sends out a pulse of 50ns length and then closes the receiver with a shutter, the receiver will receive more photons the closer the object is, but no photons if the object is farther away than 7.5m. Given a fast enough and precise circuit that acts as a shutter, it is sufficient to measure the actual amount of light that returns from the emitter.

5.4: Sensors using Sound

5.4.1. Ultra-sound Distance Sensors

An ultra-sound distance sensor operates by emitting an ultrasound pulse and measures its reflection. Unlike a light-based sensor that measures the amplitude of the reflected signal, a sound-based sensor measures the time it took the sound to travel. This is possible, because sound travels at much lower speed (300m/s) than light (300,000km/s). The fact that the sensor actually has to wait for the signal to return leads to a trade-off between range and bandwidth. (Look these definitions up above before you read on.) In other words, allowing a longer range requires waiting longer, which in turn limits how often the sensor can provide a measurement. Although US distance sensors have become less and less common in robotics, they have an advantage over light-based sensors: instead of sending out a ray, the ultra-sound pulse results in a cone with an opening angle of 20 to 40 degrees. By this, US sensors are able to detect small obstacles without the requirement of directly hitting them with a ray. This property makes them the sensor of choice in automated parking helpers in cars.

5.4.2. Texture Recognition

Audible sound consists of high frequency vibrations in the range between 20 Hz and roughly 15 kHz. Microphones are therefore ideally suited to measure vibrations in this range. This allows them to double as the Pascinian corpuscle in human skin cells, which is known to have a resonance frequency of 250 Hz and is mostly responsible for texture recognition. Indeed, rubbing a texture against a microphone can indeed be used for differentiating between tens and hundreds of different textures (Hughes & Correll 2014), with a number of commercial sensors available. These sensors usually calculate the frequency spectrum of the recorded signal, which can then be classified using machine learning techniques. Being able to recognize a texture by touch is important in applications like grasping and navigation through cluttered terrain.

5.5: Inertia-based Sensors

A moving mass does not loose its kinetic energy, but for friction. Likewise, a resting mass will resist acceleration. Both effects are due to “inertia” and can be exploited to measure acceleration and speed.

5.5.1. Accelerometer

An accelerometer can be thought of as a mass on a dampened spring. Considering a vertical spring with a mass hanging down from it, we can measure the acting force F = kx (Hooke’s law) by measuring the displacement x that the mass has stretched the spring. Using the relationship F = am, we can now calculate the acceleration a on the mass m. On earth, this acceleration is roughly 9.81 kgm/s^2. In practice, these spring/mass systems are realized using microelectromechanical systems (MEMS), such as a cantilevered beam whose displacement can be measured, e.g., using a capacitive sensor. Accelerometers measure up to three axes of translational accelerations. Infering a position from this requires integration twice, thereby amplifying any noise, making position estimates using accelerometers alone infeasible. As gravity provides a constant acceleration vector, accelerometers are very good at estimating the pose of an object with respect to gravity.

5.5.2. Gyroscopes

A gyroscope is an electro-mechanical device that can measure rotational orientation. It is complementary to the accelerometer that measures translational acceleration. Classically, a gyroscope consists of a rotating disc that could freely rotate in a system of pivots and gimbals. When moving the system, the inertial momentum keeps the original orientation of the disc, allowing to measure the orientation of the system relative to where the system was started. A variation of the gyroscope is the rate gyro, which measures rotational speed.

What a rate gyro measures can most intuitively be illustrated by considering the implementation of an optical rate gyro. In an optical gyro, a laser beam is split into two beams and send around a circular path in two opposite directions. If this setup is rotated against the direction of one of these laser beams, one laser will have to travel slightly longer than the other, leading to a measurable phase-shift at the receptor. This phase shift is proportional to the rotational speed of the setup. As light with the same frequency and phase will add, and lights with the same frequency but opposite phases will cancel each other, light at the detector will be darker for high rotational velocities. As small-scale optical rate gyros are not practical for multiple reasons, MEMS rate gyros rely on a mass suspended by springs. The mass is actively vibrating, making it subject to Coriolis forces, when the sensor is rotated. Coriolis forces can be best understood by moving orthogonally to the direction of rotation on a vinyl disk player. In order to move in a straight line, you will not only need to move forwards, but also sideways. The necessary acceleration to change the speed of this sideway motion is counteracting the Coriolis force, which is both proportional to the lateral speed (the vibration of the mass in a MEMS sensor) and the rotational velocity, which the device wishes to measure. Note that the MEMS gyro would only be able to measure acceleration if it were not vibrating.

Gyroscopes can measure the rotational speed around three axes, which can be integrated to obtain absolute orientation. As an accelerometer measures along three axes of translation, the combination of both sensors can provide information on motion in all six degrees of freedom. Together with a magnetometer (compass), which provides absolute orientation, this combination is also known as *Inertial Measurement Unit* (IMU).

5.6: Beacon-based Sensors

Localizing an object by triangulation goes back to ancient civilizations orienting themselves using the stars. As stars are only visible during unclouded nights, seafarers have eventually invented systems of artificial beacons emitting light, sound, and eventually radio waves. The most sophisticated of such systems is the Global Positioning System (GPS). GPS consists of a number of satellites in orbit, which are equipped with knowledge about their precise location and have synchronized clocks. These satellites broadcast a radio signal that travels at the speed of light and is coded with its time of emission. GPS receivers can therefore calculate the distance to each satellite by comparing time of emission and time of arrival. As not only the position (x,y,z), but also the time difference between the GPS receiver's clock and the synchronized clocks of the satellites is unknown, four satellites are needed to obtain a "fix". Due to the way information from the satellites is coded, getting an initial fix can take in the order of minutes, but eventually is available multiple times a second. GPS measurements are neither precise nor accurate enough (see below) for small mobile robots, and require to be combined with other sensors, such as IMUs and compasses. (The bearing shown on some GPS receivers is calculated from subsequent positions and is therefore meaningless if the robot is not moving.)

There exist also a variety of indoor-GPS solutions, which consists of either active or passive beacons that are mounted in the environment at known locations. Passive beacons, for example infrared reflecting stickers arranged in a certain pattern or 2D barcodes, can be detected using cameras and their pose can be calculated from their known dimensions. Active beacons instead usually emit radio, ultrasound or a combination thereof, which can then be used to estimate the robot's range to this beacon.

5.7: Terminology

It is now time to introduce a more precise definition of terms such as “speed” and “resolution”, as well as additional taxonomy that is used in a robotic context.

Roboticists differentiate between active and passive sensors. Active sensors emit energy of some sort and measure the reaction of the environment. Passive sensors instead measure energy from the environment. For example, most distance sensors are active sensors (as they sense the reflection of a signal they emit), whereas an accelerometer, compass, or a push-button are passive sensors.

The difference between the upper and the lower limit of the quantity a sensor can measure its known as its range. This should not be confused with the dynamic range, which is the ratio between the highest and lowest value a sensor can measure. It is usually expressed on a logarithmic scale (to the basis 10), also known as “decibel”. The minimal distance between two values a sensor can measure is known as its resolution. The resolution of a sensor is given by the device physics (e.g., a light detector can only count multiples of a quant), but usually limited by the analog-digital conversion process. The resolution of a sensor should not be confused with its accuracy or its precision (which are two different concepts). For example, whereas an infrared distance sensor might yield 4096 different values to encode distances from 0 to 10cm, which suggests a resolution of around 24 micrometers, its precision is far above that (in the order of millimeters) due to noise in the acquisition process.

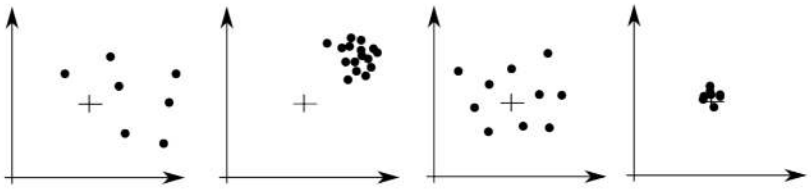


Figure 5.7.1: From left to right, the cross corresponds to the true value of the signal: neither precise nor accurate, precise but not accurate, accurate but not precise, accurate and precise.

Technically, a sensors accuracy is given by the difference between a sensor’s (average) output m and the true value v:

$$accuracy = 1 - \frac{|m - v|}{v} \tag{5.7.1}$$

This measure provides you with a quantity that approaches one for very accurate values and zero if the measurements group far away from the actual mean. In practice, however, this measure is only rarely used and accuracy is provided with absolute values or a percentage at which a value might exceed the true measurement.

A sensor’s precision instead is given by the ratio of range and statistical variance of the signal. Precision is therefore a measure of repeatability of a signal, whereas accuracy describes a systematic error that is introduced by the sensor physics. This is illustrated in Figure 5.7.1.

A GPS sensor is usually precise within a few meters, but only accurate to tens of meters. This becomes most obvious when satellite configurations change, resulting in the precise region jumping by a couple of meters. In practice, this can be avoided by fusing this data with other sensors, e.g. from an IMU.

The speed at which a sensor can provide measurements is known as its *bandwidth*. For example, if a sensor has a bandwidth of 10 Hz, it will provide a signal ten times a second. This is important to know, as querying the sensor more often is a waste of computational time and potentially misleading.

5.8: Exercises

Take-home lessons

- Most of a robot's sensors either address the problem of robot localization or localizing and recognizing objects in its vicinity.
- Each sensor has advantages and drawbacks that are quantified in its range, precision, accuracy, and bandwidth. Therefore, robust solutions to a problem can only be achieved by combining multiple sensors with differing operation principles.
- Solid-state sensors (i.e. without mechanical parts) can be miniaturized and cheaply manufactured in quantity. This has enabled a series of affordable IMUs and 3D depth sensors that will provide the data basis for localization and object recognition on mass-market robotic systems.

Exercises

1. Given a laser scanner with an angular resolution of 0.01 rad and a maximum range of 5.6 meters, what is the minimum range d a robot needs to have from an object of 1cm width to definitely sense it, i.e., hit it with at least one of its rays? You can approximate the distance between two rays with the arc length.
2. Why does the bandwidth of a Ultra-sound based distance sensor decrease significantly when increasing its dynamic range, but that of a laser range scanner does not for typical operation?
3. You are designing an autonomous electric car to transport goods on campus. As you are worried about cost, you are thinking about whether to use a laser scanner or an ultra-sound sensor for detecting obstacles. As you drive rather slow, you are required to sense up to 15 meters. The laser scanner you are considering can sense up to this range and has a bandwidth of 10Hz. Assume 300m/s for the speed of sound in the following.

- Calculate the time it takes until you hear back from the US sensor when detecting an obstacle 15m away. Assume that the robot is not moving at this point.
- Calculate the time it takes until you hear back from the laser scanner. Hint: you don't need the speed of light for this, the answer is in the specs above.
- Assume now that you are moving toward the obstacle. Which sensor will give you a measurement that is closer to your real distance at the time of reading and why?

CHAPTER OVERVIEW

6: Vision

Vision is one of the information-rich sensor system both humans and robots have available. Processing the wealth of information that is generated by vision sensors is still a key challenge, however. The goal of this chapter is to introduce

- images as two-dimensional signals,
- provide an intuition of the wealth of information hidden in low-level information,
- introduce basic convolution and threshold-based image processing algorithms.

6.1: Images as Two-Dimensional Signals

Images are captured by cameras containing matrices of charge-coupled devices (CCD) or similar semi-conductors that can turn photons into electrical signals. These matrices can be read out pixel by pixel and turned into digital values, for example an array of 640 by 480 three-byte tuples corresponding to the red, green, and blue (RGB) components the camera has seen. An example of such data, for simplicity only one color channel, is shown in Figure 6.1.1.

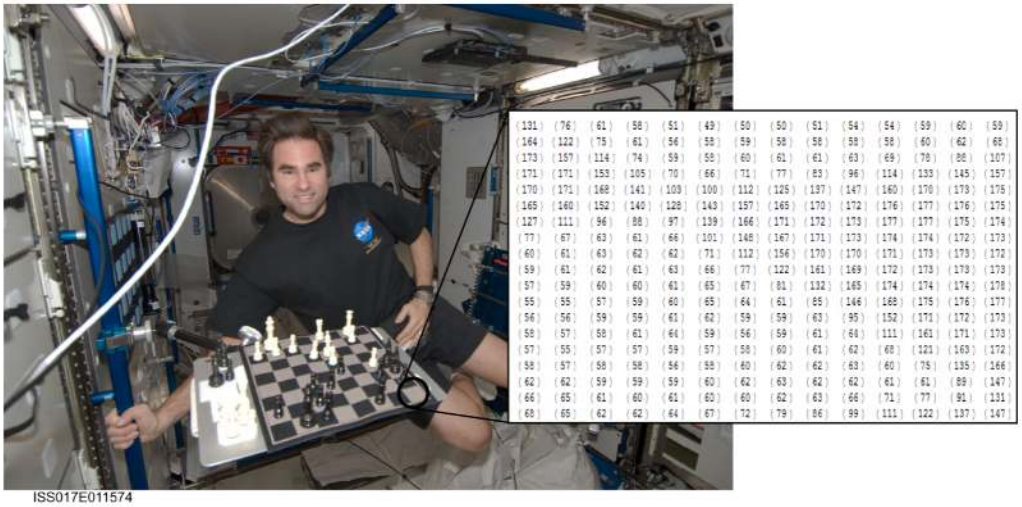


Figure **6.1.1**: A chessboard floating inside the ISS, astronaut Gregory Chamitoff. The inset shows a sample of the actual data recorded by the image sensor. One can clearly recognize the contours of the white tile.

Looking at the data clearly reveals the white tile within the black tiles at the lower-right corner of the chessboard. Higher values correspond to brighter colors (white) and lower values to darker colors. We also observe that although the tiles have to have the same color, the actual values differ quite a bit. It might make sense to think about these values much like we would do if the data would be 1D signal: taking the “derivative”, e.g., along the horizontal rows, would indicate areas of big changes, whereas the “frequency” of an image would indicate how quickly values change. Areas with smooth gradients, e.g., black and white tiles, would then have low frequencies, whereas areas with strong gradients, would contain high frequency information.

This language opens the door to a series of signal processing concepts, such as low-pass filters (supressing high frequency information), high-pass filters (suppressing low frequency information), or band-pass filters (letting only a range of frequencies pass), analysis of the frequency spectrum of the image (the distribution of content at different frequencies), or “convolving” the image with another two-dimensional function. The next sections will provide both an intuition of what kind of meaningful information is hidden in such abstract data and provide concrete examples of signal processing techniques that make this information appear.

6.2: From Signals to Information

Unfortunately, many phenomena that often have very different or even opposite meaning look very similar when looking at the low-level signal. For example, drastic changes in color values do not necessarily mean that the color of a surface indeed has changed. Similar patterns are generated by depth discontinuities, specular highlights, changing lighting conditions, or surface orientation changes. These examples are illustrated in Figure 6.2.1 and make computer vision a hard problem.

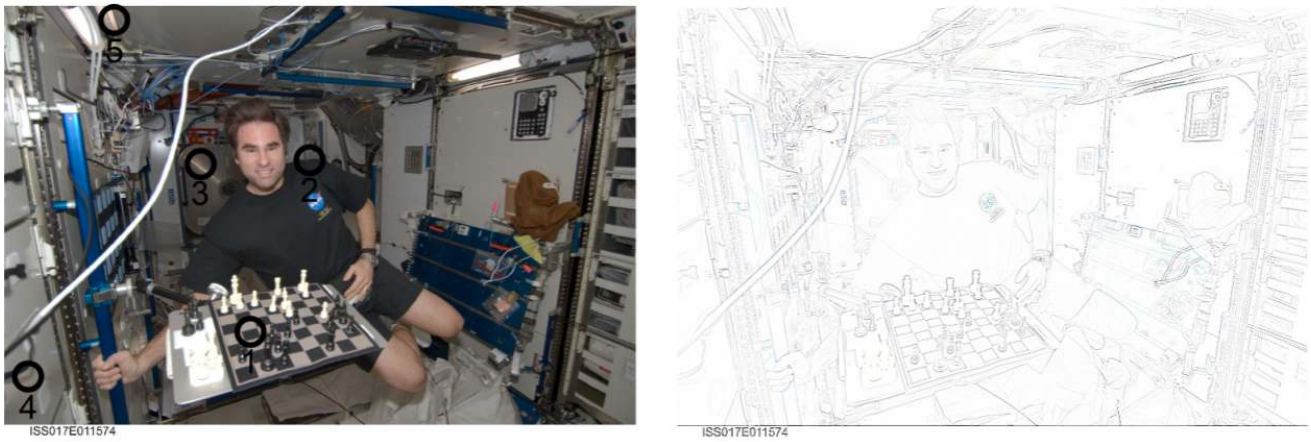


Figure **6.2.1**: Inside of the international space station (left), highlighted areas in which pixel values change drastically (right). Underlying effects that produce similar responses: change in surface properties (1), depth discontinuities (2), specular highlights (3), changing lighting conditions such as shadows (4), or surface orientation changes (5).

This example illustrates that signals alone are not sufficient to understand a phenomenon, but require context. Here, the context does not only refer to surrounding signals, but also high-level conceptional knowledge such as the fact that light sources create shadows and specular highlights, that objects in the front appear larger, and so on. How important such conceptional knowledge is, is illustrated by Figure 6.2.1.

Both images show an identical landscape that once appears to be speckled with craters, once with bubble-like hills. At first glance, both scenes are illuminated from the left, suggesting a change in the landscape. Once information that the sun is illuminating one picture from the left and the other from the right, however, it becomes clear that the craters are simply differently illuminated and what we perceive as bumps eventually turns back into craters.

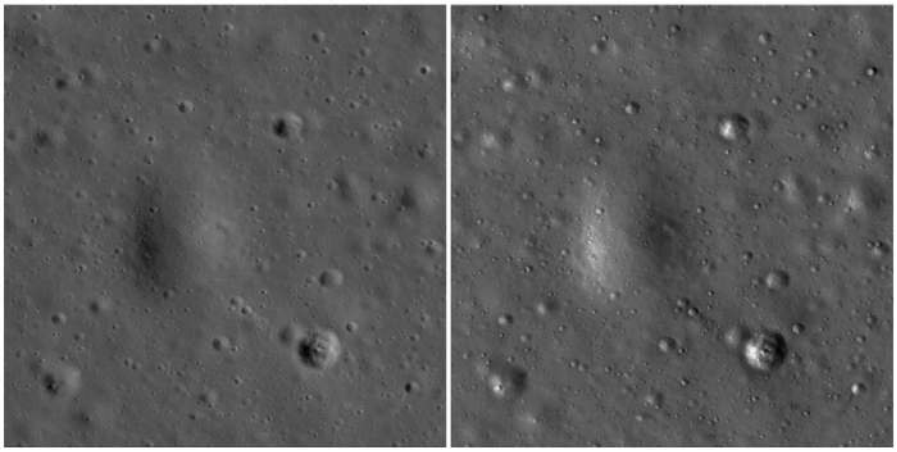


Figure **6.2.2**: Picture of the Apollo 15 landing site during different times of the day. The landscape is identical, but appears to be speckled with craters (lift) or hills (right). Knowing that the sun is illuminating the scene from the left and right, respectively, does explain this effect. Image credit: NASA/GSFC/Arizona State University.

More surprisingly, conceptual knowledge is often sufficient to make up for the lack of low-level cues in an image. An example is shown in Figure 6.4. Here, a Dalmatian dog can be clearly recognized despite absence of cues for its outline, simply by extrapolating its appearance and pose from conceptual knowledge.

These examples illustrate both the advantages and drawbacks of a signal processing approach. While an algorithm will detect interesting signals even there where we don't see, or don't expect them (due to conceptional bias), image understanding not only requires low-level processing, but intelligent combination of the low-level cue's spatial relationship and conceptual knowledge about the world.

6.3: Basic Image Operations

Basic image operations can be thought of as a filter that operates in the frequency or in the space (color) domain. Although most filters directly operate in the color domain, knowing how they affect the frequency domain is helpful in understanding the filter's function. For example, a filter that is supposed to highlight edges, such as shown in Figure 6.2.1 should suppress low frequencies, i.e., areas in which the color values do not change much, and amplify high-frequency information, i.e., areas in which the color values change quickly. The goal of this section is to provide a basic understanding of how basic image processing operation works. The methods presented here, while still valid, have been superseded by more sophisticated implementations that are widely available as software packages or within desktop graphic software.

Figure 6.3.1: The image of a Dalmatian dog can be clearly recognized by most spectators even though low-level cues such as edges are only present for ears, chin and parts of the legs. The contours of the animals are highlighted in a flipped version of the image in the inset.

6.3.1. Convolution-based filters

A filter can be implemented using the convolution operator that convolves function f() with function g().

$$f(x) \star g(x) = \int_{-\infty}^{\infty} f(\tau) g(x-\tau) d\tau \tag{6.3.1}$$

We then call function g() a filter . As will become more clear further below, the convolution literally shifts the function g() across the function f() while multiplying the two. As images are discrete signals, the convolution is usually discrete

$$f[x] \star g[x] = \sum_{i=-\infty}^{\infty} f[i] g[x-i] \tag{6.3.2}$$

For 2D signals, like images, the convolution is also two-dimensional:

$$f[x,y] \star g[x,y] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[i,j] g[x-i, y-j] \tag{6.3.3}$$

Although we have defined the convolution from negative infinity to infinity, both images and filters are usually finite. Images are constrained by their resolution, and filters are usually much smaller than the images themselves. Also, the convolution is commutative, therefore (6.3.3) is equivalent to

$$f[x,y] \star g[x,y] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[x-i, y-j] g[i,j] \tag{6.3.4}$$

Gaussian smoothing

A very important filter is the Gaussian filter. It is shaped like the Gaussian bell function and can be easily stored in a 2D matrix. Implementing a Gaussian filter is surprisingly simple, e.g., such as

$$g(x,y) = \frac{1}{10}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{6.3.5}$$

Using this filter in Equation 6.3.4 on an infinitely large image f() leads to

$$f[x,y] \star g[x,y] = \sum_{i=-1}^{1}\sum_{j=-1}^{1} f[x-i, y-j]g[i,j] \tag{6.3.6}$$

(assuming g(0, 0) addresses the center of the matrix). What now happens is that each pixel f(x, y) becomes the average of that of its neighbors, with its previous value weighted twice (as g(0, 0) = 0.2) that of their neighbors. More concretely,

$$\begin{aligned} f(x,y) = llf(x+1,y+1)g(-1,-1) + f(x+1,y)g(-1,0) + f(x+1,y-1)g(-1,1) \\ + f(x,y+1)g(0,-1) + f(x,y)g(0,0) + f(x,y-1)g(0,1) + f(x-1,y+1)g(1,-1) \\ + f(x-1,y)g(1,0) + f(x-1,y-1)g(1,1) \end{aligned} \tag{6.3.7}$$

Doing this for all x and all y literally corresponds to sliding the filter g() along the image.

An example of filter g(x, y) in action is shown in Figure 6.3.2. The filter acts as a low-pass filter , suppressing high frequency components. Indeed, noise in the image is suppressed, leading also to a smoother edge image, which is shown underneath.

Edge detection

Edge detection can be achieved using another convolution-based filter, the *Sobel* kernel

$$s_x(x,y) = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \qquad s_y(x,y) = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

Here, $s_x(x, y)$ can be used to detect vertical edges, whereas $s_y(x, y)$ highlights horizontal edges. Edge detectors, such as the Canny edge detector therefore run at least two of such filters over an image to detect both horizontal and vertical edges.

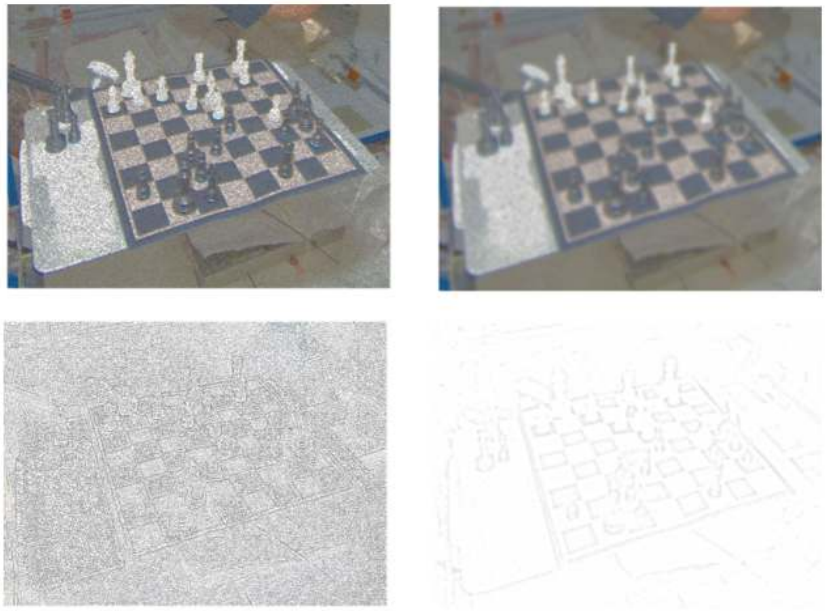


Figure **6.3.2**: A noisy image before (top left) and after filtering with a Gaussian kernel (top right). Corresponding edge images are shown underneath.

Difference of Gaussians

An alternative method for detecting edges is the Difference of Gaussians (DoG) method. The idea is to subtract two images that have each been filtered using a Gaussian kernel with different width. Both filters supress high-frequency information and their difference therefore leads to a band-pass filtered signal, from which both low and high frequencies have been removed. As such, a DoG filter acts as a capable edge detection algorithm. Here, one kernel is usually four to five times wider than the other, therefore acting as a much stronger filter.

Differences of Gaussians can also be used to approximate the *Laplacian of Gaussian*, i.e., the sum of the second derivatives of a Gaussian kernel. Here, one kernel is roughly 1.6 times wider than the other. The band-pass characteristic of DoG and LoGs are important as they highlight high-frequency information such as edges, yet suppress high-frequency noise in the image.

6.3.2. Threshold-based operations

In order to find objects with a certain color or edge intensity, thresholding an image will lead to a binary image that contains "true-false" regions that fit the desired criteria. Thresholds make use of operators like $>$, $<$, $\leq$, $\geq$ and combinations thereof. There also exist adaptive versions that would adapt the thresholds locally, e.g., to make up for changing lighting conditions.

Albeit thresholding is deceptively simple, finding correct threshold values is a hard problem. In particular, actual pixel values change drastically with changing lighting conditions and there is no such thing as "red" or "green" when inspecting the actual values under different conditions.

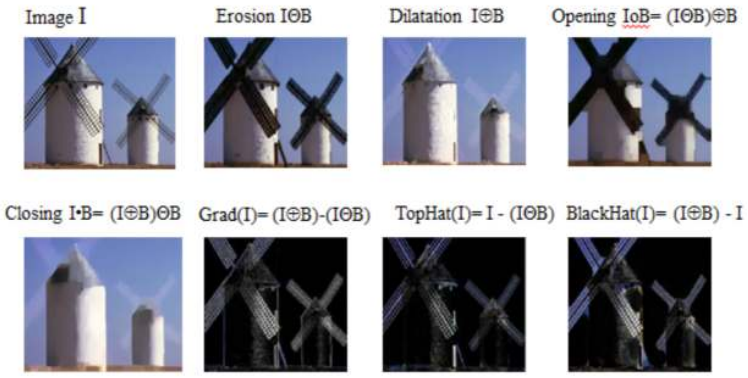


Figure **6.3.3**: Examples of morphological operators erosion and dilation and combinations thereof.

6.3.3. Morphological Operations

Another class of filters are morphological operators which consists of a kernel describing the structure of the operation (this can be as simple as an identity matrix) and a rule on how to change a pixel value based on the values in the neighborhood defined by the kernel.

Important morphological operators are erosion and dilation. The erosion operator assigns a pixel value with the minimum value that it can find in the neighborhood defined by the kernel. The dilation operator assigns a pixel value with the maximum value it can find in the neighborhood defined by the kernel. This is useful, e.g., to fill holes in a line or remove noise. A dilation followed by an erosion is known as a "Closing" and an erosion followed by a dilation as an "Opening". Subtracting erosed and dilated images from each other can also serve as an edge detector. Examples of such operators are shown in Figure 6.3.3.

6.4: Exercises

Exercises

1. Below are shown multiple “Kernels” that can be used for convolution-based image filtering.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & -1 & 0 \\ 0 & -1 & 0 \\ 0 & -1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ 1 & -4 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

   a) Identify the Kernel, which can blur an image.

   b) What kind of features can be detected by the other two kernels?

2. How many for-loops are needed to implement a 2D convolution? Explain your reasoning.

CHAPTER OVERVIEW

7: Feature Extraction

A robot can obtain information about its environment by both active (e.g., ultra-sound, light, and laser) or passive sensing (e.g., acceleration, magnetic field, or cameras). There are only few cases where this information is directly useful to a robot. Before being able to arrive at semantic information such as “I’m in the kitchen”, “this is a cup” or “this is a horse”, is identifying higher-level *features.*

The goal of this chapter is to introduce a series of standard feature detectors such as

- the Hough-transform to detect lines, circles and other shapes,
- numerical methods such as least-squares, split-and-merge and RANSAC to find high-level features in noisy data,
- Scale-invariant features.

7.1: Feature Detection as an Information-Reduction Problem

The information generated by sensors can be quite formidable. For example, a simple webcam generates 640x480 color pixels (red, green and blue) or 921600 Bytes around 30 times per second. A single-ray laser scanner still provides around 600 distance measurements 10 times per second. This is in contrast to the information that a robot actually requires. Consider for example the maze-solving competition “Ratslife” (Section 1.3) in which the robot’s camera can be used to recognize one of 48 different color patterns (Figure 1.3) that are distributed in the environment, or the presence or absence of a charger, essentially reducing hundreds of bytes of camera data to around 6 bit (2^6 = 64 different values) content. The goal of most image processing algorithms is therefore to first reduce information content in a meaningful way and then extract relevant information. In chapter 6, we have seen convolution-based filters such as blurring, detecting edges, or binary operations such as thresholding. We are now interested in methods to extract higher-level features such as lines and techniques to extract them.

7.2: Features

Lines are particularly useful features for localization and can correspond to walls in laser scans, markers on the floor or corners detected in a camera image. Whereas a Sobel filter (Section 6.3.1) can help us to highlight lines and edges in images, additional algorithms are needed to extract structured information such as the orientation and position of a line with respect to the robot.

A desirable property of a feature is that its extraction is repeatable and robust to rotation, scale, and noise in the data. We need feature detectors that can extract the same feature from sensor data, even if the robot has slightly turned or moved farther or closer to the feature. There are many feature detectors available that accomplish this, prominent examples are the Harris corner detector (essentially detecting points in the image where vertical and horizontal lines cross) and the SIFT feature detector. Feature detection is important far beyond robotics and is for example used in hand-held cameras that can automatically stitch images together. Here, feature detectors will “fire” on the same features in two images taken from slightly different perspectives, which allows the camera to calculate the transformation between the two.

This chapter focuses on two important classes of features: line features and scale-invariant features in images (SIFT). Both features provide tangible example for the least-squares and RANSAC algorithms, which are also introduced in this chapter. Both features are representative for a large class of detectors, and have been chosen for their simplicity, providing a basis for under standing the function of more complex feature detectors.

7.3: Line Recognition

Why are lines a useful feature? As you will see next chapter, the key challenge in estimating a robot's pose is unreliable odometry, in particular when it comes to turning. Here, a simple infrared sensor measuring the distance to a wall can provide the robot with a much better feel for what actually happened. Similarly, if a robot has the ability to track markers in the environment using vision, it gets another estimate on how much it is actually moving. How information from odometry and other sensors can be fused not only to localize the robot, but also to create maps of its environment, will be the focus of the remainder of this book.

A laser scanner or similar device pointed against a wall will return a suite of N points at position (xi , yi) in the robot's coordinate system. These points can also be represented in polar coordinates (ρi , θi). We can now imagine a line running through these points that is parametrized with a distance r and an angle α. Here, r is the distance of the robot to the wall and α its angle. As all sensors are noisy, each point will have distance di from the "optimal" line running through the points. These relationships are illustrated in Figure 7.2.1.

7.3.1. Line fitting using least squares

Using simple trigonometry we can now write

$$\rho_i \cos(\theta_i - \alpha) - r = d_i \tag{7.3.1}$$

Different line candidates — parametrized by r and α — will have different values for d_i . We can now write an expression for the total error $S_{r,\alpha}$ as

$$S_{r,\alpha} = \sum_{i=1}^{N} d_i^2 = \sum_i (\rho_i \cos(\theta_i - \alpha) - r)^2 \tag{7.3.2}$$

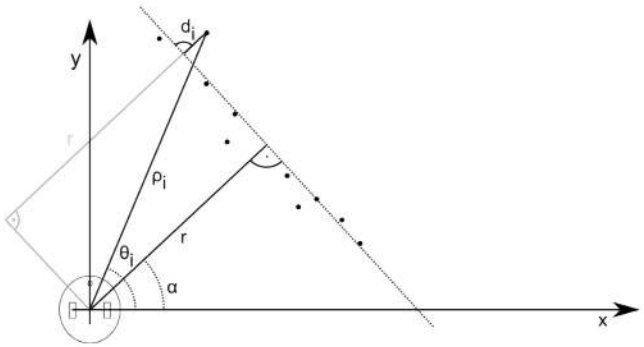


Figure **7.3.1**: A 2D point cloud recorded by a laser-scanner or similar device. A line (dashed) is fitted through the points in a least-square sense.

Here, we square each individual error to account for the fact that a negative error, i.e. a point left of the line, is as bad as a positive error, i.e. a point right of the optimal line. In order to optimize $S_{r,\alpha}$, we need to take the partial derivatives with respect to r and α, set them zero, and solve the resulting system of equations for r and α.

$$\begin{aligned} \frac{\partial S}{\partial \alpha} &= 0 \\ \frac{\partial S}{\partial r} &= 0 \end{aligned} \tag{7.3.3}$$

Here, the symbol ∂ indicates that we are taking a partial derivative. Solving for r and α is involved, but possible (Siegwart et al. 2011):

$$\alpha = \frac{1}{2} atan \left(\frac{\frac{1}{N}\sum \rho_i^2 sin2\theta_i - \frac{2}{N^2}\sum\sum \rho_i\rho_j cos\theta_i sin_j}{\frac{1}{N}\sum \rho_i^2 cos2\theta_i - \frac{1}{N^2}\sum\sum \rho_i\rho_j cos(\theta_i + \theta_j)} \right) \tag{7.3.4}$$

and

$$r = \frac{\sum \rho_i cos(\theta_i - \alpha)}{N} \tag{7.3.5}$$

We can therefore calculate the distance and orientation of a wall captured by our proximity sensors relative to the robot's positions or the height and orientation of a line in an image based on a collection of points that we believe might belong to a line.

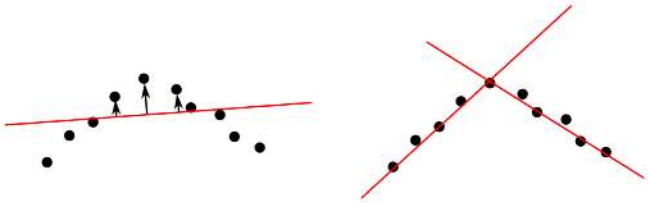


Figure **7.3.2**: Split-and-merge algorithm. Initial least-square fit of a line (left). Splitting the data-set at the point with the highest error (after picking a direction) allows fitting two lines with overall lesser error.

This approach is known as the least-square method and can be used to fit data to any parametric model. The general approach is to describe the fit between the data and the model in terms of an error. The best fit will minimize this function, which will therefore have a zero derivative at this point. If the result cannot be obtained analytically as in this example, numerical methods have to be used to find the best fit that minimizes the quadratic error.

7.3.2. Split-and-merge algorithm

A key problem with this approach is that it is often unclear how many lines there are and where a line starts and where it ends. Looking through the camera, for example, we will see vertical lines corresponding to wall corners and horizontal ones that correspond to wall-floor intersections and the horizon; using a distance sensor, the robot might detect a corner. We therefore need an algorithm that can separate point clouds into multiple lines. One possible approach is to find the outlier with the strongest deviation from a fitted line and split the line at this point. This is illustrated in Figure 7.3.2. This can be done iteratively until each line has no outliers above a certain threshold.

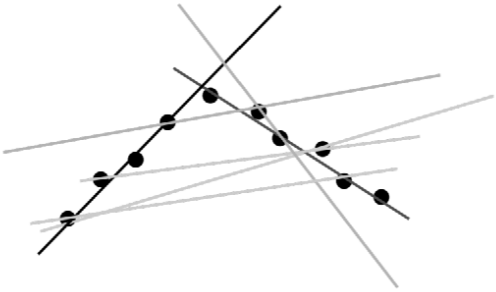


Figure **7.3.3**: Random Sample and Consensus (RANSAC). Random lines are evaluated by counting the number of points close by ("inliers"), darker lines are better fits.

7.3.3. RANSAC: Random Sample and Consensus

If the number of "outliers" is large, a least square fit will generate poor results as it will generate the "best" fit that accomodates both "inliers" and "outliers". Also, split-and-merge algorithms will fail as they are extremely susceptive to noise: depending on the actual parameters every outlier will split a potential line into two. A solution to this problem is to randomly sample possible lines and keep those that satisfy a certain desired quality given by the number of points being somewhat close to the best fit. This is illustrated in Figure 7.3.3, with darker lines corresponding to better fits. RANSAC usually requires two parameters, namely the number of points required to consider a line to be a valid fit, and the maximum di from a line to consider a point an inlier and not an outlier. The algorithm proceeds as follows: select two random points from the set and connect them with a line. Grow this line by di in both directions and count the number of inliers. Repeat this until one or more lines that have sufficient number of inliers are found, or a maximum number of iterations is reached.

The RANSAC algorithm is fairly easy to understand in the line fitting application, but can be used to fit arbitrary parametric models to any-dimensional data. Here, its main strength is to cope with noisy data.

Given that RANSAC is random, finding a really good fit will take quite some time. Therefore, RANSAC is usually used only as a first step to get an initial estimate, which can then be improved by some kind of local optimization, such as least-squares, e.g.

7.3.4. The Hough Transform

The Hough transform can best be understood as a voting scheme to guess the parametrization of a feature such as a line, circle or other curve (Duda & Hart 1972). For example, a line might be represented by y = mx+c, where m and c are the gradient and offset. A point in this parameter space (or “Hough-space”) then corresponds to a specific line in x−y-space (or “image-space”). The Hough-transform now proceeds as follows: for every pixel in the image that could be part of a line, e.g., white pixels in a thresholded image after Sobel filtering, construct all possible lines that intersect this point. (Drawing an image of this would look like a star). Each of these lines has a specifc m and c associated with it, for which we can add a white dot in Houghspace. Continuing to do this for every pixel of a line in an image will yield many m − c pair, but only one that is common among all those pixels of the line in the image: the actual m−c parameters of this line. Thinking about the number of times a point was highlighted in Hough-space as brightness, will turn a line in image space into a bright spot in Hough-space (and the other way round). In practice, a polar representation is chosen for lines. This is shown in Figure 7.4.1. The Hough transform also generalizes to other parametrization such as circles.

7.4: Scale-Invariant Feature Transforms

Scale-invariant feature transforms are a class of algorithms/signalprocessing techniques that allow to extract features that are easily detectable across different scales (or distances to an object), independent of their rotation, and to some extent robust to affine transformations, i.e., views of the same object from different perspectives, and illumination changes. An early algorithm in this class is the SIFT algorithm (Lowe 1999), which however has lost popularity due to closed-source and licensing cost, and has been replaced in the past with SURF (Speed-Up Robust Feature) (?) and ORB (?), which are freely available and have slightly different performance and varying use cases. As the math behind SURF is more involved, we focus on the intuition behind SIFT and encourage the reader to download and play with the various open-source implementations of other feature detectors that are available open source.

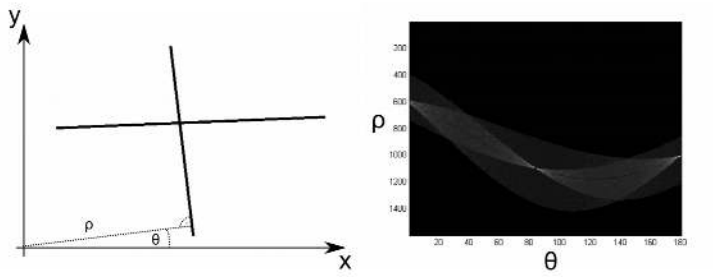


Figure **7.4.1**: Lines in an image (left) transposed into Hough-space ρ (distance from origin) and θ (angle of normal with respect to origin). Bright spots in the Hough image (right) correspond to parameters that have received the most "votes" and clearly show the two lines at around 90° and 180° .

7.4.1. Overview

SIFT proceeds in multiple steps. Descriptions of the algorithm often include its application to object recognition, but these algorithms are independent of feature generation (see below).

- Differences of Gaussians (DoG) at different scales:
- Generate multiple scaled versions of the same image by re-sampling every 2nd, 4th and so on pixel.
- Filtering each scaled picture with various Gaussian filters of different variance.
- Calculating the difference between pairs of filtered images. This is equivalent to a DoG filter.
- Detecting local minima and maxima in the DoG images across different scales (Figure 7.4.2, left) and reject those with low contrast (Figure 7.4.2, right).

Figure **7.4.2**: After scale space extrema are detected (left), the SIFT algorithm discards low contrast keypoints (center) and then filters out those located on edges (right). © Lukas Mach CC-BY 3.0

- Reject extrema that are along edges by looking at the second derivative around each extrema (Figure 7.5, right). Edges have a much larger principal curvature across them than along them.
- Assign a "magnitude" and "orientation" to each remaining extrema (keypoint). The magnitude is the squared difference between neighboring pixels and the orientation is the angle between magnitude along the y-axis vs. magnitude along the x-axis. These calculations are made for all pixels in a fixed neighborhood around the initial keypoint, e.g., in a 16x16 pixel neighborhood.
- Collect orientations of neighboring pixels in a histogram, e.g., 36 bins each covering 10 degrees. Maintain the orientation corresponding to the strongest peak and associate it with the keypoint.

- Repeat 4th step, but for four 4x4 pixel areas around the keypoint in the image scale that has the most extreme minima/maxima. Here, only 8 bins are used for the orientation histogram. As there are 16 histograms in a 16x16 pixel area, the feature descriptor has 128 dimensions.
- The feature descriptor vector is normalized, tresholded, and again normalized to make it more robust against illumination changes.
- Local gradient magnitude and orientation are grouped into bins and create a 128-dimensional feature descriptor.

The resulting 128 dimensional feature vectors are now scale-invariant (due to step 2), rotation-invariant (due to step 5), and robust to illumination changes (due to step 7).

7.4.2. Object Recognition using Scale-Invariant Features

Scale-invariant features of training images can be stored in a database and can be used to identify these objects in the future. This is done by finding all features in an image and comparing them with those in the database. This comparison is done by using the Euclidian distance as metric and searching a kd tree (with d=128). In order to make this approach robust, each object needs to be identified by at least 3 independent features. For this, each descriptor stores the location, scale and orientation of it relative to some common point on the object. This allows each detected feature to “vote” for the position of the object that it is most closely associated with in the database. This is done using a Hough-transform. For example, position (2 dimensions) and orientation (1 dimension) can be discretized into bins (30 degree width for orientation); bright spots in Hough-space then correspond to an object pose that has been identified by multiple features.

7.5: Exercises

Take-home lessons

1. Features are “interesting” information in sensor data that are robust to variations in rotation and scale as well as noise.
2. Which features are most useful depends on the characteristics of the sensor generating the data, the structure of the environment, and the actual application.
3. There are many feature detectors available some of which operating as simple filters, others relying on machine learning techniques.
4. Lines are among the most important features in mobile robotics as they are easy to extract from many different sensors and provide strong clues for localization.

Exercises

1. Think about what information would make good features in different operating scenarios: a supermarket, a warehouse, a cave.
2. What other features could you detect using a Hough transform? Can you find parameterizations for a circle, a square or a triangle?
3. Do an online search for SIFT. What other similar feature detectors can you find? Which provide source code that you can use online?
4. A line can be represented by the function y = mx + c. Then, the Hough-space is given by a 2D coordinate system spanned by m and c.

- Think about a line representation in polar coordinates. What components does the Hough-space consist of in this case?
- Derive a parameterization for a circle and describe the resulting Hough space.

CHAPTER OVERVIEW

8: Uncertainty and Error Propagation

Robots are systems that combine sensing, actuation, computation, and communication. Except for computation, all of its sub-systems are subject to a high degree of uncertainty. This can be observed in daily life: phone calls often are of poor quality, making it hard to understand the other party, characters are difficult to read from far away, the front wheels of your car slip when accelerating on a rainy road from a red light, or your wireless device has a hard time getting a connection. In robotics, measurements taken by on-board sensors are sensitive to changing environmental conditions and subject to electrical and mechanical limitations. Similarly, actuators are not accurate as joints and gears have backlash and wheels do slip. Finally, communication, in particular, wireless either via radio or infrared, is notoriously unreliable.

The goals of this chapter are to understand

- how to treat uncertainty mathematically using probability theory,
- how measurements with different uncertainty can be combined,
- how error propagates when taking multiple measurements in a row.

This chapter requires an understanding of random variables, probability density functions, and in particular the Normal distribution. These concepts are explained in a robotic sensing context in Appendix C.1.

8.1: Uncertainty in Robotics as Random Variable

As quantities such as “distance to a wall”, “position on the plane” or “I can see a blue cross (yes/no)” are uncertain, we can consider them random variables. A *random variable* can be thought of as the outcome of a “random” experiment, such as the face shown when throwing a dice.

Experiments in robotics rarely involve explicit randomness. Instead, sensors are intrinsically noisy due to the physical phenomena associated with them. As sensor readings therefore can be considered random variables, also quantities derived from one or more sensors, such as the examples above, are random variables. This chapter focusses on how to characterize the uncertainty of such aggregated quantities from the uncertainty that characterizes the individual sensors.

8.2: Error Propogation

It turns out that the Gaussian Distribution is very appropriate to model prominent random processes in robotics: the robot's position and distance measurements. A differential wheel robot that drives along a straight line, and is subject to slip, will actually increase its uncertainty the farther it drives. Initially at a known location, the expected value (or mean) of its position will be increasingly uncertain, corresponding to an increasing variance. This variance is obviously somehow related to the variance of the underlying mechanism, namely the slipping wheel and (comparably small) encoder noise. Interestingly, we will see its variance grow much faster orthogonal to the robot's direction, as small errors in orientation have a much larger effect than small errors in longitudinal direction. This is illustrated in Figure 8.2.1.

Similarly, when estimating distance and angle to a line feature from point cloud data, the uncertainty of the random variables describing distance and angle to the line are somewhat related to the uncertainty of each point measured on the line. These relationships are formally captured by the error propagation law.

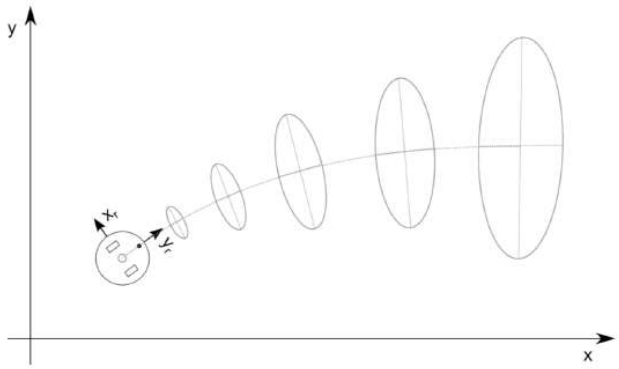


Figure 8.2.1: Two-dimensional Normal distribution depicting growing uncertainty as the robot moves. Albeit starting with equal uncertainty in x and y, the large effect of small errors in orientation lets the error grow faster in y-direction of the robot

The key intuition behind the error propagation law is that the variance of each component that contributes to a random variable should be weighted as a function of how strongly this component influences this random variable. Measurements that have little effect on the aggregated random variable should also have little effect on its variance and vice versa. "How strongly" something affects something else can be expressed by the ratio of how little changes of something relate to little changes in something else. This is nothing else than the partial derivative of something with respect to something else. For example, let y = f(x) be a function that maps a random variable x, e.g., a sensor reading, to a random variable y, e.g., a feature. Let the standard deviation of x be given by σ_x. We can then calculate the variance σ^2_y by

$$\sigma_y^2 = \left(\frac{\partial f}{\partial x}\right)^2 \sigma_x^2 \tag{8.2.1}$$

In case y = f(x) is a multivariable function that maps *n* inputs to *m* outputs, variances become *covariance matrices*. A covariance matrix holds the variance of each variable along its diagonal and is zero otherwise, if the random variables are not correlated. We can then write

$$\Sigma^Y = J\Sigma^X J^T \tag{8.2.2}$$

where Σ^X and Σ^Y are the covariance matrices holding the variances of the input and output variables, respectively, and J is a *mxn Jacobian* matrix, which holds the partial derivatives $\partial f_i/\partial x_j$. As J has n columns, each row contains partial derivatives with respect to x_1 to x_n.

8.2.1. Example: Line Fitting

Let's consider an example of estimating angle α and distance r of a line from a set of points given by (ρ_i , θ_i) using Equations 7.3.4–7.3.5. We can now express the relationship of changes of a variable such as ρ_i to changes in α by

$$\frac{\partial \alpha}{\partial \rho_i} \tag{8.2.3}$$

Similarly, we can calculate $\partial\alpha/\partial\theta_i$, $\partial r/\partial\rho_i$ and $\partial r/\partial\theta_i$. We can actually do this, because we have derived analytical expressions for α and r as a function of θ_i and ρ_i in Chapter 7.

We are now interested in deriving equations for calculating the variance of α and r as a function of the variances of the distance measurements. Let's assume each distance measurement ρ_i has variance $\sigma^2_{\rho i}$ and each angular measurement θ_i has variance $\sigma^2_{\theta i}$. We now want to calculate σ^2_α as the weighted sum of $\sigma^2_{\rho i}$ and $\sigma^2_{\theta i}$, each weighted by its influence on α. More generally, if we have I input variables X_i and K output variables Y_k, the covariance matrix of the output variables σ_Y can be expressed as $\sigma^2_Y = \partial f^2/\partial X \sigma^2_X$ where σ_X is the covariance matrix of input variables and J is a Jacobian matrix of a function f that calculates Y from X and has the form

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial X_1} & \cdots & \frac{\partial f_1}{\partial X_1} \\ \vdots & \cdots & \vdots \\ \frac{\partial f_K}{\partial X_1} & \cdots & \frac{\partial f_K}{\partial X_1} \end{bmatrix} \tag{8.2.4}$$

In the line fitting example F_X would contain the partial derivatives of α with respect to all ρ_i (i-entries) followed by the partial derivates of α with respect to all θ_i in the first row. In the second row, F_X would hold the partial derivates of r with respect to ρ_i followed by the partial derivates of r with respect to θ_i . As there are two output variables, α and r, and 2I input variables (each measurement consists of an angle and distance), F_X is a 2 x (2I) matrix.

The result is therefore a 2x2 covariance matrix that holds the variances of α and r on its diagonal.

8.2.2. Example: Odometry

Whereas the line fitting example demonstrated a many-toone mapping, odometry requires to calculate the variance that results from multiple sequential measurements. Error propagation allows us here to not only express the robot's position, but also the variance of this estimate. Our "laundry list" for this task looks as follows:

1. What are the input variables and what are the output variables?
2. What are the functions that calculate output from input?
3. What is the variance of the input variables?

As usual, we describe the robot's position by a tuple (x, y, θ). These are the three output variables. We can measure the distance each wheel travels Δs_r and Δs_l based on the encoder ticks and the known wheel radius. These are the two input variables. We can now calculate the change in the robot's position by calculating

$$\Delta x = \Delta s cos(\theta + \Delta\theta/2) \tag{8.2.5}$$

$$\Delta y = \Delta s sin(\theta + \Delta\theta/2) \tag{8.2.6}$$

$$\Delta\theta = \frac{\Delta s_r - \Delta s_l}{b} \tag{8.2.7}$$

with

$$\Delta s = \frac{\Delta s_r + \Delta s_l}{2} \tag{8.2.8}$$

The new robot's position is then given by

$$f(x, y, \theta, \Delta s_r, \Delta s_l) = [x, y, \theta]^T + [\Delta x \quad \Delta y \quad \Delta\theta]^T \tag{8.2.9}$$

We thus have now a function that relates our measurements to our output variables. What makes things complicated here is that the output variables are a function of their previous values. Therefore, their variance does not only depend on the variance of the input variables, but also on the previous variance of the output variables. We therefore need to write

$$\Sigma_{p'} = \nabla_p f \Sigma_p \nabla_p f^T + \nabla_{\Delta_{r,l}} f \Sigma_\Delta \nabla_{\Delta_{r,l}} f^T \tag{8.2.10}$$

The first term is the error propagation from a position p = [x, y, θ] to a new position p 0 . For this we need to calculate the partial derivatives of f with respect to x, y and θ. This is a 3x3 matrix

$$\nabla_p f = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial \theta} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\Delta s sin(\theta + \Delta\theta/2) \\ 0 & 1 & \Delta s cos(\theta + \Delta\theta/2) \\ 0 & 0 & 1 \end{bmatrix} \tag{8.2.11}$$

The second term is the error propagation of the actual wheel slip. This requires calculating the partial derivatives of f with respect to Δs_r and Δs_l , which is a 3x2 matrix. The first column contains the partial derivatives of x, y, θ with respect to Δs_r. The second column contains the partial derivatives of x, y, θ with respect to Δs_l :

$$\nabla_{\Delta_{r,l}} f = \begin{bmatrix} \frac{1}{2}\cos(\theta + \frac{\Delta\theta/2}{b}) - \frac{\Delta s}{2b}\sin(\theta + \frac{\Delta\theta}{b}) & \frac{1}{2}\cos(\theta + \frac{\Delta\theta/2}{b}) - \frac{\Delta s}{2b}\sin(\theta + \frac{\Delta\theta}{b}) \\ \frac{1}{2}\sin(\theta + \frac{\Delta\theta/2}{b}) + \frac{\Delta s}{2b}\sin(\theta + \frac{\Delta\theta}{b}) & \frac{1}{2}\sin(\theta + \frac{\Delta\theta/2}{b}) + \frac{\Delta s}{2b}\sin(\theta + \frac{\Delta\theta}{b}) \\ \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \tag{8.2.12}$$

Finally, we need to define the covariance matrix for the measurement noise. As the error is proportional to the distance traveled, we can define Σ_Δ by

$$\Sigma_\Delta = \begin{bmatrix} k_r |\Delta s_r| & 0 \\ 0 & k_l |\Delta s_l| \end{bmatrix} \tag{8.2.13}$$

Here k_r and k_l are constants that need to be found experimentally and $|\cdot|$ indicating the absolute value of the distance traveled. We also assume that the error of the two wheels is independent, which is expressed by the zeros in the matrix.

We now have all ingredients for Equation 8.2.10, allowing us to calculate the covariance matrix of the robot's pose much like shown in Figure 8.2.1.

8.3: Exercises

Take-home lessons

- Uncertainty can be expressed by means of a probability density function.
- More often than not, the Gaussian distribution is chosen as it allows treating error with powerful analytical tools.
- In order to calculate the uncertainty of a variable that is derived from a series of measurements, we need to calculate a weighted sum in which each measurement's variance is weighted by its impact on the output variable. This impact is expressed by the partial derivative of the function relating input to output.

Exercises

1. Given two observations $\hat{q}_1$ and $\hat{q}_2$ with variances σ_1 and σ_2 of a normal distributed process with actual value $\hat{q}$, an optimal estimate can be calculated by minimizing the expression

$$S = \frac{1}{\sigma_1^2}(\hat{q} - \hat{q}_1)^2 + \frac{1}{\sigma_2^2}(\hat{q} - \hat{q}_2)^2 \tag{8.3.1}$$

Calculate $\hat{q}$ so that S is minimized.

2. An ultrasound sensor measures distance x = cΔt/2. Here, c is the speed of sound and Δt is the difference in time between emitting and receiving a signal.

- Let the variance of your time measurement Δt be σ^2_t . What can you say about the variance of x, when c is assumed to be constant? Hint: how does a change in Δt affect x?
- Now assume that c is changing depending on location, weather, etc. and can be estimated with variance σ^2_c . What is the variance of x now?

3. Consider a unicycle that turns with angular velocity φ and has radius r. Its speed is thus a function of φ and r and is given by

$$v = f(\phi, r) = r\phi \tag{8.3.2}$$

Assume that your measurement of φ is noisy and has a standard deviation σ_φ. Use the error propagation law to calculate the resulting variance of your speed estimate σ^2_v.

CHAPTER OVERVIEW

9: Localization

Robots employ sensors and actuators that are subject to uncertainty. Chapter 8 describes how to quantify this uncertainty using probability density functions that associate a probability with each possible outcome of a random process, such as the reading of a sensor or the actual physical change of an actuator. A possible way to localize a robot in its environment is to extract high-level features (Chapter 7), such as the distance to a wall from a number of different sensors. As the underlying measurements are uncertain, these measurements will be subject to uncertainty. How to calculate the uncertainty of a feature from the uncertainty of the sensors that detect this feature, is covered by the error propagation law. The key insight is that the variance of a feature is the weighted sum of all contributing sensors' variances, weighed by their impact on the feature of interest. This impact can be approximated by the derivative of the function that maps a sensor's input to the measurement of the feature.

Unfortunately, uncertainty keeps propagating without the ability to correct measurements. The goals of this chapter are to present mathematical tools and algorithms that will enable you to actually shrink the uncertainty of a measurement by combining it with additional observations. In particular, this chapter will cover

- Using landmarks to improve the accuracy of a discrete position estimate (Markov Localization)
- Approximating continuous position estimates (Particle Filter)
- Optimal sensor fusion to estimate a continuous position estimate (Extended Kalman Filter)

9.1: Motivating Example

Imagine a floor with three doors, two of which are closer together, and the third farther down the corridor (Figure 9.1). Imagine know that your robot is able to detect doors, i.e., is able to tell whether it is in front of a wall or in front of a door. Such features can serve the robot as a landmark. Given a map of this simple environment and no information whatsoever where our robot is located, we can use landmarks to drastically reduce the space of possible locations once the robot has passed one of the doors. One way of representing this belief is to describe the robot's position with three Gaussian distributions, each centered in front of a door and its variance a function of the uncertainty with which the robot can detect a door's center. (This approach is known as a multi-hypothesis belief.) What happens if the robot continues to move? From the error propagation law we know:

1. The Gaussians describing the robot's 3 possible locations will move with the robot.
2. The variance of each Gaussian will keep increasing with the distance the robot moves.

What happens if the robot arrives at another door? Given a map of the environment, we can now map the three Gaussian distributions to the location of the three doors. As all three Gaussians will have moved, but the doors are not equally spaced, only some of the peaks will coincide with the location of a door. Assuming we trust our door detector much more than our odometry estimate, we can now remove all beliefs that do not coincide with a door. Again assuming our door detector can detect the center of a door with some accuracy, our location estimate's uncertainty is now only limited by that of the door detector.

Things are just slightly more complicated if our door detector is also subject to uncertainty: there is a chance that we are in front of a door, but haven't noticed it. Then, it would be a mistake to remove this belief. Instead, we just weight all beliefs with the probability that there could be a door. Say our door detector detects false-positives with a 10% chance. Then, there is a 10% chance to be at any location that is not in front of door, even if our detector tells us we are in front of a door. Similarly, our detector might detect false-negatives with 20% chance, i.e., tell us there is no door even though the robot is just in front of it. Thus, we would need to weigh all locations in front of a door with 20% chance and all locations not in front of a door with 80% likelihood if our robot tells us there is no door, even if we are indeed in front of one.

9.2: Markov Localization

Calculating the probability to be at a certain location given the likelihood of certain observations is nothing else as a conditional probability. There is a formal way to describe such situations: Bayes' Rule (Section C.2):

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)} \tag{9.2.1}$$

9.2.1. Perception Update

How does this map into a Localization framework? Let's assume, event A is equivalent to be at a specific location loc. Lets also assume that event B corresponds to the event to see a particular feature feat. We can now rewrite Bayes' rule to

$$P(loc|feat) = \frac{P(loc)P(feat|loc)}{P(feat)} \tag{9.2.2}$$

Rephrasing Bayes' rule in this way, we can calculate the probability to be at location loc, given that we see feature feat. This is known as *Perception Update*. For example, loc could correspond to door 1, 2 or 3, and *feat* could be the event of sensing a door. What do we need to know to make use of this equation?

1. We need to know the prior probability to be at location loc *P(loc)*
2. We need to know the probability to see the feature at this location P*(feat|loc)*
3. We need the probability to encounter the feature feat *P(feat)*

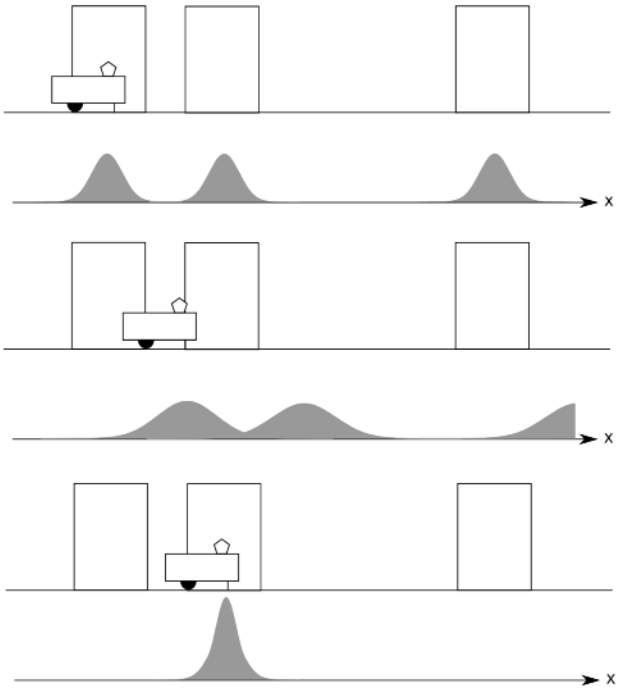


Figure 9.2.1: A robot localizing itself using a "door detector" in a known map. Top: Upon encountering a door, the robot can be in front of any of the three doors. Middle: When driving to the right, the Gaussian distributions representing its location also shift to the right and widen, representing growing uncertainty. Bottom: After detecting the second door, the robot can discard hypotheses that are not in front of the door and gains certainty on its location.

Let's start with (3), which might be the most confusing part of information we need to collect. The answer is simple, no matter what *P(feat)* is, it will cancel out as the probability to be at any of the possible locations has to sum up to 1. (A simpler, although less accurate, explanation would be that the probability to sense a feature is constant and therefore does not matter.)

The prior probability to be at location *loc*, *P(loc)*, is called the *belief model*. In the case of the 3-door example, it is the value of the Gaussian distribution underneath the door corresponding to *loc*.

Finally, we need to know the probability to see the feature *feat* at location loc *P(feat|loc)*. If your sensor were perfect, this probability is simply 1 if the feature exists at this location, or 0 if the feature cannot be observed at this location. If your sensor is not perfect, *P(feat|loc)* corresponds to the likelihood for the sensor to see the feature if it exists.

The final missing piece is how to best represent possible locations. In the graphical example in Figure 9.2.1 we assumed Gaussian distributions for each possible location. Alternatively, we can also discretize the world into a grid and calculate the likelihood of the robot to be in any of its cells. In our 3-door world, it might make sense to choose grid cells that have just the width of a door.

9.2.2. Action Update

One of the assumptions in the above thought experiment was that we know with certainty that the robot moved right. We will now more formally study how to treat uncertainty from motion. Recall that odometry input is just another sensor that we assume to have a Gaussian distribution; if our odometer tells us that the robot traveled a meter, it could have traveled a little less or a little more, with decreasing likelihood. We can therefore calculate the posterior probability of the robot moving from a position *loc'* to *loc* given its odometer input *odo*:

$$P(loc' -> loc|odo) = P(loc' -> loc)P(odo|loc' -> loc)/P(odo) \tag{9.2.3}$$

This is again Bayes' rule. The unconditional probability P(loc'− > loc) is the prior probability for the robot to have been at location loc' . The term P(odo|loc'− > loc) corresponds to the probability to get odometer reading odo after traveling from a position loc' to loc. If getting a reading of the amount odo is reasonable for the distance from loc' to loc this probability is high. If it is unreasonable, for example if the distance is larger than what is physically possible, this probability should be very low.

As the robot's location is uncertain, the real challenge is now that the robot could have potentially been everywhere to start with. We therefore have to calculate the posterior probability P(loc|odo) for all possible positions loc' . This can be accomplished by summing over all possible locations:

$$P(loc|odo) = \sum_{loc'} P(loc' -> loc)P(odo|loc' -> loc) \tag{9.2.4}$$

In other words, the law of total probability requires us to consider all possible locations the robot could have ever been at. This step is known as *Action Update*. In practice we don't need to calculate this for all possible locations, but only those that are technically feasible given the maximum speed of the robot. We note also that the sum notation technically corresponds to a convolution (Section C.3) of the probability distribution of the robot's location in the environment with the robot's odometry error probability distribution.

9.2.3. Summary and Examples

We have now learned two methods to update the belief distribution of where the robot could be in the environment. First, a robot can use external landmarks to update its position. This is known as *perception update* and relies on exterioception. Second, a robot can observe its internal sensors. This is known as action update and relies on proprioception. The combination of action and perception updates is known as *Markov Localization*. You can think about the action update to increase the uncertainty of the robot's position and the perception update to shrink it. (You can also think about the action update as a discrete version of the error propagation model.) Also here we are using the robotics kinematic model and the noise model of your odometer to calculate P(odo|loc'− > loc).

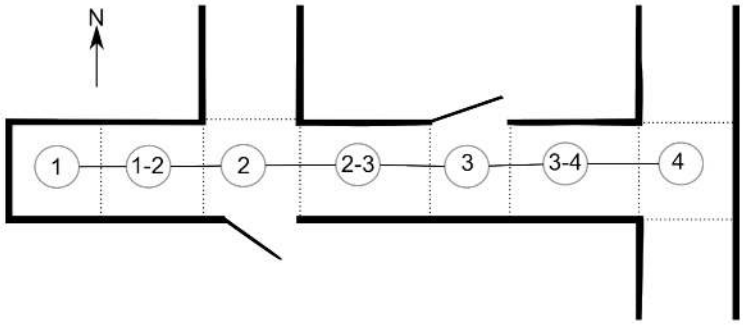


Figure **9.2.2**: *An office environment consisting of two rooms connected by a hallway. A topological map is super-imposed.*

Example 9.2.1: Topological Map

This example describes one of the first successful real robot systems that employed Markov Localization in an office environment. The experiment is described in more detail in a 1995 article of AI Magazine(?). The office environment consisted of two rooms and a corridor that can be modeled by a topological map (Figure 9.2.2). In a topological map, areas that the robot can be in are modeled as vertices, and navigable connections between them are modeled as edges of a graph. The location of the robot can now be represented as a probability distribution over the vertices of this graph.

The robot has the following sensing abilities:

- It can detect a closed door to its left or right.
- It can detect an open door to its left or right.
- It can detect whether it is an open hallway.

Table	Wall	Closed Dr	Open Dr	Open Hwy	Foyer
Nothing Detected	70%	40%	5%	0.1%	30%
Closed Door Detected	30%	60%	0%	0%	5%
Open Door Detected	0%	0%	90%	10%	15%
Open Hallway Detected	0%	0%	0.1%	90%	50%

Table **9.2.1**: Conditional probabilities of the Dervish robot detecting certain features in the Stanford laboratory.

Unfortunately, the robot's sensors are not at all reliable. The researchers have experimentally found the probabilities to obtain a certain sensor response for specific physical positions using their robot in their environment. These values are provided in Table 9.2.1.

For example, the success rate to detect a closed door is only 60%, whereas a foyer looks like an open door in 15% of the trials. This data corresponds to the conditional probability to detect a certain feature given a certain location.

Consider now the following initial belief state distribution: $p(1-2) = 0.8$ and $p(2-3) = 0.2$. Here, $1-2$ etc. refers to the position on the topological map in Figure 9.2.2. You know that the robot faces east with certainty. The robot now drives for a while until it reports "open hallway on its left and open door on its right". This actually corresponds to location 2, but the robot can in fact be anywhere. For example there is a 10% chance that the open door is in fact an open hallway, i.e. the robot is really at position 4. How can we calculate the new probability distribution of the robot's location? Here are the possible trajectories that could happen:

The robot could move from 2−3 to 3, 3−4 and finally 4. We have chosen this sequence as the probability to detect an open door on its right is zero for 3 and $3-4$, which leaves position 4 as the only option if the robot has started at $2-3$. In order for this

hypothesis to be true, the following events need to have happened, their probabilities are given in parentheses:

1. The robot must have started at 2 − 3 (20%)
2. Not have seen the open door at the left of 3 (5%) and not have seen the wall at the right (70%)
3. Not have seen the wall to its left (70%) and not have seen the wall to its right at node 3 − 4 (70%)
4. Correctly identify the open hallway to its left (90%) and mistake the open hallway to its right for an open door (10%)

Together, the likelihood that the robot got from position 2 − 3 to position 4 is therefore given by 0.2 × 0.05 × 0.7 × 0.7 × 0.7 × 0.9 × 0.1 = 0.03%, that is very unlikely.

The robot could also move from 1 − 2 to 2, 2 − 3, 3, 3 − 4 or 4. We can evaluate these hypotheses in a similar way:

- The chance that it correctly detects the open hallway and door at position 2 is 0.9 × 0.9, so the chance to be at position 2, having started at 1−2, is 0.8×0.9×0.9 = 64%.
- The robot cannot have ended up at position 2 − 3, 3, and 3 − 4 because the chance of seeing an open door instead of a wall on the right side is zero in all these cases.
- In order to reach position 4, the robot must have started at 1−2 has a chance of 0.8. The robot must not have seen the hallway on its left and the open door to its right when passing position 2. The probability for this is 0.001×0.05. The robot must then have detected nothing at 2−3 (0.7× 0.7), nothing at 3 (0.05×0.7), nothing at 3−4 (0.7×0.7), and finally mistaken the hallway on its right for an open door at position 4 (0.9 × 0.1). Multiplied together, this outcome is very unlikely.

Given this information, we can now calculate the posterior probability to be at a certain location on the topological map by adding up the probabilities for every possible path to get there.

Example 9.2.2: Grid-Based Markov Localization

Instead of using a coarse topological map, we can also model the environment as a fine-grained grid. Each cell is marked with a probability corresponding to the likelihood of the robot being at this exact location (Figure 9.2.3). We assume that the robot is able to detect walls with some certainty. The images in the right column show the actual location of the robot. Initially, the robot does not see a wall and therefore could be almost anywhere. The robot now moves northwards. The action update now propagates the probability of the robot being somewhere upwards. As soon as the robot encounters the wall, the perception update bumps up the likelihood to be anywhere near a wall. As there is some uncertainty associated with the wall detector, the robot cannot only be directly at the wall, but anywhere — with decreasing probability — close by. As the action update involved continuous motion to the north, the likelihood to be close to the south wall is almost zero. The robot then performs a right turn and travels along the wall in clockwise direction. As soon as it hits the east wall, it is almost certain about its position, which then again decreases.

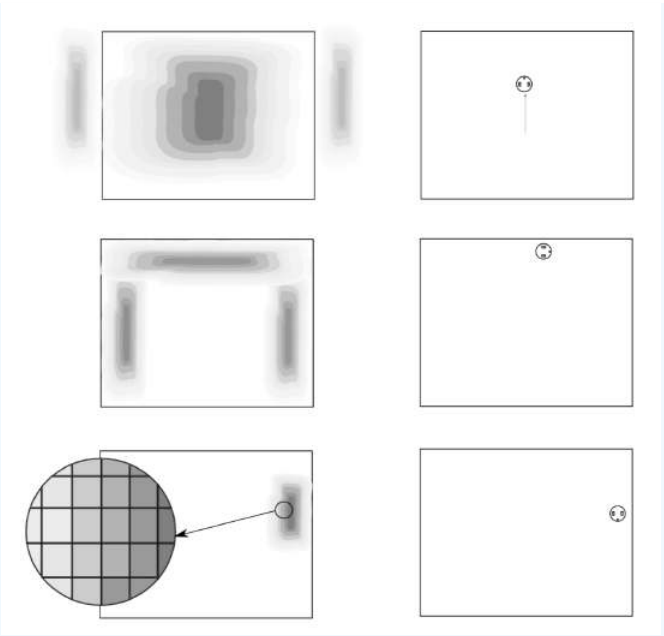


Figure **9.2.3**: Markov localization on a grid. The left column shows the likelihood to be in a specific cell as grey value (dark colors correspond to high likelihoods). The right column shows the actual robot location. Arrows indicate previous motion. Initially, the position of the robot is unknown, but recorded upwards motion makes positions at the top of the map more likely. After the robot has encountered a wall, positions away from walls become unlikely. After rightwards and down motions, the possible positions have shrunk to a small area.

9.3: Particle Filter

Although grid-based Markov Localization can provide compelling results, it can be computationally very expensive, in particular when the environment is large and the resolution of the grid is small. This is in part due to the fact that we need to carry the probability to be at a certain location forward for every cell on the grid, regardless of how small this probability is. An elegant solution to this problem is the particle filter. It works as follows:

1. Represent the robot's position by N particles that are randomly distributed around its estimated initial position. For this, we can either use one or more Gaussian distributions around the initial estimate(s) of where the robot is, or choose an uniform distribution (Figure 9.4.1).
2. Every time the robot moves, we will move each particle in the exact same way, but add noise to each movement much like we would observe on the real robot. Without a perception update, the particles will spread apart farther and farther.
3. Upon a perception event, we evaluate every single particle using our sensor model. What would the likelihood be to have a perception event such as we observed at this location? We can then use Bayes' rule to update each particle's position.
4. Once in a while or during perception events that render certain particles infeasible, particles that have a too low probability can be deleted, while those with the highest probability can be replicated.

9.4: The Kalman Filter

The location of a robot is subject to uncertainty due to wheelslip and encoder noise. We learned in the past how the variance in position can be derived from the variance of the robot's drivetrain using the error propagation law and the forward kinematics of the robot. One can see that this error is continuously increasing unless the robot has additional observations, e.g., of a static object with known location. This update can be formally done using Bayes' rule, which relates the likelihood to be at a certain position given that the robot sees a certain feature to the likelihood to see this feature at the hypothetical location. For example, a robot that drives towards a wall will become less and less certain of its position (action update) until it encounters the wall (perception update). It can then use its sensor model that relates its observation with possible positions. Its real location must be therefore somewhere between its original belief and where the sensor tells it to be. Bayes' rule allows us to perform this location for discrete locations and discrete sensor error distributions. This is inconvenient as we are used to represent our robot's position with a 2D Gaussian distribution. Also, it seems much easier to just change the mean and variances of this Gaussian instead of updating hundreds of variables. The goals of this section are

- to introduce a technique known as the Kalman filter to perform action and perception updates exclusively using Gaussian distributions.
- to formally introduce the notion of a feature map.
- to develop an example that puts everything we learned so far together: forward kinematics, error propagation and feature estimation.

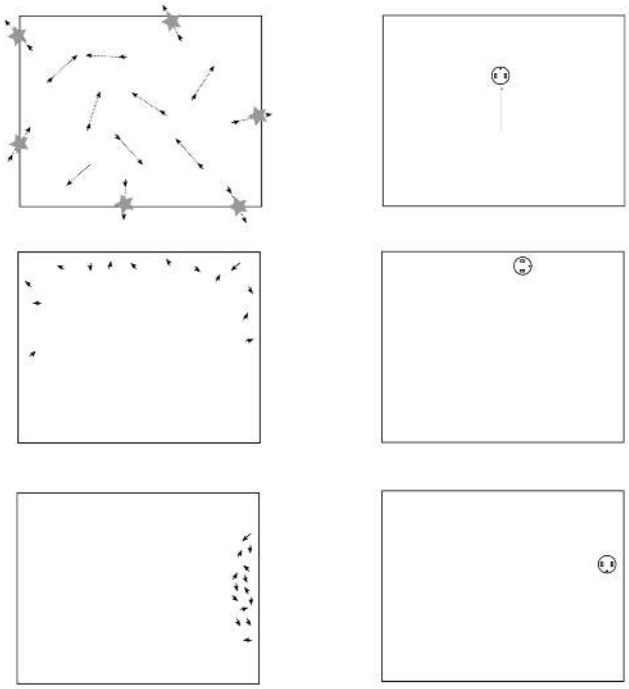


Figure **9.4.1**: *Particle filter example. Possible positions and orientations of the robot are initially uniformly distributed. Particles move based on the robot's motion model. Particles that would require the robot to move through a wall in absence of a wall perception event are deleted (stars). After a perception event, particles too far of a wall become unlikely and their positions are resampled in the vicinity of a wall. Eventually, the particle filter converges.*

9.4.1. Probabilistic Map-based localization

In order to localize a robot using a map, we need to perform the following steps

1. Calculate an estimate of our new position using the forward kinematics and knowledge of the wheel-speeds that we sent to the robot until the robot encounters some uniquely identifiable feature.
2. Calculate the relative position of the feature (a wall, a landmark or beacon) to the robot.
3. Use knowledge of where the feature is located in global coordinates to predict what the robot should see.
4. Calculate the difference between what the robot actually sees and what it believes it should see.

5. Use the result from (4) to update its belief by weighing each observation with its variance.

Steps 1-2 are based on the sections on “Forward Kinematics” and “Line detection”. Step 3 uses again simple forward kinematics to calculate the position of a feature stored in global coordinates in a map in robot coordinates. Step 4 is a simple subtraction of what the sensor sees and what the map says. Step 5 introduces the Kalman filter. Its derivation is involved, but its intuition is simple: why just averaging between where I think I am and what my sensors tell me, if my sensors are much more reliable and should carry much higher weight?

9.4.2. Optimal Sensor Fusion

The Kalman filter is an optimal way to fuse observations that follow a Gaussian distribution. The Kalman filter has an update and a prediction step. The update step uses a dynamical model of the system (such as the forward kinematics of your robot) and the prediction step uses a sensor model (such as the error distribution calibrated from its sensors). The Kalman filter does not only update the state of the system (the robot’s position) but also its variance. For this, it requires knowledge of all the variances involved in the system (e.g., wheel-slip and sensor error) and uses them to weigh each measurement accordingly. Before providing the equations for the Kalman filter, we will make use of a simple example that explains what “optimal” means in this context.

Let $\hat{q}_1$ and $\hat{q}_2$ be two different estimates of a random variable and ${\sigma^2}_1$ and ${\sigma^2}_2$ their variances, respectively. Let q be the true value. This could be the robot’s position, e.g. The observations have different variances when they are obtained by different means, say using odometry for $\hat{q}_1$ and by using the location of a known feature for $\hat{q}_2$. We can now define the weighted mean-square error

$$S = \sum_{i=1}^{n} \frac{1}{\sigma_i}(q - \hat{q_i})^2 \tag{9.4.1}$$

that is, S is the sum of the errors of each observation $\hat{q}_i$ weighted by its standard deviation σ_i . Each error is weighted with its standard deviation to put more emphasis on observations whose standard deviation is low. Minimizing S for n = 2 yields the following optimal expression for q:

$$q = \frac{\hat{q}_1 \sigma_2^2}{\sigma_1^2 + \sigma_2^2} + \frac{\hat{q}_2 \sigma_1^2}{\sigma_1^2 + \sigma_2^2} \tag{9.4.2}$$

or, equivalently,

$$q = \hat{q}_1 + \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2}(\hat{q}_2 - \hat{q}_1) \tag{9.4.3}$$

We have now derived an expression for fusing two observations with different errors that provably minimizes the error between our estimate and the real value. As q is a linear combination of two random variables (Section C.4, the new variance is given by

$$\sigma^2 = \frac{1}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}} \tag{9.4.4}$$

Interestingly, the resulting variance is smaller than both σ_1 and σ_2, that is, adding additional observation always helps reducing accuracy instead of introducing more uncertainty.

9.4.3. Integrating Prediction and Update: The Kalman Filter

Although we have introduced the problem above as fusing two observations of the same quantity and weighting them by their variance, we can also interpret the equation above as an update step that calculates a new estimate of an observation based on its old estimate and a measurement. Remember step (4) from above: $\hat{q}2 - \hat{q}1$ is nothing else then the difference between what the robot actually sees and what it thinks it should see. This term is known as innovation in Kalman lingo. We can now rewrite (9.4.3) from above into

$$\hat{x}_{k+1} = \hat{x}_k + K_{k+1} y_{k+1} \tag{9.4.5}$$

Here, $\hat{x}_k$ is the state we are interested in at time k, $K_{k+1} = {\sigma^2}_1 /({\sigma^2}_1+{\sigma^2}_2)$ the Kalman gain, and $y_{k+1} = \hat{q}_2 - \hat{q}_1$ the innovation. Unfortunately, there are few systems that allow us to directly measure the information we are interested in. Rather, we obtain a sensor measurement z_k that we need to convert into our state somehow. You can think about this the other way too and predict your measurement z_k from your state x_k. This is done using the observation model H_k, so that

$$y_k = z_k - H_k x_k \tag{9.4.6}$$

In our example H_k was just the identity matrix; in a robot position estimation problem H_k is a function that would predict how a robot would see a certain feature. As you can see, all the weighting based on variances is done in the Kalman gain K. The perception update step shown above, also known as prediction step is only half of what the Kalman filter does. The first step is the update step, which corresponds to the action update we already know. In fact, the variance update in the Kalman filter is exactly the same as we learned during error propagation. Before going into any more details on the Kalman filter, it is time for a brief disclaimer: the Kalman filter only works for linear systems. Forward kinematics of even the simplest robots are mostly non-linear, and so are observation models that relate sensor observations and the robot position. Non-linear systems can be dealt with the Extended Kalman Filter.

9.5: Extended Kalman Filter

In the extended Kalman filter, the state transition and observation models do not need to be linear functions of the state but may instead be differentiable functions. The action update step looks as follows:

$$\hat{\mathbf{x}}_{k'|k-1} = f(\hat{\mathbf{x}}_{k-1|k-1}, \mathbf{u}_{k-}) \tag{9.5.1}$$

Here f() is a function of the old state $\mathbf{x}_{k-1}$ and control input $\mathbf{u}_{k-1}$. This is nothing else as the odometry update we are used to, where f() is a function describing the forward kinematics of the robot, $\mathbf{x}_k$ its position and $\mathbf{u}_k$ the wheel-speed we set.

We can also calculate the covariance matrix of the robot position

$$\mathbf{P}_{k'|k-1} = \nabla_{x,y,\theta} f \mathbf{P}_{k-1|k-1} \nabla_{x,y,\theta} f^T + \nabla_{\Delta_{r,l}} f \mathbf{Q}_{k-1} \nabla_{\Delta_{r,l}} f \tag{9.5.2}$$

This is nothing else as the error propagation law applied to the odometry of the robot with $\mathbf{Q_k}$ the covariance matrix of the wheel-slip and the Jacobian matrices of the forward kinematic equations f() with respect to the robot's position (indicated by the index x, y, θ) and with respect to the wheel-slip of the left and right wheel.

The perception update (or prediction) step looks as follows:

$$\hat{\boldsymbol{x}}_{k|k'} = \hat{\boldsymbol{x}}_{k'|k-1} + \boldsymbol{K}_{k'} \boldsymbol{y}_{k'} \tag{9.5.3}$$

$$\boldsymbol{P}_{k|k'} = (\boldsymbol{I} - \boldsymbol{K}_{k'} \boldsymbol{H}_{k'}) \boldsymbol{P}_{k'|k-1} \tag{9.5.4}$$

At this point the indices k should start making sense. We are calculating everything twice: once we update from k − 1 to an intermediate result k' during the action update, and obtain the final result after the perception update where we go from k' to k.

We need to calculate three additional variables:

1. The innovation $\tilde{\mathbf{y}}_k = \mathbf{z}_k - h(\hat{\mathbf{X}}_{k|k-1})$
2. The covariance of the innovation $\mathbf{S}_k = \mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}^T_k + \mathbf{R}_k$
3. The (near-optimal) Kalman gain $\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}^T_k \mathbf{S}^{-1}_k$

Here h() is the observation model and **H** its Jacobian. How these equations are derived is involved (and is one of the fundamental results in control theory), but the idea is the same as introduced above: we wish to minimize the error of the prediction.

9.5.1. Odometry using the Kalman Filter

We will show how a mobile robot equipped with a laser scanner can correct its position estimate by relying on unreliable odometry, unreliable sensing, but a correct map, in an optimal way. Whereas the update step is equivalent to forward kinematics and error propagation we have seen before, the observation model and the "innovation" require additional steps to perform odometry.

1. Prediction Update: We assume for now that the reader is familiar with calculating $\hat{\mathbf{X}}_{k'|k-1} = f(x, y, \theta)^T$ and its variance $\mathbf{P}_{k'|k-1}$. Here, $\mathbf{Q}_{k-1}$, the covariance matrix of the wheel-slip error, is given by

$$Q_{k-1} = \begin{bmatrix} k_r|\Delta s_r & 0 \\ 0 & k_l|\Delta s_l \end{bmatrix} \tag{9.5.5}$$

where Δs is the wheel movement of the left and right wheel and k are constants. See also the odometry lab for detailed derivations of these calculations and how to estimate k_r and k_l. The state vector $\hat{\mathbf{X}}_{k'|k-1}$ is a 3x1 vector, the covariance matrix $\mathbf{P}_{k'|k-1}$ is a 3x3 matrix, and $\nabla_{\Delta r,l}$ that is used during error propagation is a 3x2 matrix. See the error propagation section for details on how to calculate $\nabla_{\Delta r,l}$.

2. Observation: Let us now assume that we can detect line features $z_{k,i} = (\alpha_i, r_i)^T$, where α and r are the angle and distance of the line from the coordinate system of the robot. These line features are subject to variances $\sigma_{\alpha,i}$ and $\sigma_{r,i}$, which make up the diagonal of $\mathbf{R_k}$. See the line detection section for a derivation of how angle and distance as well as their variance can be calculated from a laser scanner. The observation is a 2x1 matrix.

3. Measurement Prediction: We assume that we can uniquely identify the lines we are seeing and retrieve their real position from a map. This is much easier for unique features, but can be done also for lines by assuming that our error is small enough and we therefore can search through our map and pick the closest lines. As features are stored in global coordinates, we need to transpose

them into how the robot would see them. In practice this is nothing but a list of lines, each with an angle and a distance, but this time with respect to the origin of the global coordinate system. Transposing them into robot coordinates is straightforward. With $\hat{\mathbf{x}}_k = (x_k, y_k, \theta_k)^T$ and $m_i = (\alpha_i , r_i)$ the corresponding entry from the map, we can write

$$h(\hat{x}_{k|k-1}) = \begin{bmatrix} \alpha_{k,i} \\ r_{k,i} \end{bmatrix} = h(x, m_i) = \begin{bmatrix} \alpha_i - \theta \\ r_i - (xcos(\alpha_i) + ysin(\alpha_i)) \end{bmatrix} \tag{9.5.6}$$

and calculate its Jacobian $\mathbf{H}_k$ as the partial derivatives of α to x, y, θ in the first row, and the partial derivatives of r in the second. How to calculate h() to predict the radius at which the robot should see the feature with radius ri from the map is illustrated in the figure below.

- Example on how to predict the distance to a feature the robot would see given its estimated position and its known location from a map.

4. Matching: We are now equipped with a measurement z_k and a prediction $h(\hat{\mathbf{x}}_{k|k-1})$ based on all features stored in our map. We can now calculate the innovation

$$y_k = z_k - h(\hat{x}_{k|k-1}) \tag{9.5.7}$$

5. Estimation: We now have all the ingredients to perform the perception update step of the Kalman filter:

$$\hat{x}_{k'|k-1} = \hat{x}_{k'|k-1} + K_{k'} y_{k'} \tag{9.5.8}$$

$$P_{k|k'} = (I - K_{k'} H_{k'}) P_{k'|k-1} \tag{9.5.9}$$

It will provide us with an update of our position that fuses our odometry input and information that we can extract from features in the environment in a way that takes into account their variances. That is, if the variance of your previous position is high (because you have no idea where you are), but the variance of your measurement is low (maybe from a GPS or a symbol on the Ratslife wall), the Kalman filter will put more emphasis on your sensor. If your sensors are poor (maybe because you cannot tell different lines/walls apart), more emphasis will be on the odometry.

As the state vector is a 3x1 vector and the innovation a 2x1 matrix, the Kalman gain must be a 3x2 matrix. This can also be seen when looking at the covariance matrix that must come out as a 3x3 matrix, and knowing that the Jacobian of the observation function is a 2x3 matrix. We can now calculate the covariance of the innovation and the Kalman gain using

$$S_k = H_k P_{k|k-1} H_k^T + R_k \tag{9.5.10}$$

$$K_k = P_{k|k-1} H_k^T S_k^{-1} \tag{9.5.11}$$

9.6: Exercises

Take Home Lessons

- If the robot has no additional sensors and its odometry is noisy, error propagation will lead to ever increasing uncertainty of a robot's position regardless of using Markov localization or the Kalman filter.
- Once the robot is able to sense features with known locations, Bayes' rule can be used to update the posterior probability of a possible position. The key insight is that the conditional probability to be at a certain position given a certain observation can be inferred from the likelihood to actually make this observation given a certain position.
- A complete solution that performs this process for discrete locations is known as Markov Localization.
- The Extended Kalman Filter is the optimal way to fuse observations of different random variables that are Gaussian distributed. It is derived by minimizing the leastsquare error between prediction and real value.
- Possible random variables could be the estimate of your robot position from odometry and observations of static beacons with known location (but uncertain sensing) in the environment.
- In order to take advantage of the approach, you will need differentiable functions that relate measurements to state variables as well as an estimate of the covariance matrix of your sensors.
- An approximation that combines benefits of Markov Localization (multiple hypothesis) and the Kalman filter (continuous representation of position estimates) is the Particle filter.

Exercises

1. Assume that the ceiling is equipped with infra-red markers that the robot can identify with some certainty. Your task is to develop a probabilistic localization scheme, and you would like to calculate the probability *p(marker|reading)* to be close to a certain marker given a certain sensing reading and information about how the robot has moved.
    - Derive an expression for *p(marker|reading)* assuming that you have an estimate of the probability to correctly identify a marker *p(reading|marker)* and the probability *p(marker)* of being underneath a specific marker.
    - Now assume that the likelihood that you are reading a marker correctly is 90%, that you get a wrong reading is 10%, and that you do not see a marker when passing right underneath it is 50%. Consider a narrow corridor that is equipped with 4 markers. You know with certainty that you started from the entry closest to marker 1 and move right in a straight line. The first reading you get is "Marker 3". Calculate the probability to be indeed underneath marker 3.
    - Could the robot also possibly be underneath marker 4?

CHAPTER OVERVIEW

10: Grasping

Grasping is the activity in which the robot extends its body by attaching an external object to its kinematic chain. This allows the robot to move this object and potentially manipulate it, that is change its shape or pose. Grasping has the interesting, and very confusing, property that its relatively easy in practice, but very difficult in theory. Consequently, this chapter describes a variety of strategies that will lead to successful grasps for a wide range of objects, but has difficulties to answer questions such as *What makes a good grasp?* or *How to find good grasps?* in any more depth than by providing simple heuristics.

10.1: The Theory of Grasping

The theory of grasping is quite involved, with the state of the art comprehensively described in (Rimon & Burdick 2019), yet has difficulties to mathematically exactly capture the mechanics of grasping mechanisms that are successful in practice. Rather than describing these developments here — which will be well beyond the scope of this book — we will briefly describe different approaches to model grasping, and their limitations, to provide a better understanding of what the reasons for grasps that work are and what matters when designing a gripper.

In its most simple form, grasping requires immobilizing an object, at least agains the forces of gravity, by providing appropriate forces in the opposite direction, also known as constraints. Specifically, contact points on a robotic finger, gripper or hand are assumed to exert localized forces, thereby constraining the object sufficiently. By this, fingers act essentially as miniature robotic arms, allowing us to apply the methods and tools from previous chapters 2–??

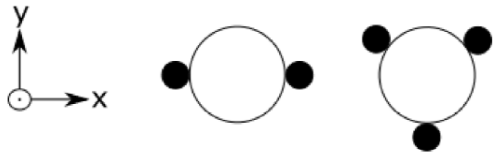


Figure **10.1.1**: *Cross-section from above showing an idealized twofinger (left) and three finger (right) gripper holding a cylinder.*

10.1.1. Friction

While already very involved for anything but very simple mechanisms, such a model only captures a very small slice of realistic grasps. In any real application, contacts between a gripper and hand are not friction-less. This is the reason a grasp such as shown in Figure 10.1 actually works. If there were really no friction between the fingers and the object, the object would be ejected from the hand for every grasp that is not exactly aligned with a principal axis of the cylinder in Figure 10.1, left. Furthermore, even the three-finger grasp shown in Figure 10.1, right, would always fail as there is no force constraining the object from below. Fortunately, the existance of friction makes grasping much easier in practice, yet much harder to describe mathematically.

The reason that the grasps shown in Figure 10.1.1 do work in most circumstances is that the normal forces shown have a tangential component that is due to friction and covered by *Coloumb's Friction law*, which states that the higher the friction coefficient of a material, the more normal force translates into tangential forces that can resist two surfaces from moving against each other:

It is governed by the equation:

$$F_t \leq \mu F_n \tag{10.1.1}$$

Here F_t is the force of friction exerted by each surface on the other and F_n is the normal force. The force F_t acts in tangential direction of the normal force applied by, e.g., a finger's tip, where μ is an empirical coefficient of friction.

The friction coefficient μ is low for glass on glass and high for rubber on wood. We are therefore interested in designing grippers with high friction coefficients to avoid objects from slipping.

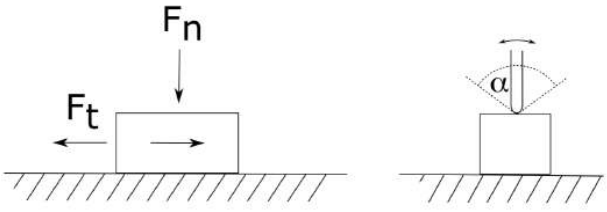


Figure **10.1.2**: Left: Coloumb friction relates normal to tangential reaction forces that are required to overcome friction, here shown for rightwards motion. Right: Friction cone for point forces. As long as the force is within the cone cone, the finger will not slip.

When do objects slip? Lets say we have a fingertip pressing down on a surface in any orientation. There will be a force normal to the surface F_n, which defines the tangential force F_t in any direction. Sweeping the tangential force around the normal force creates a cone with an opening angle of

$$\alpha = 2tan^{-1}\mu \tag{10.1.2}$$

see (Rimon & Burdick 2019, p. 57) for a derivation. If the net force on the object is not within this cone, the object slips. This becomes more intuitive when thinking about how different values of μ affect the shape of this cone. If μ is high, the cone will be relatively flat, letting the object accept forces from many different directions without slipping. If μ is low, the cone will be relatively narrow, requiring the force to be normal to the object's surface to prevent slippage.

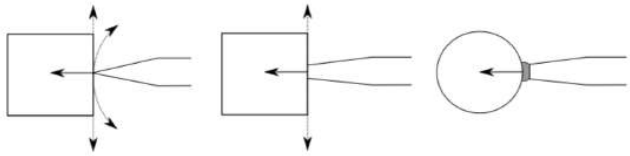


Figure **10.1.3**: From left to right: ideal force exerted via a single point of contact, forces exerted via an area of contact, contact area increasing due to pressure and conforming with the surface. Remaining degrees of freedom are indicated by dotted lines.

A force applied to a rigid body will exert both a force as well as a torque to the body's center of gravity. This is called a wrench. If we consider all the possible forces and torques that we can apply to a rigid body without having the end-effector slip to form a space (namely the cone described earlier for a single finger), we can talk about the grasping wrench space, which is the corresponding space of all suitable wrenches.

Knowing the relation between normal and tangential reaction forces can help in designing grippers that are more likely to successfully grasp an object than others, as well as when planning suitable grasp for objects with known friction.

10.1.2. Multiple Contacts and Deformation

In practice, no force will ever be applied at a single point only, but over an area, either due to the size of the finger pad itself or due to the contact area deforming. Even the smallest contact area that is not a point in the mathematical sense will add constraints on torque, thereby adding constraints in additional dimensions and therefore further stabilizing the grasp. This is illustrated in Figure 10.1.3. Whereas the object can easily pivot around the point of contact in Figure 10.1.3, left, increasing the area of contact constrains the rotational degree of freedom. It is therefore desirable to grasp an object with an as large contact area as possible. As surfaces are not ideally flat, in practice this is only possible when the contact area is deformable, Figure 10.1.3, right. A large contact area will also increase friction, which is usually desirable.

Indeed, using blank metal jaws or fingers is little successful in practice. Instead, rubber pads are used to increase force closure by conforming around the object. As the rubber is flexible, however, the grasp is not completely fixating the object, but it can move within the grasp, which might not be desirable when picking up a nut, e.g., and trying to mount it on a screw. Mathematically, this introduces additional complications into the grasp model, as flexible pads are the equivalent of a spring, increasing uncertainty and dynamics.

10.1.3. Suction

A highly capable method for grasping is using suction. Here, a suction cup is pressed against an object, using a vacuum applied by a pump to suck the object against the cup. Instead of exerting forces against the object, which always requires at least one antipodal force (or multiple forces that are distributed such that the object remains in equilibrium) to create a constraint, suction only requires one point of contact. The rim of the suction cup provides both friction and multiple contact points to prevent the object from slipping and further constraining the object beyond the normal force applied by the vacuum. Requiring only a single area of contact is a tremendous advantage from a planning perspective as only one area on an object needs to be identified, whereas other grasping approaches need to always identify two areas and coordinate motion to reach them. (Suction using multiple suction cups on custom-made rigs to grasp large car parts such as doors is very popular in the car industry, but relies on preprogrammed trajectories, which is not a focus of this book.)

The soft nature of the suction cup provides the ability for the rim to conform to the object to some extent, but makes suction impractical for objects that do not have any flat surfaces or holes, for example objects stored in a net. The elasticity of the rim also makes it difficult to further manipulate the object as all forces applied by the robot will need to be transferred via a spring-like elastic material. Finally, suction requires a vacuum pump that is able to generate sufficient force to lift an object, limiting the maximum weight of objects suitable for suction by a single suction cup in practice.

10.2: Simple Grasping Mechanisms

Understanding why grasping actually works, namely due to friction and increasing contact area due to deformation, allows us to select grasping mechanism that are both able to successfully grasp a wide range of objects, simple to construct, and easy to control. Here, properties of interest are the range of possible object sizes, given by a minimum and maximum size, the maximum weight of an object, and how fragile objects can possibly be. Here, object dimensions are directly dependent on the gripper kinematics, such as minimum and maximum aperture, whereas the maximum weight is given by the torque the mechanism can exert as well as the number of contacts and their friction parameters. Whether a gripper can handle fragile objects, is a function of how well this torque can be measured and controlled.

10.2.1. 1-DoF Scissor-Like Gripper

One of the simplest grippers is a simple one degree-of-freedom claw, which is a popular design in the prosthetic community, and has been refined for centuries. Actuated by a string mounted to a person's shoulder, or more recently by electric motors controlled by measuring muscle activity in the lower arm, this simple mechanism enables their wearers to perform a wide range of everyday activities. Indeed, an off-the-shelf prosthetic hand has been shown to perform a large variety of grasping and manipulation tasks when compared with other robotic hands in a tele-operation scenario, only limited by its ability to conform to specific kinematic constraints such as operating scissors (Patel, Segil & Correll 2016).

A simple design is shown in Figure 10.2.1 and consists of an active finger that presses an object against a passive finger, with both fingers often shaped as a hook. As should be clear by now, such a design can only work by relying on friction, which makes it not very common in traditional robotics.

The key advantage of this mechanism is the very simple control strategy that it enables: use the passive finger to make contact with the object, then use the active finger to close the grasp. The event "make contact" can either be detected by measuring the force at the wrist and looking for abrupt changes or using a tactile sensor on the surface with which contact is made. This approach can therefore lead to robust grasps with a minimum of sensing. A disadvantage of this mechanism is that its function relies exclusively on friction, possibly ejecting objects from its grasp if friction is not sufficient or the object is in an otherwise suboptimal conformation. Unlike most other mechanisms, it is also impossible to use the finger position to infer the width of an object, which is illustrated by the illustrations in Figure 10.2.1, center.

The mechanism shown in Figure 10.2.1 can be actuated in many different ways, for example by attaching a servo motor directly to the active finger, using a shape-memory alloy wire via a suitable lever arm, or a pneumatic piston or balloon.

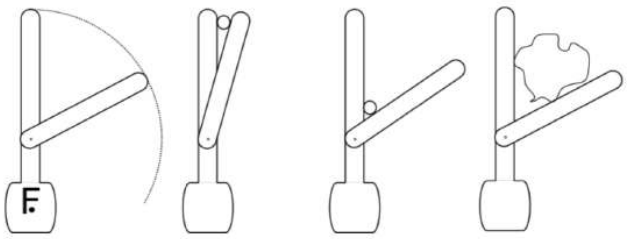


Figure **10.2.1**: Simple 1-DoF grasping mechanism that relies on friction to grasp objects with a wide variety of sizes (center, right). The mechanism has only one moving part that presses the object against a passive finger.

10.2.2. Parallel Jaw

The most common industrial mechanism is the two-finger parallel jaw gripper. It operates by squeezing an object between its two parallel jaws, which are usually driven by a single actuator and therefore move in concert. Parallel jaw grippers usually yield more contact area than a scissor-like 1-DoF gripper, but suffer from a smaller range of motion.

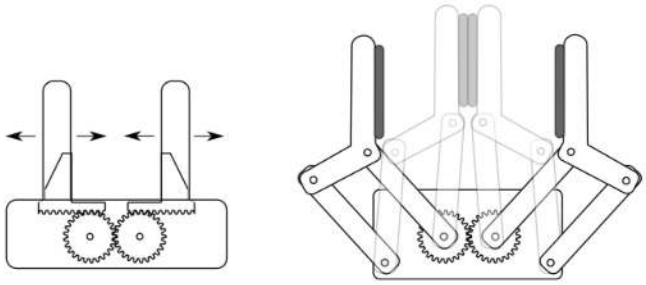


Figure 10.2.2: Left: Parallel jaw gripper driven by a single actuator via a system of coupled gears.Right: 4-bar linkage parallel jaw gripper.

Figure 10.2.2, left, shows a minimalist implementation of a parallel jaw gripper that can be actuated by a single servo motor, driving two rack gears to which the gripper jaws are mounted. While using gears on racks is unusual in an industrial design — the gripper jaws typically travel on threads actuated by worm gears or are attached to a pneumatic piston — this drawing illustrates the relationship between the range of motion of the gripper jaws, the length of the mechanism it is sliding on, here a rack gear, and the resulting body size. In order for this design to fully close, the two rack gears must be mounted at an offset in order to slide against each other. Constraints like this often make the gripper body twice as wide as the maximum aperture, making it difficult for the robot to enter tight areas. The mechanical design also affects the speed at which a gripper can operate. Pneumatic grippers, where air pressure coming in on either end of the piston can drive the gripper into an “open” or “close” position very quickly (2-3 times per second), but cannot be controlled accurately. Electric mechanisms instead trade accuracy and torque with speed.

The control strategy for parallel jaw grippers requires an accurate pose estimate of the object of interest and positioning the gripper so that the object is right in the center of the two jaws. Note that force-closure with a static object, such as a screw mounted to a structure, requires both jaws to make contact with the object at the same time, thereby imposing high accuracy requirements of both object detection and robot motion. Here, compliance can help, allowing the gripper to adjust its pose to the object. This can be accomplished by either measuring forces in the wrist and moving the gripper to minimize lateral forces or a compliant mounting mechanism or structure, such as a robot equipped with series-elastic or pneumatic actuators. An alternative approach is to actuate both gripper jaws independently.

10.2.3. 4-Bar Linkage Parallel Gripper

A parallel jaw mechanism with a larger range of motion can be accomplished using two 4-bar linkages, Figure 10.2.2, right. In a 4-bar linkage, rotation is translated into straight translation. This is accomplished by two pairs of parallel bars of equal lengths. In Figure 10.2.2, right, one of the four bars is not moving and substituted by the gripper body, to which two of the bars are mounted. Interestingly, both pairs remain parallel as one of the bars is rotating, resulting in the two gripper jaws remaining parallel to each other. This is best understood by inspecting Figure 10.2.2 and comparing the two positions the left jaw can be in.

The drawback of this design is that closing the gripper also results in a forward motion. This requires approaching an object from different heights, depending on its width. Other than this, the control strategy is the same as for the parallel jaw gripper, requiring an accurate estimate of the object’s pose. Also here, adding compliance or independent actuation of each jaw can help resolving accuracy problems.

10.2.4. Multi-Fingered Hands

Grippers with more than two fingers/jaws are rarely used in industrial practice. One common use case is grasping cylindrical objects from above, for which three-fingered hands, such as indicated in Figure 10.1.1, right, are best suited. In most other cases, three fingers are not an advantage, and might even be a hindrance, however. For example, it is difficult to perform simple pinching grasps with three fingers. This has led to designs in which two of the fingers are reconfigurable from performing an inwards motion to behave identical to a parallel jaw gripper, while the third finger is stored in a safe position. In addition to mechanical complexity, such an approach requires also additional planning steps.

How many grasps are possible and how many possible grasps are needed to grasp every possible object remains a difficult theoretical problem (which is further complicated by the fact that successful grasping often happens at the boundary of what is mathematically tractable). Generally, we can say however, that additional fingers — such as in the human hand — provide

additional redundancy, which allows grasping and manipulating (see Section 10.4) the same object in many different ways, including manipulating the object within the hand, that is without intermettent placement or handing it over to another gripper.

10.3: How to Find Good Grasps?

Finding a good grasp that fully constrains an object against all possible external forces and torques, that is a grasp that lies in the "grasping wrench space" (Section 10.1.1 is often too restrictive. For example, it might be sufficient to find a grasp that constrains an object simply against gravity. Other applications instead might require the grasp to constrain an object's movement also again lateral forces that happen due to acceleration. In practice, these considerations usually lead to simple application-specific heuristics. For example, in a warehouse picking tasks (Correll, Bekris, Berenson, Brock, Causo, Hauser, Okada, Rodriguez, Romano & Wurman 2016), the problem can be constrained to have the robot grasp only objects that are suitable to be retrieved with a simple suction cup. Finding a good grasp is then reduced to finding a flat surface close to the object's perceived center of gravity. When considering household tasks, such has handling and placing dishes, using silver ware to scoop food, or holding a pitcher, we are often interested in very specific grasps that support the intended manipulation (Section 10.4) that follows.

Theoretically speaking, grasps such as picking up an object or opening a door by turning its knob are task-specific wrench spaces. We can then say that the grasp is "good", when the task wrench space is a subset of the grasping wrench space, and will fail otherwise. We can also look at the ratio of forces actually applied to the object and the minimum needed to perform a desired wrench. If this ratio is high, for example, when the robot grasps an object far from its center of gravity or has to squeeze an object heavily to prevent it from slipping, this grasp is not as good as one, where the ratio is low and all of the force the robot is exerting is actually going into the desired wrench. It is usually not possible to find close-form expressions for the grasping wrench space. Instead, one can sample the space of suitable force vectors, e.g., by picking a couple of forces that are on the boundary of the cone's base, and calculate the convex hull over the resulting wrenches.

Finding Good Grasps for Simple Grippers

Finding good grasps for simple grippers, which have only one or at most two degrees of freedom, reduces the problem to finding geometries on the object that are suitable to place the gripper jaws, that is two parallel faces that are reasonably flat and at a distance that is below the gripper's maximum opening aperture.

In practice, an object might be perceived by a 3D perception device such as a stereo camera or a laser scanner, which would provide only one perspective of an object. A typical grasping pipeline using such a device is shown in Figure ??.

A typical algorithm proceeds as follows:

1. Acquisition: Obtain a "point cloud" or "depth image" of the objects of interest (Figure 10.3.1, b).
2. Pre-processing: Remove table plane or other points that are either too close or too far from the sensor (Figure 10.3.1, c).
3. Segmentation: Cluster points that are close enough, e.g., to identify individual objects (Figure 10.3.1, d).
4. Filtering: Filter clusters by size, geometry or other features, to down-select objects of interest (Figure 10.3.1, e).
5. Planning: Compute center-of-mass and principal axes of relevant clusters (Figure 10.3.1, f).
6. Collision-checking: Generate possible grasps and check for collisions with point clouds ((Figure 10.3.1, g).
7. Execution: Physically test a grasp by monitoring jaw distances, as well as forces and torques at the wrist ((Figure 10.3.1, h).

Some of these steps might not be necessary for all grasps, and some of them might have arbitrary complexity. For example, pre-processing is often used to remove known quanitites such as a table surface, from the data, but might be non-trivial when removing the edges of a bin, e.g.

Segmentation is the most critical step and requires some previous knowledge about the objects to grasp such as their size or the geometry of features thereon. In Figure 10.6, clustering points based on their distance is sufficient, e.g. using the DBSCAN algorithm (Ester, Kriegel, Sander, Xu et al. 1996), but requires an assumption about object size in order to select a suitable threshold. Other segmentation algorithms might use surface normals, or a combination of point cloud and image data such as color or patterns.

Filtering the resulting clusters to identify objects of interest can be as simple as rejecting those that are too small (as shown in Figure 10.6, e), but might also involve matching the points to a 3D model of a desired object or involving image data.

A simple approach to plan for possible grasps is to calculate the center-of-mass as well as the principal axes of an object using principal component analysis (Appendix B.5). Other approaches might require matching the existing points to a 3D model of the object to identify specific grasp points (such as the handle of a cup) or again rely on image features to do so.

After planning all, or some, possible grasps, grasps need to be checked for feasibility. While a collision with a point in the point-cloud might rule out a grasp, local search is sometimes being used to find a collision-free variant, for example by moving the gripper up and down as well as along the principal axes. In other applications, for example bin picking, some collisions might be ignored with the expectation that the gripper will push other objects out of its way.

Even though a grasp might look robust in a point cloud representation, it might not be effective when physically executing it. Possible failures are collisions with objects, insufficient friction with the object, or an object moving before the gripper is fully closed. For this reason, it is important to already close the gripper as much as possible before approaching the object, increasing the requirements for accurate perception.

With the ability to train neural networks to approximate complex functions, it is also possible to replace parts, or all of, the algorithmic steps shown in Figure 10.6 using a convolutional neural network trained by deep learning. While data intensive, such an approach can seamlessly merge image and depth data and adapt to application-specific data better than a hand-coded algorithm can.

Finding Good Grasps for Multi-Fingered Hands

The simple grasping pipeline described above is computationally expensive as there exist usually many possible grasp candidates, and each of them need to be checked for collisions. This problem explodes when considering grippers with articulated fingers. This can be overcome by considering only a predefined set of grasps such as two and three finger pinches for small objects and full-hand encompassing grasps for larger objects, e.g.

A suitable method to search the full space of possible grasps with an articulated hand is to use random sampling, that is bringing the end-effector to random positions, close its fingers around the object, and see what happens when generating wrenches that fulfill the task's requirements. To "see what happens", requires collision checking and dynamic simulation. Dynamic simulation applies Newtonian mechanics to an object (i.e., forces lead to acceleration of a body) and moves the object at very small time-steps. While this can be done using the connected components identified in the point cloud alone and assuming reasonable parameters for friction and contact points, point cloud data can also be augmented by object models to simulate whether a grasp has a high likelihood to be successful.

10.4: Manipulation

While grasping is only concerned with attaching an object to a robot's kinematic chain, it is often the case that such an object is not just placed or dropped, but that the intention of the grasping action was to change the pose of this object in some meaningful way. For example, cutlery and dishes on a table need to be in well-specified areas and aligned with each other, merchandise needs to be neatly stacked in a shelf, and machine parts need to be assembled with each other. These activities are known as manipulation. Discussing all the possible ways objects might be manipulated, for example inserted, screwed-in, turned, twisted, flipped, etc., and the many different contexts such actions would be required — which might dramatically change the approach a robot would need to chose — are well beyond the scope of this book.

Yet, many manipulation problems can be cast into a sequence of grasping and placing problems in which the possible grasp choices are appropriately constrained. For example, an object can be turned or flipped by planning a sequence of pick-andplace movements that each turn the object by a certain degree. Similarly, using two robotic arms, with one grasping an object out of the hand of the other, will allow a robot system to change an object's pose almost arbitrarly. (Which poses an object will be able to reach will depend on the object's exact geometry, the kinematics of the robotic arms, and constraints in the workspace.) So-called in-hand manipulation is still an active area of research as repeatedly picking and placing an object and hand-overs between different arms is considered to be too slow and otherwise impractical for many application areas.

10.5: Exercises

Exercises

1. Think about at least three mechanisms to realize a parallel jaw gripper. How does the minimum and maximum aperture of the gripper relate to the gripper width for each of these designs?

2. Think about at least three mechanisms to actuate a fourbar linkage. Which of these will keep the payload inside the gripper during power failure?

3. Derive an equation for the distance of the fingertip from the gripper base in a 4-bar linkage gripper as a function of the gripper opening width. Use appropriate parameters for all unknown parameters.

4. Write code to generate rectangles with random dimensions and orientations. Rectangles can overlap. Use a point-in-polygon test to simulate random point samples on their surface, simulating a top-down view with a depth sensor.

- Implement a segmentation routine that clusters objects based on a minimum distance.
- Implement a filter that rejects connected components based on size. For which kind of objects does this work well and where does this method fail?
- Implement a filter that rejects connected components that do not have rectangular shape. Are you able to specify a filter that works independent of the object size?
- Apply principal component analysis to compute the principal axes of the rectangle and compare with ground truth. How does the number of samples affect the accuracy of your estimate?

5. Use a function of the kind u(x − i) + rand(j) with u(x) the unit step function, rand() uniformly distributed random noise, and i, j suitable parameters to simulate a noisy depth-image a cube with width i. Use the nearest neighbor of each point to compute its normals and a suitable clustering algorithm to identify the cube. How do i and j affect the accuracy of your estimate?

CHAPTER OVERVIEW

11: Simultaneous Localization and Mapping

Robots are able to keep track of their position using a model of the noise arising in their drivetrain and their forward kinematics to propagate this error into a spatial probability density function (Section 8.2). The variance of this distribution can shrink as soon as the robot sees uniquely identifiable features with known locations. This can be done for discrete locations using Bayes' rule (Section 9.2) and for continuous distributions using the Extended Kalman Filter (Section 11.3). The key insight here was that every observation will reduce the variance of the robot's position estimate. Here, the Kalman filter performs an optimal fusion of two observations by weighting them with their variance, i.e., unreliable information counts less than reliable one. In the robot localization problem, one of the observations is typically the robot's position estimate whereas the other observation comes from a feature with known location on a map. So far, we have assumed that these locations are known. This chapter will introduce

- the concept of covariance (or, what all the non-diagonal elements in the covariance matrix are about),
- how to estimate the robot's location and that of features in the map at the same time (Simultaneous Localization and Mapping or SLAM)

11.1: Introduction

The SLAM problem has been considered as the holy grail of mobile robotics for a long time. This chapter will introduce one of the first comprehensive solutions to the problem, which has now be superseded by computationally more efficient versions. We will begin with studying a series of special cases.

11.1.1. Special Case I: Single Feature

Consider a map that has only a single feature. We assume that the robot is able to obtain the relative range and angle of this feature, each with a certain variance. An example of this and how to calculate the variance of an observation based on sensor uncertainty is described in the line fitting example (Section 8.2.1). This feature could be a wall, but also a graphical tag that the robot can uniquely identify. The position of this measurement $m_i = [\alpha_i , r_i]$ in global coordinates is unknown, but can now easily be calculated if an estimate of the robot's position $\hat{\mathbf{x}}_k$ is known. The variance of m_i's components is now the variance of the robot's position plus the variance of the observation.

Now consider the robot moving closer to the obstacle and obtaining additional observations. Although its uncertainty in position is growing, it can now rely on the feature mi to reduce the variance of its old position (as long as its known that the feature is not moving). Also, repeated observations of the same feature from different angles might improve the quality of its observation. The robot has therefore a chance to keep its variance very close to that with which it initially observed the feature and stored it into its map. We can actually do this using the EKF framework from Section 9.5. There, we assumed that features have a known location (no variance), but that the robot's sensing introduces a variance. This variance was propagated into the covariance matrix of the innovation (**S**). We can now simply add the variance of the estimate of the feature's position to that of the robot's sensing process.

11.1.2. Special Case II: Two Features

Consider now a map that has two features. Visiting one after the other, the robot will be able to store both of them in its map, although with a higher variance for the feature observed last. Although the observations of both features are independent from each other, the relationship between their variances depend on the trajectory of the robot. The differences between these two variances are much lower if the robot connect them in a straight line than when it performs a series of turns between them. In fact, even if the variances of both features are huge (because the robot has already driven for quite a while before first encountering them), but the features are close together, the probability density function over their distance would be very small. The latter can also be understood as the covariance of the two random variables (each consisting of range and angle). In probability theory, the covariance is the measure of how much two variables are changing together. Obviously, the covariance between the locations of two features that are visited immediately after each other by a robot is much higher as those far apart. It should therefore be possible to use the covariance between features to correct estimates of features in retrospect. For example, if the robot returns to the first feature it has observed, it will be able to reduce the variance of its position estimate. As it knows that it has not traveled very far since it observed the last feature, it can then correct this feature's position estimate.

11.2: The Covariance Matrix

When estimating quantities with multiple variables, such as the position of a robot that consists of its x-position, its y-position and its orientation, matrix notation has been a convenient way of writing down equations. For error propagation, we have written the variances of each input variable into the diagonal of a covariance matrix. For example, when using a differential wheel robot, uncertainty in position expressed by σ_x, σ_y and σ_θ were grounded in the uncertainty of its left and right wheel. We have entered the variances of the left and right wheel into a 2x2 matrix and obtained a 3x3 matrix that had σ_x, σ_y and σ_θ in its diagonal. Here, we set all other entries of the matrix to zero and ignored entries in the resulting matrix that were not in its diagonal. The reason we could actually do this is because uncertainty in the left and right wheel are independent random processes: there is no reason that the left wheel slips, just because the right wheel slips. Thus the covariance — the measure on how much two random variables are changing together — of these is zero. This is not the case for the robot's position: uncertainty in one wheel will affect all output random variables (σ_x, σ_y and σ_θ) at the same time, which is expressed by their non-zero covariances — the non-zero entries off the diagonal of the output covariance matrix.

11.3: EKF SLAM

The key idea in EKF SLAM is to extend the state vector from the robot's position to contain the position of all features. Thus, the state

$$\hat{x}_{k'|k-1} = (x, y, \theta)^T \tag{11.3.1}$$

becomes

$$\hat{x}_k = (x, y, \theta, \alpha_1, ..., \alpha_N, r_N)^T \tag{11.3.2}$$

assuming N features, which is a (3 + 2N)x1 vector. The action update (or "prediction update") is identical to that if features are already known; the robot simply updates its position using odometry and updates the variance of it s position using error propagation. The covariance matrix is now a (3+2N)x(3+2N) matrix that initially holds the variances on position and those of each feature in its diagonal.

The interesting things happen during the perception update. Here it is important that only one feature is observed at a time. Thus, if the robot observes multiple features at once, one needs to do multiple, consecutive perception updates. Care needs to be taken that the matrix multiplications work out. In practice you will need to set only those values of the observation vector (a (3+2N)x1 vector) that correspond to the feature that you observe. Similar considerations apply to the observation function and its Jacobian.

11.4: Graph-based SLAM

Usually, a robot obtains an initial estimate of where it is using some onboard sensors (odometry, optical flow, etc.) and uses this estimate to localize features (walls, corners, graphical patterns) in the environment. As soon as a robot revisits the same feature twice, it can update the estimate on its location. This is because the variance of an estimate based on two independent measurements will always be smaller than any of the variances of the individual measurements. As consecutive observations are not independent, but rather closely correlated, the refined estimate can then be propagated along the robot's path. This is formalized in EKF-based SLAM. A more intuitive understanding is provided by a spring-mass analogy: each possible pose (mass) is constrained to its neighboring pose by a spring. The higher the uncertainty of the relative transformation between two poses (e.g., obtained using odometry), the weaker the spring. Every time a robot gains confidence on a relative pose, the spring is stiffened instead. Eventually, all poses will be pulled in place. This approach is known as Graph-based SLAM , see also (?).

11.4.1. SLAM as a Maximum-Likelihood Estimation Problem

The classical formulation of SLAM describes the problem as maximizing the posterior probability of all points on the robot's trajectory given the odometry input and the observations. Formally,

$$p(x_{1:T}, m | z_{1:T}, u_{1:T}) \tag{11.4.1}$$

where $x_{1:T}$ are all discrete positions from time 1 to time T, z are the observations, and u are the odometry measurements. This formulation makes heavily use of the temporal structure of the problem. In practice, solving the SLAM problem requires

1. A motion update model, i.e., the probability $p(x_t | x_{t-1}, u_t)$ to be at location x_t given an odometry measurement u_t and being at location x_{t-1}.
2. A sensor model, i.e., the probability $p(z_t | x_t , m_t)$ to make observation z_t given the robot is at location x_t and the map m_t .

A possible solution to this problem is provided by the Extended Kalman Filter, which maintains a probability density function for the robot pose as well as the positions of all features on the map. Being able to uniquely identify features in the environment is of outmost importance and is known as the data association problem. Like EKF-based SLAM, graph-based SLAM does not solve this problem and will fail if features are confused.

In graph-based SLAM, a robot's trajectory forms the nodes of a graph whose edges are transformations (translation and rotation) that have a variance associated with it. An alternative view is the spring-mass analogy mentioned above. Instead of having each spring wiggle a node into place, graph-based SLAM aims at finding those locations that maximize the joint likelihood of all observations. As such, graph-based SLAM is a maximum likelihood estimation problem.

Lets revisit the normal distribution:

$$\frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \tag{11.4.2}$$

It provides the probability for a measurement to have value x given that this measurement is normal distributed with mean μ and variance σ^2 . We can now associate such a distribution with every node-to-node transformation, aka constraint. This can be pairs of distance and angle, e.g. In the literature the measurement of a transformation between node i and a node j is denoted z_{ij}. Its expected value is denoted $\hat{z}_{ij}$. This value is expected for example based on a map of the environment that consists of previous observations.

Formulating a normal distribution of measurements z_{ij} with mean $\hat{z}_{ij}$ and a covariance matrix Σ_{ij} (containing all variances of the components of z_{ij} in its diagonal) is now straightforward. As graph-based SLAM is most often formulated as information filter, usually the inverse of the covariance matrix (aka information matrix) is used, which we denote by $\Omega_{ij} = \Sigma^{-1}{}_{ij}$.

As we are interested in maximizing the joint probability of all measurements Πz_{ij} over all edge pairings ij following the maximum likelihood estimation framework, it is customary to express the PDF using the log-likelihood. By taking the natural logarithm on both sides of the PDF expression, the exponential function vanishes and $\ln \Pi z_{ij}$ becomes $\Sigma \ln z_{ij}$ or Σl_{ij} , where l_{ij} is the log-likelihood distribution for z_{ij} .

$$l_{ij} \alpha (z_{ij} - \hat{z}_{ij}(x_i, x_j))^T \Omega_{ij} (z_{ij} - \hat{z}_{ij}(x_i, x_j)) \tag{11.4.3}$$

Again, the log-likelihood for observation zij is directly derived from the definition of the normal distribution, but using the information matrix instead of the covariance matrix and is ridden of the exponential function by taking the logarithm on both sides.

The optimization problem can now be formulated as

$$x^* = \arg\min_x \sum_{<ij>\in C} e_{ij}^T \Omega_{ij} e_{ij} \tag{11.4.4}$$

with e_{ij} (x_i , x_j) = z_{ij} − $\hat{z}_{ij}$ (x_i , x_j) the error between measurement and expected value. Note that the sum actually needs to be minimized as the individual terms are technically the negative log-likelihood.

11.4.2. Numerical Techniques for Graph-based SLAM

Solving the MLE problem is non-trivial, especially if the number of constraints provided, i.e., observations that relate one feature to another, is large. A classical approach is to linearize the problem at the current configuration and reducing it to a problem of the form Ax = b. The intuition here is to calculate the impact of small changes in the positions of all nodes on all e_{ij} . After performing this motion, linearization and optimization can be repeated until convergence.

Recently, more powerful numerical methods have been developed. Instead of solving the MLE, one can employ a stochastic gradient descent algorithm. A gradient descent algorithm is an iterative approach to find the optimum of a function by moving along its gradient. Whereas a gradient descent algorithm would calculate the gradient on a fitness landscape from all available constraints, a stochastic gradient descent picks only a (non-necessarily random) subset. Intuitive examples are fitting a line to a set of n points, but taking only a subset of these points when calculating the next best guess. As gradient descent works iteratively, the hope is that the algorithm takes a large part of the constraints into account. For solving Graph-based SLAM, a stochastic gradient descent algorithm would not take into account all constraints available to the robot, but iteratively work on one constraint after the other. Here, constraints are observations on the mutual pose of nodes i and j. Optimizing these constraints now requires moving both nodes i and j so that the error between where the robot thinks the nodes should be and what it actually sees gets reduced. As this is a trade-off between multiple, maybe conflicting observations, the result will approximate a Maximum Likelihood estimate.

More specifically, with e_{ij} the error between an observation and what the robot expects to see, based on its previous observation and sensor model, one can distribute the error along the entire trajectory between both features that are involved in the constraint. That is, if the constraint involves features i and j, not only i and j's pose will be updated but all points in between will be moved a tiny bit.

In Graph-based SLAM, edges encode the relative translation and rotation from one node to the other. Thus, altering a relationship between two nodes will automatically propagate to all nodes in the network. This is because the graph is essentially a chain of nodes whose edges consist of odometry measurements. This chain then becomes a graph whenever observations (using any sensor) introduce additional constraints. Whenever such a "loop-closure" occurs, the resulting error will be distributed over the entire trajectory that connects the two nodes. This is not always necessary, for example when considering the robot driving a figure-8 pattern. If a loop-closure occurs in one half of the 8, the nodes in the other half of the 8 are probably not involved.

This can be addressed by constructing a minimum spanning tree (MST) of the constraint graph. The MST is constructed by doing a Depth-First Search (DFS) on the constraint graph following odometry constraints. At a loop-closure, i.e., an edge in the graph that imposes a constraint to a previously seen pose, the DFS backtracks to this node and continues from there to construct the spanning tree. Updating all poses affected by this new constraint still requires modifying all nodes along the path between the two features that are involved, but inserting additional constraints is greatly simplified. Whenever a robot observes new relationships between any two nodes, only the nodes on the shortest path between the two features on the MST need to be updated.

CHAPTER OVERVIEW

12: RGB-D SLAM

Range sensors have emerged as one of the most effective sensors to make robots autonomous. Unlike vision, range data makes the construction of a 3D model of the robot's environment straightforward and the Velodyne sensor, that combines 64 scanning lasers into one package, was key in mastering the DARPA Grand Challenge. 3D range data has become even more important in robotics with the advent of cheap (priced at a tenth than the cheapest 2D laser scanner) RGB-D (color image plus depth) cameras. Point cloud data allows fitting of lines using RANSAC, which can serve as features in EKF-based localization, but can also be used for improving odometry, loopclosure detection, and mapping. The goals of this chapter are

- introduce the Iterative Closest Point (ICP) algorithm
- show how ICP can be improved by providing initial guesses via RANSAC
- show how SIFT features can be used to improve point selection and loop-closure in ICP to achieve RGB-D mapping

12.1: Converting Range Data into Point Cloud Data

Point cloud data can be thought of a 3D matrix that maps a certain volume in 3D space. Each cell in this matrix, also known as Voxel, corresponds to whether there is an obstacle in this volume or not. Different intensity values could correspond to the uncertainty with which this space is to be known to be an obstacle. An efficient method to turn range information into such an uncertainty 3D map is described in (Curless & Levoy 1996) and became known as Truncated Surface Distance Function (TSDF), commonly referred to as “Point cloud”.

12.2: The Iterative Closest Point (ICP) Algorithm

The Iterative Closest Point (ICP) algorithm was presented in the early 1990s for registration of 3D range data to CAD models of objects. A more in-depth overview of what is described here is given in (Rusinkiewicz & Levoy 2001). The key problem can be reduced to find the best transformation that minimizes the distance between two point clouds. This is the case when matching snapshots from a range sensor or matching a range image with a point cloud sampled from a 3D representation of an object.

In robotics, ICP found an application to match scans from 2D laser range scanners. For example, the transformation that minimizes the error between two consecutive snapshots of the environment is proportional to the motion of the robot. This is a hard problem as it is unclear, which points in the two consecutive snapshots are “pairs”, which of the points are outliers (due to noisy sensors), and which points need to be discarded as not all points overlap in both snapshots. Stitching a series of snapshots together theoretically allows to create a 2D map of the environment. This is difficult, however, as the error between every snapshots — similar to odometry — accumulates. The ICP algorithm also works in 3D where it allows to infer the change in 6D pose of a camera and creation of 3D maps. In addition, ICP has proven useful for identifying objects from a database of 3D objects.

Before providing a solution to the mapping problem, we will focus on the ICP algorithm to match 2 consecutive frames. Variants of the ICP algorithm can be broken down into 6 consecutive steps:

1. Selection of points in one or both meshes or point clouds.
2. Matching/Pairing these points to samples in the other point cloud/mesh.
3. Weighting the corresponding pairs.
4. Rejecting certain pairs.
5. Assigning an error metric based on the point pairs.
6. Minimizing the error metric.
7. Point Selection

Depending on the number of points generated by the range sensor, it might make sense to use only a few selected points to calculate the optimal transformation between two point clouds, and then test this transformation on all points. Depending on the source of the data, it also turns out that some points are more suitable than others as it is easier to identify matches for them. This is the case for RGB-D data, where SIFT features have been used successfully. This is also the case for planar objects with grooves, where sampling should ensure that angles of normal vectors of sampling points are broadly distributed. Which method to use is therefore strongly dependent on the kind of data being used and should be considered for each specific problem.

12.2.1. Matching Points

The key step in ICP is to match one point to its corresponding point. For example, a laser scanner hits a certain point at a wall with its 67th ray. After the scanner has been moved by 10 cm, the closest hit on the wall to this point might have been by the 3rth ray of the laser. Here, it is actually very unlikely that the laser hits the exact same point on the wall twice, therefore introducing a non-zero error even for optimal pairing. Prominent methods are to find the closest point in the other point cloud or to find the intersection of the source points normal with the destination surface (for matching point clouds to meshes). More recently, SIFT has allowed to match points based on their visual appearance. Similarly to sorting through SIFT features, finding the closest matching point can be accelerated by representing the point cloud in a k-d tree.

12.2.2. Weighting of Pairs

As some pairs are better matches than others, weighting them in some smart way might drastically improve the quality of the resulting transformation. One approach is to give more weight to points that have smaller distances from each other. Another approach is to take into account the color of the point (in RGBD images) or use the distance of their SIFT features (weighting pairs with low distances higher than pairs with high distances). Finally, expected noise can be used to weight pairings. For example, the estimates made by a laser scanner are much more faithful when taken orthogonally to a plane than when taken at a steep angle.

12.2.3. Rejecting of Pairs

A key problem in ICP are outliers either from sensor noise or simply from incomplete overlap between two consecutive range images. A prime approach in dealing with this problem is to reject pairings of which one of the points lies on a boundary of the point cloud as these points are likely to match with points in non-overlapping regions. As a function of the underlying data, it might

also make sense to reject pairings with too high of a distance. This is a threshold-based equivalent to distance-based weighting as described above.

12.2.4. Error Metric and Minimization Algorithm

After points have been selected and matched, pairs have been weighted and rejected, the match between two point clouds needs to be expressed by a suitable error metric, which needs then to be minimized. A straightforward approach is to consider the sum of squared distances between each pair. This formulation can often be solved analytically. Let

$$A = \{a_1, ..., a_n\} \tag{12.2.1}$$

$$B = \{b_1, ..., b_n\} \tag{12.2.2}$$

be point clouds in R^n . The goal is now to find a vector $t \in R^n$ so that an error function $\varphi(A+t, B)$ is minimized. In 6D (translation and rotation), an equivalent notation can be found for a transformation (see forward kinematics). An error function for the squared distance is then given by

$$\phi(A+t, B) = \frac{1}{n} \sum_{a \in A} \|a+t-N_B(a+t)\|^2 \tag{12.2.3}$$

Here $N_B(a+t)$ is a function that provides the nearest neighbor of a translated by b in B. A key problem now is that the actual value of t affects the outcome of the pairing. What might look like a good match initially often turns out not be the final pairing. A simple numerical approach to this problem is to find t iteratively.

Initially $t = 0$ and nearest neighbors/pairings are established. We can now calculate a δt that optimizes the least-square problem based on this matching using any solver available for the optimization problem (for a least-square solution δt can be obtained analytically by solving for the minimum of the polynomial by setting its derivative to zero). We can then shift all points in A by δt and start over. That is, we calculate new pairings and derive a new δt. We can continue to do this, until the cost function reaches a local minimum.

Instead of formulating the cost function as a “point-to-point” distance, a “point-to-plane” has become popular. Here, the cost function consist of the sum of squared distances from each source point to the plane that contains the destination point and is oriented perpendicular to the destination normal. This makes particularly sense when matching a point cloud to a mesh/CAD model of an object. In this case there are no analytical solutions to finding the optimal transformation, but any optimization method such a Levenberg-Marquardt can be used.

12.3: RGB-D Mapping

The ICP algorithm can be used to stitch consecutive range images together to create a 3D map of the environment (Henry, Krainin, Herbst, Ren & Fox 2010). Together with RGB information, it is possible to create complete 3D walk throughs of an environment. An example of such a walk through using the method described in (Whelan, Johannsson, Kaess, Leonard & McDonald 2013) is shown in Figure 12.3.1. A problem with ICP is that errors in each transformation propagate making maps created using this method as odd as maps created by simple odometry. Here, the SLAM algorithm can be used to correct previous errors once a loop closure is detected.

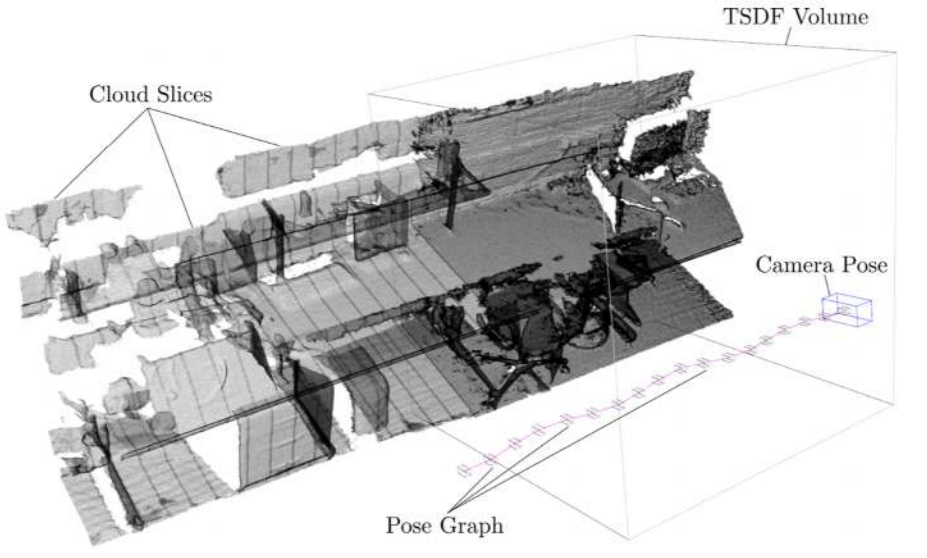


Figure **12.3.1**: Fused point cloud data from a walk trough of an office environment using "Kintinious". Picture courtesy of John Leonard.

The intuition behind SLAM is to consider each transformation between consecutive snapshots as a spring with variable stiffness. Whenever the robot returns to a previously seen location, i.e., a loop-closure has been determined, additional constraints are introduced and the collection of snapshots connected by springs become a mesh. Everytime the robot then re-observes a transformation between any of the snapshots, it can "stiffen" the spring connecting the two. As all of the snapshots are connected, this new constraints propagates through the network and literally pull each snapshots in place.

RGB-D Mapping uses a variant of ICP that is enhanced by SIFT features for point selection and matching. Maps are build incrementally. SIFT features, and their spatial relationship, are used for detecting loop closures. Once a loop closure is detected, an additional constraint is added to the pose graph and a SLAM-like optimization algorithm corrects the pose of all previous observations.

As ICP only works when both point clouds are already closely aligned, which might not be the case for a fast moving robot with a relatively noisy sensor (the XBox Kinect has an error of 3cm for a few meters of range vs. millimeters in laser range scanners), RGB-D Mapping uses RANSAC to find an initial transformation. Here, RANSAC works as for line fitting: it keeps guessing possible transformations for 3 pairs of SIFT feature points and then counts the number of inliers when matching the two point clouds, one of which being transformed using the random guess.

CHAPTER OVERVIEW

13: Trigonometry

Trigonometry relates angles and lengths of triangles. Figure A.1 shows a right-angled triangle and conventions to label its corners, sides, and angles. In the following, we assume all triangles to have at least one right angle (90 degrees or π/2) as all planar triangles can be dissected into two right-angled triangles.

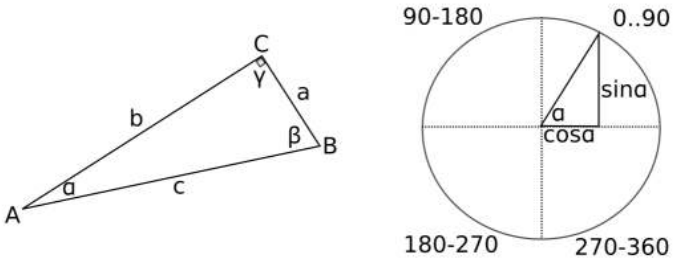


Figure 13.1: Left: A right-angled triangle with common notation. Right: Trigonometric relationships on the unit circle and angles corresponding to the four quadrants.

The sum of all angles in any triangle is 180 degrees or 2π, or

$$\alpha + \beta + \gamma = 180^\circ \tag{13.1}$$

If the triangle is right-angled, the relationship between edges a, b, and c, where c is the edge opposite of the right angle is

$$a^2 + b^2 = c^2 \tag{13.2}$$

The relationship between angles and edge lengths are captured by the trigonometric functions:

$$\sin\alpha = \frac{opposite}{hypothenuse} = \frac{a}{c} \tag{13.3}$$

$$\cos\alpha = \frac{adjacent}{hypothenuse} = \frac{b}{c} \tag{13.4}$$

$$\tan\alpha = \frac{opposite}{adjacent} = \frac{\sin\alpha}{\cos\alpha} = \frac{a}{b} \tag{13.5}$$

Here, the hypothenuse is the side of the triangle that is opposite to the right angle. The adjacent and opposite are relative to a specific angle. For example, in Figure 13.1, the adjacent of angle α is side b and the opposite of α is edge a.

Relations between a single angle and the edge lengths are captured by the *law of cosines*:

$$a^2 = b^2 + c^2 - 2bc\cos\alpha \tag{13.6}$$

13.1: Inverse trigonometry

In order to calculate an angle given two edges, one uses inverse functions $\sin^{-1}$, $\cos^{-1}$, and $\tan^{-1}$. (Not to be confused with 1/sin etc.) As functions can, by definition, only map one value to exactly one other value, $\sin^{-1}$ and $\tan^{-1}$ are only defined in the interval [−90°; +90°] and cos−1 is defined in the interval [0°; 180°]. This makes it impossible to calculate angles in the 2nd and 3rd, or the 3rd and 4th quadrant, respectively (Figure 13.1). In order to overcome this problem, most programming languages implement a function **`atan2(opposite,adjacent)`**, which evaluates the sign of the numerator and denumerator, provided as two separate parameters.

13.2: Trigonometric Identities

Sine and cosine are periodic, leading to the following identities:

$$\sin\theta = -\sin(-\theta) = -\cos(\theta + \frac{\pi}{2}) = \cos(\theta - \frac{\pi}{2}) \tag{13.2.1}$$

$$\cos\theta = \cos(-\theta) = \sin(\theta + \frac{\pi}{2}) = -\sin(\theta - \frac{\pi}{2}) \tag{13.2.2}$$

The sine or cosine for sums or differences between angles can be calculated using the following identities:

$$\cos(\theta_1 + \theta_2) = \cos(\theta_1)\cos(\theta_2) - \sin(\theta_1)\sin(\theta_2) \tag{13.2.3}$$

$$\sin(\theta_1 + \theta_2) = \sin(\theta_1)\cos(\theta_2) + \cos(\theta_1)\sin(\theta_2) \tag{13.2.4}$$

$$\cos(\theta_1 - \theta_2) = \cos(\theta_1)\cos(\theta_2) + \sin(\theta_1)\sin(\theta_2) \tag{13.2.5}$$

$$\sin(\theta_1 - \theta_2) = \sin(\theta_1)\cos(\theta_2) - \cos(\theta_1)\sin(\theta_2) \tag{13.2.6}$$

The sum of the squares of sine and cosine for the same angle is one:

$$\cos(\theta)\cos(\theta) + \sin(\theta)\sin(\theta) = 1 \tag{13.2.7}$$

CHAPTER OVERVIEW

14: Linear Algebra

Linear algebra concerns vector spaces and linear mappings between them. It is central to robotics as it allows describing positions and speeds of the robot within the world as well as moving parts connected to it, as well as in processing image and depth data, which is often presented in matrix form.

14.1: Dot Product

The dot product (or scalar product) is the sum of the products of the individual entries of two vectors. Let *hata* = (a_1, . . . , a_n) and ˆb= (b1, . . . , bn) be two vectors. Then, their dot product â · ˆb is given by

$$\hat{a} \cdot \hat{b} = \sum_{i}^{n} a_i b_i \tag{14.1.1}$$

The dot product therefore takes two sequences of numbers and returns a single scalar. In robotics, the dot product is mostly relevant due to its geometric interpretation:

$$\hat{a} \cdot \hat{b} = \|\hat{a}\| \left\|\hat{b}\right\| \cos\theta \tag{14.1.2}$$

with θ the angle between vectors â and ˆb

If â and ˆb are orthogonal, it follows â · ˆb = 0. If â and ˆb are parallel, it follows â · ˆb = ||â|| ||ˆb||.

14.2: Cross Product

The cross product â × ˆb of two vectors is defined as a vector ĉ that is perpendicular to both â and ˆb. Its direction is given by the right-hand rule and its magnitude is equal to the area of the parallelogram that the vectors span.

Let $\hat{a} = (a_1, a_2, a_3)^T$ and $\hat{b} = (b_1, a_2, a_3)$ be two vectors in R^3 . Then, their cross product â × ˆb is given by

$$\hat{a} \times \hat{b} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix} \tag{14.2.1}$$

14.3: Matrix Product

Given an n × m matrix **A** and a m × p matrix **B**, the matrix product **AB** is defined by

$$(AB)_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj} \tag{14.3.1}$$

where the index *ij* indicates the i-th row and j-th column entry of the resulting n × p matrix. Each entry therefore consists of the scalar product of the i-th row of **A** with the j-th column of **B**.

Note that for this to work, the right hand matrix (here **B**) has to have as many columns as the left hand matrix (here **A**) has rows. Therefore, the operation is not commutative, i.e., **AB** ≠ **BA**.

For example, multiplying a 3x3 matrix with a 3x1 matrix (a vector), works as follows: Let

$$\boldsymbol{A} = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix} \quad \boldsymbol{B} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{14.3.2}$$

Then their matrix product is:

$$\boldsymbol{AB} = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} ax + by + cz \\ px + qy + rz \\ ux + vy + wz \end{pmatrix} \tag{14.3.3}$$

14.4: Matrix Inversion

Given a matrix $\mathbf{A}$, finding the inverse $\mathbf{B} = \mathbf{A}^{-1}$ involves solving the system of equations that satisfies

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I} \tag{14.4.1}$$

with $\mathbf{I}$ the identity matrix. (The identity matrix is zero everywhere except at its diagonal entries, which are one.)

In the particular case of orthonormal matrices, which columns are all orthogonal to each other and of length one, the inverse is equivalent to the transpose, i.e.

$$\mathbf{A}^{-1} = \mathbf{A}^{T} \tag{14.4.2}$$

This is important, as rotation matrices are orthonormal. In case a matrix is not quadratic, we can calculate the pseudo-inverse, which is defined by

$$\mathbf{A}^{+} = \mathbf{A}^{T}(\mathbf{A}\mathbf{A}^{T})^{-1} \tag{14.4.3}$$

and is often used in finding an inverse kinematic solution.

14.5: Principal Component Analysis

PCA breaks n-dimensional data into n vectors so that each data point can be represented by a linear combination of the n vectors. These n vectors have two interesting properties: first, they are ordered by their variance so that the first vector is representative of the data with the highest variation in the data, and second, they are orthogonal. These vectors are therefore called *principal components*.

This approach has a strong geometrical interpretation: given data such as two-dimensional points, say in the shape of a rectangle, the points along the long axis of the rectangle have higher variance than those along the the short axis. Every point in this point cloud can then be reconstructed by a linear combination of the principal component along the long axis and the principal component along the short axis. Finding these vectors is therefore akin finding the principal axes of the rectangle regardless of its orientation.

CHAPTER OVERVIEW

15: Statistics

15.1: Random Variables and Probability Distributions

Random variables can describe either discrete variables, such as the result from throwing a dice, or continuous variables such as measuring a distance. In order to learn about the likelihood that a random variable has a certain outcome, we can repeat the experiment many times and record the resulting random variates, that is the actual values of the random variable, and the number of times they occurred. For a perfectly cubic dice we will see that the random variable can hold natural numbers from 1 to 6, that have the same likelihood of 1/6.

The function that describes the probability of a random variable to take certain values is called a *probability distribution*. As the likelihood of all possible random variates in the dice experiment is the same, the dice follows what we call a *uniform distribution*. More accurately, as the outcomes of rolling a dice are discrete numbers, it is actually a discrete uniform distribution. Most random variables are not uniformly distributed, but some variates are more likely than others. For example, when considering a random variable that describes the sum of two simultaneously thrown dice, we can see that the distribution is anything but uniform:

$$\begin{aligned}
2 &: 1+1 \rightarrow \frac{1}{6}\frac{1}{6} \\
3 &: 1+2, 2+1 \rightarrow 2\frac{1}{6}\frac{1}{6} \\
4 &: 1+3, 2+2, 3+1 \rightarrow 3\frac{1}{6}\frac{1}{6} \\
5 &: 1+4, 2+3, 3+2, 4+1 \rightarrow 4\frac{1}{6}\frac{1}{6} \\
6 &: 1+5, 2+4, 3+3, 4+2, 5+1 \rightarrow 5\frac{1}{6}\frac{1}{6} \\
7 &: 1+6, 2+5, 3+4, 4+3, 5+2, 6+1 \rightarrow 6\frac{1}{6}\frac{1}{6} \\
8 &: 1+5, 2+4, 3+3, 4+2, 5+1 \rightarrow 5\frac{1}{6}\frac{1}{6} \\
9 &: 1+4, 2+3, 3+2, 4+1 \rightarrow 4\frac{1}{6}\frac{1}{6} \\
10 &: 1+3, 2+2, 3+1 \rightarrow 3\frac{1}{6}\frac{1}{6} \\
11 &: 1+2, 2+1 \rightarrow 2\frac{1}{6}\frac{1}{6} \\
12 &: 1+1 \rightarrow \frac{1}{6}\frac{1}{6}
\end{aligned}$$

As one can see, there are many more possibilities to sum up to a 7 than there are to a 3, e.g. While it is possible to store probability distributions such as this one as a look-up table to predict the outcome of an experiment (or that of a measurement), we can also calculate the sum of two random processes analytically (Section C.3).

15.1.1. The Normal Distribution

One of the most prominent distribution is the Gaussian or Normal Distribution. The *Normal distribution* is characterized by a *mean* and a *variance*. Here, the mean corresponds to the average value of a random variable (or the peak of the distribution) and the variance is a measure of how broadly variates are spread around the mean (or the width of the distribution). The Normal distribution is defined by the following function

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{15.1.1}$$

where μ is the mean and σ^2 the variance. (σ on its own is known as the standard deviation.) Then, f(x) is the probability for a random variable X to have value x. The mean is calculated by

$$\mu = \int_{-\infty}^{\infty} x f(x) dx \tag{15.1.2}$$

or in other words, each possible value x is weighted by its likelihood and added up.

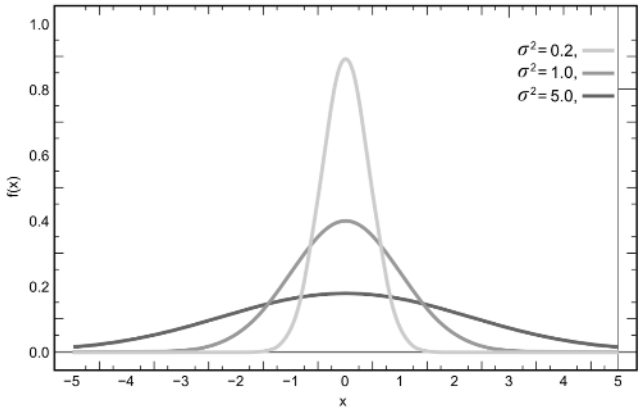


Figure **15.1.1**: Normal distribution for different variances and μ = 0.

The variance is calculated by

$$\sigma^2 = \int_{-\infty}^{\infty} (x-\mu)^2 f(x) dx \tag{15.1.3}$$

or in other words, we calculate the deviation of each random variable from the mean, square it, and weigh it by its likelihood. Although it is tantalizing to perform this calculation also for the double dice experiment, the resulting value is questionable, as the double dice experiment does not follow a Normal distribution. We know this, because we actually enumerated all possible outcomes. For other experiments, such as grades in the classes you are taking, we don't know what the real distribution is.

15.1.2. Normal Distribution in Two Dimensions

The Normal Distribution is not limited to random processes with only one random variable. For example, the X/Y position of a robot in the plane is a random process with two dimensions. In case of a multi-variate distribution with k dimensions, the random variable X is a k-dimensional vector of random variables, μ is a k-dimensional vector of means, and σ gets replaced with Σ, a k-by-k dimensional *covariance matrix* (a matrix that carries the variances of each random variable in its diagonal).

15.2: Conditional Probabilities and Bayes Rule

Let A and B be random events with probabilities P(A) and P(B). We can now say that the probability P(A ∩ B) that event A and B happen is given by

$$P(A \cap B) = P(A)P(B|A) = P(B)P(A|B) \tag{15.2.1}$$

Here, P(B|A) is the conditional probability that B happens, knowing that event A happens. Likewise, P(A|B) is the probability that event A happens given that B happens.

Bayes' Rule relates a conditional probability to its inverse. In other words, if we know the probability of event A to happen given that event B is happening, we can calculate the probability of B to occur given that A is happening. Bayes' rule can be derived from the simple observation that the probability of A and B to happen together (P(A∩B)) is given by P(A)P(B|A) or the probability of A to happen and the probability of B to happen given that A happens (Equation C.4). From this, deriving Bayes' rule is straightforward:

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)} \tag{15.2.2}$$

In words, if we know the probability that B happens given that A happens, we can calculate that A happens given that B happens.

15.3: Sum of Two Random Processes

Let X and Y be the random variables associated with the numbers shown on two dice (see above), and Z = X + Y . With P(X = x), P(Y = y), and P(Z = z) being the probabilities associated with the random variables taking specific values x, y or z. Given z = x + y, the event Z = z is the union of the independent events X = k and Y = z − k. We can therefore write

$$P(Z = z) = \sum_{k=-\infty}^{\infty} P(X = k)P(Y = z - k) \tag{15.3.1}$$

which is the exact definition of a convolution, also written as

$$P(Z) = P(X) \star P(Y) \tag{15.3.2}$$

Numerically calculating the convolution always works, and can be done analytically for some probability distributions. Conveniently, the convolution of two Gaussian distributions is again a Gaussian distribution with a variance that corresponds to the sum of the variances of the individual Gaussians.

15.4: Linear Combinations of Independent Gaussian Random Variables

Let $X_1, X_2, \ldots, X_n$ be n independent random variables with means $\mu_1, \mu_2, \ldots, \mu_n$ and variances $\sigma^2_1, \sigma^2_2, \ldots,$ and σ^2_n. Let Y be a random variable that is a linear combination of X_i with weights a_i so that $Y = \sum_{i=1}^{n} a_i X_i$.

As the sum of two Gaussian random variables is again a Gaussian, Y is Gaussian distributed with a mean

$$\mu_Y = \sum_{i=1}^{n} a_i \mu_i \tag{15.4.1}$$

and a variance

$$\sigma^2_Y = \sum_{i=1}^{n} a_i^2 \sigma_i^2 \tag{15.4.2}$$

15.5: Testing Statistical Significance

- Robotics is an experimental discipline. This means that algorithms and systems you develop need to be validated by real hardware experiments. Doing an experiment to validate your hypothesis is at the core of the scientific method and doing it right is a discipline on its own. The key is to show that your results are not simply a result of chance. In practice, this is impossible to show. Instead, it is possible to express the likelihood that your results have not been obtained by chance. This is known as the statistical significance level. How to calculate the statistical significance level depends on the problem you are studying. This section will introduce three common problems in robotics:

1. testing whether data is indeed distributed according to a specific distribution
2. testing whether two sets of data are generated from different distributions
3. testing whether true-false experiments are a sequence of luck or not

15.5.1. Null Hypothesis on Distributions

The Null Hypothesis is a term from the statistical significance literature and formally captures your main claim. A statistical test can either reject the Null Hypothesis or fail to reject it. It can never be proven as there will always be a non-zero probability that all your experiments are just a lucky coincidence. The statistical significance level of a Null Hypothesis is known as the p-value.

An import class of Null Hypothesis are on the distribution of data. Consider the following example from Lab 1 (message passing in ROS). Students were asked to experimentally study the time it takes to pass a message from one process to another:

- Histogram of the time it takes to send a ROS message from one process to another based on 10 trials.

We observe three peaks in this Histogram. What can we say about message passing times? For example

- H0: Message passing times follow a Gaussian distribution.
- H0: Message passing times follow a bi-modal distribution.
- H0: Message passing times follow a log-normal distribution.

The first Null Hypothesis implies that messages take sometimes a little longer and sometimes a little shorter, but have an average and a variance. The second Null Hypothesis implies that usually messages take some low average time, but occasionally are delayed due to the influence of some other process, for example operating system duties. You can now test each of these hypotheses by calculating the parameters of the distribution to expect and calculate the joint probability that each of your measurements are actually drawn from this distribution. You will find, that all of the above hypotheses are almost equally likely. Together, none of your tests will reject your hypothesis. You therefore will need more data:

- Histogram of message passing times in ROS based on a 1000 trials.

You can now again calculate parameters for each distribution you suspect. For example, you can calculate the mean and variance of this data and plot the resulting Gaussian distribution. In this example, the Gaussian distribution will have a mean slightly offset to the right of the peak. You can also fit the data to a log-normal distribution. You can now calculate the likelihood for the data actually be drawn from either of the two distributions. You will see that the joint probability (the product of all likelihoods) for all data points is actually much higher than that for any Gaussian distribution or any bimodal distribution that you are able to fit.

Formally, this can be done by following Pearsons χ^2 -Test (read Chi-Squared Test). This test calculates a value that will approximate a χ^2 -distribution from all samples and the likelihood of that sample based on the expected distribution. Plugging the resulting value into the χ^2 -distribution leads to the statistical significance level (or p-value).

The value of the test-statistic is calculated as follows:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \tag{15.5.1}$$

where

- χ^2 = Pearson's cumulative test statistic, which asymptotically approaches a chi-squared distribution.
- O_i = an observed frequency in the data histogram
- E_i = an expected (theoretical) frequency, asserted by the null hypothesis, i.e., the distribution you think the data should follow

- n = the number of samples.

This example also illustrates how statistical tests can be used to determine if you have enough data. If you don't, you will get very poor p-values. In practice, it is up to you what likelihood you determine to be significant. Standard significance levels are 10%, 5% and 1%. If you are unsatisfied with your p-values you can collect more data and check, whether your p-value improves.

15.5.2. Testing Whether Two Distributions are Independent

Testing whether the data of two experiments are independent is probably the most common statistical test. For example, you might run 10 experiments using algorithm 1 and 10 experiments using algorithm 2. It is up to you to show that the resulting distributions are indeed statistically significantly different. In other words, you need to show that the differences between the algorithm indeed lead to a systematic improvement, and that it was not purely luck that one set of experiments turned out "better" than another.

If you have good reasons to believe that your data is normal distributed, there exist a series of simple tests. For example, to test whether two sets of data are distributed with Gaussian distributions that have the same mean, can be done using Student's t-test. A generalization of Student's t-test to 3 or more groups is ANOVA. These tests have to be done with care as most distributions in robotics are not normal distributed. Examples where Gaussian distributions are commonly assumed are sensor noise on distance measurements such as obtained by infrared or odometry.

If data is not Gaussian distributed, there exist a series of numerical tests to test the likelihood that two distributions are independent. For example, you could test the message passing time with and without running some computationally expensive image processing routines. You can then test whether the additional computation affects message passing time. If it does, both distributions need to be significantly different. Just using Student's t-test does not work as the distributions are not Gaussian!

Instead, testing whether two sets of data have the same mean, needs to be done numerically. A common test is MannWilcoxon's Ranked Sum test. An implementation of this test is part of most mathematical calculation programs such as Matlab or Mathematica. An algorithm to calculate this test statistic and the corresponding p-values is available on the Wikipedia page above. An extension of the Mann-Wilcoxon's Ranked Sum test for 3 or more groups is the Kruskal-Wallis one-way analysis of variance test.

15.5.3. Statistical Significance of True-False Tests

There exists a class of experiments that do not lead to distributions, but result in simple true-false outcomes. For example, a question one might ask is "does the robot correctly understand a spoken command". This class of experiments is captured by the Lady tasting tea example. Here, a lady claims that she can identify the brewing method of a cup of tea: tea prepared by first adding milk and tea prepared by later adding milk. Unfortunately, it is easy to cheat as the likelihood of guessing right is 50%. Testing the hypothesis that the lady can indeed differentiate the two brewing methods therefore requires to conduct a series of experiments to reduce the likelihood of winning by guesswork. In order to do this, one needs to calculate the number of total permutations (or, possible outcomes over the entire series of experiments). For example, one could present the lady 8 cups of tea, 4 brewed one way and four the other. One can now enumerate all possible outcomes of this experiment, ranging from all cups guessed correctly to all cups guessed wrong. There are a total of 70 possible outcomes (see the example provided here). Guessing all cups correctly has now a likelihood of 1/70 or 1.4%. The likelihood to make a single mistake (16 possible outcomes in this example) is around 23%.

15.5.4. Summary

Statistical significance test allow you to express the likelihood that your experiment is not just the result of chance. There exist different tests for different underlying distributions. Therefore, your first task is to convincingly argue what the underlying distribution of your data is. Formally testing how your data is distributed can be achieved using the Chi-Square Test. In order to test whether two sets of data are coming from two different distributions can then be achieved using Student's t-test (if the distribution is Gaussian) or using the Mann-Wilcoxon Ranked Sum test if the probability distribution is non-parametric.

CHAPTER OVERVIEW

16: How to Write a Research Paper

The final deliverable of a robotics class often is a write-up on a “research” project, modeled after research done in industry or academia. Roughly, there are three classes of papers:

1. Original research
2. Tutorial
3. Survey

The goal of this chapter is to provide guidelines on how to think about your project as a research project and how to report on your results as original research.

16.1: Original

Classically, a scientific paper follows the following organization:

1. Abstract
2. Introduction
3. Materials & Methods
4. Results
5. Discussion
6. Conclusion

The *abstract* summarizes your paper in a few sentences. What is the problem you want to solve, what is the method you are employing, what are you doing to assess your work, and what is the final outcome.

The *introduction* should describe the problem that you are solving and why it is important. A good guideline to write a good introduction are the Heilmeier questions:

1. What are you trying to do? Articulate your objectives using absolutely no jargon.
2. How is it done today, and what are the limits of current practice?
3. What's new in your approach and why do you think it will be successful?
4. Who cares?
5. If you're successful, what difference will it make?
6. What are the midterm and final "exams" to check for success?

Originally conceived for proposal writing by the head of DARPA, there are additional questions including "What will it cost?", "How long will it take?", and "What are the risks and pay-off", which are left out for the purpose of writing a research paper. In the context of scientific research, the question "What are you trying to do?" is best answered in the form of a hypothesis, see below.

The *materials & matters* section describes all the tools that you used to solve your problem, as well as your original contribution, e.g., an algorithm that you came up with. This section is hardly ever labeled as such, but might consist of a series of individual section describing the robotic platform you are using, the software packages, and flowcharts and descriptions on how your system works. Make sure you motivate your design choices using conclusive language or experimental data. Validating these design choices could be your first results.

The results section contains data or proofs on how to solve the problem you addressed or why it cannot be solved. It is important that your data is conclusive! You have to address concerns that your results are just a lucky coincidence. You therefore need to run multiple experiments and/or formally prove the workings of your system either using language or math, see also Section 15.5.

The *discussion* should address limitations of your approach, the conclusiveness of its results, and general concerns someone who reads your work might have. Put yourself in the role of an external reviewer who seeks to criticize your work. How could you have sabotaged your own experiment? What are the real hurdles that you still need to overcome for your solution to work in practice? Criticizing your own work does not weaken it, it makes it stronger! Not only does it become clear where its limitations are, it is also more clear where other people can step in.

The *conclusion* should summarize the contribution of your paper. It is a good place to outline potential future work for you and others to do. This future work should not be random stuff that you could possibly think about, but come out of your discussion and the remaining challenges that you describe there. Another way to think about is that the "future work" section of your conclusion summarizes your discussion.

It is important not to mix the different sections up. For example, your result section should exclusively focus on describing your observations and reporting on data, i.e., facts. Don't conjecture here why things came out as they are. You do this either in your hypothesis — the whole reason you conduct experiments in the first place — or in the discussion. Similarly, don't provide additional results in your discussion section.

Try to make the paper as accessible to as many reader styles and attention spans as possible. While this sounds impossible at first, a good way to address this is to think about multiple avenues a reader might take. For example, the reader should get a pretty comprehensive picture on what you do by just reading the abstract, just reading the introduction, or just reading all the figure

captions. (Think about other avenues, every one you address makes your paper stronger.) It is often possible to provide this experience by adding short sentences that quickly recall the main hypothesis of your work. For example, when describing your robotic platform in the materials section, it does not hurt to introduce the section by something like “In order to show that [the main hypothesis of our work], we selected...”. Similarly, you can try to read through your figure captions if they provide enough information to follow the paper and understand its main results on their own. It’s not a problem to be repetitive in a scientific paper, stressing your one-sentence elevator pitch (or hypothesis, see below) throughout the paper is actually a good thing.

16.2: Hypothesis- Or, What Do We Learn From this Work?

Classically, a hypothesis is a proposed explanation for an observed phenomenon. From this, the hypothesis has emerged as the corner stone of the scientific method and is a very efficient way to organize your thoughts and come up with a one sentence summary of your work. A proper formulation of your hypothesis should directly lead to the method that you have chosen to test your hypothesis. A good way to think about your hypothesis is “What do you want to learn?” or “What do we learn from this work?”.

It can be somewhat hard to actually frame your work into a single sentence, so what to do if a single hypothesis seems not to apply? One reason might be that you are actually trying to accomplish too many things. Can you really describe them all in depth in a 6-page document? If yes, maybe some are very minor compared to the others. If this is the case, they are either supportive of your main idea and can be rolled into this bigger piece of work or they are totally disconnected. If they are disconnected, leave them out for the sake of improving the conciseness of your main message. Finally, you might feel that you don’t have a main message, but consider all the things you have done to be equally worthy, and despite answering the Heilmeier questions you cannot fill up more than three pages. In this case you might consider picking one of your approaches and dig deeper by comparing it with different methods.

Being able to come up with a one-sentence elevator pitch framed as a hypothesis will actually help you to set the scope of the work that you need to do for a research or class project.

How good do you need to implement, design or describe a certain component of your project? Well, good enough to follow through with your research objective.

16.3: Survey and Tutorial

The goal of a survey is to provide an overview over a body of work — potentially from different communities — and classify it into different categories. Doing this synthesis and establishing common language and formalism is the survey's main contribution. A survey following such an outline is a possible deliverable for an independent study or a PhD prelim, but it does not lend itself to describe your efforts on a focused research project. Rather, it might result from your involvement in a relatively new area in which you feel important connections between disjoint communities and common language have not been established.

A different category of survey critically examines concurring methods to solve a particular problem. For example, you might have set out to study manipulation, but got stuck in selecting the right sensor suite from the many available options. What sensor is actually best to accomplish a specific task? A survey which answers this question experimentally will follow the same structure as a research paper (see above).

A *tutorial* is closely related to a survey, but focuses more on explaining specific technical content, e.g, the workings of a specific class of algorithms or tool, commonly used in a community. A tutorial might be an appropriate way to describe your efforts in a research project, which can serve as illustration to explain the workings of a specific method you used.

16.4: Writing it up!

Writing a research report that contains equations, figures and references requires some tedious book-keeping. Although technically possible, word processing programs quickly reach their limitations and will lead to frustration. In the scientific community LATEX has emerged as a quasi standard for typesetting research documentation. LATEX is a mark-up language that strictly divides function and layout. Rather than formatting individual items as bold, italic and the like, you mark them up as emphasized, section head etc, and specify how things look elsewhere. This is usually provided by a template provided by the publisher (or your own). While LATEX has quite a learning curve compared to other word processing software, it is quickly worth the effort as soon as you need to start worrying about references, figures or even indices.

CHAPTER OVERVIEW

17: Sample Curricula

This book is designed to cover two full semesters at undergraduate level, CSCI 3302 and CSCI 4302 at CU Boulder, or a single semester “crash course” at graduate level. There are multiple avenues that an instructor could take, each with their unique theme and a varying set of prerequisites on the students.

17.1: An Introduction to Autonomous Mobile Robots

This describes a possible one semester curriculum, which takes the students from the kinematics of a differential-wheel platform to SLAM. This curriculum is involved and requires a firm background in trigonometry, probability theory and linear algebra. This might be too ambitious for third-year Computer Science students, but fares well with Aerospace and Electrical Engineering students, who often have a stronger, and more applied, mathematical background. This curriculum is therefore also well suited as "advanced class", e.g. in the fourth year of a CS curriculum.

17.1.1. Overview

The curriculum is motivated by a maze-solving competition that is described in Section 1.3. Solving the game can be accomplished using a variety of algorithms ranging from wall following (which requires simple proportional control) to Depth-first Search on the maze to full SLAM. Here, the rules are designed such that creating a map of the environment leads to a competitive advantage on the long run.

17.1.2. Materials

The competition can be easily re-created using card board or LEGO bricks and any miniature, differential wheel platform that is equipped with a camera to recognize simple markers in the environment (which serve as landmarks for SLAM). The setup can also easily be simulated in a physics-based simulation environment, which allows scaling this curriculum to a large number of participants. The setup used at CU Boulder using the e-Puck robot and the Webots simulator is shown in Figure 17.1.1.

Figure **17.1.1**: The "Ratslife" maze competition created from LEGO bricks and e-Puck robots (left). The same environment simulated in Webots.

17.1.3. Content

After introducing the field and the curriculum using Chapter 1 "Introduction", another week can be spent on basic concepts from Chapter 2 "Locomotion and Manipulation", which includes concepts like "Static and Dynamic Stability" and "Degreesof-Freedom". The lab portions of the class can at this time be used to introduce the software and hardware used in the competition. For example, students can experiment with the programming environment of the real robot or setup a simple world in the simulator themselves.

The lecture can then take up pace with Chapter 3. Here, the topics "Coordinate Systems and Frames of Reference", "Forward Kinematics of a Differential Wheels Robot", and "Inverse Kinematics of Mobile Robots" are on the critical path, whereas other sections in Chapter 3 are optional. It is worth mentioning that the forward kinematics of non-holonomic platforms, and in particular the motivation for considering their treatment in velocity rather than position space, are not straightforward and therefore at least some treatment of arm kinematics is recommended. These concepts can easily be turned into practical experience during the lab session.

The ability to implement point-to-point motions in configuration space thanks to knowledge of inverse kinematics, directly lends itself to "Map representations" and "Path Planning" treated in Chapter 4. For the purpose of maze solving, simple algorithms like Dijkstra's and A* are sufficient, and sampling-based approaches can be skipped. Implementing a path-planning algorithm both in simulation and on the real robot will provide first-hand experience of uncertainty.

The lecture can then proceed to "Sensors" (Chapter 5), which should be used to motivate uncertainty using concepts like accuracy and precision. These concepts can be formalized using materials in Chapter C "Statistics", and quantified during lab. Here, having students record the histogram of sensor noise distributions is a valuable exercise.

Chapters 6 and 7, which are on “Vision” and “Feature extraction”, do not need to extend further than needed to understand and implement simple algorithms for detecting the unique features in the maze environment. In practice, these can usually be detected using basic convolution-based filters from Chapter 6, and simple post-processing, introducing the notion of a “feature”, but without reviewing more complex image feature detectors. The lab portion of the class should be aimed at identifying markers in the environment, and can be scaffolded as much as necessary.

In-depth experimentation with sensors, including vision, serves as a foundation for a more formal treatment of uncertainty in Chapter 8 “Uncertainty and Error Propagation”. Depending on whether the “Example: Line Fitting” example has been treated in Chapter 7, it can be used here to demonstrate error propagation from sensor uncertainty, and should be simplified otherwise. In lab, students can actually measure the distribution of robot position over hundreds of individual trials (this is an exercise that can be done collectively if enough hardware is available), and verify their math using these observations. Alternatively, code to perform these experiments can be provided, giving the students more time to catching up.

The localization problem introduced in Chapter 9 is best introduced using Markov localization, from which more advanced concepts such as the particle filter and the Kalman filter can be derived. Performing these experiments in the lab is involved, and is best done in simulation, which allows neat ways to visualize the probability distributions changing.

The lecture can be concluded with “EKF SLAM” in Chapter 11. Actually implementing EKF SLAM is beyond the scope of an undergraduate robotics class and is achieved only by very few students who go beyond the call of duty. Instead, students should be able to experience the workings of the algorithm in simulation, e.g., using one of the many available Matlab implementations, or scaffolded in the experimental platform by the instructor.

The lab portion of the class can be concluded by a competition in which student teams compete against each other. In practice, winning teams differentiate themselves by the most rigorous implementation, often using one of the less complex algorithms, e.g., wall following or simple exploration. Here, it is up to the instructor incentivizing a desired approach.

Depending on the pace of the class in lecture as well as the time that the instructor wishes to reserve for implementation of the final project, lectures can be offset by debates, as described in Section 17.3.

17.2: An Introduction to Autonomous Manipulation

Although robotic manipulation is a much less mature field than autonomous mobile robots, teaching its basics, such as those treated in this book, is slightly easier, mainly due to the fact that concepts like uncertainty and non-holonomy are mostly absent. Robotic manipulation is also well suited for a practicebased curriculum due to the wide array of cheap, multi-DOF robotic arms. These, of course, quickly reach their limitations to demonstrate advanced topics such as dynamics or force control, which are beyond the scope of this book.

17.2.1. Overview

A manipulation-driven curriculum can be motivated by a “grand challenge” task such as robotic agriculture, robotic construction or assisted living, all of which have a manipulation problem at their core. Although a class project is likely to be limited to a toy-example, taking advantage of modern motion-planning frameworks and visualization tools, e.g. ROS/Moveit!, makes it easy to put the class into an industry-relevant framework and expose the students to state of the art platforms in simulation.

17.2.2. Materials

Possible class project range from “robot gardening” or “robots building robots”, for which setups can easily be created. These include real or plastic cherry tomato or strawberry plants and robotic construction kits such as Modular Robotics “Cubelets”, which easily snap together and have the advantage to form structures that are robots themselves, adding additional motivation. The robot arm, such as the open-source, 7-DOF CLAM arm, can be mounted on a portable structure that contains fixed a set of fixed (3D) cameras. In order to allow a large number of students to get familiar with the necessary software and hardware, the instructor can provide a virtual machine with a preinstalled Linux environment and simulation tools. In particular, using the “Robot Operating Systems” (ROS) allows recording so-called “bag”-files of sensor values, including entire sequences of joint recordings and RGB-D video. This allows the students to work on a large part of the homeworks and project preparation from a computer lab or from home, maximizing availability of real hardware.

17.2.3. Content

The first two weeks of this curriculum can be mostly identical to that described in Section E.1.3. If a message passing system such as ROS is used, a good exercise is to record a histogram of message passing times in order to get familiar with the software.

In Chapter 3, the focus is instead on manipulating arms, including the Denavit-Hartenberg scheme and numerical methods for inverse kinematics. In turn, the topics “Forward Kinematics of a Differential Wheels Robot”, and “Inverse Kinematics of Mobile Robots” do not necessarily need to be included. Forward and inverse kinematics can be easily turned into lab sessions using Matlab/Mathematica, simulation or a real robot platform. If the class uses a more complex or industrial robot arm, an alternative path is to record joint trajectories in a ROS bag and letting the students explore this data, e.g., sketching.

17.3: Class Debates

Class debates are a good way to decompress at the end of class and require the students to put the materials they learned in a broader context. Student teams prepare pro and contra arguments for a statement of current technical or societal concern, exercising presentation and research skills. Sample topics include *Robots putting humans out of work is a risk that needs to be mitigated; Robots should not have the capability to autonomously discharge weapons / drive around in cities (autonomous cars);* or *Robots need to be made from components other than links, joints, and gears in order to reach the agility of people.*

The students are instructed to make as much use as possible of technical arguments that are grounded in the course materials and in additional literature. For example, students can use the inherent uncertainty of sensors to argue for or against enabling robots to use deadly weapons. Similarly, students relate the importance and impact of current developments in robotics to earlier inventions that led to industrialization, when considering the risk of robots putting humans out of work.

Although suspicious as first, students usually receive this format very well. While there is agreement that debates help to prepare them for the engineering profession by improving presentation skills, preparing engineers to think about questions posed by society, and reflecting up-to-date topics, the debates seem to have little effect on changing the students' actual opinions on a topic. For example, in a questionnaire administered after class, only two students responded positively. Students are also undecided about whether the debates helped them to better understand the technical content of the class. Yet students find the debate concept important enough that they prefer it over a more in-depth treatment of the technical content of the class, and disagree that debates should be given less time in class. However, students are undecided whether debates are important enough to merit early inclusion in the curriculum or to be part of every class in engineering.

Concerning the overall format, students find that discussion time was too short when allotting 10 minutes per position and 15 minutes for discussion and rebuttal. Also, students tend to agree that debates are an opportunity to decompress ("relaxing"), which is desirable as this period of class coincides with wrapping up the course project.

Index

D

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Overview

Title: Introduction to Autonomous Robots (Correll)

Webpages: 107

Applicable Restrictions: Noncommercial

All licenses found:

- CC BY-NC 4.0: 91.6% (98 pages)
- Undeclared: 8.4% (9 pages)

By Page