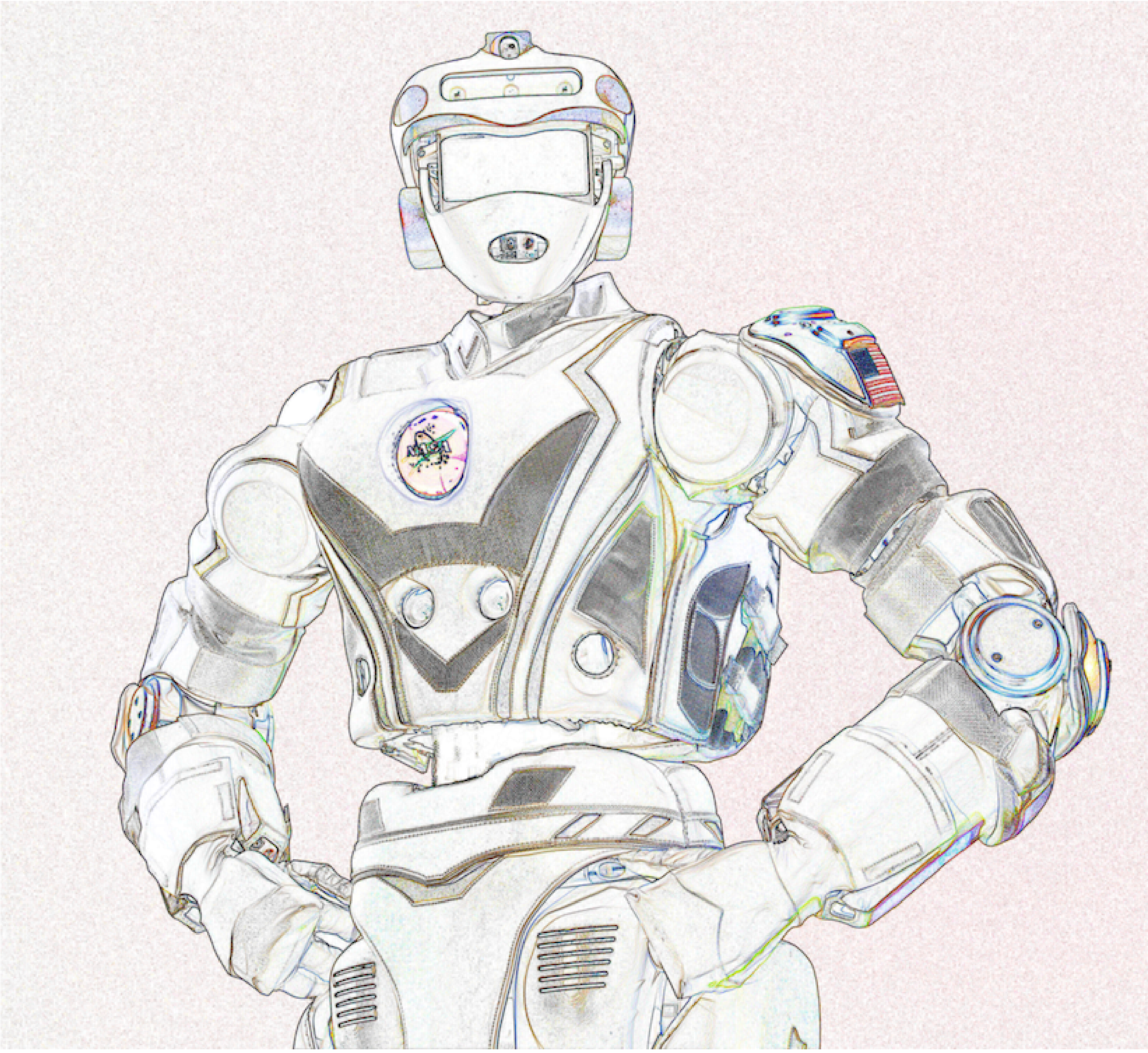


Robotics

Release 1.4

Jeff McGough

Sept 22, 2019

PRELIMINARIES

PREFACE

Historically, the academic study of robotics has been limited to the Mechanical Engineering or Electrical Engineering departments. This makes perfect sense if the target field is manufacturing. Often requiring significant power, reliable repetition or precise positioning, these departments focused on machine design, controls and targeted instrumentation. Leaving the factory floor, however, requires a great deal more in generalized sensing and machine ability. It demands powerful computing and far more advanced algorithms. To address this need, the Computer Science department at the South Dakota School of Mines and Technology decided to include Robotics as part of the Computer Science curriculum. We developed a course devoted to robotics. The original course began with the text *Introduction to Autonomous Mobile Robots* by Siegwart and Nourbakhsh [SN04] . It is a really good survey text. The issue that arose was that we found ourselves having to fill in details, add content, provide more current examples, provide information on frameworks and enhance focus on software. The slides, notes and handouts ended up growing into this text.

The word “robotics” encompasses many different meanings and fields. This is very different from a subject like Calculus, for example, for which all the books written in the last two centuries cover the exact same material - nearly lining up in chapter and section numbers. Not so with robotics. This subject touches on all of the engineering disciplines, mathematics, computer science (if this was not on your engineering list) and several of the sciences. It can be aimed at children, hobbyists, engineers, researchers, managers, evil scientists and many more. Finding a textbook that fell at the right level and contained the specific subjects that we wanted turned out to be a difficult task. There are many, many very great books on the subject, but none that addressed the needs of juniors and seniors studying computer science.

This text exists to present the particular subjects we wanted to cover and relate them at the level that is appropriate for our students. Our course is aimed at computer science students who have had three semesters of Calculus, a semester of differential equations, a semester of linear algebra, exposure to computer hardware, and significant experience with software development. This book will take a computer science perspective and a software focus. The ordering of the material is based on the typical interests and goals for someone who is a senior in computer science wanting to learn about controlling mechatronic systems.

0.1 Acknowledgements

This text would not exist if it were not for the support of many individuals. The author extends his thanks for the support by the South Dakota School of Mines and Technology, Mathematics and Computer Science Department, specifically, Dr. Kyle Riley, Dept Head. Several individuals helped significantly in their day to day efforts to support software and chapter development: Scott Logan, Caleb Jamison, Chris Smith, Remington Bullis, Joe Lillo, Lisa Woody and Kali Regenold. I would also like to thank Stephanie Athow, Kelsey Bellew, Paul Blasi, Julian Brackins, John Brink, Andrew Carpenter, Michael Cerv, Marshall Gaucher, Yun Gwon, Lawrence Hoffman, Travis Larson, Scott Samson, Derek Stotz, Donovan Torgerson and Kyle Macmillan for feedback and editing assistance.

0.1.1 Student Notes

These notes are intended for both formal students and casual readers. In 2009, SDSMT Computer Science offered a Mobile Robotics course for the first time. As mentioned above, we used *Introduction to Autonomous Mobile Robots* by Siegwart and Nourbakhsh [SN04] . This book has evolved from course notes. There is a significant lack of textbooks on robotics for those individuals who approach the subject with a computing background and interest. This is not suprising since the bulk of the effort in robotics over the last 50 years has been with the design and control of the mechanical and electrical systems.

It is only recently that the embedded computing and the supporting software has been powerful enough to attempt to deal with the complexity found in unstructured environments outside manufacturing. The landscape is changing and with the advent of self-driving cars and other autonomous systems, the need for computer scientists in robotics is accelerating. This book attempts to balance the competing forces of presenting a complete thorough coverage with the normal constraints on time and resources. Robotics covers many different disciplines. Some specific techniques can require a great deal of mathematics or other specialized skill. Our goal is to get you up and running as quickly as possible without sacrificing the core concepts needed to progress later. We will iterate through the subject preceeding deeper on each pass.

With this approach we can get you running with a robotic system without months of theory. But we will come back and build on what you have learned. A bit like an Agile method applied to your learning. In past versions of the course, we have separated material. So, we would do all of the mechanical content, then all of the electrical content, then algorithms, tools and so forth. This approach was well criticized on Medium.com with a burrito analogy. Imagine having the burrito with all of the beans on one end, then a section of only meat, a section of cheese and the other end was the sauce and seasoning. Not very appealing. Our goal is, when it makes sense, to "mix it up". Not only does this avoid boredom, it should help with understanding how to integrate the concepts.

We have developed the chapters to be as independent as possible. So, one could read the chapters out of order. However, we have a progression that has developed over a decade presenting the concepts and we think this will be the most effective for those who want a more complete understanding of the subject. In addition, there is an iterative approach to trying to get you up and running quickly, so jumping around may extend the learning time. So, we suggest that you consider reading the chapters in the order presented.

A common question is "what background is really needed?" We will be using algorithms and software to solve problems. So, you need to know how to program. Language is not important since once you know one language it is very easy to learn another. This book uses Python to demonstrate concepts. Python is really easy to learn and use. For our purposes it is plenty fast enough and is the choice for robotics prototyping. The algorithms will employ certain data structures which if you don't have a course in data structures can be understood after a good second programming course.

There are times when we need to model the robot's electrical or mechanical systems. These models are based in calculus and physics. You should have a year of calculus and a year of physics. Ideally you would have seen a course in probability and a course in linear algebra. This is not the case for everyone and so we have the essential math covered in the appendix.

0.1.2 Course Instructor Notes

This course began in 2009 as mixed senior undergraduate and first year graduate student offering. Very quickly it became clear that most of the challenge was in the breadth of material and skill set of the students.

Since then I have proceeded to make every mistake possible with this class. Fortunately at a small school we have forgiving students. This book, specifically the content and organization, is a result of the decade of running the class and correcting those errors.

The goal of this text is several fold. First, there are limited offerings in this space. You can find K-12 materials on robots, especially with the Lego system, easily on the internet. There is a ton of hobbyist sites (Arduino, Raspberry Pi, etc) that are fabulous. The problem is that they lack the completeness and rigor needed for a college course. Since robotics is a research area, finding graduate level references is not hard, but we have the opposite problem of that being too difficult or not sufficiently general in scope. [And honestly, even some undergraduate texts outside your personal area of expertise can feel like a graduate text.]

The second goal was to create a text aimed at students with more of a computer science background and not Mechanical/Electrical Engineering background as with most of the college level texts available. In addition, we are working to expose these students to a diverse curriculum since it is just not reasonable to get three or four Bachelor degrees to be a well schooled roboticist.

So, this text attempts to survey the field, provide sufficient depth for the student and cover current industry tools, all while keeping the course engaging to students. We have worked to balance theory with practice since it seems to help the learning and retention process. There is some mixing of topics to keep student interest. Earlier versions of the course were partitioned over fields and application. Students liked having material mixed and then progressing in level. So, because of this, the ordering of chapters might not be as expected. We have worked hard to get the student up and running in virtual as well as real robots as soon as practical. Some theoretical (important) material is left to later in the text. One benefit is that when students try the simple approaches, and those fail, they more easily understand the more advanced and mathematical approaches later.

It is easier to limit or focus the students in the class. However, a diversity of students makes for a better course. The problem then is that you cannot assume all students have the same background. In the version we offer at SDSMT, we have Data Structures and Differential Equations as background. However, we want students from Mechanical and Electrical Engineering to participate and at SDSMT they normally don't take Data Structures. Likewise, many Computer Science students will not have Differential Equations in their background. We normally meet and make sure they understand what elements that they might need to make up. Then sign them in. This works well for us.

We run a one semester course. There is more material here than a three or four credit semester course (15 week) can address. Now that the text has been open sourced and placed online, we expect the text to grow. There are a couple of directions we can proceed. One is to fork the text repo and build multiple textbooks for different applications (such as student levels or backgrounds, or goals). Another is having a modular system which the instructor selects sections/chapters and have Sphinx build a custom text. If you have thoughts on this or would like a custom text built now, please contact Jeff McGough, jeff.mcgough@sdsmt.edu .

ABOUT THIS TEXT

Robotics is an engineering discipline stemming from the fusion of science and art. One of the most interesting and exciting aspects of robotics is how many different fields it touches. At some institutions, for example Worcester Polytechnic Institute (WPI), it is now a undergraduate degree program. Many courses can be taught within the field to cover the different aspects just like what we do with Electrical or Mechanical Engineering. So any attempt at a survey course is challenging. This book is not a really a great survey of the field. For such a large field, a survey would be a long list of titles, authors, researchers and so on. There would be no room to "get down into the weeds". We want the student (reader) to also get in the detail and acquire real skills. To do that, we must focus. So, the book is oriented around mobile robots. You will see that our coverage of manipulators is very thin. Fortunately, there are fabulous texts on manipulators available.

Our goal is to get the reader to the point where they can build a simple mobile robot which can interact with the environment. This requires covering basics of sensing, computation and motion. Our target will be the creation of autonomous vehicles.

There are many issues to resolve with this book. Many small ones and some large ones. The obvious problems relate to content, coverage, background and other academic concerns. Smaller ones relate to formatting, specific examples and images.

0.2 Copyright concern

This book came from course notes. The notes were restricted to class and as such subject to *Fair Use* guidelines. Materials from the web such as images were used as is. Going to publication (either print or online) brings a different set of rules. We have gone to great effort to make sure that all of the content is legally used. Meaning that externally derived content is in the public domain or has a "use with attribute" style license.

Note: If you see something that violates a use agreement, please contact us immediately: jeff.mcgough@sdsmt.edu

0.3 Conversion Issues

The original text was written in LaTeX. We have converted over to restructuredText using a fabulous tool called Pandoc. Pandoc is good, but not perfect. We had a very complicated latex document which Pandoc was not able to completely convert. Although we have spent many hours with the conversion, there are aspects we have missed. The labels and references were a particular problem and many are still broken. Let us know about format problems (broken links, etc).

0.4 Document Content

This is a living document. There are many sections of the original notes that need to be converted over to to rst AND there are many sections that need to be written. Robotics is a active field with new technology daily. New stuff needs to be written and some old tools need to be updated. We hope to get a bunch of this done by 9/1/18.

0.5 Robotics Education Community

Our goal is to build an open source robotics education community. If you want to help edit, add sections, add homework problems, rework sections, etc, please contact us. If you have comments or concerns, again please contact us.

CHAPTER ONE

INTRODUCTION

Growing up in the modern age means many things to many people. For those of us reading (or writing) a book on robotics, it means getting a healthy dose of technology. Hollywood has raised us on spaceships, cloning, alien worlds and intelligent machines. This was, however, not incredibly unrealistic as progress in science and technology has been advancing at an ever increasing rate. We have come to expect something new, maybe even dramatic, every single year…and for the most part haven't been disappointed! Over the years we have seen significant developments in medicine, space, electronics, communications and materials. There has always been excitement regarding the latest development. Even though the world around us has been struggling with war, poverty and disease, science and technology offers us a reprieve. It is an optimistic view that things can and will get better. In full disclosure - we love technology! It is the magic of our age and learning how the magic works only makes it more fun. In no way does the author claim that technology is the answer to our problems. That clearly lies with our willingness to look beyond our differences with acceptance, compassion and grace. If technology can bring us together, then it has succeeded in helping us far beyond our wildest dreams.

Robotics is a shining example of technical optimism - a belief that the human condition can be improved through sufficiently advanced technology. It is the premise that a machine can engage in the difficult, the tedious, and the dangerous; leaving humans out of harms way. Technology is a fancy word for tool utilization. Even though biologists have long shown that humans were not alone in their usage of tools, we are indisputably the master tool users on the planet. Tools extend our grasp, our strength and our speed. We know of no technology that aims to extend human capability like robotics does.

Humans have an insatiable curiosity, a drive to create, and a considerable amount of self-interest. Building machines which look like us, act like us and do things like us was the engineering manifest destiny. Although we have succeeded in building machines that do complicated tasks, we really place the value in what we learn about ourselves in the process. A process we embark on here.

1.1 What exactly is a robot?

1.1.1 Definition

What is a robot? This is actually a complicated question. Wikipedia defines a robot in the following manner: *A robot is a mechanical intelligent agent which can perform tasks on its own, or with guidance; usually an electro-mechanical machine which is guided by computer and electronic programming.* (There are plenty of opinions on Wikipedia. I find that it is pretty good for math, science and engineering quick reference but not always an expository presentation. It is also good at reflecting opinions, which in this case is useful.)

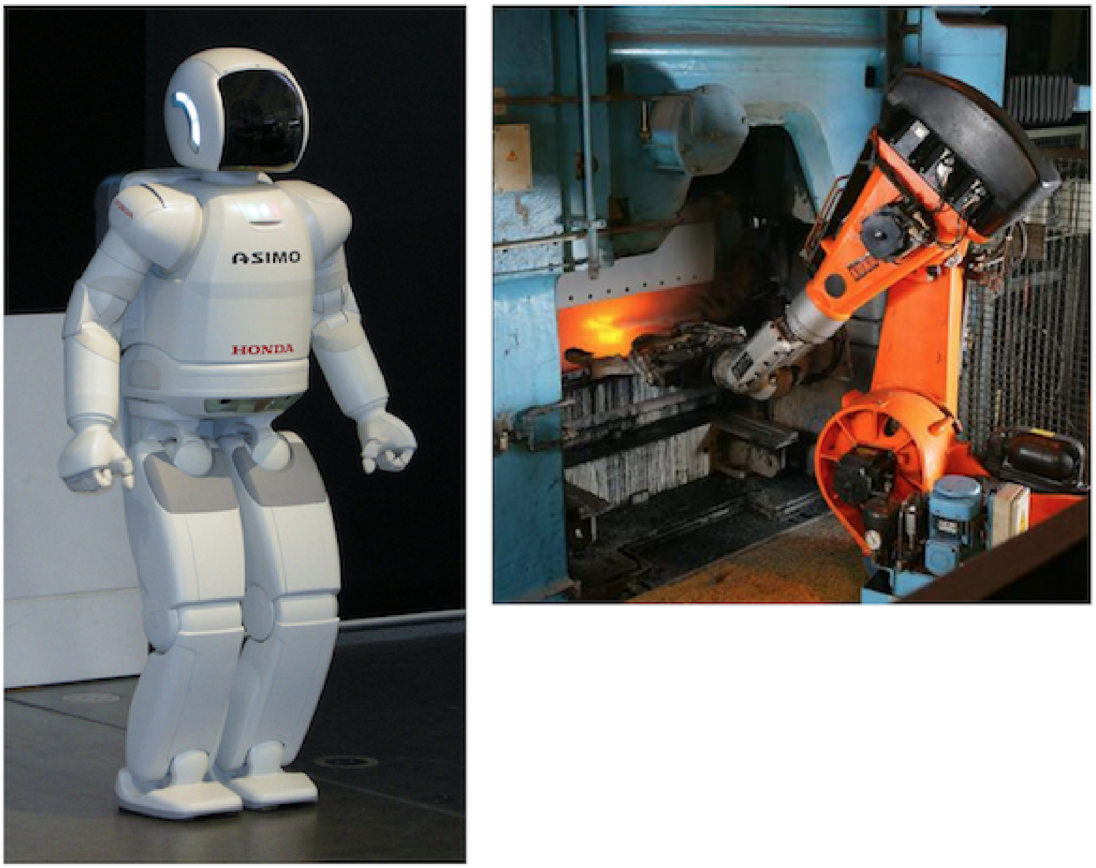


Fig. 1.1: Two different versions of robots. a) Honda Asimo Robot [Wik18c] b) Kuka robot [Wik18d]

Merriam-Webster, on the other hand, says a robot is *a real or imaginary machine that is controlled by a computer and is often made to look like a human or animal.* According to the Encyclopaedia Britannica, a robot is *any automatically operated machine that replaces human effort, though it may not resemble human beings in appearance or perform functions in a humanlike manner*.

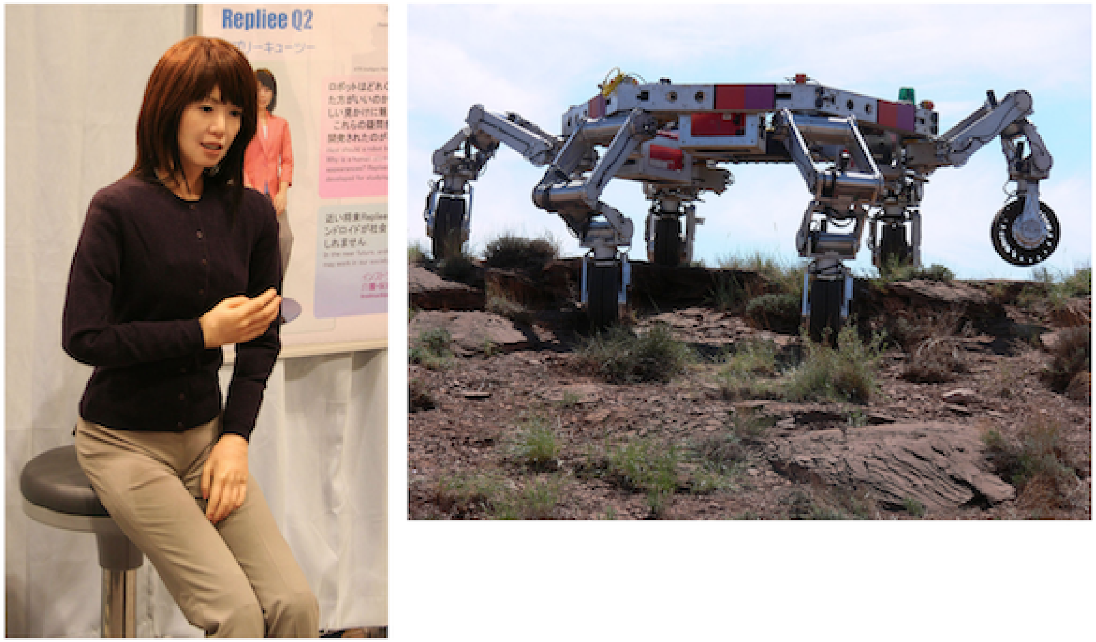


Fig. 1.2: Robots design to replace human effort. a) Repliee Q2 - developed by Osaka University and manufactured by Kokoro Company Ltd. [Bra18] . b) The NASA ATHLETE rover as it climbs to the top of a hill. [Wik18a]

This latter definition includes washing machines, bread makers, and other devices not generally seen as a robot. However, as we will argue, that does not matter! A definition of robot that uses form or motion is flawed. What if we made the statement broader? **A robot is seen as a sophisticated machine that, as stated above, replaces human effort**. Nothing else really defines robotics as well.

Regardless of definition, these machines surround us. Today we can see them used from manufacturing to exploration, from assistive technologies and medicine to entertainment, from research to education, and much more.

There is no consensus on which machines qualify as robots. However, there *is* a general agreement that robots exhibit behaviors which mimic humans or animals - that is, *behavior which seems intelligent.* We expect the robot to interact with its environment and the objects within that environment. Most of us may expect that the robot performs this interaction through movement and sensation.

Many may expect the robot to perform complex tasks or deal with harsh, unforgiving environments. Some may expect a robot to be an extension of themselves through teleoperation or remote control, while others expect it to be a fully autonomous device.

We can boil down our notion of robot abilities to three things:

Perception: sensing the environment and to a limited degree understanding the sensory information.

Cognition: ability to make decisions and responses based on the sensory information and not acting in a

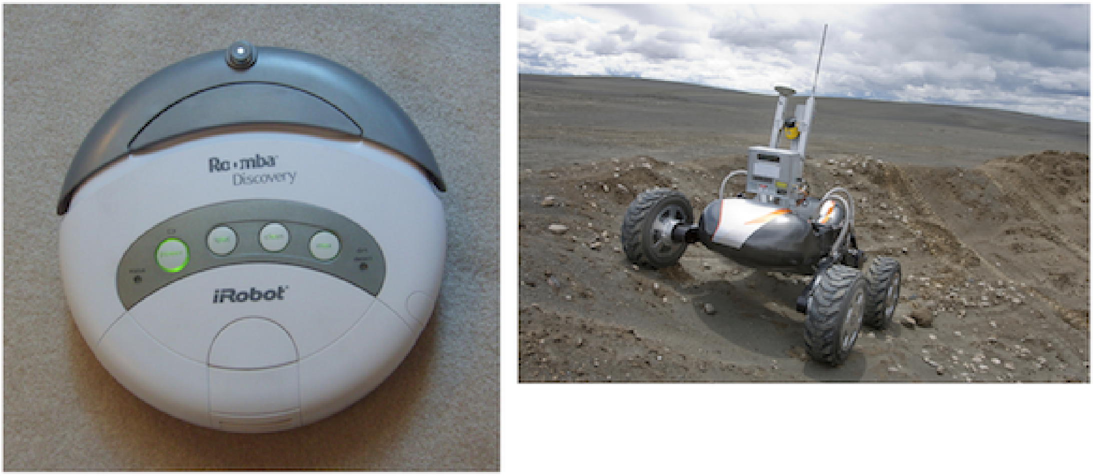


Fig. 1.3: More examples of assistive robots. a) iRobot Roomba Discovery 2.1. [Com18d] b) NASA experimental drilling robot. [NAS18a]

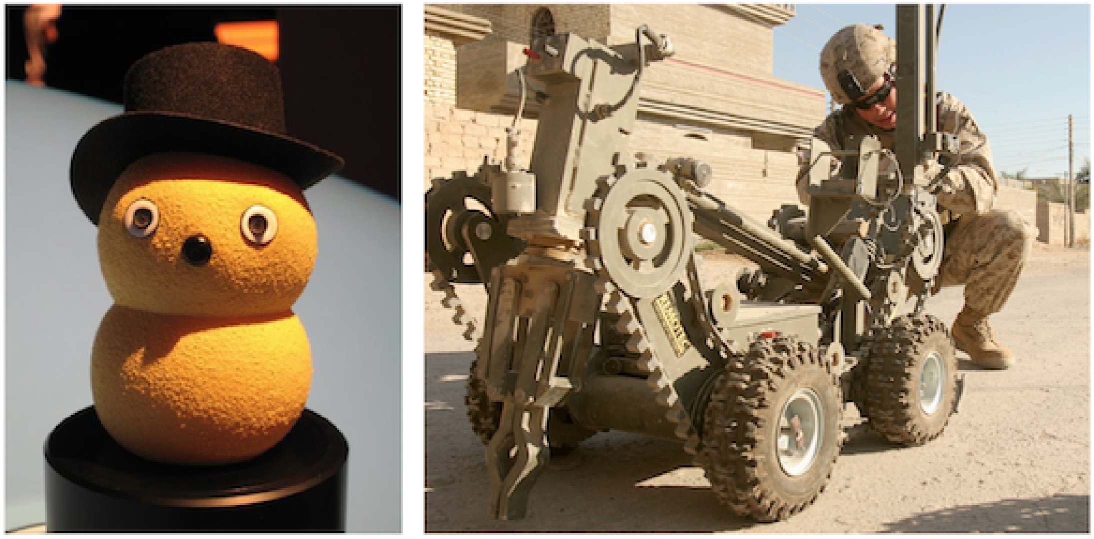


Fig. 1.4: There are a wide range of roles for robots. a) Keepon - therapy robot. [Com18b] b) Robot tasked to detonate a buried improvised explosive device. [Com18a]

pre-programmed manner.

Actuation: full mobility of the machine or control of a tool through a manipulator.

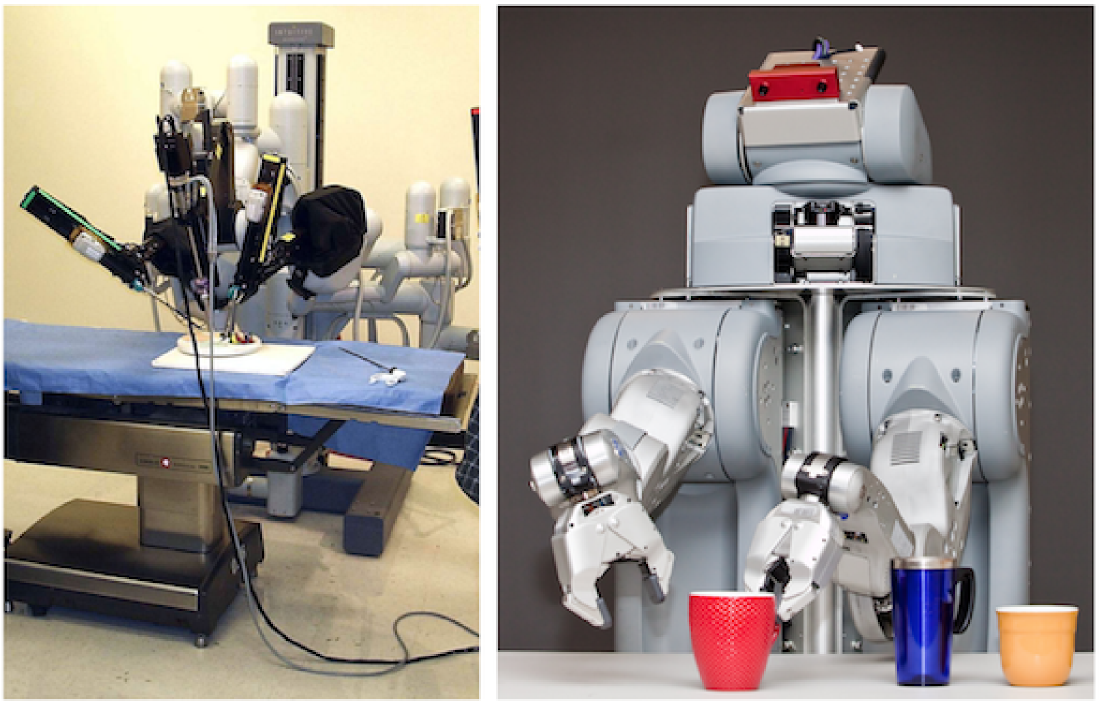


Fig. 1.5: Systems which focus on manipulation: a) Da Vinci Surgical System [Nim18a] . b) Willow Garage's PR2 robot. [Nim18b]

One interesting phenomenon that could be influencing the lack of a solid definition for the term is that what we label a "robot" varies with time. When a new capability arises, one that was previously considered to be solely in the domain of humans and animals, we tend to label it a robot. As soon as that capability becomes routine, the device is thought of a mechatronic device.

Robots embody technological magic. So, it is natural that some previously unseen ability programmed into a machine will have a magic quality for humans, thus making that machine more of a robot. But with time, we get accustomed to it, and the magic gets replaced with expectation.

It can be argued that there is nothing new in the subject of robotics - that all we are doing is building machines. Nothing different than what engineers have been doing all along. The term robotics has more to do with our ego and psychology than anything to do with science and technology. However, there is a body of knowledge related to building machines that interact in human or physical environments. This is what we will consider robotics.

1.1.2 A brief history

1023 BC

In ancient China, a curious account on automata is found in the Lie Zi text, written in the 3rd century BC. Within it there is a description of an encounter between King Mu of Zhou (1023-957 BC) and a mechanical

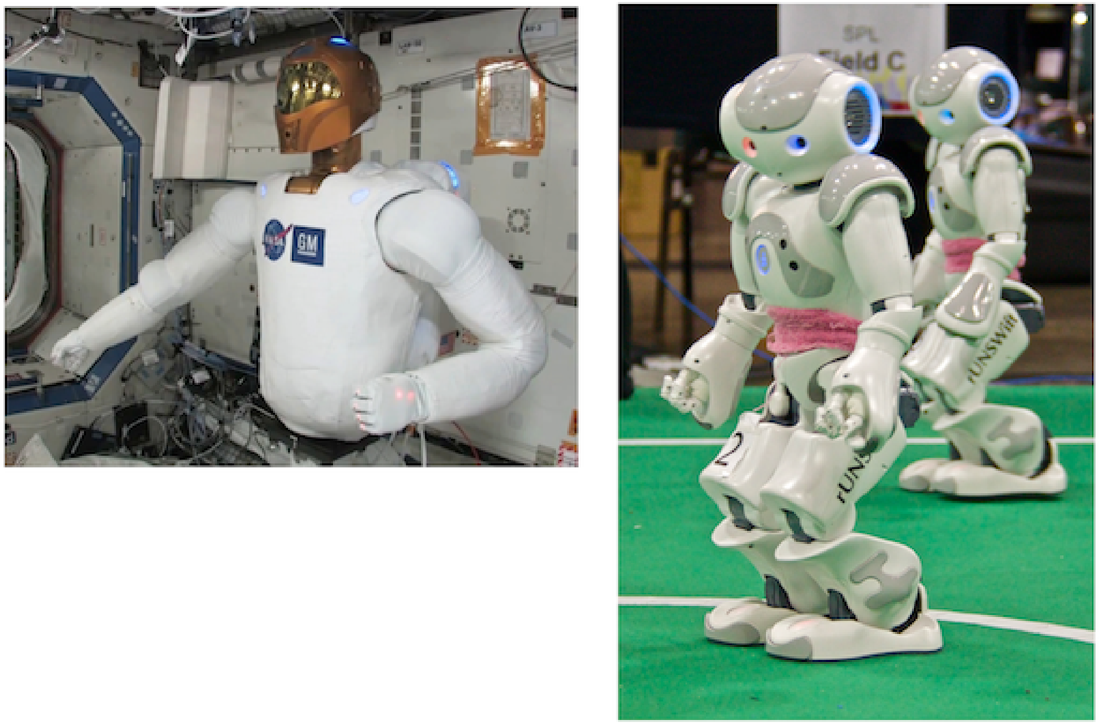


Fig. 1.6: Systems which focus on mobility: a) NASA’s Robonaut. [NAS18b] b) RoboCup Standard Platform League (Image from 2010). [Com18c]

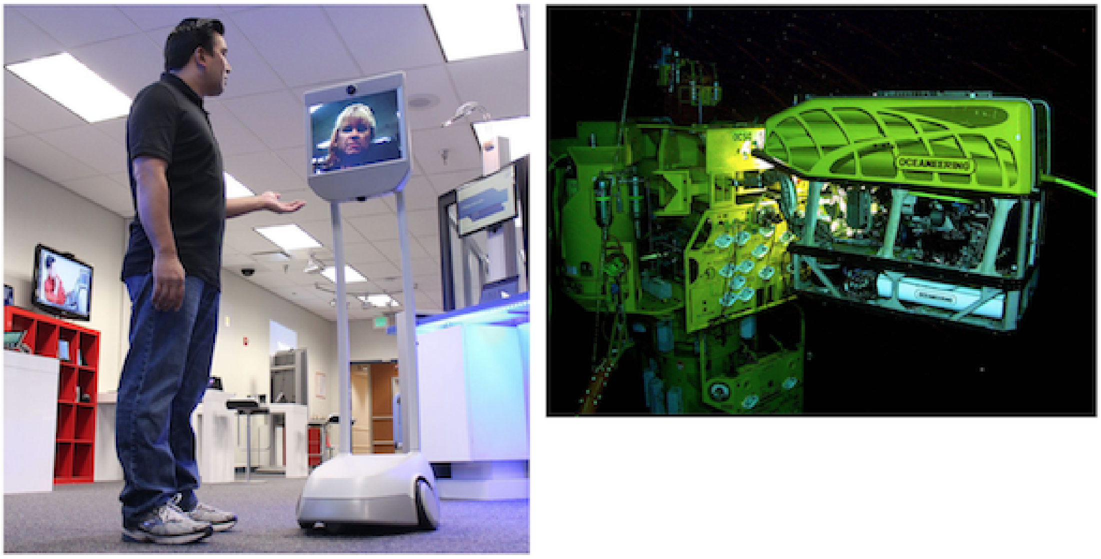


Fig. 1.7: Telepresence or remote work is a growth area in robotics. a) An Intel IT Labs researcher working on a remote telepresence robot pilot project that uses Suitable Technologies’ Beam robot. [Com18e] b) ROV working on a subsea structure. [Com18f]

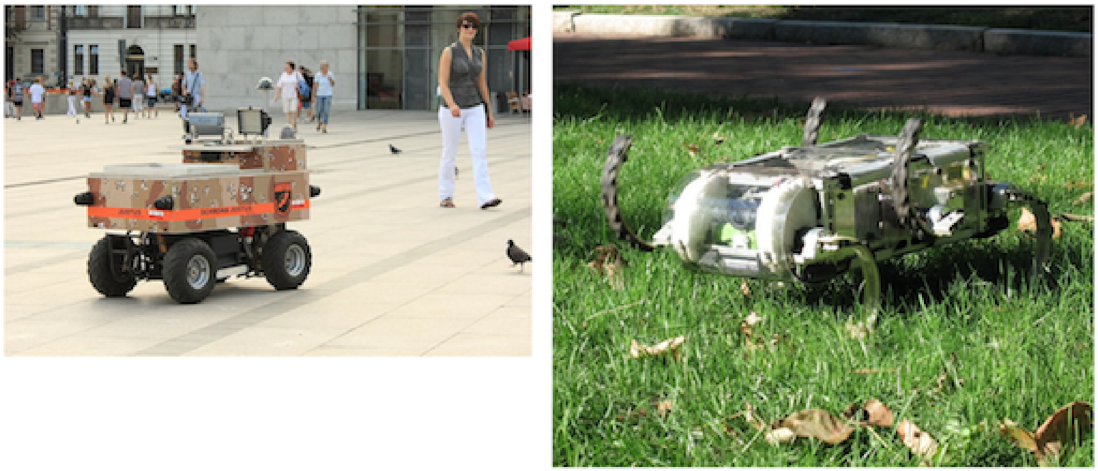


Fig. 1.8: Mobility in simple and complex domains. a) Justus security robot in front of Krakow railway station [MP18]. b) Rhex: DARPA project on compliant six legged robots. [Wik18e]

engineer known as Yan Shi, who was an ’artificer’. According to the text, the artificer proudly presented the king with a life-size, human-shaped figure of mechanical handiwork which could sing and move in a life-like manner.

205 BC

In ancient Greece, an orrery known as the Antikythera Mechanism is developed. This device is credited as being the first analog computer.

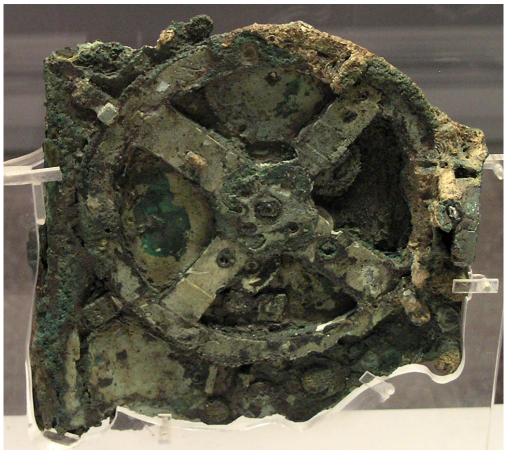


Fig. 1.9: Antikythera Mechanism [Eft18].

270 BC

The Greek engineer Ctesibius (c. 270 BC) applies a knowledge of pneumatics and hydraulics to produce the first organ and water clocks with moving figures.

1088 AD

The Cosmic Engine, a 10-meter (33 ft) clock tower built by Su Song in Kaifeng, China. It featured mechanical mannequins that chimed the hours, ringing gongs or bells among other devices.

1206 AD

Al-Jazari (1136-1206), an Arab Muslim inventor during the Artuqid dynasty, designed and constructed a number of automatic machines, including kitchen appliances, musical automata powered by water, and the first programmable humanoid robot in 1206. Al-Jazari's robot was a boat with four automatic musicians that floated on a lake to entertain guests at royal drinking parties. His mechanism had a programmable drum machine with pegs (cams) that bump into little levers that operate the percussion. The drummer could be made to play different rhythms and different drum patterns by moving the pegs to different locations.

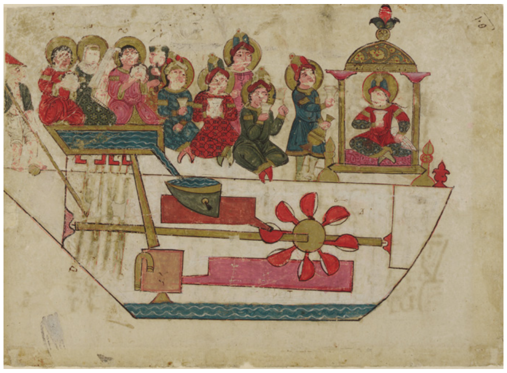


Fig. 1.10: Al-Jazari's Mechanical Musical Boat. [Wik18b].

1495

Leonardo da Vinci draws plans for a mechanical knight.

1922

The word *robot* is introduced to the English language through the play Rossum's Universal Robots by the Czech writer Karel Capek. The play is centered around a factory staffed by intelligent cyborgs. The English term robot comes from the Slavic word *robota* which roughly translates as work or labor. Credit for the term goes to Karel's brother Josef.

1954

Following World War II, efforts in automation increased. Early advances were seen in teleoperation and computer numerically controlled (CNC) machining. General Electric produced machines that had a master slave approach where the master manipulator would control the slave. The CNC machines gained popularity in the aircraft industry by milling high performance parts in lower volumes. The merger of these two technologies produced the first programmed articulated device by George Devol in 1954. He replaced the master manipulator with CNC technology. Joseph Engelberger purchased the rights and founded Unimation in 1956. Unimation placed its first robot arm in a General Motors plant in 1961.

1969

The 1960's saw significant experimentation with manipulator designs, feedback systems and actuator types. One such example of a robotic manipulator is the Stanford Hydraulic Arm and Stanford Manipulator, designed in 1969 by Victor Scheinman, a Mechanical Engineering student working in the Stanford Artificial Intelligence Lab (SAIL).

1973

The Cincinnati Milacron T^3 is released. It was a heavy lift assembly line manipulator. In 1978, Unimation introduced the PUMA, (Programmable Universal Machine for Assembly) and JPL started a research program to develop space based teleoperated manipulators. By the late 1970's, applications for industrial robots grew quickly and robots in industry became established.

The history for mobile robots is much more recent. The challenges for mobile robots, as we will see later on, are fundamentally different than industrial automation. An early example is the Johns Hopkins *Beast*. It was a simple autonomous mobile system that navigated using touch sensors and could recharge itself. This system required an instrumented environment. A notable development is *Shakey*, by the Stanford Research Institute (SRI) from 1966-72. This robot implemented computer vision and natural language processing and is responsible for the development of the A* search algorithm, the Hough transform, and visibility graphs.

1.1.3 Robots in the news

Items are hyperlinked to web pages.

2017

- Tertill (Franklin Robotics) - Fully autonomous weeding robot.
- Minitaur (Ghost Robotics) - Legged version of the Rhex but with enhanced obstacle response.
- Fast Foward. Autonomous delivery robot. Paggio.
- Cobalt Indoor Security Robots. Collaboratory security robots.
- Ekso GT, exoskeleton to assist paraplegics. Ekso Bionics
- Kuri. Home "social" robot. Mayfield Robotics.

2016

- SpotMini, a compact version of Boston Dynamics' Spot robot.
- Pleurobot - experiments in salamander motion through robotics.
- Vyo - Different approach to social domestic robots.

2015

- DRC Hubo - UNLV finished 8th place in the DRC.

- Momaro - experimentation in rescue robots.
- iCub - The iCub is the humanoid robot developed at IIT as part of the EU project RobotCub.
- Walkman Robot - EU humanoid.
- Deepfield Robotics targeting agriculture.

2014

- Robocup 2014: Goal! Although the human team was not really aggressive, the goal was well setup and the defender did try to block the shot.
- Pronking. RHex is used to experiment with new gaits. Pronking is commonly known with the African Springbok and is used to understand very dynamic locomotion.
- Boston Dynamic's descendent of Big Dog is LS3. LS3 is getting field testing for use as ground support for Marines.
- CMU's Biorobotics lab has a new generation of robotic snakes. This one uses elastic actuators for smooth motion.
- Festo announces a robot kangaroo. Why? Well who wouldn't want a kangaroo robot?

2013

- Boston Dynamic's BigDog gets an arm which can throw heavy objects.
- Google's robotic car gets a full test.
- Watch Flying Robots Build a 6-Meter Tower.

2012

- Boston Dynamics announces Legged Squad Support System (LS3) which is a militarized variant of Big Dog.

Our notions about robots are driven by literature, movies and television. The nearly universal images of robots in fiction have driven our expectations and to some degree affected the robots we currently have. The stories present robots in a vast array of situations with a range of technologies. These robots offer a canvas that opens exploration of themes where the characters can have dramatically different abilities or views than human agents. It allows the author to ask big questions about what it means to be human and that of friendship or relationships. It also allows the author to suspend all reality by painting robotics characters as pure evil or immensely powerful giving a backdrop for character growth. But how is this important? It is because the role fiction has played, it, as much as the needs of society and economic forces, influences what we do in robotics.

1.2 An Overview

Robotics as a discipline is often described as an interdisciplinary field constructed from Mechanical Engineering, Electrical Engineering, Industrial Engineering and Computer Science. It is fairly new as an academic area and mostly grew out of Mechanical or Electrical Engineering programs. Previously, various aspects of the robotics trade was found in subjects such as kinematics, dynamics, controls, mechatronics, embedded systems, sensing, signal processing, communications, algorithms and planning.

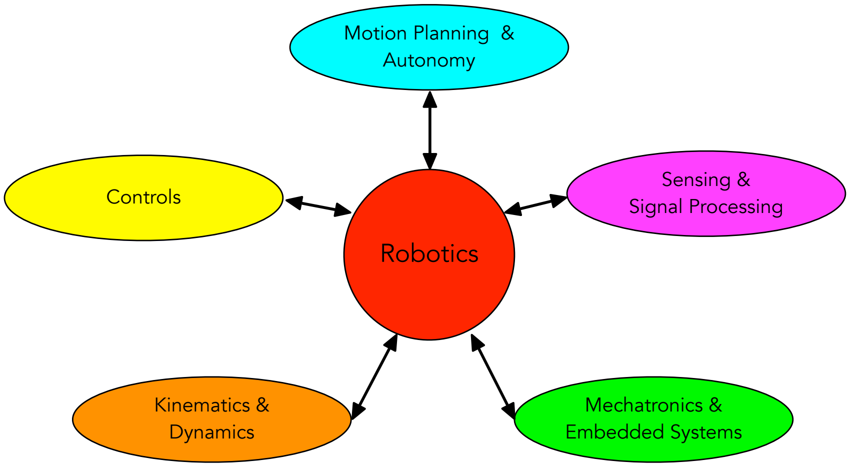

Motion Planning & Autonomy
Controls
Sensing & Signal Processing
Robotics
Kinematics & Dynamics
Mechatronics & Embedded Systems

Fig. 1.11: Robotics is a blend of mechatronics, embedded systems, controls, sensing, signal processing, kinematics, dynamics, communications, algorithms and planning.

Application domains for robotics is a quickly growing list. We are quite used to seeing robots in large industrial settings like automotive manufacturing and palletizing. They have made a name in welding, painting, inspection, product loading, parts placement and a variety of other industrial tasks. Hazardous environments (space, underwater, chemical/nuclear, military) are a significant growth area for mobile robotics. Applications that manipulated radioactive materials, toxic chemicals and other hazards have been prime choices for teleoperated systems since the human operator can be kept safely away. During flu season, the workplace can be considered a hazardous area and telepresence can address the issue. A recent version of a standard teleoperated robot is the surgical robot. This device can follow human motion but scale it down to be effective in regions where human motor control is too crude and dangerous. It is like having a gear reduction in motion leading to more precise and accurate manipulation.

Roboticists often view robots as systems comprised of three components: **Sensors, Software and Effectors**. In other words, there is **perception, cognition and actuation**. One could break a text down into those three major components. Although it has a certain taxonomic appeal, the reality is that these aspects are intertwined and should be studied together. We are finding that in robotics, they must be integrated.

1.2.1 A simplistic taxonomy

To get started, we use a rather crude taxonomy of robots: **mobile robots** and **industrial robots**. The mobile systems are best known through examples like the NASA Rovers and the IED detecting robots of our armed forces. Industrial robots have been in use for a half of a century and are well known in manufacturing and more recently with surgical robots. Typical examples are shown in Figure Fig. 1.12 .

It is important to note that partitioning these machines into two categories ignores the full spectrum of systems available. As the application areas grow, this distinction will vanish. However, it is useful at the moment to illustrate some concepts. Useful in that we are able to isolate various challenges and technologies in existence. Later we will dismiss the artificial categories and look at mobile autonomous systems in unified manner.

Robotics built a name in manufacturing. The ability to repeat a task exactly for thousands or hundreds of

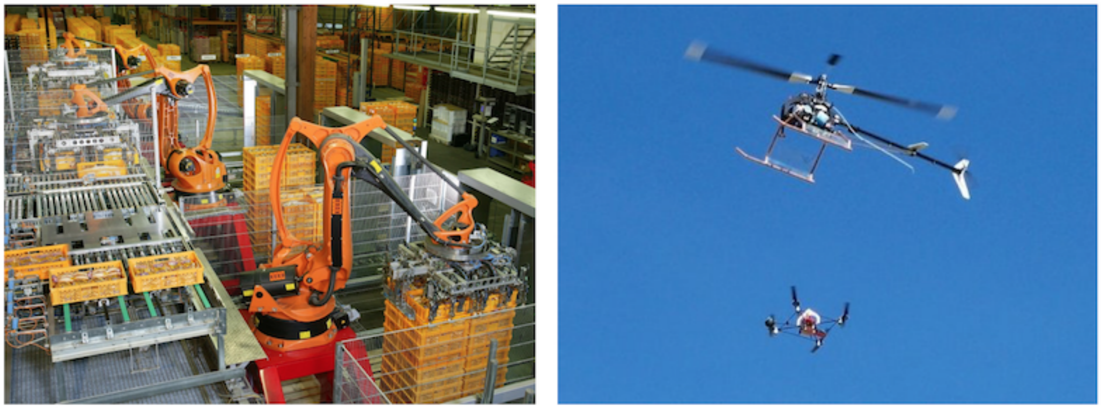


Fig. 1.12: A contrast in uses. a) The traditional factory setting for robotics, Kuka robotics. b) An autonomous helicopter releasing an autonomous quadrotor - SDSMT UAV Team.

thousands of times is essential to take advantage of scale. It enables a market advantage by keeping assembly costs down. This may be due to human labor costs, human speed, human error, human environmental restrictions or some combination.

Thus industrial systems grew out of the need to do a specific task quickly, accurately and cheaply. These systems live in an instrumented and structured environment. The task, the interaction between robot and objects, is understood and predetermined. Highly accurate positioning for tools, exact tool paths and application of specific tool forces dominated the designs.

Contrast this view with the mobile machine. By its very intent, this device leaves the confines of the lab or shop. It moves into new and possibly unexpected environments. Lack of instrumentation outside the lab and lack of pre-determined structure removes any possibility of predetermined interactions. They must be novel and thus requiring a great deal more from the system. The possible types of interactions are enormous and as such the machine must not be specifically programmed, but must be a generalist. Although the precision of interaction and speed of task may be greatly reduced, the increase in complexity for the system in the new untamed world is much more complex. It requires behaviors that mimic intelligence. It is in this arena that computer scientists can contribute best. The contrasting elements are given in Table 1.1.

Table 1.1: Typical aspects of Mobile vs Manufacturing Robots.

Manufacturing	Mobile machines
Dedicated	General
Fixed environment	Changing environment
Predetermined tasks	Adapting tasks
Fixed interactions	Novel interactions

1.2.2 A less simplistic view

The *industrial robot* verses *mobile robot* is one way to partition up the robot design space, but is one that really does not do justice to the vast array of creative designs which have emerged. Robots are machines which help reduce human effort in some manner. We create them to assist us. Understanding robots in terms of how they are used or how we interact with them, although rather human centered, is another way to classify these machines. It is also a way to classify newer systems that don't really fit into one of the two boxes described above.

Take, for example, the new surgical robots. These systems are not mobile. They share many attributes of the industrial robotics designs. However, these systems operate (pun intended) in a vastly dynamic environment since no human is the same. These systems are not performing repetitive tasks but are carefully controlled by the surgeon. A similar issue arises when you examine the current class of telepresence robots. They are not autonomous and are confined to simple office environments. So how should we understand these systems as robots. Or are they?

Let's try a thought experiment. Say you are a surgeon. The scalpel is directly controlled by the surgeon's hands and eyes. That instrument can be placed on a rod to access difficult regions. Maybe a long linked or flexible rod. To see in the hard to access regions, we can place a small video camera. We bundle and run the camera and scalpel through linked rods and cables. Instead of controlling the position of these instruments by hand, we decide to control using servos. Because we are not using our hands to control, we have lost the "feel" of the instrument interacting with the tissue, so we add some types of feedback in the grips. We now have a surgical robot. But where did it cease being a tool and become a robot?

Surgical robots, telepresence robots, and remotely piloted drones all extend human capability. They extend our reach and our senses. They can operate autonomously in the limited sense of physical separation from the human, but not without constant direction. Although they can be very sophisticated, they are automatons or appliances. We will use robotic appliance to describe this class of robots which is an extension of us and not worry so much as to their construction or mobility. Simply that they are not collaborators with us; merely extensions of the pilot. The classic industrial robots, cleaning robots and 3D printers easily fall in this category. Pre-programmed systems extend our work hours by replicating the programmer's first successful (remote) run.

The efforts you see with the PR2 or the Baxter show a different trend. These are robots that are collaborators. They work with us, maybe beside us, but semi-autonomously. This means that they are not simply reflecting our directions, but are adding something to create a team and ultimately something greater than the sum of the parts. These robots are agents acting independently to some degree. Home care robots and autonomous vehicles are two such examples. The rise of robot agents is strictly due to the recent successes in machine learning. It is the new forms of artificial intelligence that are making robotic agents a reality, and appears to be in a rapid growth phase.

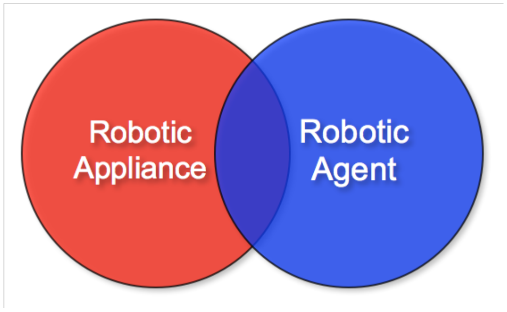


Fig. 1.13: Another way to view robotics.

The value of classifying is to help one understand the landscape. Its utility ceases the moment it restricts innovation. So we will leave the classifications behind us and refer to them only when required.

1.2.3 Electronic components of a small mobile robot

It is useful exercise to open up a small mobile robot and become familiar with the hardware. There has been an explosion in options for robotics. Low cost microcontrollers are immensely powerful. There is an ever growing list of sensors, actuators and support electronics. This is very helpful for the computer scientist since one no longer needs custom equipment to get a mobile system operational. Using USB interfaces, it is possible to connect the various systems just like we do with Legos. (Later we will note that USB may not be the best choice due to electromagnetic interference.) Before we get any further, however, lets go over the basic terms we need to know for this section.

Manipulator the movable part of the robot, often this is the robotic arm.

End Effector the end of the manipulator.

Actuator the motor, servo or other device which translates commands into motion.

Sensor any device that takes in environmental information and translates it to a signal for the computer such as cameras, switches, ultrasonic ranges, etc.

Controller can refer to the hardware or software system that provides low level control of a physical device (mostly meaning positioning control), but may also refer to the robot control overall.

Processor the cpu that controls the system. There may be multiple cpus and controllers or just one unit overall.

Software all of the code required to make the system operate.

Fig. 1.14 shows the basic hardware elements of a typical low cost small mobile robot. We can see sensors, software and effectors in this unit. There are two sensing systems described in Figure Fig. 1.14. The familiar sensor is the Microsoft Kinect. The Kinect is a type of sensor known as a ranger which is any device that provides distance or range information. It also has a built in camera which is integrated with unit. The depth sensor returns an array of distances that are registered with the the pixels in the camera image. This is very useful because you then have a distance approximation for features seen in the image and have both 3D reconstruction and color mapping for a scene.

The second sensor found on this unit is the LIDAR. This is a laser ranging unit. It does a horizontal sweep (the pictured unit sweeps roughly $240°$ arc) and returns the distances along the arc. The LIDAR only returns depth information along the arc so can only give a cross-section of the scene. Placing the LIDAR on a pan or tilt system then can scan a region if required. Many human environments are just extensions of a 2D floor plan into 3D by extending the vertical direction and so a LIDAR is a very useful ranging device.

A camera can be a useful sensor and paired with a second camera the pair can provide depth of field. Stereo vision for robots works on the same principles as stereo vision in humans. Since the Kinect does not operate in sunlight, a stereo camera setup is a cost effective alternate to more expensive ranging equipment. Other inexpensive approaches use a type of sonar. An ultrasonic transducer can send a chirp. Knowing the speed of sound one can determine the distance of an object in front of the sonar unit.

Simple sensing systems can detect touch or impact (bump sensors for example). Sensors are available to measure pressure and force. These are important in manipulation where the object is fragile relative to the robot gripper. There is a vast array of sensors available measure light, radiation, heat, humidity, magnetic fields, acceleration, spin, etc.

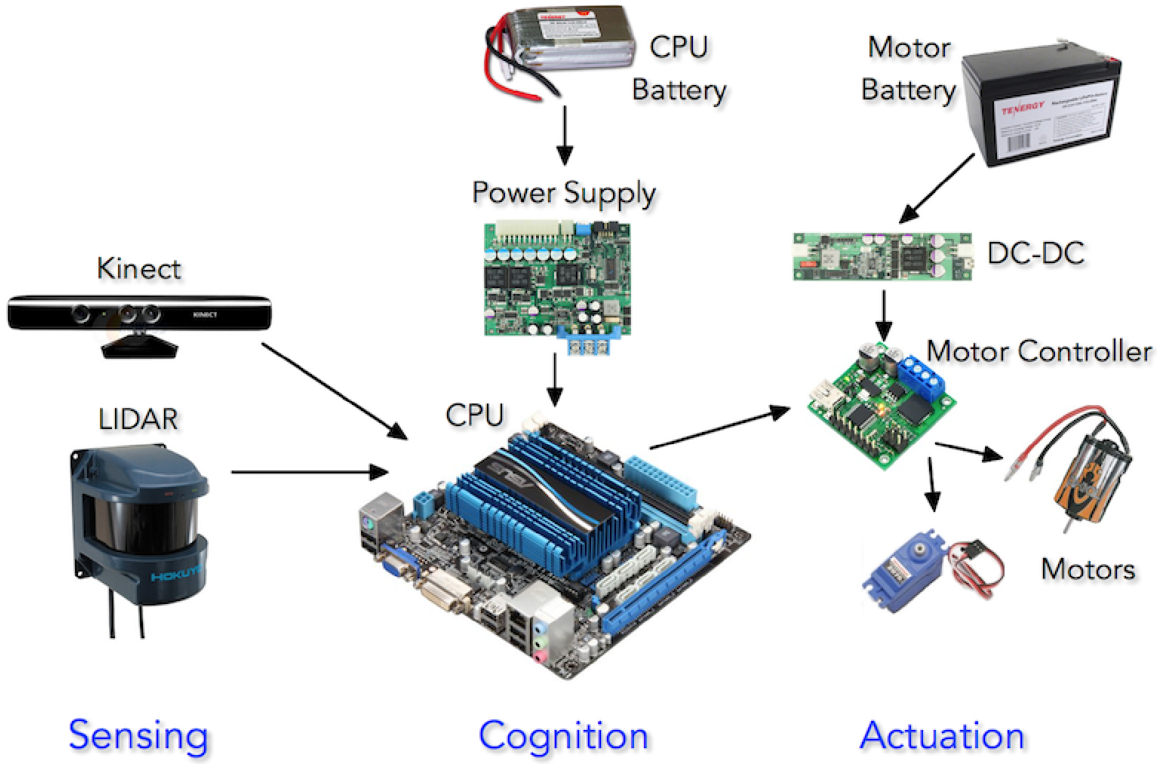


Fig. 1.14: The small mobile robot illustrates the three aspects of robotic systems: sensing, computing and actuating.

1.2.4 Popular Hobby Robots

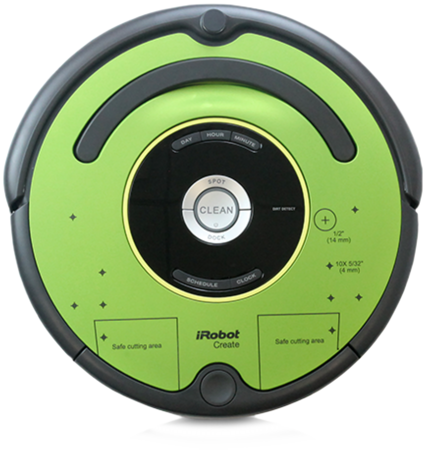


Fig. 1.15: iRobot Create (image from iRobot).

- iRobot Create This robot based on the iRobot Roomba, was introduced in 2007 and has been used as a platform for many ground robotics projects. Most notably is the Create was the base for the ROS based research robot, the turtlebot. In 2014 the Create 2 was released. Unlike the Create, the command module was not updated and made available, the create 2 assumes that the robot will be controlled via an Arduino or Raspberry Pi.

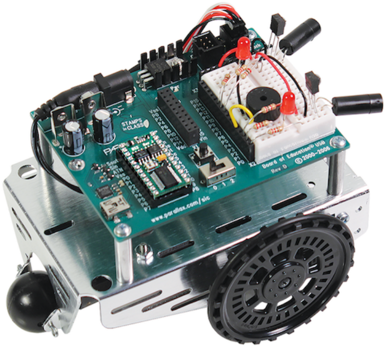


Fig. 1.16: Parallax Boe Bot (image from Parallax).

- Parallax Boe Bot Parallax has been in the educational and hobby robotics business since 1987. The Boe Bot is based on an early electronics education board, BOE (Board of Education). This board used a chip that supported the BASIC programming language. The Boe Bot robot chassis design has been very popular due to its robustness and versatility.

Note: Add additional robots

1.2.5 Autonomy

Autonomy or Autonomous appears quite often in the current press. What does this mean? Dictionary.com will define this as “acting independently or having the freedom to do so”. We should be careful to distinguish autonomous (and probably autonomy) from automated. The root meaning of autonomy is self-governance verses the idea of automated which is “to make automatic”. Although similar in sound, automatic carries the sense of preprogrammed or pre-sequenced. The difference being that autonomy hints at using information from the environment, making decisions to arrive at some goal, but not programmed in a fixed set of actions.

In common usage, we see autonomous and unmanned as inter-changeable. Whether or not a person is involved, the idea is that the system can operate successfully without human guidance. However, a self-driving car is a significant challenge and the industry is looking at partial levels of autonomy as achievable goals in the near term. SAE has released definitions of levels of autonomy for automobiles. This is strictly a characterization for commercially available ground vehicles. These are intended to provide a common set of definitions for the industry. A description of these levels can be found at the NHTSA (<https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety>).

Level 0	The human driver does all the driving.
Level 1	An advanced driver assistance system (ADAS) on the vehicle can sometimes assist the human driver with either steering or braking/accelerating, but not both simultaneously.
Level 2	An advanced driver assistance system (ADAS) on the vehicle can itself actually control both steering and braking/accelerating simultaneously under some circumstances. The human driver must continue to pay full attention (“monitor the driving environment”) at all times and perform the rest of the driving task.
Level 3	An Automated Driving System (ADS) on the vehicle can itself perform all aspects of the driving task under some circumstances. In those circumstances, the human driver must be ready to take back control at any time when the ADS requests the human driver to do so. In all other circumstances, the human driver performs the driving task.
Level 4	An Automated Driving System (ADS) on the vehicle can itself perform all driving tasks and monitor the driving environment - essentially, do all the driving - in certain circumstances. The human need not pay attention in those circumstances.
Level 5	An Automated Driving System (ADS) on the vehicle can do all the driving in all circumstances. The human occupants are just passengers and need never be involved in driving.

There are plenty of very interesting developments in new materials, new mechanical systems and electrical systems. Recently the options for mechanical and electrical components has increased to the point that for many designs, off-the-shelf options are available. This allows for very rapid prototyping. A system can be assembled quickly so that developers may focus on the software and it allows much more time on the software aspect enabling contribution by software engineers. The control systems are very mature and are done at the lowest levels. This allows the developers to move to the highest levels of the software. The interesting questions from a computer science perspective relate to robot autonomy.

Autonomy is a significant challenge for those who work in robotics and artificial intelligence. Sensors can easily provide immense amounts of data. Understanding this data is a completely different and formidable issue. Thus we arrive at the fundamental distinction between syntax and semantics. Autonomous systems need to perceive the world, recognize objects, know their location and plan their activities (Table 1.2). Perception of the world around requires sufficient sensory data to reconstruct the world, but also requires a conceptualization of the world leading to understanding. Recognition of objects is essentially the same issue, again requiring conceptualization. Conceptualization requires a model or framework. A model is needed for localization and activity planning. Having robust and flexible models that operate in realtime is a complex task; a task that we will touch on in detail later in this text.

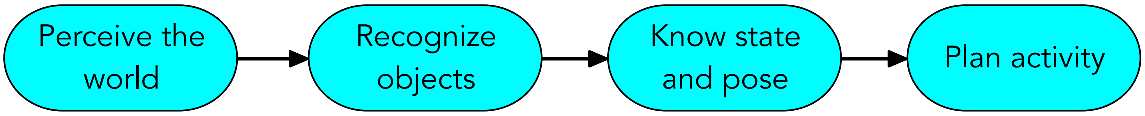


Fig. 1.17: The traditional challenge for the software.

Table 1.2: The challenge of autonomy

Requirement	**Implementation**
Have a model of the environment	Maps and Sensor Data
Perceive and analyze the environment	Data filtering and Sensor Fusion
Find its position within the environment	Localization, Mapping, Navigation
Plan and execute the movement	Path planning and Optimal paths

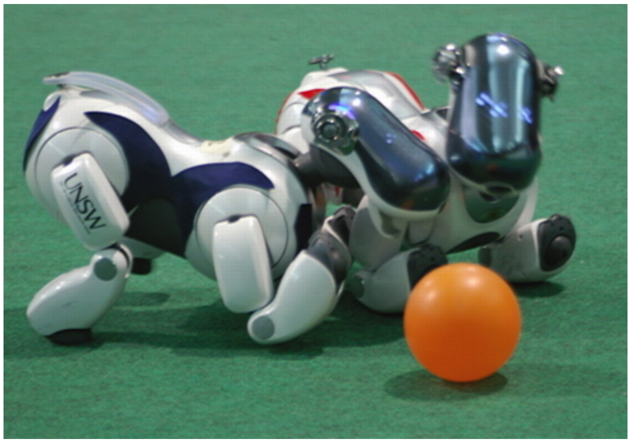


Fig. 1.18: Robots in RoboCup, [Wik18f]

Autonomy presents additional challenges. The environment is very dynamic. Objects can enter, leave and change shape. The landscape changes, location and orientation are unsure. However there are more subtle issues. Think about how the day progresses. The light changes as with the angle of the sun. There might be changes in natural versus artificial light. As the robot moves, the perspective on objects change. For example, look at your coffee cup (or tea cup . . .). As you rotate the cup, the handle can slip out of view. Now we see a cylinder and not a mug. Without higher order cognitive functions like object permanence, the object has changed type.

Modeling the environment is difficult. There are no simple ways to do this. You may have a compact representation, but the enormous storage requirements brings large computational complexity. For example, you might decide to use a simple grid system to mark areas of occupied or free space. Say the grid is a cube 4 inches on a side. In a typical warehouse which is 20,000 sq ft by 15 ft high gives us 2.7 million grid points to filter through. Larger outdoor domains are not possible with grid based object referencing and so other more complicated storage approaches are needed.

Another aspect which makes autonomy challenging is the multitude of sources of uncertainty. Sensors are noisy devices. At times they seem more like random number generators than physical sensors. From moment to moment, the picture that an autonomous system has changes due to the noise of the sensors. The noise needs to be filtered out while keeping relevant data and doing so quickly.

1.2.6 Symbiotic Autonomy

The thrust has been to attempt solutions to each aspect of the chain of challenges a robot will face. However, current machine learning cannot resolve solutions to all of the challenges facing an autonomous machine. So, to mimic how humans would address this problem would be to ask another person - either for information or actual assistance. Consider a mail delivery robot that needs to operate in a building with an older elevator. The elevator has a simple push button system to call the elevator. For the robot to move from floor to floor, we might want to add the ability to the robot to use the elevator. One solution would be to augment the elevator system to talk to the robot over wifi or similar. This would work for one system, but in general since we cannot go and overhaul all of the elevators. Another approach might be to add some type of manipulator. This requires a robotic arm, control system and vision system. However a very simple solution is to ask a nearby human to press the button. Although this is a simple thing, it illustrates that a robot does not need all of the expertise and capabilty. It can ask the internet, other robots or humans around for information. It can ask humans for physical assistance. We will call this symbiotic autonomy.

1.3 Fundamental Challenges

1.3.1 Navigation and Localization

Navigation is the process of routing the robot through the environment. Localization is the process of determining where the robot is in the environment. Most of the robots we imagine can move around. So, we expect that a mobile robot can navigate its environment. This really seems pretty simple. After all, worms and insects can do it, so machines should have no problem. Right? Navigation in three dimensions requires that the robot have a full understanding of the obstacles in the environment as well as the size and shape of the robot. Determining a path through the environment may also come with constraints on the path or robot pose. Typically to route a robot to some location, the current location is needed. Clearly just moving and avoiding obstacles does not require any knowledge of location, but there are plenty of times where the routing and localization problem are intertwined.

Navigation requires sensory information. The availability and type of information is critical to how effectively the robot can navigate or localize. Having only sensors that measure wheel location makes localization difficult and path planning impossible. Dead Reckoning is the method of determining the speed and run times for the motors. Then repeating this in different combinations in order to navigate the course. Essentially this is the game that you memorize your steps and turns and then try to retrace them with a blindfold. Modifying the environment allows for much better control of the robot but with the added costs of environment modification, see Figure Fig. 1.19. Dead reckoning normally has very poor results due to normal variations in motors. Environmental instrumentation can be very successful if available.

The approaches and algorithms are based on the underlying representations of space. We can represent space as a grid, or a continuum or an abstract system, Figure Fig. 1.21. Each method will determine the way we

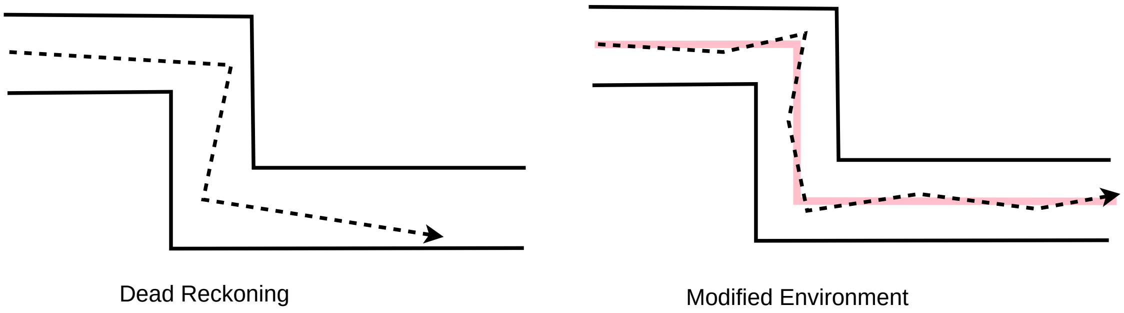


Fig. 1.19: Navigation approaches. a) A very simple approach to navigating the robot. The programmer codes in the times and velocities to run the motors for linear travel and turns. b) By instrumenting the environment, for example placing lines or grooves on the floor, the robot can successfully navigate.

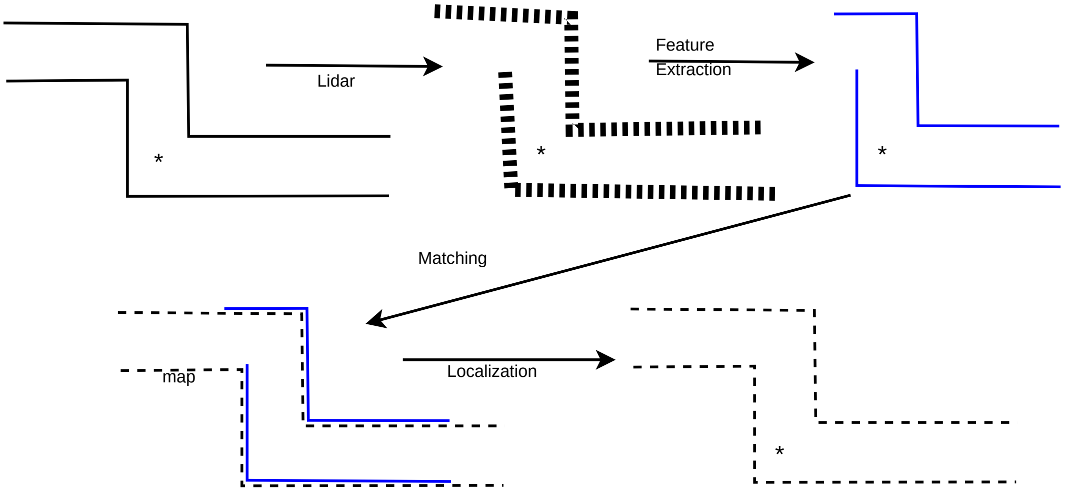


Fig. 1.20: Localization can be very difficult. In this example, a LIDAR scan is compared to a known map to deduce the location of the robot.

index the object (integers or floating point values), the resolution on location and the algorithm for accessing the object. We could also represent space in a discrete manner. This makes grid based approaches available. Space could also have a graph structure. The algorithms to navigate then will use or exploit these different ways space is represented. The differences give rise to different performance, accuracy, and results.

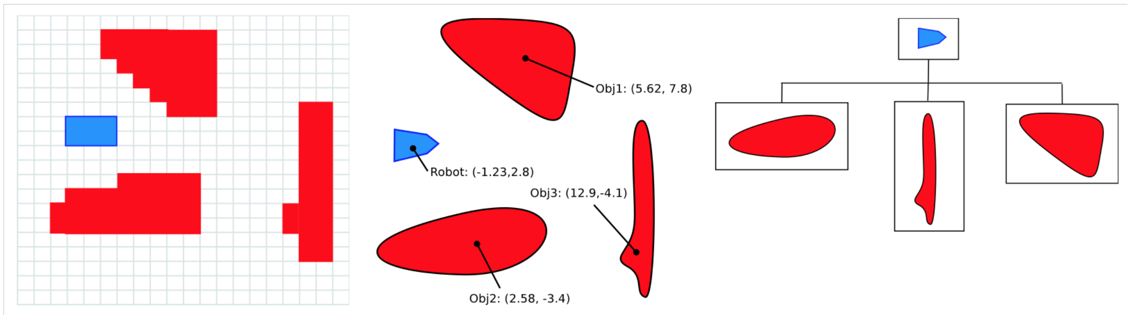


Fig. 1.21: An example of different map types.

Although challenging, navigation is a core skill in mobile robotics. Autonomous navigation is a focus for many industries. Farming is looking at conversion to autonomous machines as well as autopilot systems for automobiles. Of great current interest is a vision based autopilot system, Figure Fig. 1.22. This is an active area of research and we touch on it in the next section.

Fig. 1.22: Vision based driver assist system (Bosch).

1.3.2 Vision and Mapping

For many of us our dominant sense is vision and we have readily available sensors - the camera. Cameras can be much more sensitive than our eyes as they can deal with a greater intensity and frequency range. For all of the improvements in digital imaging, processing all of that data into a meaningful information is still a significant challenge. One of the major goals in computer vision is to develop vision systems modeled after our own capability.

With the rise of convolutional neural networks (since 2012), we have witnessed dramatic improvements in computer vision. The field is commonly known as deep learning and is addressing some fundamental

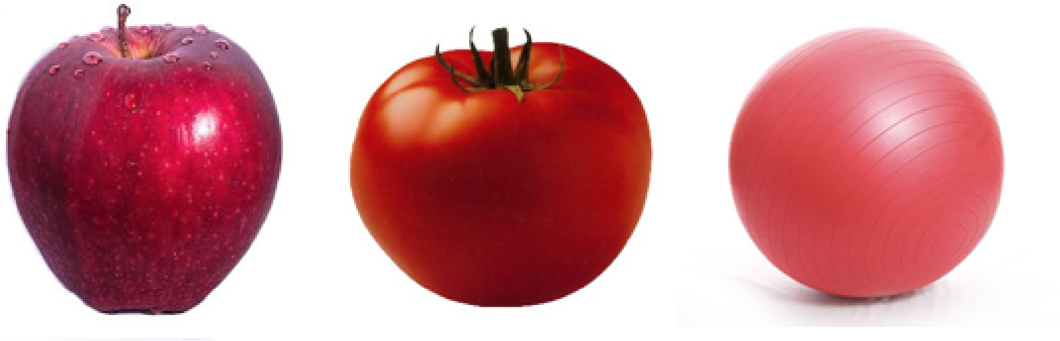


Fig. 1.23: For humans, it is very easy to distinguish apples, tomatoes and PT balls, but has presented considerable challenge for machine vision systems.

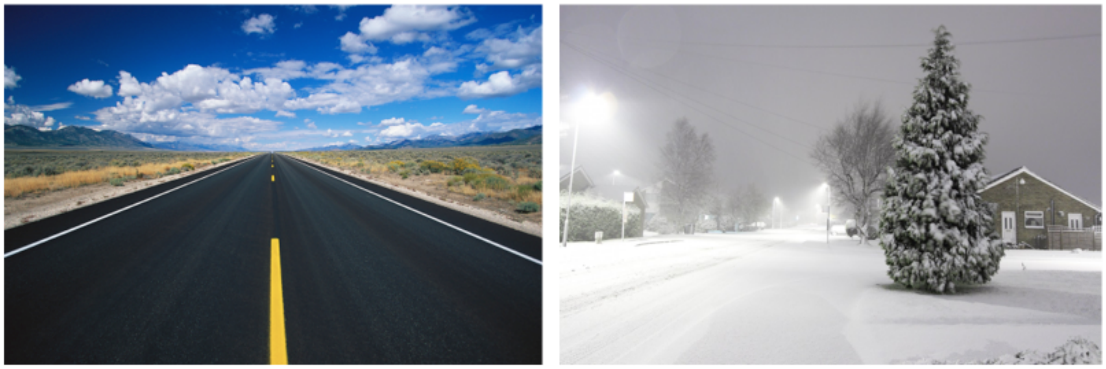


Fig. 1.24: It is easy for a human but hard for a computer to track the road in a variety of lighting conditions and road types.

problems in vision as well as a host of other applications. Advances in deep learning are starting to impact robotics and will significantly as times goes.

Mapping, in robotics, is the building of a representation of the robot's environment. The assumption often made is that either a map is available or not required. In some cases a map is required, but not available. If the application is surveying, the map is the goal. When reasonable localization is present, mapping just follows from the onboard sensors. If range sensors are available, then a map can be produced by knowing the location of the sensor (we assume the relation between the robot and its sensors are known) and the range data to objects. A map can be produced as the robot moves about the environment collecting the data. Again, the details on how this is done is dependent on the environmental representation (such as metric versus grid maps). The details are also affected by the accuracy and resolution of the sensing system.

If location is not known, but the sensors do provide some metric or range information, then mapping is still possible. SLAM, Simultaneous Localization and Mapping, is the process to determine the local map as well as the robot's location on the map. We will discuss SLAM later on in the text.

An interesting *chicken and egg* problem arises. Map building requires knowledge about localization. Conversely, localizing a robot on a map requires a map.

- If I have a map, then I can figure out my location from landmarks.
- If I know my location, then I can build a map.
- If I don't know where I am and I don't have a map ..then..?

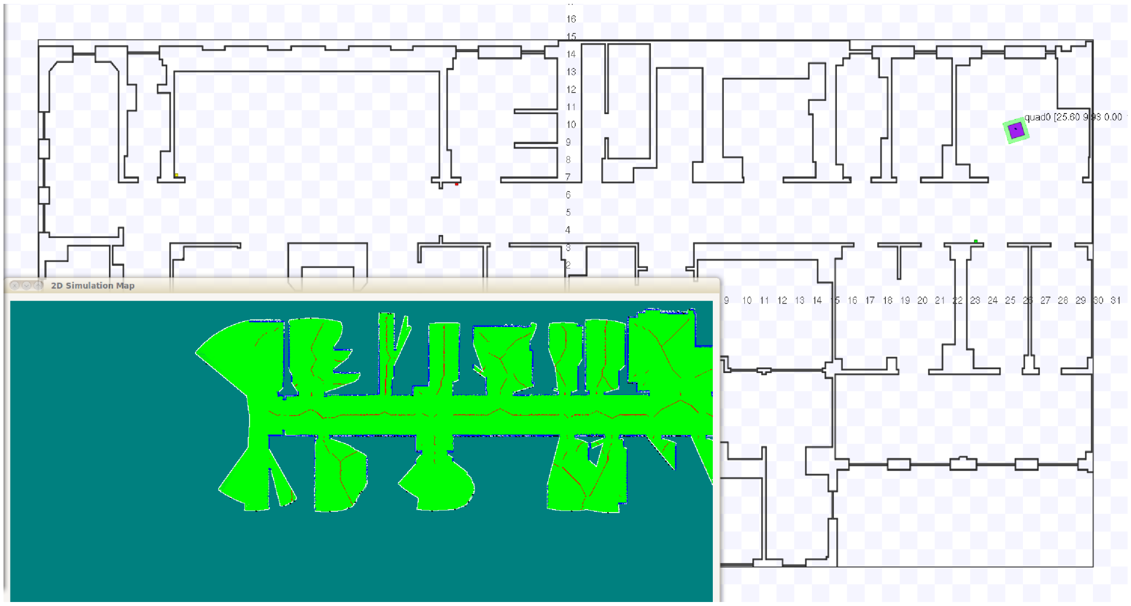


Fig. 1.25: SLAM: Simultaneous Localization and Mapping

When a robot enters an unknown environment, neither the map of the environment for the location of the robot on the map are understood. These two processes must occur together, simultaneous localization and mapping. This is done often enough that it has a name: SLAM (Fig. 1.25:). The 2D SLAM problem has been well addressed for interior environments, however 3D SLAM is an active area of research.

There are limits of course. It is possible to confuse any SLAM system. Generally, if humans cannot map or localize, then expect the robot cannot either. Consider highly repetitive environments or featureless

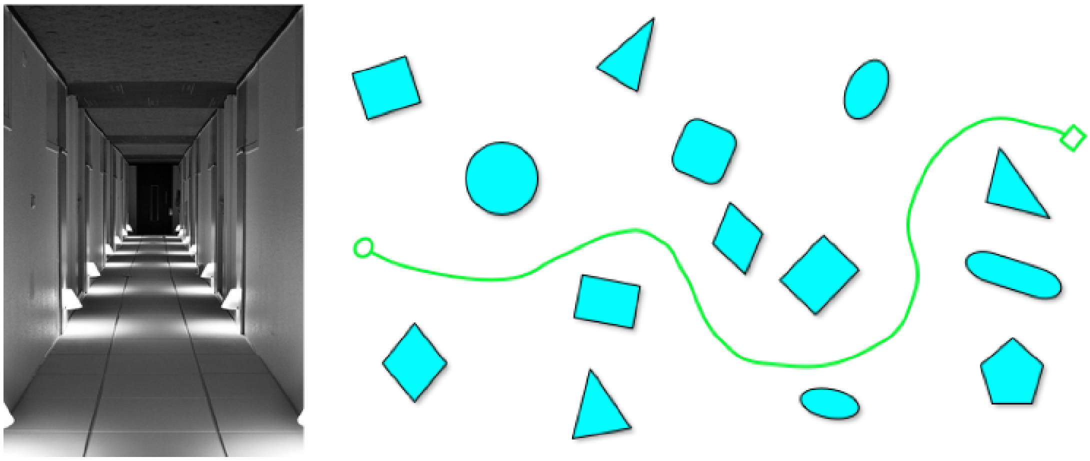


Fig. 1.26: Localization and Routing: a) How does the robot know where it is? b) Hard to plan a route if location is unknown.

environments, Figure (Fig. 1.26); it is easy to see how a vision system could get confused. These are special cases where there is very little information available however and we don't expect the vision system to perform without adequate data.

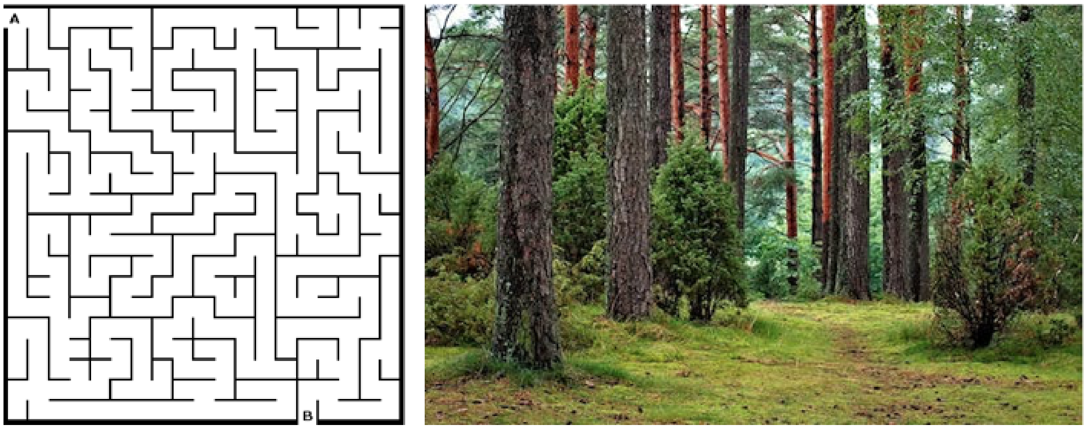


Fig. 1.27: Compare the structure of a maze to that of a forest scene. Very simple robots can plan a route and escape a maze. Routing through random obstacles in three dimensions is still very difficult for a robot.

If the robot knows the environment, either from a successful application of a SLAM algorithm or predetermined in the case of industrial robots with structured workspaces, then it is reasonable to ask about planning motion which is optimal in some sense. The field of planning is interested in deriving motion paths for articulator arms or mobile robots, Figure (Fig. 1.26). The environment will have obstacles, the robot will have constraints, and the task will have certain goals. Based on these requirements, the system attempts to compute a path in the environment or working space that satisfies the goals.

It is interesting to note that tasks which are easy for humans can be hard for robots and tasks which are hard for human may be easy for robots. Meaning tasks with lots of structure and rigid environments, the robot can succeed and maybe succeed better than a human. Other tasks which lack structure for which humans are quite adept, the robot may not succeed at all.

1.4 Robot Control

Assume that you want to build a robot that can deliver mail to the residents in a elder care facility. This is akin to the drug delivery robots in hospitals. The halls are straight and corners are 90 degrees. The layout does not change much over time and the build plans are available before the robot goes into service. The first temptation would be to try a form of dead reckoning. Of course it is clear that the wheels and motors are not ideal or identical. Drift will occur. The dead reckoning approach, meaning an approach which does not take in position information is known as open loop control. The open refers to not having feedback. Open loop control has problems with drift.

To address this problem the system will gather information from sensors and use this information to update the position. Meaning it will correct for drift. Not that we are completely stopping the drift since error will creep in and we cannot eliminate this. However we can adjust the system and hopefully compensate enough to navigate successfully. Using the feedback is known as closed loop. It is more complicated than open loop control but necessary for real world position control.

When one designs and builds a robot, it is natural to focus on the intended abilities. We think about having the robot perform some set of tasks. After laying out what the robot should do and what sensory data it needs, then we tend to think about how we will coordinate those activities. The coordination of the activities is an important element in the system design. Much of the way the robot behaves can be traced to the coordination approach used. There are two ways to proceed here; one based on a classical artificial intelligence approach and ones based on newer methods in artificial intelligence.

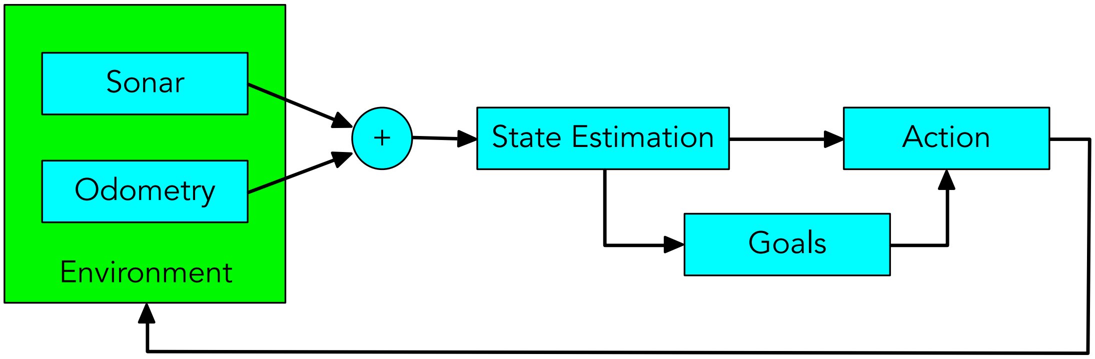


Fig. 1.28: Control system for a simple navigation system which fuses odometry and sonar.

For the classical methods we need complete modeling of the system in the environment. Typically this is a complete mathematical model of the different ways that the robot moves: kinematic model, control inputs, environment description, etc. The approach is then function based and follows a sequential decomposition of the tasks, see Figure Fig. 1.29:. Independent of how things operate "under the hood", we tend to view these systems as interacting with the environment using a four stage conceptual framework Fig. 1.30.

Most of the time the developer will want to code up the robot behaviors. This may involve a set of actions or reactions to events. They can be simple rules sets, or finite state machines or very complicated expert systems. The goal is to impart the robot with enough machine intelligence so that it can operate in the

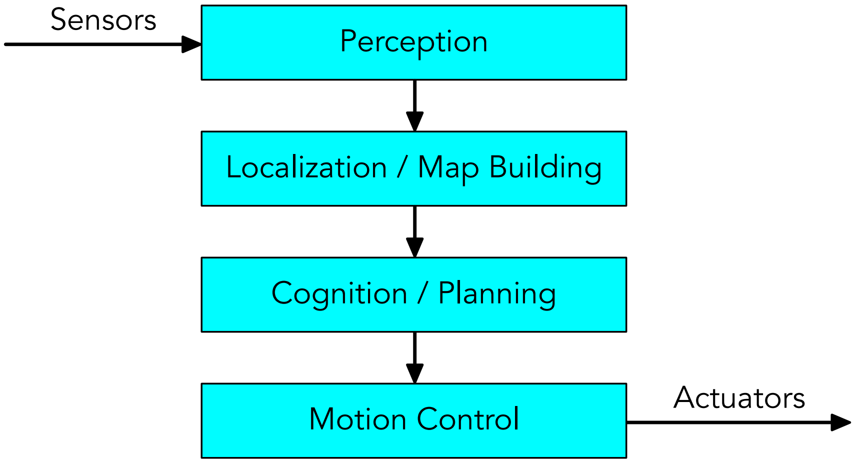


Fig. 1.29: A more traditional approach to robot control.

environment which it is deployed but keep the code simple enough to run on the onboard processors. For example, a number of years ago, one of the authors used a state machine for a simple exploration robot Fig. 1.31. In this case the decision process is completely defined before the robot is sent out.

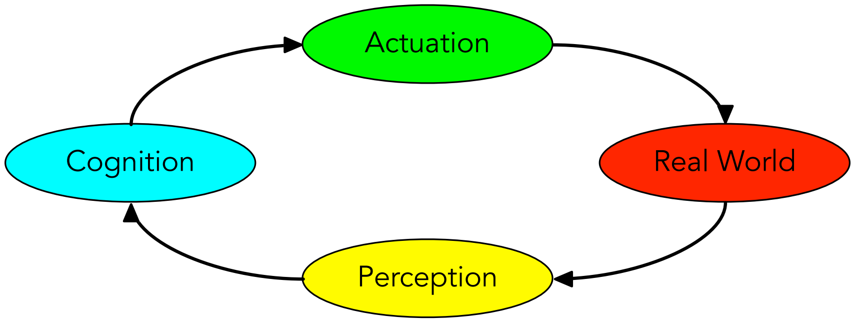


Fig. 1.30: A more traditional approach to robot control.

The Prussian general Helmuth von Moltke the Elder has been paraphrased in “No battle plan ever survives contact with the enemy.” This is certainly true for the preprogrammed robots. Unless in the situation of an industrial robot which has as consistent environmental presentation, the issues in the natural world are overwhelming. Beyond things like noise and drift are unexpected objects or events in the world around the robot. The programmer is hard pressed to anticipate, design and program for all the contingencies. The sensors can be inconsistent or unreliable. All of this leads to difficulties in obtaining accurate position/orientation estimates.

From early in its history, engineers have been dealing with the vast separate of the perfect world in one’s mind and the messy dirty world we live in. Tools such as digital signal filters like the Kalman Filter aimed at cleaning up sensor input or higher fidelity motor encoders to increase accuracy and resolution have been, and still are, widely embraced. Fuzzy logic or Bayesian based algorithms gave some measure of robustness, with the latter being exceptionally effective at dealing with uncertainty. Recent state of the art systems are a bundle of Sigma Point Kalman filters, Markov localization algorithms, motion planning and goal determination routines, actuator control codes, glued together by some type of interprocess communication.

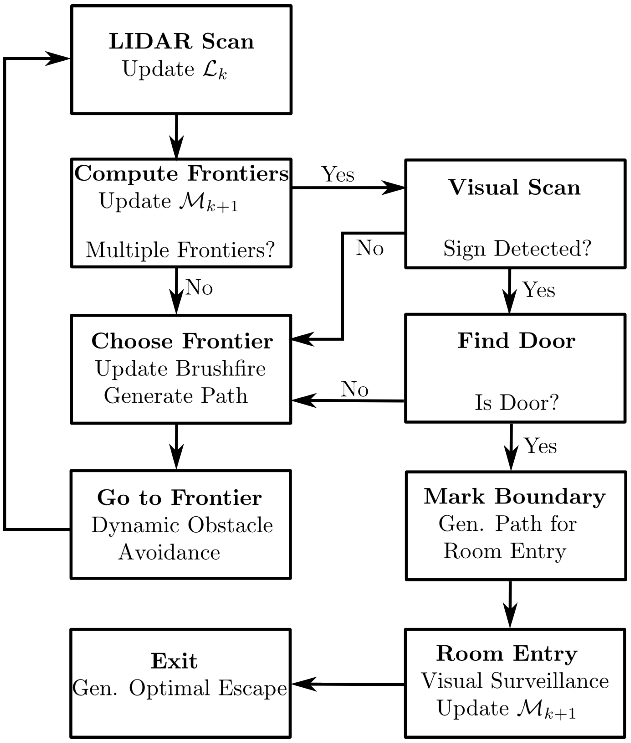


Fig. 1.31: A finite state machine for an exploration robot.

All of this is supported by some modern OS and middleware.

For fully autonomous mobile robots, such as seen in planetary exploration, it is impossible for the system designer to anticipate all of the situations the robot will find itself in. Even when we can anticipate, we tend to think and use language with significant lack of precision. This allows us to say things like "drive to the gas station, turn left and head up until you see . . . ", which are easy to say but very hard to program. Increases in data, mission scope, environment means the computational task increases at a geometric rate. To address this, we turn to lessons learned in the biological world. Clearly evolution has solved these problems in nature and so we engage the tools of natural computing to solve the problems in robotics. It has been said that the killer app in artificial intelligence is robotics. Although I believe this to be true, given the difficulty in defining a robot, the statement is mostly a catchy one liner.

Google, Nvidia, Amazon, Facebook all have embraced some form of machine learning as critical to their futures. Some of these approaches are statistical, but many are biologically motivated. For example, convolutional neural networks and reinforcement learning are two very current popular approaches in machine learning. Neither is new, but has benefited from years of research in algorithmic tuning and massive increases in hardware performance. The connectionist approaches tend to be highly parallelizable and see dramatic improvements in performance on GPUs, FPGAs and DSP hardware (TPUs). Thus modern robot control architectures see a parallel decomposition of the elements in the sensing, cognitive and actuation stages of the control algorithm and reflect the biological roots, see Figure Fig. 1.32.

The strengths of these new machine learning tools are in the ability to learn, the robustness to faults and errors, as well as a much reduced human design. Rules or patterns are not programmed in. Cases, especially edge cases need not to be defined. Kinematic models can be dispensed. Digital Signal Filters and sensor fusion models may be removed. Having a system which can learn can by orders of magnitude reduce

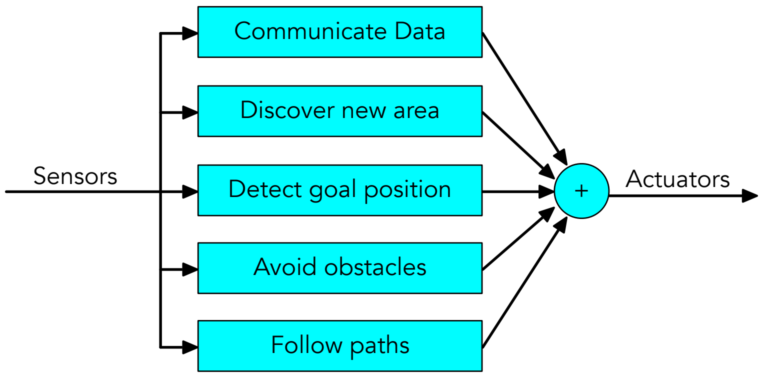


Fig. 1.32: Newer approaches parallelize the control architecture. The details of the final fusion step are discussed later.

development hours for a specific system. The machine learning methods we will examine in later chapters will mostly be based on biology, specifically on neural networks or behavioral learning theories.

1.4.1 Sense, Plan and Act

Robin Murphy in her text *AI Robotics* [Mur00], discusses the fundamental processes that robots must have. Sensors gather information about the environment surrounding the robot. A control system takes in the sensed information, merges it with knowledge contained within and plans a response, then acts accordingly.

The sense, plan, act architecture was the obvious first attempt at control. Sensory data is received and processed into sensory information. For example a laser ranging device returns raw data. This raw data is processed into a distance map. The distance map might be corollated with an existing environment map. Using the map information the system can plan a response. This could be a trajectory for the robot or robot manipulator. Once the response is decided, the system would determine how to engage the actuators to gain the desired response using the kinematics models for the system.

Few, if any, biological systems operate this way. Biological systems react to stimulus more directly. There is a sense-act architecture that is in place. For a particular sensory input, a predetermined action is defined. This reflex system can be fast and effective. The limitations are obvious. The responses to the environment need to be predetermined. General purpose robots or robots in new environments cannot use this approach. Often the situation requires more complex responses which need planning that takes into account local data.

Hybrid approaches can be built from the sense-act architecture. Murphy describes a plan, sense-act approach. The robot will plan out the task and then embark on the task. During execution, the robot will work in a sense-act reactive mode while carrying out the plan. These ideas are abstractions and we will have opportunity to see how each can play out in detail when we look at more complicated tasks.

Bugs, bats and rats

The natural world is simply amazing. It is filled with incredible solutions to some very difficult challenges and the engineering has often looked at the natural world for ideas and insipration. An ant is a very simple creature which manages to survive all around the world in a vast array of environments. Ants can navigate large habitats with local sensing only (that we are aware of currently). We can use these small creatures as a model for some basic path planning and navigation. It is not our goal to imitate the natural world and so we make no attempt at an accurate insect model. For our purposes a generic "bug" will suffice.

All of us have watched ants wander the landscape. I often wonder how they actually manage to cover such large distances and return to the nest. Ants have three very important senses - touch, smell and sight (ants can sense sound through sensing the vibrations and so we lump this into touch). The sense of smell is very important for ants. They use pheromones to leave markers. In a sense, ants are instrumenting the landscape. As we will see this is similar to the Northstar style navigation systems used in many commercial systems.

Touch based navigation is the most elementary approach to sensory system. It can be used in conjunction with chemical detection or taste. Although possible for robotics, we will not discuss chemical detection sensors here. Another approach is to use sound. We use sound in a subconscious manner as a way to feel the room. It is an extension of touch. We infer hard or soft surfaces as well as room size. This is a passive use. An active use would be listening to our own voice. The feedback gained again give us some information about our surroundings. The most active use commonly illustrated is by bats and dolphins. They use sonar which gives them obstacle avoidance when vision is inadequate. Sensing using sound is easier than using light or radio waves due to the slower propagation speeds. Basic distance sensors using sound are inexpensive and readily available so many robots have successfully employed them for use.

We tend to use animals as models for robot capability. Placing a rodent in a maze was done early on to test memory and learning skills. It gives a benchmark to compare robot and animal capability, as well as providing a comparison.

1.4.2 Transparency and Verbalization

When we send an autonomous machine out to perform some task, we may need information from the robot about the details and decisions that occurred during the task. We may need to know why the robot was late on arrival or why it did not retrieve the requested item. Or maybe we need to know updates on the environment that was just sensed. The concept of transparency is to provide the humans interacting with the robots the information about how the task was performed just like we do with each other. For example, when a friend says "sorry I am late, I was delayed by traffic".

Robots are capable of logging everything. All of the sensory data, the internal and external configuration data, decisions based on that data and so on. All of this is stored in large files populated by numbers. For humans this is not very useful. Imagine your late arriving friend giving you a gigabyte of data that contains freeway traffic density and velocities of 50 millisecond samples. We want our robots to convert this into human terms. We want the robot to say "due to high volume of cars, the traffic slowed down and I was delayed." This is known as verbalization. It is an application of natural language processing applied to the data logs in the robot. It is an active but important intersection of co-robotics and machine learning.

1.4.3 Self Driving Cars

Although self-driving cars really fits as part of the previous discussion, we would like to address it specifically here. Of all of the robots that the population will encounter over the next decade, it is the self-driving vehicle that will be most common. We already have robotic vacuum cleaners in the home. Having a humanoid robot in home is still outside current technology. Not everyone will interact with manufacturing robots either. But automobiles are everywhere and a good example to illustrate the concepts presented in this text.

According to *asirt.org* 1.3 million people globally die in automobile accidents each year. Another 20-50 million are injured. Globally it sits in the top 10 causes of death. It is expected that the numbers will rise of the next couple of decades. The population is increasing and so is population density. Not only does this increase accidents, it increases travel time. Commute time to work is significant for some cities and occupations.

Automation of the vehicle has promise to address both of these concerns. The system will not get drowsey, intoxicated, distracted, or angry. This has clear potential to reduce accidents. The driver is freed to focus attention on other activities during the commute. Individuals who have issues that prevent them from driving can now commute and are no longer hostage to external services. Cars that can communicate with each other are able to smooth traffic flow, increase flow rates and provide early warnings to other vehicles when dangers are detected.

Self-driving cars have a long history. In the 1950s GE imagined vast highway systems that automated travel. In the mid-1990's Mercedes experimented with driverless technologies and achieved some impressive results. The public started to hear more about autonomous vehicles with the DARPA Grand Challenge.

Fast forward to today and we have virtually every auto manufacturer working on some level of autonomy for their products. We now have hundreds of thousands of miles traveled in a variety of conditions. Clearly highway miles are the easiest, but we have successes in cities and other dense obstacle rich locations.

What is needed to self-drive? Here are the steps:

- Sense the surroundings
- Model the environment
- Find vehicle location and pose
- Plan action
- Execute action

The sensor we are most familiar with is the camera. The camera produces a sequence of images for which we apply a series of software techniques known as computer vision. In the driving application this is how we would perform lane detection. It also is part of the road sign identification system and in general, obstacle identification.

Some classical tools can be used, but the state of the art is moving towards using Deep Neural Networks (DNNs) and specifically Convolutional Neural Networks (CNNs). For example, we may feed a DNN lots of various road sign images so the network learns to recognize stop signs, yield signs, route markers and so forth.

A very critical task is lane identification. Through a series of standard operations the bounding lines of the lane are extracted from the road image. The midline of the lane is estmated for later use in driving the car.

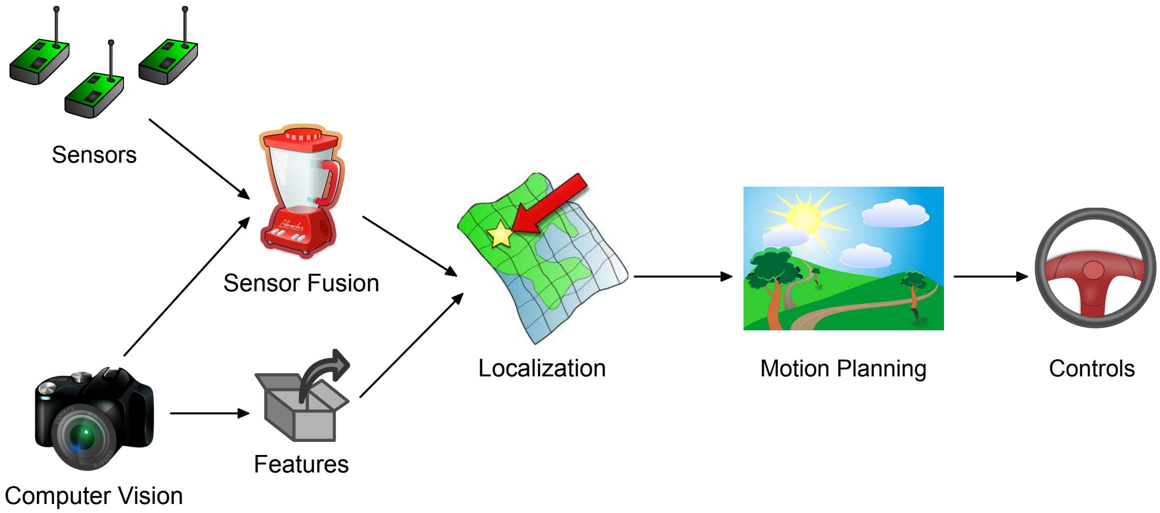


Fig. 1.33: Software chain for self-driving cars.

A single camera is a fabulous sensor, but there are some really critical things that the camera cannot do. Fundamentally the camera is a 2D sensor. It is not able to resolve the 3D world, specifically the depths of sensed objects. It is also unable to resolve the absolute scale of objects in the image. If you happen to know the size of the object, then you can infer the distance based on image height. However, this is really only practical in the near field due to the errors introduced by the pixelated image (more in the computer vision chapter).

One solution to this would be to use two cameras, known as stereo vision, just like we do. One can approximate the two camera concept by moving one camera around which is known as *structure from motion*. Limitations in hardware and computational complexity pushed the industry to try other sensors to measure distances and velocities directly.

Two popular distance or range sensors are radar and lidar. They use radio waves or light to measure the effective distance and through some mathematical techniques can also determine velocities. Radar is fairly inexpensive, at least in comparison to lidar. Lidar has very high accuracy and can produce millimeter resolution maps. Both operate at high data rates. Compared to lidar, radar is unaffected by light, snow, rain, fog and other conditions and combined with the lower cost, is a popular option.

Other systems such as sonar can also be deployed. Each has pros and cons which will be discussed in detail in the sensor chapter. In addition to ranging sensors, the vehicle will have a collection of other devices that perfom direct measurement of wheel speeds, have knowledge of orientation through a compass, know forces through an accelerometer, rotation rates through a gyroscope and position using a high accuracy GPS.

That said, we now have significant amount of information. Some of this information is redundant. That is a good thing since we can use the redudancy to increase our accuracy. This process is called sensor fusion. We can deduce the current state of our vehicle (speed, orientation, etc) by combining all of these data streams (and even using camera data) with a Kalman Filter.

Using this state information and combining it with a map is how we determine our location. If you have an accurate map with known landmarks, then we can trilaterate to determine our location. The localization process can also be done with modifications of the Kalman Filter or using another tool known as the Particle Filter.

At this stage in the pipeline, we know where we are on the map and where we are on the road (recall the computer vision aspect). The lane detection algorithm will tells us about road curvature, obstacles, other vehicles and their states and so forth. Other modules would be predicting collisions with other moving or stationary objects.

Next we need to look at our goals and compare to our current state. We may want head to a certain location, avoid other cars, stop at stop lights, and obey other traffic laws. So the motion planning system or path planning code will determine the next course of action. This would be desired velocity, turn angle, and braking.

Those motion decisions are sent to the servo and motor controllers which will translate the coded signals into higher voltage electrical power. Sensors measure the response and the controller adjusts to get the desired result. This is known as controls and is our final step in the pipeline. This is all there is, although the elements of a real system may be more complicated they are essentially the elements described above.

1.5 A Few Last Words…

This text aims to survey the subject of robotics. However, that is complete fantasy. Robotics is a huge field and it is not possible to really touch on all the different areas, delve into some of them and keep this text under several thousand pages (and a university course lasting one semester) as well as keeping your interest. So, we must compromise. This text will focus more on mobile systems and the technology to implement them than it will on manipulators (robotic arms and industrial assembly systems) - but not exclusively.

We will approach the subject from a computer science point of view and write for a computer science audience. This does not imply that mechanical or electrical engineers should set this down, just that the presentation will have a distinctly software orientation. Our coverage will balance more on higher level systems, machine intelligence, communications and algorithms with less time towards hardware, controllers, control systems, mechanics, electrical and materials. In essence we will see the robot as a type of distributed computing system but one which is aware of how the input data is gathered (sensory devices) and how the computational results are used (motors, etc).

As a computer scientist, what do you need to know to get started in Robotics? The list below provides an overview of the topics we will touch on.

- Simulation and Mathematics
- Behaviors and Motion
- Mechanics, Kinematics and Controls
- Electronics, Signals and Power
- Embedded Systems and Communications
- Distributed Systems
- Sensing, Vision and Ranging
- Planning, Routing, Localization and Navigation
- Mathematics

So, we begin our course in mobile robotics fundamentals. Robotics combines mechanical, electrical and software systems and some of these systems you need to understand as they fundamentally impact each other. The goal of this course is to develop sufficient background and understanding in the subject of mobile autonomous robotics so that you may become involved with a very dynamic growing industry. As Murphy's text will indicate, we will break the subject down into three aspects: perception or sensing, cognition or planning, navigation, localization, object recognition, and actuation or motion. Perception, cognition and actuation (sense, plan, act) is a basic theme for this course. However, there is a bit of the "chicken and egg" problem. The subjects are tied together. Each one can affect the other. It does not make sense to march entirely through planning, then through sensing and finish with actuation, no more than it would make sense to give you all of the lines of one actor in a play, followed by the next actor, and so forth.

The development of the subject has been a bit of a conversation between engineers and nature. Writing this book in complete historical accuracy is an interesting idea, but I bet it would become tedious after a couple of chapters. Our approach here is to give you a taste of a concept and put it into practice; then relate it to other concepts. Later we return and go into more detail, put that into practice and relate it to more involved concepts. This process will cycle through the sense, plan, act aspects - just as a real robotic system would. In short, I am applying Agile development to you. You are being rapid prototyped into a roboticist.

1.5.1 Supplementary Reading

There are many very good books on robotics. The field is well served by individuals who want to share their knowledge at many different levels and viewpoints. The important differences are the goals of the books. Some texts will focus on presenting the mathematics of articulated manipulators. Some will want to focus on mobile robot path planning. Others will want to talk about robot controllers using biological models. All of these points of view are important.

Below is a list of some texts with a brief description on the focus and audience:

- ***Principles of Robot Motion*, Choset et al. -** This is a great book that focuses on the algorithms behind autonomy. It presents a more theoretical treatment of mobile systems and does not spend much time on the classic kinematic tools like the D-H formalism. For most schools, this would be a graduate level text based on the mathematics used in the book although this could be used as an elective in a senior course if topics were carefully chosen. [CLH+05]
- ***Autonomous Mobile Robots*, Siegwart & Nourbakhsh -** This is a good shorter book which restricts itself to exactly what the title implies. As indicated in the Preface, the text you are reading follows the outline setout by Siegwart & Nourbakhsh and both are heavily influenced by Choset's text. The material on wheels and the associated kinematics is more in depth than other subjects in the text. Vision, Navigation, Localization and Mapping are briefly touched upon but supplementary material is probably warranted. [SN04]
- ***Computational Principles of Mobile Robotics*, Dudek & Jenkin -** This text is similar in topics and level to Autonomous Mobile Robots. Selection between the two would be based on specific topics of interest. [DJ00]
- ***Embedded Robotics*, Braunl -** Braunl's book surveys the field at a level that a junior in most engineering programs could easily understand. It has a wealth of information based on the author's personal experiences. It describes many projects and systems at a high level but does not delve deeply into the topics. If the hardware discussed in the text were more mainstream or current (Arduino, Raspberry Pi, etc), it would make the text much more approachable. [Braunl06]

- ***Introduction to Robotics, Analysis, Control, Applications*, Niku -** Niku's text is a great text for the more mechanical side of robotics. There is a wealth of material on kinematic models, inverse kinematics, and control. There are well done examples for basic kinematics as well. [Nik10]

The cultural attitudes are strongly affected by books and film:

METROPOLIS
MANUSKRIPT: THEA VON HARBOU
MUSIK: GOTTFRIED HUPPERTZ
EIN FILM VON FRITZ LANG
IN DEN HAUPTROLLEN: BRIGITTE HELM · GUSTAV FRÖHLICH
AN DER KAMERA: KARL FREUND, GÜNTHER RITTAU
UFA FILM IM VERLEIH DER PARUFAMET

M·G·M PRESENTS
FORBIDDEN PLANET
AMAZING!
STARRING WALTER PIDGEON · ANNE FRANCIS · LESLIE NIELSEN
WITH WARREN STEVENS AND INTRODUCING ROBBY, THE ROBOT
SCREEN PLAY BY CYRIL HUME
PHOTOGRAPHED IN EASTMAN COLOR
DIRECTED BY FRED McLEOD WILCOX · PRODUCED BY NICHOLAS NAYFACK IN CINEMASCOPE AND COLOR
A METRO-GOLDWYN-MAYER PICTURE

1.6 Problems

1. How would you define a robot?
2. What are the two main types of robots as presented by the text?
3. The text has broken robots into fixed or industrial manipulators and mobile machines. Often the industrial manipulator is performing repeated tasks or is remotely operated. Although many mobile systems are also remotely operated, we don't consider them doing repeated tasks.
4. Can you think of another way to classify robotic systems? What are the strengths and weaknesses of the classification?
5. What problems does a mobile robot face that a stationary robot does not? What about the other way around?
6. A stationary robot does not encounter new or novel environments. The workspace for a stationary robot is relatively fixed. It's tasks are predetermined for the most part. A mobile robot constantly

moves into new environments. Even if the task is to be repeated, in a new environment the details of performing the task will be different.

7. What are the differences between robots that are considered Mobile Machines and those that are considered to be Manufacturing Machines?
8. Do you think the *Robotic Appliance* and *Robotic Agent* partitioning is a more effective way to classify robots? Why or why not?
9. List several approaches that industry has used so robots can navigate in an environment; mentioning an advantage and disadvantage of each.
10. Describe the information gathered by a RGBD sensor such as the Microsoft Kinect.
11. Describe the information gathered by a stereo camera pair.
12. List out different ways one could assist a robot in navigating around a building or inside a building when GPS is not an option. [Think sensors.]
13. What is an FPGA? What is a GPU? What is a TPU? What are their strengths and weaknesses compared to traditional CPUs?
14. What are Isaac Asimov's Three Laws of Robotics? What do they mean? Are they complete, meaning are they a sufficient set of rules?
15. Work in robotics can replace people with machines. This results in job loss. Discuss the ethics of working in the robotics industry.
16. Military robotics is a growing industry. Although many systems have a high degree of autonomy, use of deadly force is left for the human. Discuss the ethical issues in allowing the robot to make these decisions.
17. If an autonomous system by design or error causes an accident, who is liable?
18. List a few ways biology has inspired robotics.

CHAPTER TWO

TERMS AND BASIC CONCEPTS OF ROBOTICS

Getting the language down is the first step. Robotics is like any other engineering field with lots of jargon and specialized terms. The terms do convey important concepts which we will introduce here.

There are three examples which we will examine: the serial two link arm, the parallel two link arm and the differential drive mobile robot. The serial two link manipulator is a simple robot arm that has two straight links each driven by an actuator (like a servo). They serve as basic examples of common robotic systems and are used to introduce some basic concepts. The parallel two link arm is a two dimensional version of a common 3D Printer known as the Delta configuration. Last is the differential drive mobile robot. This design has two drive wheels and then a drag castor wheel. The two drive wheels can operate independently like a skid steer “Bobcat”.

2.1 Terminology

In the Introduction, several terms were introduced such as end effector or actuator. We will round out the common robotics terminology in the section.

- Manipulator: the movable part of the robot, often this is the robotic arm.
- Degrees of Freedom: the number of independently adjustable or controllable elements in the robot. It is also the number of parameters that are needed to describe the physical state of the robot such as positions, angles and velocities.
- End Effector: the end of the manipulator.
- Payload: the amount the robot can manipulate or lift.
- Actuator: the motor, servo or other device which translates commands into motion.
- Speed: the linear or angular speed that a robot can achieve.
- Accuracy: how closely a robot can achieve its desired position.
- Resolution: the numerical precision of the device, usually with respect to the end-effector. This can also be measured in terms of repeatability. Related to accuracy vs precision in general measurements.
- Sensor: any device that takes in environmental information and translates it to a signal for the computer such as cameras, switches, ultrasonic ranges, etc.
- *Controller*: can refer to the hardware or software system that provides low level control of a physical device (mostly meaning positioning control), but may also refer to the robot control overall.

- Processor: the cpu that controls the system. There may be multiple cpus and controllers or just one unit overall.
- Software: all of the code required to make the system operate.
- Open Loop control: a form of robot control that does not use feedback and relies on timed loops for placement.
- Closed Loop control: using sensor feedback to improve the control accuracy.

Motion is achieved by some device that converts some energy source into motion. Most often these are electric motors (even non-electric systems often have electrically controlled components like valves.) However, it is useful to not focus on the type of equipment, but just the type motion induced. For simplicity anything that induces motion will be called actuators for most of the text. Actuators apply forces to the various robotic components in the system which in turn generates motion. The connections between actuators are known as *links*. For this work we will assume they are rigid and fixed in size. Connecting links are joints. This allows the links to move with respect to each other. There are two types of common joints, *rotary* and *linear* joints. The name essentially indicates what it does. A rotary actuator allows the relative angle between the links to change. A linear actuator changes the length of a link. Examples of rotary joints are revolute, cylindrical, helical, universal and spherical joints Fig. 2.1. Linear joints are also referred to as prismatic joints.

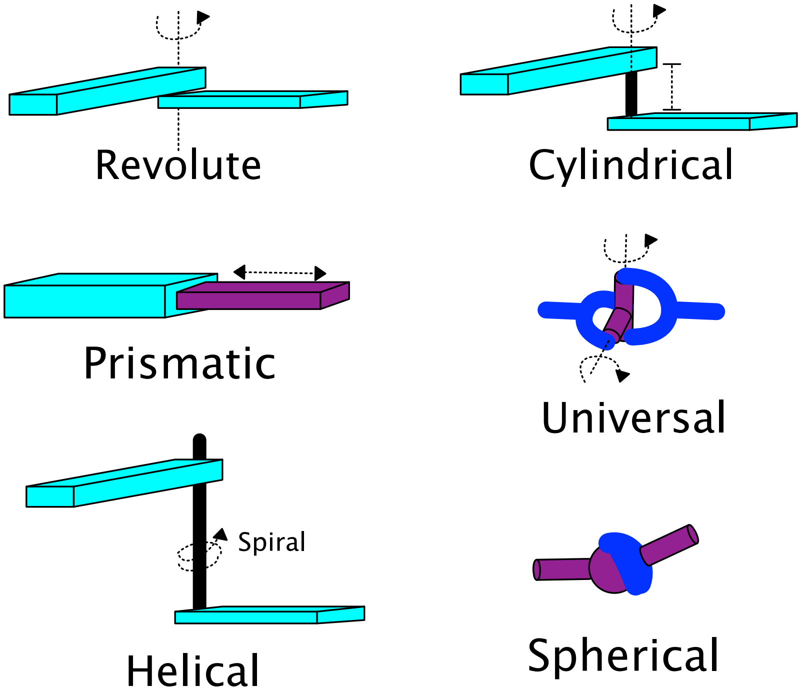


Fig. 2.1: Some common robot joints.

All of the machines we will study have moving components. The complexity of the system depends on the number of components and the interconnections therein. For example, a robotic arm may have three or four joints that can be moved or varied. A vehicle can have independently rotated wheels. The number of independently moving components is referred to as the *degrees of freedom*; the number of actuators that can induce unique configurations in the system. This mathematical concept comes from the number of independent variables in the system. It gives a measure of complexity. Higher degrees of freedom, just as higher dimensions in an equation, indicate a system of higher complexity. This concept of degrees of freedom is best understood from examples.

Consider a computer-controlled router that can move the tool head freely in the x and y directions. This device has *two degrees of freedom*. It is like a point in the plane which has two parameters to describe it. Going one step further, consider a 3D printer. These devices can move the extruder head back and forth in the plane like the router, but can also move up and down (in z). With this we see three degrees of motion or freedom. While it may seem from these two examples that the degrees of freedom come from the physical dimensions, please note that this is not the case. Consider the 3D printer again. If we added a rotating extruder head, the degrees of freedom would equal to four (or more, depending on setup), but the physical dimensions would stay at three.

Consider a welder that can position its tool head at any point in a three dimensional space. This implies three degrees of freedom. We continue and assume that this welder must be able to position its tool head orthogonal to the surface of any object it works on. This means the tool must be able to rotate around in space - basically pan and tilt. This is two degrees of freedom. Now if we attach the rotating tool head to the welder, we have five degrees of freedom: 5DOF.

Each joint in a robotic arm typically generates a degree of freedom. To access any point in space from any angle requires five degrees of freedom $(x, y, z, pan, tilt)$. So why would we need more? Additional degrees of freedom add flexibility when there are obstacles or constraints in the system. Consider the human arm. The shoulder rotates with two degrees of freedom. The elbow is a single degree of freedom. The wrist can rotate (the twisting in the forearm) as well as limited two degree motion down in the wrist. Thus the wrist can claim three degrees of freedom. Without the hand, the arm has six degrees of freedom. So you can approach an object with your hand from many different directions. You can drive in a screw from any position.

2.1.1 Serial and Parallel Chain Manipulators

Manufacturing robots typically work in a predefined and restricted space. They usually have very precise proprioception (the knowledge of relative position and forces) within the space. It is common to name the design class after the coordinate system which the robot naturally operates in. For example, a cartesian design (similar to gantry systems) is found with many mills and routers, heavy lift systems, 3D Printers and so forth, see Fig. 2.2-left. Actuation occurs in the coordinate directions and is described by variable length linear segments (links) or variable positioning along a segment. This greatly simplifies the mathematical model of the machine and allows efficient computation of machine configurations.

In two dimensions, one can rotate a linear actuator about a common center producing a radial design which would use a polar coordinate description. Adding a linear actuator on the z axis gives a cylindrical coordinate description, Fig. 2.2-right

A serial chain manipulator is a common design in industrial robots. It is built as a sequence of links connect by actuated joints (normally seen as a sequence starting from an attached base and terminating at the end-effector. By relating the links to segments and joints as nodes, we see that serial link manipulators can be seen as graphs with no loops or cycles. The classical robot arm is an example of a serial chain manipulator, Fig. 2.3-(left). Robot arms normally employ fixed length links and use rotary joints. This are often called articulated robots or the arm is called an articulator. Very general tools exist to construct mathematical descriptions of arm configuration as a function of joint angles. A formalism developed by Denavit and

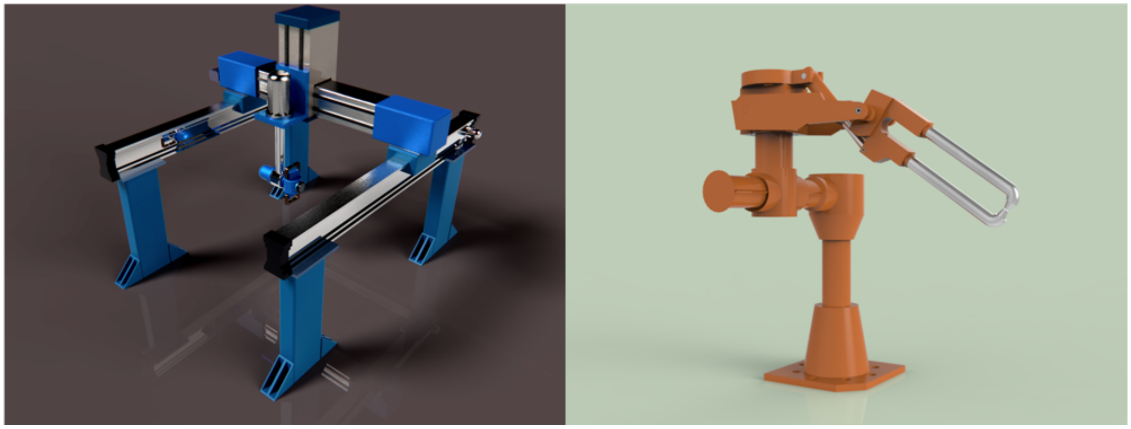


Fig. 2.2: Basic designs. (left) Cartesian design - Muhammad Furqan, grabcad.com (right) Cylindrical design - Mark Dunn, grabcad.com

Hartenberg can be used to obtain the equations for position.

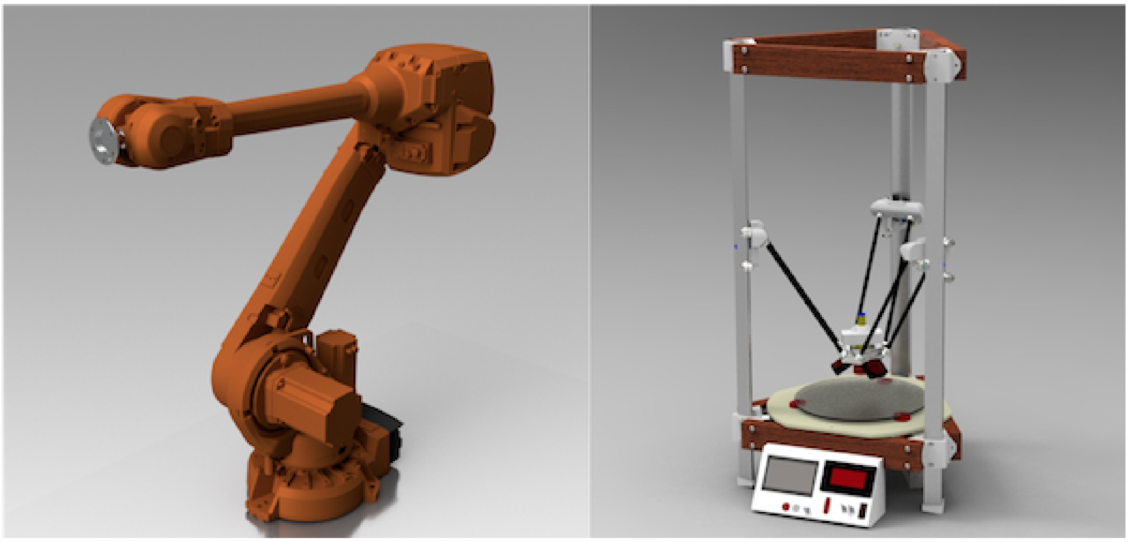


Fig. 2.3: Robot arms(left) Articulated - Ivo Jardim, grabcad.com, (right) Delta Design - Ivan Volpe, grabcad.com

Another popular approach is the parallel chain manipulator, which uses multiple serial chains to control the end-effector. A couple of examples, a Delta Robot, seen in Fig. 2.3-(right) and Fig. 2.4. Combinations of articulators can built to mimic a human hand as seen in Fig. 2.5.

2.1.2 Basic Machine Elements

We have been designing and using machines for thousands of years. There is a wealth of very interesting designs to do a myriad of things. We will review a few common designs in terms of basic function.

Sources of force in a robot arise from elecromagnetic devices such as DC motors or chemical processes such as internal combustion. The force produced is normally not in correct form for use on the robot. It could be

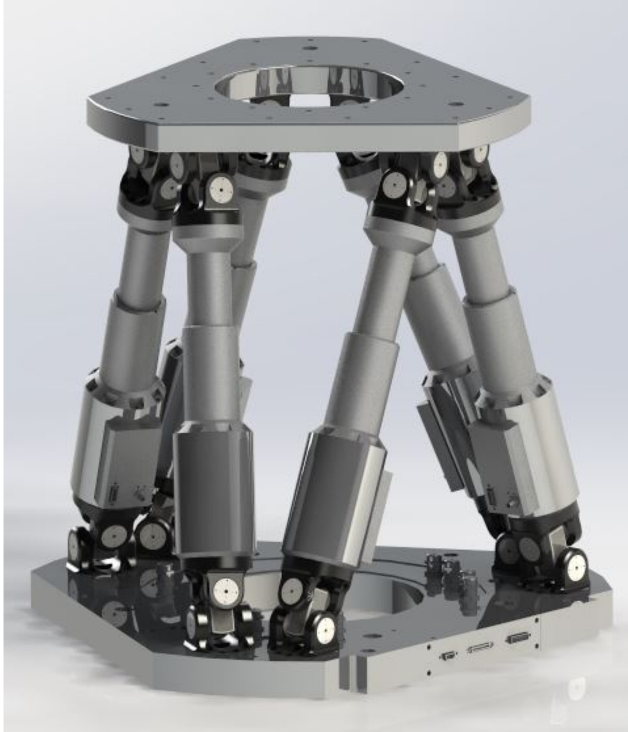


Fig. 2.4: Stewart Platform - Micheal Meng, grabcad.com

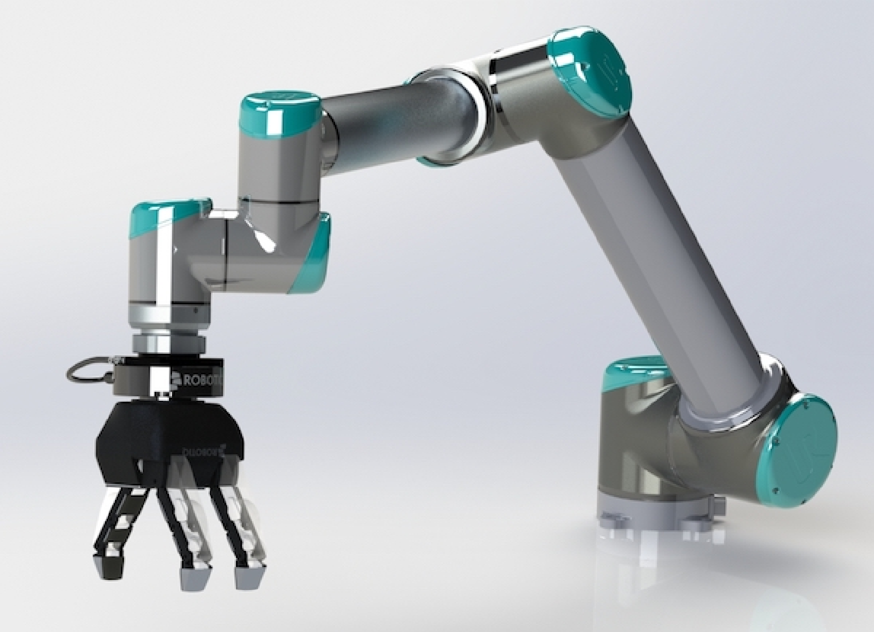


Fig. 2.5: Articulated with hand gripper - Chris Christofferson, grabcad.com

in the wrong direction, speed, magnitude of force, etc and needs to be changed. This is where gears, joints, rods and other essential components enter the design.

In robotics we see many elecrically powered systems. The main method for converting electrical power into mechanical power is to use a rotational motor design. For driving wheels, propellors, or other devices that use rotation energy this works very well. Getting linear motion requires some additional components. Although direct linear motion is possible through a solenoid type design, it is very common to find a electric motor and some type of gearing system to create the linear motion.

Gears are rotating elements with cut teeth that mesh with each other. They are capable of transmission of power and can change speed, torque or direction. Gears can be categorized according to relative relation of their rotation axes: parallel axes, intersecting axes, nonparallel-nonintersecting, and other.

Parallel axes contain spur gears, helical gears, internal gears and gear rack designs. Intersecting axes include straight and spiral bevel gears, miter gears. Nonparallel-nonintersecting gears have worm gears designs and screw gear designs, Fig. 2.6.

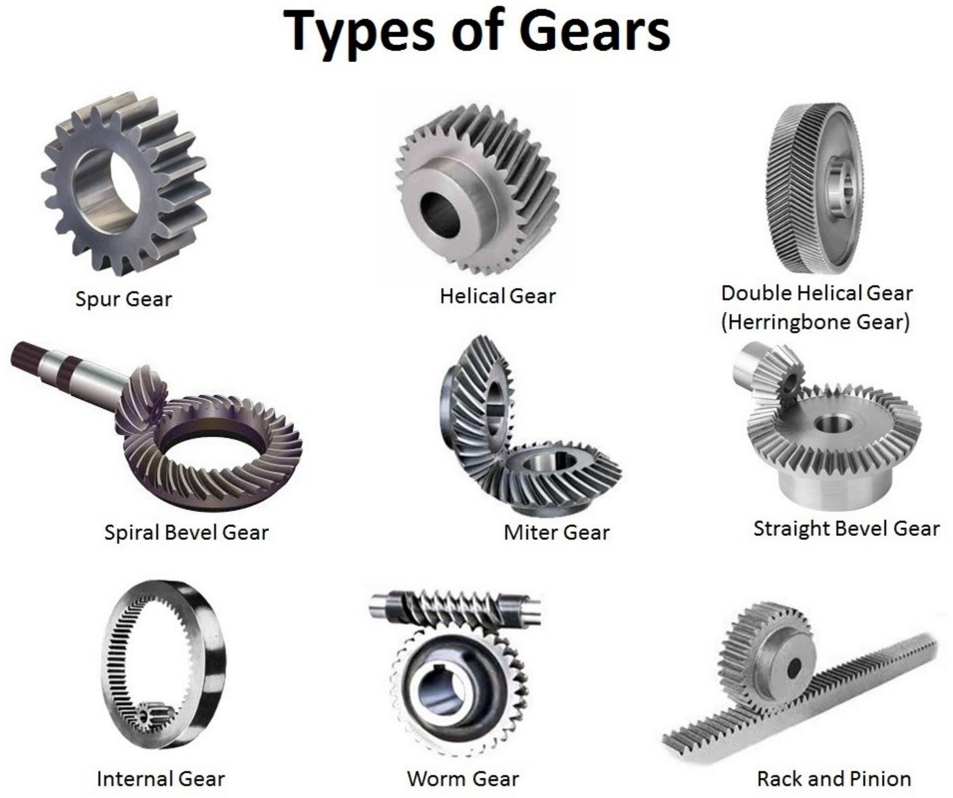


Fig. 2.6: Sample of different gear designs.

Beyond axes, the way the teeth fall on the gear are important. An external design is where the teeth lie on the outside (outer surface) of the gear and internal gears are ones where the teeth lie on the inside. Internal gears are nice in that they don’t reverse the shaft rotation direction. The teeth can run parallel to the rotation axis such as seen in spur gears or straight cut gears. These are the simpliest designs and only work with parallel axes. Helical gears use teeth that are not parallel to the rotation axis.

In spur designs, the teeth mesh with mostly static contact points and the engagement is all at once. Spur gears can be noisy at high rotational speeds. Helical or spiral designs the teeth engage more gradually and also slide against each other. This produces less noise or vibrations at the cost of higher energy loss and heat production.

Bevel gear designs (especially spiral) are used in differentials to transmit driveshaft rotation 90 degrees to axle rotation. Rack and pinon systems can be used to transmit the rotational motion of a steering wheel to the linear motion of a rod used to change the wheel angle (steering). Worm gear designs can provide significant torque as well as having the property of self-locking which means they are stationary when no power is applied to the worm gear (the worm wheel cannot drive the mechanism in reverse).

Most gear designs change angle by 90 degrees (although bevel angles and teeth designs can work at other angles). However the angle is fixed for the specific gear. If thesee angles are not fixed, which happens when suspension systems and steering are employed, then one needs joints that can address variable angles. Universal and flexible joints are used to allow for variable changes in rotation axes.

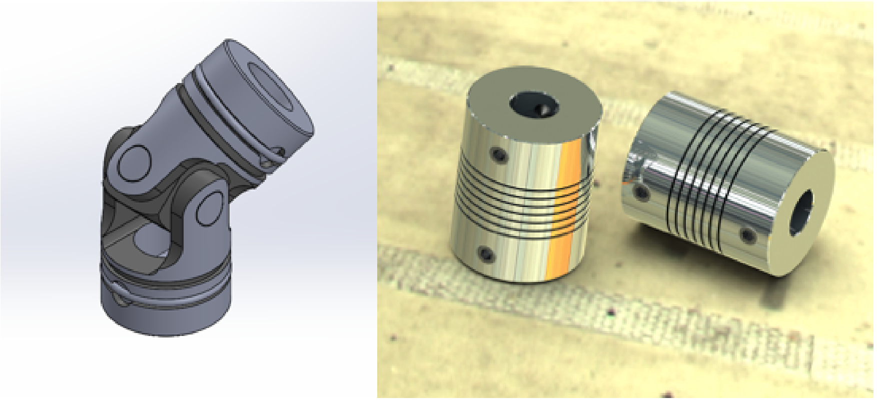


Fig. 2.7: Joint examples. (Left) Universal Joint - Devin Dyke, grabcad.com, (Right) Flexible joint - Chintan (CK) Patel, grabcad.com

Changing rotational motion to linear motion is important and for us arises often in manufacturing robots such as CNC machines and 3D Printers. A simple system is just a threaded rod and nut design (which is can be seen as a variation of the worm gear concept). By spinning the rod and not allowing the nut to spin, the nut will move up or down the rod. The relative motion is determined by the thread pitch. Because all of the threads of the nut are engaged, there can be considerable friction.

An improvement over a threaded rod is the ball screw, Fig. 2.8 . The idea is to add small balls (bearings) between the threads (or teeth if thinking of this like a worm gear). This reduces friction and backlash as well as increases accuracy. Using recirculating balls (in a wormdrive design) is how early automobiles provided smooth blacklash free lower force steering.

Other common methods to transfer rotational motion are sprocket and chains. Using different sizes of sprockets attached via the chain provides changes in angular speed and torque, called gearing based on the direct analogy to gears. The chains can be attached to tracks and a track or tank drive system is produced. A toothed belt is a variation of this system and is found from timing belts and Gilmer belts to the head positioning belts for 3D printers.

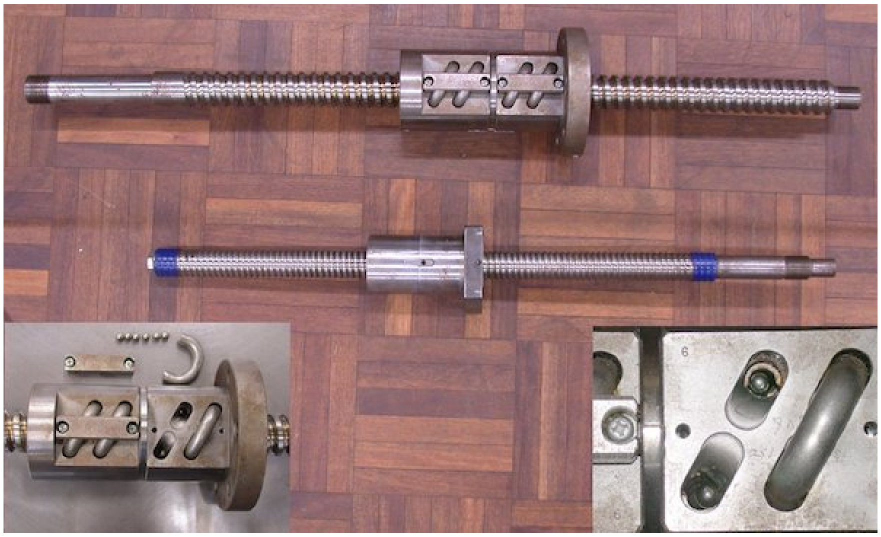


Fig. 2.8: Ball Screw - Glenn McKechnie, June 2006, Wikipedia

2.2 Reference Frames, Workspaces and Configuration Space

There are several frames of reference which are important to the robot. The robot operates in the physical world and it can be tracked by an external frame of reference. This is known as a *world* or *global* reference frame. This is normally the observer's frame of reference. Frames of reference on or within the robot are known as *local reference frames*. Each joint or actuator can have a frame of reference known as the *joint reference*. These are useful in understanding the transformations induced by each joint which leads to a kinematic model of the manipulator. For manipulation, the position and orientation of the tool needs tracking and will require a frame of reference known as the *tool reference*. Relating the various frames of reference requires some knowledge of coordinate systems and transforms which is found in standard courses in linear algebra.

The operating environment for a robot is known as the *workspace*. It is defined as the volume or region for which the robot can operate. For robots with manipulators, traditionally the workspace is all points that the tool end can reach. For mobile robots it is the region that the robot can and may move into. Any obstacle or constraint will be called a workspace obstacle or workspace constraint. Motion of a robot or articulator through the workspace is referred to by the term *workspace path*. Building a robot from rotary and linear joints means that our effector end (tool end) can be moved in an angular or linear fashion. This means that there is a natural coordinate system for the workspace. The three that are commonly used are Cartesian, Cylindrical or Spherical.

The range of all possible parameter values that the robot can modify is known as the *configuration space*. It is the span of the machine when the actuators are run through their different positions. The dimension of the configuration space is the degrees of freedom. The difference between workspace and configuration space might be confusing at first. Workspace is the physical one, two or three dimensions, where is robot operates, whereas configuration space is made up from the different configurations that the links and servos can define. In the case of manipulators, it is common to represent configuration space by the joint variables implicitly knowing the relation between the joint variable and the configuration. We will use q for a configuration and $\mathcal{Q}$ for configuration space to distinguish it from workspace variables.

We can denote the configuration space occupied by the robot by $R(q)$. A configuration space obstacle is

$\mathcal{QO}_i$,

$$\mathcal{QO}_i = \left\{ q \in \mathcal{Q} \mid R(q) \bigcap \mathcal{WO}_i \neq \emptyset \right\}.$$

Free configuration space is then

$$\mathcal{Q}_{\text{free}} = \mathcal{Q} \setminus \left(\bigcup_i \mathcal{QO}_i \right).$$

Reaching a point in space requires a particular configuration of the joints or motors. So there is a point in configuration space that relates to a point in the workspace. Understanding the relation between the articulators and the point in space turns out to be a very hard problem. If you know the position of each joint, you can then generate a mapping from the set of joint positions to a point in the workspace. This is known as forward kinematics. Having a target point out in space and asking what are the joint positions has to do with inverting the kinematic equations and is known as *inverse kinematics*. The forward kinematics are expressed as a system of algebraic equations. There is no general rule that these equations will be invertible. So, *IK*, inverse kinematics equations may not be available. Later in this text we will explore numerical approaches .

Just having a relation between the physical workspace and the joint (wheel, motor, etc) configuration is not the goal in robots. We normally want to do something. We want to move. We might be welding along a seam or driving a path. Our controls are working with the actuators and those are translated over into the workspace through some rather complicated mathematical expressions. The interesting challenge is to take a desired path in the workspace, say the welder path, and figure out the motion of the joints (motors) that produce this path. Then optimize based on workspace obstacles, machine constraints and other considerations. This is the subject of *planning* which will be touched on later as well as in courses on planning.

2.2.1 Forward Position Kinematics

The forward position kinematics (FPK) solves the following problem: "Given the joint positions, what is the corresponding end effector's pose?" If we let $x = (x_1, x_2, x_3)$ be the position as a function of time and $p = (p_1, p_2, \ldots, p_n)$ the equations that transform p into x are the forward kinematic equations

$$x = F(p).$$

2.2.2 Forward Position Kinematics for Serial Chains

The solution is always unique: one given joint position vector always corresponds to only one single end effector pose. The FK problem is not difficult to solve, even for a completely arbitrary kinematic structure. We may simply use straightforward geometry, use transformation matrices or the tools developed in standard engineering courses such as statics and dynamics.

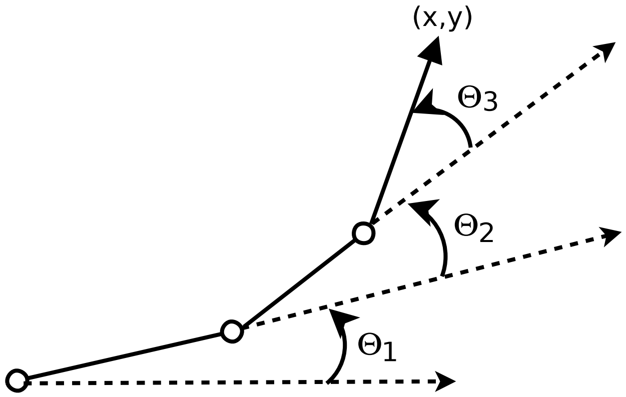


Fig. 2.9: A three link planar manipulator.

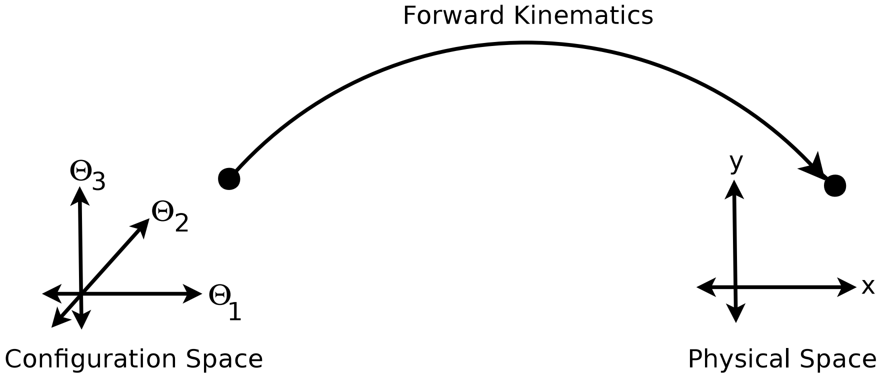


Fig. 2.10: The mapping from configuration space to workspace.

2.2.3 Forward Position Kinematics For Parallel Chains (Stewart-Gough Manipulators)

The solution is not unique: one set of joint coordinates has more different end effector poses. In case of a Stewart platform there are 40 poses possible which can be real for some design examples. Computation is intensive but solved in closed form with the help of algebraic geometry.

2.2.4 Inverse Position Kinematics

The inverse position kinematics (IPK) solves the following problem: "Given the actual end effector pose, what are the corresponding joint positions?" In contrast to the forward problem, the solution of the inverse problem is not always unique: the same end effector pose can be reached in several configurations, corresponding to distinct joint position vectors. A 6R manipulator (a serial chain with six revolute joints) with a completely general geometric structure has sixteen different inverse kinematics solutions, found as the solutions of a sixteenth order polynomial.

2.2.5 Forward Velocity Kinematics

The forward velocity kinematics (FVK) solves the following problem: "Given the vectors of joint positions and joint velocities, what is the resulting end effector twist?" The solution is always unique: one given set of joint positions and joint velocities always corresponds to only one single end effector twist. Using x to the the position vector as a function of time and p the joint parameters as a function of time, let the forward position kinematics be given by $x = F(p)$. Then the forward velocity kinematics can be derived from the forward position kinematics by differentiation (and chain rule). A compact notation uses the Jacobian of the forward kinematics:

$$v = J_F(p)q, \qquad \text{where} \qquad v = \frac{dx}{dt},\; q = \frac{dp}{dt}.$$

2.2.6 Inverse Velocity Kinematics

Assuming that the inverse position kinematics problem has been solved for the current end effector pose, the inverse velocity kinematics (IVK) then solves the following problem: "Given the end effector twist, what is the corresponding vector of joint velocities?" Under the assumption that the Jacobian is invertible (square and full rank) we can find J^{-1} and express

$$q = J_F(p)^{-1}v = J_F\left(F^{-1}(x)\right)v$$

2.2.7 Forward Force Kinematics

The forward force kinematics (FFK) solves the following problem: "Given the vectors of joint force/torques, what is the resulting static wrench that the end effector exerts on the environment?" (If the end effector is rigidly fixed to a rigid environment.)

2.2.8 Inverse Force Kinematics

Assuming that the inverse position kinematics problem has been solved for the current end effector pose, the inverse force kinematics (IFK) then solves the following problem: “Given the wrench that acts on the end effector, what is the corresponding vector of joint forces/torques?”

We will not treat forward or inverse force kinematics in this text. These concepts are treated in courses in statics and mechanics.

2.3 Simple Planar Manipulators

2.3.1 Serial Two Link Manipulator

The last few paragraphs have introduced lots of jargon. To understand them, it helps to see them in action. The simple two link manipulator, Fig. 2.11 is a good place to start. Imagine a robotic arm that has two straight links with a rotary joint at the base and a rotary joint connecting the two links. In practice, these rotary joints would be run by motors or servos and probably have some limits, but for now we will assume full 360° motion.

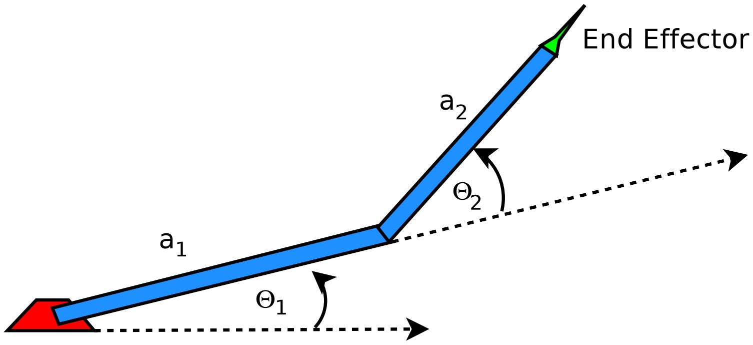


Fig. 2.11: The two link manipulator.

The workspace that the arm operates inside is a disk, Fig. 2.12. This is a two dimensional workspace. The figure indicates the workspace in gray. It may also be the case that there is something in the workspace, a workspace obstacle indicated in red. This unit has two joints which define a two dimensional configuration space, Fig. 2.13. The dimension of the configuration space is the degrees of freedom, and so this has two degrees of freedom. Since the joint is rotary and moving a full 360° degrees returns you to the same angle, the two directions wrap back on themselves. This actually makes the configuration space a two dimensional torus or “donut”.

We will illustrate what is meant by kinematics and inverse kinematics using the two link manipulator. Forward kinematics will identify the location of the end effector as a function of the joint angles, Fig. 2.14-(a). This is easily done using a little trigonometry. First we find the location of (ξ, η) as a function of θ_1 and the link length a_1, Fig. 2.14-(b):

$$\xi = a_1 \cos\theta_1, \quad \eta = a_1 \sin\theta_1$$

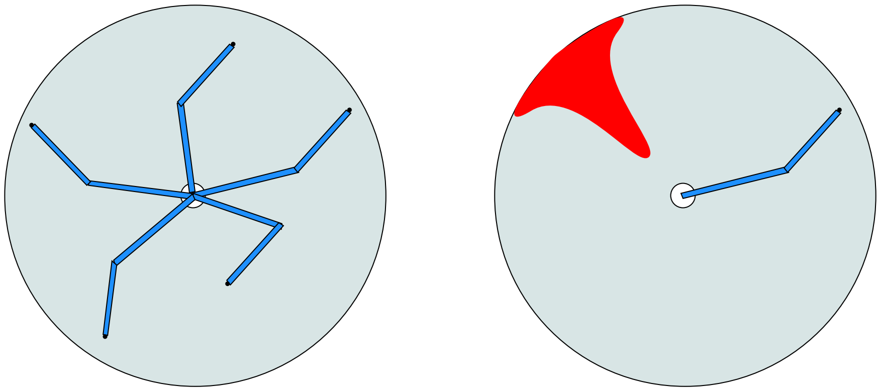


Fig. 2.12: Two link manipulator: (a) Workspace with equal link lengths and (b) Workspace obstacle.

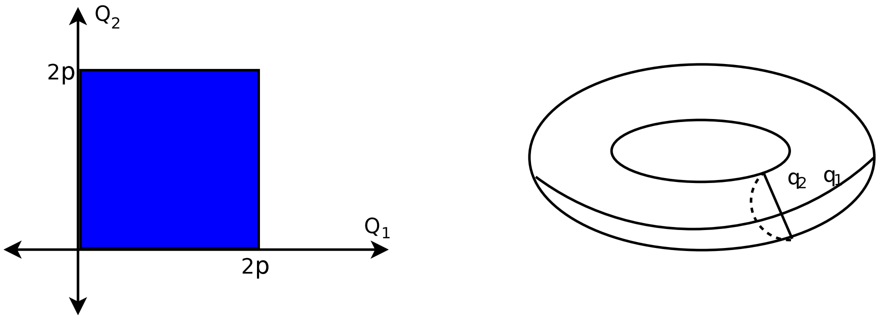


Fig. 2.13: Configuration domain and configuration topology which is a torus.

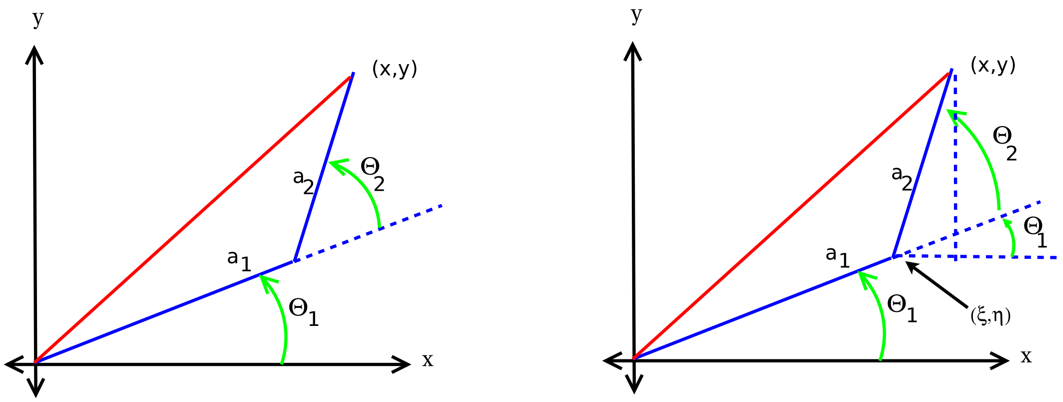


Fig. 2.14: a) The two link manipulator with the links and joints labeled. b) Location of the middle joint.

The next link can be included with

$$\Delta x = a_2 \cos(\theta_1 + \theta_2), \quad \Delta y = a_2 \sin(\theta_1 + \theta_2)$$

Note that $x = \xi + \Delta x$ and $y = \eta + \Delta y$. Combining the expressions, the forward kinematics are:

$$\begin{aligned} x = a_2 \cos(\theta_1 + \theta_2) + a_1 \cos\theta_1 \\ y = a_2 \sin(\theta_1 + \theta_2) + a_1 \sin\theta_1 \end{aligned} \tag{2.1}$$

As you move the servos in the system, you can change the angles θ_1 and θ_2. The formula (2.1) gives the location of the end effector (x, y) as a function of (θ_1, θ_2). The values x, y live in the workspace. The values θ_1, θ_2 live in the configuration space. This is a holonomic system. A common application is to move the end effector along some path in the workspace. How does one find the "path" in configuration space? Meaning how do we find the values of θ_1, θ_2 that give us the correct x, y values? This requires inverting the kinematics equations, hence the term inverse kinematics. The mathematics required is some algebra and trigonometery for solving θ_1, θ_2 in terms of x, y.

To find the inverse kinematics formulas we must appeal to some trigonometry (law of cosines):

$$x^2 + y^2 = a_1^2 + a_2^2 - 2a_1a_2\cos(\pi - \theta_2). \tag{2.2}$$

Using $\cos(\pi - \alpha) = -\cos(\alpha)$, we solve for cos in Eqn (2.2):

$$\cos(\theta_2) = \frac{x^2 + y^2 - a_1^2 - a_2^2}{2a_1a_2} \equiv D$$

Using a trig formula:

$$\sin(\theta_2) = \pm\sqrt{1 - D^2}$$

Dividing the sin and cos expressions to get tan and then inverting:

$$\theta_2 = \tan^{-1}\frac{\pm\sqrt{1 - D^2}}{D} = \text{atan2}(\pm\sqrt{1 - D^2}, D)$$

The +/- gives the elbow up and elbow down solutions. A source of errors arises with the arctan or inverse tangent of the ratio. The inverse function is multivalued and calculators (as with most software) will return a single value known as the principle value. However, you may want one of the different values. The problem normally is that since $-y/-x = y/x$ the inverse tangent function will not know which quadrant to select. So it may hand you a value that is off by $\pm\pi$. We suggest that you use atan2 in your calculations instead of atan which will isolate quadrant and also avoid the divide by zero problem. We will do the mathematics with $\tan^{-1}$, but keep our code using atan2.

Continuing with the derivation, from Figure Fig. 2.15, we have

$$\theta_1 = \phi - \gamma = \tan^{-1}\frac{y}{x} - \gamma. \tag{2.3}$$

If you look at the two dotted blue lines you can see that the line opposite γ has length $a_2\sin\theta_2$. The segment adjacent to γ (blue solid and dotted lines) has length $a_1 + a_2\cos\theta_2$. Then

$$\tan\gamma = \frac{\text{Opposite}}{\text{Adjacent}} = \frac{a_2\sin\theta_2}{a_1 + a_2\cos\theta_2}$$

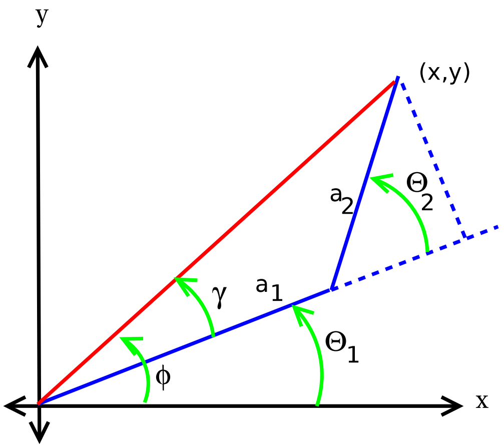


Fig. 2.15: The interior angles for the two link manipulator.

which gives us γ:

$$\gamma = \tan^{-1} \frac{a_2 \sin\theta_2}{a_1 + a_2\cos\theta_2}.$$

Plug γ into Eqn (2.3) and we obtain

$$\theta_1 = \tan^{-1}\frac{y}{x} - \tan^{-1}\frac{a_2\sin\theta_2}{a_1 + a_2\cos\theta_2}$$

Given the two link manipulator kinematic equations:

$$\begin{aligned} x &= a_2\cos(\theta_1+\theta_2) + a_1\cos\theta_1 \\ y &= a_2\sin(\theta_1+\theta_2) + a_1\sin\theta_1 \end{aligned}$$

The inverse kinematics (IK) are

$$D = \frac{x^2+y^2-a_1^2-a_2^2}{2a_1a_2}$$

$$\theta_1 = \tan^{-1}\frac{y}{x} - \tan^{-1}\frac{a_2\sin\theta_2}{a_1+a_2\cos\theta_2}, \qquad \theta_2 = \tan^{-1}\frac{\pm\sqrt{1-D^2}}{D} \tag{2.4}$$

Note the kinematic equations only involve the position variables and not the velocities so they are holonomic constraints.

Let $a_1 = 15$, $a_2 = 10$, $x = 10$, $y = 8$. Find θ_1 and θ_2:

1. $D = (10^2 + 8^2 - 15^2 - 10^2)/(2 * 15 * 10) = -0.53667$
2. $\theta_2 = \tan^{-1}(-\sqrt{1-(-0.53667)^2}/(-0.53667)) \approx -2.137278$
3. $\theta_1 = \tan^{-1}(8/10) - \tan^{-1}[(10\sin(-2.137278))/(15 + 10\cos(-2.137278))] \approx 1.394087$

Check the answer:
$x = 10 * \cos(1.394087 - 2.137278) + 15 * \cos(1.394087) = 10.000$

$y = 10 * \sin(1.394087 - 2.137278) + 15 * \sin(1.394087) = 8.000$

The Python code to do the computations is

```
In [1]: from math import *
In [2]: a1,a2 = 15.0,10.0
In [3]: x,y = 10.0,8.0
In [4]: d =  (x*x+y*y-a1*a1-a2*a2)/(2*a1*a2)
In [5]: print d
-0.536666666667

In [6]: t2 = atan2(-sqrt(1.0-d*d),d)
In [7]: t1 = atan2(y,x) - atan2(a2*sin(t2),a1+a2*cos(t2))
In [8]: print t1,t2
1.39408671883 -2.13727804092

In [9]: x1 = a2*cos(t1+t2) + a1*cos(t1)
In [10]: y1 = a2*sin(t1+t2) + a1*sin(t1)
In [11]: print x1, y1
10.0 8.0
```

Note that all angles in this text are in radians unless explicitly stated as degrees. This is to be consistent with standard math sources as well as the default for most programming languages. Be careful with Python 2, don't forget to include the ".0"s. Most of all, be very careful with arctan. It can bite you. Here is an example . . .

Assume that $(x, y) = (9, 10)$ and $(a_1, a_2) = (15, 15)$. We compute

$$D = \frac{x^2 + y^2 - a_1^2 - a_2^2}{2a_1a_2} = \frac{9^2 + 10^2 - 15^2 - 15^2}{2(15)(15)} = -0.5977777777777777$$

$$\theta_2 = \tan^{-1}\frac{-\sqrt{1-D^2}}{D} = 0.9300701118289644$$

$$\theta_1 = \tan^{-1}\frac{y}{x} - \tan^{-1}\frac{a_2\sin\theta_2}{a_1 + a_2\cos\theta_2} = 0.3729461690939078$$

Now check our answers . . .

$$x = a_2\cos(\theta_1 + \theta_2) + a_1\cos\theta_1 = 17.93773762042545$$
$$y = a_2\sin(\theta_1 + \theta_2) + a_1\sin\theta_1 = 19.9308195782505$$

Not close. What happened? The first problem was that in $\frac{-\sqrt{1-D^2}}{D}$ which is $\frac{-0.801...}{-0.597...}$ becomes $\frac{0.801...}{0.597...}$ and then atan returns a quadrant I angle of 0.930070. . . . We needed $\theta_2 = 0.93007 + \pi = 4.0716$. Then you get $\theta_1 = 1.94374....$

$$x = a_2\cos(\theta_1 + \theta_2) + a_1\cos\theta_1 = 9.0$$
$$y = a_2\sin(\theta_1 + \theta_2) + a_1\sin\theta_1 = 9.99999 \approx 10$$

2.3.2 Dual Two Link Parallel Manipulator

The Delta configuration is not just found in *Pick and Place* machines but has also become popular with the 3D printing community. This style of printer is fast and accurate. Just to get started, we look at a two dimensional analog shown in Fig. 2.16. The top (red) is fixed and is of length L_0. The two links on either side shown in dark blue are connected by servos (in green). These links are of length L_1. The angles are measured from the dotted line (as 0 degrees) to straight down (90 degrees), see Fig. 2.17. At the other end of the dark blue links is a free rotational joint (pivot). That connects the two light blue links which are joined together at the bottom with a rotational joint.

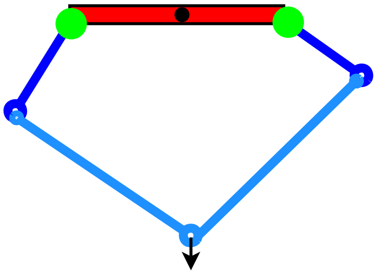


Fig. 2.16: Parallel Two Link Manipulator.

Unlike the previous two link manipulator, it is not completely obvious what the workspace looks like (although you might guess something elliptical). The configuration space is the space of all possible angles. This is limited by the red base in theory and by the servos in practice. Since 360° motion for the servos is not possible, the configuration space is a simple square $[\theta_m, \theta_M]^2$ where θ_m, θ_M are the minimum and maximum servo angles respectively.

Define the coordinate system as x is positive right and y is positive down. The origin is placed in the center of the red base link. The question is to figure out the position of the end effector at (x, y) as a function of θ_1 and θ_2 with fixed link lengths L_0, L_1, L_2, Figure Fig. 2.17. As with the serial chain manipulator, this is an exercise in trigonometry.

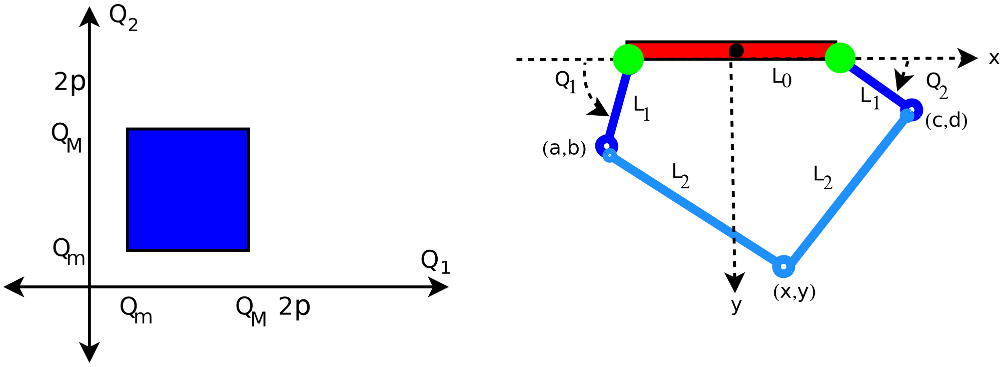


Fig. 2.17: Parallel Two Link (a) configuration space (b) with coordinates

The forward kinematics will provide (x, y) as a function of (θ_1, θ_2). The derivation is left as an exercise and so the point (x, y) is given by

$$(x, y) = \left(\frac{a+c}{2} + \frac{v(b-d)}{u}, \frac{b+d}{2} + \frac{v(c-a)}{u} \right) \tag{2.5}$$

Where

$$(a, b) = (-L_1 \cos(\theta_1) - L_0/2, L_1 \sin(\theta_1))$$

$$(c, d) = (L_1 \cos(\theta_2) + L_0/2, L_1 \sin(\theta_2))$$

and $u = \sqrt{(a-c)^2 + (b-d)^2}$, $v = \sqrt{L_2^2 - u^2/4}$.

If you guessed that the workspace was an ellipse like the author did, that would be wrong. If you guessed some type of warped rectangle, then you have great intuition. Fig. 2.18 shows the workspace for the configuration domain $[0, \pi/2]^2$. The figure graphs y positive going upwards and for the manipulator y positive goes down (so a vertical flip is required to match up). The workspace can be created by running a program that traces out all the possible arm angles and plots the resulting end effector position (not all points, but a dense sample of points will do just fine). Sample code to plot this workspace is given in Listing 2.1. It uses a double loop over θ_1 and θ_2, which places these values in the forward kinematics and then gathers the resulting (x, y) values. Like the serial manipulator, this is a holonomic robot as well.

Listing 2.1: Configuration Domain Code

```python
from math import *
import matplotlib.pyplot as plt

# Set the link lengths
L0 = 8
L1 = 5
L2 = 10

# Initialize the arrays
xlist = []
ylist = []

# Loop over the two angles,
#  stepping about 1.8 degrees each step
for i in range(100):
    for j in range(100):
        th1 = 0 + 1.57*i/100.0
        th2 = 0 + 1.57*j/100.0

        a = -L1*cos(th1) - L0/2.0
        b = L1*sin(th1)
        c = L1*cos(th2) + L0/2.0
        d = L1*sin(th2)

        dx = c-a
        dy = b-d
        u = sqrt(dx*dx+dy*dy)
        v = sqrt(L2*L2 - 0.25*u*u)

        x = (a+c)/2.0 + v*dy/u
        y = (b+d)/2.0 + v*dx/u

        xlist.append(x)
        ylist.append(y)

plt.plot(xlist,ylist, 'b.')
plt.show()
```

The inverse kinematics will give you (θ_1, θ_2) as a function of (x, y). This is another exercise in trigonometry.

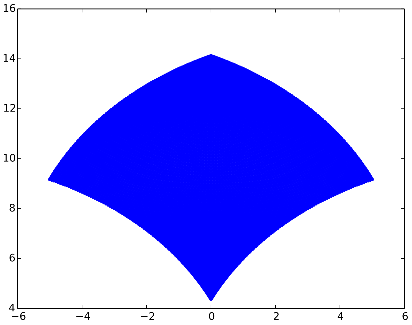


Fig. 2.18: Parallel Two Link Workspace

For (x, y) given, we obtain

$$\theta_1 = \pi - \beta - \eta, \qquad \theta_2 = \pi - \alpha - \gamma \tag{2.6}$$

where

$$\|G\| = \sqrt{(x - L_0/2)^2 + y^2}, \qquad \|H\| = \sqrt{(x + L_0/2)^2 + y^2}$$

$$\alpha = \cos^{-1}\frac{G^2 + L_0^2 - H^2}{2GL_0}, \qquad \beta = \cos^{-1}\frac{H^2 + L_0^2 - G^2}{2HL_0}$$

$$\gamma = \cos^{-1}\frac{G^2 + L_1^2 - L_2^2}{2GL_1}, \qquad \eta = \cos^{-1}\frac{H^2 + L_1^2 - L_2^2}{2HL_1}$$

Listing 2.2 illustrates using the inverse kinematic formulas for a specific pair of (x, y) values.

Listing 2.2: Inverse Kinematics Code for Parallel Two Link

```
from math import *
# Set the link lengths and starting location
L0 = 8
L1 = 5
L2 = 10
x = 0.2
y = 0.1*x + 10

# Compute IK
G = sqrt((x-L0/2.0)*(x-L0/2.0)+y*y)
H = sqrt((x+L0/2.0)*(x+L0/2.0)+y*y)

alpha = acos((G*G + L0*L0 - H*H)/(2.0*G*L0))
beta = acos((H*H + L0*L0 - G*G)/(2.0*H*L0))
gamma = acos((G*G + L1*L1 - L2*L2)/(2.0*G*L1))
eta = acos((H*H + L1*L1 - L2*L2)/(2.0*H*L1))

th1 = pi - beta - eta
th2 = pi - alpha - gamma

print th1, th2
```

If we want to convert a list of (x, y) points like we saw in previous examples, we needed to embedd our code into a loop. Using NumPy and SciPy one can leverage existing code considerably, Listing 2.3. The scalar (single) operations can be made into array operations (a type of iterator) with little change in the code. The normal arithmetic operators are overloaded and the iteration is done elementwise. Although Python is normally much slower than a C equivalent, numpy is highly optimized and the code runs close to the speed of C.[1]

Listing 2.3: Inverse Kinematics Code for Parallel Two Link Using Numpy

```
import numpy as np
from math import *
# Set the link lengths and
L0 = 8
L1 = 5
L2 = 10
x = np.arange(-3, 3, 0.2)
y = 0.1*x + 10

# Work out the IK
G = np.sqrt((x-L0/2.0)*(x-L0/2.0)+y*y)
H = np.sqrt((x+L0/2.0)*(x+L0/2.0)+y*y)

alpha = np.arccos((G*G + L0*L0 - H*H)/(2.0*G*L0))
beta = np.arccos((H*H + L0*L0 - G*G)/(2.0*H*L0))
gamma = np.arccos((G*G + L1*L1 - L2*L2)/(2.0*G*L1))
eta = np.arccos((H*H + L1*L1 - L2*L2)/(2.0*H*L1))

th1 = pi - beta - eta
th2 = pi - alpha - gamma

print th1, th2
```

The command np.arange generates a range of values starting at -3, ending at 3 and stepping 0.2. The x array can be manipulated with simple expressions to yield the y array (four function operator expressions acting pointwise on the arrays). To gain functions that act pointwise on the numpy arrays, you need to call them from the numpy library such as np.sqrt. Very little modification is required to get array operations in Python. To see how this works, comment out all of the previous code block. Then uncomment and run (adding a print statement to see the variable values) line by line. SciPy is very powerful and we will use many more features in later chapters.

2.4 Mobile Disk Robot

The next example a simple mobile ground robot in the shape of a small disk. The workspace is region in 2D for which the devices can operate, meaning "drive to". Mobile robots are not rooted to some point (an origin) through a chain of links. This is simply not the case for mobile systems. Configuration space is then taken to be the orientation and location of the robot. This turns out to be a complicated problem. Clearly

[1] Well, this is true on one Tuesday afternoon a long time ago with one little comparison of some loop/math code. Your results may be very different.

it depends on the underlying drive system. We will later study a drive system known as differential drive. Using differential drive, we are able to move to any position for which there is a sufficiently clear path (to be explained below). The differential drive can rotate in place to orientation is not a problem. The freedom to orient in place is not something found in automobiles which tend to have a smaller configuration space than differential drive systems.

For this example, we assume we have something like the differential drive robot. Assume that you have a simple mobile robot with two driven wheels and a third free unpowered wheel which can easily pivot or slide, Fig. 2.19 or Fig. 2.20. The drive wheels are not steered but can be spun at different rates which will steer the robot. This system is known as differential drive and is roughly analogous to how a tank drive operates. It is necessary to develop equations of motions for two reasons. The first reason is for simulating the dynamics or motion of the robot so we can see the results of our robot control software. The second reason is that the equations will be required in localization algorithms.

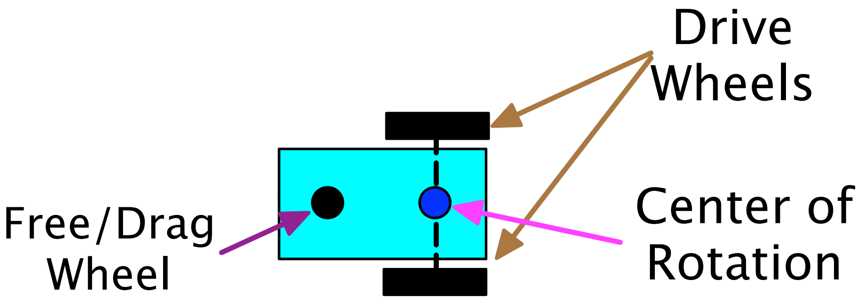


Fig. 2.19: Rectangular frame.

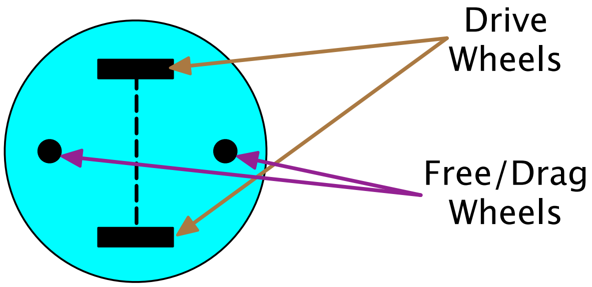


Fig. 2.20: Circular frame.

2.4.1 Reference Frames

There are two frames of reference that are used. The coordinate system used in the environment (without a robot around) is known as the global or inertial reference frame. It is the predetermined coordinate system that everyone will use. It is also a static coordinate system which we assume does not change. In a simulation, we normally take x to be along the horizontal direction with respect to the screen. The coordinate y is taken as the vertical screen direction and z points out of the screen. If we are working with actual robots, then it is whatever the coordinate system that exists in the area.

The other coordinate system used is one relative to the robot and is known as the local coordinate system.

You can think of it as a mini coordinate system for an ant living on the robot. We will use the convention that x points forward or in the direction of travel. y is set along the wheel axle and z is in the vertical direction. To remove any ambiguity, we assume that x, y, z also follow a right hand rule (which in this case sets the direction of y).

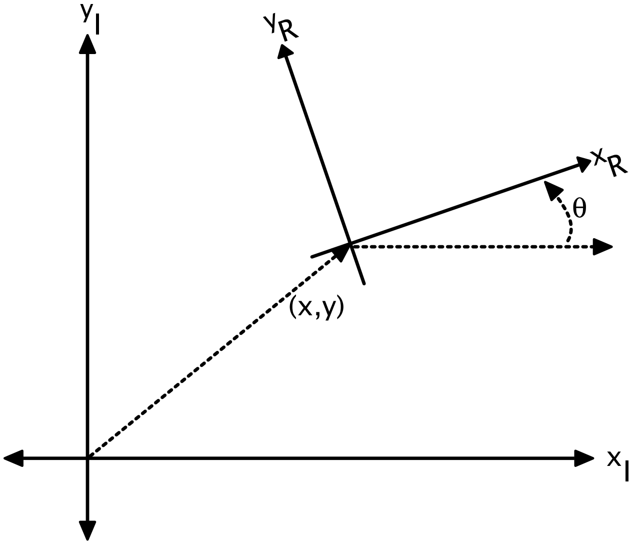


Fig. 2.21: The global and local frames of reference.

The global coordinate system already has an origin defined. However, we can choose the local frame origin. Our choice to simplify the mathematics by using the center of rotation of the vehicle. Thus when the robot rotates, the origin of the local coordinate system remains fixed. For planar motion, we don't really need to track z movement so we will drop z for now. The global or inertial basis will be identified as X_I, Y_I, and the local or relative basis will be identified as X_R, Y_R.

Any point in the plane can be represented in either coordinate system. So a particular point p can have coordinates (x_I, y_I) and (x_R, y_R). How do these relate? In two dimensions, a coordinate system can be translated and rotated relative to another. We can write this translation as the displacement of one origin to another or in our case, we can just use the location of the robot (local frame) origin relative to the global frame. In other words, the local frame origin position is (x_I, y_I). We then need to track the orientation of the frame or in our case the robot. The angle, θ, can be measure from either coordinate system and to be consistent, we take it as the angle from the global frame to the local frame. Graphically θ the amount of rotation applied to X_I to line it up with X_R.

We can track the robot position by tracking its coordinate system origin and orientation relative to the global coordinate system, ξ_I. So, we define the object relative to the robot by coordinates ξ_R and rotate into the inertial frame:

$$\xi_I = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix}, \quad \xi_R = \begin{pmatrix} x' \\ y' \\ 0 \end{pmatrix}.$$

The movement of the robot traces a path, $x(t)$, $y(t)$, in the global coordinate system which is our motion in the environment or in the simulation window. It is possible to track this motion through information obtained in the local frame. In order to do this, we need a formula to relate global and local frames. Using

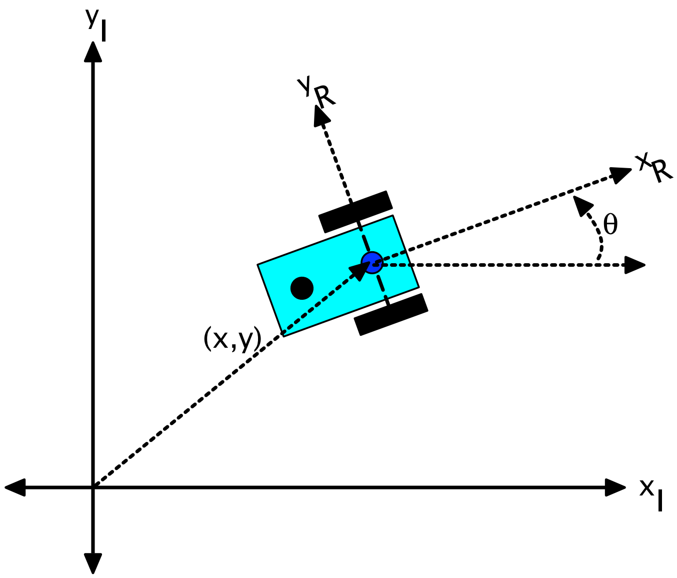


Fig. 2.22: The two frames of reference for a mobile robot: the inertial or global frame and the relative or local frame.

the standard tools from Linear Algebra, the relation is done through translation and rotation matrices. The rotation matrix:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

For example, a 45 degree rotation, $\theta = 45°$, produces a rotation matrix

$$R(\theta) = \begin{bmatrix} \sqrt{2}/2 & -\sqrt{2}/2 & 0 \\ \sqrt{2}/2 & \sqrt{2}/2 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The relation depends on the orientation of the robot which changes as the robot navigates in the plane. However, at a snapshot in time, the robot does have an orientation so, we can relate orientation at an instantaneous time:

$$\dot{\xi_I} = R(\theta)\dot{\xi_R}.$$

We can undo the rotation easily. Since R is an orthogonal matrix, the inverse is easy to compute, [Str88]

$$R(\theta)^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

You may have noted that we are not working with a translation. This is not required for the instantaneous coordinates because the derivative removes the translation.

2.4.2 Equations of Motion

Working in instantaneous local coordinate enables us to determine the motion easily. We then use the rotation matrix to relate the robot position in the global frame. To progress in the modeling process, we need to know the specifics of the robot, illustrated in Fig. 2.23.

- Wheel size: D, so the radius $r = D/2$
- Axle length: $2L$ (L is the distance from the origin of the coordinate system to a wheel)
- Origin of local coordinate system: P is placed on the midpoint of the axle.

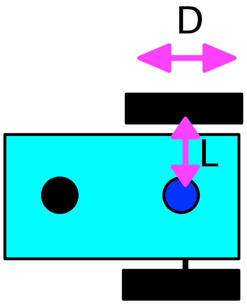


Fig. 2.23: Robot Dimensions.

Recall that the goal was to compute the motion of the robot based on the rotational speed of the wheels. Let $\dot{\phi}_1$ and $\dot{\phi}_2$ be the right and left wheel rotational speeds (respectively). Note: ϕ is an angle and measured in radians, $\dot{\phi}$ is measured in radians per unit time, and $\dot{\phi}/2\pi$ is the "rpm" (or rps, etc).

Next we determine the contribution of each wheel to linear forward motion. The relation between linear and angular velocities gives us for the right wheel $\dot{x_1} = r\dot{\phi}_1$ and for the left wheel: $\dot{x_2} = r\dot{\phi}_2$, Fig. 2.24. The differential speeds then produce the rotational motion about the robot center and the average forward velocity.

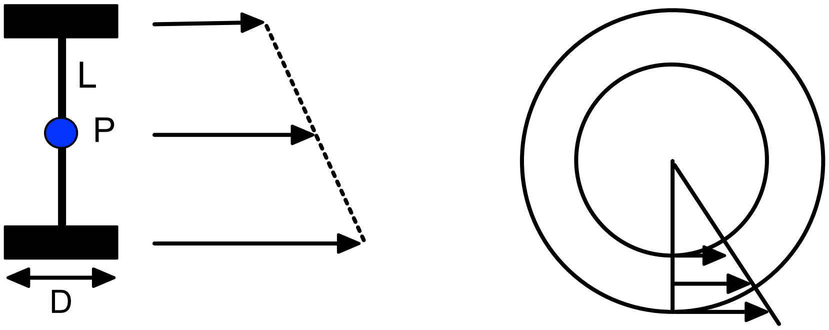


Fig. 2.24: Velocity of axle induced by wheel velocities.

The speed of point P is given by the weighted average based on distances of the wheels to P. To see this, we consider a couple of cases. If the two wheel velocities are the same, then the average works trivially. If the two velocities are different (but constant), then the motion of the robot is a circle. Fig. 2.24 shows the robot motion. Assuming the outer circle radius is $\rho + 2L$ with velocity $r\dot{\phi}_1$ and the inner circle is radius ρ

with wheel velocity $r\dot{\phi}_2$, we have that the motion of a similar wheel at point P would be:

$$\frac{\dot{\phi}_2}{\rho} = \frac{\dot{\phi}_1}{\rho + 2L} = \frac{\dot{\phi}_P}{\rho + L}.$$

Solving for ρ with the left two terms: $\rho = 2L\dot{\phi}_2/(\dot{\phi}_1 - \dot{\phi}_2)$. Using the outer two terms, plug in for this value of ρ:

$$\frac{\dot{\phi}_2}{2L\dot{\phi}_2/(\dot{\phi}_1 - \dot{\phi}_2)} = \frac{\dot{\phi}_P}{2L\dot{\phi}_2/(\dot{\phi}_1 - \dot{\phi}_2) + L} \Rightarrow \frac{\dot{\phi}_1 + \dot{\phi}_2}{2} = \dot{\phi}_P$$

This velocity is in the direction of x_R.

$$\dot{x_R} = r\dot{\phi}_P = \frac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)$$

For this example, there is no motion parallel to the axle so $\dot{y_R} = 0$.

Each wheel will act like a lever arm rotating the craft as well as moving it forward. To determine the amount of rotation, we examine the contribution of the wheels separately. For example, if the right wheel moves faster than the left wheel, then we have positive rotation of the vehicle. The contribution from the right wheel is $2L\dot{\theta} = r\dot{\phi}_1$ or $\dot{\theta} = r\dot{\phi}_1/(2L)$ and the contribution from the left wheel is $2L\dot{\theta} = -r\dot{\phi}_2$ or $\dot{\theta} = -r\dot{\phi}_2/(2L)$, see Figure Fig. 2.25. The rotation about P is given by adding the individual contributions:

$$\dot{\theta} = \frac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2).$$

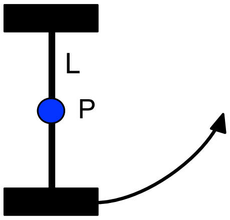


Fig. 2.25: The contribution of the two wheels towards rotational motion.

In local or robot coordinates we obtain the following equations of motion

$$\begin{aligned}
\dot{x_R} &= \tfrac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2), \\
\dot{y_R} &= 0, \\
\dot{\theta} &= \tfrac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2).
\end{aligned}$$

To get the model in global (or inertial) coordinates we must apply the transformation (the rotation) to our local coordinate model. This is done by applying the rotation matrix R to the position vector $\dot{\xi}_R$:

$$\dot{\xi}_I = R(\theta)\dot{\xi}_R = R(\theta)\begin{bmatrix} \frac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2) \\ 0 \\ \frac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2) \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{r}{2}(\dot{\phi_1} + \dot{\phi_2}) \\ 0 \\ \frac{r}{2L}(\dot{\phi_1} - \dot{\phi_2}) \end{bmatrix} = \begin{bmatrix} \frac{r}{2}(\dot{\phi_1} + \dot{\phi_2})\cos(\theta) \\ \frac{r}{2}(\dot{\phi_1} + \dot{\phi_2})\sin(\theta) \\ \frac{r}{2L}(\dot{\phi_1} - \dot{\phi_2}) \end{bmatrix}$$

This leads to the following equations of motion in the global reference frame:

$$\boxed{\begin{aligned} \dot{x} &= \tfrac{r}{2}(\dot{\phi_1} + \dot{\phi_2})\cos(\theta) \\ \dot{y} &= \tfrac{r}{2}(\dot{\phi_1} + \dot{\phi_2})\sin(\theta) \\ \dot{\theta} &= \tfrac{r}{2L}(\dot{\phi_1} - \dot{\phi_2}) \end{aligned}} \tag{2.7}$$

These are non-holonomic constraints, this is left as a homework exercise.

Assume that you have a differential drive robot. If the drive wheel is 20cm in diameter and turns at 10 rpm (revolutions per minute), what is the linear speed of the rolling wheel (with no slip or skid)?

We see that distance covered $s = \theta r$

and so $v = ds/dt = r d\theta/dt$. Note that $d\theta/dt = 2\pi\omega$, where ω is the rpm. So

$$v = 2\pi r\omega = 2\pi * 10 * 10 = 200\pi.$$

Let the distance between the wheels be 30cm (axle length). If the right wheel is turning at 10 rpm (revolutions per minute) and the left is turning at 10.5 rpm, find a formula for the resulting motion.

As stated earlier, the motion for this robot would be a circle. Thus the two wheels trace out two concentric circles Fig. 2.26. The two circles must be traced out in the same amount of time:

$$t = \frac{d_1}{v_1} = \frac{d_2}{v_2} \Rightarrow \frac{d_1}{10.5 * 20\pi} = \frac{d_2}{10 * 20\pi} \Rightarrow \frac{2\pi(R+30)}{210\pi} = \frac{2\pi R}{200\pi}$$

$$\frac{30}{105} = R\left(\frac{1}{100} - \frac{1}{105}\right) = \frac{5R}{100 * 105}$$

$$\Rightarrow R = \frac{100 * 105}{5}\frac{30}{105} = 600$$

Thus we have $x^2 + y^2 = 600^2$ as the basic formula for the curve of motion.

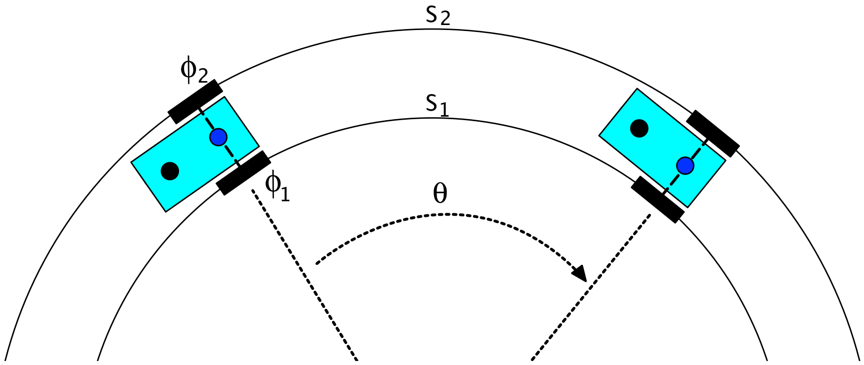


Fig. 2.26: A differential drive robot with constant wheel velocity drives in straight lines and circles.

We can attempt to formally integrate the differential drive equations

$$\dot{\theta} = \frac{d\theta}{dt} = \frac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2)$$

integrate from 0 to t (and be careful about integration variables)

$$\int_0^t \frac{d\theta}{d\tau}\, d\tau = \int_0^t \frac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2)\, d\tau$$

and we have

$$\theta(t) - \theta(0) = \int_0^t \frac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2)\, d\tau$$

Assume that you know $\phi_i(t)$ (or if you know $\dot{\phi}_i(t)$), then what can you say?

From $\dot{\phi}_i(t)$ we can compute θ by integrating the last equation:

$$\theta(t) = \theta(0) + \int_0^t \frac{r}{2L}\left(\frac{d\phi_1}{d\tau} - \frac{d\phi_2}{d\tau}\right) d\tau$$

Using this result we can write down formulas for x and y

$$x(t) = x(0) + \int_0^t \frac{r}{2}\left(\frac{d\phi_1}{d\tau} + \frac{d\phi_2}{d\tau}\right)\cos(\theta(\tau))d\tau$$

$$y(t) = y(0) + \int_0^t \frac{r}{2}\left(\frac{d\phi_1}{d\tau} + \frac{d\phi_2}{d\tau}\right)\sin(\theta(\tau))d\tau$$

These equations are easy to integrate if you know the wheel velocities are constants. First integrate the θ equation:

$$\theta(t) = (r/2L)(\omega_1 - \omega_2)t + \theta_0$$

This can be plugged into the x and y equations and then integrated:

$$x(t) = \frac{L(\omega_1 + \omega_2)}{(\omega_1 - \omega_2)}\left[\sin((r/2L)(\omega_1 - \omega_2)t + \theta_0) - \sin(\theta_0)\right] + x(0)$$

$$y(t) = -\frac{L(\omega_1 + \omega_2)}{(\omega_1 - \omega_2)}\left[\cos((r/2L)(\omega_1 - \omega_2)t + \theta_0) - \cos(\theta_0)\right] + y(0)$$

Thus the solution is a sequence of circular arcs.

From these solutions (and from the differential equations as well), you can see that there is a problem when $\omega_1 = \omega_2$ or when $\omega_1 = -\omega_2$. These are exactly the cases we are often given. However, the equations are trivial (meaning they reduce) for these special cases. When $\omega_1 = \omega_2$,

$$d\theta/dt = 0$$

and when $\omega_1 = -\omega_2$

$$dx/dt = 0, dy/dt = 0.$$

So, you can just work out the solution from rate-time-distance formulas.

Solve these equations for the given values of $\omega_1 = \dot{\phi}_1$ and $\omega_2 = \dot{\phi}_2$ below. Assume that the wheels are 18cm in diameter and L is 12cm. Find an analytic solution and compute the position of the robot starting at t=0, x=0, y=0, theta=0, after the following sequence of moves:

$t = 0 \to 5$:	$\omega_1 = \omega_2 = 3.0$,
$t = 5 \to 6$:	$\omega_1 = -\omega_2 = 2.0$,
$t = 6 \to 10$:	$\omega_1 = \omega_2 = 3.0$,
$t = 10 \to 11$:	$\omega_1 = -\omega_2 = -2.0$,
$t = 11 \to 16$:	$\omega_1 = \omega_2 = 3.0$,

Begin at $(x, y, \theta) = (0, 0, 0)$

$t = 0 \to 5$:	$\omega_1 = \omega_2 = 3.0$, $\Rightarrow (0, 0, 0) + (135, 0, 0) = (135, 0, 0)$
$t = 5 \to 6$:	$\omega_1 = -\omega_2 = 2.0$, $\Rightarrow (135, 0, 0) + (0, 0, 3/2) = (135, 0, 3/2)$
$t = 6 \to 10$:	$\omega_1 = \omega_2 = 3.0$, $\Rightarrow (135, 0, 3/2) + (108 \cos 3/2, 108 \sin 3/2, 0) \approx (142.6, 107.7, 1.5)$
$t = 10 \to 11$:	$\omega_1 = -\omega_2 = -2.0$, $\Rightarrow (142.6, 107.7, 1.5) + (0, 0, -1.5) = (142.6, 107.7, 0)$
$t = 11 \to 16$:	$\omega_1 = \omega_2 = 3.0$, $\Rightarrow (142.6, 107.7, 0) + (135, 0, 0) = (277.6, 107.7, 0)$

You may have noticed that these equations related derivatives of the parameters and variables. Hence these are known as differential equations. Specifically these are nonlinear differential equations due to the sine and cosine terms. The standard methods seen in elementary courses such as Laplace Transforms and Eigenvector Methods do not apply here. However, there is enough structure to exploit that one can solve the equations in terms of the wheel rotations. So, if you know ϕ_1 and ϕ_2, you can determine position by integration. They are used to track the position of the middle of the robot. The derivation of these equations and the inverse kinematics will be discussed in the motion modeling chapter. We will also hold off on running some path computations as we did with the manipulators and show those in the motion chapter as well. The next thing to address is the workspace and configuration space.

The main difference for our mobile robot is that for the manipulators we only focused on the end effector position. We tracked the single point which was at the tip of the end effector. In real situations, however, we may need to track the entire manipulator. Surgical robots are a fine example. They have to operate in very narrow corridors to reduce skin incisions. In those cases a full geometric model may be required and constraints are placed on all of the intermediate links. For mobile robots, this problem seems to arise often.

If the mobile robot was extremely small, like a point, it is pretty easy to deal with. There is only the point to track, no orientation to worry about and we don't worry about any manipulator that got it there. A relatively small robot that can move in any direction can be approximated by a point robot. In this case, the workspace and configuration spaces are identical and two dimensional. Although this seems a bit silly to treat our robot as a point, it can be a useful simplification when planning routes for the robot. You can also think of this as tracking the centroid of the robot. If there is no admissible route for the centroid, then no route exists. The computation for the point can be much faster than the computation that includes the full geometry.

Unfortunately, our robots do have size. A circular robot would be the natural next step to investigate. The question is what is the effect on the configuration and workspace. If the robot is round, has no orientation and can move in any direction, then again the configuration and workspaces are the same. By moving the robot around in the world and tracking the centroid, we can determine the configuration space. Since the middle of the robot cannot touch the obstacle boundary, the interaction between the robot and the obstacle reduces the configuration space as shown in Fig. 2.27, Fig. 2.28. In this case the size of the robot affects the configuration space, Fig. 2.29. For a mobile ground robot that is not a point, orientation will enter as a variable in the system.

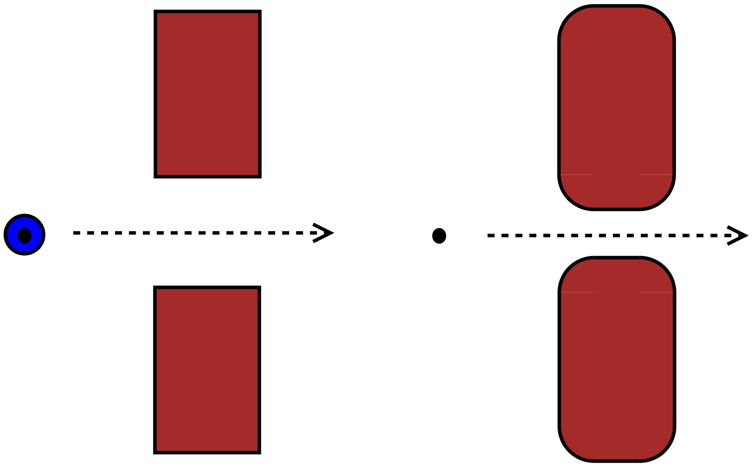


Fig. 2.27: Configuration space as a function of robot size.

For a round or disk robot with radius, r, the center of the robot can only get to within distance r of an obstacle boundary. Assume the obstacle is also round with radius, R and is the only one. The configuration space for the robot is all of the points that the robot centroid can reach. This situation is the same as if the robot was a point and the obstacle had radius $R + r$. We can study the configuration space problem by shrinking the robot to a point and *inflating* the obstacle by the robot's radius. This can be done for all the obstacles in the workspace. It is clear that the obstacle does not need to be round. Move the robot up to the place where it touches the obstacle. Mark the robot's center on the workspace. Do this for all points of contact between the robot and the obstacle. This draws an outer boundary around the obstacle and makes the obstacle larger. We have inflated the obstacle.

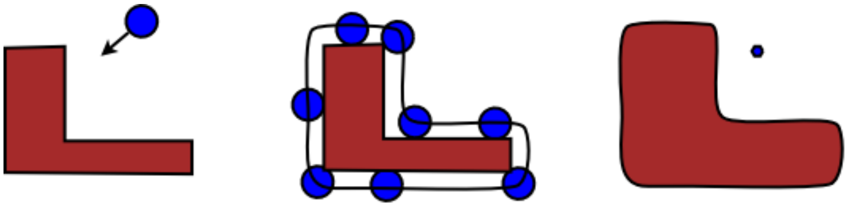


Fig. 2.28: Example of the inflation process.

The previous examples looked at a circular robot. What about a robot which is a rectangle? What would be the configuration space about some obstacle? Fig. 2.30. The basic shape of the robot is important as well as its orientation, Fig. 2.31. Inflation in this case depends on the fixed orientation of the robot. One follows the same process and pushes the robot up until it touches the obstacle. Doing this for all locations around the obstacle all while keeping the same orientation will describe the configuration space. Marking the robot's centroid at each contact allows us to trace a curve around the obstacle and thus inflate the obstacle. We then

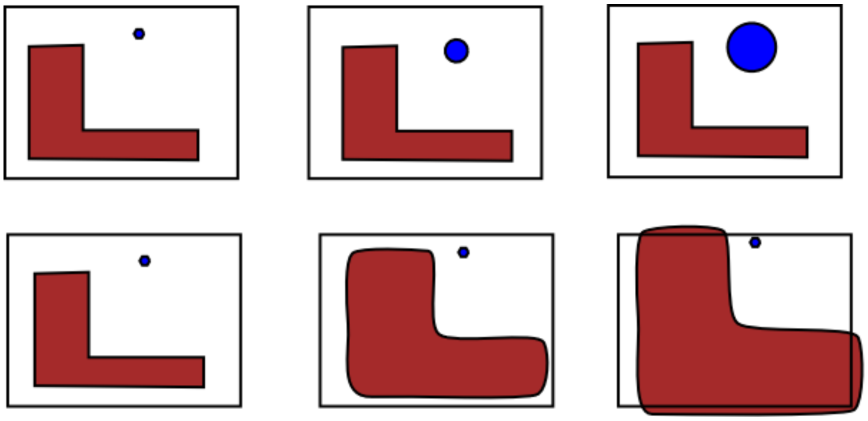


Fig. 2.29: Relation between robot size and configuration space.

can shrink the robot to a point. We can then study robot paths through the open space. Of course in practice this is absurd since the robot orientation is not fixed. But it does help transition to the general case.

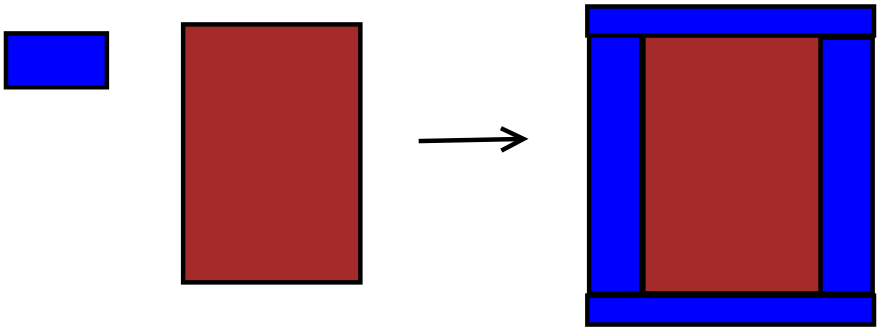


Fig. 2.30: Changing robot shape also affects c-space.

It is helpful to see some examples of the inflation process. A rectangular object does not just change scale. It changes shape as well. For a rectangle, he inflated obstacle is a “rectangle” with rounded corners. It is important to note that each rotation of the rectangle generates a new and different configuration space, Fig. 2.32. This process can be very complicated and often one will want to make simplifications.

Robot orientation then makes the configuration space question more complicated since the configuration space is a function of the robot orientation. A planning algorithm would then need to either fix the robot orientation or be able to adjust to a changing landscape. To fix orientation ultimately means that the orientation is independent of travel direction. This is not the case for the vast majority of vehicles. The orientation for a car, for example, is pointed in the direction of travel.[1] To obtain this independence a holonomic robot is required. The term holonomic will be carefully defined later, for now, consider it a mobile robot that can set position and orientation independently. Independent of the type of motion, it should be clear now that position and orientation are separate and important variables in the system which is addressed next.

[1] Under normal conditions this is true, however, icy roads will allow for much greater freedom of vehicle orientation and travel direction.

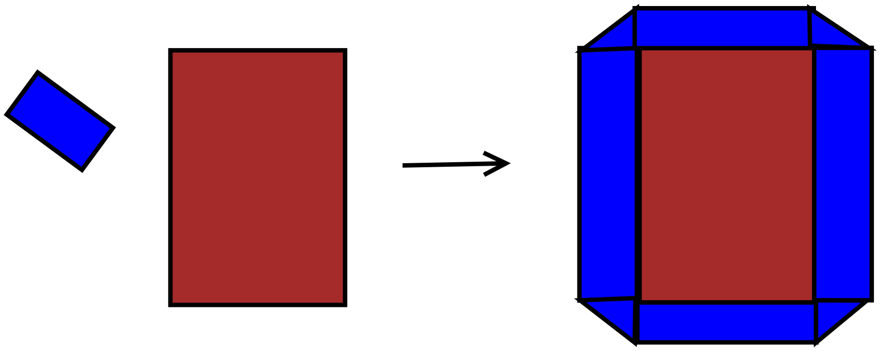


Fig. 2.31: Changing robot orientation affects c-space as well.

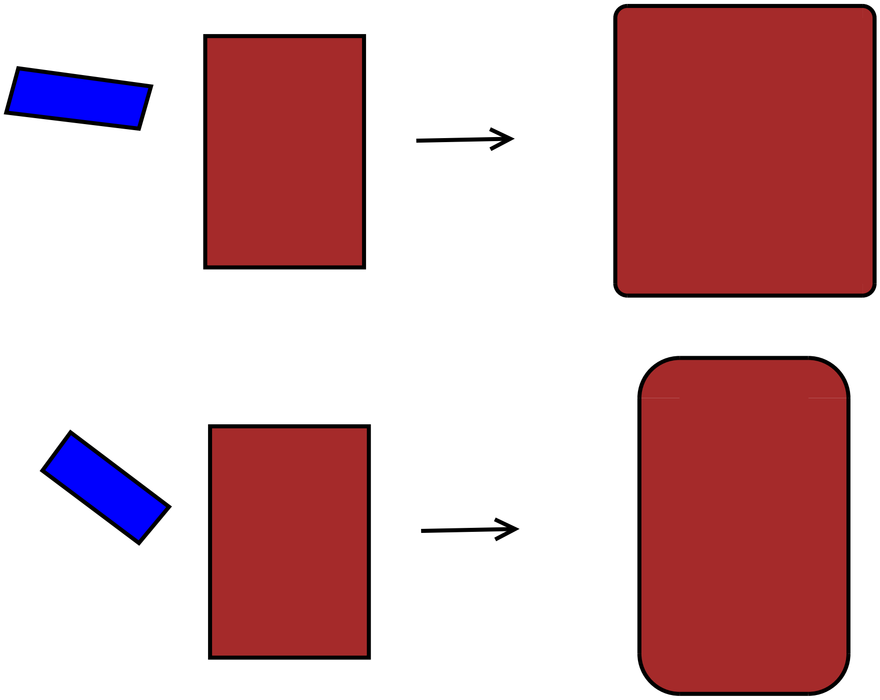


Fig. 2.32: Two sample rotations and the configuration obstacle.

2.5 Constraints and Kinematics

Constraints on a mechanical system are the conditions which restrict possible geometric positions or limits certain motions. The are written as expressions of the state variables in the system. It is useful to distinguish geometric and kinematic constraints. The term geometric is concerned only with position where kinematic also includes motion. We will use the term *Holonomic* instead of geometric since it is standard usage in robotics.

Kinematics describes the geometry of motion. It describes the motion through a set of constraints on the way the robot will move through space. For rigid bodies, we focus on displacement and orientation for which the kinematics restricts in some manner. Assume that you want to move in the plane from (x_1, y_1) to (x_2, y_2). If you are driving a traditional front wheel steer automobile (Ackerman Steering), then your final orientation depends on your path. If you drive straight then your final orientation is in line with the line between the start and end points. However, you could have made a large detour and ended up at another orientation as shown in Fig. 2.33.

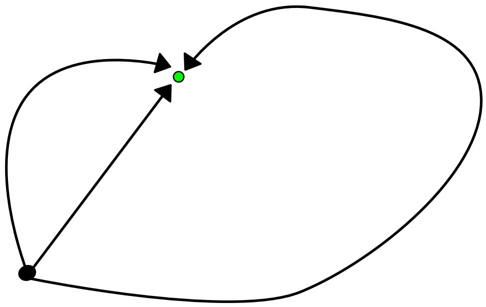


Fig. 2.33: Final orientation depends on path.

Assume you decide to replace your auto wheels with caster wheels and have someone push you. In this case you can travel from point to point with arbitrary orientation.[1] This simple example implies that we have two fundamentally different types of motion. One that depends on the path and one that does not. The independence of path boils down to the types of motion constraints given by the system. Our goal here is to formally describe these two types of constraints. You may notice a strong similarity between what we are discussing here and the concepts of independence of path and conservative vector fields taught in calculus. Indeed these concepts are related. For this section, let $x_i(t)$ be coordinate variables.

2.5.1 Kinematic Constraints

A constraint is called kinematic if one can express it as

$$f(x_1, x_2, \dots, x_n, \dot{x}_1, \dot{x}_2, \dots, \dot{x}_n, t) = 0$$

f is a function in phase space for the system. This constraint places restrictions on motion through the expression relating velocities and positions.

[1] Like the office chair races in the hallway.

2.5.2 Pfaffian Constraints

Often the constraints appear linear in the velocity terms as

$$\sum_i F_i(x)\dot{x}_i = 0$$

and are known as Pfaffian constraints. This can be written as $F \cdot \dot{x} = 0$. Which states that the motion of the system, $\dot{x}$, is orthogonal to the vector field F. For multiple constraints, these can be bundled as rows into a constraint matrix **F**:

$$\mathbf{F}\dot{x} = 0$$

so the motion $\dot{x}$ is along the nullspace of **F**.

2.5.3 Holonomic Constraints

A constraint is called holonomic (or geometric) if it is integrable or one can express it as

$$h(x_1, x_2, \ldots, x_n, t) = 0 \tag{2.8}$$

A holonomic constraint only depends on the coordinates and time and does not depend on derivatives. If all the system constraints are holonomic then we say the system is *holonomic*. Otherwise we say the system is *non-holonomic*. Wikipedia has a nice way of expressing non-holonomic:

> A nonholonomic system in physics and mathematics is a system whose state depends on the path taken in order to achieve it. Such a system is described by a set of parameters subject to differential constraints, such that when the system evolves along a path in its parameter space (the parameters varying continuously in values) but finally returns to the original set of parameter values at the start of the path, the system itself may not have returned to its original state.

A holonomic constraint implies a kinematic constraint:

$$\frac{dh(x)}{dt} = \sum_{i=1}^{n} \frac{\partial h(x)}{\partial x_i}\dot{x}_i = \sum_i f_i(x)\dot{x}_i, \quad \text{where} \quad f_i(x) = \frac{\partial h(x)}{\partial x_i}$$

But it is not true in general the other way around. It should be clear that if the expression is not in Pfaffian form, then it cannot integrated. This integrability is a special case. If the Pfaffian expression, $\sum_i f_i(x)\dot{x}_i$ is holonomic, then using a non-zero integrating factor $\sigma(x)$, we can integrate and express as

$$H(x) = c$$

This implies that the mechanical system is constrained to a level surface of H which depends on the initial configuration of the system. This reduces the degrees of freedom to $n-1$. Having k holonomic constraints then reduces the degrees of freedom to $n-k$.

An example of how a holonomic constraint may be used to reduce the number of degrees of freedom is helpful. If we want to remove x_k in the constraint equation f_i we algebraically rearrange the expression into the form

$$x_k = g_i(x_1,\ x_2,\ x_3,\ \ldots,\ x_{k-1},\ x_{k+1},\ \ldots,\ x_n,\ t),$$

and replace every occurrence of x_k in the system using the above expression. This can always be done, provided that f_i is C^1 so the expression g_i is given by the implicit function theorem. Then using this expression it is possible to remove all occurrences of the dependent variable x_k.

Assume that a physical system has N degrees of freedom and there are h holonomic constraints. Then, the number of degrees of freedom is reduced to $m = N - h$. We now may use m independent (generalized) coordinates q_j to completely describe the motion of the system. The transformation equation can be expressed as follows:

$$x_i = x_i(q_1,\ q_2,\ \ldots,\ q_m,\ t)\ , \qquad\qquad i = 1,\ 2,\ \ldots n.$$

For our use, it tells us about the maneuverability for the robot. For holonomic robots, the controllable degrees of freedom is equal to total degrees of freedom. Kinematic constraints restrict movement of the robot. Non-holonomic constraints restrict the motion without restricting the workspace. Holonomic constraints reduce the dimensionality of the workspace and restricts the motion of the robot. Having a non-holonomic constraint means that there are restrictions on velocity but less so on position. So local movement is restricted, but global positioning is less resricted.

2.5.4 Integrability Conditions

If the kinematic constraint is holonomic, then it comes from differentiating some function $f(t, x)$. So, we consider only first order expressions,

$$\frac{df}{dt} = \sum_{i=1}^{n} \frac{\partial f(t,x)}{\partial x_i}\dot{x}_i + \frac{\partial f(t,x)}{\partial t} = \sum_{i=1}^{n} a_i(x,t)\dot{x}_i + a_t(x,t) = 0. \tag{2.9}$$

These expressions are Pfaffian (linear in the velocity terms, $\dot{x_i}$). If your kinematic expression is nonlinear in velocities terms, it did not come from differentiation of a holonomic constraint. That is enough to eliminate many expressions as candidates.

Since the terms a_i are the partials $\partial f/\partial x_i$, the mixed partials are equal

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i} \Rightarrow \frac{\partial a_j}{\partial x_i} = \frac{\partial a_i}{\partial x_j}$$

Because the constraints are set to zero, it is possible that a common factor has been divided out

$$\sum_{i=1}^{n} a_i(x,t)\dot{x}_i + a_t(x,t) = \sum_{i=1}^{n} \sigma(x) b_i(x,t)\dot{x}_i + \sigma(x) b_t(x,t) = \sigma(x)\sum_{i=1}^{n} b_i(x,t)\dot{x}_i + b_t(x,t) = 0$$

$$\Rightarrow \sum_{i=1}^{n} b_i(x,t)\dot{x}_i + b_t(x,t) = 0$$

when $\sigma(x) \neq 0$.

The term σ is known as an integrating factor and it complicates the second partial test. Given a Pfaffian expression,

$$\sum_{i=1}^{n} b_i(x,t)\dot{x}_i + b_t(x,t) = 0$$

the second partial test appears as

$$\frac{\partial\left(\sigma(x) b_j\right)}{\partial x_i} = \frac{\partial\left(\sigma(x) b_i\right)}{\partial x_j} \tag{2.10}$$

2.6 Integration

To find the antiderivative, one can follow a fixed process. Assume that you are given the form $a_1(x_1, x_2)\dot{x_1} + a_2(x_1, x_2)\dot{x_2} = 0$. Since a_1 comes from a partial derivative with respect to x_1 then we should integrate with respect to that variable. This gives us some function A_1. We can do a similar process for a_2 and gain A_2. We use both a_2 and A_2 to find the correct term.

Examples: are the following holonomic?

1. $\dot{x_1} + \dot{x_2} = 0$. For this example, you can just integrate and see that $x_1 + x_2 = c$ is the antiderivative. So it is holonomic.

2. $x_2 e^{x_1}\dot{x_1} + e^{x_1}\dot{x_2} = 0$. Yes. Since

   $$\frac{\partial(x_2 e^{x_1})}{\partial x_2} = \frac{\partial(e^{x_1})}{\partial x_1}$$

   Integrate the first expression, $x_2 e^{x_1}$, wrt to x_1 and we obtain $h(x) = x_2 e^{x_1} + c$. Differentiate wrt to x_2 to verify no missing terms.

3. $x_2\dot{x_1} + x_1\dot{x_2} = 0$. Since

   $$\frac{\partial(x_2)}{\partial x_2} = \frac{\partial(x_1)}{\partial x_1} \Rightarrow 1 = 1$$

   it is holonomic. Integrate the first expression, x_2, wrt to x_1 and we obtain $h(x) = x_1 x_2 + c$. Differentiate wrt to x_2 to verify no missing terms.

4. $x_1\dot{x}_1 + x_2\dot{x}_2 + x_3\dot{x}_3 = 0$. There are several mixed partials to check. This constraint can be integrated to $x_1^2 + x_2^2 + x_3^2 = c$. which means this is a holonomic constraint.

5. $\dot{x}_1/x_2 + \dot{x}_2/x_1 = 0$ Note that the mixed partials do not agree. Multiply the expression by $x_1 x_2$ (a guess) and check

   $$\frac{\partial(x_1)}{\partial x_2} = \frac{\partial(x_2)}{\partial x_1}$$

   So the term $x_1 x_2$ is called the integrating factor and the constraint is holonomic.

6. $x_1\dot{x_1} + x_1 x_2\dot{x_2} = 0$ We try guessing a couple of integrating factors but none succeed. We seek a function $\sigma(x)$ so that

   $$\frac{\partial(\sigma(x)x_1)}{\partial x_2} = \frac{\partial(\sigma(x)x_1 x_2)}{\partial x_1}$$

   Expand and solve for σ

   $$\frac{\partial(\sigma(x)x_1)}{\partial x_2} = x_1\frac{\partial(\sigma(x))}{\partial x_2}$$

   and

   $$\frac{\partial(\sigma(x)x_1 x_2)}{\partial x_1} = x_1 x_2\frac{\partial(\sigma(x))}{\partial x_1} + \sigma(x)x_2$$

We can equate these

$$x_1\frac{\partial(\sigma(x))}{\partial x_2} = x_1x_2\frac{\partial(\sigma(x))}{\partial x_1} + \sigma(x)x_2$$

We try a simplification by assuming a form on $\sigma(x) = \sigma_1(x_1)\sigma_2(x_2)$. Divide the entire expression by $\sigma_1(x_1)\sigma_2(x_2)x_1x_2$ and we obtain

$$\frac{1}{x_2\sigma_2}\frac{\partial(\sigma_2)}{\partial x_2} = \frac{1}{\sigma_1}\frac{\partial(\sigma_1)}{\partial x_1} + \frac{1}{x_1}$$

The right side is a function of only x_1 and the left side only of x_2. The only way for them to be equal is if they are constant. Set each side to a constant, λ and solve the two resulting ordinary differential equations. This gives us both σ's.

$$\sigma_1 = \frac{c_1}{x_1}e^{\lambda x_1}, \ \ \sigma_2 = c_2e^{\lambda x_2^2/2} \Rightarrow \sigma = \frac{c}{x_1}e^{\lambda(x_1 - x_2^2/2)}$$

So we conclude this expression is holonomic. We also see that this was a very complicated route and there were multiple stages in which this process would stall. The general approach to finding an integrating factor requires finding an analytic solution to a quasi-linear first order partial differential equation which in general is not possible. In our application we try a few tricks to solve for the integrating factor and then look to see if we can prove none exists. The next example will illustrate this.

7. $\dot{x_1} + \dot{x_2} + x_1\dot{x_3} = 0$. Using (2.10) we gain the following equations

$$\frac{\partial\sigma}{\partial x_2} = \frac{\partial\sigma}{\partial x_1}$$

$$\frac{\partial\sigma}{\partial x_3} = \sigma + x_1\frac{\partial\sigma}{\partial x_1}$$

$$\frac{\partial\sigma}{\partial x_3} = x_1\frac{\partial\sigma}{\partial x_2}$$

Setting the second two equations equal

$$\sigma + x_1\frac{\partial\sigma}{\partial x_1} = x_1\frac{\partial\sigma}{\partial x_2}$$

Then use the first equation

$$\sigma + x_1\frac{\partial\sigma}{\partial x_1} = x_1\frac{\partial\sigma}{\partial x_1}$$

one concludes that $\sigma \equiv 0$ and so this constraint is *non-holonomic*.

8. The vertical rolling wheel produces a constraint of the form $\sin\theta\dot{x} - \cos\theta\dot{y} = 0$ where (x, y) is the location of the wheel (contact point) in the plane and θ is the orientation of the wheel. [This will be discussed in detail later.]

Apply (2.10) and we have

$$\sin\theta\frac{\partial\sigma}{\partial y} = -\cos\theta\frac{\partial\sigma}{\partial x}$$

$$\cos\theta\frac{\partial\sigma}{\partial\theta} = \sigma\sin\theta$$

$$\sin\theta\frac{\partial\sigma}{\partial\theta} = -\sigma\cos\theta$$

Squaring the last two equations and adding together, we gain $\partial\sigma/\partial\theta = \pm\sigma$ and plugging this back in to either gives $\pm(\cos\theta)\sigma = (\sin\theta)\sigma$. As with the previous example we can conclude that $\sigma = 0$ so the constraint is non-holonomic.

Systems of Pfaffian constraints are a more complicated matter. It is possible to have a collection of constraints which are individually non-holonomic, but the collection turns out to be integrable. The theory is outside the scope of this text and when we need a result we will quote the literature.

2.7 Problems

1. What is the difference between *effect* and *affect*?
2. What is the difference between accuracy and precision?
3. Define domain and range.
4. Is the following constraint holonomic: $\dot{x}_2\sin(x_1) + x_2\cos(x_1)\dot{x}_1 = 0$.
5. Sketch the workspace of a two-link manipulator centered at the origin with $a_1 = 15$ and $a_2 = 10$.
6. Assume that you have a two link planar manipulator. θ_1 is the angle between the x axis (measured counter-clockwise as positive) and the first link arm. θ_2 is the angle between the second link arm and the first link arm (again measured counter-clockwise as positive). Given the length of the first link $a_1 = 12$ and the second link $a_2 = 7$ solve the following:
   a. If $\theta_1 = 45°$, $\theta_2 = 45°$, find x and y.
   b. If $\theta_1 = 45°$, $\theta_2 = 45°$, $d\theta_1/dt = 5° s^{-1}$, $d\theta_2/dt = 10° s^{-1}$ find dx/dt and dy/dt.
   c. If $x = 12$, $y = 14$, find θ_1 and θ_2
7. Assume that you have a two link planar manipulator. θ_1 is the angle between the x axis (measured clockwise as positive) and the first link arm. θ_2 is the angle between the second link arm and the first link arm (again measured clockwise as positive). Due to servo limitations: $-100° < \theta_1 < 100°$, $-150° < \theta_2 < 150°$. Also assume the first link is 20cm long and the second link is 15cm.
   a. What is the configuration space?
   b. What is the workspace?
8. Find the forward velocity kinematics equations for the two link manipulator.
9. Assume that you have a two link manipulator that is operating in the vertical plane $x - z$. Attach the base to a rotational joint so the arm rotates around the z axis. See Fig. 2.34 .
   a. Find the position of the end effector as a function of joint angles.
   b. Find the inverse kinematic formula.

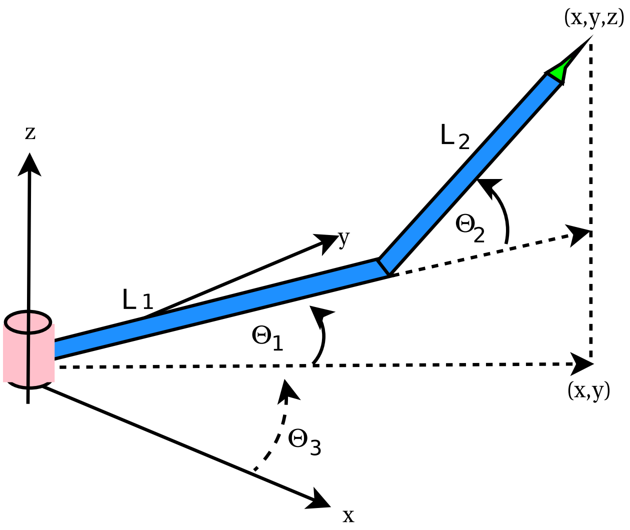


Fig. 2.34: Two link manipulator.

10. Assume that you have a two link manipulator with $a_1 = 15$cm and $a_2 = 15$cm and that the base of the manipulator is at the origin of the coordinate system. Write a Python program to take the list of workspace points and plug them into the inverse kinematics formulas for the two link manipulator. Plot these points on a graph where θ_1 is the horizontal axis and θ_2 is the vertical axis. You will have to adjust some aspects to get a good looking plot. (Scale factors etc.) Test your code on the workspace line (a) $x + y = 25$, $x, y > 0$ and (b) $x = 10\cos(t) + 15$, $y = 10\sin(t)$ for $0 \leq t \leq \pi$. The point here is to see what the configuration space curve looks like.

11. Assume that you have a two link manipulator with $a_1 = 15$cm and $a_2 = 15$cm and that the base of the manipulator is at the origin of the coordinate system. Write a two link manipulator location program (Python). This program will take a list of angles and compute the location of the end effector. Show how this program works with the list of angles you generated in the previous problem. If the angle inputs are generated by a square, the simulated robot arm's end effector should trace a square. Plot the end effector points. You need to plot the input shape and the final shape to see if your code is correct. You will need to use the previous problem for this problem. Demonstrate your code to trace out the four segments which form the square with endpoints (5,0), (5, 15), (20, 15), (20,0).

12. Typos can creep up in textbooks, papers and reference materials. How would test the accuracy of the formulas given in equations (2.5) and (2.6)? Discuss.

13. Find the forward velocity kinematics equations for the parallel two link manipulator.

14. Derive the formula for (2.5):

$$(x, y) = \left(\frac{a+c}{2} + \frac{v(b-d)}{u}, \frac{b+d}{2} + \frac{v(c-a)}{u}\right)$$

Hint: define the segment from (a, b) to (c, d) as B (the base of the triangle), and $\vec{A}$ as a vector which is a perpendicular to B, see Fig. 2.35 .

15. Derive the formulas for the parallel two link manipulator inverse kinematics given in (2.6). Hint: Fig. 2.36.

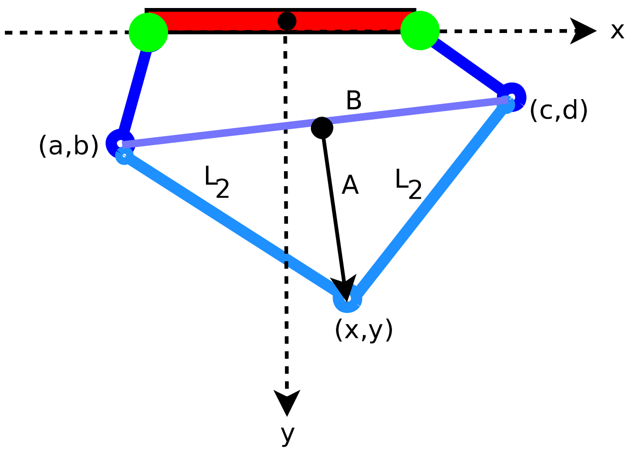


Fig. 2.35: Extraction of the isosceles triangle.

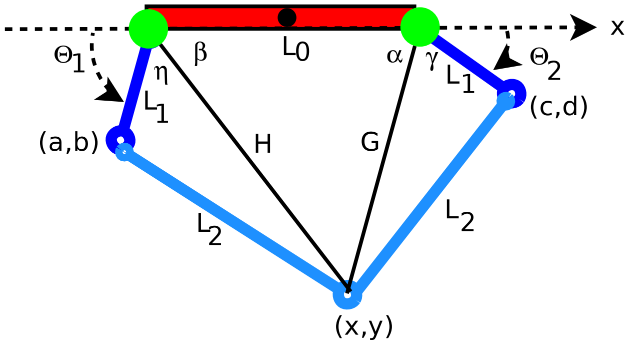


Fig. 2.36: Parallel Two Link Inverse Kinematics variables

16. Assume that you have a parallel two link manipulator with $L_0 = 10$cm, $L_1 = 15$cm and $L_2 = 20$cm. Write a Python program to take the list of workspace points given and plug them into the inverse kinematics formulas for the two link manipulator. Plot these points on a graph where θ_1 is the horizontal axis and θ_2 is the vertical axis. As above, you will have to adjust some aspects to get a good looking plot. The point here is to see what the configuration space curve looks like. The workspace points are the list of points for the rectangle with corners (-5, 18), (5, 18), (5, 27), (-5,27). Use 10 points in each side of the rectangle.

17. Assume that you have a parallel two link manipulator with $L_0 = 10$cm, $L_1 = 15$cm and $L_2 = 20$cm. Write a Python program that will take a list of angles and compute the location of the end effector. Show how this program works with the list of angles you generated in the previous problem. [If the angle inputs are generated by a rectangle, the simulated robot arm's end effector should trace a rectangle.] Plot the end effector points. You will need to use the previous problem for this problem.

18. Using Numpy and the linspace command, build an array of points for Fig. 2.37. The top is given by $(x-10)^2+(y-8)^2 = 25$ and the bottom is the line segment along $y = 8$. Traverse the figure starting at the right corner, going counter clockwise (circle first) and ending on the line segment. Check this with the Python plot command. Show the result.

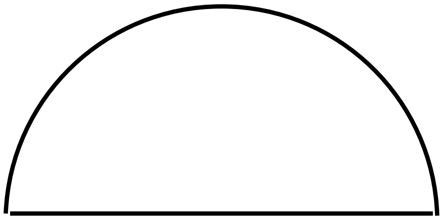


Fig. 2.37: Half disk.

19. Is the differential drive motion model given by (2.7) holonomic? Why or why not?

20. When inflating an obstacle, how much do you inflate it by?

21. Find the rotation matrix that will rotate clockwise by $30°$.

22. Vector review

    a. Given the vector $< 1, 2 >$, rotate this by 37 degrees (positive),

    b. If an axle is rotated off of the x-axis by 64 degrees, what is the vector that is in-line (parallel) the the axle?

    c. What is the projection of $< 3, 1 >$ onto the axle direction in the previous part?

23. Analytically show that the inverse rotation matrix is the same matrix as replacing θ by $-\theta$.

24. Assume that your differential drive robot has 10 cm diameter wheels and a total axle length (wheel to wheel) of 20 cms. If both wheels are turning at 0.8 revolutions per second, what is the speed of the robot.

25. Using the same robot as previous problem, but where the left wheel is turning at 1.5 radians per second and the right wheel is turning at 1.8 radians per second. Determine the linear velocity and path of the robot. You may assume the initial pose is (0,0,0) at $t = 0$.

26. For the differential drive robot, let $r = 10$, $L = 15$, $\dot{\phi}_1 = 0.9$ $\dot{\phi}_2 = 1.2$.

a. What is the angular velocity of the robot?

b. What is the velocity vector for the robot when $\theta = 45°$?

27. Let $r = 10$, $L = 15$. If you program the robot to drive straight and the robot traces out a circle of diameter 3 meters while traveling 1 m/s, what are the two wheel speeds?

28. Say you have a differential drive robot that has an axle length of 30cm and wheel diameter of 10cm. Find the angular velocity for the left and right wheel if the robot is going to

    a. Spin in place at a rate of 6 rpm (revolutions per min),

    b. Drive a circle of radius 1 meter (measured center of circle to middle of axle) at 3 rpm,

    c. Drive a straight line at 1 meter / min.

29. Given a differential drive robot starting from (0,0,0) find the final position when wheel velocities are given by:
    t=0 to t=5: ω_1 = 2, ω_2 = 2
    t=5 to t=6: ω_1 = 3, ω_2 = 4
    t=6 to t=10: ω_1 = 1, ω_2 = 2
    where D=10, L=16.

30. List the variables in the configuration space of a circular ground robot that can drive around and use a telescopic arm with a rotational base, lifting servo and elbow joint servo.

31. Show that the differential drive kinematic equations are non-holonomic constraints.

CHAPTER

THREE

SOFTWARE AND SIMULATION

In this chapter, we will discuss the collection of software tools used in developing and deploying robots. Most of these tools are not robotics specific such as Python or ZeroMQ. Others like ROS have been developed just for robotics. We begin with an overview of the specific robotics tools. Next we will cover so basic concepts in operating systems architecture and software design. The core tools we will employ, ZeroMQ and SciPy are introduced.

With the tools in hand, we can then move on to discuss using them to model robots, robot motion and the environment. Modeling is important for the design process as well as software testing. In addition, the components of the modeling process are used in the filtering of sensor data and the reconstruction of the robot's environment.

3.1 Robotics Frameworks and Languages

A *Robotics Framework* is currently a "catch-all" term. To most roboticists it means a collection of tools to support robotics software development. Typically a framework will provide some form of interprocess communication and a collection of hardware drivers. Interprocess communication is either shared memory and semaphore wrappers or TCP/IP socket support. [If these terms don't sound familiar, we will discuss them later in the text.] There are many simulation systems available. These range from fairly simplistic 2D single robot with a few obstacles to very sophisticated 3D full physics engine support systems. It is similar to what is seen in computer gaming. We will discuss a few of the more popular approaches below.

3.1.1 Player

The beginnings of ROS date back to the Player project, which was founded in 2000 by Brian Gerkey. This model included a hardware-abstracted robotic system known as the player, which interfaced with its simulated environment, known as the stage.

Player is a robotics framework. It provides communications and robot control interfacing. This is open source freely available software. Player is one of the leaders in the distributed approach to robotic control software. It provides a network interface to a variety of hardware devices and systems. Using a client-server approach, it gives the ability to control any device from any location. This allows multiple languages and multiple platforms to be used as a single robot control system; as long as they support sockets (TCP). It is especially useful in research when the low level software is in C, the sensor package is in Java and the behavior system is written in Python, allowing the best tool for the job to be used. Player is still maintained, but development ceased in 2010 (mostly due to ROS).

Stage is the simulation system that is loosely coupled with Player. They are separate but have been extensively used together. Stage is consider a 2.5D (more than 2D but less than 3D) simulation environment. Stage is oriented towards a world which is described by a two dimensional map of objects with some height. Fine details in the z direction are not modeled so the tool is not designed for simulation of grasping or manipulation. Stage is a very popular tool for modeling ground robots (and multiple ground robots). It has options to be compiled with control code or communicate with Player via the network interface. In this case, your control code would talk to Player which interfaces with Stage. The concept is that you develop your control software interacting with Stage. Then when ready to deploy, you disconnect Player-Stage and connect Player to the real hardware.

Player-Stage is a great idea. Getting it to compile and run is rather difficult. Since development has slowed and many developers have moved on, finding the right combination of Player version, Stage version, OS version and library collection can be frustrating. When it compiles and runs, it is a great tool. It can be found at http://playerstage.sourceforge.net.

3.1.2 Switchyard

The common API used by player was a major part of the next step on the road to ROS, the Stanford project known as *Switchyard*. Switchyard was developed by Morgan Quigley in 2007 under the Stanford Artificial Intelligence Robot (STAIR) project. Development of the system was shifted to a Stanford robotics start-up known as Willow Garage in 2008. The platform matured for about 2 years, and in 2010, Willow Garage released the first version of ROS.

3.1.3 ROS

ROS, the Robot Operating System, is an open source robotics framework. This project grew out of Player and many of the lessons learned with Player are found in ROS. ROS provides the communication system, a filesystem, distribution system and several thousand packages for device support, robot control, machine vision, sensor fusion, mapping, localization, etc. ROS was supported by Willow Garage (robotics company out of Stanford), but now maintained by the Open Source Robotics Foundation.

ROS is able to connect to Stage however the current focus is on Gazebo. Gazebo is an open source 3D simulation envirnoment for robotics (which began life with Player). It includes a full physics engine such as ODE, Bullet, Simbody, and DART.

ROS and Gazebo are extensions in some sense to Player-Stage. The idea of developing code in simulation then redirecting to real hardware is essentially the same outside the differences in interface syntax.

3.1.4 OSRF

In 2012, development of ROS began to shift from Willow Garage to the newly formed, Open Source Robotics Foundation, OSRF also oversees development of the Gazebo robot simulator, as well as the annual ROSCon, where ROS developers meet and discuss various ROS-related topics. Development using ROS still continues at Willow Garage, but the framework as a whole is developed at OSRF.

3.1.5 MS Robotics Developer Studio

Microsoft Robotics Developers Studio, MSRDS, is a full featured robotics development environment. First released in 2006 and the current stable release in 2012, made this framework an early player in the robotics community. It provides support tools for developing applications, supporting communications, visual authoring and simulation. MSRDS is a commercial application. The tool includes an asynchronous runtime environment which supports threading and interprocess communication. VPL is the Visual Programming Language which is in the spirit of Visual Studio and Visual Basic. This tool provides a drag and drop GUI for application development as well as export to C#. DSSME is a configuration editor to support application configuration and distribution. VSE, Visual Simulation Environment provides 3D simulation with physics. Robotics control software may be developed, simulated and tested without hardware. No updates have been released since 2012. On Sept. 22, 2014 Microsoft suspended their robotics division and so no further development is expected. MSRDS can found at http://www.microsoft.com/robotics/.

3.1.6 Webots

Like MSRDS, Webots is a full featured robotics development and simulation environment as well. It is a commercial application and is more oriented to instruction/simulation than the others described here. This tool provides a large choice of simulated sensors and hardware. Robotic control code can be prototyped in simulation and then ported to hardware for tuning. The goal is to provide a realistic simulation to reduce development time using their Model, Program, Simulate, Transfer approach. Unlike MSRDS, Player and ROS; Webots is more of a real physics engine, with collision detection and dynamics simulation and less of a robot OS/communications framework. It can be found at http://www.cyberbotics.com.

3.1.7 Robotics Programming Languages

There is not a “best” robotics programming language just as there is not a best programming language in general. Arguments about a best language are left to novices attempting to justify the language they most recently learned. Programming languages are tools like pliers, screwdrivers and hammers. It depends on what you want to do, what resources you have available and your personal skill set. Some languages are more popular however, like C and C++. The C family is used heavily since it has a small footprint (C fits on microcontrollers) and is very efficient. C++ provides the object oriented approach to a code base and is widely adopted in industry. Recent languages like Java and C# are popular when the robot has a full computer available as a controller. One can even find older languages like BASIC and FORTH as well. In this text we will focus on two: C/C++ and Python.[1]

Why C/C++ and Python? C is the major language for embedded systems. It can compile down to very compact code to run on a variety of microprocessors. C++ is the object oriented extension to C and both remain in the most popular programming language lists even though they have been around for some time. Python is an object oriented scripting language and is very popular as well, especially with regards to programming education. Another simple reason is that these are the two languages supported by ROS, the Robot Operating System. The bulk of the examples in this text are written in Python.

Also, for this text, we will assume that you are running a relatively current version of Ubuntu (possibly in a virtual machine). Python, C and C++ are part of the standard Ubuntu distribution. Normally it is not critical which version of Python is used (Python 2 versus 3).

3.2 Software Architectures

Consider a typical robot or the architecture of the mobile robot described in Fig. 1.14. There is only one computer involved in the sample system. How could distributed computing be part of the equation? It is plausible that parallel computing could be involved, but at first blush we might suppose that this would be a multi-core issue which is managed by the operating system. So again, why would distributed systems be of interest?

At a very low level, we have basic sensor hardware, motor and servo hardware and communications hardware. These live in a real time world and have dedicated hardware. These are embedded devices which can then communicate with the main system through standard interfaces such as USB or i2c. Each one runs at different frequencies with different control systems. It is interesting to watch the development of current robot infrastructures. In some ways it wants to follow the path the traditional operating systems did over the last 50 years. In other ways this development benefits from the last half century.

Think for a moment about traditional program design and execution. The process lives in an address space which is defined as all of the memory addresses for which the program resides and accesses, Fig. 3.1.

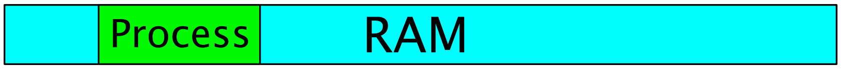


Fig. 3.1: Address space for a process.

The underlying operating system will go to great lengths to contain the process in it's allocated section of memory. Separate processes are run in separate contexts where the hardware isolates the code. In addition to the address space, processes no longer share registers, stack, files, etc. The OS does this to protect code blocks from each other. Errors in one system do not corrupt other systems (such as a faulty end condition on a loop writing to memory). Unintended interactions are greatly reduced. This is the design philosophy of any modern operating system. The OS manages the resources and provides the programs the illusion that they live alone on the computer. Each program runs as if it owns the entire machine. It appears to the

[1] Please don't send me email telling me that there are more than two languages since C and C++ are "actually" different languages. A discussion can be found: https://stackoverflow.com/questions/14330370/is-c-c-one-language-or-two-languages

program that it gets the entire memory system, disk system and networking. The OS does all the heavy lifting.

The other thing the OS is doing is giving the program a common interface to the hardware below. Meaning that the program (well, the programmer) need not worry about if the storage device is from vendor A or B. You don't need to know anything about the device specifics. There are general interfaces for the memory system, file system, devices, etc. This provides portability with software and really reduces the programming effort. So the OS provides the illusion of an abstracted computer or a virtual computer Fig. 3.2.

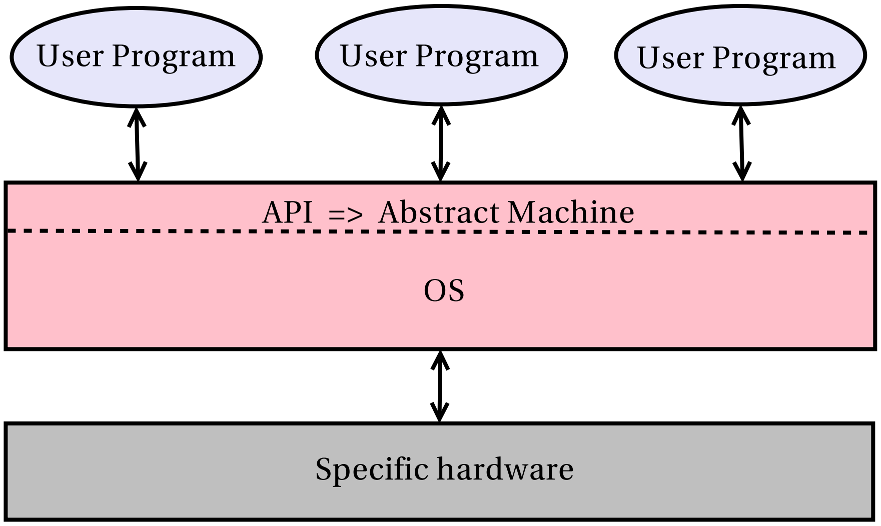


Fig. 3.2: The fundamental machine abstraction.

The common interface is implemented by a series of system calls. These are very special functions which allow access to the hardware. They are not traditional function calls since the thread of execution is moved over to the operating system. The kernel manages access and permissions, performs the requested function or returns error codes, and the process execution resumes. A current OS attempts to provide an abstract machine for a process. Low level OS routines (drivers and modules) are expected to translate the specifics of talking to a particular piece of hardware to the general abstracted interface. This means the programmer just interacts with a generic storage device and is not concerned about the specific details of that device. Actually, the interface makes one not even worry about the type of technology, for example magnetic vs solid state. This same approach is needed in the robotics world. The USB interface has helped with modular hardware in a limited sense. This makes the development and maintenance of software much easier. It also makes the system much more secure and robust. Being able to program using a fixed set of system calls makes the developer's job easier which in turn reduces errors. It means that the tricky part of accessing the hardware is done by individuals experienced in that domain. The collection of system calls really defines the OS. Not so much the collection of software shipped or the choice of desktop GUI.

It begs the question, if the operating system is really designed to separate processes, then how do they communicate. Processes must have communication. So various types of interprocess communication have been devised to support the model of breaking computation into multiple execution contexts, but still providing a way for the processes to communicate and coordinate.

For the moment, assume you are going to write your robot control code. Your code is a large sequence of sensing, planning and moving. The planning code probably runs on the CPU and at megahertz speeds. The sensing at kilohertz speeds and the movement at hertz speeds. As mentioned above you have lots of different activities at different speeds. We should take a page from the CS history books. We need modular code. We need code that is interrupt driven. We need to separate the different components.

Just like with desktop processing, it is neither possible or desirable to place all of the code into a single address space running on a single event loop. Even if we could place all of the sensor/actuator driver routines into the same program, good design demands modular code. It is essential to break the software into components. Separate them. This is done for ease of design, maintenance, security, robustness, and fault tolerance. At times you don't even have a choice about modularity. The current state of robotics development is that no single vendor builds all of the parts for the robot. You must assemble the hardware from different systems. The drivers for the components are provided. Robotics systems are too large to write from scratch. They live on top of existing traditional computing devices. What does this mean?

The first thing we want to address is the separation of data. This is often approached by data encapsulation approaches found in object oriented programming. Robotics has grown out of an embedded world focused on controls. These were real time systems with hard constraints on response times. By design the real time operating system and the underlying hardware was not running full operating systems on high performance computing hardware. So object oriented programming may not have been viable due to lack of system support. However, now one can get very powerful machines and full featured operating systems on postage stamp sized systems. OOP provides ways to limit access of data and deal with the complexities of large code installations.

We also want to separate the different functional blocks into different execution blocks. Again OOP support can support the programmer in moving to concurrent execution of methods. At a lower level, concurrency is supported by the notion of threads. A thread is an execution context. This means that the thread has a program counter, registers and a stack, but may share the address space which contains the data. Multithreaded programming gives the developer concurrency, but possibly at great cost. Some of the most subtle and difficult errors can arise when multiple threads are working on a common data block. Constructs such as semaphores have been created to manage access to common data regions. However, semaphores can cause deadlocking or process starvation.

Experience in both OOP and shared memory programming is important to avoid disastrous results. Another issue is the pace of robotics software. Systems have become increasing complicated over time. Expertise in all areas is hard to find. The ability to use external routines for certain aspects of the system - especially in development is critical. Having a large collection of functionally distinct modules makes the software akin to the building blocks found in hardware. Just as hardware systems are separate but use common interfaces (such as common pinouts in Arduino, or interfaces such as USB), software systems need to do the same thing to realize their potential.

Programs then must communicate with other programs using standard communication channels. One approach is to build each program as a function in a library or a class. Pushing code into a library can be a software engineering trap. Development is challenging enough when you have a huge interconnected codebase and then add hardware uncertainty. There are a thousand variations to a robot due to the number of sensors, actuators, and software libraries. One does not want to rebuild the system each time an update is released. A class will help with encapsulation. Still, this metaphor is one of single address space programming (yes, threads can help). Shared memory has been a favorite due to it's speed. Even so, it is fraught with danger. A course in operating systems shows you how shared memory programming can lead to problems far worse than low performance with the ability to completely deadlock a system. Another issue is that there are probably multiple processing units involved which don't share memory and so threaded models do not apply.

Multithreaded computation or shared memory programming is not the only way to proceed. Another form of interprocess communication is known as message passing. Data and computation requests are actively managed. Data is packaged and sent off to remote processes; processes which do not share the address space. These processes can be on different machines with different operating systems. This is increasingly important since the sensors and controllers are requiring their own cpus. Message passing is a way to address the interprocess communication need and also support multiple CPUs which do not share memory.

To support message passing interprocess communication, we need a way to send a packet of data to a remote host. The Unix world developed sockets as a method to send packaged data. Sockets and their supporting infrastructure are the backbone of the internet. Network sockets are the foundation of the internet which is probably the largest distributed system on the planet. Using message passing interprocess communication built over network sockets, we can build our collaborating process groups. Sockets allow us to define a standard interface for communication and then indirectly for computation. Building our software components on a message passing architecture built on TCP/IP simplifies the software engineering process. It embraces the robot as a distributed system from the start. Asynchronous concurrent computing can proceed in this environment. Scaling the number of devices is easier. Moving to swarms of robots and having them act as a single system is a natural outgrowth.

Robotics software followed some of the development seen in the general computing world. Microcontrollers without an operating system running programs resident in a single memory space. Adding functions, hardware and external devices pushed for having more complicated operating system support. Real time operating systems and desktop operating systems found their way into robot hardware. As more demands on motion planning occurred, increasingly powerful machines entered. This was made possible by the increasing power and shrinking size of the cpu.

Operating Systems development saw large monolithic kernels like unix, Fig. 3.3. They were powerful, provided sufficient performance and were complicated. Protection of resources and program portability became common. A complicated system call interface was produced to support the separation of user program from hardware. However, difficulties in development and debugging lead to layered OS designs such as early NT and OS/2, Fig. 3.4.

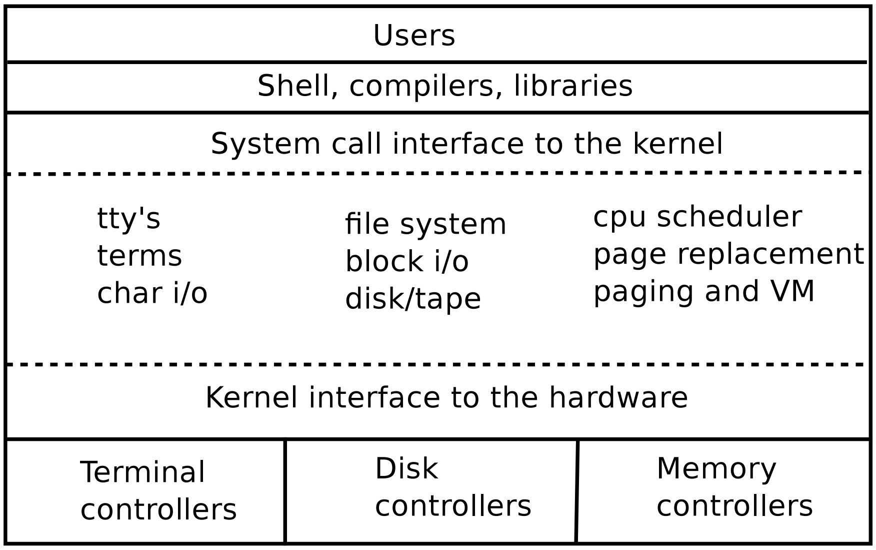


Fig. 3.3: Monolithic

Separation of code blocks is not complete in either of the previous designs and so experiments to build a minimal kernel, one which used message passing to support interprocess communication, was created.

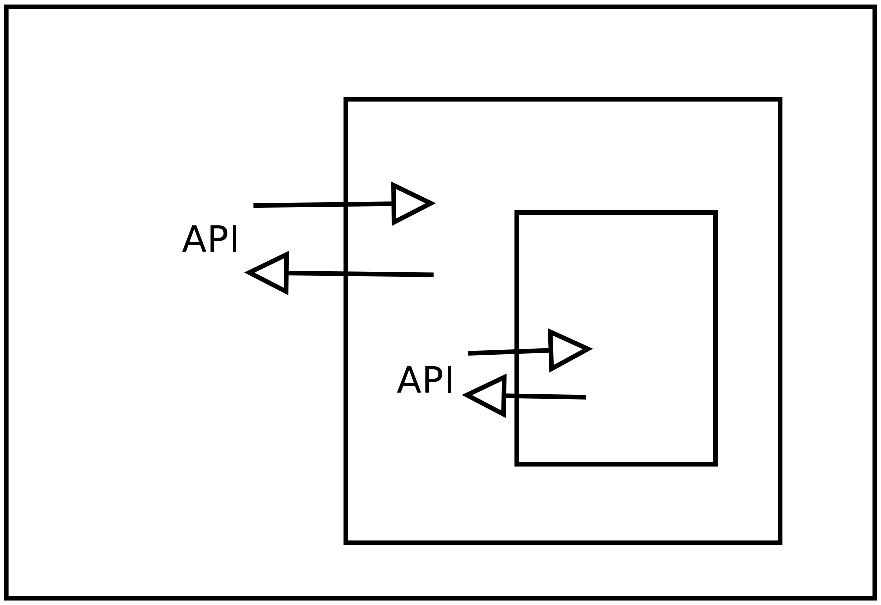


Fig. 3.4: Layered

These were known as microkernels since the design promoted moving all but the bare minimum out of the kernel leaving a very small kernel code base.

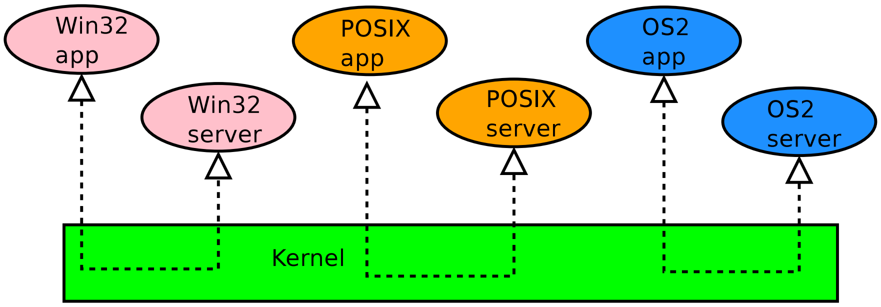


Fig. 3.5: Microkernel architecture.

The concept of a micro-kernel is very appealing. So much so that the Mach and NT kernels adopted the approach. The downfall was performance. As we embark on robotics development we cannot forget past experience. Performance drove many systems back to a monolithic design. Certainly the real time systems that run the hardware need real time code. Linux and Solaris decided against a microkernel approach and went with loadable modules. For an operating system, performance or speed is critical.

So, should we follow the OS path? Is the situation the same? There are two important differences in robotics. First is the domain of operation and the second is the measure of performance. The domain for a robot is the physical world. Mechanical systems operate in the millisecond range. The gigahertz range is well beyond what can be expected from mechatronic systems. Any code that interacts with the mechatronic system does not take the performance hit like what is seen with CPU process groups and so the benefits of this design stand out. The other aspect is the measure of performance. Once the processor can respond in time for a request, speeding it up may have no impact on the operation. Our measure now turns to the effectiveness of the robot in the task, development ease, security issues, cost, etc. So, again, we can see the benefit of message passing architectures.

However, processing sensor data or the planning operations could require considerable resources and partitioning the code into separate processes must be done with care. A careful study of data flow and data dependencies is required. This allows one to exploit available concurrency. Then the design decisions can be made regarding how to handle selection of the hardware and the resulting interprocess communication.

Computer vision can lead to massive amounts of concurrent simple arithmetic operations. A CPU may not be the best choice. Not that it cannot be done since most of the time it is. However, we know that specialized hardware can vastly outperform CPUs when confronted with structured operations. Use of FPGAs and GPUs are two great examples of different architectures that have been applied. This type of asymmetric computing can greatly enhance the performance of a robot which is simultaneously running vision, navigation and mapping. A system that is able to distribute different types of computation over asymmetric processors is now entering the distributed computing realm.

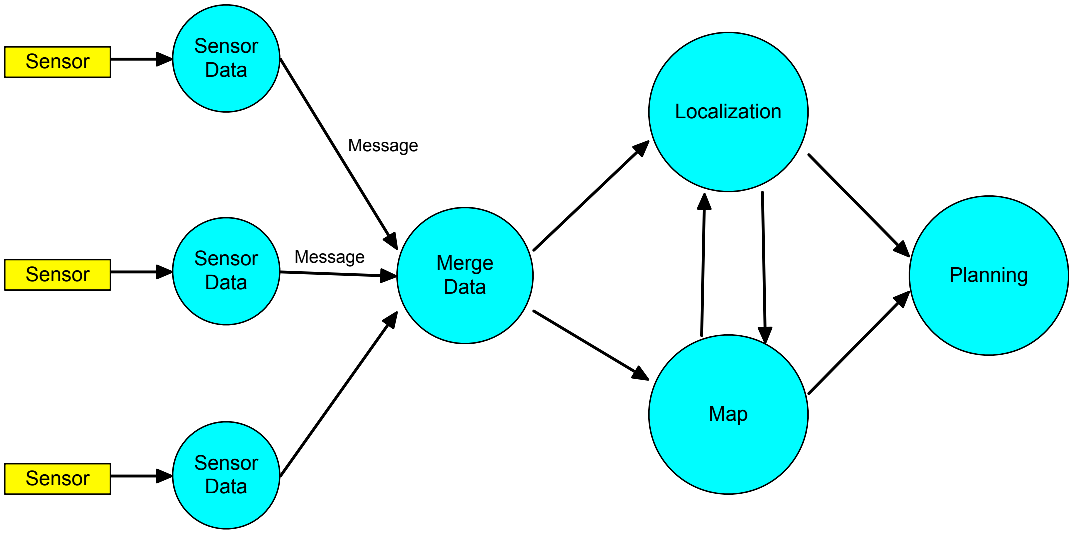
Consider a couple of applications of robotics. One is teleoperation and another is telepresence (arguably related, but are good examples). One of the driving forces in robotics is to remove people from dangerous and harmful situations. To this end, we require that the user is some distance away. Both applications require local and remote processing, and both require very robust communication.

A generalized communication system is needed. Something that provides uniform interfaces and is not dependent on specific hardware; a system that allows for modules to reside in separate address spaces and even separate processing units connected over a LAN. This system must be able to operate in an asynchronous fashion and be tolerant of faults (such as restarting a module).

3.3 Distributed Computation and Communications

Robotics is evolving from having completely integrated monolithic control systems to modular distributed architectures. As the hardware becomes more powerful and the goals more sophisticated, the complexity of the control system increases. It is increasing in a superlinear manner. We may view the workings of robotics software as a collection of interconnected computations and thus view the collection in a graph. Nodes would represent computational blocks, specifically processes. Interprocess communication is represented by the edges connecting the nodes in the graph. The connection between two nodes is the point to point communication we discussed above. A single robot could have many nodes. Some that control low level aspects like drive motors or wheel encoder data. Others higher level like processing data for computer vision algorithms, estimating position or routing the robot over the landscape.

Many of the nodes will be producing data for other nodes. Some nodes are producers, some consumers and some are both. The underlying client server architecture appears to be required. For a particular node that produces data for several other nodes, it needs to be a server to those client nodes. With multiple servers running and each delivering a different service, how should we manage this? The system connects to a host:port combination. So, one would need to know the host:port pair apriori. The host name might be known, but what about ports? A system could have external software that uses any particular port range. Having the vast collection of sensors and user contributed computational nodes means that a port numbering and classification system needs to be devised. Unix systems used to have remote procedures bound to port numbers. This works when there is a limited list. When the list gets long we need something like the Dewey Decimal system in the library. Of course we know that as the scale of node types increases, the predetermined mapping will eventually break.



For small and medium sized systems, one can manage the communciations using a central server. This works well in a variety of systems. For larger systems, where hundreds or thousands of nodes appear in the computation/communication graph, this produces a significant bottleneck and will not scale at all. And so a centralized system will not work. The system needs to be dynamic and configurable. However, we need a way to allow the data producer to connect with the data consumer.

A peer to peer connection is desired to avoid bottlenecks and other network issues related to a single central server. We also need a way to dynamically map hosts and ports as the system needs. This means that a database is required. The information can be centralized or distributed. If scale allows, a centralized system will have better response bounds since we know exactly how long it will take to find the required data. A distributed database may require several requests to get the information.

When a service starts up, it should register itself in a publicly available database. It would register with a central server and record that a particular service may be found at host:port pair. When the client is ready, it can query the central repository, request the service location and then connect to the correct server. This main server or master is a nameserver. Having only the job of handing out names at the start of the service, it does not affect the communications later on. We will say a particular node with data (the service) will publish this data. This means that it registers with the name server and accepts connections. A client requiring the data will subscript to the data by requested the publishing node from the nameserver and then requesting a connection to that node.

Having a specific service with multiple clients can complicate matters. The point of the service is to produce something, not worry about communications. So, to address this, a publish-subscribe mechanism can be built that treats the data as a topic. That topic is available on a type of message bus. The publisher and subscriber should be separated and not know about each other. This way one can deal with issues of scale, broken connections, reconnections and other real world issues without disturbing either the publisher or subscriber. Of course this will eliminate request-reply types of communication which should be addressed using a direct point to point type of channel.

The communication systems discussed above are normally implemented using a one of the standard Interprocess Communication interfaces. For this text, will focus on Sockets. Sockets provide a bidirectional channel between two processes. Although one side is setup like a server and one side like a client, this is basically point to point type of communication. With only two processes one could call this peer to peer or client server, however, in this case it is strictly one process to one process. The socket mechanism underneath is used to implement a vast array of process to process communication methods. We will not program sockets directly or natively, but will do this through a communication library known as ZeroMQ. ZeroMQ is introduced in the next section.

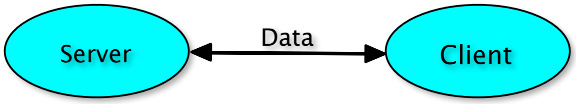
3.4 ZeroMQ

ZeroMQ is an open-source universal messaging library. It will provide a clean and portable communication API that sits on top of the basic socket implementation. For us, it means that we can focus our attention on the communications without getting mired down in the details of the way the socket is implemented in the language and operating system. It will help us develop portable code as well as more robust code. ZeroMQ supports the various communication styles we have discussed (and more).

At this point you may ask, "why not use ROS"? ROS indeed will do what we need and is vastly popular in the robotics community. This book is about concepts. With a few exceptions, we are going to develop the programs we need. We do not need the whole ROS ecosystem. We need interprocess communications; we need some type of messaging system. ROS is large. ROS is under active development and can, at times, be challenging to install as well as use. ZeroMQ is much smaller, with bindings for many languages and, for us, is a library available to the Python interpreter.

We are only going to touch on ZeroMQ. To really learn about it, especially for applications outside what we need, please refer to the Guide: http://zguide.zeromq.org/ . The guide is mostly examples in C, but enough Python is provided that often you can cut/paste to get good starting code. Plus, the Python docs address ZeroMQ: https://pyzmq.readthedocs.io/en/latest/ and https://learning-0mq-with-pyzmq.readthedocs.io/en/latest/ .

The easiest to understand is the REP pattern. It is a direct peer to peer client server communication pattern. This is known as REQUEST - REPLY.



A basic example is taken from the ZeroMQ guide. Here is a client server example. The Server code:

Listing 3.1: Simple 0MQ Server Example

```
#
#   Hello World server in Python
#   Binds REP socket to tcp://*:5555
#   Expects b"Hello" from client, replies with b"World"
#

import time
import zmq

context = zmq.Context()
```

(continues on next page)

(continued from previous page)

```
socket = context.socket(zmq.REP)
socket.bind("tcp://*:5555")

while True:
    #  Wait for next request from client
    message = socket.recv()
    print("Received request: %s" % message)

    #  Do some 'work'
    time.sleep(1)

    #  Send reply back to client
    socket.send(b"World")
```

And the client:

Listing 3.2: Simple 0MQ Client Example

```
#
#   Hello World client in Python
#   Connects REQ socket to tcp://localhost:5555
#   Sends "Hello" to server, expects "World" back
#

import zmq

context = zmq.Context()

#  Socket to talk to server
print("Connecting to hello world server...")
socket = context.socket(zmq.REQ)
socket.connect("tcp://localhost:5555")

#  Do 10 requests, waiting each time for a response
for request in range(10):
    print("Sending request %s ..." % request)
    socket.send(b"Hello")

    #  Get the reply.
    message = socket.recv()
    print("Received reply %s [ %s ]" % (request, message))
```

Copy these two programs to two files, server.py and client.py. You can run on the command line using:

```
bash> python server.py
```

and:

```
bash> python client.py
```

Note that control-c will kill the server process. We will go line by line through the code to understand how this works. To bring in the ZeroMQ library:

```
import zmq
```

Each process needs a container for the sockets. This container is called a context:

```
context = zmq.Context()
```

We can create a socket in the context:

```
socket = context.socket(zmq.REP)
```

A socket is a communication conduit. We need to select the communication protocol (tcp), label the address and select the port (5555):

```
socket.bind("tcp://*:5555")
```

To receive a message:

```
message = socket.recv()
```

To send a message:

```
socket.send(b"Hello")
```

A word about the messages. In Python 3, strings are stored using Unicode. ZeroMQ uses byte strings, not Unicode. So, we must convert back and forth from bytes to Unicode. For string literals, place a *b* in front: b”string”. To convert a string variable:

```
bmessage = bytes(message,'ascii')
```

We take the previous example and code up a client server example where the client sends 10 (x,y) pairs to a server which computes the Two Link Manipulator Inverse Kinematics for each.

Listing 3.3: IK Server

```
import zmq
from math import *

context = zmq.Context()
socket = context.socket(zmq.REP)
socket.bind("tcp://*:5555")

a1,a2 = 15.0,10.0

while True:
    #  Wait for next request from client
    message = socket.recv()
    list = message.split(b" ")
    x = eval(list[0])
    y = eval(list[1])
    d =  (x*x+y*y-a1*a1-a2*a2)/(2*a1*a2)
    t2 = atan2(-sqrt(1.0-d*d),d)
```

(continues on next page)

(continued from previous page)

```
    t1 = atan2(y,x) - atan2(a2*sin(t2),a1+a2*cos(t2))
    #  Send reply back to client
    reply = bytes(str(t1) + " " + str(t2),'ascii')
    socket.send(reply)
```

Listing 3.4: Simple 0MQ Server Example

```
import zmq
import time

context = zmq.Context()

#  Socket to talk to server
print("Connecting to IK server...")
socket = context.socket(zmq.REQ)
socket.connect("tcp://localhost:5555")

#  Do 10 requests, waiting each time for a response
for i in range(10):
    x = 5 + 0.2*i
    y = 3 + 0.3*i
    print("Compute IK for (%s,%s)..." % (x,y))
    message = bytes(str(x) + " " + str(y),'ascii')
    socket.send(message)
    #  Get the reply.
    reply = socket.recv()
    print("Received reply %s [ %s ]" % (i, reply))
    time.sleep(0.2)
```

And the output is:

```
(base) alta:0MQ jmcgough$ python client_IK.py
Connecting to IK server...
Compute IK for (5.0,3.0)...
Received reply 0 [ b'0.9704752657646376 -2.896027136074501' ]
Compute IK for (5.2,3.3)...
Received reply 1 [ b'1.0565361872984023 -2.846929421795498' ]
Compute IK for (5.4,3.6)...
Received reply 2 [ b'1.1267594030030246 -2.802128816232321' ]
Compute IK for (5.6,3.9)...
Received reply 3 [ b'1.185432519250973 -2.7600732877375735' ]
Compute IK for (5.8,4.2)...
Received reply 4 [ b'1.2351575978007627 -2.7199062657491724' ]
Compute IK for (6.0,4.5)...
Received reply 5 [ b'1.277684949617325 -2.681099228530734' ]
Compute IK for (6.2,4.8)...
Received reply 6 [ b'1.3142715256788646 -2.643300360150102' ]
Compute IK for (6.4,5.1)...
Received reply 7 [ b'1.3458611495026371 -2.6062619970976555' ]
Compute IK for (6.6,5.4)...
Received reply 8 [ b'1.373185595974717 -2.569802002720451' ]
```

(continues on next page)

(continued from previous page)

```
Compute IK for (6.8,5.699999999999999)...
Received reply 9 [ b'1.3968260058595374 -2.533781548607152' ]
(base) alta:0MQ jmcgough$
```

The next model we introduce is the Publisher-Subscriber model which is nicely supported in 0MQ.

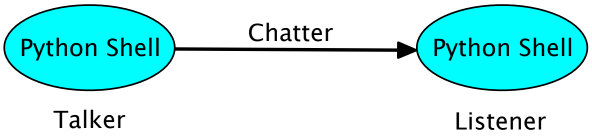


Fig. 3.6: Simple PubSub example

Listing 3.5: Simple Publisher example.

```
import time
import zmq

context = zmq.Context()
publisher = context.socket(zmq.PUB)
publisher.bind("tcp://*:5555")

i = 0

topic = "Chatter"
message = "Hello World"
btopic = bytes(topic,'ascii')
bmessage = bytes(message,'ascii')


while True:
    #  Do some 'work'
     time.sleep(1)
     i = i+1
     print(i)
    #  Send reply back to client
     publisher.send_multipart([btopic, bmessage])
```

Listing 3.6: Simple Subscriber Example

```
import zmq

context = zmq.Context()

#  Socket to talk to server
print("Connecting to hello world server...")
subscriber = context.socket(zmq.SUB)
subscriber.connect("tcp://localhost:5555")

```

(continues on next page)

(continued from previous page)

```
topic = "Chatter"
subscriber.setsockopt_string(zmq.SUBSCRIBE, topic)

for r in range(100):
    #  Get the reply.
    [address,message] = subscriber.recv_multipart()
    print("Received %s [ %s ]" % (r, message))
```

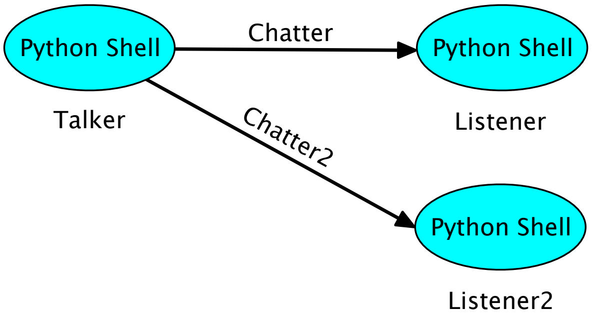


Fig. 3.7: Simple PubSub example cont.

Listing 3.7: Simple Publisher example - multiple topics.

```
import time
import zmq

context = zmq.Context()
publisher = context.socket(zmq.PUB)
publisher.bind("tcp://*:5555")

i = 0

topic1 = "Chatter"
message1 = "Hello World"
btopic1 = bytes(topic1,'ascii')
bmessage1 = bytes(message1,'ascii')

topic2 = "Chatter2"
message2 = "Hello Other World"
btopic2 = bytes(topic2,'ascii')
bmessage2 = bytes(message2,'ascii')


while True:
    #  Do some 'work'
     time.sleep(1)
     i = i+1
     print(i)
    #  Send reply back to client
     publisher.send_multipart([btopic1, bmessage1])
```

(continues on next page)

(continued from previous page)

```
    publisher.send_multipart([btopic2, bmessage2])

publisher.close()
context.term()
```

Listing 3.8: Simple Subscriber Example - multiple topics, pt1.

```
import zmq

context = zmq.Context()

#  Socket to talk to server
print("Connecting to hello world server...")
subscriber = context.socket(zmq.SUB)
subscriber.connect("tcp://localhost:5555")


topic = "Chatter"

subscriber.setsockopt_string(zmq.SUBSCRIBE, topic)

for r in range(100):
    #  Get the reply.
    [address,message] = subscriber.recv_multipart()
    print("Received %s [ %s ]" % (r, message))

subscriber.close()
context.term()
```

Listing 3.9: Simple Subscriber Example - multiple topics, pt2.

```
import zmq

context = zmq.Context()

#  Socket to talk to server
print("Connecting to hello world server...")
subscriber = context.socket(zmq.SUB)
subscriber.connect("tcp://localhost:5555")


topic = "Chatter2"

subscriber.setsockopt_string(zmq.SUBSCRIBE, topic)

for r in range(100):
    #  Get the reply.
    [address,message] = subscriber.recv_multipart()
    print("Received %s [ %s ]" % (r, message))

subscriber.close()
```

(continues on next page)

(continued from previous page)

```
context.term()
```

There are a full range of possibilities here. A program can setup multiple sockets:

```
context = zmq.Context()
publisher = context.socket(zmq.PUB)
publisher.bind("tcp://*:5555")
publisher2 = context.socket(zmq.PUB)
publisher2.bind("tcp://*:5556")
```

which are published via:

```
publisher.send_multipart([btopic1, bmessage1])
publisher2.send_multipart([btopic2, bmessage2])
```

Likewise one can connect to multiple sockets:

```
subscriber = context.socket(zmq.SUB)
subscriber.connect("tcp://localhost:5555")
subscriber2 = context.socket(zmq.SUB)
subscriber2.connect("tcp://localhost:5556")
```

Which are subscribed via:

```
[address,message] = subscriber.recv_multipart()
[address,message] = subscriber2.recv_multipart()
```

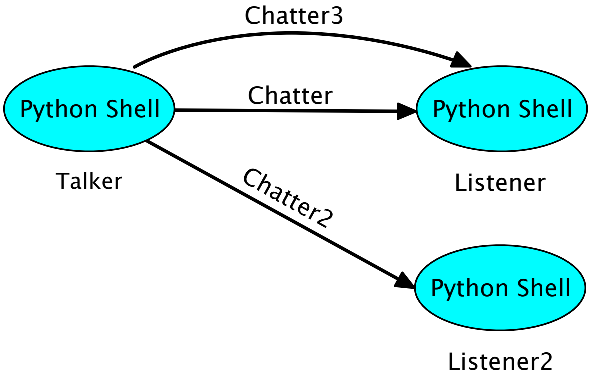


Fig. 3.8: One can easily create multiple data paths.

If you would like the files to act like a program and not require passing them to Python, then place at the top of the file:

```
#!/usr/bin/env python
```

or:

```
#!/usr/bin/env python3
```

And then make the file executable by:

```
chmod +x <filename>
```

3.5 SciPy and Mathematics

The following material assumes that you are familiar with Python. Python reads like pseudocode and so it is possible to follow along without a background in Python if you have seen some other programming language. A quick introduction is given in the Appendices for those who want to ramp up before reading on.

SciPy, is a collection of open-source packages for Scientific Computing. One of the packages, redundantly named, SciPy library is a collection of numerical methods including special functions, integration, optimization, linear algebra, interpolation, and other standard mathematics routines. NumPy is an open-source Python package supporting data structures and low level algorithms for scientific computing which is used by SciPy.[1] The main data structure of interest to us from numpy is an array type and efficient methods to access array elements.

Many of the implementations of iPython load NumPy and SciPy (as well as the plotting package matlibplot) automatically. The idea is that most users of iPython are going to use these. To use the NumPy or SciPy libraries you need to import them. Since the scientific libraries are large, we don't want to drop them into the main namespace. The Python community often uses the following names for the namespaces:

```
>>> import numpy as np
>>> import scipy as sp
>>> import matplotlib as mpl
>>> import matplotlib.pyplot as plt
```

Scipy sub-packages need to be imported separately, for example:

```
>>> from scipy import linalg, optimize
```

As stated above, some versions of iPython (some IDEs) will import certain libraries for you. Say you are tired of typing the five import lines above each time you run iPython. There is a full configuration system available. To find the location of the config files, type at the command prompt

```
$ ipython locate
/home/yourloginname/.ipython

$ ipython locate profile foo
/home/yourloginname/.ipython/profile_foo
```

You may create a profile for each of the different iPython activities. We will stick with the default which is profile_default. The startup files, files that get run when you start iPython, are located in the startup subdirectory. In my case this is:

[1] Thanks to the NumPy and SciPy online tutorials for great examples.

/Users/jmcgough/.ipython/profile_default/startup

Inside the startup directory, I created a file: 05-early.py containing

```
import numpy as np
import scipy as sp
import matplotlib as mpl
import matplotlib.pyplot as plt
```

which then runs those import commands each time iPython is invoked. In this next section, we will review some needed mathematics and introduce SciPy as we proceed.

3.6 Vectors and Matrices

Creation of an array is easy:

```
In [1]: import numpy as np

In [2]: x = np.array([2,3,6,4,5,0])

In [3]: x
Out[3]: array([2, 3, 6, 4, 5, 0])

In [4]: len(x)
Out[4]: 6
```

The array command takes a list and converts it to an array object. Arrays are stored like C does it, in row major order. To create an array of numbers from 0 to 10:

```
In [2]: x = np.arange(10)

In [3]: x
Out[3]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [4]: 2*x+1
Out[4]: array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19])

In [5]: x*x
Out[5]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])

In [6]: np.sqrt(x)
Out[6]:
array([ 0.        ,  1.        ,  1.41421356,  1.73205081,  2.        ,
        2.23606798,  2.44948974,  2.64575131,  2.82842712,  3.        ])

In [7]: np.sin(0.2*x)
Out[7]:
array([ 0.        ,  0.19866933,  0.38941834,  0.56464247,  0.71735609,
        0.84147098,  0.93203909,  0.98544973,  0.9995736 ,  0.97384763])
```

(continues on next page)

(continued from previous page)

```
In [8]: np.sin(0.2*x)[3]
Out[8]: 0.56464247339503548

In [9]: x < 7
Out[9]: array([ True,  True,  True,  True,  True,
                True,  True, False, False, False], dtype=bool)

In [10]: x.sum()
Out[10]: 45

In [11]: x.max()
Out[11]: 9

In [12]: x.min()
Out[12]: 0

In [13]: x.mean()
Out[13]: 4.5

In [14]: x.std()
Out[14]: 2.8722813232690143

In [15]: np.where(x < 7)
Out[15]: (array([0, 1, 2, 3, 4, 5, 6]),)
```

Note that indexing works like normal Python lists. A few vector operations are also available as methods.

```
In [2]: x = np.arange(10)

In [3]: y = np.ones(10)

In [4]: x
Out[4]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [5]: y
Out[5]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])

In [6]: np.dot(x,y)
Out[6]: 45.0

In [7]: np.outer(x,y)
Out[7]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.,  3.,  3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.,  4.],
       [ 5.,  5.,  5.,  5.,  5.,  5.,  5.,  5.,  5.,  5.],
       [ 6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.,  6.],
       [ 7.,  7.,  7.,  7.,  7.,  7.,  7.,  7.,  7.,  7.],
       [ 8.,  8.,  8.,  8.,  8.,  8.,  8.,  8.,  8.,  8.],
```

(continues on next page)

(continued from previous page)

```
       [ 9.,  9.,  9.,  9.,  9.,  9.,  9.,  9.,  9.,  9.]])
```

Some NumPy examples using 2D arrays (or matrices):

```
In [2]: A = np.array([[1,2,3],[4,5,6]])

In [3]: print A
[[1 2 3]
 [4 5 6]]

In [4]: B = np.array([[9,8],[7,6],[5,4]])

In [5]: print B
[[9 8]
 [7 6]
 [5 4]]

In [6]: A*B
---------------------------
ValueError                             Traceback (most recent call last)
<ipython-input-6-e2f71f566704> in <module>()
----> 1 A*B

ValueError: operands could not be broadcast together with shapes
(2,3) (3,2)

In [7]: np.dot(A,B)
Out[7]:
array([[ 38,  32],
       [101,  86]])

In [8]: A.T
Out[8]:
array([[1, 4],
       [2, 5],
       [3, 6]])

In [9]: A.T + B
Out[9]:
array([[10, 12],
       [ 9, 11],
       [ 8, 10]])
```

Note: Most of the python overloaded math operators are defined elementwise. As such $*$ does not make sense for $A * B$ since the arrays are not the same dimension. The point is that you need to be careful and in this case you need to call the correct function to do matrix multiplication and not array multiplication.

One can easily create a two dimensional array by reshaping:

```
In [10]: z = np.arange(16)
```

(continues on next page)

(continued from previous page)

```
In [11]: z
Out[11]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12,
                                13, 14, 15])

In [12]: z.shape = (4,4)

In [13]: z
Out[13]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])

In [14]: z[1,3]
Out[14]: 7

In [15]: z[1,-4]
Out[15]: 4
```

Using previous examples of A and B:

```
In [16]: import numpy.linalg as npl

In [17]: npl.det(np.dot(A,B))
Out[17]: 35.99999999999968
```

3.7 Linear Algebra

We use both NumPy and SciPy for Linear Algebra problems. NumPy is used to provide the array data structure and the numerical methods are provided in SciPy.

```
In [1]: import numpy as np

In [2]: import scipy as sp

In [3]: from scipy import linalg as spl

In [4]: A = np.array([[3,1,0],[1,5,1],[0,2,6]])

In [5]: A
Out[5]:
array([[3, 1, 0],
       [1, 5, 1],
       [0, 2, 6]])

In [6]: b = np.array([[3,2,1]]).T

In [7]: b
Out[7]:
```

(continues on next page)

(continued from previous page)

```
array([[3],
       [2],
       [1]])

In [8]: x1 = spl.inv(A).dot(b)   # x = inverse(A)*b

In [9]: x1
Out[9]:
array([[ 0.93589744],
       [ 0.19230769],
       [ 0.1025641 ]])

In [10]: x2 = spl.solve(A,b)   # solve Ax = b

In [11]: x2
Out[11]:
array([[ 0.93589744],
       [ 0.19230769],
       [ 0.1025641 ]])

In [12]: A.dot(x1)
Out[12]:
array([[ 3.],
       [ 2.],
       [ 1.]])
```

One question that arises is regarding performance. There is a significant difference between plain Python and NumPy. This author's experiments have shown that NumPy performs very well and has fallen within 10-20% of plain C in some cases. Given how powerful the Python-NumPy combination is, this is a small price.

```
In [2]: import scipy.linalg as lin

In [3]: a = np.array([[3, 1, 0], [1, 5, 1],  [0, 2, 6]])

In [4]: lin.eig(a)
Out[4]:
(array([ 2.48586307+0.j,  4.42800673+0.j,  7.08613020+0.j]),
 array([[ 0.86067643,  0.39715065,  0.11600488],
       [-0.44250554,  0.5671338 ,  0.47401104],
       [ 0.25184308, -0.72154737,  0.87284386]]))
```

Eigenvalues pop up all through engineering computations and we will use the built in SciPy routines to compute them. The most common application later will be finding the error ellipses for variance-covariance matrices in the Kalman Filter.

3.7.1 Least Squares Examples

Assume that you have the raw data ready in arrays x and y. Then Fig. 21.6 and Fig. 21.7 can be produced by:

```
one = np.ones((N))
A = np.array([ x, one]).T
AT = A.T
AA = np.dot(AT,A)
ATy = np.dot(AT,y)
t = np.arange(0,10, 0.2)
B = np.array([t,np.ones(len(t))]).T

c = linalg.solve(AA,ATy)
line1 = np.dot(B,c)

weights =[]
sum = 0
for i in range(1,N+1):
    v = 1.0/(i*i*i)
    sum = sum + v
    weights.append(v)

for i in range(N):
    weights[i] = weights[i]/sum

ww = np.diagflat(weights)
A1 = np.dot(ww,A)
AA = np.dot(AT,A1)
y1 = np.dot(ww,y)
ATy = np.dot(AT,y1)
coeff2 = linalg.solve(AA,ATy)
line2 = np.dot(B,coeff2)

# Plot result: red is data, blue is uniformly weighted,
#  green is weighted to points near the origin.
plt.plot(t,line1, 'b-', t,line2, 'g-', x,y, 'r.')
plt.show()
```

3.8 MatPlotLib

MatPlotLib is the SciPy library used for generating plots. We will be using the *pyplot* functions from it. The standard import convention is import matplotlib.pyplot as plt. The basic tool is the function plot. Let x and y be lists of numbers representing the points (x_i, y_i). Simple plots can be made using plot(x,y).

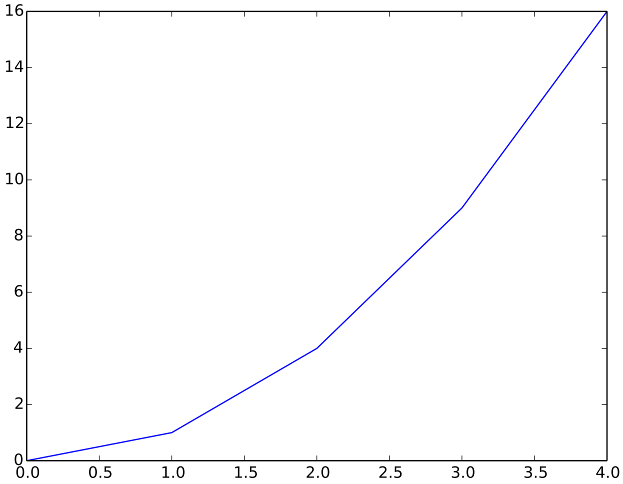
```
In [1]: x = [0,1,2,3,4]

In [2]: y = [0,1,4,9,16]

In [3]: plt.plot(x,y)
Out[3]: [<matplotlib.lines.Line2D at 0x105079490>]

In [4]: plt.show()

In [5]: plt.plot([1,2,3,4], [1,4,9,16], 'ro')
```

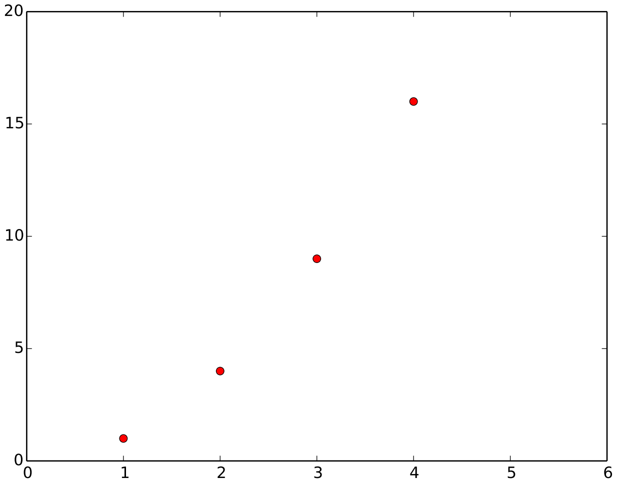
(continues on next page)

(continued from previous page)

```
Out[5]: [<matplotlib.lines.Line2D at 0x110a586d0>]

In [6]: plt.axis([0, 6, 0, 20])
Out[6]: [0, 6, 0, 20]

In [7]: plt.show()
```

This code produces the following two plots:





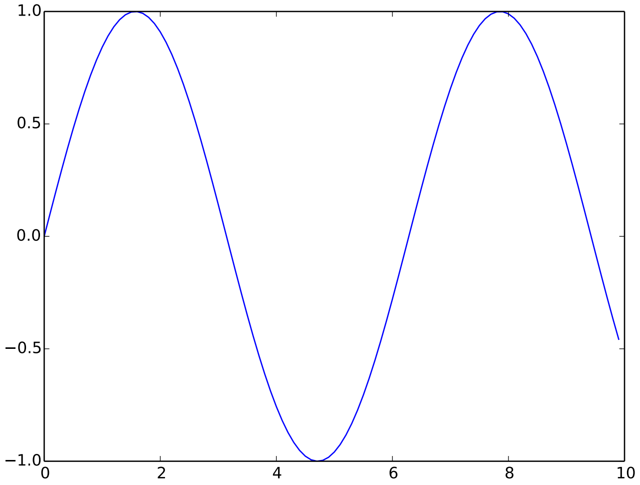
This is efficiently done using NumPy arrays instead of lists and using NumPy functions to generate the arrays.

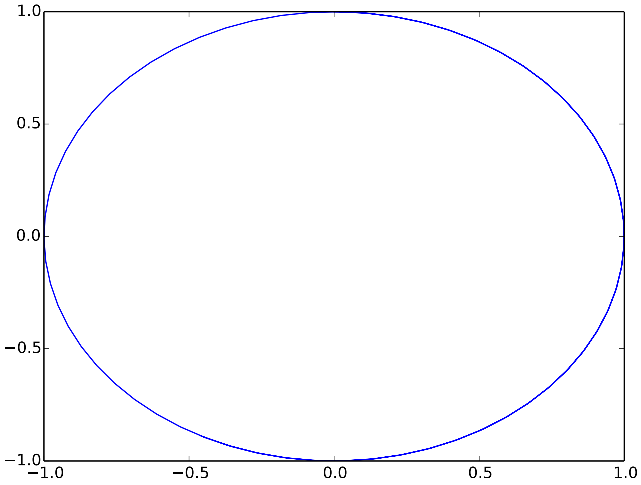
```
In [1]: x = np.arange(0,10,0.1)
```

(continues on next page)

(continued from previous page)

```
In [2]: y = np.sin(x)

In [3]: plt.plot(x,y,'b-')
Out[3]: [<matplotlib.lines.Line2D at 0x104880490>]

In [4]: plt.show()

In [5]: z = np.cos(x)

In [6]: plt.plot(y,z)
Out[6]: [<matplotlib.lines.Line2D at 0x105ea7810>]

In [7]: plt.show()
```





Surface plots may be done by importing the library mpl_toolkits.mplot3d. For surface plotting to work, a meshgrid needs to be created. This can be easily built from the x and y array data. The 3D plotting support is in a toolit shipped wiht matplotlib. It is accessed via the axis setting in the figure function:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

An example of a quadratic surface in Fig. 3.9. Many other plot examples can be found at the MatPlotLib website.

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 10, 0.2)
y = np.arange(0, 10, 0.2)
N,M = x.size, y.size

x,y = np.meshgrid(x,y)
z = (x-5)*(x-5) + (y-6)*(y-6)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, z, rstride=1, cstride=1, color='b')
plt.show()
```

Another example will illustrate both the plotting capability as well as linear regression Fig. 3.10.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg

xl = [  0.        ,   1.11111111,   2.22222222,   3.33333333,
       4.44444444,   5.55555556,   6.66666667,   7.77777778,
       8.88888889,  10.        ]
yl = [  1.86113482,   3.81083902,   4.1465256 ,   7.37843476,
       10.76437019,  11.99975421,  14.59486508,  16.0576472 ,
       20.77206089,  20.4204027 ]

N = len(xl)
x = np.array(xl)
y = np.array(yl)
A = np.array([x, np.ones((N))]).T
AT = np.array([x, np.ones((N))])
AA = np.dot(AT,A)
ATy = np.dot(AT,y)

c = linalg.solve(AA,ATy)
t = np.arange(0,10, 0.25)
B = np.array([t,np.ones(len(t))]).T
s = np.dot(B,c)

plt.plot(t,s, 'b-', x,y, 'ro')
```

(continues on next page)

(continued from previous page)

```
plt.xlim(0,10)
plt.ylim(0,20)
plt.show()
```

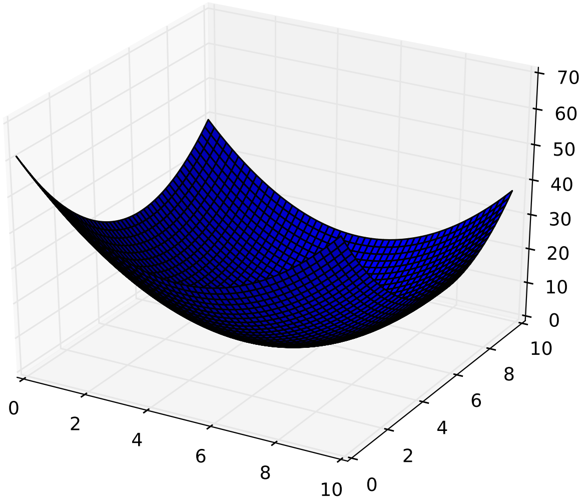


Fig. 3.9: Surface plot example.

3.8.1 Animation

Animation is done using the draw command. Create a plot with the plot command and then update the lists using the set_ydata command. The draw commend will draw the updated data into the existing plot window.

```
from pylab import *
import time

ion()

tstart = time.time()                 # for profiling
x = arange(0,2*pi,0.01)              # x-array
line, = plot(x,sin(x))
for i in arange(1,200):
    line.set_ydata(sin(x+i/10.0))    # update the data
    draw()                           # redraw the canvas

print 'FPS:' , 200/(time.time()-tstart)
```

Interactive mode needs to be toggled using ion() and an empty plot created. Next a loop runs through the positions of the points. The setp command updates the plot data values. Appended to the plot values (the plot comand) is the previous points to give the effect of a traced path. After the animation, interactive mode

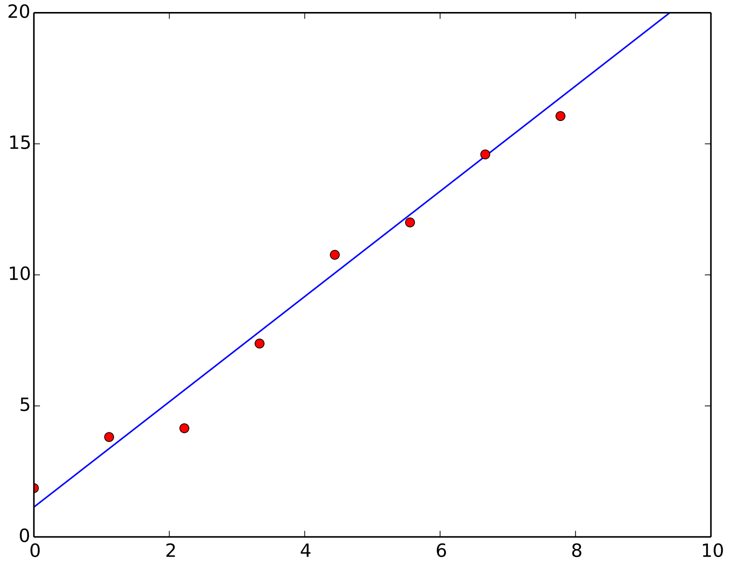
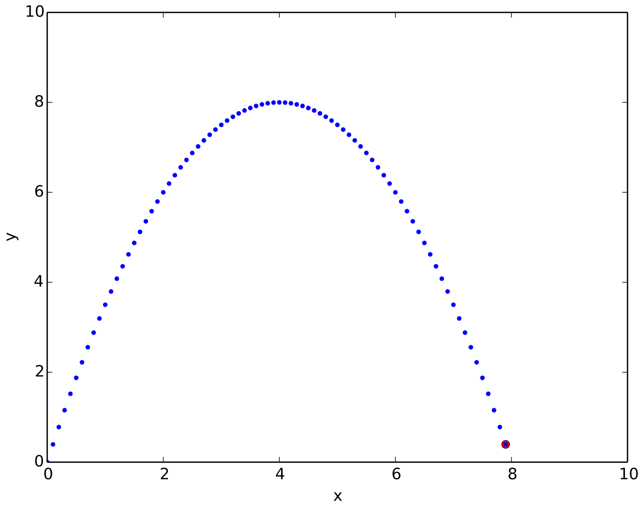


Fig. 3.10: Line fit and plot example.

is toggled, ioff() and the show() command is executed to hold the image.

```
import numpy as np
import matplotlib.pyplot as plt
import time
from math import *

plt.ion()

line = plt.plot([],[],'ro')
plt.xlim(0, 10)
plt.ylim(0, 10)
plt.xlabel('x')
plt.ylabel('y')
plt.draw()
dt = 0.1

for t in np.arange(0,8,dt):
    x = t
    y =  x*(8-x)/2.0
    plt.setp(line,xdata = x, ydata = y)
    plt.draw()
    plt.plot([x],[y],'b.')

plt.ioff()
plt.show()
```

Another animation example is to give virtual velocity commands to move a point. Say you wanted to animate



an object which was moving by

$$\left(\frac{dx}{dt}, \frac{dy}{dt}\right) = \begin{cases} (0.5, 0.0), & 0 \le t < 2, \\ (0.25, 1.0), & 2 \le t < 5, \\ (1.0, 0.0), & 5 \le t < 8, \\ (0.3, -1.0), & 8 \le t < 10, \end{cases}$$

and starting at $t = 0$, $(x, y) = (0.1, 3)$. Using the approximation of the derivative

$$\frac{dx}{dt} \approx \frac{x(t+\Delta t) - x(t)}{\Delta t} \qquad \Rightarrow \qquad \left[x_{\text{current}} + \left(\frac{dx}{dt}\right)\Delta t\right] \to x_{\text{new}}$$

```
import numpy as np
import matplotlib.pyplot as plt
import time
from math import *

plt.ion()

line, = plt.plot([],[],'bo')
plt.xlim(0, 10)
plt.ylim(0, 10)
plt.xlabel('x')
plt.ylabel('y')
plt.draw()
x = 0.1
y = 3
dt = 0.1


for t in np.arange(0,10,dt):
```

(continues on next page)

(continued from previous page)

```
    if t < 2:
        x = x + 0.5*dt
    if (t>=2) and (t<5):
        x = x + 0.25*dt
        y = y + dt
    if (t>=5) and (t<8):
        x = x + dt
    if (t>=8):
        x = x+0.3*dt
        y = y - dt
    line.set_xdata([x])
    line.set_ydata([y])
    plt.draw()
    time.sleep(0.1)

plt.ioff()
plt.show()
```

Thanks to the NumPy and SciPy online tutorials for great examples.

3.9 Graphing parametric functions

The Python plot command (well, this is actually the MatPlotLib library for Python) takes an array of x values and an array of y values. This means that it is very easy to generate explicit plots, $y = f(x)$ or parametric plots, $x = f(t)$, $y = g(t)$. So, for example one can easily plot a regular function via

```
import numpy as np
import pylab as plt

x = np.linspace(0,5,25) # 25 equally spaced points on [0,5]
y = 0.15*x*x*x   #  Generate the y values from y = 0.15x^3

plt.plot(x,y,'bo')   #  Plot x-y values using blue dots
plt.show()

plt.plot(x,y,'b-')   #  Plot x-y values using a blue line
plt.show()
```

The two plots should look like Fig. 3.11. You will notice that the line plot hides the fact that the underlying data is actually discrete. The point plot provides the actual points. The same thing can be done using a parametric version making the small change in the code:

```
t = np.linspace(0,5,25)
x = t
y = 0.15*t*t*t
```

You will also notice that the space between the points is not the same even though x (or t) was generated using uniform spacing. The x spacing is uniform, but the y value is s nonlinear function of x and the spacing between is not constant.

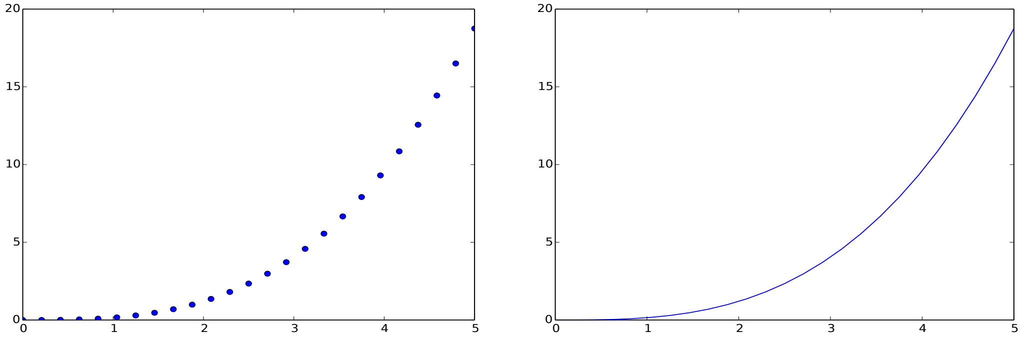


Fig. 3.11: The plot of $y = 0.1x^3$ using a) points b) a line.

3.9.1 Code Sample (heart):

```
import numpy as np
import pylab as plt
import math
t = np.linspace(-math.pi,math.pi,200)
x = 16*(np.sin(t))**3
y = 13*np.cos(t) - 5*np.cos(2*t) - 2*np.cos(3*t) - np.cos(4*t)
plt.plot(x,y,'r-')
plt.show()
```

3.9.2 Error Ellipses

In the section on Kalman filters, we will want to track the progress of the filter by tracking the error of the estimate. It is normally represented by an error ellipse where the ellipse size is the variances or standard deviations of the Kalman estimate. Thus the larger the standard deviations then the larger the ellipse. As you will see later Kalman process produces a covariance, P. The eigenvalues and eigenvectors of P can be used for the basic variance information. The eigenvectors represent the major and minor axis directions and the eigenvalues represent the lengths of those axes. Note: in some applications it makes sense to graph the standard deviations instead of the variances and so one should take the square root of the eigenvalues. The algorithm follows.

- Compute the eigenvalues and eigenvectors of P: (λ_1, v_1), (λ_2, v_2). Call the larger one a and the smaller one b.
- Compute the square roots of the eigenvalues IF desired (if the variances are really small or really huge).
- Compute the smaller angle between the eigenvector and the x-axis. Call this θ and assume it is for v_1.
- Call an ellipse routine to plot.

Let a, b be the major and minor axis lengths, x0, y0 be the center and angle be the tilt angle. The function to plot an rotated ellipse is given by:

```
def Ellipse(a,b,angle,x0,y0):
    points=100
    cos_a,sin_a=math.cos(angle*math.pi/180),math.sin(angle*math.pi/180)
    theta=np.linspace(0,2*np.pi,points)
    X=a*np.cos(theta)*cos_a-sin_a*b*np.sin(theta)+x0
    Y=a*np.cos(theta)*sin_a+cos_a*b*np.sin(theta)+y0
    return X,Y
```

The following is an example of how to plot an error ellipse for the covariance matrix

$$P = \begin{pmatrix} 0.9 & 0.1 \\ 0.1 & 0.5 \end{pmatrix}$$

about the point $(4, 5)$. We use the eigenvalues and eigenvectors to plot the major and minor axes. The following is a quick example on how to extract eigenvalues and plot an ellipse.

```
import math
import numpy as np
import pylab as plt
from numpy import linalg
P = np.array([[0.9, 0.1],[0.1, 0.5]])
w, v = linalg.eig(P)
angle = 180*math.atan2(v[1][0],v[0][0])/math.pi
u,v = Ellipse(w[0],w[1],angle, 4,5)
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(u,v,'b-')
ax.set_aspect('equal')
fig.savefig("Ellipse.pdf")
plt.show()
```

3.9.3 Data Plots

To plot the data used in the curve fitting examples:

```
import numpy as np
import pylab as plt
x = []
y = []
f = open('data.txt','r')
for line in f:
  item = line.split()
  xt = eval(item[0])
  yt = eval(item[1])
  x.append(xt)
  y.append(yt)

plt.plot(x,y, 'ro')
plt.show()
```

For Figure in subsection Fig. 15.9:

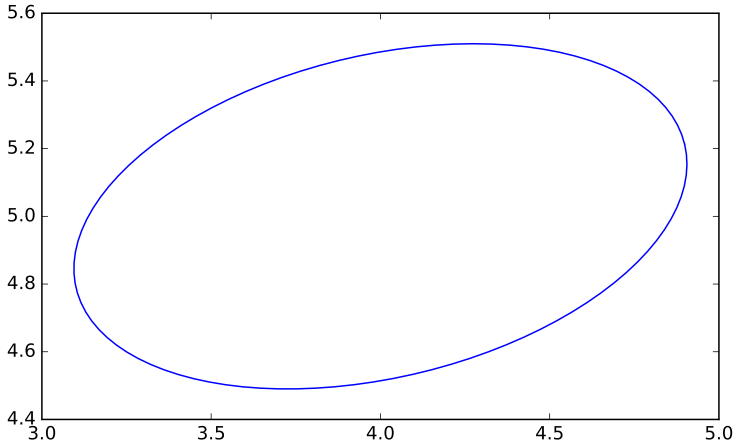


Fig. 3.12: Tilted ellipse

```
import numpy as np
import pylab as plt
from scipy import linalg

xl = []
yl = []
f = open('data.txt','r')
for line in f:
  item = line.split()
  xt = eval(item[0])
  yt = eval(item[1])
  xl.append(xt)
  yl.append(yt)

N = len(xl)
x = np.array(xl)
y = np.array(yl)
xx = x*x
A = np.array([xx, x, np.ones((N))]).T
AT = np.array([xx, x, np.ones((N))])
AA = np.dot(AT,A)
ATy = np.dot(AT,y)

c = linalg.solve(AA,ATy)
t = np.arange(0,3, 0.1)
tt = t*t
B = np.array([tt,t,np.ones(len(t))]).T
s = np.dot(B,c)
```

(continues on next page)

(continued from previous page)

```
plt.plot(t,s, 'b-', x,y, 'ro')
plt.xlim(0,3)
plt.ylim(0,2)
plt.show()
```

Note that NumPy/SciPy provides some built in functions to fit polynomials to lines. The NumPy function linalg.lstsq will compute the pseudoinverse via the normal equations directly and the NumPy function polyfit will do this assuming you are fitting a polynomial. In terms of speed, doing it ourselves tends to be fastest, with the next fastest is the lstsq function and the polyfit function the slowest.

3.10 Why simulate?

Learning how to operate any robotic system can be rather rough on your budget. A sufficiently robust robot that can support the standard array of sensors, processing and drive system can run well into the thousands of dollars. And there is nothing more annoying than having a silly little software error send the robot tumbling down a flight of stairs. Based on cost and possible system damage, many researchers and instructors elect to run the robot in simulation. Physical robots can take a long time to build, configure and get operational. They require all the details to be just right or it will not power up.

Compare this to simulation. Getting the initial simulation software running can be very time consuming, but once this is done, changes to simulation software can be fast. Realistic simulations are very hard however. It requires that all of the physical parts are carefully modeled in the code. So the robot design or geometry with masses and components needs to be included. Next the physics needs to be carefully entered. All of the servo or motor dynamics, friction, torque, acceleration, etc involved must be modeled. In many cases, building the robot is faster than building a very accurate simulation.

So why would one simulate? One answer is that one can focus on a part of the system and not worry about the entire design to be competed. The simulation does not need all the components of the robot included and you can focus on parts of the system. You can swap out parts quickly and simulate components that don’t actually exist. A simulation can act as a proof of concept leading to a new design. It can be cheaper than prototyping. Simulations don’t wear out or run out of battery power.

Of course, your design might, by accident, optimize aspects unique to the simulation and not work in the real world. The conclusion of many roboticists is that simulation is a valuable tool in the initial prototyping stage, but there is no substitute for a physical robot. Much of this applies to a first course in robotics. Every reader can run simulations at no cost (well assuming that we are using an open source package). The simulated robot cannot be broken. It won’t run out of power. The most valuable aspect is that the simulation can be restricted to just a few elements making it much simpler. This avoids overwhelming the novice and so will be our approach.

There are some really great motion simulation packages available. Often these codes fall under more general motion simulation codes and overlap gaming systems, kinematics and dynamics codes. These use physics engines like the Open Dynamics Engine which can simulate the kinematics, forces, collisions and other physical phenomenon.

For the roboticist interested in simulating a robot, it is very nice not to have to look at the simulation code and focus on the control code. This allows more time devoted to the robotics control problem. However, if we have a single code base, the code must be designed to be “compiled” together. This is a challenge in the face of shared resources. By separating the code into different programs which communicate via messages, we achieve data encapsulation and security, modularity, module language independence, and location independence. ROS will allow us to do just that.

But, looking back over the code, is it really worth go to the trouble of having separate programs control the robot? Most of us just sit at one computer and running multiple computers can be challenging. There are several reasons to consider. Large blocks of code are hard to design, develop and, mostly, debug. Good software practice would have us develop classes or modules that address specific functions in the software. So, separation of the graphics from the control code is good design. Of course, we can do this without using our socket based design.

One reason for this design is that we can select the best environment and programming language for the windowing and then the best environment and language for the control code. The graphics might be written in C++ or Java. The control code could be written in Python.

The windowing code and the driving code are fundamentally different. They run concurrently yet they operate on different schedules and interact asynchronously. As we incorporate more detail into the graphics side, it will require additional resources. It makes sense to assign these different tasks to different cpus. The graphics side will want to add maps, simulated sensors and other simulated hardware which needs to be separate from the control code. When multiple developers become involved, the interfaces between the device code must be established - otherwise code chaos will emerge.

As stated before, producing motion in a real robot is not difficult. Deciding on the correct actions and controlling the system are the more challenging aspects. The first is known as *Motion Planning* or just *Planning* and the second is called *Controls*. To get started we will borrow algorithms from nature since we see so many successful autonomous agents in the biological world. Worms and insects are very successful animals. They can sense the world and move around in it. We can borrow from notions in physics and chemistry when we see simple systems moving in constrained manners. The simplest solution is the best solution. It is best to use no more components or technology than necessary. Beyond basic elegance is the fact that the more components something has, the greater probability the system will fail. This is true in our simulations as well.

We return to the two examples in the previous section, the Two Link Manipulator and the Mobile Disk Robot. Using these two systems, we will introduce methods to simulate motion. These very basic systems can be used as the prototypes for developing a simulation and for the simple motion planning algorithms. First we need to give an overview of NumPy, SciPy and MatPlotLib.

3.11 Two Link Simulation Example in Python

For the arm in the two link example, determine the joint angles to trace out a circle centered at (10,8) of radius 5. The circle can be parametrized by $x(t) = 5\cos(t) + 10$, $y(t) = 5\sin(t) + 8$, $0 \leq t \leq 2\pi$. Generate an array of points on the circle and plug them into the inverse kinematics.

```
import numpy as np
import matplotlib.pyplot as plt
import time
from math import *

```

(continues on next page)

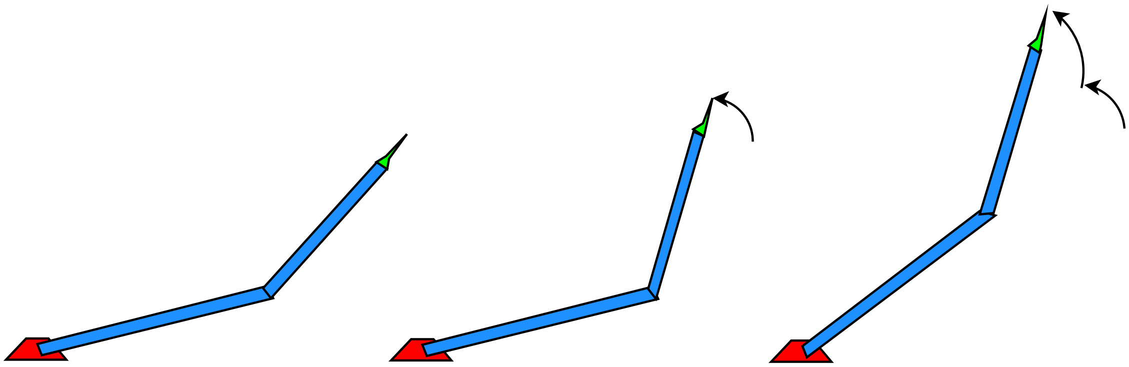


Fig. 3.13: Try a simple position control. Send a discrete set of control points.

(continued from previous page)

```
a1 = 15
a2 = 10
step = 0.1

#Setup Arrays
t = np.arange(0, 2*np.pi+step, step)
x = 5*np.cos(t) + 10
y = 5*np.sin(t) + 8

#Compute joint angles and check them
a1 = 15.0
a2 = 10.0
d = (x*x + y*y - a1*a1 - a2*a2)/(2*a1*a2)
t2 = np.arctan2(-np.sqrt(1.0-d*d),d)
t1 = np.arctan2(y,x) - np.arctan2(a2*np.sin(t2),a1
                +a2*np.cos(t2))
xsim1 = a1*np.cos(t1)
ysim1 = a1*np.sin(t1)
xsim = a2*np.cos(t1+t2) + xsim1
ysim = a2*np.sin(t1+t2) + ysim1

plt.figure(1)
plt.subplot(221)
plt.xlim(0, 20)
plt.ylim(0, 15)
plt.ylabel('Y')
plt.title('Requested Path')
plt.plot(x,y)

plt.subplot(222)
plt.xlabel('Theta1')
plt.ylabel('Theta2')
plt.title('Joint Angles for Path')
plt.plot(t1,t2)

plt.subplot(223)
```

(continues on next page)

(continued from previous page)

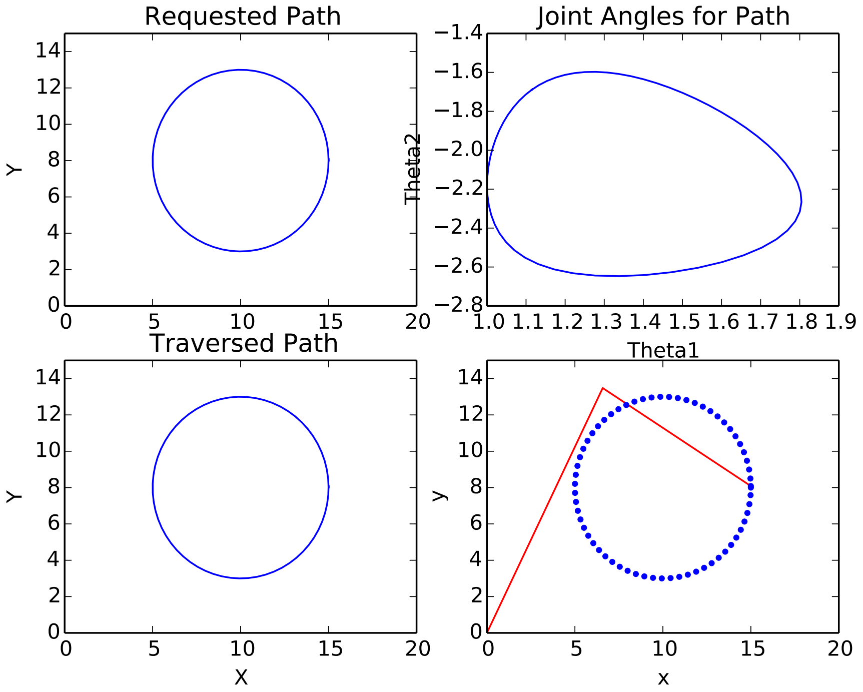
```
plt.xlim(0, 20)
plt.ylim(0, 15)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Traversed Path')
plt.plot(xsim,ysim)

plt.ion()  #  Turn on interactive mode
plt.subplot(224)
arm = plt.plot([],[],'b-')  # Create empty plot
plt.xlim(0, 20)
plt.ylim(0, 15)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

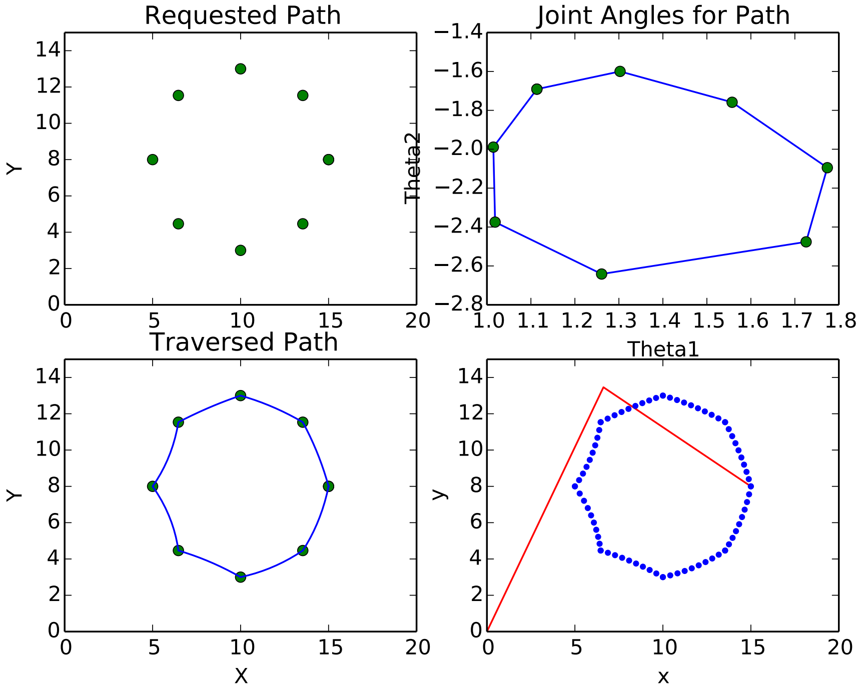
```
#Animation
for i in range(t.size):
    x1 = xsim1[i]
    y1 = ysim1[i]
    x2 = xsim[i]
    y2 = ysim[i]
    plt.setp(arm,xdata = [0,x1,x2], ydata = [0,y1,y2])
    plt.draw()
    plt.plot([x2],[y2],'b.')
    time.sleep(0.05)

plt.ioff()
plt.show()
```

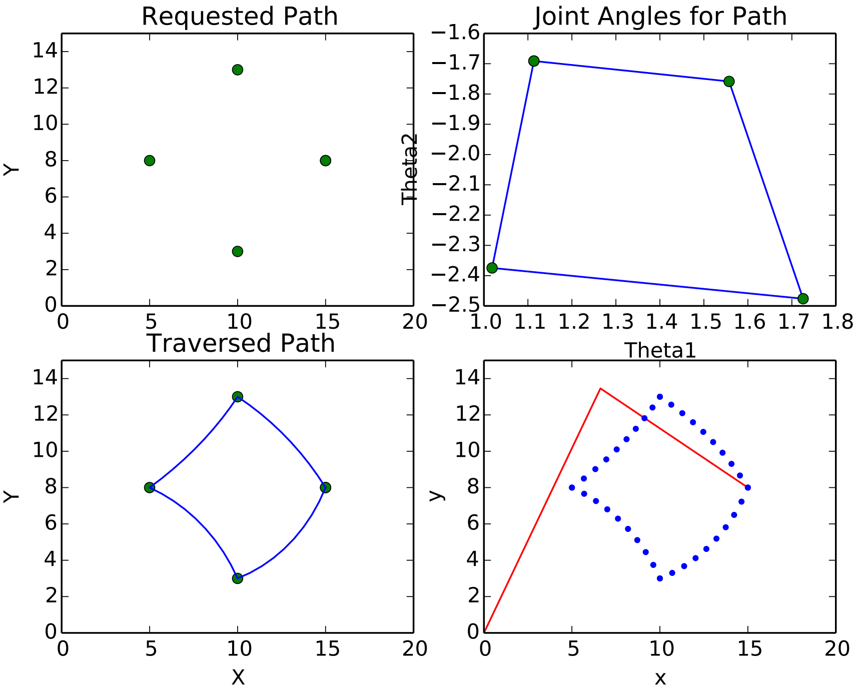
3.12 Moving the Differential Drive robot

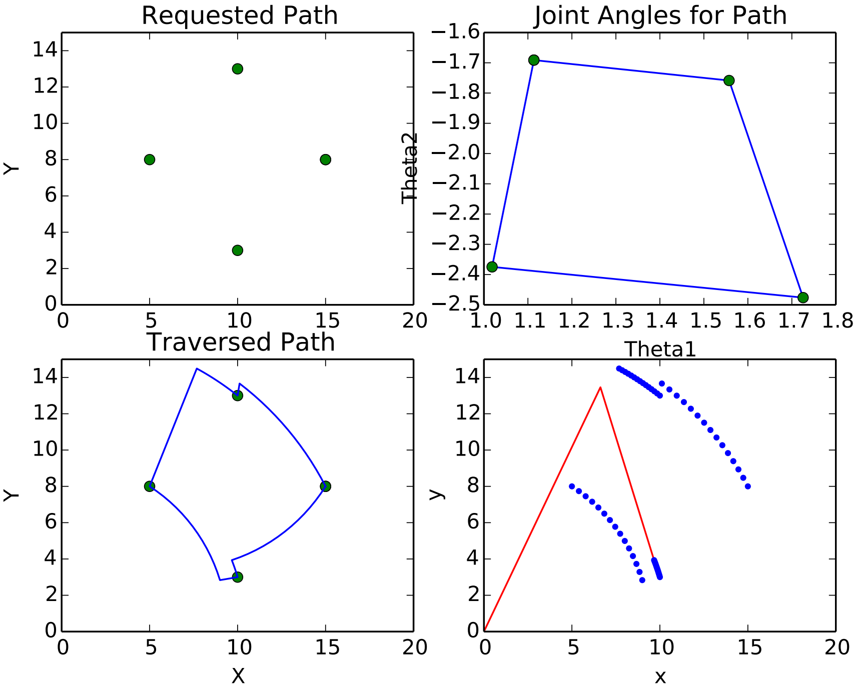
In the last section, we moved the two link articulator by updating the position. It is certainly possible to simulate a robot moving through space by simply jumping positions. Motion effect is produced like a movie projector gives the impression of motion. So, for animation, this approach can and often does suffice. However, objects in the world don't jump positions. Momentum, inertia and limits on acceleration and velocity do play a significant role. To move a ground robot, the position should be controlled by velocity. In reality the position is controlled by control signals to a motor which in turn generates a velocity, but we will assume we have a perfect motor controller for now; one that can take a velocity command and achieve that velocity.

In this section we simulate the motion of the differential drive robot that we introduced in the Terms Chapter shown in Fig. 3.14.


Requested Path
Joint Angles for Path
Traversed Path
Y
X
Theta1
Theta2
x
y
0
5
10
15
20
0
2
4
6
8
10
12
14
−1.4
−1.6
−1.8
−2.0
−2.2
−2.4
−2.6
−2.8
1.0
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9


Requested Path
Joint Angles for Path
Traversed Path
Y
X
Theta1
Theta2
x
y
0
5
10
15
20
0
2
4
6
8
10
12
14
−1.4
−1.6
−1.8
−2.0
−2.2
−2.4
−2.6
−2.8
1.0
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8


Requested Path
Joint Angles for Path
Theta2
Theta1
Traversed Path
Y
X
y
x
0
5
10
15
20
0
2
4
6
8
10
12
14
−1.6
−1.7
−1.8
−1.9
−2.0
−2.1
−2.2
−2.3
−2.4
−2.5
1.0
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8


Requested Path
Joint Angles for Path
Theta2
Theta1
Traversed Path
Y
X
y
x
0
5
10
15
20
0
2
4
6
8
10
12
14
−1.6
−1.7
−1.8
−1.9
−2.0
−2.1
−2.2
−2.3
−2.4
−2.5
1.0
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8

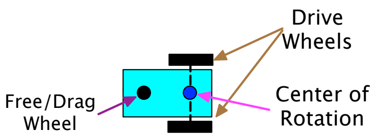


Fig. 3.14: Simple differential drive robot.

and the associated equations (2.7)

$$\boxed{\begin{aligned} \dot{x} &= \tfrac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\cos(\theta) \\ \dot{y} &= \tfrac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\sin(\theta) \\ \dot{\theta} &= \tfrac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2) \end{aligned}} \tag{3.1}$$

where $\dot{\phi}_1$ and $\dot{\phi}_2$ are the right and left wheel rotational speeds (respectively), r is wheel radius and $2L$ is the axle length.

What can be said about these equations? Can these be partially solved so we can run simulations? Our first attempt is to solve the differential equations by integration. Starting with the third equation, the one for the angular velocity,

$$\dot{\theta} = \frac{d\theta}{dt} = \frac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2)$$

integrate from 0 to t (and be careful about integration variables)

$$\int_0^t \frac{d\theta}{d\tau}\, d\tau = \int_0^t \frac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2)\, d\tau$$

and we have

$$\theta(t) = \theta(0) + \int_0^t \frac{r}{2L}\left(\frac{d\phi_1}{d\tau} - \frac{d\phi_2}{d\tau}\right) d\tau$$

Normally one can determine $\dot{\phi}_i$, but it might not have a standard functional form. These values are wheel velocities and do correspond with the standard collection of calculus functions. For the moment, assume that you know $\phi_i(t)$, then what can you say? From $\dot{\phi}_i(t)$ we can compute θ by integrating the last equation. This will be used in the formulas for x and y. Integrating the formulas for x and y

$$x(t) = x(0) + \int_0^t \frac{r}{2}\left(\frac{d\phi_1}{d\tau} + \frac{d\phi_2}{d\tau}\right)\cos(\theta(\tau))d\tau$$
$$y(t) = y(0) + \int_0^t \frac{r}{2}\left(\frac{d\phi_1}{d\tau} + \frac{d\phi_2}{d\tau}\right)\sin(\theta(\tau))d\tau$$

These equations are easy to integrate if you know the wheel velocities are constants. First integrate the θ equation:

$$\theta(t) = (r/2L)(\omega_1 - \omega_2)t + \theta(0).$$

Theta can be plugged into the x and y equations and integrated, under the assumption that $\omega_1 \neq \omega_2$ or $\omega_1 \neq -\omega_2$:

$$x(t) = \frac{L(\omega_1 + \omega_2)}{(\omega_1 - \omega_2)} \left[\sin((r/2L)(\omega_1 - \omega_2)t + \theta(0)) - \sin(\theta(0))\right]$$

$$y(t) = -\frac{L(\omega_1 + \omega_2)}{(\omega_1 - \omega_2)} \left[\cos((r/2L)(\omega_1 - \omega_2)t + \theta(0)) - \cos(\theta(0))\right]$$

This solution is a sequence of circular arcs. For the special case where $\omega_1 = \omega_2 = \omega$, we have that $d\theta/dt = 0$, so,

$$\begin{aligned} x &= r\omega\cos(\theta_0)t + x_0 \\ y &= r\omega\sin(\theta_0)t + y_0 \\ \theta &= \theta_0. \end{aligned}$$

And when $\omega_1 = -\omega_2 = \omega$, we have $dx/dt = 0$ and $dy/dt = 0$, so

$$\begin{aligned} x &= x_0 \\ y &= y_0 \\ \theta &= \frac{r\omega}{L}t + \omega_0. \end{aligned}$$

As long as you have piecewise constant angular velocities on the wheels, you have the robot path made up from circular arcs. A simulation program can connect these up to produce a path for any sequence of wheel velocities. The path is made up of combinations of lines and arcs. Note that a pivot in place is possible so the resulting path need not be differentiable. Fig. 3.15 shows a sample path.

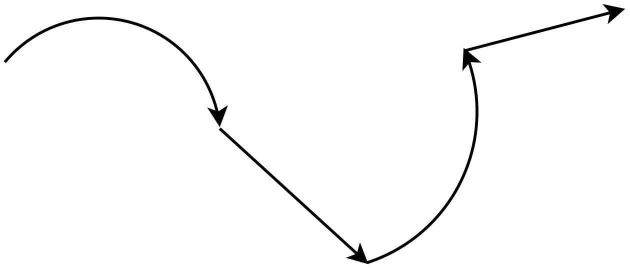


Fig. 3.15: Piecewise circular/linear arc paths

In practice it is not possible to instantaneously jump wheel speeds. Inertia in the system (mass, inductance, power limits) means that it is not possible to instantaneous jumps in velocity. In addition, it is not possible to have perfect velocities when surfaces and power are not consistent. So what if we relax the constant velocity assumption. This gives rise to two additional issues. The first is that you may not be able to gain an antiderivative of the wheel velocities to find $\theta(t)$. If you are able to find θ, the right hand sides for $\dot{x}$ and $\dot{y}$ normally are not integrable. A simple example below demonstrates issues with finding antiderivatives.

Let $\dot{\phi_1} = e^{-t^2}$ and $\dot{\phi_2} = t$

$$\theta(t) = \theta(0) + \int_0^t \frac{r}{2L}\left(e^{-\tau^2} - \tau\right) d\tau = ??? \tag{3.2}$$

This integral cannot be resolved. Meaning we cannot find an analytic antiderivative. It is possible to approximate it either with a Taylor expansion or numerical formulation, but it is an example of a vast number of functions which we must stop at this step.

There is another problem that this example indicates. In general, looking for an analytic function for the position is not possible. Practically you don't actually have a function representation of $\phi(t)$ and are normally measuring the wheel angular velocity during runtime? How should we formulate and proceed in that case.

3.12.1 A numerical approach

We will use Euler's (*Oil-ler's*) method for solving the differential equations. Euler's method approximates the derivatives with a forward finite difference and converts the differential equation into a difference equation. The difference equations are algebraic and can be evaluated numerically. This is also known as a finite difference method. Let the time between measurements be denoted by Δt. We discretize (or approximate) the time variable and the three state variables using discrete variables. This simple means we have a sequence of numbers $\{x_k\}$ instead of a function $x(t)$. Technically we should use a different variable, but I will often be efficient[1] and reuse the variable even though one denotes a function of time and one denotes a sequence.

$$t_k \equiv k\Delta t, \quad t_{k+1} = (k+1)\Delta t$$

$$x_k \equiv x(t_k), \quad y_k \equiv y(t_k)$$

$$\omega_{1,k} \equiv \dot{\phi}_1(t_k), \quad \omega_{2,k} \equiv \dot{\phi}_2(t_k)$$

Recall that if x is position then $\dot{x} = dx/dt$ is velocity (and $\ddot{x} = d^2x/dt^2$ is acceleration). From basic calculus, we recall that we may approximate a derivative using a forward finite difference:

$$\dot{x} \approx \frac{x(t+\Delta t) - x(t)}{\Delta t}.$$

Using this we can take a time step of Δt forward (meaning $t_{k+1} = t_k + \Delta t$) and Euler's method gives us

$$x(t_{k+1}) = x(t_k) + (\Delta t)x'(t_k) \quad \text{and} \quad y(t_{k+1}) = y(t_k) + (\Delta t)y'(t_k).$$

And so we can write our differential equations as difference equations,

$$\frac{x(t+\Delta t) - x(t)}{\Delta t} \approx \dot{x} = \frac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\cos(\theta)$$

$$\frac{y(t+\Delta t) - y(t)}{\Delta t} \approx \dot{y} = \frac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\sin(\theta)$$

$$\frac{\theta(t+\Delta t) - \theta(t)}{\Delta t} \approx \dot{\theta} = \frac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2)$$

After some algebra, we obtain:

$$x(t+\Delta t) \approx x(t) + \tfrac{r\Delta t}{2}(\dot{\phi}_1 + \dot{\phi}_2)\cos(\theta)$$

$$y(t+\Delta t) \approx y(t) + \tfrac{r\Delta t}{2}(\dot{\phi}_1 + \dot{\phi}_2)\sin(\theta)$$

$$\theta(t+\Delta t) \approx \theta(t) + \tfrac{r\Delta t}{2L}(\dot{\phi}_1 - \dot{\phi}_2).$$

[1] that would be a *codeword* for sloppy

Using the discrete (sample) variables, $x(t_k) \to x_k$, etc, we can rewrite the expression in terms of the discrete variables. Given starting configuration and wheel velocity measurements, we have the following difference equations:

$$\begin{aligned}
x_{k+1} &= x_k + \tfrac{r\Delta t}{2}(\omega_{1,k} + \omega_{2,k})\cos(\theta_k) \\
y_{k+1} &= y_k + \tfrac{r\Delta t}{2}(\omega_{1,k} + \omega_{2,k})\sin(\theta_k) \\
\theta_{k+1} &= \theta_k + \tfrac{r\Delta t}{2L}(\omega_{1,k} - \omega_{2,k})
\end{aligned} \tag{3.3}$$

These equations are the main model for approximating motion of a differential drive robot. It has also been used as a first approximation for a tractor or tank drive system. This function is easily coded into Python:

```
def ddstep(xc, yc, qc,r,l,dt,w1,w2):
   xn = xc + (r*dt/2.0)*(w1+w2)*cos(qc)
   yn = yc + (r*dt/2.0)*(w1+w2)*sin(qc)
   qn = qc + (r*dt/(2.0*l))*(w1-w2)
   return (xn,yn,qn)
```

You will need to bring in the math functions:

```
from math import *
```

Assume that $r = 1$, $dt = 0.1$, $w1 = w2 = 2$ and $l = 6$ and take the initial pose to be $x = 1$, $y = 2$ and $\theta = q = 0.7$. The following is a Python program to take 10 steps with the 0.1 time step:

```
xc = 1; yc = 2; qc  = 0.7
t = 0
dt = 0.1
for i in range(10):
   xc, yc, qc = ddstep(xc, yc, qc,1.0,6.0,dt,2.0,2.0)
   t = t + dt
   print t, xc, yc, qc
```

The output:

```
0.1 1.15296843746 2.12884353745 0.7
0.2 1.30593687491 2.2576870749 0.7
0.3 1.45890531237 2.38653061234 0.7
0.4 1.61187374983 2.51537414979 0.7
0.5 1.76484218728 2.64421768724 0.7
0.6 1.91781062474 2.77306122469 0.7
0.7 2.0707790622 2.90190476213 0.7
0.8 2.22374749966 3.03074829958 0.7
0.9 2.37671593711 3.15959183703 0.7
1.0 2.52968437457 3.28843537448 0.7
```

The Euler approximation amounts to assuming the vehicle has constant wheel velocity over the interval Δt, see Fig. 3.16. The assumption of piecewise constant velocity does not hold in the general case and so we see accumulating drift when comparing the robot's true path and the approximated one.

A simple modification of the code can accept other wheel speeds. For example, if the wheel speeds are given by $w1 = 0.1 + 2 * t$ and $w2 = 0.1$, we would have

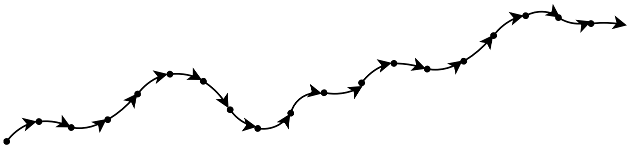


Fig. 3.16: Piecewise Constant nature of the Euler Approximation.

```
xc = 1; yc = 2; qc  = 0.7
t = 0;  dt = 0.1
for i in range(10):
   w1 = 0.1 + 2*t
   w2 = 0.1
   xc, yc, qc = ddstep(xc, yc, qc,1.0,6.0,dt,w1,w2)
   t = t + dt
   print t, xc, yc, qc
```

```
0.1 1.00764842187 2.00644217687 0.7
0.2 1.02294526562 2.01932653062 0.701666666667
0.3 1.0458582885 2.03869127648 0.705
0.4 1.07632275057 2.06461262966 0.71
0.5 1.11424084437 2.09720431822 0.716666666667
0.6 1.15948081421 2.13661681787 0.725
0.7 1.21187577374 2.18303629886 0.735
0.8 1.27122223402 2.23668327131 0.746666666667
0.9 1.33727835762 2.29781091264 0.76
1.0 1.40976195869 2.36670305715 0.775
```

You can plot the motion in Python. Another example with circular motion:

```
import pylab as plt
import numpy as np
from math import *
N=200
x = np.zeros(N)
y = np.zeros(N)
q = np.zeros(N)
x[0] = 1; y[0] = 2; q[0]  = 0.7
t = 0;  dt = 0.1
for i in range(N-1):
   w1 = 0.1
   w2 = 0.5
   x[i+1], y[i+1], q[i+1] = ddstep(x[i], y[i], q[i],1.0,6.0,dt,w1,w2)
   t = t + dt

plt.plot(x,y,'b')
plt.show()
```

3.12.2 Differential Drive Inverse Kinematics

Recall the DD forward kinematics:

$$\dot{x} = \tfrac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\cos(\theta)$$

$$\dot{y} = \tfrac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\sin(\theta)$$

$$\dot{\theta} = \tfrac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2)$$

Starting with the velocity $v = \sqrt{\dot{x}^2 + \dot{y}^2}$, plug in the first two differential equations:

$$\begin{aligned} v &= \sqrt{\left(\frac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\cos(\theta)\right)^2 + \left(\frac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\sin(\theta)\right)^2} \\ &= \sqrt{\left(\frac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\right)^2 \left(\cos^2(\theta) + \sin^2(\theta)\right)} \\ &= \frac{r}{2}|\dot{\phi}_1 + \dot{\phi}_2|. \end{aligned}$$

So, we finally have:

$$|\dot{\phi}_1 + \dot{\phi}_2| = \frac{2v}{r}.$$

Using the third differential equation, $\dot{\phi}_1 = \dot{\phi}_2 + \frac{2L\dot{\theta}}{r}$, we can solve for $\dot{\phi}_2$. We get,

$$|\dot{\phi}_2 + \frac{L\dot{\theta}}{r}| = \frac{v}{r}.$$

Solving for $\dot{\phi}_2$ and then plugging back in for $\dot{\phi}_1$, we have

$$\dot{\phi}_1 = \frac{L\dot{\theta}}{r} \pm \frac{v}{r}, \quad \dot{\phi}_2 = -\frac{L\dot{\theta}}{r} \pm \frac{v}{r}$$

The direction of the robot is the direction of the curve shown in Fig. 3.17.

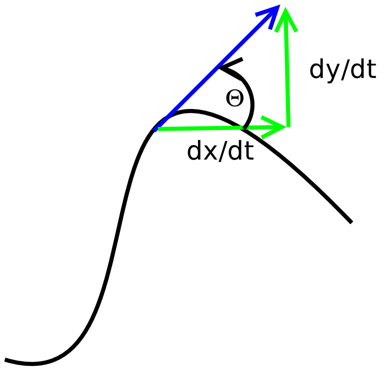


Fig. 3.17: The relation between θ and $\dot{x}$, $\dot{y}$.

$$\theta = \arctan\frac{\dot{y}}{\dot{x}}\ .$$

Differentiation gives

$$\dot{\theta} = \frac{\dot{x}\ddot{y} - \dot{y}\ddot{x}}{\dot{x}^2 + \dot{y}^2}$$

Plugging in we have

$$\begin{aligned} \dot{\phi}_1 &= \frac{L}{r}\left(\frac{\dot{x}\ddot{y} - \dot{y}\ddot{x}}{\dot{x}^2 + \dot{y}^2}\right) \pm \frac{\sqrt{\dot{x}^2 + \dot{y}^2}}{r} \\ \dot{\phi}_2 &= -\frac{L}{r}\left(\frac{\dot{x}\ddot{y} - \dot{y}\ddot{x}}{\dot{x}^2 + \dot{y}^2}\right) \pm \frac{\sqrt{\dot{x}^2 + \dot{y}^2}}{r} \end{aligned}$$

Direction along the path is selected depending on the ±. We will pick the positive root to be consistent with the front of the robot.

$$\boxed{\begin{aligned} \dot{\phi}_1 &= \frac{L}{r}\left(\frac{\dot{x}\ddot{y} - \dot{y}\ddot{x}}{\dot{x}^2 + \dot{y}^2}\right) + \frac{\sqrt{\dot{x}^2 + \dot{y}^2}}{r} \\ \dot{\phi}_2 &= -\frac{L}{r}\left(\frac{\dot{x}\ddot{y} - \dot{y}\ddot{x}}{\dot{x}^2 + \dot{y}^2}\right) + \frac{\sqrt{\dot{x}^2 + \dot{y}^2}}{r} \end{aligned}} \tag{3.4}$$

Note that the curvature of a parameterized plane curve is given by

$$\kappa = \frac{\dot{x}\ddot{y} - \dot{y}\ddot{x}}{(\dot{x}^2 + \dot{y}^2)^{3/2}} = \frac{\dot{x}\ddot{y} - \dot{y}\ddot{x}}{v(\dot{x}^2 + \dot{y}^2)} = \frac{\dot{\theta}}{v}$$

and we can rewrite the inverse kinematic equations, IK, as

$$\boxed{\begin{aligned} v &= \sqrt{\dot{x}^2 + \dot{y}^2} \\ \kappa &= \frac{\dot{x}\ddot{y} - \dot{y}\ddot{x}}{v^3} = \frac{\dot{\theta}}{v} \\ \dot{\phi}_1 &= \frac{v}{r}\left(\kappa L + 1\right) \\ \dot{\phi}_2 &= \frac{v}{r}\left(-\kappa L + 1\right) \end{aligned}} \tag{3.5}$$

Find the wheel velocities for a robot moving in a circle of radius 20. Assume that $r = 1$ and $L = 4$ and using the following parameterization:

$$x = R\cos(t/R), \quad y = R\sin(t/R), \quad \text{where } t \in [0, 2\pi R]$$

and so for our example we have that

$$x = 20\cos(t/20), \quad y = 20\sin(t/20), \quad \text{where } t \in [0, 40\pi].$$

First we must compute, $v = \sqrt{\dot{x}^2 + \dot{y}^2} = \sqrt{\sin^2(x) + \cos^2(x)} = 1$. Next we compute κ:

$$\kappa = \dot{x}\ddot{y} - \dot{y}\ddot{x} = \frac{\sin^2(t/20)}{20} + \frac{\cos^2(t/20)}{20} = \frac{1}{20}.$$

This makes sense since we know the curvature is the reciprocal of the radius. By selecting to go counter-clockwise (increasing θ) we use + in (5.1). Plugging the values into (5.1), we obtain wheel velocities

$$\dot{\phi}_1 = 6/5$$

$$\dot{\phi}_2 = 4/5$$

Assume that you want to follow the path

$$x(t) = t^2, \quad y(t) = t$$

with a differential drive robot (leaving L and r as variables). We must first compute the derivatives

$$\dot{x} = 2t, \quad \ddot{x} = 2, \quad \dot{y} = 1, \quad \ddot{y} = 0$$

and then plug into the equations

$$\kappa = \frac{(2t)(0) - (1)(2)}{((2t)^2 + (1)^2)^{3/2}} = -\frac{2}{(4t^2+1)^{3/2}}$$

$$v = \sqrt{(2t)^2 + 1^2} = \sqrt{4t^2+1}$$

$$\dot{\phi}_1 = \frac{v}{r}\left(\kappa L + 1\right), \quad \dot{\phi}_2 = \frac{v}{r}\left(-\kappa L + 1\right).$$

```
N=100
t0 = 0.0
t1 = 2.0
t = np.linspace(t0,t1,N)
dt = (t1-t0)/N
one = np.ones((N))
xp = np.zeros((N))
yp = np.zeros((N))
th = np.zeros((N))

x = t*t
y = t

plt.figure()
plt.plot(x,y,'g-')
plt.legend(['Path'],loc='best')
plt.title('Quadratic Path')
plt.show()
```

Generate wheel speeds:

```
doty=one
dotx=2*t
ddoty=0
ddotx=2*one

r = 1.0
L = 4.0
```

(continues on next page)

(continued from previous page)

```
v = np.sqrt(dotx*dotx + doty*doty)
kappa = (dotx*ddoty - doty*ddotx)/(v*v*v)
dotphi1 = (v/r)*(kappa*L +1)
dotphi2 = (v/r)*(-kappa*L+1)

plt.plot(t,dotphi1,'b-', t,dotphi2,'g-')
plt.title('Wheel Speeds')
plt.legend(['Right', 'Left'],loc='best')
plt.show()
```

And the section of code to check:

```
xp[0] = 0.0
yp[0] = 0.0
th[0] = 1.5707963267949

for i in range(N-1):
    xp[i+1] = xp[i] + (r*dt/2.0)*(dotphi1[i]+dotphi2[i])*math.cos(th[i])
    yp[i+1] = yp[i] + (r*dt/2.0)*(dotphi1[i]+dotphi2[i])*math.sin(th[i])
    th[i+1] = th[i] + (r*dt/(2.0*L))*(dotphi1[i]-dotphi2[i])

plt.figure()
plt.plot(x,y,'g-', xp, yp, 'bx')
plt.legend(['Original Path', 'Robot Path'],loc='best')
plt.title('Path')
plt.show()
```

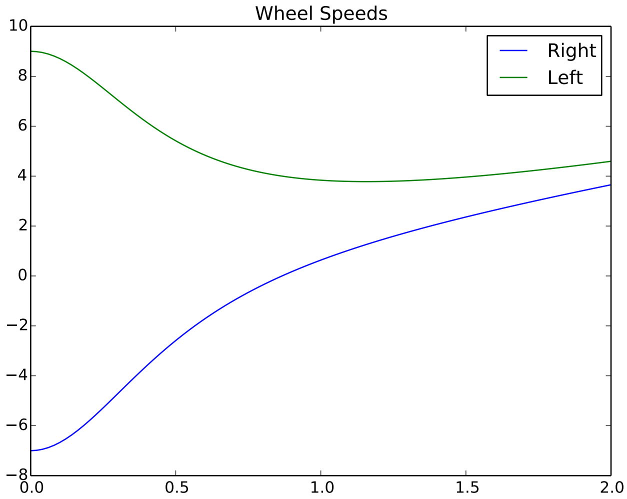


Fig. 3.18: The wheel velocities.

On a robot, the motor controllers will be taking digital commands which means the wheel velocities are discrete. This implies that the robot has fixed wheel velocities during the interval between velocity updates. We know in the case of the differential drive robot, fixed wheel speeds means the robot is driving a line or circle. Therefor the DD robot in this case is following a connected path made up of line or circle segments,

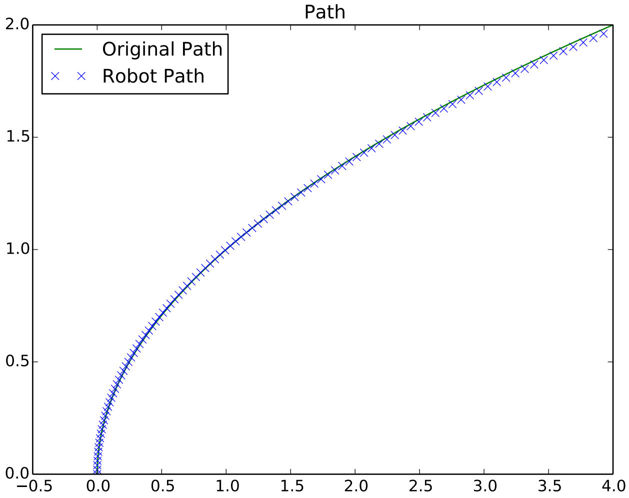


Fig. 3.19: Comparison of the path and driven path.

see Fig. 3.16. Even when we do have functional forms for the wheel speeds, the implementation is still discrete.

It makes sense to treat this as a discrete formula and to write as such:

$$\boxed{\begin{array}{ll} v_k = \sqrt{\dot{x}(t_k)^2 + \dot{y}(t_k)^2}, & \kappa_k = \dfrac{\dot{x}(t_k)\ddot{y}(t_k) - \dot{y}(t_k)\ddot{x}(t_k)}{v_k^3}, \\ \omega_{1,k} = \dfrac{v_k}{r}(\kappa_k L + 1), & \omega_{2,k} = \dfrac{v_k}{r}(-\kappa_k L + 1) \end{array}} \tag{3.6}$$

Determine the wheel velocities to drive through the way points (0,1), (1,2), (2,5). First we compute the derivatives

$$\dot{x} = 1, \quad \ddot{x} = 0, \quad \dot{y} = 2t, \quad \ddot{y} = 2$$

and then plug into the equations

$$\kappa = \frac{(1)(2) - (2t)(0)}{(1+4t^2)^{3/2}} = \frac{2}{(1+4t^2)^{3/2}},$$

$$\dot{\phi}_1 = \frac{v}{r}\left(\kappa L + 1\right), \quad \dot{\phi}_2 = \frac{v}{r}\left(-\kappa L + 1\right).$$

3.12.3 Limitations

In the previous sections we have shown how to drive a robot along any path that the kinematics admits. In the mathematical examples, there are no problems with following a precomputed path. However, this is an example of open loop control and it suffers from many types of error such as discretization error, non-uniform components, variations in power, signals and an unpredictable environment. The robot will drift from the intended path. This drift grows over time.

In practice, we will normally not compute the analytic path from which to compute the derivatives and such to plug into the inverse kinematics. We will use more traditional control algorithms to direct the robot such as a PID controller. We may have a path to follow, but we will not plug that path into the inverse kinematics. Instead we will extract samples from the path and feed destination points into the control algorithm. This does not mean that our efforts working out the inverse kinematics was wasted. Very much to the contrary. We will still use the IK formulas in our controllers. Understanding the IK will help in the controller design. The IK can often help isolate aspects of the system dynamics which eases controller development or makes it possible to gain a stable controller.

3.13 Simulation with Noise

You will see later on we go to great efforts to remove noise from a dataset. So, it might seem odd to have a section on generating noise. However, it is very useful to be able to generate noise for more realistic simulations and to test the filters that are intended to remove the noise. The Numpy library supports the generation of random numbers as well as some convenient functions to draw numbers from certain types of distributions. Most of our work will use normal distributions. The numpy function random.normal will generate random (well, approximately) values drawn from a normal distribution. For example, the following code will generate a scatter plot, see Fig. 3.20-(a).

```
mu1 = 1.0
sigma1 = 0.5
mu2 = 2.0
sigma2 = 1.0
x = np.random.normal(mu1,sigma1,100)
y = np.random.normal(mu2,sigma2,100)
plt.plot(x,y,'b.')
plt.show()
```

This code will generate a sampled line with noise, see Fig. 3.20-(b).

```
mu = 0.0
sigma = 1.0
error = np.random.normal(mu,sigma,100)
x = np.linspace(0,5,100)
y = 2*x+1.0 + error
plt.plot(x,y,'b.')
plt.show()
```

Above we are sampling from a single normal distribution (univariate), however, later on we will need to sample from multivariate distribution. We provide the algorithm below or this can be done with np.random.multivariate_normal.

```
>>> mean = [0,0]
>>> covar = [[.5,.05],[.05,1.0]]
>>> N = 10
>>> np.random.multivariate_normal(mean,covar,N)
array([[ 0.88598172, -0.4423436 ],
       [ 0.13454988, -0.72543919],
       [-0.37652703,  0.74301719],
```

(continues on next page)

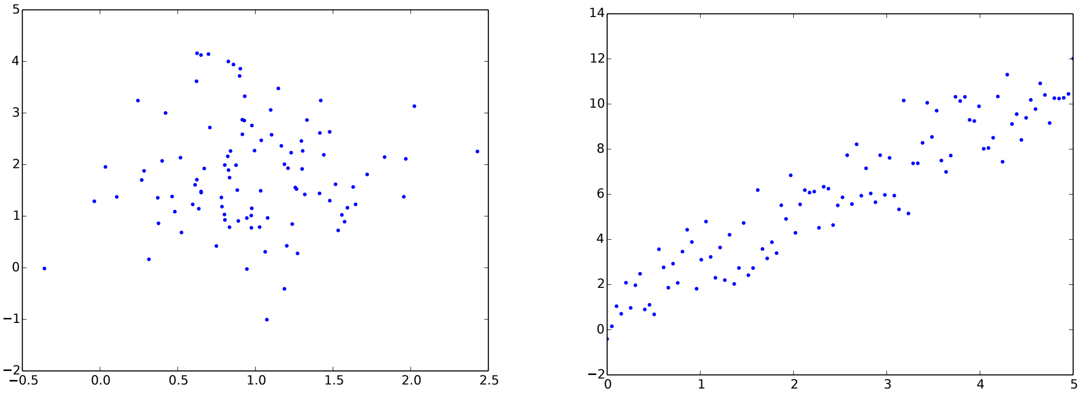


Fig. 3.20: Scatter type plots. a) A scatter type plot. b) line with lots of noise.

(continued from previous page)

```
       [ 0.25273237, -0.63923146],
       [-1.43009133, -0.53752537],
       [ 0.27189567, -0.56165933],
       [-0.23506435,  0.82847583],
       [ 0.47206316,  0.46425447],
       [-0.33998358,  0.4583102 ],
       [-1.07647896,  0.90586496]])
>>>
```

3.13.1 Creating your own distribution

If you want to do this by hand:

1. Generate the random numbers for each variable.
2. Place them into an array.
3. Compute their variance-covariance matrix.
4. Perform a Cholesky factorization on the variance-covariance matrix.
5. Invert the Cholesky factor and multiple it by the random matrix data. This normalizes the dataset.
6. Compute a Cholesky factorization of the desired variance-covariance matrix.
7. Multiply the last Cholesky factor times the normalized data.

```
import numpy as np

N = 100
sigma = 1.0

# Create two vectors of random numbers
#
```

(continues on next page)

(continued from previous page)

```
ex = np.random.normal(0,sigma,N)
ey = np.random.normal(0,sigma,N)

# Stack them into an array
#
D = np.vstack((ex,ey))

# Normalize the distribution
M = np.cov(D)
MC = np.linalg.cholesky(M)
MCI = np.linalg.inv(MC)
MD = np.dot(MCI,D)

# Enter the desired covariance matrix:
#
W = np.array([[0.1, 0.01],[0.01,0.2]])

# Perform the Cholesky decomposition
#
L = np.linalg.cholesky(W)

# Multiply the Cholesky factor L with the data
# (which transforms the data to having the correct
# covariance)
#
# LD is the random data with the correct covariance
LD = np.dot(L,MD)

# Print the result to check if it is close to w
#print(np.cov(LD))
```

The previous gives you a method to generate random values from a distribution. Next we want to use them for various simulation events, normally to understand the system in the presence of noise.

3.13.2 Noise in the DD Robot

The differential drive robot has two control inputs, the right and left wheel speeds. To simulate motion with noise, we can inject small random values into each iteration of the simulation. Assume that we have random values (or vector) ϵ_i, $i = 1, 2, 3$ drawn from some normal distribution $N(\mu, \sigma)$. Note that the distribution N in this example is a Gaussian distribution, but it need not be in general.

Recall the basic discrete motion equations for the differential drive:

$$\begin{aligned}
x_{k+1} &= x_k + \tfrac{r\Delta t}{2}(\omega_{1,k} + \omega_{2,k})\cos(\theta_k) \\
y_{k+1} &= y_k + \tfrac{r\Delta t}{2}(\omega_{1,k} + \omega_{2,k})\sin(\theta_k) \\
\theta_{k+1} &= \theta_k + \tfrac{r\Delta t}{2L}(\omega_{1,k} - \omega_{2,k})
\end{aligned}$$

Noise can be injected directly into the state variables (x, y, θ):

$$\begin{aligned} x_{k+1} &= x_k + \tfrac{r\Delta t}{2}(\omega_{1,k} + \omega_{2,k})\cos(\theta_k) + \epsilon_1 \\ y_{k+1} &= y_k + \tfrac{r\Delta t}{2}(\omega_{1,k} + \omega_{2,k})\sin(\theta_k) + \epsilon_2 \\ \theta_{k+1} &= \theta_k + \tfrac{r\Delta t}{2L}(\omega_{1,k} - \omega_{2,k}) + \epsilon_3 \end{aligned} \tag{3.7}$$

You will note that we are adding a small amount of noise at each iteration step. This is not the same as adding the noise at the end since for the iterative process with noise injected at each step, the noise modifies the path at each step and has a cumulative effect. Adding noise at the end, will just create an end distribution which mirrors the distribution that the noise was drawn from. However, noise injected into the DD forward kinematics time step is subject to a non-linear process and the final distribution is not Gaussian.

Simulation with random variables can be very helpful in understanding the exact impact of noise in a particular state's update. It also models the aggregate noise from various sources into a single additive term. If one wants to study the effects of just noise in the wheel speed, then we inject the noise into the ω terms:

$$\begin{aligned} x_{k+1} &= x_k + \tfrac{r\Delta t}{2}(\omega_{1,k} + \omega_{2,k} + \epsilon_1)\cos(\theta_k) \\ y_{k+1} &= y_k + \tfrac{r\Delta t}{2}(\omega_{1,k} + \omega_{2,k} + \epsilon_1)\sin(\theta_k) \\ \theta_{k+1} &= \theta_k + \tfrac{r\Delta t}{2L}(\omega_{1,k} - \omega_{2,k} + \epsilon_2) \end{aligned}$$

Using (3.7), we can illustrate adding noise. This example uses a robot with r=20, L = 12, $\Delta t = 0.01$ and has a simple turn:

$$\begin{aligned} &\phi_1 = 1.0, \phi_2 = 1.0, 0 \leq t < 1.5 \\ &\phi_1 = 2.0, \phi_2 = 1.01.5 \leq t < 3.0 \\ &\phi_1 = 1.0, \phi_2 = 1.03.0 \leq t \end{aligned}$$

```
def wheels(t):
  if (t < 1.5):
    w1 = 1.0
    w2 = 1.0
    return w1, w2
  if (t < 3):
    w1 = 2.0
    w2 = 1.0
    return w1, w2
  w1 = 1.0
  w2 = 1.0
  return w1, w2
```

The setup for the simulation is

```
r = 20.0
l = 12.0
N = 10
dt   = 0.01
Tend = 5
NumP = int(Tend/dt)
```

(continues on next page)

(continued from previous page)

```
mu1, sigma1 = 0.0, 0.05
mu2, sigma2 = 0.0, 0.01
tp = np.arange(0,Tend,dt)

xpath  = np.zeros((N,NumP))
ypath = np.zeros((N,NumP))
thpath = np.zeros((N,NumP))
```

We selected the same noise range for the x, y variables but a smaller range for the θ variable. Small changes in θ variable can have a greater impact on the final location than small changes in x, y. The arrays xpath, ypath and thpath are declared as two dimensional arrays. This is so we can store multiple paths. Meaning we are storing N paths which are comprised of $Nump$ points.

To create the paths we run a double loop, where the outside loop is over the paths and the inside loop creates the points on a specific path.

```
for k in range(N):
    thv = 0.0
    xv = 0.0
    yv = 0.0
    i = 0
    xerr = np.random.normal(mu1,sigma1, NumP)
    yerr = np.random.normal(mu1,sigma1, NumP)
    therr = np.random.normal(mu2,sigma2, NumP)
    for t in tp:
      w1,w2 = wheels(t)
      dx = (r*dt/2.0)*(w1+w2)*cos(thv) + xerr[i]
      dy = (r*dt/2.0)*(w1+w2)*sin(thv) + yerr[i]
      dth = (r*dt/(2.0*l))*(w1-w2) + therr[i]
      xv = xv + dx
      yv = yv + dy
      thv = thv + dth
      xpath[k][i] = xv
      ypath[k][i] = yv
      thpath[k][i] = thv
      i = i+1
```

This can be visualized by

```
for k in range(N):
    plt.plot(xpath[k],ypath[k], 'b-')
plt.xlim(-3, 130)
plt.ylim(-20, 130)
plt.show()
```

which is shown in Fig. 3.21.

We can keep adding additional paths to see the distribution of the final locations. It gets too hard to see what it going on, so we only plot the last point on a particular path. The noise free path is also included and result is shown in Fig. 3.22.

For linear Gaussian processs, if we ran this millions of times and then produced a histogram of the results,

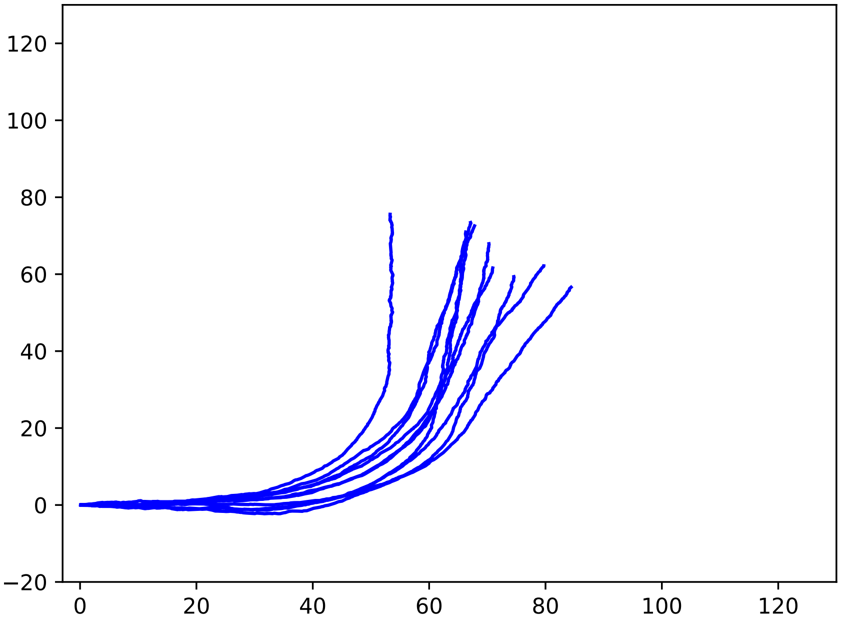


Fig. 3.21: Multiple paths from the same starting point using a noisy model.

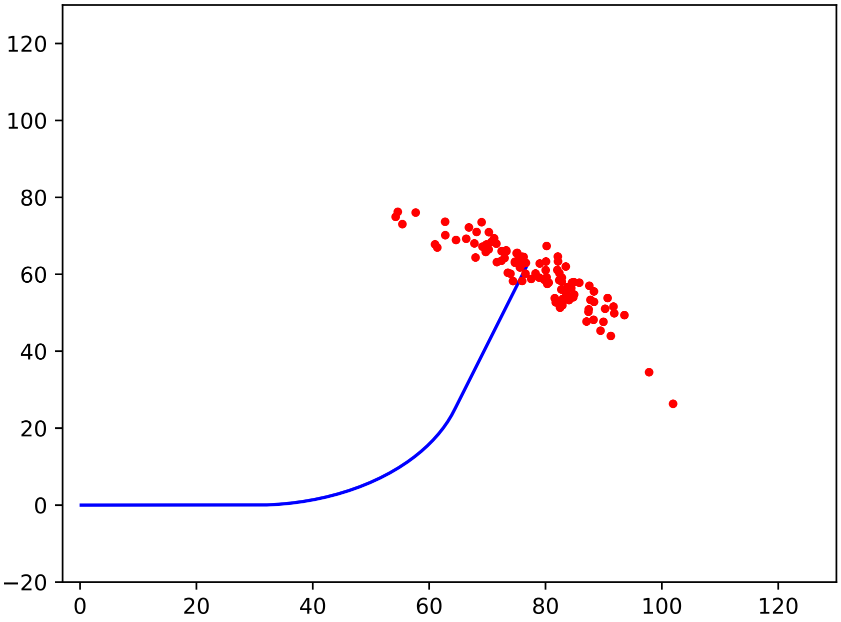


Fig. 3.22: Showing the noise free path and the endpoints for the noisy paths.

we would see a 2D normal distribution emerge. Cross sections of the 2D normal would be ellipses. The larger the ellipse the greater confidence value we have. Since this is *not* a linear process, we don't expect a normal distribution, but we do expect some distribution. So we will treat this in a similar fashion. To compute the error ellipse for the 95% confidence, we store the final points in parallel arrays for x and y, and run the following code block:

```
s = 2.447651936039926
mat = np.array([x,y])
cmat = np.cov(mat)
evals, evec = linalg.eigh(cmat)
r1 = 2*s*sqrt(evals[0])
r2 = 2*s*sqrt(evals[1])
angle = 180*atan2(evec[0,1],evec[0,0])/np.pi
ell = Ellipse((np.mean(x),np.mean(y)),r1,r2,angle)
a = plt.subplot(111, aspect='equal')
ell.set_alpha(0.2)
a.add_artist(ell)

plt.plot(xpath,ypath, 'b-', x,y, 'r.')
plt.xlim(-3, 130)
plt.ylim(-20, 130)
plt.show()
```

What this does is to take the two data sets, x and y and compute the covariance matrix, stored in cmat. The eigenvectors and eigenvalues for cmat are computed. The eigenvectors tell you the tilt angle of the array, The eigenvalues (which are variances) are used to find the major and minor axes lengths (r1, r2).

$$r = 2\sqrt{5.991\lambda}$$

3.14 The Ground Robot World

One of the main differences many see between a vehicle and a robotic vehicle is whether or not a person is *onboard*. If you are driving a car, then we would not call this a robot. But if your car was remotely operated, then some would call it a robotic car.[1] Can we make the robot simulation remotely operated? In this case we mean, *can this be controlled from an external program?* The answer is yes.

The previous robot code examples allow the user to move a simulated device around an open rectangle. The world has obstacles and a simulation should reflect this. So, how should we include obstacles? The simulation is in two dimensions and so the obstacle will also be in 2D. The obstacle is then represented as a 2D shape as viewed from above. The presentation of the simulation is in a window which means at some point the robot and obstacles are presented on a grid or in a discrete fashion. This means we have some choices on how to represent the world, obstacles and other objects, Fig. 3.24.

The environment can be represented in three different manners: continuous, discrete and topological. Continuous is how we tend to think about the world. All of the locations and distances for objects, ourselves

[1] The author would simply call this a remotely operated car, but either way, teleoperation does change how one looks at a vehicle.

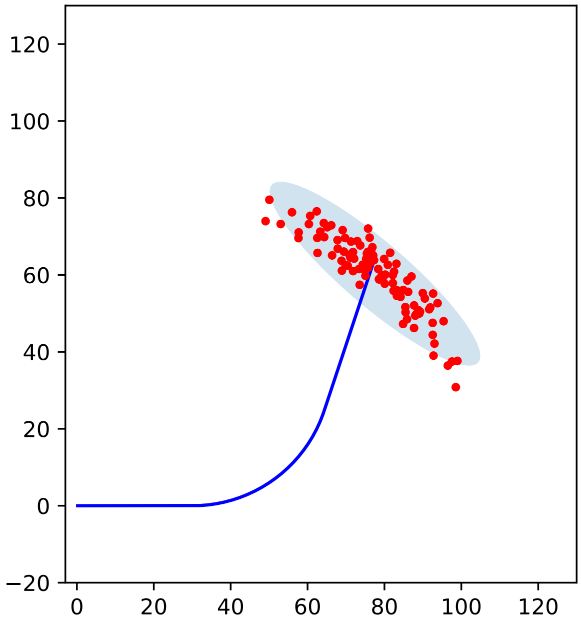


Fig. 3.23: The error ellipse.

and the robots use floating point values. For example, the center of the robot would be located by a pair of floating point values and exact information about the robot shape stored in a database, Fig. 3.25.

For a discrete representation, the world is discretized and objects are located using integer values, Fig. 3.26. The world is then a large checkerboard with a square (pixel) either occupied or not occupied. Simple two or three color bitmaps then suffice (two for object maps and optionally a third to track the robot). Painting a pixel white will indicate that pixel or location is unoccupied. Painting it colored indicates the pixel is occupied. This approach is known as an occupancy grid. The obstacle is simply the collection of black pixels on the occupancy map. A B/W image file can then be used to generate obstacle maps. [One handy way to accomplish this task is to use a paint program (or image editing tool) which can export the image into a format that is easy to read.]

Topological representations do not include metric information like the other two, Fig. 3.27. Relationships are through graphs that indicate two things are connected via a path. How they are connected is another issue. This is very much how humans store maps. You probably know that to get to your favorite restaurant, you have to pass the Home Depot and take the next right. Then you keep going until you pass the Whole Foods market. Then a quick left and there you are. In this description, no distances were provided and even the notion of left and right are flexible since we don't require the streets intersect at right angles.

For the case of the robot simulation, the choice has been partially made. The robot's world appears as an image which is a discretization or a grid. Thus we have a discrete environment. We might decide to go with an obstacle map. Each obstacle is just written into the map and then disappears in to the large collection of filled pixels. Or we may elect to keep our obstacles in a continuous representation. However, this means that translations between the continuous and discrete forms must happen often.

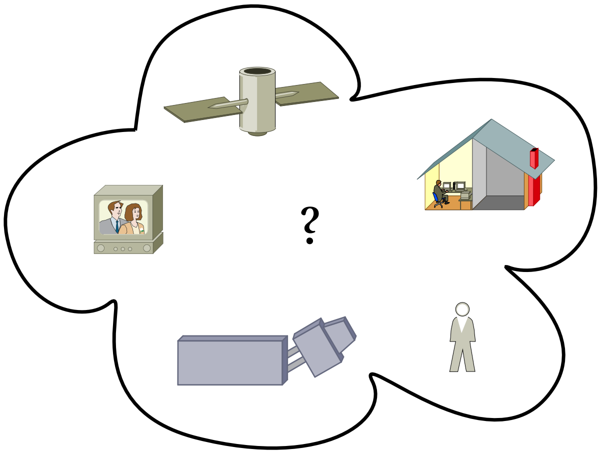


Fig. 3.24: How one should represent the environment.

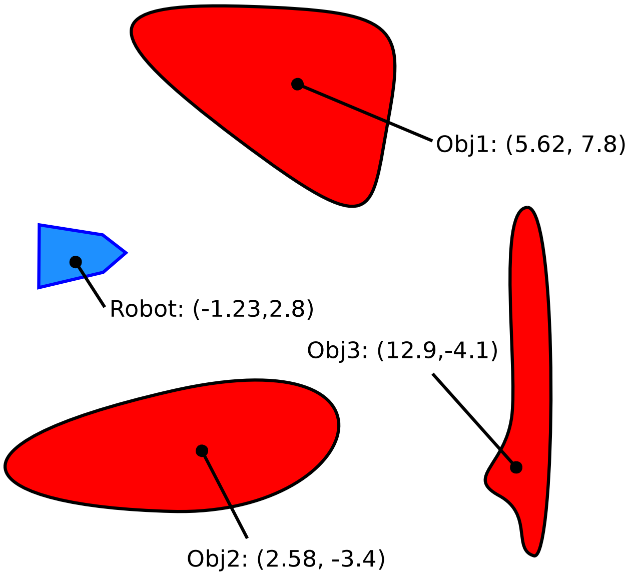


Fig. 3.25: Continuous representation.

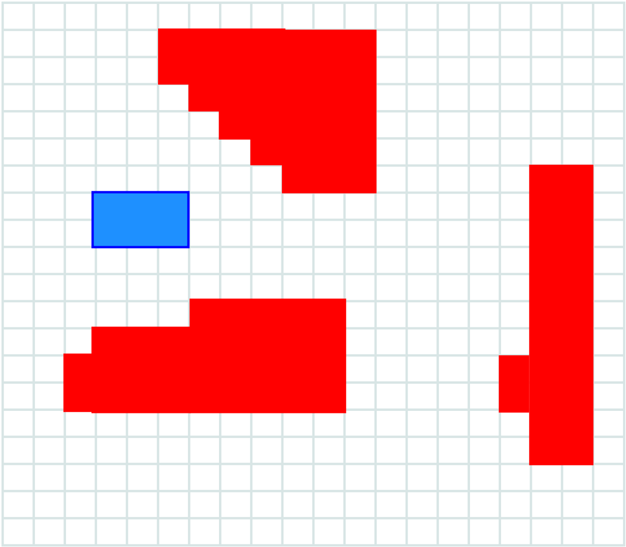


Fig. 3.26: Discrete representation.

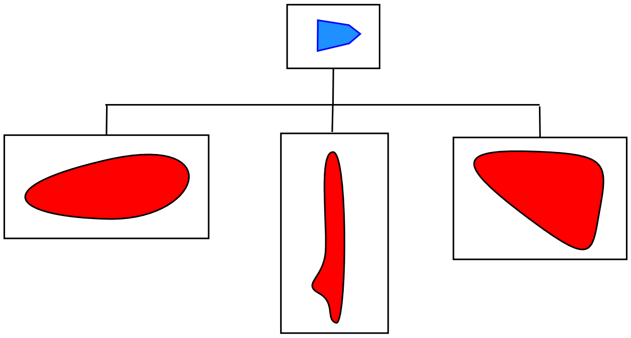


Fig. 3.27: Topological representation.

Continuous and discrete forms each have strengths and weaknesses. We have very precise information in the continuous form. To increase precision in the discrete world, we must decrease pixel size which increases the array storage dramatically or forces a more sophisticated data format over a simple 2D array. Although storage has increased, many operations in the discrete world are much easier.

Consider the problem of simulating a robot impact on a object. Say that the object has an irregular shape. This shape can be approximated by the pixelized version in the discrete world or by a cubic spline approximation using a continuous approximation. True that you have much better accuracy with the cubic spline. The problem is in determining intersection of the robot boundary with the object boundary. In the continuous world, we need to take both of the functions and look for intersecting boundaries at each time step. This requires a complex nonlinear equation solving routine. [Just work out the algebra for two circles intersecting.] For the bitmap version we just check that the front of the robot is on an occupied pixel or cell (if cell[i][j] == 1 then . . . .).

The continuous version will keep objects as objects. For example, if you have disks that touch, the continuous representation will track the centers and radii of the two disks. You always know you have multiple objects. Once converted to a bit map, it could be two adjacent objects or one connected object or multiple partial objects, etc. It is the difference between high and low level representations. A topological representation takes this approach to the next level by removing metric information and just keeping object description in a connectivity graph. Many factors enter into the choice of representation. It is always a trade off between speed, accuracy and simplicity.

3.14.1 Simple Obstacles

The simplest object to study is a disk. It is simple not only in geometry, but in the more difficult task of determining collision. We know that if any part of our robot is within a radius of the center, we have collided. Our robots are round, so collision is just checking the distance between centers minus the radii. It makes a good stage for a first path planning exercise. We assume for the moment that our robot can move freely around the plane (in the open space) and that the plane is covered with disk shaped obstacles. We also assume that the robot knows its coordinate location and heading. For a given obstacle map, can we find a path connecting two points in the plane?

The Python code to check if two disks intersect is fairly straightforward:

```
def collide(center1, r1, center2, r2):
    x1 = center1[0]
    y1 = center1[1]
    x2 = center2[0]
    y2 = center2[1]
    d = sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2))-r1-r2
    return d
```

Where center is a list and r is the radius.[2]

To check for intersection, we only need to check that d is small. Using this we may build a method for a contact sensor. You can treat a contact sensor as a disk of zero radius and use the formula above (adjusting for the relation between the center of the robot and the sensor). Many early robots had sensors placed in a

[2] Keep in mind that the robot graphics circle method draws from the bottom of the circle and so the center for this formula and the one for the circle method need to be adjusted by the radius.

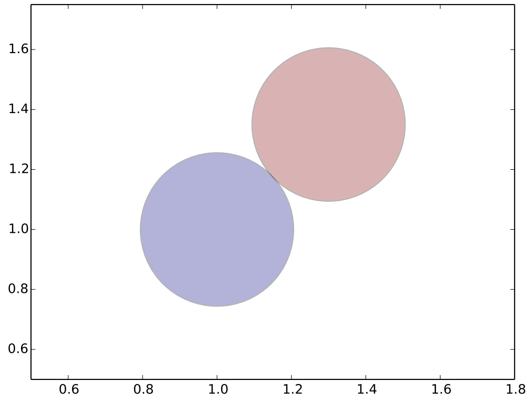


Fig. 3.28: Collision detection with circular robots.

ring around the body of the robot, Fig. 3.29. For this example, they will be contact or touch sensors, but in experimental units often low cost ultrasonic ranging sensors would be used.

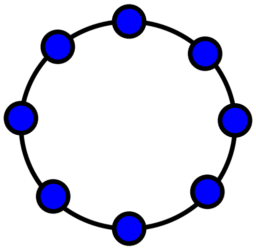


Fig. 3.29: A circular robot (like a Create) with touch sensors mounted around the body.

Assume that you have a circular robot with a ring of touch or bump sensors around the body. Knowing the direction of travel, it is possible to estimate the boundary of the obstacle relative to the robot, Fig. 3.32. The boundary normal can be estimated from the vector created by the sensor location to the robot center. This is a local estimate only as Fig. 3.32 shows. Being able to estimate the boundary means that a robot can follow the boundary. The tangent to the boundary is required for this task.

Using the normal vector, $\hat{n} =< n_1, n_2 >$, the tangent to the boundary is computed via

$$T = \pm < n_2, -n_1 >$$

where the sign is taken so that motion is to the right (right hand rule). This tangent direction will provide the motion direction for a boundary following approach. Estimation of the tangent or the direction of travel can be done with a ring of touch sensors, Fig. 3.32.

Using a range sensor

Recall the components in Fig. 1.14. There was not a touch or impact sensor listed. However, there are two types of range sensors shown. One is a LIDAR and the other is a Kinect. The next simple planner presented assumes that the robot has a ranging device. The simplest to model is the LIDAR.

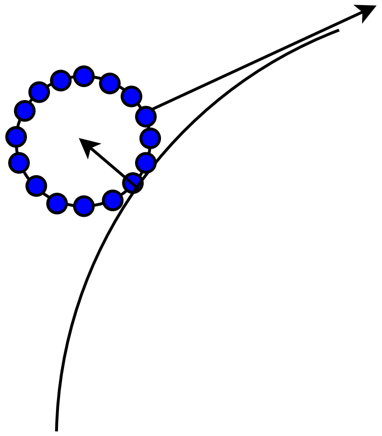


Fig. 3.30: Estimating the object boundary.

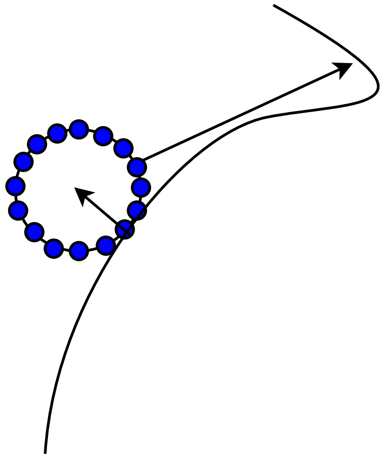


Fig. 3.31: Bump sensors can only determine the nature of the boundary at the contact location.

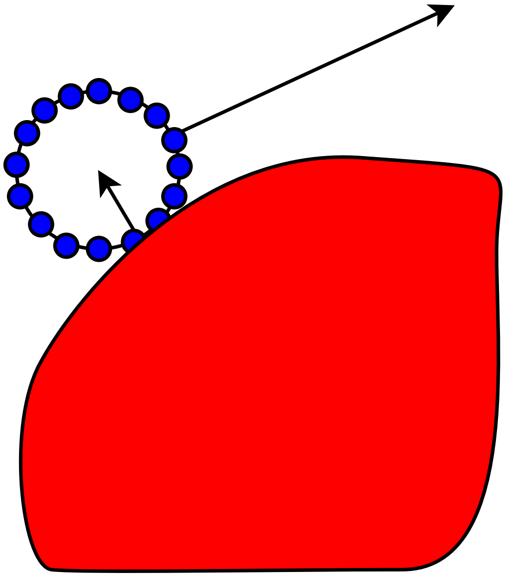


Fig. 3.32: Using touch sensors to estimate the boundary normal and tangent.

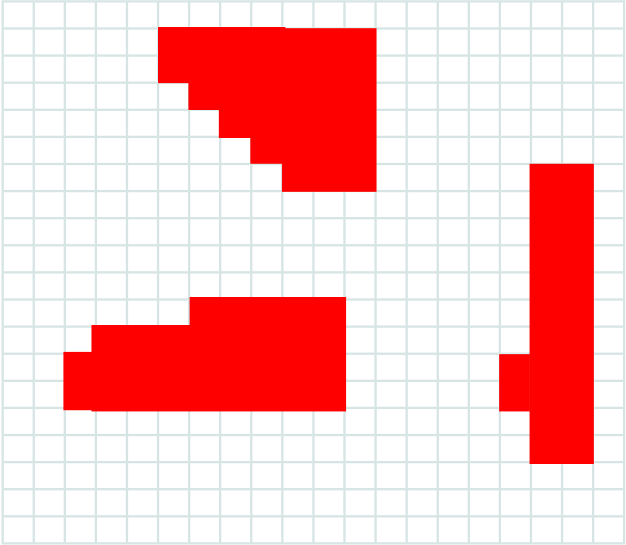


Fig. 3.33: Discrete object map.

A lidar is a simple device conceptually. The unit is able to sweep or turn in one direction which for our discussion we assume it is horizontal. It chops up the angular variable into some number of discrete angles. At each angle or direction, the lidar unit projects a laser beam out. It receives the reflected signal and computes the distance. Naively one simply measures the time of flight, divides by two (for the round trip) and multiplies by c (the speed of light): $D = RT$. This provides the distance of the nearest obstacle at the current angle. Record the number and move to the next angle.

A sweep creates an array of values where the array index is a function of the angle and array values are distances. The unit will return the array. Angles can be reconstructed if you know the starting angle and the angular increment: $\theta_i = \theta_0 + i\Delta\theta$. If you are simulating a given LIDAR unit, then one would use the increment angle of that unit. If not, then you will decide on the details of angular increment, maximum range, minimum range and data rate.

How is this done in a discrete environment? Using a two colored image, let white be free space and red or black indicate occupied space. To simulate the beam out of the LIDAR, create a virtual line out of the lidar and follow a straight line along white pixels until you run into a colored pixel. Stop at the first colored pixel. Using the endpoints of the line segment (virtual lidar to object pixel), the distance can be computed. Let (n, m) be the start of the line and let (i, j) be the location of the object pixel and recall the distance is $d = \sqrt{(i-n)^2 + (j-m)^2}$.[3]

Any actual lidar unit has an effective range, R. In simulation one could certainly compute d as you move out along the ray (or line) and stop when the max range occurred. This approach will work but it requires computing the distance function within the innermost loop and will not result in efficient code. A more effective approach is to just step out in the radial variable. This means you need to represent the line or ray in polar coordinates. We will assume that R is given in the pixel coordinates and the range would be $0 \leq r \leq R$. The other issue is increment value for the lidar simulation. Again, if this value is taken from an actual unit, then that is the value to use. Otherwise, at the maximum range, R, we would like that an increment in the angle selects the "next" (adjacent) pixel. So we want $\Delta\theta$ to be small enough to hit all the pixels, but no smaller for performance reasons, see Fig. 3.34 (b).The circumference is $2\pi R$. If a pixel is 1^2

[3] If you wanted an integer array you would cast this as an `int`.

units, then we select $\Delta\theta \approx 1/(2\pi R)$ (or slightly smaller).

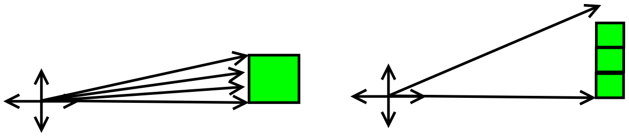


Fig. 3.34: Laser angle increments. (a) The first is too small and we resample the same pixel. (b) The second increment is too large and we miss pixels.

The lidar simulation algorithm is given in algorithm: *Lidar Simulation*

Lidar Simulation Algorithm

```
$k = 0$
$\Delta\theta = 1/(2\pi R)$
for $\theta = 0$ to $2\pi$
    for $r = 0$ to $R$
        i = (int) $r \cos\theta$
        j = (int) $r \sin\theta$
        if Map(i,j) is occupied then
            break from $r$ loop
        endif
    endfor
    dist(k) = $r$
    k++
    $\theta += \Delta\theta$
endfor
```

3.15 Problems

1. Why is simulation useful to roboticists?
2. List some of the advantages and disadvantages of simulating a robot vs. working with physical robots.
3. Using Python and ZeroMQ, write a chat program (call it *chat.py*). First prompt the user for their name. Write to all members in the chat group that this person has entered the chat. In a loop, grab user inputs and broadcast to the chat with format: name: <user input> . Echo to the terminal all strings sent to the chat.
4. Using Python and ZeroMQ, modify the example programs in the text on the kinematics of the two link manipulator.
    1. Write a program that creates a list of 100 equally spaced points along the path $y = 15 - x$ for

   $0 \leq x \leq 10$ and publishes those points on the topic *physData* using a multiarray floating data type, i.e. values x and y are floats. Publish the data at 5Hz.

   2. Write a program that subscribes to topic *physData*, plugs the values in, computes the serial two link inverse kinematics to gain the servo angles (pick one of the +/-) and publishes the angles to the topic /thetaData. You may assume the link arms are $a_1 = a_2 = 10$. Format will be the same as the previous topic.

   3. Write a program that subscribes to both *physData* and *thetaData*. The program should plug the angles into the forward kinematics and check against the data in *physData*. It should plot the original curve in green and the “check” in blue.

5. Assume that you have a parallel two link manipulator with $L_0 = 10$cm, $L_1 = 15$cm and $L_2 = 20$cm.

   1. Write a ZeroMQ program that creates a list of 100 equally spaced points along the path $x = 7\cos(t) + 10$, $y = 5\sin(t) + 15$ and publishes those points on the topic *physData* using a multiarray floating data type, i.e. values x and y are floats. Publish the data at 5Hz.

   2. Write a ZeroMQ program that subscribes to topic *physData*, plugs the values in, computes the serial two link inverse kinematics to gain the servo angles and publishes the angles to the topic *thetaData*. Format will be the same as the previous topic.

   3. Write a ZeroMQ program that subscribes to both *physData* and *thetaData*. The program should plug the angles into the forward kinematics and check against the data in *physData*. It should plot the original curve in green and the “check” in blue.

6. Using Python and 0MQ write a program that will add padding to obstacles while shrinking the footprint of the robot to a point. Assume that you have a circular robot with radius 10 and starting pose (15,15,90).

   1. Write a program that will publish the pose of the robot on the topic *robot_pose* and the footprint type of the robot on *robot_footprint* as a string (For example circle or polygon). Also publish the radius of the robot on *robot_radius*.

   2. Write a program that will publish a list of obstacles as polygons on the topic *obstacles*. For this program let the obstacles be the following:

      1. Rectangle with the vertices (40,30), (50,5), (50, 30) (40,30).

      2. Rectangle with the vertices (40,5), (50,5), (50,0), (40,5).

   3. Write a program that subscribes to *robot_pose*, *robot_footprint*, and *obstacles*. Based on the footprint string, this program should be able to subscribe to either the robot radius or dimension topics for circular and rectangular robots. This program will reduce the robot footprint to a point, add padding to the obstacles, and plot the robot as a point and padded obstacles with the maximum x and y values being 70 and 30.

7. Rework the previous problem assuming that you have a rectangular robot with $width = 10$ and $length = 20$ and initial pose (0,10,0).

8. Using Python and ZeroMQ, write a program to calculate the motion of a differential drive robot.

   a. Write a program that publishes a sequence of wheel velocities on the topic *WheelVel* at 10Hz. Use the multiarray datatype. This node should be named *Control*. This program should also publish on a topic named *Active* either 1 or 0 at 1 Hz to say whether or not the robot is active (meaning done with wheel velocities and you can plot now: active =1, done = 0). Demonstrate the code on $\dot{\phi}_1 = 2 + 2e^{-t_n}$ and $\dot{\phi}_2 = 2 + e^{-2t_n}$ for $0 \leq t \leq 10$.

b. Write a program that uses the differential drive kinematics to derive the robot linear and angular velocities. Publish the velocities using a message and name the topic *RobotVel*. This node should be named *ForwardK*. Assume that $D = 10$, $L = 20$ and the robot starts at (0,0,0).

c. Write a program that will subscribe to the twist message and plot the robot's path using Python plotting when it gets the signal on the Active topic. This node should be named *RobotPlot*.

9. Using the above problem, replace the initial pose (0,0,0) in the second part, (b), with the pose (2,2,45).

10. Using the forward difference on $x(t) = t^2$, what is the error on the derivative value for $\Delta t = 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$ at the location $t = 1$.

11. Let $r = 10$, $L = 20$, $\Delta t = .1$. Find the discrete kinematic model if the wheel velocities are $\dot{\phi}_1 = 2(1 - e^{-t})$, $\dot{\phi}_2 = 2(1 - e^{-2t})$.

12. Using the discrete model equations in the previous, plot the path for $0 \leq t \leq 5$.

13. For the integral in (3.2), use a numerical differential equation solver (with some software package) to integrate the equations. Compare this to using a Taylor expansion on the equations to work out the integrals.

14. What is the smooth ($\dot{x}$, $\dot{y}$ are continuous) parametric form of

    a. $y = (3/2)x + 5/2$

    b. $y = x^{2/3}$.

    c. $(x - 3)^2/16 + (y - 2)^2/9 = 1$.

15. Find the analytic wheel velocities and initial pose for a differential drive robot tasked to follow ($r = 1$, $L = 4$) the given paths. Plot the paths and compare to the actual functions to verify.

    a. $y = (3/2)x + 5/2$

    b. $y = x^{2/3}$

    c. $(x - 3)^2/16 + (y - 2)^2/9 = 1$

16. Find the wheel velocities and initial pose for a differential drive robot tasked to drive a square with corners (0,0), (10,0), (10,10), (0,10). You should stop and turn at a corner. Drive the edges at unit speed. Plot the paths and compare to the actual functions to verify.

17. Find the wheel velocities and initial pose for a differential drive robot in an infinity (∞) shape. Plot the paths and compare to the actual functions to verify.

18. Write robot control code to drive the robot along the indicated paths and plot the obstacles and paths in matplotlib.

    a. the triangular path with vertices (0,0), (15,0) , (5,20),

    b. the square path with corners (0,0), (10,0), (10,10), (0,10),

    c. the circular path centered at the origin and radius is 15.

19. Create two circular obstacles. The first obstacle is a disk centered at (5,5) with radius 2. The second is a disk centered at (15,15) with radius 3. Write the control code to drive a figure 8 around the two obstacles. Run at least two loops. Plot the obstacles and path in matplotlib.

CHAPTER FOUR

NAVIGATION

It can be argued that the single most important aspect of a robot is its ability to move. Roughly motion is a necessary condition but not a sufficient condition. Motion itself is not complicated. Requiring only servos and motors, motion is easily accomplished. The complexity arises through the interaction of the environment. In this chapter we explore how robots move in the plane and navigate around simple landscapes. Although ground robots have to address three dimensional environments, the restriction to the plane simplifies the mathematics and the algorithms allowing us to focus on concepts and not the complexities of the extra dimension.

Motion planning is an entire field of study, we will highlight some aspects here. The solution to the planning problem routes from an initial configuration, start location and pose, to a final configuration, end location and pose or goal.

The basic path planning problem refers determining a path in configuration space such that the robot does not collide with any obstacles and the path is consistent with the vehicle constraints.

4.1 Basic Motion Planning

4.1.1 Simple Planning

When controlling the robot without feedback, open loop control, we preplan the route and then code up a list of motion instructions. For differential drive robots, the easiest routes to drive are combinations of lines and circles, Fig. 4.1. If you have a rough idea of the route, place some points along the route and connect with line (or circle) segments. Along those segments, the differential drive has constant wheel speed. In practice this is difficult since one cannot have instant jumps in wheel velocity. This makes accurate turns challenging. If stopping and turning in place on the route is acceptable, paths with just straight lines are the easiest to develop, Fig. 4.2. Then is is just a matter of starting with the correct orientation and driving for a given amount of time.

There is a clear problem with open loop control (preprogrammed on any path without sensor feedback). Any variation in the physical system can cause drift. This drift accumulates over time and at some point the robot is not driving the intended course. The other problem is that the path is tuned to a specific obstacle field. We must know the obstacles and their locations prior to moving. A more advanced algorithm would be able to take a goal point and using knowledge of the current robot location, drive itself to the goal. The basic motion algorithm attempts this next step.[1]

[1] This algorithm is slightly more general in that it does not need the goal location, but just the direction to the goal during the

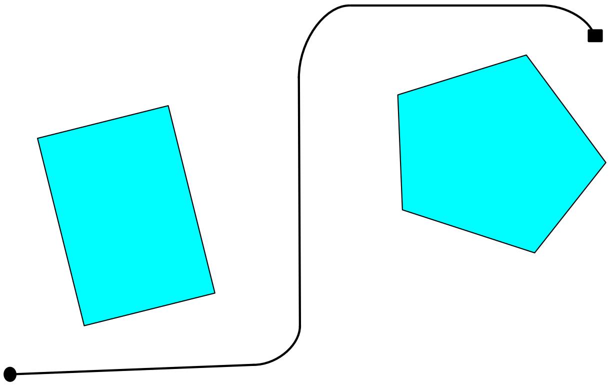


Fig. 4.1: Path with arcs

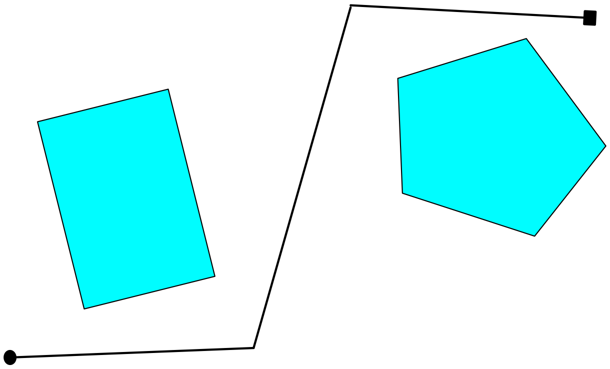


Fig. 4.2: Path *without* arcs

4.1.2 Basic Motion Algorithm

Assuming we have a simple obstacle map, how should we proceed? Try the following thought experiment. Pretend that you are in a dark room with tall boxes. Also pretend that you can hear a phone ringing and you can tell what direction it is. How would you navigate to the phone? Figuring that I can feel my way, I would start walking towards the phone. I keep going as long as there are no obstructions in my way. When I meet an obstacle, without sight (or a map) I can't make any sophisticated routing decisions. So, I decide to turn right a bit and head that way. If that is blocked, then I turn right a bit again. I can continue turning right until the path is clear. Now I should take a few steps in this direction to pass the obstacle. Hopefully I am clear and I can turn back to my original heading. I head in this direction until I run into another obstacle and so I just repeat my simple obstacle avoidance approach.

Basic Motion Algorithm

```
Set heading towards goal
while Not arrived at goal do
    while No obstacle in front do
        Move forward
    end while
    count = 0
    while count <= N do
        while Obstacle in front do
            Turn right
        end while
        Move forward
        incr count
    end while
    Set heading towards goal
end while
```

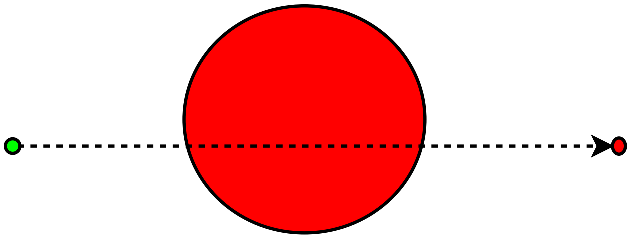


Fig. 4.3: The direct path to the goal.

Fig. 4.4 illustrates the idea. This algorithm is not completely specified. The amount of right turn and the distance traveled in the move forward steps is not prescribed above. Assuming values can be determined,

process.

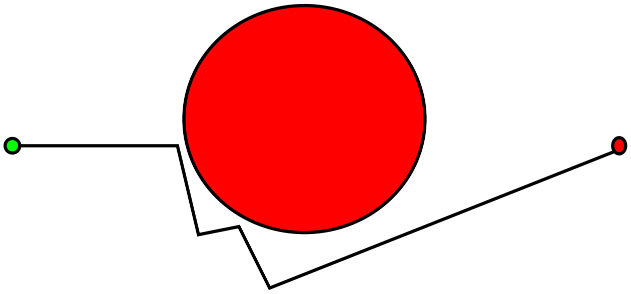


Fig. 4.4: Path using the Basic Motion algorithm.

will this approach work? We expect success when faced with convex obstacles but not necessarily for non-convex obstacles, Fig. 4.5. Using Fig. 4.5 as a guide, we can construct a collection of convex obstacles which still foil the algorithm; this is expressed in Fig. 4.6. The robot bounces from obstacle to obstacle like a pinball and is wrapped around. Leaving the last obstacle the robot reaches the cutoff distance and then switches back to the "motion to goal" state. However, this sets up a cycle. So, the answer to the question "does this work" is not for all cases.

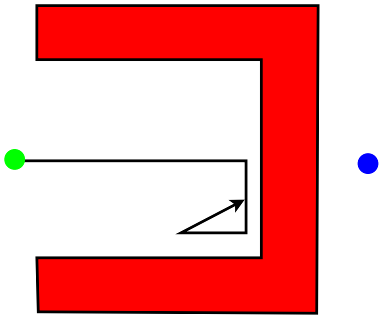


Fig. 4.5: Getting trapped in a non-convex solid object.

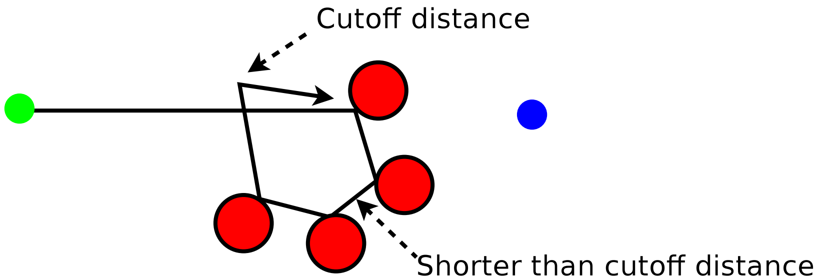


Fig. 4.6: A collection of convex objects can mimic a non-convex obstacle.

In the Chapter on Motion Planning, we will fully explore the challenge of motion planning in an environment with obstacles. It is easy to see how the thought experiment above can fail and more robust approaches are needed. Before we jump into motion planning, we want to understand what view of the world we can get from sensors. This is necessary so we know what kind of assumptions can be made when developing our algorithms.

4.2 Exploration and Navigation

Motion planning is complicated. Even when we restrict ourselves to two dimensions and have a polygonal constraint set, the algorithms for successful motion are much harder than one would expect. We will use the term navigation to mean the process of guiding the robot from one point in the workspace (or configuration space) to another. Implicitly we mean that we look for an acceptable path from one point to another that avoids collisions and respects machine constraints. To navigate, it is necessary to know the starting point of the process, the ending point of the process and the current location. Determining the current location of the robot relative to the landscape turns out to be a very difficult problem in general. This is known as localization.

In many applications, a map of the environment is necessary. The map might be given apriori. The map might be the desired result of the robot exploration. In the latter case, the process of generating a map, simply known as mapping is another significant challenge in the exploration process. There is often a "chicken and egg" problem that arises. If you don't have a map, then it is not possible to localize. Without localization, one can't build a map from sensor data. It least this is how it seems at first. The process known as SLAM, Simultaneous Localization and Mapping, addresses this problem by building the map dynamically and tracking robot location. This is discussed in a later chapter.

There are many approaches to navigation. We will separate them based on algorithmic properties as well as computation difficulty. Does the algorithm provide the optimal result (such as the shortest path) or does it return a satisfactory solution. Will the algorithm always return a solution if one exists (meaning the algorithm is complete) or can it fail without definitive answer? Is the algorithm deterministic or stochastic?

Some algorithms are fast (and this is relative to the current hardware) and some are slow. Algorithms that construct a solution as the robot navigates are said to be online. Those that construct a path apriori, in a batch mode, are said to be offline. Sometimes offline and online are used as a speed issue and this is a function of the computing power on the robot. Does the algorithm increase in memory or computational requirements as a function of runtime or obstacles encountered? If so, is the growth polynomial or exponential? These questions and more arise in the study of motion planning as the do in any course on algorithms. We will start with some relatively simple problems.

The wonderful text *Principles of Robot Motion* [CLH+05] begins the study of planning with three basic navigation algorithms or planners that are similar to the maze routines presented in the last section. These are adaptations of basic planners by Lumelsky and Stepanov [LS87]. Using very simple models one can strip out the non-essential elements and focus on the core issues related to path planning. Even so, as Choset points out "even a simple planner can present interesting and difficult issues." The bug algorithms of Lumelsky and Stepanov can illustrate the adaptations of depth first and greedy search approachs to more general domains.

Choset's notation for the bug algorithms is standard usage and we will be consistent with their choice. The real line or any one dimensional quantity will be indicated by $\mathbb{R}$; the two dimensional plane by $\mathbb{R}^2$; and three dimensional space will be denoted by $\mathbb{R}^3$. Higher dimensional spaces will be denoted in the same manner by $\mathbb{R}^n$. The robot workspace will be denoted by $\mathcal{W} \subset \mathbb{R}^n$ and the workspace obstacles (specifically the i^{th} obstacle) by $\mathcal{WO}_i$. Free space is then the workspace minus the obstacles: $\mathcal{W} \setminus \bigcup_i \mathcal{WO}_i$. A point in space has the usual notation $x = (x_1, x_2, x_3)$ and we distinguish vectors by using a bracket: $\vec{v} \in \mathbb{R}^n$ or more often $v = [v_1, v_2, \ldots, v_n]^T$. A workspace is said to be bounded if $\mathcal{W} \subset B_r(x) \equiv \{y \in \mathbb{R}^n | d(x, y) < r\}$ for some $0 < r < \infty$.

We will make several assumptions for this section:

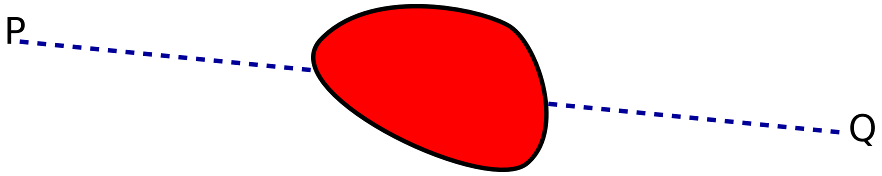


Fig. 4.7: The bot's direction and the obstacle. How does the bot arrive at the desired destination?

- The robot is a single point. Thus we can ignore the boundary-obstacle intersection problem.
- The robot is able to detect an obstacle by touching it.
- Robot knows its pose (location and orientation): (x, y, θ) and it knows the direction to the goal.
- The robot is able to measure distance between any two points: $d[(x_1, y_1), (x_2, y_2)]$.

Planning or routing problems are often more than just navigating a path around obstacles that does not violate vehicle constraints. There are additional issues. We might require the algorithm to produce the minimal distance path or the minimum travel time path.[1] A very common problem that humans must resolve is moving obstacles. Driving is a fine example of moving the vehicle along an obstacle free path within the vehicle contraints and dealing with other moving vehicles.

Driving is also an example of another type of constraint. We normally resolve safe paths. These may be defined as paths which maximize distance from obstacles or have some other relation to the landscape. Information may be incomplete when planning and so we require that the algorithm can run in an interactive manner which can monotonically improve the solution as additional information or computation is provided.

As we did earlier, we will make some simplifying assumptions to get started. We assume we have a point (mass) robot. Essentially this is done by assuming the robot is rigid and we can reduce the robot to the center of mass where we compensate by inflating the obstacles. In addition, we will assume that the domain boundary is smooth and there are a finite number of obstacles all with piecewise smooth boundary.

When designing an algorithm, we must keep in mind issues of the environment and the robot, robot geometry and capability. We must concern ourselves with the soundness of the path, optimality of path as well as the computation resources which are available, The algorithm must balance the needs for a fast robust solution with the time available to obtain a solution.

Our first foray into planners develops several very simple planners which emulate insects. These will be used to illustrate the issues involved with motion planning in unstructured domains. These are also local planners in that they don't need to know the entire obstacle domain.

4.3 The Bug Algorithms

4.3.1 Bug 1

The Bug 1 algorithm is a very simple planner. In the absence of an obstacle, it makes sense to head towards the goal, and if an obstacle is met, then it makes sense to go around the obstacle. So, Bug 1 follows our basic intuition for how the robot should move. This robot is blind - although it knows where the goal is (as

[1] These need not be the same. For example certain paths may be traversed at different speeds depending on location and path geometry.

a direction). For example if you are walking on a very dark night and cannot see your surroundings, but can see the north star. This provides a direction, but does not illuminate the landscape.

Adding on an exit strategy completes the algorithm. As the robot circumnavigates the obstacle, it computes the distance from itself to the goal. After circumnavigation, the robot will continue on the boundary until it finds the closest point to the goal along the boundary. This point will be the exit point for the obstacle. The idea behind this is, the longer the traverse from the boundary to the goal, the higher chance we encounter another obstacle, so we slide along the boundary until this distance is at a minimum.

The bug 1 algorithm [Choset:2005:PRM]

```
Input A point robot with a tactile sensor
Output A path to the $q_{\text{goal}}$ or a conclusion no such path exists.
while True do
    repeat
        From $q^L_{i-1}$ move toward $q_{\text{goal}}$
    until $q_{\text{goal}}$ is reached or obstacle is encountered at hit point $q^H_i$
    if Goal is reached then Exit endif
    repeat
        Follow obstacle boundary
    until $q_{\text{goal}}$ is reached or $q^H_i$ is re-encountered.
    Determine the point $q^L_i$ on the perimeter that has the shortest distance to the goal
    Go to $q^L_i$
    if the robot were to move toward the goal then
    Conclude $q_{\text{goal}}$ is not reachable and exit
    endif
end while
```

By assumption, Bug 1 has contact sensors so will determine the obstacle by direct contact. The contact point will be labeled q^H_i (where i indicates the i-th contact point). After contact with the obstacle, the robot switches to boundary following mode. Similarly, point of departure will be denoted q^L_i. In terms of a state machine, we have moved from the movement to goal state to the boundary following state. We will use Choset's terminology here and call the point of contact, the *hit point*. When the bug departs from the object, we call it the *leave point*. This point is the closest point on the boundary to the goal, but does not mean the line of sight (later defined as the m-line) is obstacle free.

Bug 1 completely investigates each obstacle. It is exhaustive in terms of the boundary search. By looking at the paths in Fig. 4.8, it is appears that Bug 1 is not the most efficient path planner. It does not, nor does it claim to, find the shortest valid path from the start to the finish. Not all problems are even solvable. The planning problem shown in Fig. 4.9 does not have a solution, so Bug 1 will exit without success on this one.

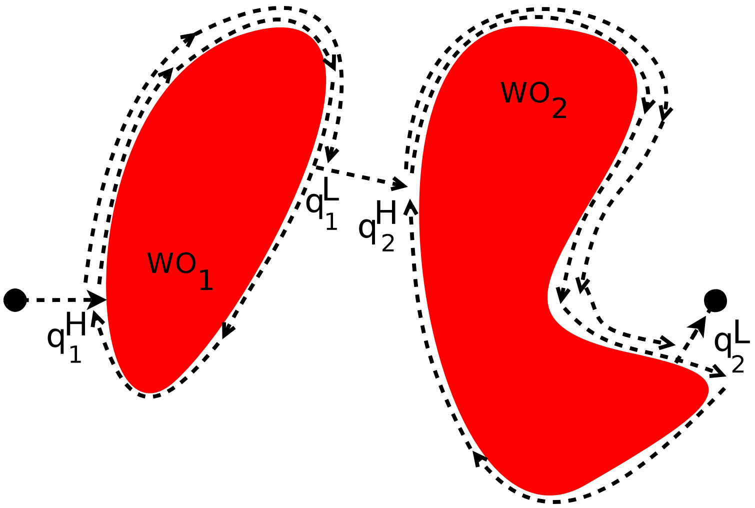


Fig. 4.8: An example of a path using the Bug 1 algorithm.

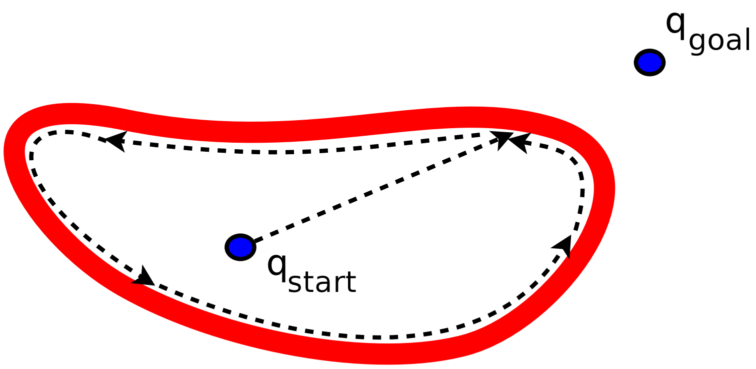


Fig. 4.9: An example of an unreachable goal.

4.3.2 Bug 2

The path that Bug 1 takes is clearly not the shortest path from start to goal, as shown in Fig. 4.10. The first thing you might ask, is "why go all the way around the obstacle"? Once you go around the obstacle and you can resume your original path. Define the line between the start point and the goal point as the m-line (motion to goal line).

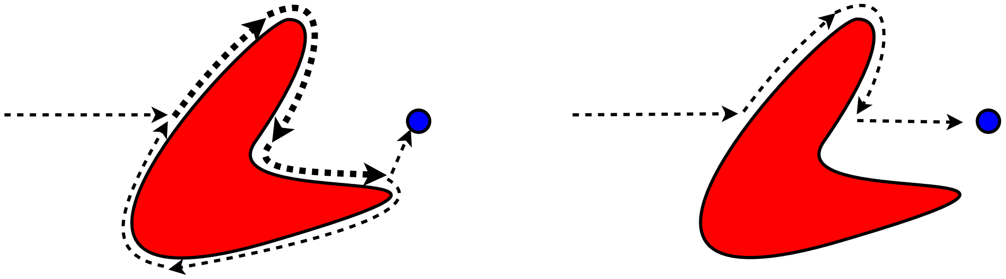


Fig. 4.10: Shortening the path by eliminating the circum-navigation used in Bug1. Thus we no longer have an exhaustive search process.

For the Bug 2 algorithm, motion begins along the m-line in the direction of the goal. When an obstacle is encountered, motion switches to boundary following mode. It is customary to select boundary traversal direction to be "in the direction of travel". [If the direction of travel is $\vec{v}$ and the boundary direction or boundary tangent is $\vec{a}$, then $\vec{v} \cdot \vec{a} > 0$. In the case where $\vec{v} \cdot \vec{a} = 0$, then pick a convention like "go left".] During boundary following mode continue until the m-line is re-emcountered. If the bug can depart in the direction of the goal, it proceeds along the m-line towards the goal or the next obstacle. If the bug cannot depart, then conclude that there is no path to the goal.

The bug 2 algorithm [Choset:2005:PRM]

```
Input A point robot with a tactile sensor
Output A path to the $q_{\text{goal}}$ or a conclusion no such path exists.
while True do
    repeat
        From $q^L_{i-1}$ move toward $q_{\text{goal}}$ along $m$-line
    until $q_{\text{goal}}$ is reached or obstacle is encountered at hit point $q^H_i$
    if Goal is reached then Exit endif
    repeat
        Follow obstacle boundary
    until $q_{\text{goal}}$ is reached or $q^H_i$ is re-encountered
        or m-line is re-encountered at a point m, such that $m \neq q^H_i$ (robot did not reach hit point),
        and $d(m, q_{\text{goal}}) < d(m, q^H_i)$ (robot is closer), and if robot moves toward goal, it would not
        hit obstacle.
    Let $q^L_{i+1} = m$, increment i
    if the robot were to move toward the goal then
```

```
            Conclude $q_{\text{goal}}$ is not reachable and exit
      endif
end while
```

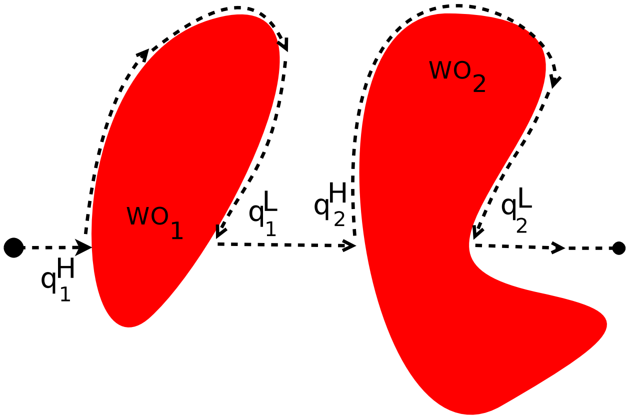


Fig. 4.11: An example of a path using the Bug 2 algorithm.

If free space between the start and goal are not path-wise connected, then we have no hope of finding a path between the two points. In other words, Bug2 will fail to find a path. This is shown in Fig. 4.12.

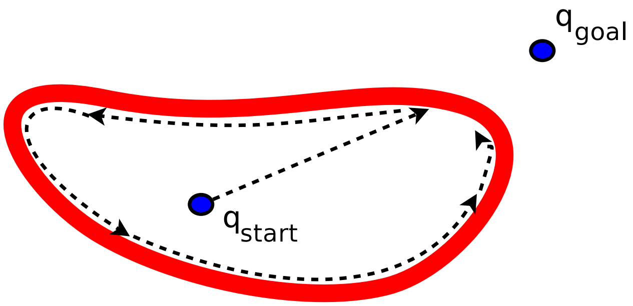


Fig. 4.12: An example of an unreachable goal for Bug 2.

From Fig. 4.11, it appears that the length of Bug 2's path would be shorter than the length of Bug 1's path. This seems obvious since we don't circumnavigate the obstacle, leaving roughly have of the obstacle's perimeter untraversed. Fig. 4.13 and Fig. 4.14 shows that Bug 1 can indeed have a shorter path than Bug 2. The basic shape is given in Fig. 4.13. The vertical obstacle can be made arbitrarily long. This means that traversing around it can have an arbrarily long path. Alternatively in Fig. 4.14, we can increase the number of vertical bars. What are the path lengths for Bug 1 and Bug 2 when they encounter Fig. 4.14?

To make the analysis easier, actual numbers are used, Fig. 4.15. The units are not really important, but included for those who like it to seem real. The path for Bug 1 is given in Fig. 4.16 and the path for Bug 2 is given in Fig. 4.17.

Following Bug 1 we accumulate the distance is 76.[2] For Bug 2, we obtain the distance is $7.5 + 26.5n$ where n is the number of vertical obstacles. The figure shows the case where $n = 6$ which provides a distance

[2] I am being a bit sloppy. I have ignored the thickness of the wall in a couple of the distance computations and so will round up here.

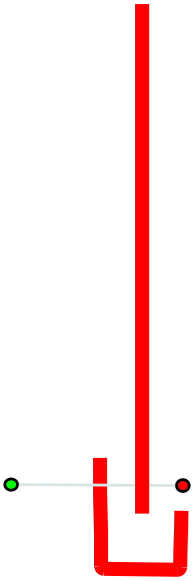


Fig. 4.13: A more disceptive obstacle. This provides the basic obstacle shape and relative pose.

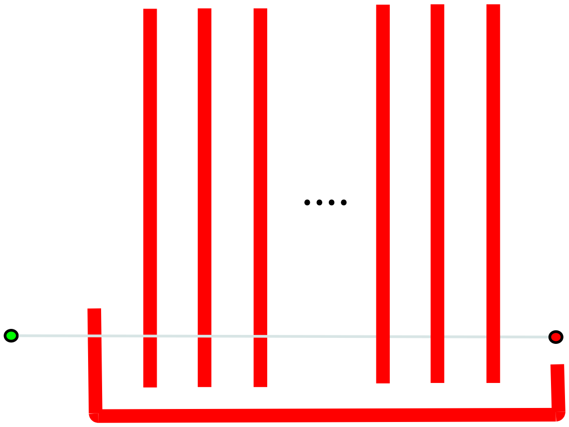


Fig. 4.14: Extending the difference in the obstacle shape to increase the path difference between Bug 1 and Bug2.

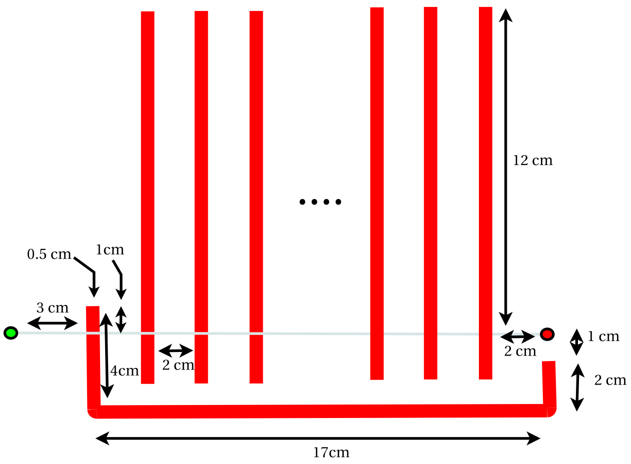


Fig. 4.15: Some dimensions for this obstacle.

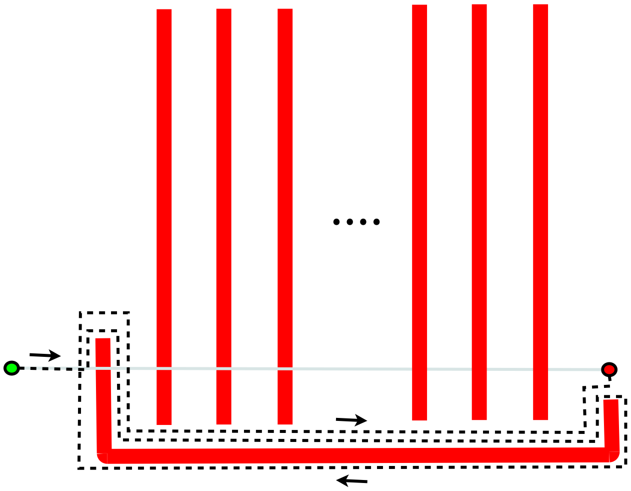


Fig. 4.16: Bug2's path.

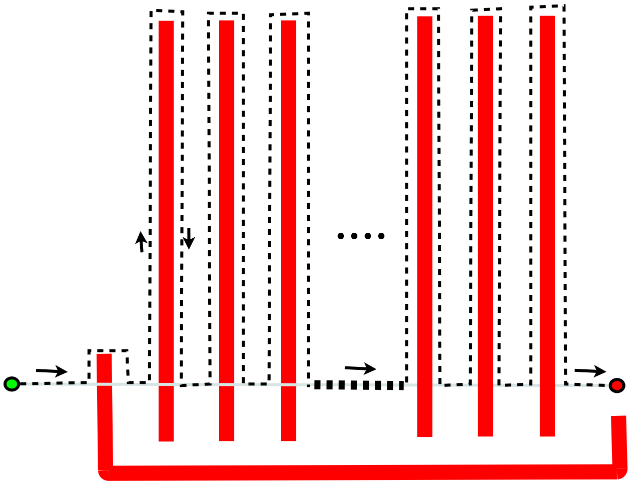


Fig. 4.17: Bug1 can outperform Bug2.

of 166 (rounding down). For the specific horizontal length of 17 cm and the current spacing used, we can replace the dots by one additional vertical obstacle, making $n = 7$. Beyond that, we need to increase the horizontal length. The horizontal length then scales roughly by $3n$ and the path then would scale by $9n$. Beyond $n = 3$, the path length for Bug 2 is larger than for Bug 1.

Lumelsky and Stepanov has illustrated is two basic approaches to searching - exhaustive and greedy. Bug 1 is an exhaustive search where Bug 2 is a greedy search. For simple domains, the greedy approach works well and thus Bug 2 is the better performer. In complicated domains, an exhaustive search may work better (not assured) and so Bug 1 may outperform Bug 2. If you look at Choset's text, you will see another example of a domain for which Bug 1 outperforms Bug 2. It is a spiral (or G shaped) domain. Although it is not hard to find domains, start and end points, which give this result; geometrically classifying them is a much more difficult problem which we leave for the reader.

There is one additional modification to the bug path that can intuitively decrease path length. The idea is that when the obstacle no longer blocks the goal during the boundary following state, leave the obstacle and head for the goal. This is shown in Fig. 4.18. This modification has the bug leave the obstacle when the obstacle becomes visible.

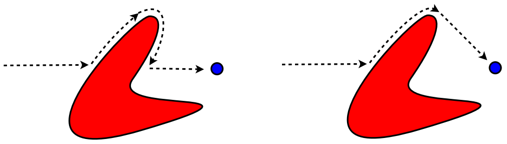


Fig. 4.18: What about reducing the path even more?

4.3.3 Bug 3

The bug 3 algorithm

```
Input A point robot with a tactile sensor
Output A path to the $q_{\text{goal}}$ or a conclusion no such path exists.
while True do
    repeat
        From $q^L_{i-1}$ move toward $q_{\text{goal}}$ along $m$-line
    until $q_{\text{goal}}$ is reached or obstacle is encountered at hit point $q^H_i$
    if Goal is reached then Exit endif
    Turn left (or right)
    repeat
        Follow obstacle boundary
    until $q_{\text{goal}}$ is reached or $q^H_i$ is re-encountered or
    tangent line at a point m points towards the goal, such that $m \neq q^H_i$ (robot did not reach hit point),
    and $d(m, q_{\text{goal}}) < d(m, q^H_i)$ (robot is closer), and if robot moves toward goal, it would not hit
    obstacle.
```

```
        Let $q^L_{i+1} = m$, increment i
        if the robot were to move toward the goal then
            Conclude $q_{goal}$ is not reachable and exit
        endif
end while
```

Bug 3 appears to effectively equivalent to Bug 2. It will suffer from many of the same types of problems as Bug 2 suffers from and get trapped in the same types of domains. The advantage often is the possible use of direct routes which can shorten travel distances.

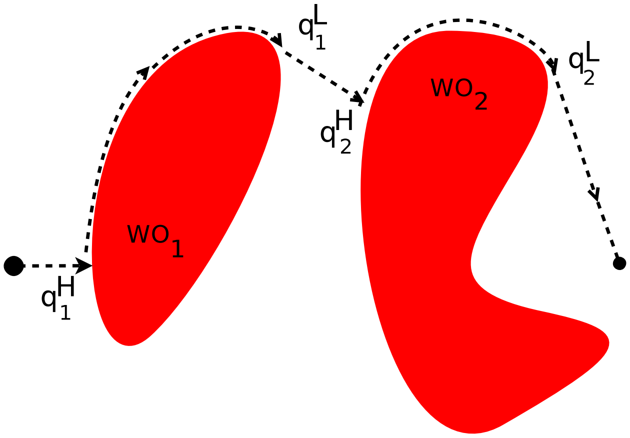


Fig. 4.19: An example of a path using the Bug 3 algorithm.

However, note that for Fig. 4.20, Bug 2 will difficulties reaching the goal where Bug 1 and 3 succeed.

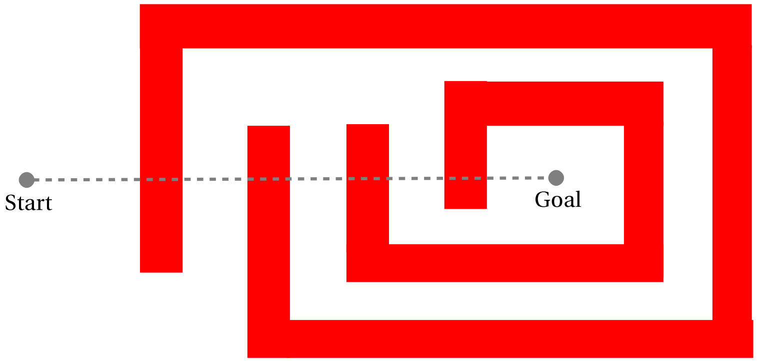


Fig. 4.20: Trace this with the different bug algorithms: bug 1 and 3 succeed and bug 2 fails.

4.3.4 Tangent Bug

The tangent bug algorithm will follow the basic idea in Bug 3, with the addition of a range sensor to the bug. As before our bug will have motion to goal in the absence of obstacles in the path. When an obstacle is encountered, the bug will switch to a boundary following mode. With a range sensor, there is more than one way to address the transition to boundary following mode which we will see. For simplicity, we assume that the range sensor has 360 degree infinite orientation resolution: $\rho : \mathbb{R}^2 \times S^1 \to \mathbb{R}$

$$\rho(x, \theta) = \min_{\lambda \in [0,\infty]} d(x, x + \lambda[\cos\theta, \sin\theta]^T), \tag{4.1}$$

such that

$$x + \lambda[\cos\theta, \sin\theta]^T \in \bigcup_i \mathcal{WO}_i \tag{4.2}$$

and a finite range:

$$\rho_R(x, \theta) = \begin{cases} \rho(x, \theta), & \text{if } \rho(x, \theta) < R \\ \infty, & \text{otherwise.} \end{cases} \tag{4.3}$$

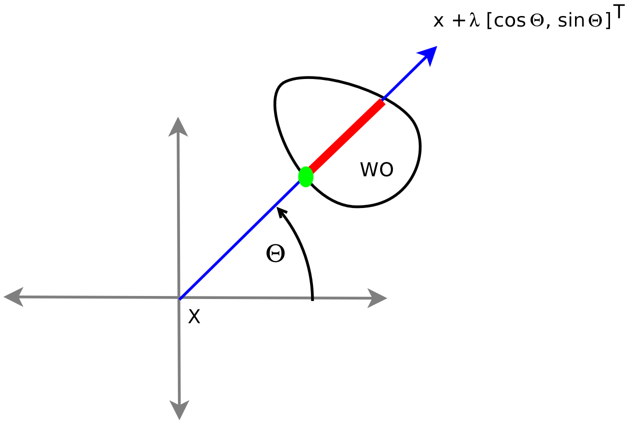


Fig. 4.21: The LIDAR ray from the location x at the angle θ.

(4.1) and (4.2) find the shortest distance between the point x and all of the points in the obstacle which intersect the ray eminating from x. A real sensor has a finite range. (4.3) truncates the result at some maximum range R.

The range sensor returns a polar map, meaning a function $\rho = \rho_R(x, \theta)$. This function will be be piecewise continuous. Discontinuities will occur by occlusion of one object by another or by reaching the maximum range, Fig. 4.22 and Fig. 4.23. Having a discrete function makes finding discontinuities a bit subtle.

Normally one uses

$$\rho_R(x, \theta_{k+1}) - \rho_R(x, \theta_k) > \delta \geq 1$$

for some δ as the criterion.

Using this idea, we obtain some number of discontinuities, call them O_1, O_2, …, O_n. It is not possible in general to tell if O_1, O_2, …, O_n indicate boundaries of separate obstacles, Fig. 4.24. Since we are only concerned about obstacles that prevent us from moving to the goal, we will only focus on those, Fig. 4.26 (left).

If the goal is obscured by an obstacle, then the robot moves towards the O_i that minimizes the heuristic distance: $d(x, O_i) + d(O_i, q_{\text{goal}})$. In Fig. 4.26, two variations are shown. The middle figure shows that

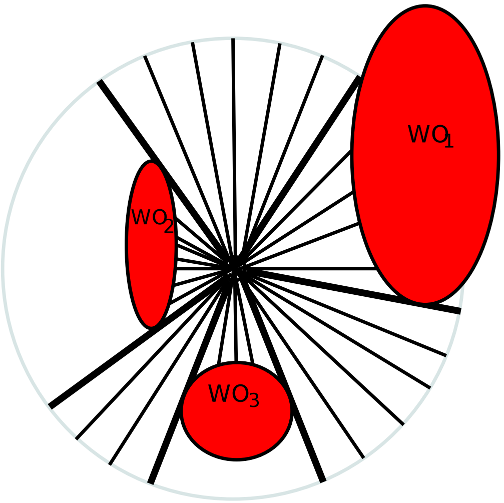


Fig. 4.22: Obstacles producing discontituities in the range map. Assume that one can determine discontinuities in the distance function ρ_R.

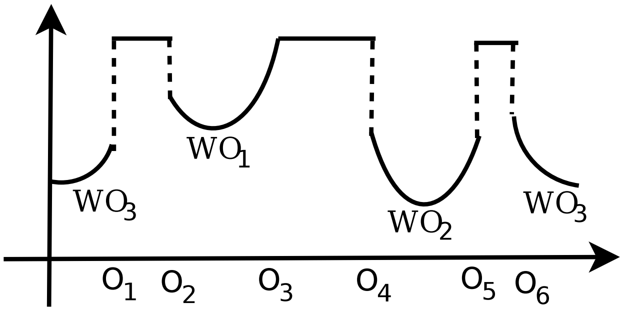


Fig. 4.23: Range map for the obstacle above.

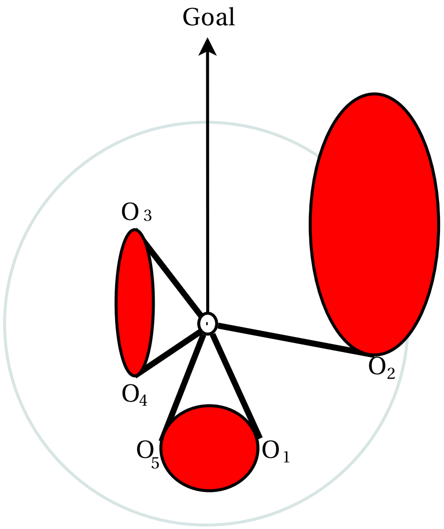


Fig. 4.24: Points of discontinuity: $O_1, O_2, \dots, O_n$

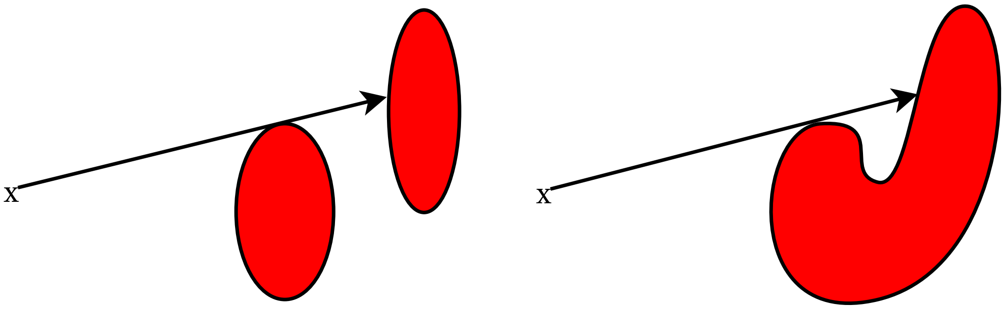


Fig. 4.25: Object ambiguity.

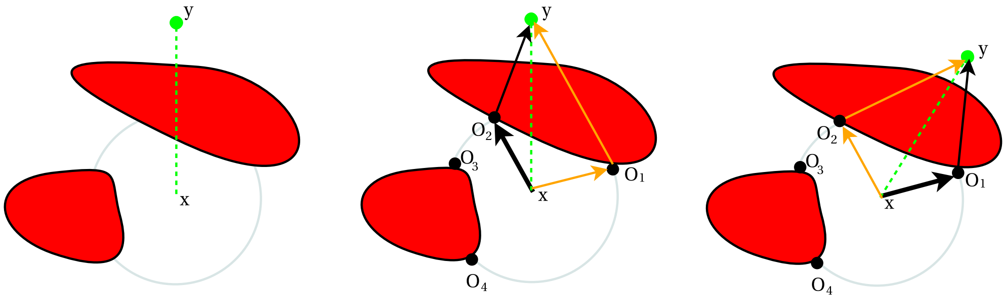


Fig. 4.26: Sensing an object does not mean it is a problem, only if it blocks the path. The robot will then move toward the discontinuity point O_i which most decreases the distance $d(x, O_i) + d(O_i, q_{\text{goal}})$

$d(x, O_2) + d(O_2, y)$ is less than $d(x, O_1) + d(O_1, y)$, so O_2 is the first target for motion. In the right figure where the goal y has moved, $d(x, O_1) + d(O_1, y)$ is less than $d(x, O_2) + d(O_2, y)$. Thus the target in that case is O_1. The points O_i are continuously updated as the robot moves. New points may enter the list and some points may leave.

We have seen two types of motion to goal. One is the free space motion where the robot moves towards the goal without an obstacle. The other is the motion towards a boundary point which is the minimizing discontinuity point discussed above. These two can be merged into just motion towards goal where goal is selected from $n = \{T, O_i\}$, $i = 1 \ldots k$ where T is defined as the intersection of the circle of radius R centered at x with the line segment from x to the goal, Fig. 4.27.

The robot will continue with the motion to goal until it can no longer decrease the heuristic distance, then it switches to boundary following. The robot follows the same direction in boundary following mode as it did in motion to goal mode. As the robot approaches the boundary, the direction will change due to pursuit of temporary goal n. The distance $d(x, n) + d(n, \text{goal})$ will start to increase. If you are far from the boundary, you are heading roughly in the direction of the goal. Once close enough and with the direction strongly affected by the obstacle boundary, it makes sense to just switch to boundary following mode. Fig. 4.28 shows the three states. The left figure indicates the robot motion to goal in free space. In the middle figure, the robot has sensed the obstacle and computed that the lower boundary discontinuity is the one to set as the temporary goal.

We define the point M which is the closest point on the sensed boundary to the goal, Fig. 4.29. This is used in the computation of the departure point.

Boundary following mode can get you around the obstacle. The next question is when to release and return to motion to goal (or to the next obstacle). We define d_{followed} as the shortest distance between boundary that

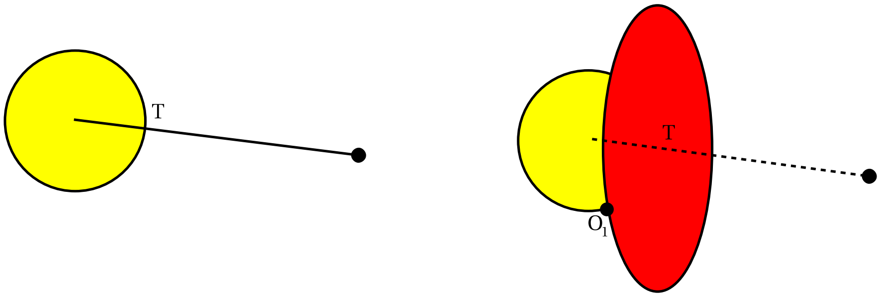


Fig. 4.27: The free space point T (left). T and O_1 (right). [defnT]

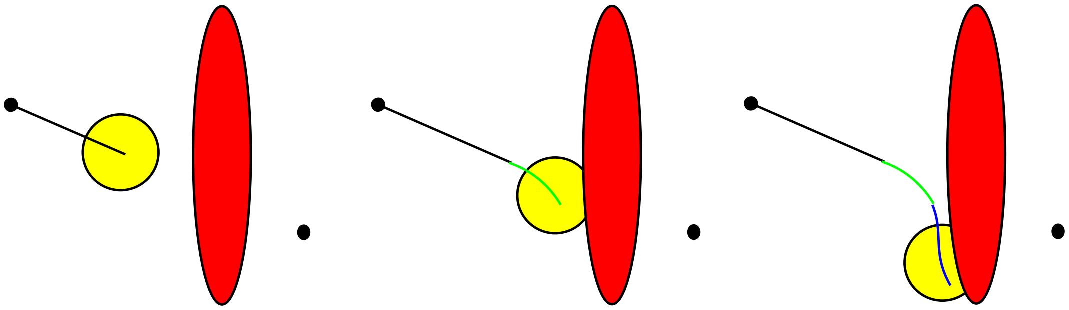


Fig. 4.28: Motion to goal (left), motion to boundary discontinuity point (middle) and boundary following (right).

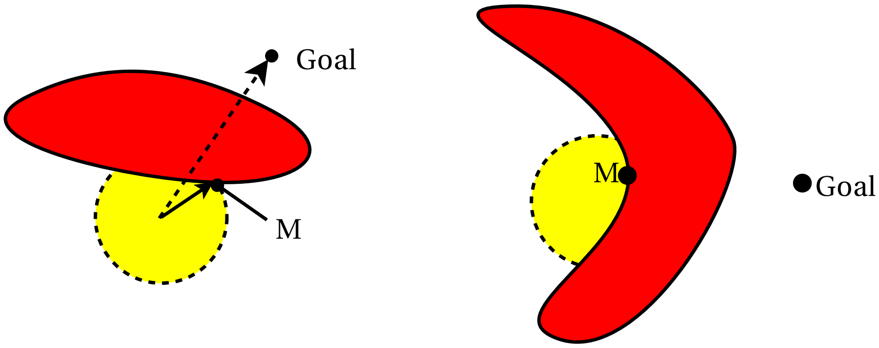


Fig. 4.29: M - the closest point on the sensed boundary to the goal. Can be one of the discontinuity points from the ranger or simply a boundary point.

has been sensed and the goal, Fig. 4.30.

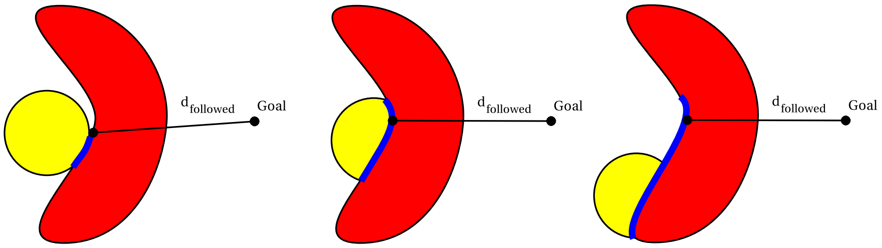


Fig. 4.30: The value d_{followed}.

Define Λ as all of the points between the robot, x and the boundary of the obstacle, ∂WO which are visible to the robot and within range R (the range of the sensor). Precisely this is $\Lambda = \{y \in \partial WO : \lambda x + (1-\lambda)y \in Q_{\text{free}} \quad \forall \lambda \in [0,1]$, Fig. 4.31. We define d_{reach} as the minimum distance point in Λ to the goal: $d_{\text{reach}} = \min_{c \in \Lambda} d(c, q_{\text{goal}})$. See Fig. 4.32, Fig. 4.33 for a description of this distance.

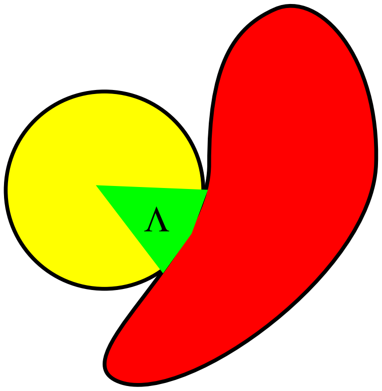


Fig. 4.31: The region Λ.

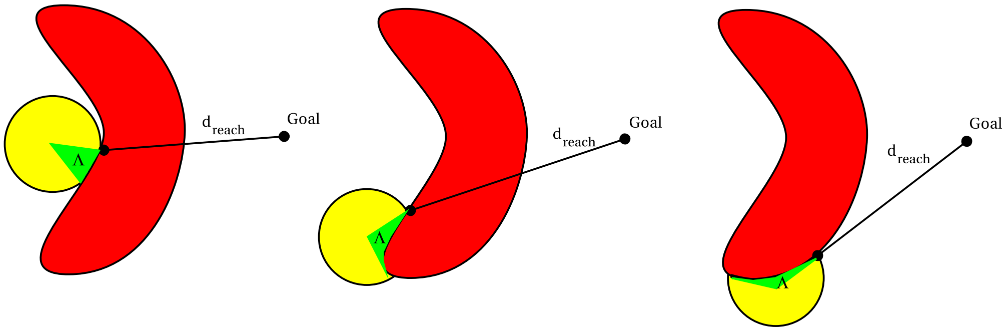


Fig. 4.32: The value d_{reach}.

These values are continuously updated as the robot traverses the boundary. When $d_{\text{reach}} < d_{\text{followed}}$ then we terminate the boundary following and return to motion to goal. Fig. 4.34 shows when the values become equal. Fig. 4.35 shows when the boundary following termination condition is satisfied. The planner is summarized in Algorithm *alg:tangentbug*.

The bug algorithms are biased towards motion along the original direct route. This last algorithm stayed in boundary following mode longer than did the Bug 3 algorithm. This behavior, however, depends on the max

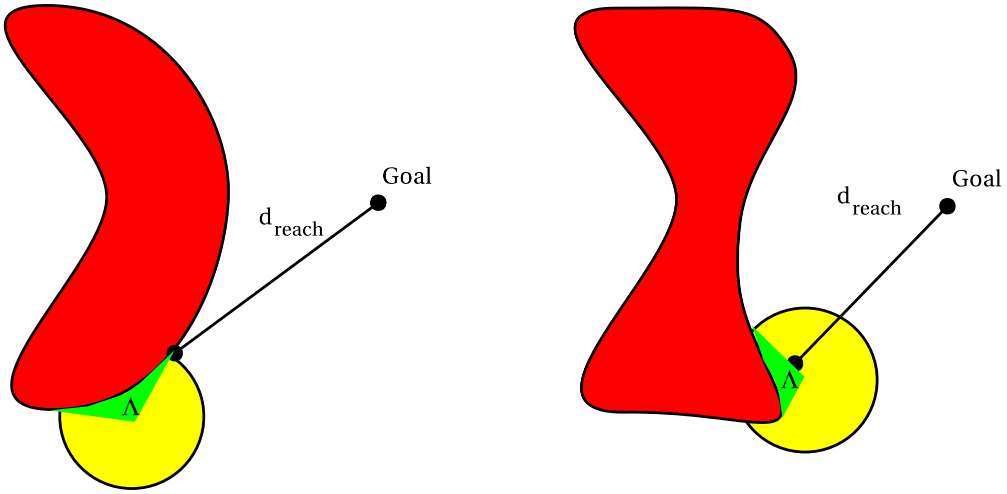


Fig. 4.33: The value d_{reach} with a different domain.

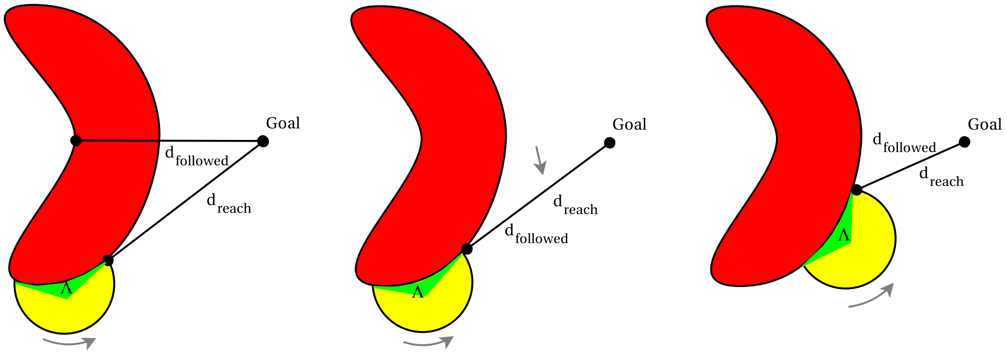


Fig. 4.34: The process and location where $d_{\text{reach}} = d_{\text{followed}}$.

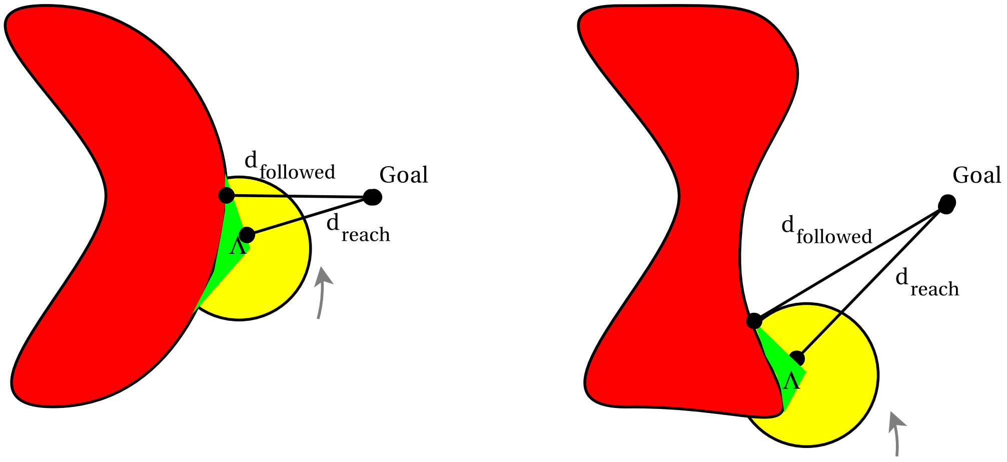


Fig. 4.35: The process and location where $d_{\text{reach}} < d_{\text{followed}}$.

range of the range sensor and is thus "tunable". An interesting experiment would modify the Tangent Bug to have the boundary exit behavior the same as Bug 3 and compare paths.

The tangent bug algorithm

```
Input A point robot with a tactile sensor
Output A path to the $q_{\text{goal}}$ or a conclusion no such path exists.
while True do
    repeat
        Continuously move from the point $n \in \{T, O_i\}$ which minimizes $d(x,n) + d(n, q_{\text{goal}})$.
    until $q_{\text{goal}}$ is reached or the direction that minimizes $d(x,n) + d(n, q_{\text{goal}})$ begins to increase $d(n, q_{\text{goal}})$
    if Goal is reached then Exit endif
    Choose a boundary following direction which continues in the same direction as the most recent motion-to-goal direction.
    repeat
        Continuously update $d_{\text{reached}}$, $d_{\text{followed}}$ and $\{O_i\}$.
        Continuously moves toward $n \in O_i$ that is in the chosen boundary direction.
    until $q_{\text{goal}}$ is reached or the robot completes a full cycle around the obstacle or $d_{\text{reached}} < d_{\text{followed}}$.
    if the robot were to move toward the goal then
        Conclude $q_{\text{goal}}$ is not reachable and exit
    endif
end while
```

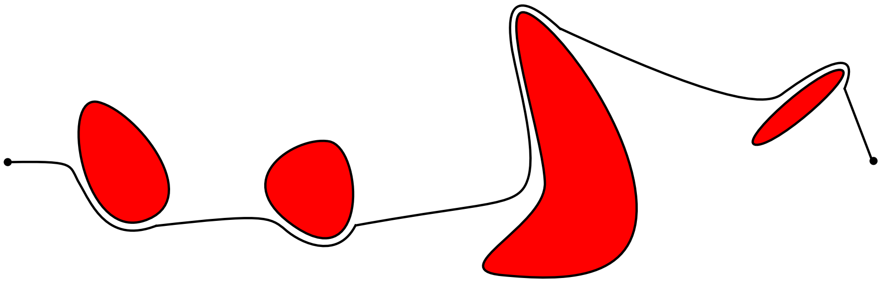


Fig. 4.36: Finite Sensor Range

4.3.5 Bug Comparison

The best paths we have seen from the bug algorithms have been the Tangent Bug paths with infinite sensor range. A sufficiently large sensor range would effectively be an infinite range sensor, so we just assume we have infinite range. We can reexamine the complicated obstacle with the tangent bug.

From Fig. 4.38, we see that Tangent Bug performs well on the obstacle field that caused so much headache for Bug 2. Fig. 4.39 (left) shows a obstacle domain for which the path for Bug 1 and Bug2 are equivalent

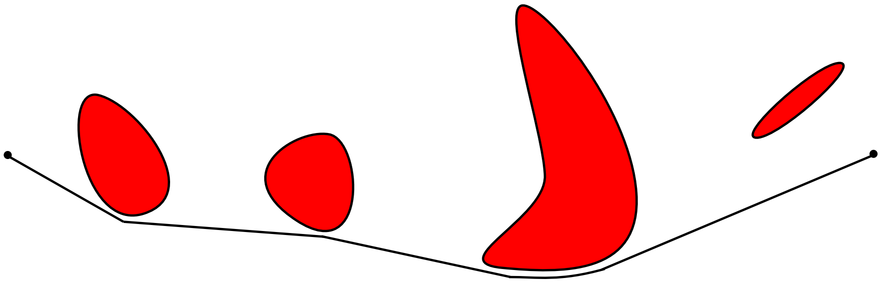


Fig. 4.37: Infinite Sensor Range.

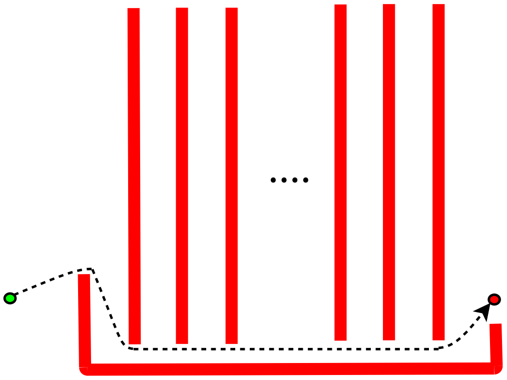


Fig. 4.38: The path of the tangent bug on the difficult obstacle field.

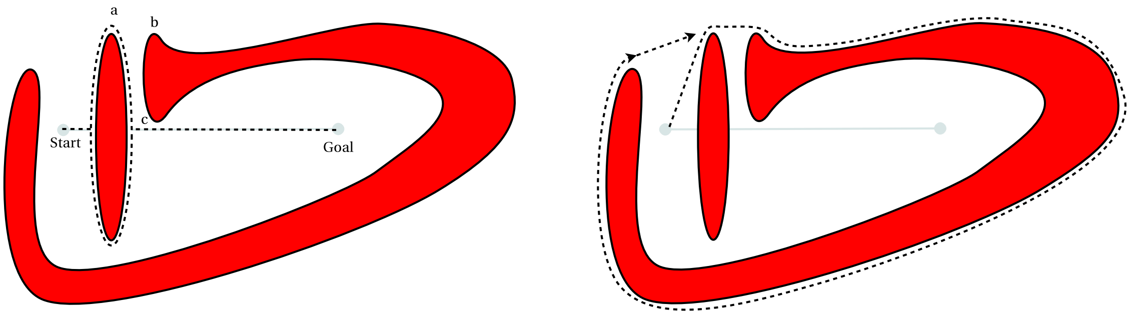


Fig. 4.39: (left) Bug 1 and Bug 2 suceed. (right) Tangent Bug does not.

and arrive at the goal. The bugs begin at the start position and head to goal. Upon arrival they turn left and head up over point a. Heading down the back side of the ellipse, Bug 2 will split off when it crosses the M-line. Bug 2 will then head straight for the goal. Bug 1 will continue to circumnavigate the ellipse. After return to the M-line it too will head to the goal. By construction Bug 1's leave point is the M-line as well. Both arrive at the goal.

The right figure shows how the Tangent Bug does not arrive at the goal and cycles around the outside. [Had the left side of the figure dropped lower, this would have been an example of a longer path, but the Tangent Bug would have arrvied at the goal.] In this case, T-Bug leaves the start location and heads towards the goal. Although rather subtle, the line from the start to goal is slight above the vertical symmetry axis. This means that the top of the ellipse, location a, will be the closest point of discontinuity for the ranger. Thus it will minimize the heuristic traveling to a. The points b and c are the next two discontinuities to choose. By construction, the point b minimizes the heuristic over the location c. After arriving at c, the robot will shortly transition to boundary following mode. This will carry the robot around the obstacle back to a location above the starting point. The robot will head to the discontinuity a. Any implementation that stores locations will note that we have done a cycle and exit or needs to switch algorithms.

4.4 Bug simulations

Even though the bug algorithms are intended to illustrate particular ideas, they are valid algorithms for path planning. In addition, roboticists get some odd sense of pleasure out of watching planners succeed. We will restrict our work here to discrete (bitmap) domains only because it makes the headache of determining collision go away. A collision requires the robot to move. How is this done?

There are two ways initially you can approach controlling the robot: through position or velocity. A position control sets the location directly. The other method, velocity, control adjusts the velocities and the position is updated indirectly. Position control is easy to start with when you directly work with the simulation window or canvas. It is not really how a robot is controlled since it models stop and go motion. Velocity control follows how most mobile robots are actually controlled. The movement must be consistent with vehicle kinematics which can vary depending on robot design. For example a differential drive robot can set forward velocity and rotational (turn) velocity. Using the kinematic equations (presented later), position and orientation can be computed.

Earlier we discussed how to determine obstacle impact or collision for a circular robot. For a general robot shape, impact is more difficult to compute. Collision can be defined as when the distance between the robot and obstacle becomes zero. In the continuous world, the distance is

$$D(t) = \min \sqrt{(x_R(t) - x_O)^2 + (y_R(t) - y_O)^2}$$

where (x_R, y_R) lies on the robot boundary and (x_O, y_O) lies on the object boundary. Setting $D = 0$ and solving for t then provides the time and location of the collision. However, the robot and the obstacle will have some shape defined by a bitmap. Collision between the robot and the obstacle boils down to determining if a pixel or pixels will overlap after a robot position update. You might notice that this is not quite correct. It is possible for the time step to be large enough that the robot appears to jump over obstacles. In that case, we need to select a smaller time update so that motion begins to appear continuous and teleporting robots are avoided.

Assume that you have n objects in the landscape which have a general discrete (bitmap) shape. This means each obstacle is a union of squares. After the motion step, one needs to check if any pixel occupied by the robot is also occupied by an obstacle. After the move, all of the pixels associated with the robot location are checked. If any pixel is painted black, then the robot has driven into an obstacle. This should generate an event associated with a crash and signal both the graphics window as well as the control program.

This same approach may be used to simulate a bump or touch sensor. The boundary pixels of the robot can be used as the sensors. If those pixels are adjacent to an object, then a touch is registered. This can be done via a direct adjacency test - looking at neighbor pixels. Or this can be done by a ghost pixel method. Inflate the objects by one row of pixels. If the robot overlaps a ghost pixel then the corresponding robot pixel/sensor would register a touch. Inflation is essentially one step of a flood fill. Flood fill is normally implemented as a type of DFS algorithm, but does not need this machinery here. A common error here is to write the adjacent pixel into the current array, but if you are sweeping through the image row by row, you might use this new pixel as an obstacle pixel. Then this new sweep will mark the neighbor. This can continue and cause lines to grow down with the sweep.

An array used for the image data which stores multiple values can be very handy. Pixels are marked according to type. For example, zero for open pixel, one for a robot occupied pixel, two for an obstacle, three for the inflated set. Later on having data stored per pixel (or cell) is essential for our motion planners.

4.4.1 Getting a Map

There are a variety of approaches for getting map data created. Various range sensors and cameras have been used to create viable maps. Pixel based occupancy grids are relatively easy to create. For mazes, we might only be interested in the topology of occupied or free space. A skeletonization of free space returns a graph that can be used for graph based search algorithms.

4.4.2 Implementation

The algorithms presented above have two basic modes. One is motion to goal. This behavior assumes that the robot knows the target location or at least knows the direction to head. This is done in practice using a type of localization system. In a simulated environment, it is of course very easy since you always have absolute knowledge of the robot and goal's location. The more challenging problem is boundary following. Unless you have very accurate maps to start with or apriori knowledge of the objects in the environment, the boundaries of the obstacles are unknown. This means they must be discovered during the planning process. How does the robot move around the boundary? What information is required? What information is provided by the sensors and so what information needs to be computed? How is the path determined?

We will assume that object boundaries are smooth curves and would be locally a function, $y - f(x)$. If this is the case, we can compute the tangent and normal directions as shown in Fig. 4.40. An offset curve is a curve that follows the boundary at some fixed distance from the boundary. It looks like a level set curve. We can compute the tangents and normals for offsets as well, Fig. 4.41.

An offset curve can be found analytically using only the Tangent direction vector $v(t)$ [where v is a basis vector in $(n(c(t)))^{\perp}$]. Assume that the curve is given in parametric form $\{c_1(t), c_2(t)\}$. Solving the differential equations $\dot{c}(t) = v$, $\{c_1(0), c_2(0)\} = c_0$ provides the offset curve.

Example: If the tangent to an offset curve is $v =< -y, 2x >$, find the offset curve $\dot{c}(t) = v$ when $c_0 = (1, 2)$.

$$\dot{c}(t) = dc/dt =< dx/dt, dy/dt >=< -y, 2x >$$

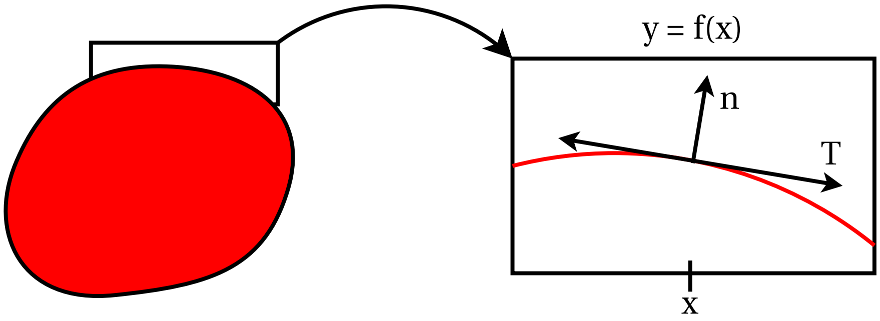


Fig. 4.40: We assume that the boundary is a smooth function.

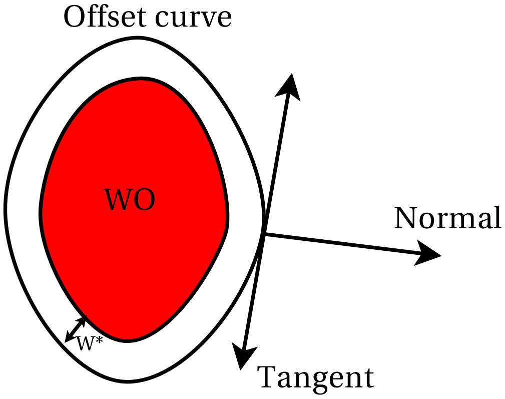


Fig. 4.41: The normal and tangent directions to the offset curve.

so (1) $dx/dt = -y$ and (2) $dy/dt = 2x$. Differentiate the first equation to get $d^2x/dt^2 = -dy/dt$ and then plug into the second equation: $d^2x/dt^2 = -2x$. We can solve this equation to obtain

$$x(t) = A\cos\sqrt{2}t + B\sin\sqrt{2}t.$$

The condition $x(0) = 1$ means $x(0) = A = 1$. From the first equation we obtain

$$y(t) = \sqrt{2}\sin\sqrt{2}t - B\sqrt{2}\cos\sqrt{2}t$$

Using the second condition, $y(0) = 2$, we see that $B = -\sqrt{2}$.

We have already discussed computing an obstacle boundary normal and tangent, Fig. 3.32, using a ring of touch sensors. In a real application, you may stop once the tangent has been determined. The robot can be steered in that direction. The act of driving the robot continuously in the direction of v is the same as solving the differential equations (other than the different errors that arise).

Simple boundary following using a range sensor

If a range sensor is available, it is a better choice for determining the boundary normal (avoids contact with the obstacle). Assume that you are looking to follow the boundary of obstacle 2 in Fig. 4.42. Let $D(x)$ be the distance from x to the followed obstacle:

$$D(x) = \min_{c\in\mathcal{WO}_i} d(x, c)$$

Look for global minimum to find the point on the followed obstacle. The gradient of distance is given by

$$\nabla D(x) = \begin{bmatrix} \dfrac{\partial D(x)}{\partial x_1} \\ \dfrac{\partial D(x)}{\partial x_2} \end{bmatrix}$$

The closest point by definition is the point that is a minimum of the distance function between the ranging device, x, and the obstacle boundary, y. This means that the tangent must be orthogonal to the line segment connecting x and y. Once the direction to y is determined then the travel direction can be computed. Assume the direction to y is given by $\nabla D(x) =< a_1, a_2 >$. The travel direction is $\pm < a_2, -a_1 >$ which is orthogonal to ∇D.

A ranging device in practice returns discrete data. You can detect the approximate nearest point on the obstacle boundary, say at index k in the range array data: d[]. You can convert (k-1, d[k-1]),(k, d[k]),(k+1, d[k+1]) into (x,y) points in the robots coordinates: (x_{k-1}, y_{k-1}), (x_k, y_k), (x_{k+1}, y_{k+1}):

$$(x_k, y_k) = (d[k]\cos(\Delta\theta k + \theta_0), d[k]\sin(\Delta\theta k + \theta_0))$$

where θ_0 is the angle for the start of the sweep. Knowing the closest point on the boundary to the robot is again sufficient to compute the tangent direction. We can smooth out the boundary motion using algorithm Boundary Motion

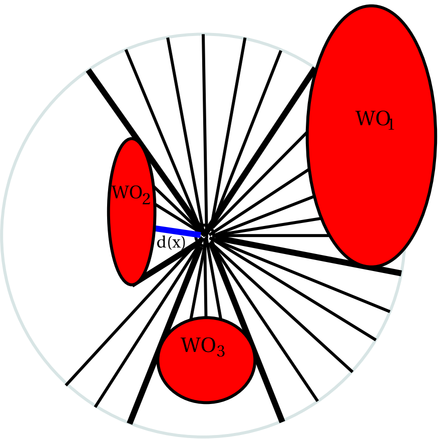


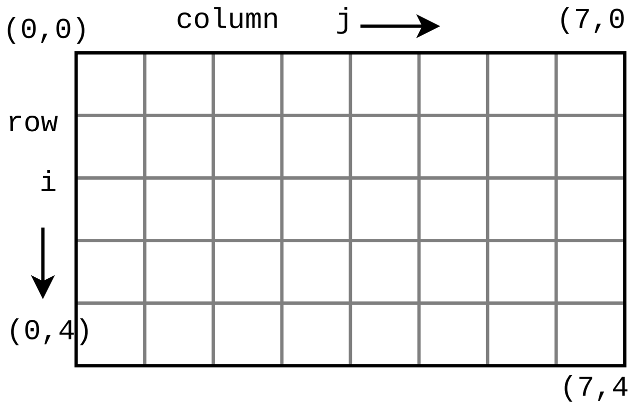
Fig. 4.42: Obtaining information from range data.

Boundary Motion

```
List all neighbor cells adjacent to occupied cells.
Select neighbor according to policy (right or left hand travel): (m,n).
Mark (i,j) as visited.
Set current cell: (m,n) $\rightarrow$ (i,j).
while Not arrived at leave point do
    repeat
        List unvisited neighbor cells adjacent to occupied cells.
        Select neighbor: {tt (m,n)}
        Mark {tt (i,j)} as visited.
        Set current cell: {tt (m,n)} $to$ {tt (i,j)}.
    end while
```

Image coordinates and coordinates

Images are simply two dimensional arrays of integers. Much like matrices in your math courses. There are a couple of differences you need to know. First, the coordinate system for an image has the y coordinate increasing as you head down. Second, the origin is the top left pixel. The graphic below indicates this coordinate system. The way we store two dimensional arrays is Array[row][col]. Increasing row will increase in the y direction downwards. So the two dimensional array is consistent with the image coordinate system. We will call neighbor pixels the eight pixels surrounding the center pixel. The graphic below shows the standard mathematical notation for plots and graphs (x, y) and the array notation for the pixels.



(x-1,y+1) A[i-1][j-1]	(x,y+1) A[i-1][j]	(x+1,y+1) A[i-1][j+1]
(x-1,y) A[i][j-1]	(x,y) A[i][j]	(x+1,y) A[i][j+1]
(x-1,y-1) A[i+1][j-1]	(x,y-1) A[i+1][j]	(x+1,y-1) A[i+1][j+1]

4.4.3 C++ easy access of neighbor pixels

Assume that you have an image stored in the two dimensional array map. Many algorithms require you to access all eight neighbor pixels. You can write these out by hand or you can do a two dimensional loop. The following code accesses all eight neighbors and the point itself (nine pixels):

```
for(i=-1;i<=1;i++) {
     for(j=-1;j<=1;j++) {
           value = map[row+i][col+j];
      }
}
```

Most likely your implementation will not care about the center point. But if you explicitly want to skip it, try adding a conditional[1]. Be very careful about stepping outside map array bounds. Either you need to check for stepping outside the array or your loops need to stay inside the array. For example, instead of running $i = 0$ to $i = n$ you run from $i = 1$ to $i = n - 1$. The outer layer of pixels are the “walls” and you don’t touch them. This is why we suggest having a layer or two of black pixels around the map.

4.4.4 Impacts in grid environments

The last issue that needs to be addressed is object interaction. How should we handle an impact? In Fig. 2.27, we saw that for circular robots, we could just add the radius of the robot to the obstacle and then treat the robot as a point mass. For path planning of circular robots we can then inflate the obstacle using a truncated flood fill algorithm and proceed with path planning using just a point as the robot. The flood fill algorithm will be discussed later on in this chapter. We can then assume that all of the obstacle maps have been preprocessed and just focus on the planning aspect.

Detecting a collision is now very easy. The robot is a point and so impact is determined if the point is adjacent to an obstacle. Assume that the robot is at location and the obstacle map is obstMap[i,j]. Also assume that empty space is represented by 0 in the array, filled space is represented by 1, and the boolean variable impact records impact or not.

```
if (obstMap[i+1,j] == 1) or (obstMap[i-1,j] == 1) or \
        (obstMap[i,j+1]== 1)  or (obstMap[i,j-1]== 1) :
    impact = 1
```

This will work to determine if is adjacent to a filled pixel. The problem that arises is with the array boundaries. For example if i = 0 then the comparison obstMap[i-1,j] == 1 falls out of the array bounds. The literature has two standard approaches for this issue. One way to proceed is to treat the four sides and four corners of the array as special cases with code lines of the form if i == 0 then omit the left neighbor check. One must do this for top and bottom, left and right boundaries.

The second common approach in the literature is known as ghost points. The idea is to inflate the array by one pixel on each boundary. Say that the obstacle map is 800 wide x 600 high. Normally your array runs i = 0..799 and j = 0..599. Declare the storage array to be 802 x 602. Then place the obstacle map in i = 1..800 x j = 1..600. We define an open landscape as no solide boundary on the edges of the obstacle map (meaning no walls around the region). A closed landscape will have walls. For an open landscape set the arrray entries for , [i = 801,j] , [i, j=0] , [i,j = 601] equal to zero. For a closed landscape set those values to 1.

[1] `if((i!=0)||(j!=0))`

The boundaries no longer generate out of array errors and the need for special boundary cases is eliminated. The code above will work as is.

The simplest approach is to flood fill about the obstacle the full radius plus one. This means that when the center of the robot overlaps the obstacle on the configuration space, the physical robot is adjacent in the physical workspace. It neither requires a list of comparisons or an inflated array. In this case the code is very simple:

```
if (obstMap[i,j] == 1):
    impact = 1
```

It is now time to put everything together. We first list the server code example. As above, a few lines have a backslash continuation character which are for typesetting here and are not needed in the code. For simplicity the obstacle map will use 0 for occupied and not 0 for open. These just follows the image where black is 0 and which is 255. The code first sets up the Turtle canvas. It places the robot at (-300,0) and selects not to draw the path. Then we take a break and setup the sockets. The program will block until a socket is established (recall the discussion on event loops). Finally the progam enters the turtle loop. It reads a comment on the socket and then issues that command to the Turtle. The client program discussed above is used to communicate with the turtle server.

The impact aspect is not really robust. The focus is on planning, not on physics. We make no attempt to stop the robot and allow it to pass through walls. It is the responsibility of the planner to stop, backup, turn and move around.

4.5 Mazes

A very common demonstration of mobile robot problem solving is a maze escape. Even though a maze escape routine is standard fair for a data structures course, it is still impressive to see it implemented with a robot in a maze.

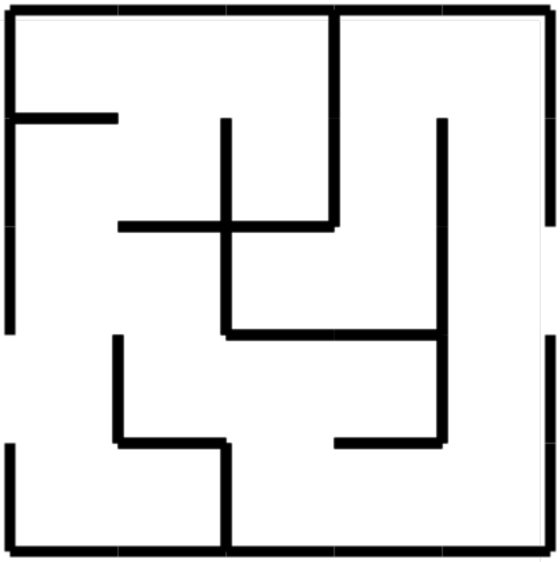


Fig. 4.43: A very simple maze.

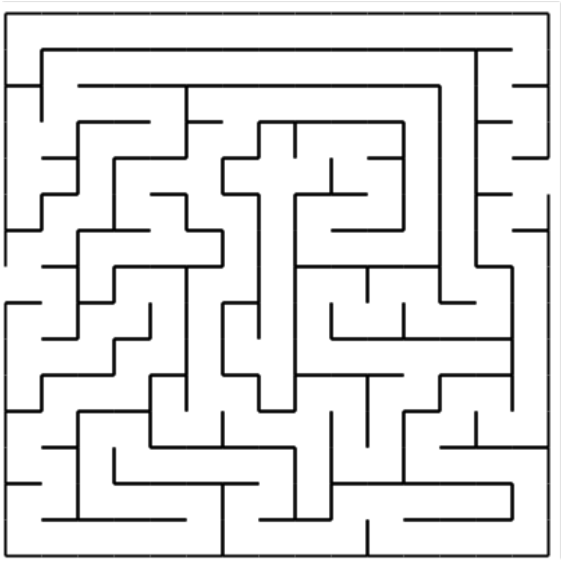


Fig. 4.44: A more complicated maze.

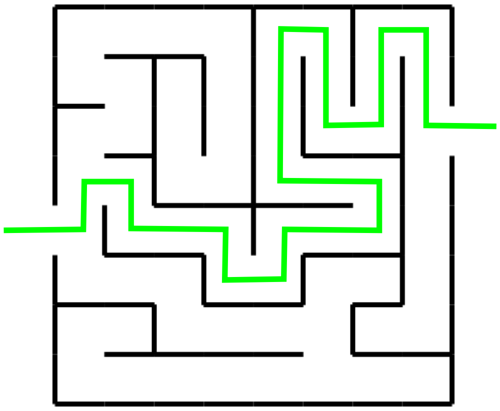


Fig. 4.45: Solution path through a maze.

Fig. 4.45 shows a solution path through a maze. The random mouse algorithm is one approach to finding a route. The algorithm has the "mouse" travel straight until a wall is encountered. Then the "mouse" randomly selects a new direction to follow. This approach is a form of random search which eventually finds a route, although rather slowly.

4.5.1 Wall Following

The best known method to traverse a maze is the wall following method. The idea is to place your left or right hand on the wall as you traverse the maze. If the maze is simply connected, the method is proven to provide a path out of the maze. By looking at Fig. 4.45, the solution path partitions the maze. A simply connected maze is partitioned into two objects which are deformable to a disk. To see this, focus on the right (or in the figure the lower) part of the separated maze. Tracing the path, Fig. 4.47, we record our motion through the maze. This path can be extracted, Fig. 4.49 to see that it is indeed a circle. The topology as not changed.

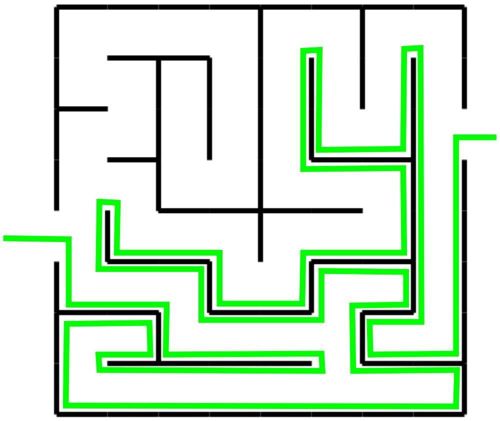


Fig. 4.46: Wall following (right hand) to solve the maze.

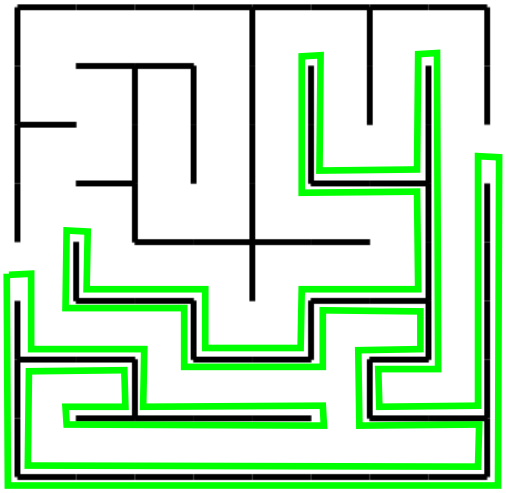


Fig. 4.47: Connecting the outside to make a circle.

Since the path is a circle, then the algorithm will transport the robot between any two points on the circle.

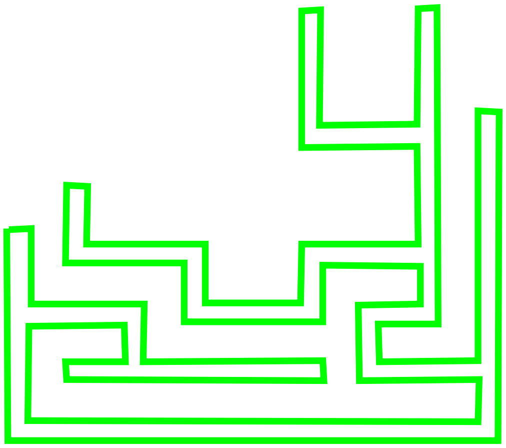


Fig. 4.48: Wall path extracted from the maze.

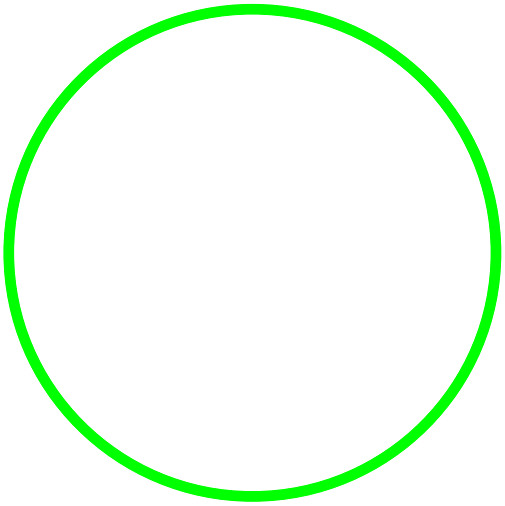


Fig. 4.49: Moving the nodes on the path to show the circle.

Not having a simply connected maze or having interior starting/finishing points can break this method - which does not mean it will necessarily fail.

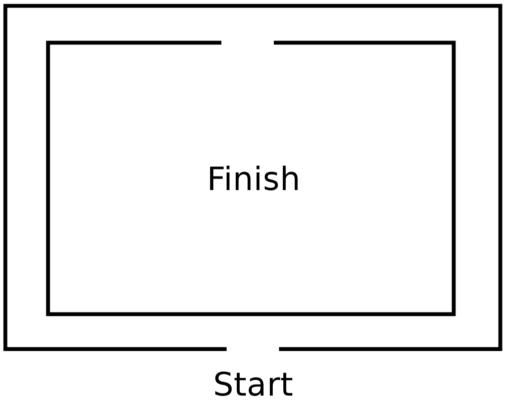


Fig. 4.50: A maze for which wall following can fail.

The Pledge algorithm is designed to address the problem of exiting a maze which has non-simply connected components. This algorithm does not work in reverse, meaning that it can escape a maze, but not enter one.

Pledge Algorithm

```
Input A point robot with a tactile sensor
Output A path to the $q_{\text{goal}}$ or a conclusion no such path exists.
Set arbitrary heading.
while No obstacle in front do
    repeat
        Move forward
    end while
Select right or left side and place that side against the obstacle.
while Note original heading and sum of turns not zero do
    repeat
        Move along obstacle while keeping "hand" on obstacle
        Sum turn angles
end while
```

The final escape algorithm presented here is Trémaux's Algorithm. This is a form of a recursive backtracker. From Wikipedia:

> Trémaux's algorithm, invented by Charles Pierre Trémaux, is an efficient method to find the way out of a maze that requires drawing lines on the floor to mark a path, and is guaranteed to work for all mazes that have well-defined passages. A path is either unvisited, marked once or marked twice. Every time a direction is chosen it is marked by drawing a line on the floor (from junction to junction). In the beginning a random direction is chosen (if there is more than one). On arriving at a junction that has not been visited before (no other marks), pick a random

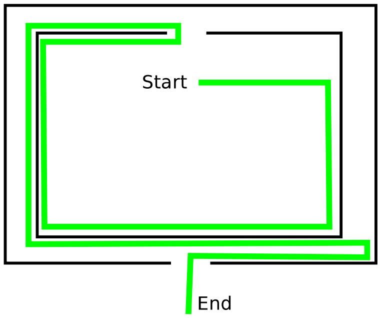


Fig. 4.51: The Pledge Algorithm.

> direction (and mark the path). When arriving at a marked junction and if your current path is marked only once then turn around and walk back (and mark the path a second time). If this is not the case, pick the direction with the fewest marks (and mark it, as always). When you finally reach the solution, paths marked exactly once will indicate a direct way back to the start. If there is no exit, this method will take you back to the start where all paths are marked twice. In this case each path is walked down exactly twice, once in each direction. The resulting walk is called a bidirectional double-tracing.

In most maze solving applications, the maze is represented by a graph. If you have seen some basic graph search algorithms you will recognize this as a type of Depth First Search (DFS). For the robot however, there is more than the DFS maze solving code. There is also the details of navigating corridors and turns. Using only bump sensors this can be a challenge, one we will address with ranging sensors later in this chapter. However, without good sensors, using the algorithms like Trémaux's algorithm might not work out. Without the ability to drop and sense breadcrumbs, the recursive backtracker will fail. One way to approach this problem is to create a map of the maze as you work your way through it. Acting on the map means you are working on existing trails and this is just another way of marking the domain.

The robot is running on a more complicated lanscape than the just operating in the maze. Working on a solution to the maze in the Pledge Algorithm or Trémaux's algorithm is simply working along the abstracted paths. We are neglecting all the issues relevant to a robot such as driving straight down the corridor, detecting walls, keeping distance from walls, navigating turns, etc. All of this low level navigation is ignored in the maze algorithms above and they focus on the higher level aspect of maze escape. This makes sense in that separating the levels helps to separate tasks leading to better code design.

To reduce the complexity we separate the maps for the robot, the landscape map, which will have a precision set by the sensors and the map or graph required by the maze, maze map. The maze map can use a grid with larger cells. Large cells would mean lower precision but smaller arrays. However, this is not a problem since the low level routines are doing the positioning on the high resolution map leaving the high level routines to navigate.

The maze map can be thought of as a low resolution version of the landscape map. Each cell can still be an occupancy map, but with large cells. In this case it is useful to take the cell as large as possible so that corridors or walls are one cell wide. Using the centers of unoccupied cells, these are nodes. Adjacent free cells can have their center nodes connected. This builds a graph representation, see Fig. 4.52. So, now we have a high resolution grid map and the corresponding graph representation of free space. This concept will be used later in more advanced path planniing algorithms. For now we employ a simple path planner.

One of the simpliest planners is the flood fill approach. Begin at the endpoint and run a flood fill algorithm.

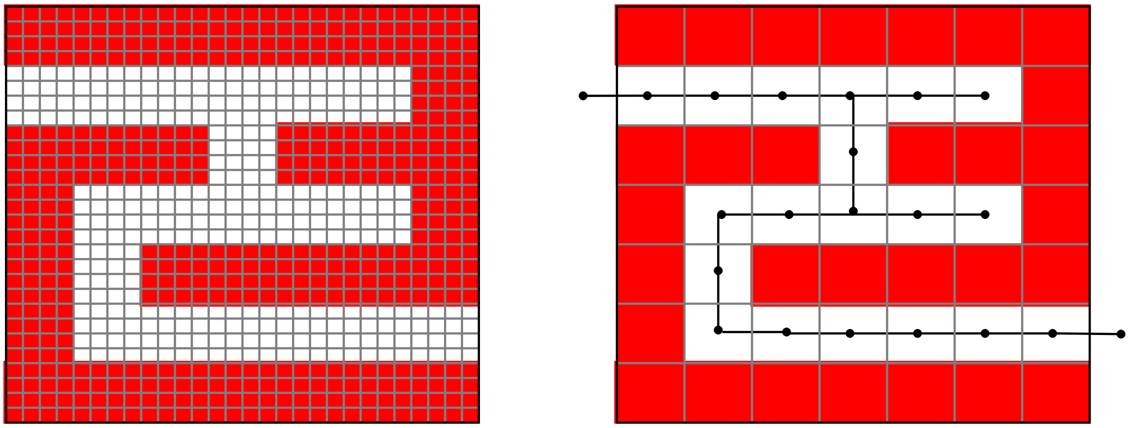


Fig. 4.52: The coarsening of the grid map for a maze and the construction of the graph representation. Left side image is a maze on a finer grid. The right side image is a coarser grid with graph drawn.

If the flood fill paints the starting point then a path has been discovered. You can run the flood fill algorithm on the landscape map, the reduced maze map or the maze graph. For illustration, we focus on the second one.

There is a fundamental difference between exploring the domain and a route, and having a map available to discover a route. If the entire domain is known and the question is simply to find the route, there are routing tools available. The route can be found before exploration. We will see later that flood fill approaches can help even in partially explored (or mapped) domains.

The maze is a high regular and artificial structure. We don't have anything like them in nature and few things in our day to day surroundings really resemble a maze. So, why discuss them? The maze has setup some fundamental approaches which we will employ next. First, we see that it makes sense to approach an obstacle, like a wall, and then follow the obstacle. This is the "place a hand on the wall" idea. We see that that approach is not sufficient from more complicated mazes and we also need to know when and where to break free of the obstacle. We have learned that seeing the domain in terms of a graph is useful in that we can apply algorithms designed for graphs, such as a depth first search. We see that certain solutions are comprehensive in how they solve the problem and others are not. The maze is then the launching point for planners which live in unstructured worlds.

4.6 Wave-front Planner

In this section we introduce the Wavefront planner. This planner is a breadth first search algorithm applied to the grid map domain. The implementation is similar to a flood fill algorithm. The Wavefront Algorithm searches for the minimal path from start to goal in structured and unstructured domains, Fig. 4.53. Just like a flood fill, Wavefront is rather simple. Assume that free space is represented by white and occupied space is red or black (colored). Zoom in so you can see the actual pixels as shown in Fig. 4.53.

The process to find the path through the maze is simple. It is completed in two stages. Stage one fills the map with distance numbers from the goal. Stage two steps down the distances until the goal is reached. Tracking the steps generates the path. So, we have two parts. First is an algorithm called "Fill" which is like a flood fill in your paint program. The second part is the "Descent" algorithm. Think of the fill algorithm as building a hill where the start is at the top and the end is at the bottom. All we do is walk downhill.

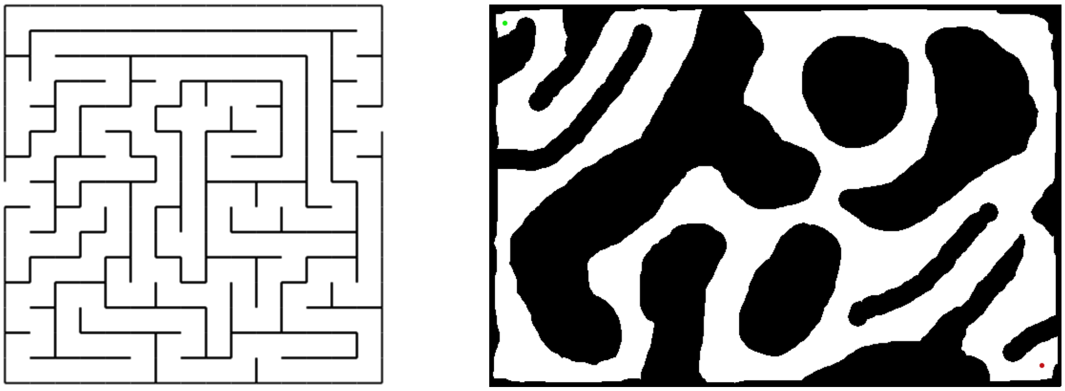


Fig. 4.53: Wavefront will apply to maze and unstructured domains.

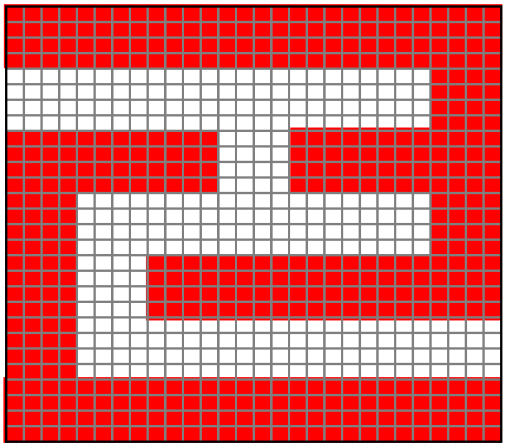


Fig. 4.54: Initial Maze.

The Fill algorithm is easy to state. Label the goal pixel “1”. Next, label all unlabeled neighbors of the “1” pixel the number “2”. Then label all of the unlabeled neighbors of the “2” pixel the number “3”. You repeat this process by labeling all of the unlabeled neighbors of the pixels with the label “k” the number “k+1”. Do this until you run out of unlabeled pixels.

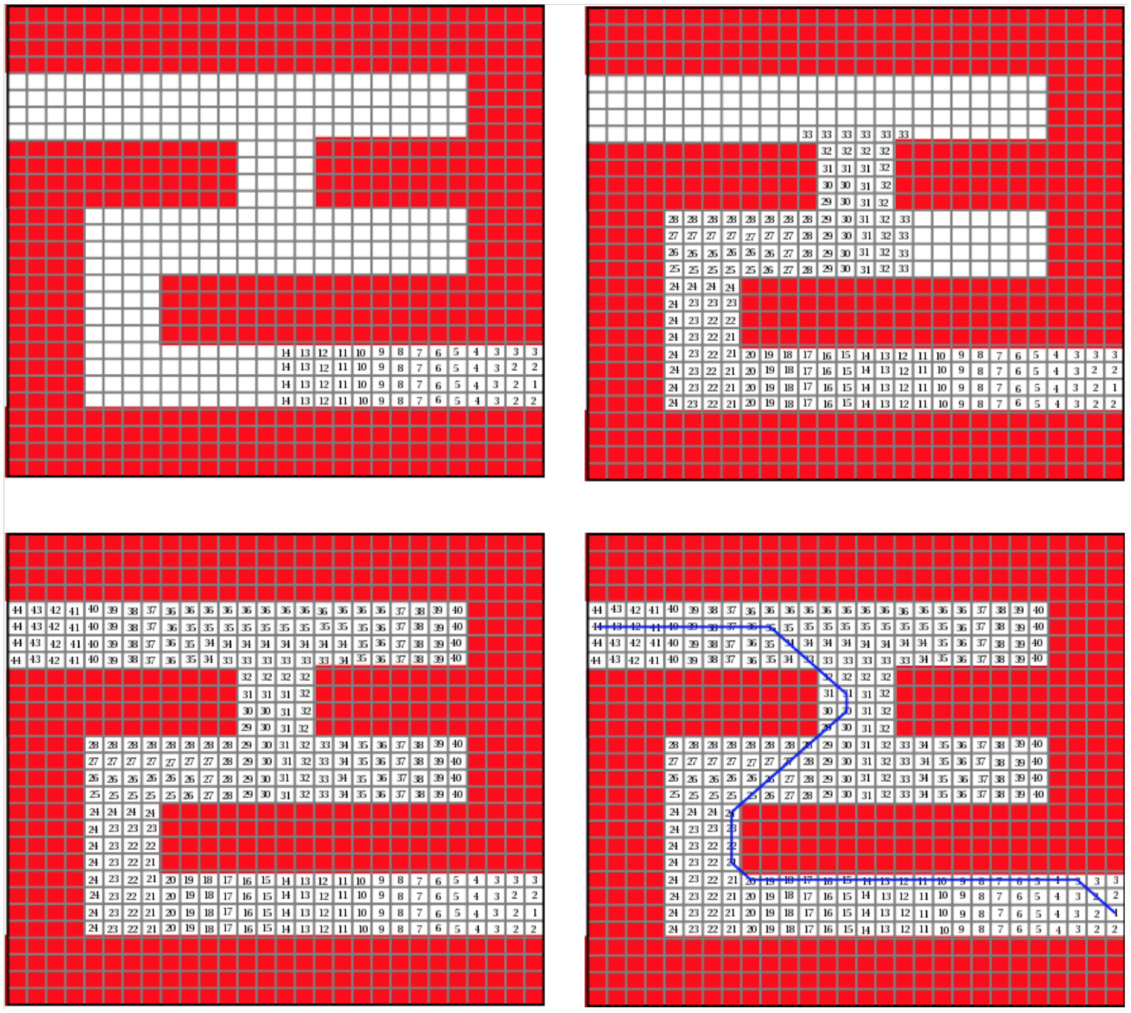


Fig. 4.55: Wavefront algorithm progress.

The first three images in Fig. 4.55 give you a few snapshots of the process on a maze. You may note that these numbers are just the number of pixel steps from your current location to the goal. It is a travel distance. Next is the Descent algorithm. Starting at the start point, look around for the pixel with the smallest label or value. Step there and repeat the process. Continue stepping downhill until you reach the goal pixel.

Wavefront Label “Fill”

Begin at goal pixel.

Label the goal pixel 1

```
repeat
    Label all of the unlabeled neighbors of the pixels with the label “k” using the label “k+1”.
until you run out of unlabeled pixels
```

```
Wavefront Descent “Path”

Begin at start pixel.
repeat
    Pick the neighbor pixel with the smallest label (or value).
    Step to that pixel.
until you arrive at goal pixel.
```

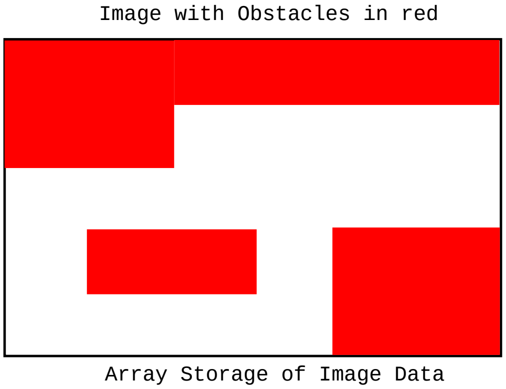
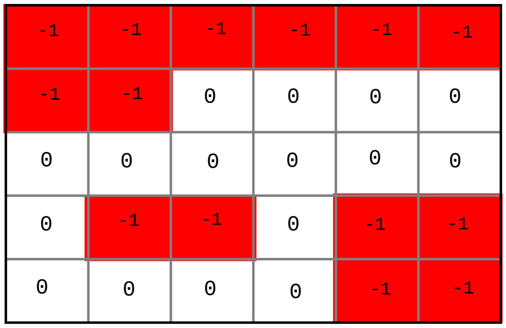


Fig. 4.56: Wave front progression (0)

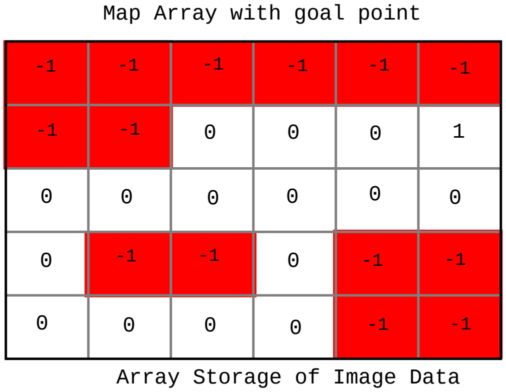
Initial Map Array



-1	-1	-1	-1	-1	-1
-1	-1	0	0	0	0
0	0	0	0	0	0
0	-1	-1	0	-1	-1
0	0	0	0	-1	-1

Array Storage of Image Data

Fig. 4.57: Wave front progression (1)

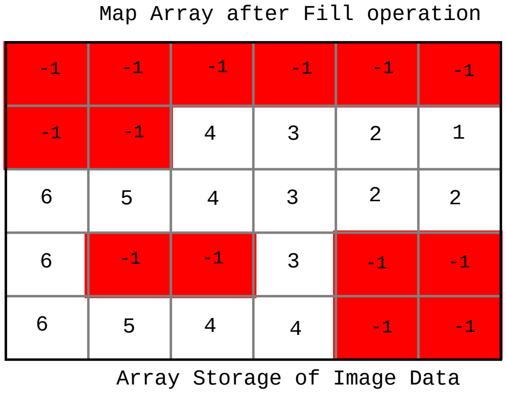


Map Array with goal point

-1	-1	-1	-1	-1	-1
-1	-1	0	0	0	1
0	0	0	0	0	0
0	-1	-1	0	-1	-1
0	0	0	0	-1	-1

Array Storage of Image Data

Fig. 4.58: Wave front progression (2)

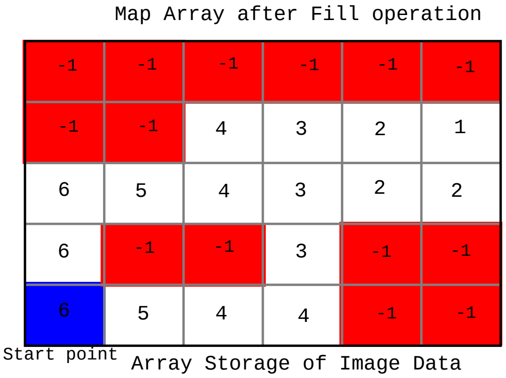


Map Array after Fill operation

-1	-1	-1	-1	-1	-1
-1	-1	4	3	2	1
6	5	4	3	2	2
6	-1	-1	3	-1	-1
6	5	4	4	-1	-1

Array Storage of Image Data

Fig. 4.59: Wave front progression (3)



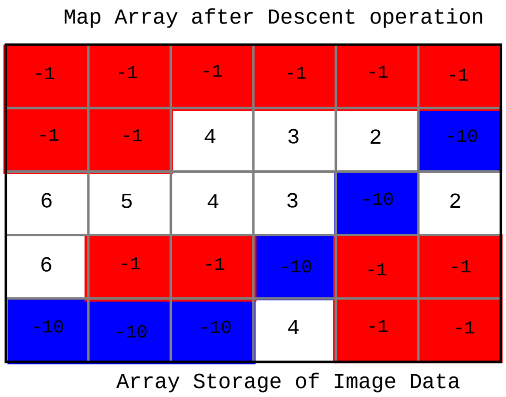
Map Array after Fill operation

-1	-1	-1	-1	-1	-1
-1	-1	4	3	2	1
6	5	4	3	2	2
6	-1	-1	3	-1	-1
6	5	4	4	-1	-1

Start point

Array Storage of Image Data

Fig. 4.60: Wave front progression (4)


Map Array after Descent operation
Array Storage of Image Data

-1	-1	-1	-1	-1	-1
-1	-1	4	3	2	-10
6	5	4	3	-10	2
6	-1	-1	-10	-1	-1
-10	-10	-10	4	-1	-1

Fig. 4.61: Wavefront fill example complete.

4.7 Guidance

First a few definitions are in order. According to Wikipedia, GNC, Guidance, Navigation and Control:

- Guidance refers to the determination of the desired path of travel (the "trajectory") from the vehicle's current location to a designated target, as well as desired changes in velocity, rotation and acceleration for following that path.
- Navigation refers to the determination, at a given time, of the vehicle's location and velocity (the "state vector") as well as its attitude.
- Control refers to the manipulation of the forces, by way of steering controls, thrusters, etc., needed to execute guidance commands whilst maintaining vehicle stability.

Planners can find paths through free space. They may be planning based on certain requirements such as minimal path, minimal curvature or maximum distance to obstacle. Some but not all will respect the kinematic constraints. Computationally, it may be rather expensive to have the planner compute the path at a high resolution. The resolution may be sufficiently coarse that it provides online performance in terms of the planning, but opens the door for excessive drift. One may interpolate between distant points using a polynomial interpolant. Cubic splines are a very common interpolant which allows for endpoint values and slope to be selected. Cubics will also provide smooth paths which minimize curvature.

It should be noted that we are not trying to find $x(t)$ and $y(t)$ to plug into the inverse kinematics for some robot. We will use the parametric equations to create a finer grid of points which is in turn handed to the speed and heading controller. Between the fine grid points, the controller is driving and we are not using the inverse kinematics.

4.7.1 Cubic Spline Example

Assume you want the spline that connects the points (1,-1) with (3,4). Also assume that the derivative at (1,-1) is given by $< 1, -3 >$ and at (3,4) is given by $< 0, 2 >$. We can take $t_0 = 0$ and $t_1 = 1$. This gives $z = t$, $\dot{z} = 1$, $a = 1 - 2 = -1$, $b = 2$, $c = -8$, $d = 3$. This gives us the two splines for the parametric description of the curve:

$$x(t) = (1 - t) + 3t + t(1 - t)[-1(1 - t) + 2t] = -3t^3 + 4t^2 + t + 1$$

$$y(t) = -(1-t) + 4t + t(1-t)[-4(1-t) + 3t] = -11t^3 + 19t^2 - 3t - 1$$

$$\dot{x} = -9t^2 + 8t + 1, \quad \ddot{x} = -18t + 8$$

$$\dot{y} = -33t^2 + 38t - 3, \quad \ddot{y} = -66t + 38$$

See Fig. 4.62 for a plot.

```
t0, t1 = 0, 1
x0, y0 = 1, -1
x1, y1 = 3, 4
xd0 , yd0 = 1, -3
xd1 = 0
yd1 = 2
dt = (t1-t0)
dx = (x1-x0)
dy = (y1-y0)
a = xd0*dt- dx
b = -xd1*dt+dx
c = yd0*dt-dy
d = -yd1*dt+dy
t = np.linspace(t0,t1,100)
dotz = 1.0/dt
z = (dotz)*(t-t0)
x = (1-z)*x0 + z*x1+z*(1-z)*(a*(1-z)+b*z)
y = (1-z)*y0 + z*y1+z*(1-z)*(c*(1-z)+d*z)
ptx = np.array([x0,x1])
pty = np.array([y0,y1])

plt.figure()
plt.xlim(0,4)
plt.ylim(-2,5)
plt.plot(ptx,pty, 'ro',x,y,'g-')
plt.legend(['Data', 'Interpolant'],loc='best')
plt.title('Cubic Spline')
plt.show()
```

Example

Assume that your planner has provided the following points (0,0), (10,50), (30,20). Also assume that you start with zero derivative at (0,0), would like to pass through (10,50) with slope $m = 1$ and have slope $m = -1$ at (30,20). You would like to create set of points on the curve separated by a distance of roughly 1 and not 10 or 50. How can you do this? The solution is to create two cubic splines which will match slopes at the three points. First we convert the problem into a parametric problem: (t, x, y): (0,0,0), (10,10,50) and (t, x, y): (0,10,50), (20,30,20) This was an arbitrary choice for the time values. Working on the first segment and the spline formulas, for $t_0 = 0$, $(x, y) = (0, 0)$ and $(\dot{x}, \dot{y}) = (1, 0)$ and for $t_1 = 10$, $(x, y) = (10, 50)$ and $(\dot{x}, \dot{y}) = (1, 1)$. From the data we then obtain for the first segment $z = 0.1t, a = 0, b = 0, c = -50, d = 40$, and on the next segment $z = 0.05t, a = 0, b = 0, c = 50, d = -10$.

The plot, Fig. 4.63 is produced by the following code with setting the plot command to lines, g-. The following code as is produces Fig. 4.64.

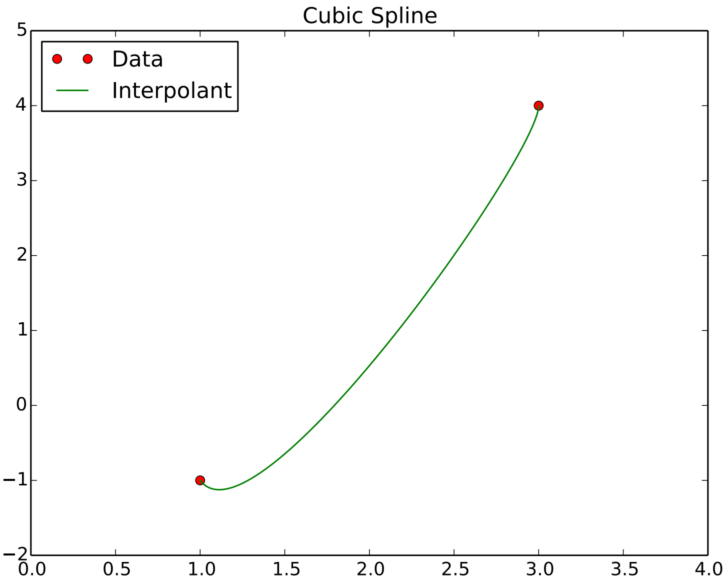


Fig. 4.62: Graph of the spline for example *cubicsplineexample*.

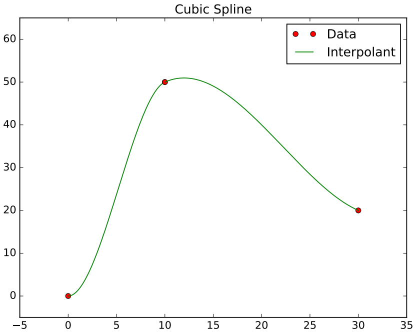


Fig. 4.63: The two cubic splines from the three data points.

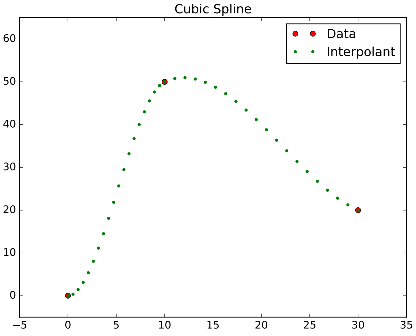


Fig. 4.64: Sampling the two splines to get guidance data.

```
import numpy as np
import pylab as plt

def spline(t0,t1, x0, x1, y0, y1, xd0 , yd0, xd1, yd1, N):
  dt = (t1-t0)
  dx = (x1-x0)
  dy = (y1-y0)
  a = xd0*dt- dx
  b = -xd1*dt+dx
  c = yd0*dt-dy
  d = -yd1*dt+dy
  t = np.linspace(t0,t1,N)
  dotz = 1.0/dt
  z = (dotz)*(t-t0)
  x = (1-z)*x0 + z*x1+z*(1-z)*(a*(1-z)+b*z)
  y = (1-z)*y0 + z*y1+z*(1-z)*(c*(1-z)+d*z)
  ptx = np.array([x0,x1])
  pty = np.array([y0,y1])
  return x, y, ptx, pty

N = 20
t0, t1 = 0, 10
x0, y0 = 0, 0
x1, y1 = 10, 50
xd0 , yd0 = 1, 0
xd1, yd1 = 1, 1
xc1, yc1, ptx1, pty1 = spline(t0,t1, x0, x1, y0, y1, xd0 , yd0, xd1, yd1, N)

t0, t1 = 0, 20
x0, y0 = 10,50
x1, y1 = 30, 20
xd0 , yd0 = 1, 1
xd1, yd1 = 1, -1
xc2, yc2, ptx2, pty2 = spline(t0,t1, x0, x1, y0, y1, xd0 , yd0, xd1, yd1, N)

plt.figure()
plt.xlim(-5,35)
plt.ylim(-5,65)
plt.plot(ptx1,pty1, 'ro')
plt.plot(ptx2,pty2, 'ro')
plt.plot(xc1,yc1,'g.')
plt.plot(xc2,yc2,'g.')
plt.legend(['Data', 'Interpolant'],loc='best')
plt.title('Cubic Spline')
plt.savefig("cubicexample2.pdf")
plt.show()
```

4.8 Problems

1. Using Python with Matplotlib, code the basic motion algorithm.

   a. Demonstrate your approach with one obstacle.

   b. Demonstrate with several obstacles.

2. Using Python with Matplotlib and the basic motion algorithm, place a set of obstacles that cause the robot to cycle and not find the goal. In other words, build a robot trap.

3. Write a Python algorithm to perform boundary following on a grid domain.

4. Write a boundary following routine for the DD robot in Gazebo using the Lidar. Use a video screen capture program to record the results.

5. Assume that you have a finite number of convex solid obstacles (solid means you are not starting inside). Prove or provide a counter-example.

   a. Will Bug 1 succeed in navigating from any start point to any goal point?

   b. Will Bug 2 succeed in navigating from any start point to any goal point?

   c. Will Tangent Bug succeed in navigating from any start point to any goal point?

6. Sketch equations (4.1) and (4.2).

7. Assume that you have a grid map of the type found in the left image of Fig. 4.52 which was stored in an array. If the start point was an interior cell, implement the Bug 1 algorithm to find the sequence of cells which describe an escape path if one exists.

8. Is it possible to have a single non-convex obstacle trap Bug 1 or Bug 2?

9. Assume that you have a grid domain and the obstacles are represented in the grid map. Write a Python program to implement:

   a. Bug 1

   b. Bug 2

   c. Bug 3

   d. Tangent Bug

10. Write a Python program to implement the Wavefront algorithm.

    1. Demonstrate on a map with multiple obstacles.

    2. Compare to the A^* approach in the previous exercise.

CHAPTER

FIVE

VEHICLE MOTION

In this chapter, we model several motion systems and focus one of the most common robot drive systems known as the differential drive. We begin with ground contact components such as wheels and tracks. Move to steering and drive systems. The differential drive will work as the canonical example for which other drive systems will be discussed. Following the differential drive we derive a general approve to modeling drive systems. This is used to model steered vehicles and several types of Omniwheel systems.

5.1 Locomotion

The word locomotion means “The act of moving from place to place.” [Com09] from Latin combining location and motion. Biology has explored a variety of very interesting ways to move around. Animals can be carried on currents, swim, crawl, slide, walk, run, jump, and fly.

Most of the locomotion solutions found in nature are hard to imitate with machines. Although legs are the base for human locomotion, humans have valued wheels in their motion solutions. The earliest recorded appearance of wheels is in Mesopotamia (Sumerian) at the mid-4th millenium BC. The is evidence of independent discovery in the new world, the lack of domesticated large animals prevented any development beyond children’s toys.

Rolling is very efficient, especially compared to dragging or carrying materials. At the macroscopic scale, nature has not developed wheels. This is not surprising since the wheel needs to be disconnected from the rest of the system for free rotation reasons, but would then not have the required nutrient supply (blood or something similar) to grow, develop and maintain the structure. Although nature did not evolve large wheels, human motion has some similarities to rolling - a rolling polygon when motion models are examined.

We are no longer bound to wheels as the only choice for motion. We can implement a number of nature inspired motion solutions. The type of motion used by a robot is often the most notable aspect of the machine. Certainly a great deal of interest and entertainment can be found in implementing novel locomotion into a robot.

As a robotics engineer, you may be asked to choose the type of motion, meaning you must choose “fly, hop, swim, walk, roll . . . ” Often the environment decides for you, for example, if you must operate in the air or water, or in very rough terrain. You may have other constraints involved like power consumption, weight or robustness. These constraints will normally push the design towards one locomotion system.

When looking at a wheel verses an articulator there are some standard issues that must be addressed. Articulation is much more complicated in design, control and expense (both energy and financial). Legs (articulators) require more actuators and thus more control components. The control system is more complicated than with a wheeled design. Another consideration is energy. Articulators move their center of mass for locomotion and may have to move a significant amount of hardware. Thus there is internal mass movement along with the external mass (the vehicle). Wheels by their very design keep the center of mass at a constant distance from the ground. This reduces power usage.

The tradeoff for the efficiency gain is that articulated motion has the possibility of enhanced environmental robustness. This is clear when watching any household spider run across a textured ceiling. The most efficient wheel, the rail wheel is also the least robust in that it does not operated outside an instrumented environment. Cars use wheels that can run on a road or flat ground. To go beyond this we need to bring the rail or road along with - the idea behind tracks. Track systems can operate in a larger set of environments, but at cost of energy.

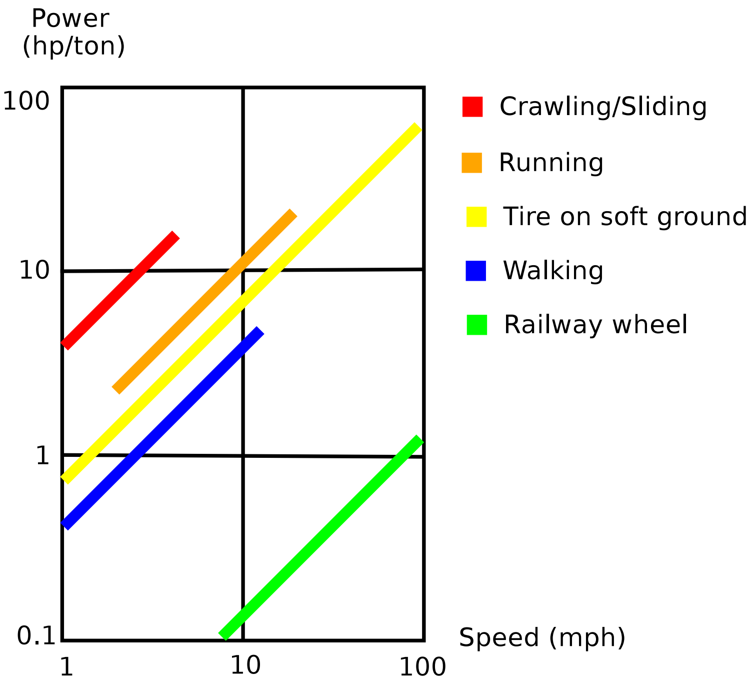


Fig. 5.1: The relations between energy, speed and motion type.

5.2 Wheels

For thousands of years wheels have worked very well. Most vehicles we imagine are two or four wheeled systems. Two wheels use the gyroscopic effect to provide stability. Static (passive) stability of the vehicle is assured by having three wheels and the center of gravity in the triangle formed by the ground contact points of the wheels. Trikes are less popular on the road due to concerns about instabilities during turning. Additional wheels, additional ground contact points, can improve the stability. Four wheels provides the stability in the turn at the cost of needing a suspension system (more than three require suspension). Suspension systems do more than level the ride as they can keep all the wheels on the ground when traveling over rough terrain. This provides better traction as well as avoids digging in too deeply. Larger wheels give greater obstacle traversal due to the decreased angle of attack which reduces the required torque. Bigger wheels are heavier and require greater reductions in the gear box however. Selection of wheels is based on the surface and the application. Hard dry smooth surfaces may use smooth wheels and rougher or slicker surfaces demand tires that are rough and maybe soft.

The effectiveness of the wheel is given by the contact area of the wheel. This is a combination of wheel width and tread design. Angle of contact combined with tire shape will affect the steering response. Wheel tread, width and other parameters will affect the rolling friction and the energy loss in motion. To gain maneuverability, wheels can be steered or replaced with omni-wheels. This requires additional hardware and controls which increases complexity, weight and cost. Most designs do not allow the craft to maneuver and orient simultaneously and independently which increases the control effort. As with many aspects of engineering, this is a tradeoff between simple, robust and inexpensive design verses a flexible, maneuverable, adaptable design.

5.2.1 Omni, Mecanum and Spherical Wheels

Of all the wheel types, none captures attention like the omni and Mecanum wheels. Their operation is unexpected and at first seems to defy intuition. These wheels are capable of motion in the rolling direction as well as motion along the axle direction which leads to holonomic robots. Which means the robot can position and orient independently in the plane. It makes for a very maneuverable robot which is very popular. These wheels require hard flat surfaces to work properly. If the dirt or small stones get lodged into the rollers or if the rollers lose contact with the surface, the holonomic motion is compromised. So these wheels are used exclusively indoors. Since fine precision maneuvering is normally required for indoor systems and not outdoor systems, there has not been much effort expended to make outdoor versions.

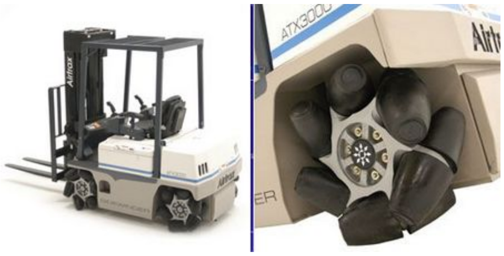


Fig. 5.2: The Airtrax forklift.

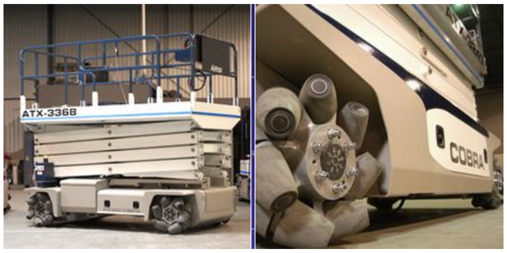


Fig. 5.3: The Airtrax scissor lift.

The omni wheel's first patent was in 1919 by Grabowiecki. The Mecanum wheel was developed by Bengt Erland Ilon in 1972 while working for the Mecanum company. Airtrax, an American forklift company purchased patent rights and briefly manufactured forklifts with a heavy duty version of the Mecanum wheel. These wheels have much less ground friction in a turn in comparison to a skid steer requiring much less torque.

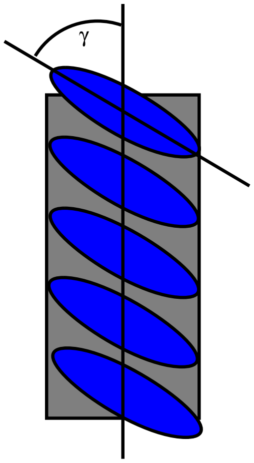


Fig. 5.4: The γ measure.

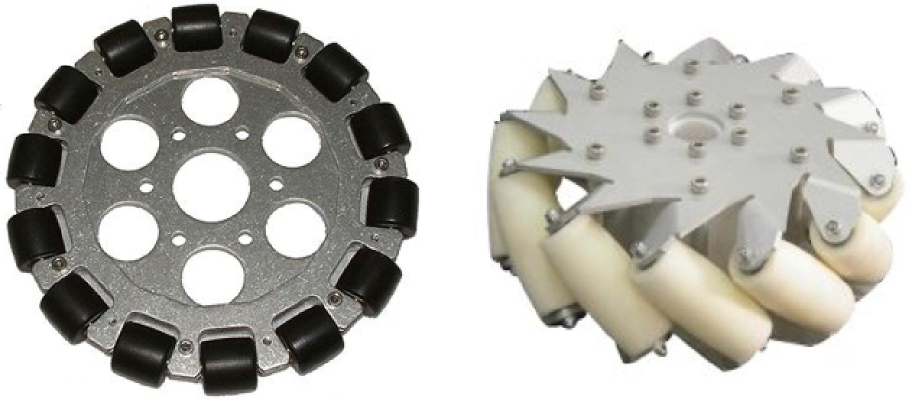


Fig. 5.5: The (a) $\gamma = 0$ configuration and (b) $\gamma = 45^\circ$ configuration.

For this text, we will combine the omni and Mecanum wheels and just call them omniwheels. The difference between them is only in the angle the rollers are mounted on the wheel body. Fig. 5.5 shows some sample types of omniwheels using the $\gamma = 0$ configuration and $\gamma = 45^\circ$ configuration. Normally the $\gamma = 0$ style of wheel is used in non-parallel mounting as shown in the first robot in the Fig. 5.6 and the parallel mounting is used for the other standard type of wheel design using $\gamma = 45^\circ$.

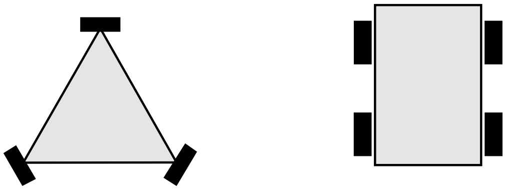


Fig. 5.6: Normal mounting style for $\gamma = 0$ and $\gamma = 45^\circ$.

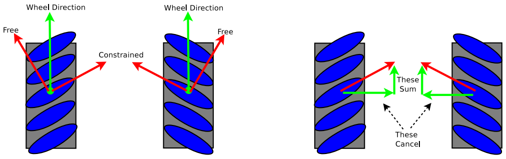


Fig. 5.7: Force vectors induced by rotation with the $\gamma = 45^\circ$ configuration.

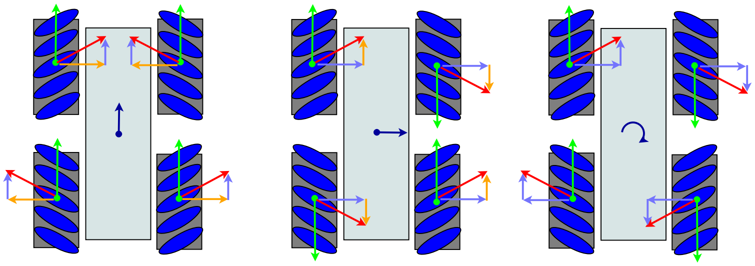


Fig. 5.8: Mecanum rotation directions and vector forces for different vehicle directions.

- Driving forward: all four wheels forward
- Driving backward: all four wheels backward
- Driving left: 1,4 backwards; 2,3 forward
- Driving right: 1,4 forward; 2,3 backward
- Turning clockwise: 1,3 forward; 2,4 backward
- Turning counterclockwise: 1,3 backward; 2,4 forward

A variation of the omni wheel is the omni ball developed by Kaneko Higashimori Lab at Osaka University, see Fig. 5.10. This wheel will be used to drive tracks in a very novel approach described in the tracks section below. This wheel fails to be a true spherical wheel as far as two directional motion is concerned and has motion equations similar to the omniwheel systems.

Omni and Mecanum wheels can be driven on only one direction and only when combined with other wheels are they able to move against the rolling directions. To gain two dimensional directional capability the wheel needs to be a sphere or at least approximate the sphere in a significant manner. This can be done by reversing

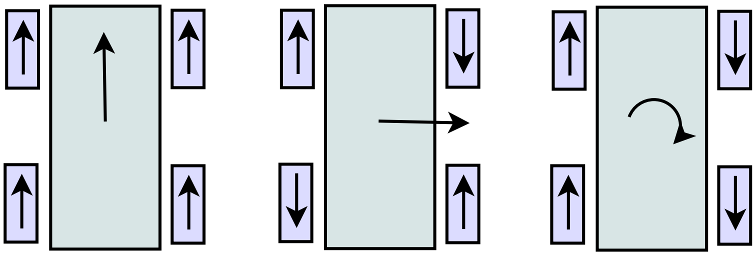


Fig. 5.9: Summary of wheel motion and directions

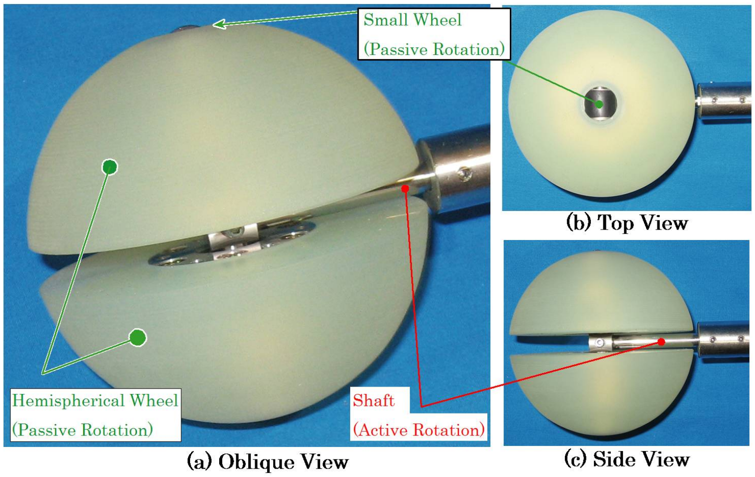


Fig. 5.10: The Omni Ball Wheel developed at the Kaneko Higashimori Lab at Osaka University

the power direction from the classical mechanical computer mouse. In the mechanical mouse the ball is forced around which drives small disks inside in the component directions. By mounting three omniwheels on top of a ball, one can gain motion in two directions. Fig. 5.12 shows one design by Dr. Masaaki Kumagai, director of the Robot Development Engineering Laboratory at Tohoku Gakuin University.

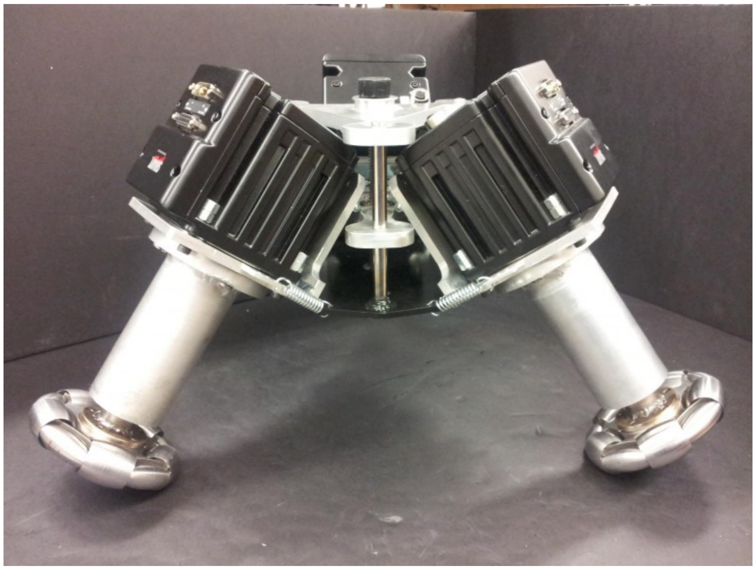


Fig. 5.11: Omniwheel drive system

5.2.2 Mobility Issues

The stability of the craft is given by several factors. Having less than three contact points requires dynamic balance for a system which is "at rest". Having less than 6 contact points means that during locomotion, the system requires dynamic balance during motion or one is moving at most two legs at a time making a more complicated control system. The location of the center of gravity is an important aspect of dynamic stability. A lower center of gravity helps to avoid falling over.

For ground systems, the terrain will have more influence than with air or sea. We have to worry about the terrain roughness, slickness, grades and other issues. The number of wheels, type of wheels, type of suspension, and steering will all have a large affect on the effectiveness of motion.

5.2.3 Tracks

For the purposes of this text, we will treat unsteered tracked systems (tank treads) as two-wheel differential drive (wheeled) systems. The modeling is more difficult than with wheels. Modeling the skid-steer turns requires details about the track system and the surface. Since rocks, mud and other aspects of the surface can have significant effects on turning friction, models have limited utility.

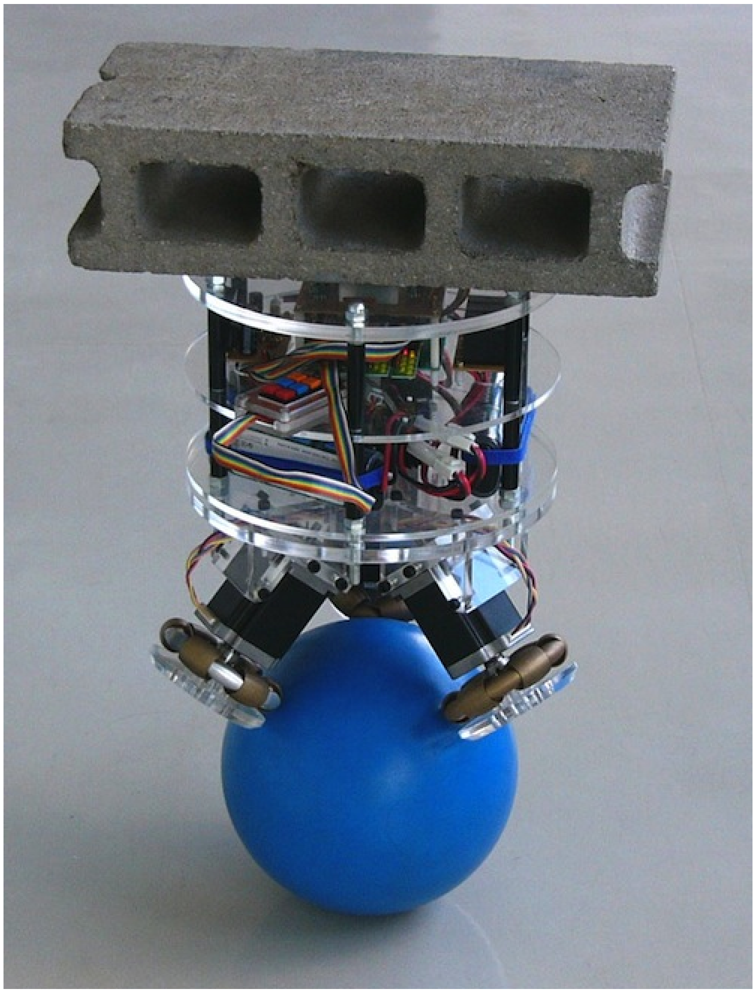


Fig. 5.12: Omniwheel balancing robot

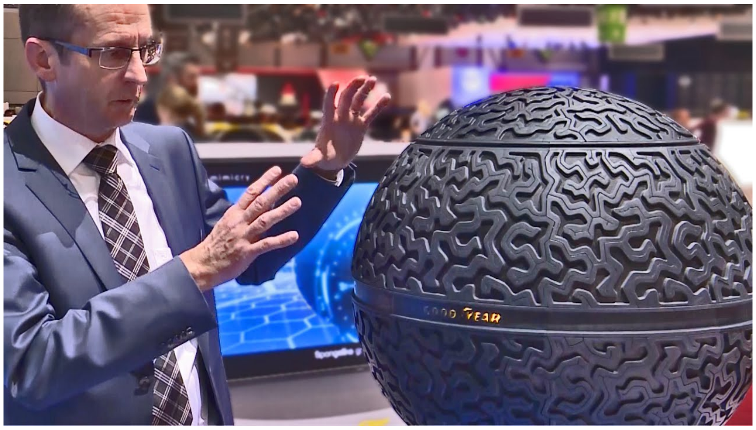


Fig. 5.13: GoodYear Spherical Wheel Concept Tire

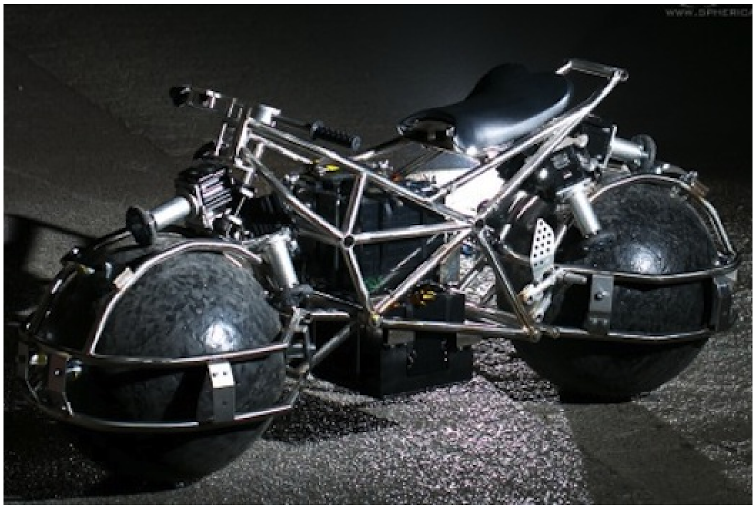


Fig. 5.14: Prototype omnidirectional motorcycle

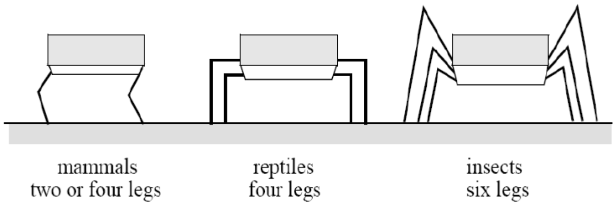
5.3 Legs

We now move over to a more biological approach. The use of legs in locomotion has been very successful. Animals range in sizes from single millimeter to multiple meter range. Legs have proved invaluable at many space and speed dimensions. In the robotics view, a leg is articulated manipulator (serial chain). This means that it requires many of the same controls that a robot arm would require, but adapted to the specific task of moving the robot. Although a hobby level robot can implement 6 simple articulators to produce insect like motion, getting natural, efficient, robust and fast bipedal and quadrupedal motion is very difficult.

5.3.1 Aspects of Articulated Based Motion

Engineers are experimenting with 2 - 8 leg designs to learn more about articulator locomotion as well as what it can teach us about the animals that have similar designs. We know that for static stability, at least three points of contact are required. So, three legs are required to stand still. When we move, some of the legs must move forward to initiate the gait. If we want three points on the ground, this means six legs is the minimum for stable walking. Robots using six or more legs do not need a balance control system.

Systems using four legs have a static balance when still, but must use a dynamic control approach to maintain balance during the step. This is a variation of the inverted pendulum problem which is discussed in the chapter on motion control. Finally our system of using two legs requires a control system for moving and standing.



Increasing the number of legs will increase weight, power requirements, coordination problems and control hardware. Adding legs, as mentioned before, will help with stability and provide a greater number of contact points. Increasing ground contact can increase traction and robustness, depending on the contact area, angle of contact, friction, surface roughness and friction. For static stability we want the center of mass to fall inside the span of the legs. Unlike wheels, the center of mass for a leg moves up and down as the robot walks. This decreases power efficiency, increases the chance of a shifting center of gravity which makes path planing more difficult. In complicated environments, one may have to plan the motion of each articulator. The configuration space for a 6 legged robot with three servos is 18, which is rather large for path planning and might fail due to size.

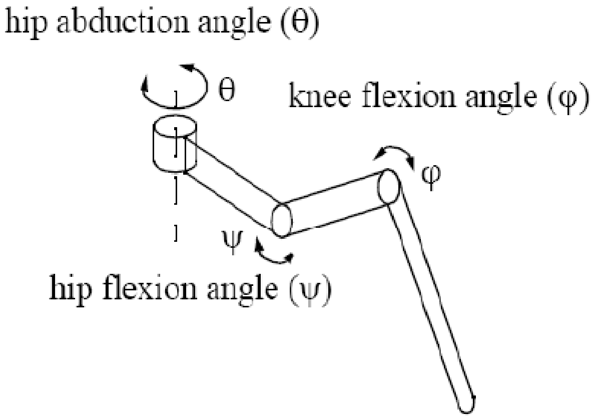


Fig. 5.15: Leg joints and their use.

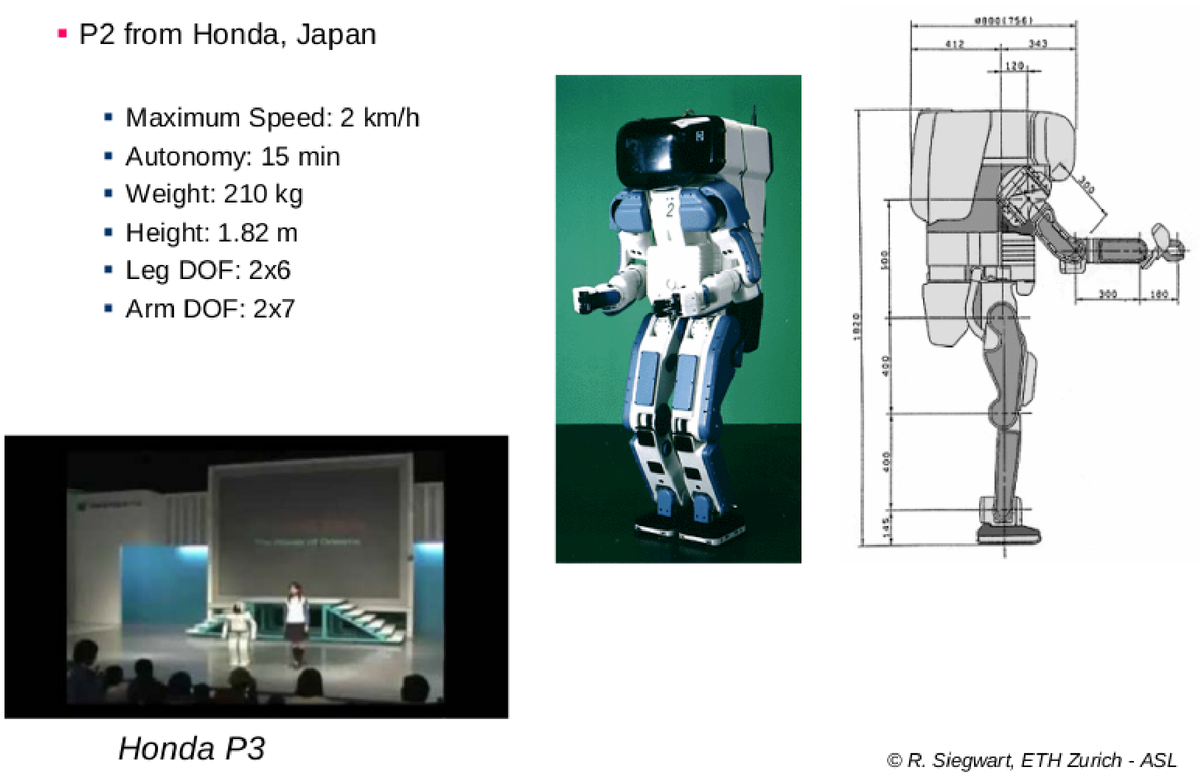
- Two DOF is required:
  lift and swing
- Three DOF is needed in most cases:
  lift, swing and position
- Fourth DOF is needed for stability:
  ankle joint - improves balance and walking

Consider a humanoid robot. How complex are they? A leg has a hip (two degrees of freedom) and knee plus ankle which gives another two degrees of freedom. So a leg is a 4 DOF (degrees of freedom) structure. An arm is at least 5 DOF. A head has pan and tilt so at least 2 DOF. This adds up to 20 DOF for a humanoid style robot. Motion planning and control is challenging for high DOF robots. Good tools for doing this is an active area of research.

There are several attempts to combine legs and wheels. The Shrimp is one such design, Fig. 5.16. Many fun and interesting innovations come from the suspension system. Adaptive (passive or active) suspension is a current area of development, Fig. 5.17. Other lines of development look to blending sensing with the wheel or suspension system. One example is the flexible wheel, Fig. 5.18.

5.4 Drive Systems

There are as many different drives systems as ways to mount wheels on frames. Well, almost. What we will do here is to cover a few common designs and leave the more exotic systems to the mechanical engineering



- P2 from Honda, Japan
  - Maximum Speed: 2 km/h
  - Autonomy: 15 min
  - Weight: 210 kg
  - Height: 1.82 m
  - Leg DOF: 2x6
  - Arm DOF: 2x7

Honda P3

© R. Siegwart, ETH Zurich - ASL

φ800(756)
412
343
120
330
300
180
1820
500
400
400
145

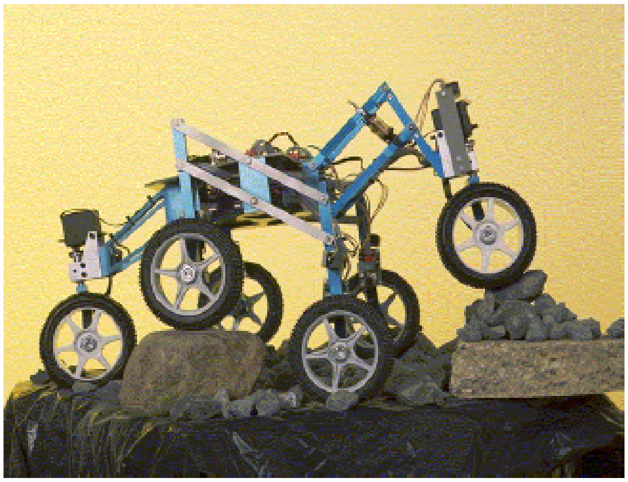


Fig. 5.16: Walking Wheels

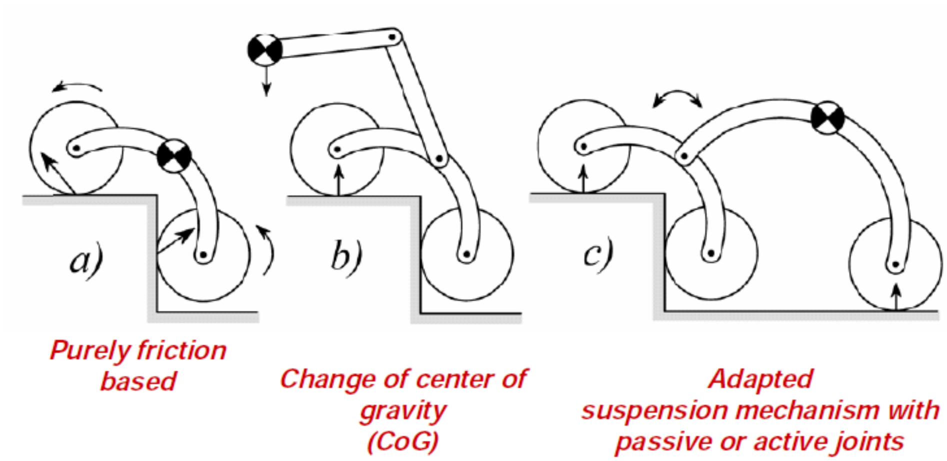

a)
b)
c)
Purely friction based
Change of center of gravity (CoG)
Adapted suspension mechanism with passive or active joints

Fig. 5.17: Adaptive Suspension

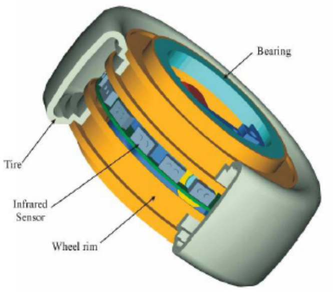


Fig. 5.18: Flexible Wheel

literature. Vehicle design is fun and we could devote many pages to the subject. However, in practice we really only see a limited number of drive systems and those are the one we will discuss. So, we get started with a simple system, the bicycle, which we will use to introduce some basic concepts. Then we present the equations of motion for the differential drive, omniwheel designs and the front wheel steered designs. The derivations for these systems are presented in the Kinematics chapter later on.

5.4.1 Basic constraints and the ICR

We will assume two basic contraints about our wheels. The first is *no slip* which will mean that the wheel has perfect traction and the distance around the wheel translates into the same linear distance. This condition is broken when we lose traction on surfaces with ice, snow or mud. The next constraint is the *no slide* constraint. This constraint means that we always move in the direction the wheel is pointed and not in the axle direction. This constraint can be violated on slick surfaces where the rotational motion of the craft causes the vehicle to slide sideways, again on ice, snow or mud. There are a couple of common vehicle designs that explicitly violate one or both conditions in order to turn. Examples include tank or treaded systems and skid steer systems like Bobcats. Many hobby robots have four driven but not steered wheels which are skid steer. This is not a bad thing, just that we cannot use the kinematic equations based on the no slip / no slide to model steered motion. In practice this is not a significant problem.

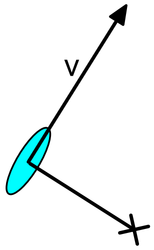


Fig. 5.19: No slide condition means the wheel motion is in the direction of v. No slip means the distance traveled is the same as the wheel arc distance traveled.

The no-slide assumption means that there is no motion in the direction of the axle. All of the motion is perpendicular to the axle. This means for each wheel, the no slide constraint generates a zero motion line orthogonal to the wheel plane.

The intersection of the zero motion lines is the ICR - Instantaneous Center of Rotation. Having a common intersection, an ICR, implies that each wheel is moving on a concentric circle. If the zero motion lines do not intersect at a single point, then no motion is possible when we have no-slip and no-slide for our wheels. We can easily see that this is the case for the simple steering approach shown above. The rear wheels have overlapping zero motion lines. The front wheels have parallel non-overlapping zero motion lines. All intersect at the ICR. Note that parallel lines intersect at infinity so we don't have to have a finite intesection.

5.4.2 Differential Drive

One of the most common drive system in robotics is the differential drive. Differential drive is a two wheeled drive system. For stability a third support must be employed. A castor wheel or ball is normally used. The

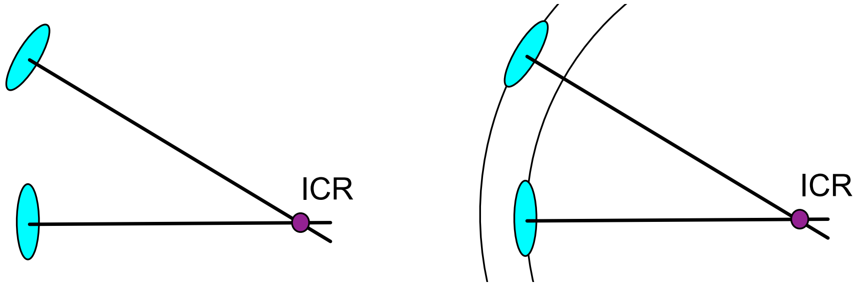


Fig. 5.20: ICR - Instantaneous Center of Rotation.

well known Rumba floor cleaning robot uses this system. It is stable, maneuverable, easy to control, and simple. Fig. 5.21 gives the basic layout and variables involved in the model.

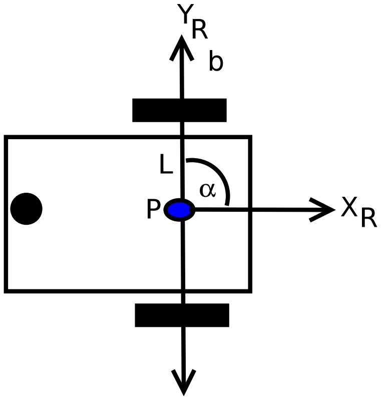


Fig. 5.21: The differential drive robot dimensions and variables.

Recall the Differential Drive robot setup Fig. 5.22:

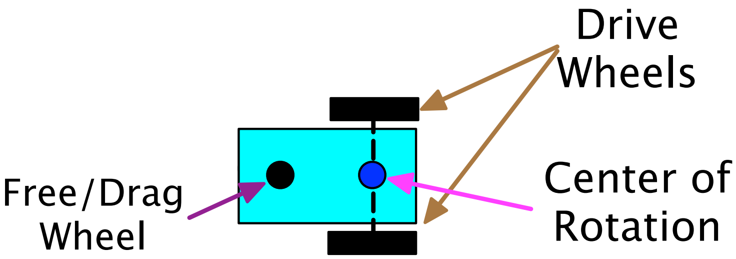


Fig. 5.22: Simple differential drive robot.

and the forward kinematics:

$$\boxed{\begin{aligned}\dot{x} &= \tfrac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\cos(\theta) \\ \dot{y} &= \tfrac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\sin(\theta) \\ \dot{\theta} &= \tfrac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2)\end{aligned}}$$

and the inverse kinematics:

$$\boxed{\begin{aligned} v &= \sqrt{\dot{x}^2+\dot{y}^2}, \\ \kappa &= \frac{\dot{x}\ddot{y}-\dot{y}\ddot{x}}{v^3} \\ \dot{\phi}_1 &= \frac{v}{r}\left(\kappa L+1\right) \\ \dot{\phi}_2 &= \frac{v}{r}\left(-\kappa L+1\right) \end{aligned}} \tag{5.1}$$

where $\dot{\phi}_1$ and $\dot{\phi}_2$ be the right and left wheel rotational speeds (respectively), r is wheel radius and L is the axle length from the center to the wheel ("half axle").

Alternate Form

In some cases we only need to know the forward velocity and the vehicle rotation rate. By computing v from (2.7) and using $\omega = \dot{\theta}$, we obtain

$$\begin{aligned} v &= \tfrac{r}{2}(\dot{\phi}_1+\dot{\phi}_2) \\ \omega &= \tfrac{r}{2L}(\dot{\phi}_1-\dot{\phi}_2) \end{aligned} \tag{5.2}$$

and the inverse of these are

$$\begin{aligned} \dot{\phi}_1 &= \tfrac{1}{r}(v+L\omega) \\ \dot{\phi}_2 &= \tfrac{1}{r}(v-L\omega) \end{aligned} \tag{5.3}$$

5.4.3 Omniwheels

Fig. 5.5 shows some sample types of omniwheels using the $\gamma = 0$ configuration and $\gamma = 45^\circ$ configuration. Also recall that $\gamma = 0$ style of wheel is used in non-parallel mounting as shown in the first robot in the Fig. 5.6 and the parallel mounting is used for the other standard type of wheel design using $\gamma = 45^\circ$.

For this section we assume that we have a traditional care design frame and wheel mounting as described in Fig. 5.23 ($\gamma = 45^\circ$). The following notation is used in the kinematics:

- r - wheel radius.
- L_1 - distance between left and right wheel pairs, L_2 - distance between front and rear wheel pairs.
- v_x, v_y, ω - the robot velocity and angular velocity in robot coordinates.
- $\dot{x}$, $\dot{y}$, $\dot{\theta}$ - robot velocity in x, y and robot angular velocity in global coordinates.
- $\dot{\phi}_{FL}, \dot{\phi}_{FR}, \dot{\phi}_{BL}, \dot{\phi}_{BR}$ - front left, front right, back left, back right, radians per minute.

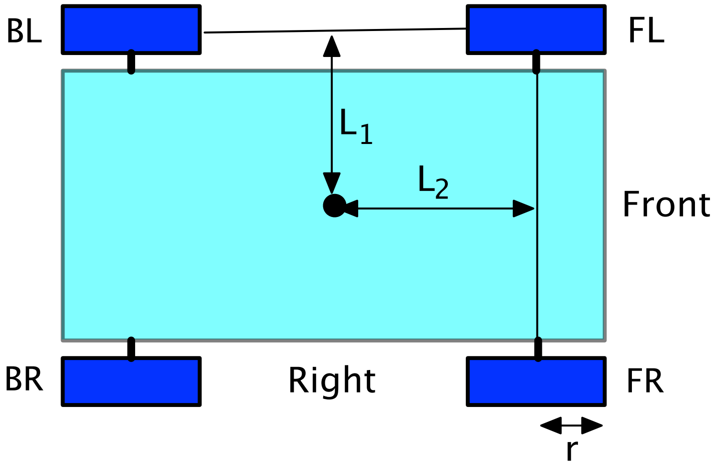


Fig. 5.23: Dimensions for the Mecanum Kinematics.

Forward kinematics

The forward local kinematics for this architecture is

$$\begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix} = \frac{r}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 \\ -\frac{1}{(L_1+L_2)} & \frac{1}{(L_1+L_2)} & -\frac{1}{(L_1+L_2)} & \frac{1}{(L_1+L_2)} \end{bmatrix}\begin{bmatrix} \dot\phi_{FL} \\ \dot\phi_{FR} \\ \dot\phi_{BL} \\ \dot\phi_{BR} \end{bmatrix}.$$

Applying the rotation to move to global coordinates

$$\begin{bmatrix} \dot x \\ \dot y \\ \dot\theta \end{bmatrix} = \frac{r}{4}R(\theta)\begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 \\ -\frac{1}{(L_1+L_2)} & \frac{1}{(L_1+L_2)} & -\frac{1}{(L_1+L_2)} & \frac{1}{(L_1+L_2)} \end{bmatrix}\begin{bmatrix} \dot\phi_{FL} \\ \dot\phi_{FR} \\ \dot\phi_{BL} \\ \dot\phi_{BR} \end{bmatrix}$$

$$= \frac{r}{4}R(\theta)\begin{bmatrix} \dot\phi_{FL}+\dot\phi_{FR}+\dot\phi_{BL}+\dot\phi_{BR} \\ -\dot\phi_{FL}+\dot\phi_{FR}+\dot\phi_{BL}-\dot\phi_{BR} \\ \frac{1}{(L_1+L_2)}\left(-\dot\phi_{FL}+\dot\phi_{FR}-\dot\phi_{BL}+\dot\phi_{BR}\right) \end{bmatrix}$$

$$= \frac{r}{4}\begin{bmatrix} \left(\dot\phi_{FL}+\dot\phi_{FR}+\dot\phi_{BL}+\dot\phi_{BR}\right)\cos(\theta) - \left(-\dot\phi_{FL}+\dot\phi_{FR}+\dot\phi_{BL}-\dot\phi_{BR}\right)\sin(\theta) \\ \left(\dot\phi_{FL}+\dot\phi_{FR}+\dot\phi_{BL}+\dot\phi_{BR}\right)\sin(\theta) + \left(-\dot\phi_{FL}+\dot\phi_{FR}+\dot\phi_{BL}-\dot\phi_{BR}\right)\cos(\theta) \\ \frac{1}{(L_1+L_2)}\left(-\dot\phi_{FL}+\dot\phi_{FR}-\dot\phi_{BL}+\dot\phi_{BR}\right) \end{bmatrix}.$$

So, finally we obtain

$$\begin{bmatrix} \dot x \\ \dot y \\ \dot\theta \end{bmatrix} = \frac{r}{4}\begin{bmatrix} A\cos(\theta) - B\sin(\theta) \\ A\sin(\theta) + B\cos(\theta) \\ \frac{1}{(L_1+L_2)}C \end{bmatrix} \tag{5.4}$$

where

$A = \left(\dot\phi_{FL} + \dot\phi_{FR} + \dot\phi_{BL} + \dot\phi_{BR}\right)$, $B = \left(-\dot\phi_{FL} + \dot\phi_{FR} + \dot\phi_{BL} - \dot\phi_{BR}\right)$, and $C = \left(-\dot\phi_{FL} + \dot\phi_{FR} - \dot\phi_{BL} + \dot\phi_{BR}\right)$.

To perform numerical calculations, we need to discretize the differential equations. Using the same process that we used to gain (3.3), we discretize the Mecanum equations. As before the time step is Δt, $x_k = x(t_k)$, $y_k = y(t_k)$, $\theta_k = \theta(t_k)$, $\omega_{FL,k} = \dot\phi_{FL}(t_k) \ldots$, and we have

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} + \frac{r\Delta t}{4} \begin{bmatrix} A\cos(\theta_k) - B\sin(\theta_k) \\ A\sin(\theta_k) + B\cos(\theta_k) \\ \frac{1}{(L_1+L_2)}C \end{bmatrix} \tag{5.5}$$

where $A = (\omega_{FL,k} + \omega_{FR,k} + \omega_{BL,k} + \omega_{BR,k})$, $B = (-\omega_{FL,k} + \omega_{FR,k} + \omega_{BL,k} - \omega_{BR,k})$, and $C = (-\omega_{FL,k} + \omega_{FR,k} - \omega_{BL,k} + \omega_{BR,k})$.

Inverse Kinematics for the Mecanum

We used a traditional care design frame and wheel mounting as described in Fig. 5.5 ($\gamma = 45°$). The inverse kinematics in local coordinates are given by

$$\begin{bmatrix} \dot\phi_{FL} \\ \dot\phi_{FR} \\ \dot\phi_{BL} \\ \dot\phi_{BR} \end{bmatrix} = \frac{1}{r} \begin{bmatrix} 1 & -1 & -(L_1+L_2) \\ 1 & 1 & (L_1+L_2) \\ 1 & 1 & -(L_1+L_2) \\ 1 & -1 & (L_1+L_2) \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix}.$$

Applying the coordinate transformation we can move to global coordinates

$$\begin{bmatrix} \dot\phi_{FL} \\ \dot\phi_{FR} \\ \dot\phi_{BL} \\ \dot\phi_{BR} \end{bmatrix} = \frac{1}{r} \begin{bmatrix} 1 & -1 & -(L_1+L_2) \\ 1 & 1 & (L_1+L_2) \\ 1 & 1 & -(L_1+L_2) \\ 1 & -1 & (L_1+L_2) \end{bmatrix} R^{-1}(\theta) \begin{bmatrix} \dot x \\ \dot y \\ \dot\theta \end{bmatrix}$$

$$= \frac{1}{r} \begin{bmatrix} 1 & -1 & -(L_1+L_2) \\ 1 & 1 & (L_1+L_2) \\ 1 & 1 & -(L_1+L_2) \\ 1 & -1 & (L_1+L_2) \end{bmatrix} \begin{bmatrix} \cos(\theta)\dot x + \sin(\theta)\dot y \\ -\sin(\theta)\dot x + \cos(\theta)\dot y \\ \dot\theta \end{bmatrix}$$

$$= \frac{1}{r} \begin{bmatrix} \cos(\theta)\dot x + \sin(\theta)\dot y + \sin(\theta)\dot x - \cos(\theta)\dot y - (L_1+L_2)\dot\theta \\ \cos(\theta)\dot x + \sin(\theta)\dot y - \sin(\theta)\dot x + \cos(\theta)\dot y + (L_1+L_2)\dot\theta \\ \cos(\theta)\dot x + \sin(\theta)\dot y - \sin(\theta)\dot x + \cos(\theta)\dot y - (L_1+L_2)\dot\theta \\ \cos(\theta)\dot x + \sin(\theta)\dot y + \sin(\theta)\dot x - \cos(\theta)\dot y + (L_1+L_2)\dot\theta \end{bmatrix}. \tag{5.6}$$

5.4.4 Steered Systems

Automobiles are nearly exclusive to a front wheel steering system (for a variety of reasons not discussed here). There are lots of ways to approach steering and some work better than others. If the front wheels are turned, the vehicle starts a circular arc either to the left or right. Geometrically this generates two concentric circles which are not the same size. The inside and outside wheel on a given axle do not rotate at the same speed or point in the same direction. Parallel wheels will skid on a turn. The mechanical solution to the problem is listed in a patent by Ackermann, but the solution predates by more than a half century. We will discuss this issue in greater detail in the Kinematics Chapter.

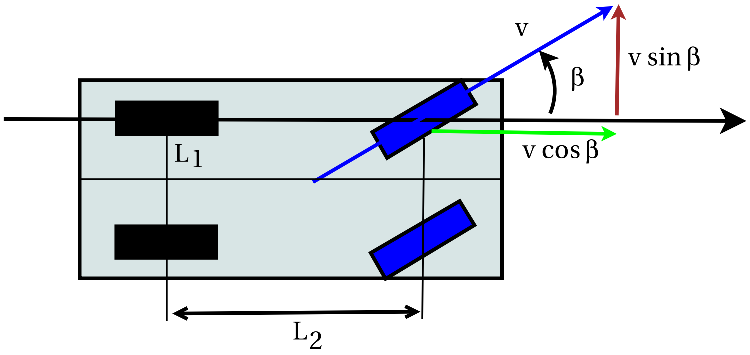


Fig. 5.24: Front Wheel Steered System.

Ackerman

The best known mobile vehicle design currently is the steered wheel, specifically the Ackerman Steering design. This is our traditional car implementation. It is a rectangular vehicle with four wheels. The front two wheels are steered. We begin with the fixed turn angle or simple steer model.

$$\begin{bmatrix} v \\ \dot{\theta} \end{bmatrix} = r\dot{\phi} \begin{bmatrix} 1 \\ \dfrac{\sin\beta}{L_2} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \dot{\phi} \\ \beta \end{bmatrix} = \begin{bmatrix} \dfrac{v}{r} \\ \sin^{-1}\dfrac{L_2\dot{\theta}}{v} \end{bmatrix}$$

There are several issues with the simple design illustrated above. During a turn the left and right wheels travel different arcs meaning different distances, Fig. 5.25. This will cause the wheels to skid if their rotation rates are the same. Part of the solution is to place a differential in the axle to deliver power and allow for different wheel speeds. The other part is to allow for differential steering with the Ackerman design. The Ackerman steering overcomes the issue of side slip due to the outside wheel traveling farther than the inside wheel.

Some history here is interesting. The invention is claimed by Georg Lankensperger (Munich) in 1817. However his agent, Rudolf Ackerman, filed the patent and now has name credit. But, this steering system was described 50 years earlier by Eramus Darwin (the grandfather of Charles Darwin) in 1758 according to Desmond King-Halle in 2002 and Mr. Darwin has claim to the invention.

To satisfy the constraint placed on by the ICR, the steering system must satisfy the Ackerman equation:

$$\cot\theta_R - \cot\theta_L = \frac{2L_1}{L_2}$$

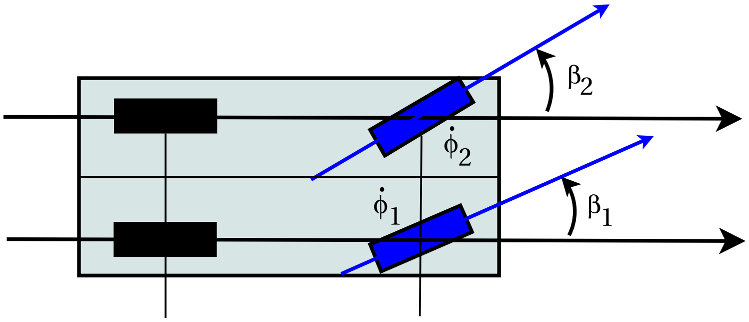


Fig. 5.25: To avoid skidding, the outside wheel must turn at a different angle and rotate at a different speed than the inside wheel.

where θ_R is the angle of the right wheel, θ_L is the angle of the left wheel, $2L_1$ is axle length and L_2 is the wheel base length, Fig. 5.26. The effective steering angle, θ_S can be found by

$$\cot\theta_S = \frac{L_1}{L_2} + \cot\theta_R \quad \text{or} \quad \cot\theta_S = \cot\theta_L - \frac{L_1}{L_2}$$

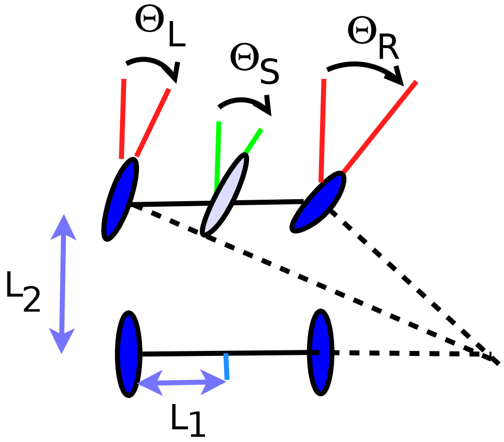


Fig. 5.26: The steering angles for the Ackerman equation.

The Ackerman design is one that approximates the geometric constraints which produces the ICR. A purely mechanical solution is to embed the geometry into the steering linkage. A triangle is formed from the attachment points at the wheels and the center of the rear axle. By moving the rear axle intersection, one can steer the wheels as well as keep the zero motion lines intersecting on the rear axle. The attachment to the wheels is called the *kingpins*. The cross piece between the Kingpins is called the *tie rod*.

Other Steered Wheel

As you delve into robot drive systems you begin to see that there are many different ways that people have mounted wheels onto frames and figured out how to steer the craft. We can only touch on a few designs in this text and encourage the reader to look beyond this text. It can be very entertaining to experiment with

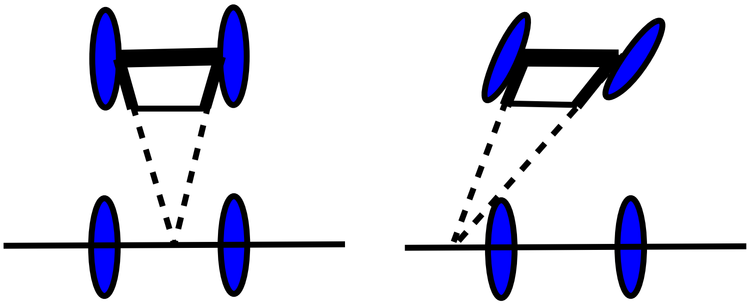


Fig. 5.27: The Ackerman steering system.

different wheel and frame designs. Using components like Actobots (https://www.servocity.com/actobotics), Lego, or Vex one can quickly assemble nearly anything that your mind can dream up. One novel approach to all wheel steering is the Syncro Drive system Fig. 5.28. Using three or four steered wheels, the wheels are connected by a chain or cable allowing all wheels to be steered. Each wheel is kept in a parallel mode so that motion is possible in any direction.

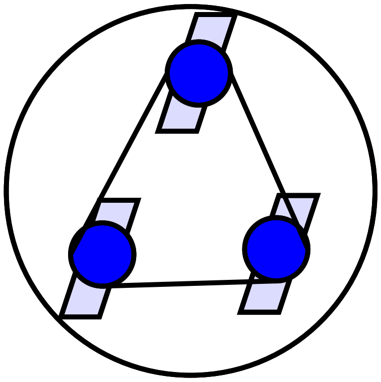


Fig. 5.28: Syncro Drive System.

5.4.5 The Dubins, Reeds-Shepps Cars and other drive systems

We investigate two vehicle designs which have a similar mechanism for steering. The first design consists of two axles with four driven wheels. The centers of the axles are attached to the frame of the robot using a lockable pivot. In essence, it is two differential drives attached to a bar with the pivot mechanism (see Fig. 5.29). We will refer to this as the Dual Differential Drive (DDD). The second design uses four axles (or one can think of splitting the axles in the DDD design) each with a driven wheel. The axles are attached to the body of the robot again using locking pivots, Fig. 5.30. We will focus on attachment points at the corners of the vehicle but other locations such as along the center line at either end of the robot would also be possible. For this design, mounting the pivots at the center of the axle or at the corners of a chassis has the effect of changing the number of pivot brakes and the costs, but does not significantly change the kinematics. This configuration will be known as the Four Wheel Steer (FWS). A traditional articulated steering design is shown in Fig. 5.31. The kinematics and motion curves for this design are essentially the same as the DDD design, and as such will be treated as a DDD steering mechanism.

When a wheel motor is activated, it will cause the axle to rotate about the pivot. Once the desired angle is achieved, the pivot is locked leaving the wheels in the steered configuration. The pivot joints are binary

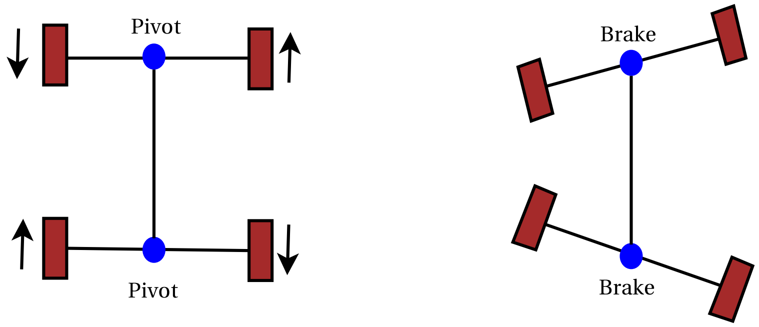


Fig. 5.29: Dual Differential Drive (DDD). This vehicle has single or connected axle in the front and a single axle in the rear. The axle is connect to the frame using a pivot which can be locked (braked) or free.

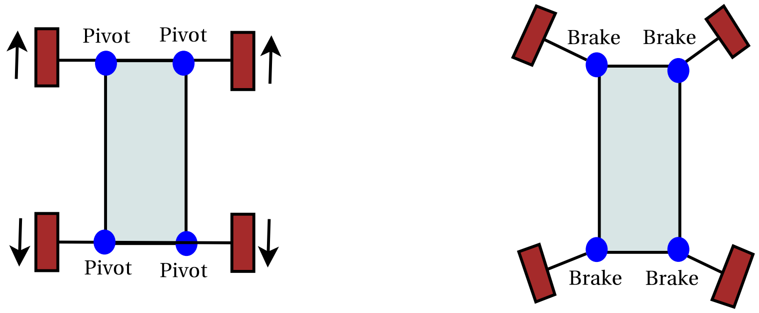


Fig. 5.30: Four Wheel Steer (FWS). This vehicle has four axles each is connected to the frame by a lockable pivot. In addition to motion see in the DDD design, if there is sufficient rotational motion in the axles, this conifguration can spin in place.

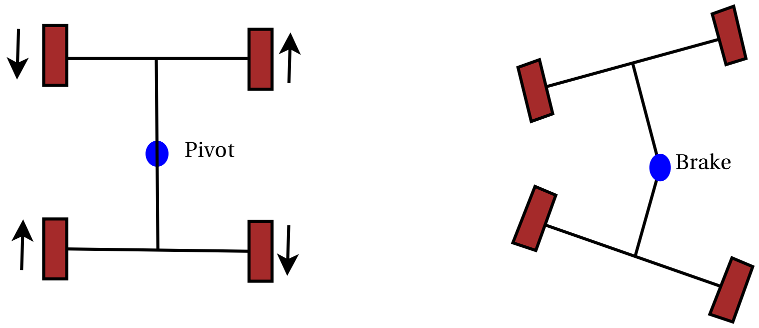


Fig. 5.31: Articulated Drive (AD). This is a common design in heavy equipment like articulated front loaders. The motion is similar to that found in the DDD design and can be driven with an unlocked pivot (brake not required).

in the sense that they are completely locked or completely free. This is done by a normally closed brake attached to the pivots and will allow free motion when power is applied to the pivot brake. When power is interrupted, the pivot brake locks down. Expected initial operation of the test unit is to alternate between a fixed position while aligning wheels and vehicle motion with the pivot brakes locked.

In terms of movement in the plane, the solid axle system is a dual differential drive. For the purposes of understanding motion curves we can view it as a two wheel (bicycle) design. Since we use four drive motors there is no need for a differential. The FWS axle mounted on the box can emulate Ackerman steering and does not suffer from wheel slip or slide. We will see that this design has greater maneuverability in comparison to a double Ackerman steered vehicle. In either case, we have two situations with a moving vehicle: driving straight paths and circular paths. Not found in Ackerman systems, the FWS design can additionally rotate in place if the axle is allowed to rotate out 45° or more.

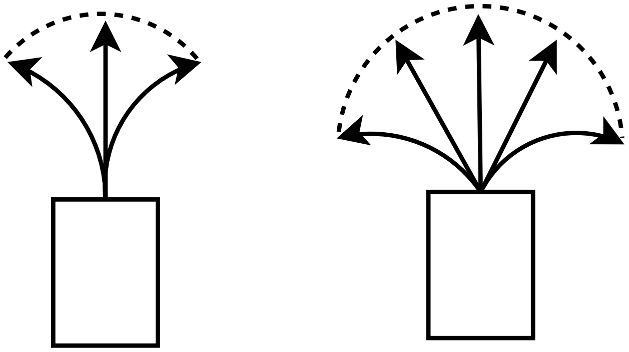


Fig. 5.32: The forward motion curves. Left: traditional Dubins Car. Right: forward motion of the DDD vehicle.

For the DDD design, using the bicycle approximation, the radius of curvature is given as a function of the maximum axle rotation and the wheelbase. Let the axle turn angle be θ and the wheel base given by d, then the radius of curvature is given by

$$r = d/(2\sin\theta)$$

Fig. 5.33 (left). In addition, the DDD can move linearly in directions angled off the forward direction of the vehicle if the axles are parallel and have nonzero axis angle in reference to the forward vehicle normal Fig. 5.32. The direction off of the forward normal direction is given by the axle angles and if the front and rear axles are not parallel, then a circular path will occur with direction off of the forward direction as seen with parallel axles.

The FWS design can adjust to the radius of curvature for both inside and outside wheels. The radius of curvature for the vehicle center is the average of the inside and outside circle radii:

$$\overline{r} = (r_1 + r_2)/2 = d\,(1/(4\sin\theta_1) + 1/(4\sin\theta_2))$$

Fig. 5.33. For this design, we have the ability to move as with the DDD and in addition rotate in place. Both systems can also move forwards and reverse. Thus orientation and direction may be changed at any point along the trajectory.

For the DDD, there are four motors (with associated electronics) and two pivot brakes (and associated electronics). The FWS design adds two pivot brakes in addition to the DDD cost. The operating assumption here is that mechanical holding torque can be gained more cheaply than electrical turning torque. The term

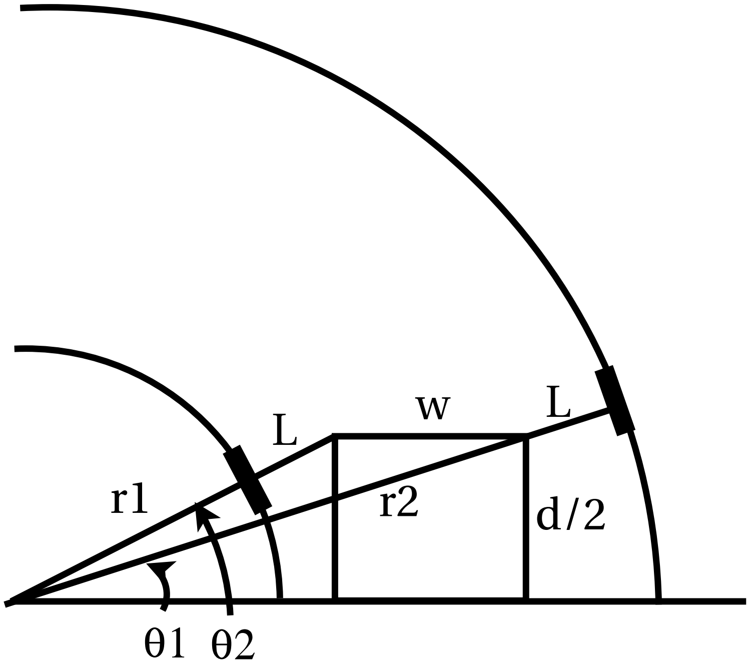


Fig. 5.33: Turn geometry for the DDD (left) and FWS (right) designs.

“cheap” will refer to dollar cost or to electrical power depending on the context. The dollar cost range for motors, motor drivers, electromagnetic brakes, etc., varies greatly. In our application, we found the prices to be fairly close between brakes and motors but the prices for driving electronics was significantly cheaper for the brakes as they operate like solenoids and the more complicated motor driver hardware was not required.

We have found that we don’t need a brake for the DDD system which removes both financial and electrical costs associated with the eliminated systems. For the FWS system, we can purchase a normally locked brake. Power is applied only when adjustments are required thus removing the need for holding current.

5.5 Configuration space and reach for simple vehicles

In this section, we determine the reach (and time limited reach) of the robot from a given point and the possible paths between two points. Does the reach cover the plane or are there some points in the plane which cannot be reached? First, we make precise what is meant by reach [LaV06]. Let X be the state space, $\mathcal{U} \subset X$ be the set of all permissible trajectories on $[0, \infty)$ and $R(q_0, \mathcal{U})$ denote the reachable set from x_0.

We define the reachable set as

$$R(x_0, \mathcal{U}) = \{x_1 \in X | \exists \tilde{u} \in \mathcal{U} \text{ and } \exists t \in [0, \infty) \text{ s.t. } x(t) = x_1\}$$

Let $R(q_0, \mathcal{U}, t)$ denote the time-limited reachable set from x_0.

We define the time-limited reachable set as

$$R(x_0, \mathcal{U}, t) = \{x_1 \in X | \exists \tilde{u} \in \mathcal{U} \text{ and } \exists \tau \in [0, t] \text{ s.t. } x(\tau) = x_1\}$$

The Dubins Car, [Dub57], is a vehicle that can move straight forward or turn at any curvature up to some maximum curvature. This vehicle provides a geometric motion model for automobiles and can be used to understand basic optimal path planning. The Reeds-Shepps Car, [RS90], extends the Dubins vehicle to include reverse motion. This greatly enhances maneuverability. Small back and forth motions can realign a vehicle to a new orientation. This means if the robot arrives at a destination point with the wrong orientation, it can be corrected locally (assuming sufficient room about the point).

Dubins showed that a vehicle which can go only forward and turn at any curvature up to some maximum curvature can reach any point in the plane in the absence of obstacles [Dub57]. Optimality of solutions is discussed in [Kel13], [LaV06]. A slight generalization is given in [RS90] for a car that can go forwards and backwards. In [RS90], [ST91], [LaV06], it is shown that optimal solutions are piecewise collections of line segments and maximum curvature circles. Since the DDD (dual differential drive) and FWS (four wheel steer) designs have less restrictive motion, we can answer the reach question. The entire plane can be covered. The question of optimal paths will be left for a future study. The FWS system we have built is targeted for an environment filled with obstacles. Our main concern is reach in the presence of obstacles, for which the reach and the optimal path results for Dubins and Reeds-Shepps are no longer valid.

Both the DDD and FWS designs are more maneuverable than the Dubins vehicle, and so we expect more flexibility in dealing with obstacles. The time limited reach of the Dubins Car is the forward fan seen in Fig. 5.32 and the time limited reach of the Reeds-Shepps car is the open set about the initial point [LaV06]. Since both the DDD and FWS systems include the motion patterns found in the Reeds-Shepps car, the time limited reach for these two designs is an open set about the initial point: there exists a set U, open, such that $U \subset R(x_0, \mathcal{U}, t)$. This is possible due to the ability to perform back and forth maneuvers like that found in parallel parking.

5.5.1 Rigid Motion

The FWS can move from point to point and then adjust orientation as required. If there exists a path between two points, the FWS axle can traverse the path via the waypoints, re-orient at each point and reach the goal location. Thus it can follow a piecewise linear path between two configuration space locations. A smooth path can be found by using a b-spline and if curvature exceeds the maximum bound, the vehicle can stop, re-orient and then continue. Traversal is possible if the start and goal locations are path connected and that path locations with curvature above R have a disk of radius r centered at the path point which does not intersect any obstacle.

The DDD design has additional constraints compared to the FWS design. The solution that [RS90], [ST91], [LaV06] suggest is to perform a series of short adjustment maneuvers as seen in Fig. 5.34. Although the results for re-orientation can be applied to arbitrarily small robots and adjustment regions, in practice for a given robot or vehicle, the region has some minimum size. Assume that the adjustment maneuvers falls in a circle of radius r. Let W be a bounded domain in $\mathbb{R}^2$, the obstacles be $\mathcal{O}_i$ and the free space be given by $\Omega = W \setminus \cup_i \mathcal{O}_i$.

For simplicity here, we assume the domain satisfies a traversability condition. Let $D(x, r)$ be the disk of radius r centered at x. Ω is said to be disk traversable if for any two points $x_0, x_1 \in \Omega$, there exists a continuous function $p(t) \in \mathbb{R}^2$ and $\epsilon > 0$ such that $D(p(t), \epsilon) \subset \Omega$ for $t \in [0, 1]$ and $x_0 = p(0)$, $x_1 = p(1)$. Note that $p(t)$ generates the curve C which is a path in Ω and the path is a closed and bounded subset of Ω. Navigation along jeep trails, bike trails and large animal trails (in our case, Cattle and Bison) produces small corridors though the forest. Along these tracks there is a corridor produced which we describe as disk traversable.

Traversability Theorem: If Ω is disk traversable, then the DDD and FWS vehicles can navigate to the goal

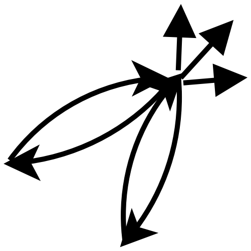


Fig. 5.34: A series of short adjustment maneuvers to re-orient the vehicle.

ending with the correct orientation. **Proof:** See *Appendix*.

5.6 The Piano Movers Problem - Orientation

Assume you want to route an object with a complicated shape through a tight sequence of corridors. Routing a complex shape through a narrow passage is often referred to as the piano movers problem. Take a simple example, move the linear robot through the two blocks, Fig. 5.35. It is clear to the human what has to happen. The robot must rotate. For a holonomic robot, this simply means the controller issues a rotation command while traveling to the corridor. For a non-holonomic robot, the control system must change the path so that upon entry and through the corridor the robot's orientation will allow for passage. A significant problem arises if the corridor is curved in a manner that is not supported by the possible orientations defined by the vehicle dynamics. In plain English, this is when you get the couch stuck in the stairwell trying to move into your new flat.

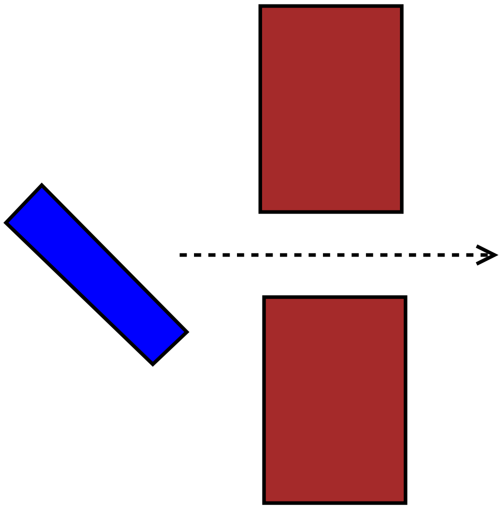


Fig. 5.35: The object must rotate to fit through the open space.

As all of us learned when we were very young, we must turn sideways to fit through a narrow opening.[1] This introduces a new aspect to routing, that of reconfiguration of the robot. Examine a simple reconfiguration which is simply a change in orientation. As we saw above, each rotation of the robot induces a different

[1] Cavers will tell you that you can crawl through a vertical gap spanned by the distance of your thumb and your fifth (pinky) finger. For the average American, this is a very small gap.

configuration space. Fig. 5.36 shows the idea for three different rotation angles, there are three different configuration obstacle maps.

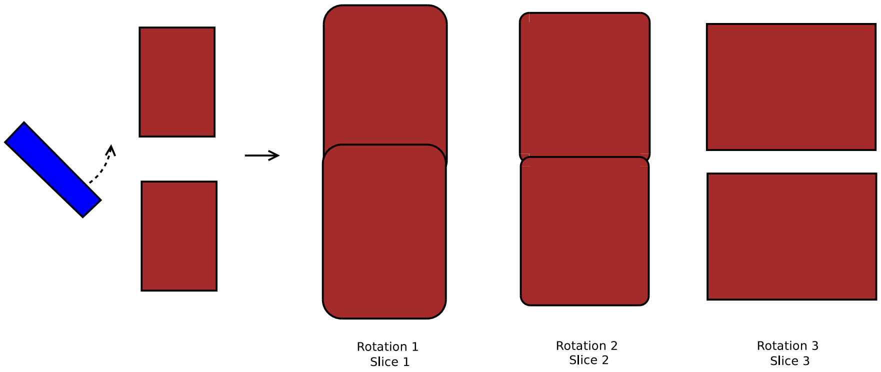


Fig. 5.36: Different rotations produce different obstacle maps in configuration space.

Since each rotation generates a two dimensional configuration space, they can be stacked up in three dimensions. So we have that configuration space includes the vertical dimension which is the rotation angle for the robot - the configuration space is three dimensional. To restate, the configuration space includes all of the configuration variables (x, y, θ) is now a three dimensional configuration space which is shown in Fig. 5.37. So, although the workspace is two dimensional, the configuration space is three dimensional and are different objects.

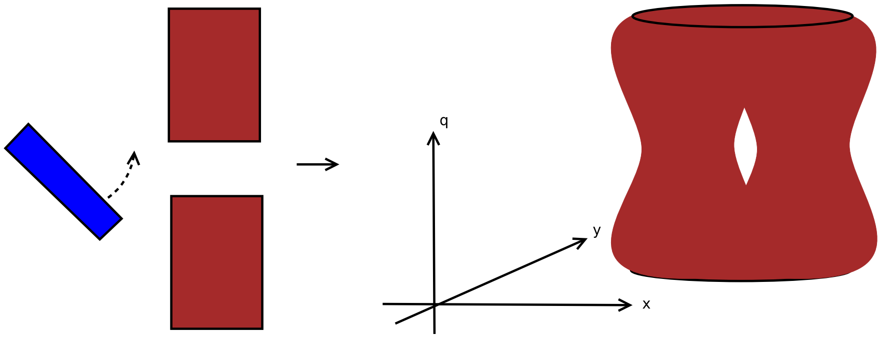


Fig. 5.37: The different rotations can be stacked where the vertical dimension is the rotation angle.

For a three dimensional object with a fixed orientation, would have a three dimensional configuration space. For toolheads, only pitch and yaw matter. To locate a point on a sphere you need two variables (think about spherical coordinates): θ the angle in the x-y plane and ϕ the angle from the z axis (or out of the plane if you prefer). For each pair (θ, ϕ) we have a 3D section. This tells us that the configuration space is five dimensional. When roll, pitch and yaw all matter then we have a 6 dimensional configuration space. If the robot is configurable with other elements, then each parameter defining the configuration would also add a variable to the mix and increase the dimension of the configuration space.

The construction of configuration space then is built like slices in a 3D printer. Routing or path planning must be done in the full configuration space. For the current example, we must route in 3D which will translate to position and orientation routing in the workspace, Fig. 5.38.

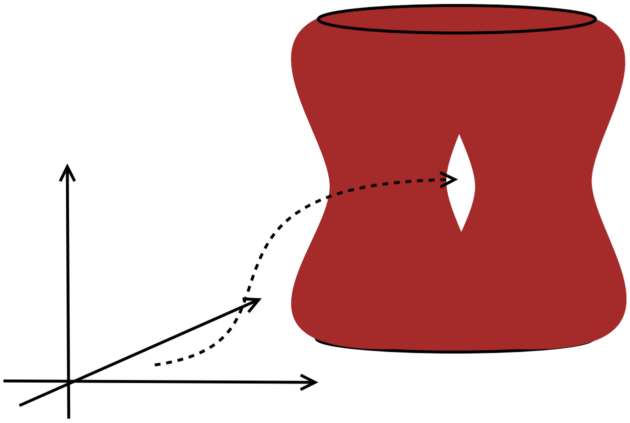


Fig. 5.38: We can see that there is a path that includes the rotation.

5.6.1 Two Link Arm Revisited

Articulated (multilink) robot arms also have size and orientation. Determining which configurations and which physical positions are actually realizable is more complicated. The size of the robot arm will affect the regions which the end effector can reach but obstacle inflation does not give the same workspace. The end effector is designed to touch an object and from that perspective little inflation is required. However the base link of the arm might be very wide and does affect the useable workspace. A simple obstacle inflation approach will not work with manipulators. The reason is that how you travel affects your reach. Fig. 5.39 shows how the path matters to access. A more situation can be found in Fig. 5.40. Even though the articulator is small enough to pass through the gap, it cannot due to the other physical restrictions.

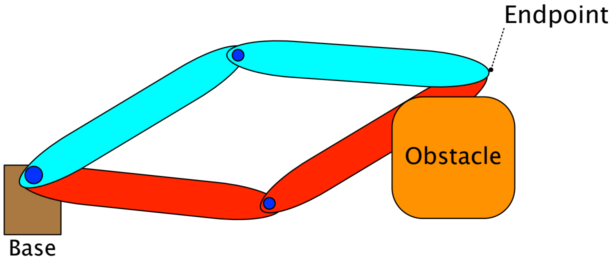


Fig. 5.39: The elbow down approach is blocked, but not the elbow up position.

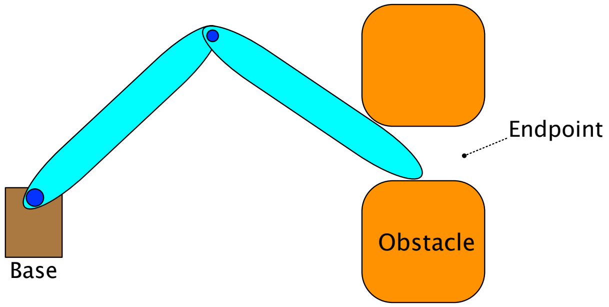


Fig. 5.40: Neither configuration of the robot arm can reach the point.

5.7 Appendix

The proof for the *Traversability Theorem*, statement reproduced below, is given here.

If Ω is disk traversable, then the DDD and FWS vehicles can navigate to the goal ending with the correct orientation.

Proof: Let C be the path from x_0 to x_1. At each point of the path there exists an open disk of radius ϵ which does not intersect an obstacle. The intersection of the curve C with the open disk induces an open set in C. The collection of open sets is an open cover of the curve C. Since the curve is a closed and bounded set, and thus compact, there is a finite subcover of open intervals [Mun00]. These correspond to a finite set of open disks which cover the path. The vehicle may travel a straight line from disk center to disk center. At each center the vehicle may reorient if required. The time limited reach for the DDD drive is a proper subset of the FWS reach, and follows from the DDD result.

5.8 Problems

1. Write a Python function to compute wheel angles in the Ackerman system given the desired vehicle turn angle and frame parameters. This the function should have arguments *(theta, l1, l2)* and function should return the two wheel angles *(theta_l, theta_r)*.

2. What are the motion equations for the Ackerman drive? [Meaning forward and angular velocity as a function of wheel speed.] Assume wheel radius is r.

3. A dual Ackerman drive would steer both front and rear wheels using an Ackerman steering approach. What would the pros and cons for this system compared to a single Ackerman drive?

4. Assume that you have a rectangular Mechanum robot with $L_1 = 0.30$m, $L_2 = 0.20$m and $r = 0.08$m. Find the path of the robot for the given wheel rotations: $\dot{\phi}_1 = 0.75*\cos(t/3.0)$, $\dot{\phi}_2 = 1.5*cos(t/3.0)$, $\dot{\phi}_3 = -1.0$, $\dot{\phi}_4 = 0.5$. Start with $x, y, \theta = 0$ and set $t = 0$, $\Delta t = 0.05$. Run the simulation for 200 iterations (or for 10 seconds). Keeping the x and y locations in an array is an easy way to generate a plot of the robot's path. If x, y are arrays of x-y locations then try

   ```
   import pylab as plt
   plt.plot(x,y,'b.')
   plt.show()
   ```

   Showing the orientation takes a bit more work. Matplotlib provides a vector plotting method. You need to hand it the location of the vector and the vector to be plotted, (x, y, u, v), where (x, y) s the vector location and (u, v) are the x and y components of the vector. You can extract those from θ using $u = s * \cos(\theta)$ and $v = s * \sin(\theta)$ where s is a scale factor (to give a good length for the image, e.g. 0.075). The vector plot commands are then

   ```
   plt.quiver(u,v,c,s,scale=1.25,units='xy',color='g')
   plt.savefig('mecanumpath.pdf')
   plt.show()
   ```

5. Real motion and measurement involves error and this problem will introduce the concepts. Assume that you have a differential drive robot with wheels that are 20cm in radius and L is 12cm. Using the

differential drive code (forward kinematics) from the text, develop code to simulate the robot motion when the wheel velocities are $\dot{\phi}_1 = 0.25t^2$, $\dot{\phi}_2 = 0.5t$. The starting location is [0,0] with $\theta = 0$.

a. Plot the path of the robot on $0 \leq t \leq 5$. It should end up somewhere near [50,60].

b. Assume that you have Gaussian noise added to the omegas each time you evaluate the velocity (each time step). Test with $\mu = 0$ and $\sigma = 0.3$. Write the final location (x,y) to a file and repeat for 100 simulations. Hint:

```
mu, sigma = 0.0, 0.3
xerr = np.random.normal(mu,sigma, NumP)
yerr = np.random.normal(mu,sigma, NumP)
```

c. Generate a plot that includes the noise free robot path and the final locations for the simulations with noise. Hint:

```
import numpy as np
import pylab as plt
...
plt.plot(xpath,ypath, 'b-', x,y, 'r.')
plt.xlim(-10, 90)
plt.ylim(-20, 80)
plt.show()
```

d. Find the location means and 2x2 covariance matrix for this data set, and compute the eigenvalues and eigenvectors of the matrix. Find the ellipse that these generate. [The major and minor axes directions are given by the eigenvectors. Show the point cloud of final locations and the ellipse in a graphic (plot the data and the ellipse). Hint:

```
from scipy import linalg
from matplotlib.patches import Ellipse
s = 2.447651936039926
#  assume final locations are in x & y
mat = np.array([x,y])
#  find covariance matrix
cmat = np.cov(mat)
# compute eigenvals and eigenvects of covariance
eval, evec = linalg.eigh(cmat)
r1 = 2*s*sqrt(evals[0])
r2 = 2*s*sqrt(evals[1])
#  find ellipse rotation angle
angle = 180*atan2(evec[0,1],evec[0,0])/np.pi
# create ellipse
ell = Ellipse((np.mean(x),np.mean(y)),r1,r2,angle)
#  make the ellipse subplot
a = plt.subplot(111, aspect='equal')
ell.set_alpha(0.1)     #  make the ellipse lighter
a.add_artist(ell)    #  add this to the plot
```

6. Describe the different styles of Swedish wheel.

7. Find the analytic wheel velocities and initial pose for a Mecanum robot tasked to follow ($r = 3$, $L_1 = 10$, $L_2 = 10$ all in cm) the given paths (path units in m). Plot the paths and compare to the

actual functions to verify.

a. $y = (3/2)x + 5/2$

b. $y = x^{2/3}$

8. What are the wheel velocity formulas for a four wheel Mechanum robot, ($r = 3$, $L_1 = 10$, $L_2 = 10$ all in cm) which drives in the circular path $(x - 3)^2/16 + (y - 2)^2/9 = 1$ and always faces the center of the circle.

9. In Veranda, drive a Mecanum robot along a square with corners (0,0), (10,0), (10,10), (0,10), $L_1 = 0.30$, $L_2 = 0.20$ and $r = 0.08$. You should stop and "turn" at a corner, but keep the robot faced in the x-axis direction. Drive the edges at unit speed. Use a video screen capture program to record the results.

10. In Veranda, drive the Mecanum robot in an infinity (∞) shape. Use a video screen capture program to record the results.

CHAPTER

SIX

REAL SYSTEMS

6.1 Real Robots

Up to this point we have focused exclusively on robotics as either an abstract mathematical concept or within computer simulation. This section hopes to home in on the more concrete side of robotics: physical implementation. The act of transitioning from the notepad or simulation to a moving, physically-present system is not to be glossed over as a footnote. In the vast majority of cases a control scheme that works on a computer screen falls apart when moved into the real world.

6.1.1 Limitations of simulation:

It is impossible to overstate the importance of simulation to the field of Robotics. Simulation allows us to test new path-planning algorithms from our living rooms and run stochastic experiments hundreds of times within minutes. However, for simulation to be useful as a tool a few concessions must be granted.

Physics models are often reduced and linearized projections of the incredibly-complex real system. A classic example of this is the inverted pendulum, where the non-linear system model is routinely linearized certain set point and very real parameters like link inertia and joint friction are omitted.

Similarly sensors in simulation land are often idealized: your LiDAR can measure from 0.1cm to 100m, or the output of the infrared distance sensor on your simulation bot is not affected by the reflectance of the sensed object. In reality these sensors are not deterministic and will often exhibit strange behavior even in optimal conditions.

Simulations also ignore communications challenges. Communication between a central controller and a motor controller, for example, can take a non-zero amount of time and introduce a delay from issued command to system response. Networking tirelessly often presents throughput and latency challenges, something never seen in simulation. By and large simulation is an incredibly powerful tool that allows us to prove a concept, but it's important to understand that simulation is not a perfect representation of a physical system.

6.1.2 Moving to the real world:

Like many things in life, the first step in taking a project from the drawing board to reality is defining the purpose of the project. For what task is your robot being built? How is your robot to complete this task? What constraints are placed on the physical design of the system? Once a purpose has been decided on and constraints identified discussions of system complexity can take place. How complex must the robot be to complete its task?

A Roomba, for example, is designed exclusively to sweep-vacuum your floor. Such a system does not need a GPU-accelerated stereo-vision SLAM localization system to adequately traverse a room. Early Roombas simply used bump sensors on the exterior perimeter and random motion to produce satisfactory results. However, the aforementioned localization system might be necessary in an eight-legged robot developed for navigating the interior of a destroyed nuclear reactor.

Several practical considerations must be accounted for as well. In simulation we are given infinite control authority over our robotic system; in reality, motors have a hard limit to the amount of torque they can apply to a shaft. Batteries can only supply so much current at once. Drivetrain slop could effect wheel telemetry if calculated from the input motor side. Designing a robot for the challenge at hand is often more art than science, but always starts with a discussion of required capability.

6.1.3 Structure of a 'real' robot:

Discussion of what a robot is or is not can be found elsewhere in this text. For this context we'll look at what constitutes the physical embodiment of our existentially indeterminate robots. Surveying a broad sample of robots, from Roombas to Willow Garage's PR2, it is easy to see that there are many shared architectural elements through the spectrum. The vast majority of robots will have

Locomotion

> One of the defining features of mobile robotics is the robot's ability to traverse its environment. This environment might be the floor of your bedroom, the airspace of your gymnasium, or even the walls of your office. Wheels are the most common form of locomotion due to their ease of implementation and extremely low complexity. Systems with articulated legs are very complex and extraordinarily difficult to control intelligently. Such systems have only recently become reasonable for robot locomotion due to increases in low-power, high-output computing.

Actuation

> In most cases a robot can affect its environment or internal state in some way. This is often accomplished by using actuators of some kind to turn, slide, or extend. For the vast majority of low-cost robotics these actuators are electrically powered, manifesting in the form of hobby servos, DC motors, steppers, or solenoids. As the budget increases high-power electric systems like three-phase AC servomotors or pneumatic pistons are common.

Power system

> Motors need power to turn. All robots have some form of power distribution system to deliver correct voltage levels and supply current to every piece of hardware on the robot. Most commonly the power source is a battery bank (usually LiPo) that is then regulated down and distributed through a wiring harness. Some bots, like Boston Dynamics' Big Dog contain a small gasoline-powered generator to produce power remotely.

Sensors

In order to make decisions robots must have some way of taking input from their environment. At a base level a sensor does one thing: transform environmental stimulus into a form the robot's computational hardware can understand. A sensor could be as complicated as an RGBD camera returning a point cloud or as simple as a bump switch. In the past decade LiDARs have become the go-to sensor for most terrestrial robotics applications.

Compute hardware

Critical to a robot's functionality is the ability to process incoming information and decide on a course of action. For complex functionality this usually necessitates serious computing horsepower; for such problems low-power, compact versions of desktop computing hardware are available. For most hobby, competition, and research work a dedicated single-board computer running a utilitarian Linux distribution provides enough number-crunching and interconnect capability. For simple robots a microcontroller may be all the processing power necessary. Very complex systems (i.e. DARPA challenge bots) almost exclusively offload the processing of sensor information and decision-making to off-site computing hardware.

Real-time controllers

Even though a dedicated onboard computer might provide direction and high-level control of a robot, many tasks associated with robot operation require the singular attention of a sub-system controller. An SBC running a Linux OS does not have the real-time capability necessary to generate the waveforms driving hobby servos, or to keep track of every single tick of a wheel encoder. To handle these tasks dedicated controllers are often used. One example of this is a motor controller that takes in a target rotational speed via UART and monitors a shaft encoder, increasing or decreasing motor current as necessary to maintain a constant shaft speed.

6.2 Trossen GeekBots

In the course of this text we have seen several different robot topologies but have focused a significant amount of time on the simple differential-drive robot. To accompany the text a basic differential-drive robot has been developed and deployed. The donor chassis for this specific design is the (now defunct) RobotGeeks GeekBot, from which we will appropriate a name. The GeekBot as constructed for use with this text has been developed with cost in mind: total parts cost comes in at just below $300. A 3D printer was used to print one mount for the SBC as well as a mount for the infrared sensor. All files will be made available online for the aspiring roboticist.

The purpose of the SDSMT GeekBot is to teach the basics of robot control using ROS on physical hardware. To accomplish this task the familiar RobotGeeks GeekBot platform was used as a mobile base and extra hardware added as needed.

6.2.1 Hardware:

Locomotion

The GeekBot comes stock with two drive wheels, powered by 6v continuous-rotation hobby servos. Two caster balls keep the bot vertical with a bit of wobble. These motors are not encoded, meaning they only have relative speed adjustment. The actual angular velocity of the wheels is unknown at any point other than stopped. Important: these motors are run directly off the unregulated input power supply. These motors will cook if supplied with more than ~8.5v.

Electrical

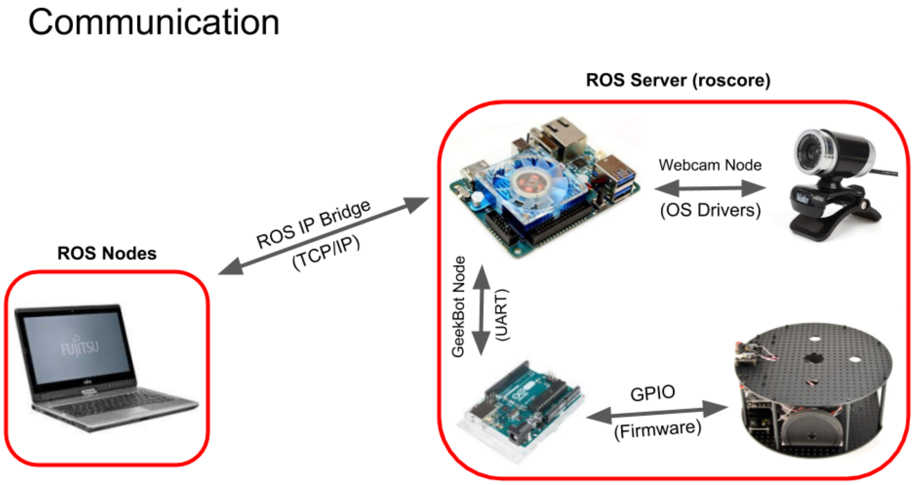
The GeekBot is supplied by an 8.4v 2.2Ah LiPo battery. This is 8.4v is passed straight to the drive motors, the onboard Arduino, and a 5v regulator for supplying the Odroid XU4. The harness switches power to the system, and a disconnect is located on the back to decouple battery power and allow the use of another <9v supply for testing.

Sensors

The GeekBot has two sensors: a front-mounted webcam and a servo-actuated IR distance sensor. The IR sensor is mounted under the front of the robot and centered forward; the full range of motion encompasses 0-180 degrees. This sensor returns a non-linear analog voltage corresponding to perceived distance that is passed to the onboard Arduino and processed. The mounted webcam is is both powered by and communicates over one of the Odroid's USB ports. This particular webcam is manual focus and captures at 640x480px, 30fps.

Compute and control

The GeekBot follows a standard distributed-control topology: a high-level controller issues commands to low-level controllers which handle command implementation for specific subsystems. In this case, the high-level controller is an Odroid XU4 running a minimal version of Ubuntu 16.04 LTS. All hardware interfacing tasks (driving servos, reading voltages, turning lights on) are controlled by the Arduino, taking commands from the Odroid via UART. Below you can see a quick graphic showing these interconnects.



You will note that there are many, many places for communication failures (cut wires, unplugged cables, etc.) in this setup. Unfortunately this is one of the downfalls of real robotics; communications will eventually fail and you will have to have systems in place to guarantee correct data transfer or robot sutdown. For now, and at our own risk, we'll assume that there will be no communication interruptions in the GeekBot system.

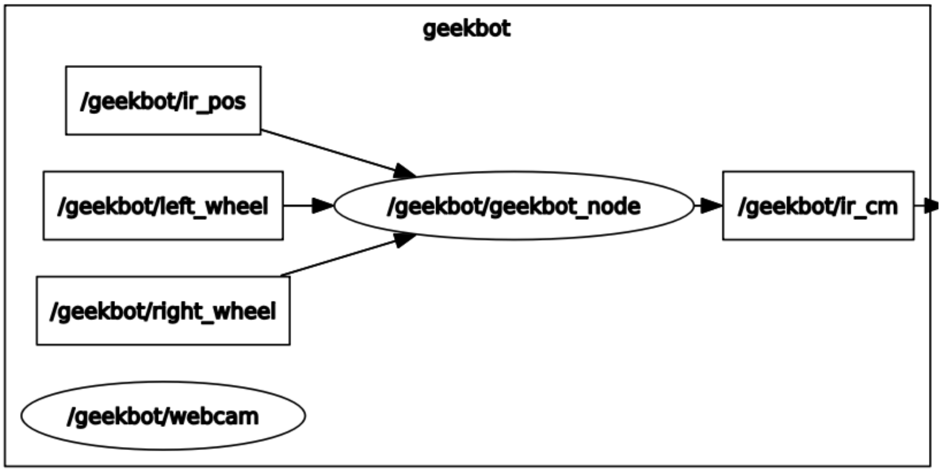
6.2.2 Software:

ROS Kinetic

Despite the rest of the text using ROS2 Ardent, the GeekBot runs ROS1 Kinetic. The reasoning behind this decision pertains mostly to the lack of documentation, compressed image transport, networking tools, and debugging tools in ROS2 as of publishing time. The concepts presented in this text for ROS2 are directly transferable to ROS Kinetic. Nodes run independently of each other and communicate via passing messages over topics, exactly like ROS2. ROS1, however, does have a supervisory piece of software called `roscore` which handles node connections and message direction.

Software architecture

On boot the GeekBot ROS subsystem starts up three things: roscore, for handling all communications; a node to handle webcam things (webcam), and another node to handle communication with the Arduino (geekbot_node). Both exist under the `/geekbot` namespace. Using ROS' node/topic mapping tool `rqt_graph` we can see the GeekBot's nodes, topics, and the namespace under which all of these exist.



Bringup on boot is handled by a service through `systemd` that points to a script contained in the `geekbot_pkg` package which resides within the `geekbot_ws` directory in the `/root` directory of the Odroid's filesystem. Handling bringup through `systemd` allows for synchronization with the networking stack as well as an easy start-stop-restart interface so the GeekBot's ROS system can be restarted with the robot powered up and online. Linking to a `systemd` service in a repository-held package gives the option for updates to bringup handling without requiring extensive filesystem rework. Using a `systemd` service to start ROS systems on boot is much less common than simply pushing a start script to `init.d`, `rc.local`, or `cron` and is certainly more work to set up. However, using `systemd` allows for much simpler logging/debugging, runtime adjustments to the underlying ROS subsystem, and (most importantly) hardware-specific startup criterion using `udev` rules to adjust targets. We won't get into implementing such things here.

Published topics

After being powered on and being assigned an IP address in the 10.42.0.X range the GeekBot will publish/subscripe to several topics. The topics we care about are as follows:

/geekbot/ir_cm

- Publishes: Int32
- Distance to nearest object in centimeters as seen by the IR sensor

/geekbot/ir_pos

- Subscribes: Int32
- Angle (0-180) at which to set the IR sensor's servo

/geekbot/left_wheel

- Subscribes: Int32
- Relative speed (-100-100) at which to set the left wheel. Negative for reverse

/geekbot/right_wheel

- Subscribes: Int32

- Relative speed (-100-100) at which to set the right wheel. Negative for reverse

/geekbot/webcam/image_raw

- Publishes: Image
- Raw image from the camera, with zero compression of any kind. Very large message

/geekbot/webcam/image_raw/compressed

- Publishes: CompressedImage
- Compressed frame from the camera, 85% JPEG quality. MUCH smaller than raw image

6.2.3 GeekBot Basics:

Initial setup

On a Ubuntu 16.04 LTS installation install ROS Kinetic alongside your ROS2 Ardent installation. Follow the instructions to install `ros-kinetic-desktop`. *HOWEVER*, do **not** add the excerpt as specified in step 1.6. Doing so will cause conflicts with your ROS2 Ardent installation. A few other packages will have to be installed to meet setup script dependencies and break out some image tools:

```
sudo apt install nmap ros-kinetic-image-view
ros-kinetic-image-common ros-kinetic-image-transport-plugins
ros-kinetic-cv-bridge
```

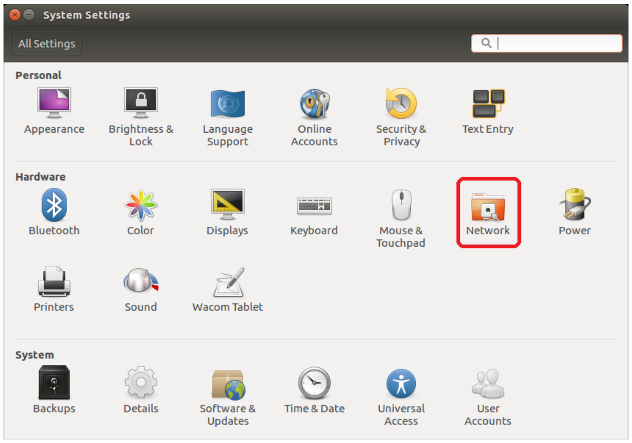
Next, clone the `geekbot_resources` repository found here to somewhere in your filesystem:

```
git clone https://github.com/sdsmt-robotics/geekbot_resources
```

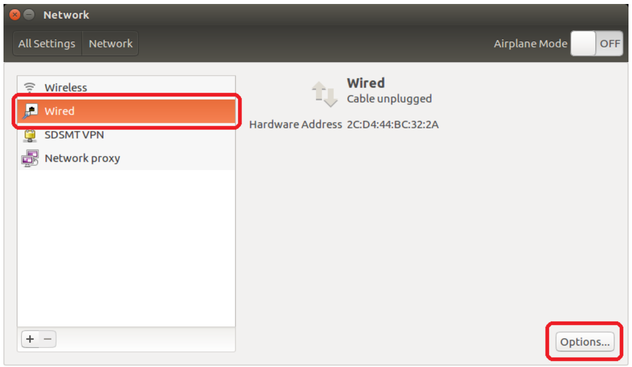
You should now have a `geekbot_resources` directory. This repository contains all client-pc information pertaining to GeekBot operation. Inside you'll find notes and handy examples as well as the Arduino code running on the GeekBot's onboard controller.

Configuring your Ethernet port

1. In Ubuntu's system settings, navigate to the 'Networking' section. You should see a list of network connections on the left side.



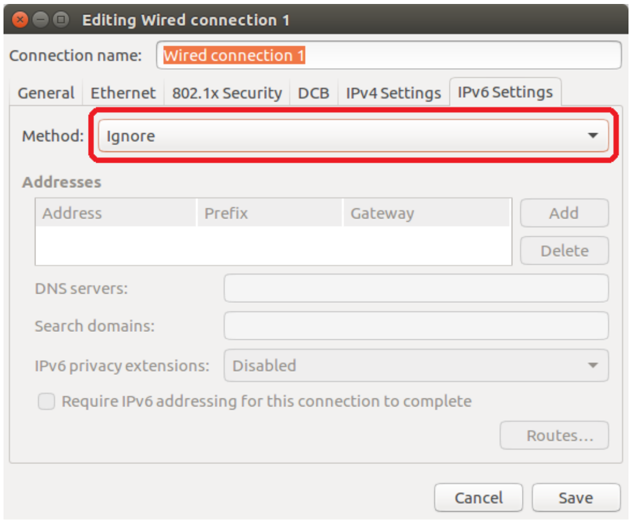
2. Select the wired network and in the lower-righthand side of the pane click 'Options'. Here we can change specific settings for how Ubuntu handles the Ethernet port of your computer.



Wired
Cable unplugged

Hardware Address 2C:D4:44:BC:32:2A

3. Click on the IPv6 tab. In the drop down, select 'Ignore'. We won't be using IPv6 to connect to the GeekBots.



Connection name: Wired connection 1

Method: Ignore

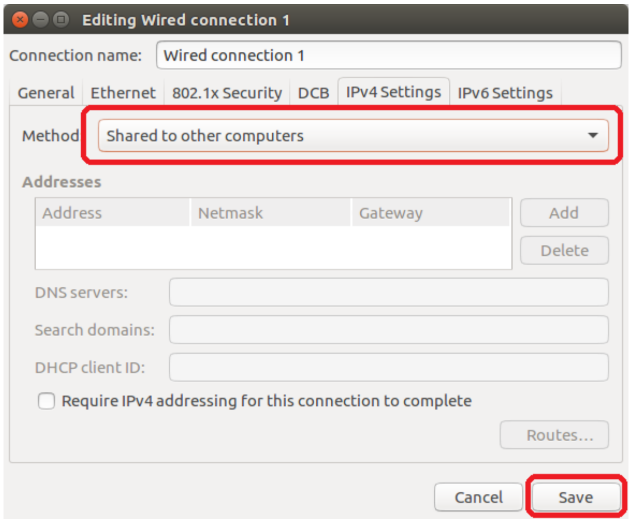
DNS servers:

Search domains:

IPv6 privacy extensions: Disabled

☐ Require IPv6 addressing for this connection to complete

4. Now select the IPv4 tab and choose 'Share to other computers' from the dropdown menu. In the lower right hand corner click 'Save'.



Editing Wired connection 1

Connection name: Wired connection 1

Method: Shared to other computers

The Ethernet port on your computer is now set to automatically assign an IP on 10.42.0.X spectrum to anything connected to it and requesting an IP address. This is the default state of the GeekBot, so if the GeekBot is connected to your computer then it will request and be assigned an IP in the 10.42.0.X range.

Connecting to the GeekBot

1. Connect an Ethernet cable between your computer and the GeekBot's Odroid.
2. Power on the GeekBot by flipping the switch in the left-rear of the bot outwards. The Odroid and Arduino should start flashing lights.
3. Wait patiently for the Odroid to boot. This should take ~30 seconds. When the Odroid has finished the booting process and has grabbed an IP from your computer, it will launch its ROS system and initiate communications with the onboard Arduino. If a successful connection is made *you will hear two beeps from the robot*.
4. Navigate to the geekbot_resources folder you cloned in the initial setup. Source the `geekbot_connect.source` file. This will use `nmap` to scrape the 10.42.0.X subnet looking for your bot, set the necessary environment variables, and automatically load in ROS Kinetic to this specific terminal instance.
5. If you see a list of topics print out to your terminal you have successfully connected! **You will have to follow step #4 for each terminal instance you would like to connect to the GeekBot.**

Shutting down the GeekBot

1. Flip the power switch in the left-rear of the bot forward, into the robot. If the power is off no lights should be on.

Charging the GeekBot

1. Locate your GeekBot battery charger. This is a wall-wart supply that has a ribbed back section, a little LED in the bottom left corner, a yellow tip, and an 8.4v 2A output.

2. Plug the charger into an outlet.
3. Locate the battery charging port on the front of the robot. This should be zip-tied down to the lower platform and will run directly into the battery.
4. Plug in the charger to the charging port. The light on the charger should become red. When fully charged, the light will turn green. These batteries have automatic over-voltage protection, so the charger can be left on the battery indefinitely.

Running the GeekBot from external power

1. Make sure the GeekBot is powered off.
2. Disconnect the battery from the 2x5.5mm splitter zip-tied to the rear-right vertical support on the bot. Connect a power supply from 7v-8.5v, or the battery charger provided with the robot.
3. If the charger for the battery is used be aware: this supply does not provide enough power to run both motors as well as intense computation on the Odroid. If your robot is intermittently losing connection when operating the motors with this supply you are most likely browning out the Odroid.

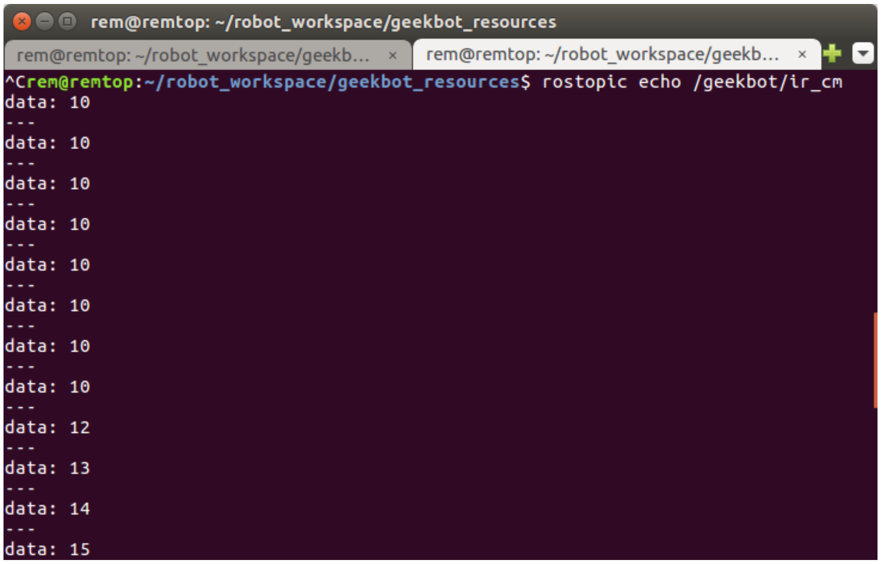
6.2.4 Using the GeekBot:

ROS' in-built tools

When debugging ROS-based systems it can be very handy to peek in on what data is being published on what topics by what nodes. We can accomplish this easily with ROS' in-built command line tools. In ROS1 these take the form of `rosxxxx` where `xxxx` roughly describes the useful area of ROS within which we want to operate. For example, listening in on a topic and pushing its published info to the screen can be accomplished by using the `rostopic` tool:

```
rostopic echo /geekbot/ir_cm
```

Running the above you should see something like this on your screen:



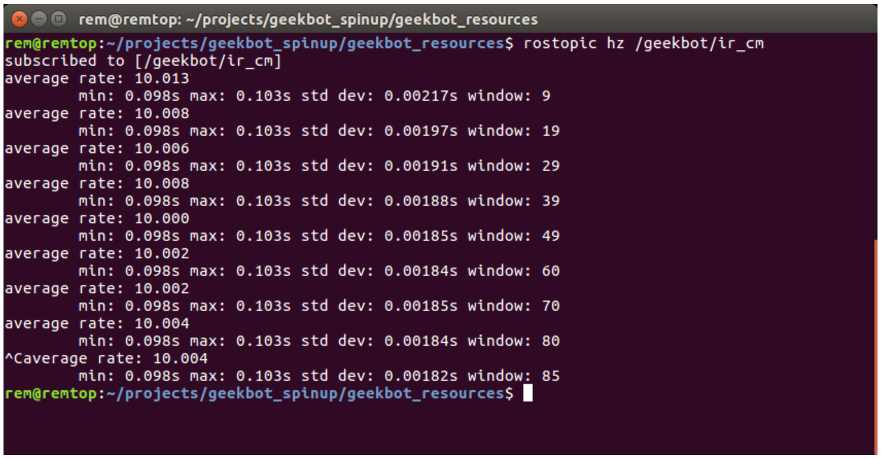
rem@remtop: ~/robot_workspace/geekbot_resources
rem@remtop: ~/robot_workspace/geekb...
rem@remtop: ~/robot_workspace/geekb...

```
^Crem@remtop:~/robot_workspace/geekbot_resources$ rostopic echo /geekbot/ir_cm
data: 10
---
data: 10
---
data: 10
---
data: 10
---
data: 10
---
data: 10
---
data: 10
---
data: 10
---
data: 12
---
data: 13
---
data: 14
---
data: 15
```

The data presented is the Int32 payload published to the topic `/geekbot/ir_cm`. Maybe we want to see how fast new information to this topic is being published. To do so we use the `rostopic` tool yet again:

```
rostopic hz /geekbot/ir_cm
```

You should now see the average publishing rate, calculated standard deviation for the publishing gap times, and also the number of samples the stats were generated from:



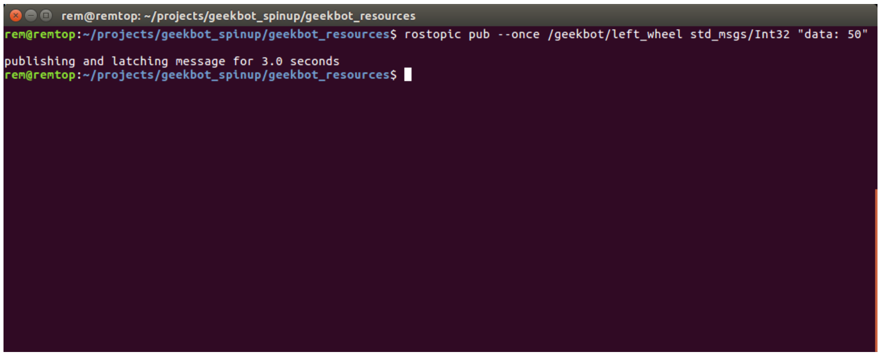
```
rem@remtop:~/projects/geekbot_spinup/geekbot_resources$ rostopic hz /geekbot/ir_cm
subscribed to [/geekbot/ir_cm]
average rate: 10.013
        min: 0.098s max: 0.103s std dev: 0.00217s window: 9
average rate: 10.008
        min: 0.098s max: 0.103s std dev: 0.00197s window: 19
average rate: 10.006
        min: 0.098s max: 0.103s std dev: 0.00191s window: 29
average rate: 10.008
        min: 0.098s max: 0.103s std dev: 0.00188s window: 39
average rate: 10.000
        min: 0.098s max: 0.103s std dev: 0.00185s window: 49
average rate: 10.002
        min: 0.098s max: 0.103s std dev: 0.00184s window: 60
average rate: 10.002
        min: 0.098s max: 0.103s std dev: 0.00185s window: 70
average rate: 10.004
        min: 0.098s max: 0.103s std dev: 0.00184s window: 80
^Caverage rate: 10.004
        min: 0.098s max: 0.103s std dev: 0.00182s window: 85
rem@remtop:~/projects/geekbot_spinup/geekbot_resources$ 
```

It looks like the IR data is coming in at exactly 10hz as expected. Neat!

We can also publish to a topic using the command line, using the `rostopic pub` command. Tab-complete is very handy here. If `rostopic` can't automatically find a list of topics you can manually list registered topics with `rostopic list` and find the topic type with `rostopic type`. Usually slapping the tab key will give you options or autofill. To publish a single message setting the left wheel speed of our GeekBot to 50% power, the command would look like the following:

```
rostopic pub --once /geekbot/left_wheel std_msgs/Int32 "data:
50"
```

This will publish to the topic one message with the payload as described:



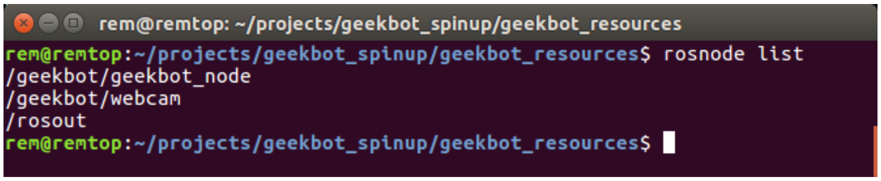
```
rem@remtop:~/projects/geekbot_spinup/geekbot_resources$ rostopic pub --once /geekbot/left_wheel std_msgs/Int32 "data: 50"

publishing and latching message for 3.0 seconds
rem@remtop:~/projects/geekbot_spinup/geekbot_resources$ 
```

Your GeekBot's left wheel should start spinning! Remember: since we only published a single message that wheel will keep spinning until we set the speed back to zero. For more complex messages, let tab-complete do the hard YAML work for you, then plug in your values. The following example call uses a six-field Twist message, which the Geekbots do not use:

```
rostopic pub --once /takes_a/twist_msg geometry_msgs/Twist \
    "linear:
   x: 1.3
   y: 0.0
   z: 0.0
    angular:
   x: 0.0
   y: 3.7
   z: 0.0"
```

We can also list the nodes currently running and tracked by the GeekBot's `roscore`:

```
rosnode list
```



```
rem@remtop: ~/projects/geekbot_spinup/geekbot_resources
rem@remtop:~/projects/geekbot_spinup/geekbot_resources$ rosnode list
/geekbot/geekbot_node
/geekbot/webcam
/rosout
rem@remtop:~/projects/geekbot_spinup/geekbot_resources$
```

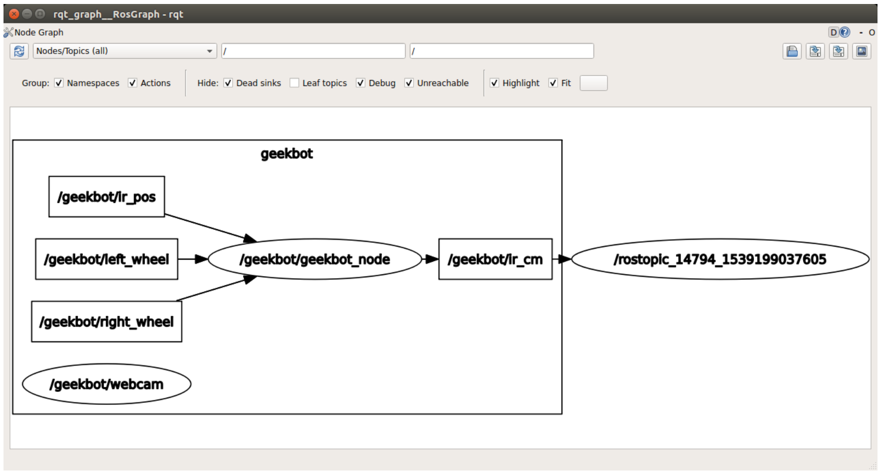
This shows the three nodes running in the GeekBot's namespace: geekbot_node, webcam, and rosout. Wouldn't it be neat if we could easily visualize what nodes were running and over what topics they were communicating? ROS has a tool for this! In one terminal echo the output of the topic `/geekbot/ir_cm`:

```
rostopic echo /geekbot/ir_cm
```

In another, use `rosrun` to run the ROS tool `rqt_graph`:

```
rosrun rqt_graph rqt_graph
```

This will run the pre-installed 'node' `rqt_graph` in the package `rqt_graph` and generate a topic/node map! You should see something like this:



rqt_graph__RosGraph - rqt

Node Graph

Nodes/Topics (all) | / | /

Group: ☑ Namespaces ☑ Actions | Hide: ☑ Dead sinks □ Leaf topics ☑ Debug ☑ Unreachable | ☑ Highlight ☑ Fit

You can see the `/geekbot` namespace contains two nodes and several topics. The node that exists outside of the `/geekbot` namespace is the node created by `rostopic echo`. `geekbot_node` publishes messages to the `/geekbot/ir_cm` topic, and the `/rostopic_.........` node is subscribed to this topic. The arrows indicate message flow. `rqt_graph` is a very handy tool for debugging and conceptualizing ROS systems. For large systems it is a necessity!

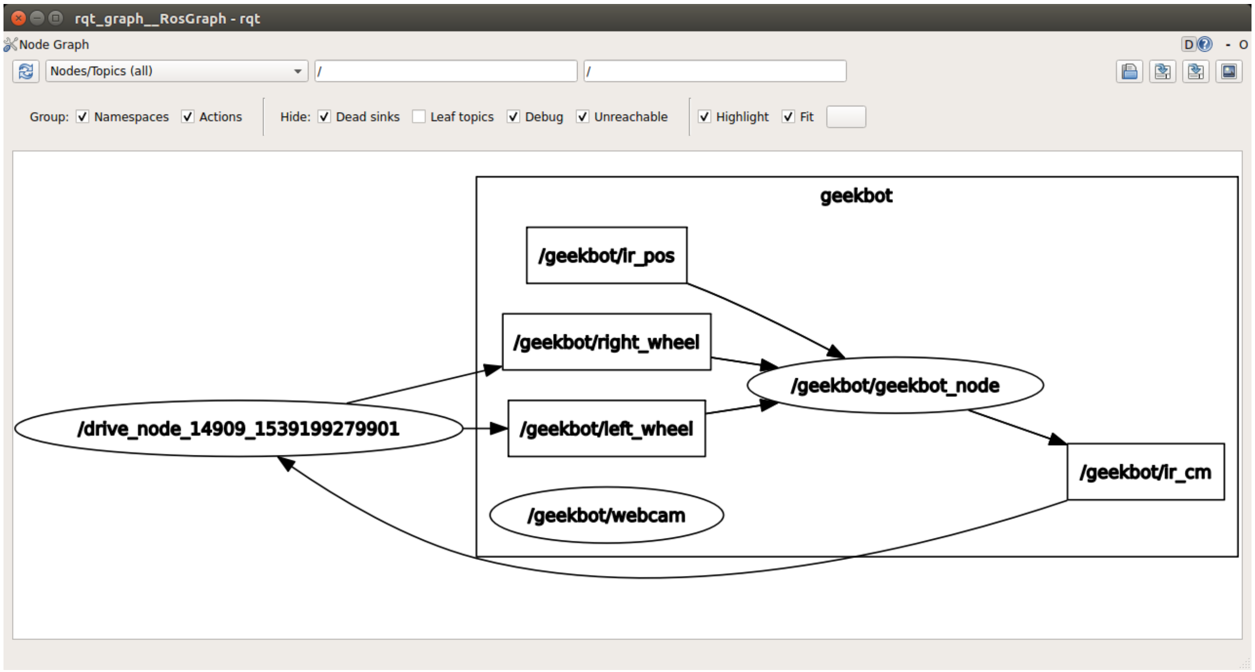
Sensor-driven motion

Inside the `geekbot_resources/examples/python_examples/basic_ir` directory you should see a file named `ir_drive.py`. This script contains:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import Int32

left_pub = rospy.Publisher('/geekbot/left_wheel', Int32, queue_size=10)
right_pub = rospy.Publisher('/geekbot/right_wheel', Int32, queue_size=10)

def callback(data):
        rospy.loginfo(rospy.get_caller_id() + " IR: %s", data.data)
        if data.data < 15:
        print("Driving forward.")
        msg = Int32()
        msg.data = 50
        left_pub.publish(msg)
        right_pub.publish(msg)
        else:
            print("Too far! Stop!")
            msg = Int32()
            msg.data = 0
            left_pub.publish(msg)
            right_pub.publish(msg)

def listener():
    rospy.init_node('drive_node', anonymous=True)
    rospy.Subscriber("/geekbot/ir_cm", Int32, callback)
    rospy.spin()

if __name__ == '__main__':
    listener()
```

All this code does is instantiate a ROS node that subscribes to the `/geekbot/ir_cm` topic and publishes wheel speeds based on the distance to the nearest object as determined by the front-mounted IR sensor. When run on your computer the GeekBot should sit patiently until you hold your hand within 15cm of the IR sensor. Then, it will drive forward! This is reversed logic from what we often want (drive until something gets in the way), but this way the GeekBot won't drive off your desk the moment the code starts executing. Consider this a friendly reminder to always prop up your robots so the wheels are off the ground when testing motion code.

Plotting the ROS graph with `rqt_graph` shows us that, indeed, the `ir_drive` node is indeed subbed to one topic and publishing to both wheels:



Node Graph

Nodes/Topics (all) / /

Group: ☑ Namespaces ☑ Actions Hide: ☑ Dead sinks □ Leaf topics ☑ Debug ☑ Unreachable ☑ Highlight ☑ Fit

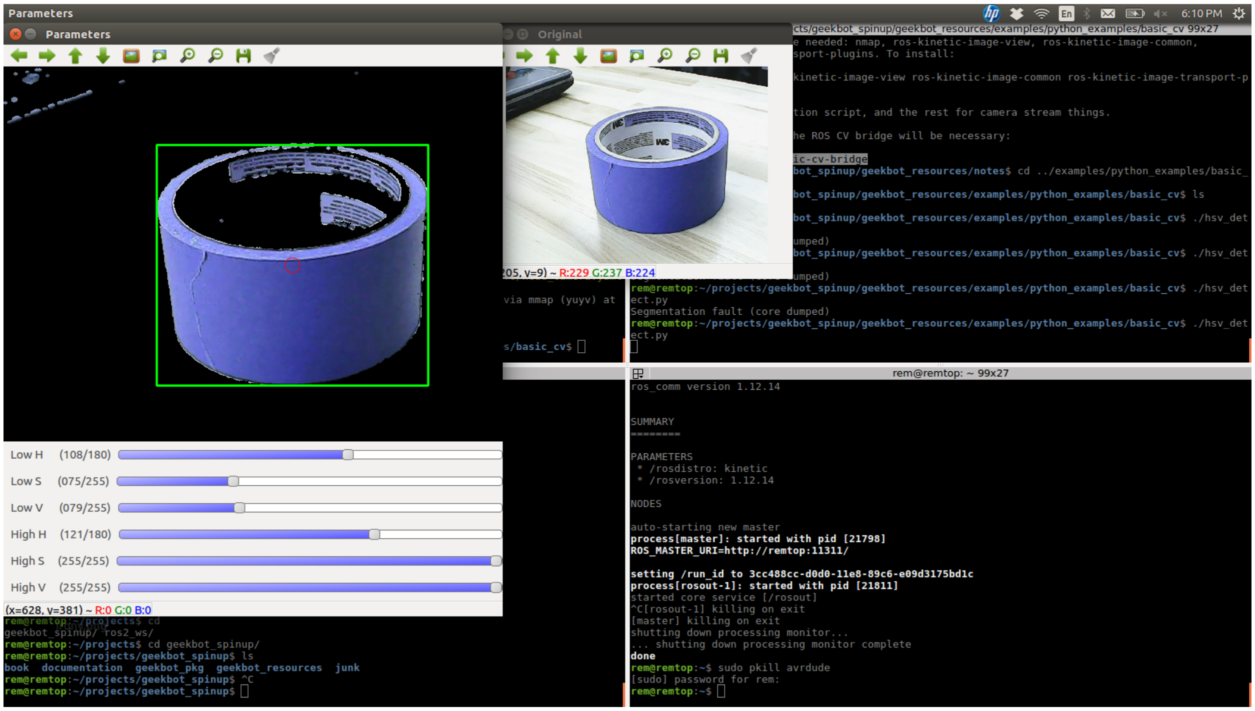
Basic computer vision

Inside the `geekbot_resources/examples/python_examples/basic_cv` directory you should see a file named `hsv_detect.py`. This script contains a bit too much code to prudently list here, so please open it in your editor of choice and read along. An online copy can be found on this repo page. **BE WARNED** There's a bug in Python2.7's muli-threading library that is known and will not be fixed. More info in the code itself. *Expect crashes.*

This code demonstrates 'elementary' computer vision capability using OpenCV to process images communicated over a ROS-based image stream. To use it you will need to install the `cv_bridge` package to convert between ROS CompressedImage messages and an image format OpenCV understands:

```
sudo apt install ros-kinetic-cv-bridge
```

With the bridge installed we can run the script. Upon running you should see two windows: in one a camera stream from the GeekBot, the other a black image. Move the 'maximum' sliders around. You should see pieces of the camera feed start to come into frame and maybe a few rectangles popping up. Set a high-color object in the camera's field of view and adjust the sliders to home in on just this color. Blues and yellows work well for this. With a bit of tinkering you should see something like this:



```
Low H   (108/180)
Low S   (075/255)
Low V   (079/255)
High H  (121/180)
High S  (255/255)
High V  (255/255)
(x=628, y=381) ~ R:0 G:0 B:0
```

```
205, y=9) ~ R:229 G:237 B:224
```

```
cts/geekbot_spinup/geekbot_resources/examples/python_examples/basic_cv 99x27
e needed: nmap, ros-kinetic-image-view, ros-kinetic-image-common,
sport-plugins. To install:

kinetic-image-view ros-kinetic-image-common ros-kinetic-image-transport-p


tion script, and the rest for camera stream things.

he ROS CV bridge will be necessary:

ic-cv-bridge
bot_spinup/geekbot_resources/notes$ cd ../examples/python_examples/basic_

bot_spinup/geekbot_resources/examples/python_examples/basic_cv$ ls

bot_spinup/geekbot_resources/examples/python_examples/basic_cv$ ./hsv_det

umped)
bot_spinup/geekbot_resources/examples/python_examples/basic_cv$ ./hsv_det

umped)
rem@remtop:~/projects/geekbot_spinup/geekbot_resources/examples/python_examples/basic_cv$ ./hsv_det
ect.py
Segmentation fault (core dumped)
rem@remtop:~/projects/geekbot_spinup/geekbot_resources/examples/python_examples/basic_cv$ ./hsv_det
ect.py
```

```
via mmap (yuyv) at

s/basic_cv$
```

```
rem@remtop:~/projects$ cd
geekbot_spinup/ ros2_ws/
rem@remtop:~/projects$ cd geekbot_spinup/
rem@remtop:~/projects/geekbot_spinup$ ls
book  documentation  geekbot_pkg  geekbot_resources  junk
rem@remtop:~/projects/geekbot_spinup$ ^C
rem@remtop:~/projects/geekbot_spinup$
```

```
rem@remtop: ~ 99x27
ros_comm version 1.12.14


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES

auto-starting new master
process[master]: started with pid [21798]
ROS_MASTER_URI=http://remtop:11311/

setting /run_id to 3cc488cc-d0d0-11e8-89c6-e09d3175bd1c
process[rosout-1]: started with pid [21811]
started core service [/rosout]
^C[rosout-1] killing on exit
[master] killing on exit
shutting down processing monitor...
... shutting down processing monitor complete
done
rem@remtop:~$ sudo pkill avrdude
[sudo] password for rem:
rem@remtop:~$
```

You will notice that the rectangle on screen is now bounding the object with a red circle drawn in the center of the rectangle. Hooray: object tracking by color extraction!

Here's a rough rundown of what's happening in the code, starting at initial execution:

1. Create an object_tracker object, initialize the node and start processing
2. When an image is published to the `/geekbot/webcam/image_raw/compressed` topic, convert it for OpenCV use
3. Make a copy of the original image and convert this copy to HSV from RGB
4. Threshold the HSV copy by the min and max values determined from the sliders. This returns a binary mask
5. Denoise the mask with a series of dilations and erodes
6. Find all connected contours in the mask. These contours are individual blobs in the mask
7. Given a contour, generate a bounding rectangle for the contour. If the bounded area is too small, reject the rectangle
8. Given a list of all viable bounding rectangles, find the largest
9. Bitwise AND the adjusted mask with the original image. This will block anything not captured in the mask
10. Draw the largest rectangle (and its center) on on the now-reduced original image
11. Display all images to their respective windows

The code might look intimidating off the cuff but it really is this straightforward. Conversion to the HSV colorspace is necessary to produce a consistent color lock regardless of color brightness. A quick online search will yield a variety of explanations that will be far more useful than any discussion of HSV's merits for computer vision in this text.

Notice that this code tracks the center of a bounding box. For the most part this center will be near the centroid of a tracked object. Using this information our GeekBot could be configured to automatically turn itself when a tracked object gets too close to the edge of the frame. The implementation of this (and other computer vision adventures) is left to the reader.

6.3 iRobot Create

Note: Write the section on using the iRobot creates.

6.4 Custom System - SMP

Note: Write the section on using the SMPs.

Note: Add SMP and other robots. This chapter is not yet written. It is listed here so we don't forget :) Links above are not populated yet.

CHAPTER

SEVEN

ELECTRICAL CONCEPTS

There are a wealth of useful sensors available to the robotics engineer. These include force, pressure, temperature, light, radiation, and more. If there is a form of energy that can be measured, then there is a sensor to do the measuring. The vast majority of sensors in robotics are electrically powered and operated. This section provides an overview of the standard concepts found in Physics course. A course in sensors as well as courses in mechatronics and instrumentation are the next steps for the developing roboticist.

7.1 Electrical Terms and Components

In this section, review the concepts of voltage, current and resistance, as well as some of the basic components. We begin with a common analogy relating the flow of electricity to water flow. This analogy is probably over used and has its limitations, but it does help in getting started. **Voltage** is the electrical pressure in the circuit. If we use a water pipe as an analogy, you can think of water pressure in the pipe as the voltage. The electrical pressure is measured in Volts. The symbol used in computation is v. Current is the flow of electrons along a conductor. Again, using the water pipe analogy, think of water flow in a pipe. The current flow is measured in Amps and the symbol used is i. **Resistance** is the measure of the difficulty to pass an electric current through that element. It is denoted by R and measured in units of Ohms. The power in an electrical circuit is measured in Watts and Watts = Volts * Amps. About 770 Watts makes up one horsepower. The fundamental components are given in Fig. 7.1, Fig. 7.2, Fig. 7.3.

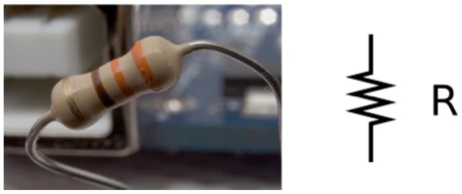


Fig. 7.1: Resistor (Ohms): resists the current flow, $V = iR$ (think of a narrowing in a pipe (a) Element (b) Resistor circuit diagram.

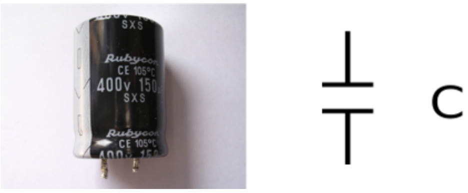


Fig. 7.2: Capacitor (Farads): stores energy in an electrical field, $i = C\frac{dV}{dt}$ (think of a storage tank). (a) Element (b) Capacitor circuit diagram.

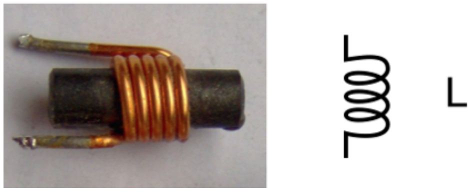


Fig. 7.3: Inductor (Henrys): stores energy in a magnetic field, $V = L\frac{di}{dt}$ (think of a flywheel in the pipe.) (a) Element (b) Inductor circuit diagram.

Table 7.1: Resistor color codes

Color	1st Band	2nd Band	(3rd Band)	Multiplier	Tolerance
Black	0	0	0	10^0 = 1	
Brown	1	1	1	10^1 = 10	1%
Red	2	2	2	10^2 = 100	
Orange	3	3	3	10^3 = 1000	
Yellow	4	4	4	10^4 = 10000	
Green	5	5	5	10^5 = 100000	0.5%
Blue	6	6	6	10^6 = 1000000	0.25%
Violet	7	7	7	10^7 = 10000000	0.1%
Gray	8	8	8	10^8 = 100000000	
White	9	9	9	10^9 = 1000000000	
Gold				10^{-1} = 0.1	5%
Silver				10^{-2} = 0.01	10%

The fundamental law in circuits is Ohm's Law, Fig. 7.5:

$$V = iR$$

where V is in volts, i is in amps, R is in Ohms.

Current flow can be in one direction or vary in direction. These are known as direct current (DC) and alternating current (AC).

Electronic devices run on direct current and this is the type of power delivered by batteries. Large scale power distribution is most efficiently done using alternating current (and at much higher voltages). So the power that enters our homes is AC. To get alternating current down from the high voltage levels that are used in transmission lines to an outlet, a transformer is used. You have often heard them as they make that

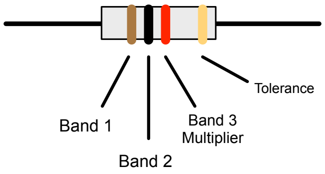


Fig. 7.4: The color bands on a resistor. To read, you place the tolerance band to the right (it is normally the band closest to the edge). In this example we have brown, black, red, gold. This is a four band resistor. The first three bands (or four bands if it is a five band resistor) will indicate resistance and the last will give tolerance. Band 1 is brown (1), Band 2 is black (0) and band 3 is red (100): 10*100 = 1000 ohms and 5% tolerance.

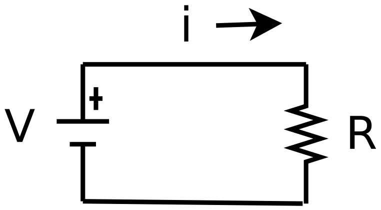


Fig. 7.5: Ohms Law. Note the direction of current flow is the opposite electron flow.

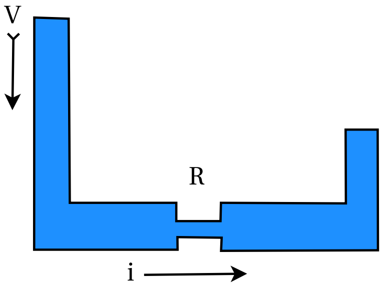


Fig. 7.6: Using the water metaphor, the water pressure is like voltage, the water flow is like the current and the narrowing of the pipe is similar to the resistor (pipe resistance).

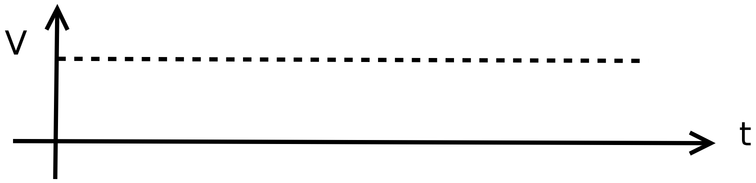


Fig. 7.7: Direct current.

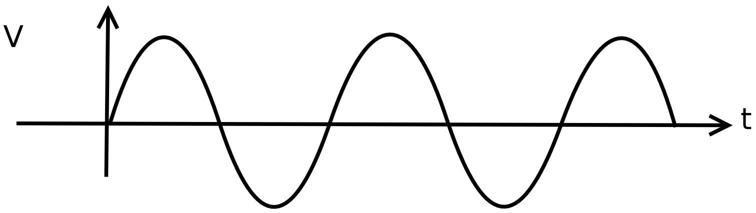


Fig. 7.8: Alternating current.

characteristic hum. To convert from AC to DC, another approach is used. A device called a diode has the property that it allows current to flow one way, in essence it is an electrical one way valve, Fig. 7.9.

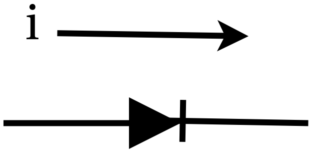


Fig. 7.9: Diode.

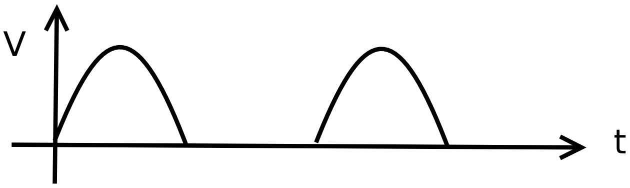


Fig. 7.10: The change in the current flow after the diode.

A clever connection of four diodes known as a diode bridge reroutes current so that it flows in one direction only (will still vary, but at least stay the same direction), Fig. 7.11. This bridge can also be used to protect inputs to electronic devices in case positive and negative lines get reversed.

The current headed out of the diode bridge flows in one direction, but the voltage is still fluctuating. Another device is employed, a capacitor. Using the water analogy, think of the capacitor as a storage tank. It will smooth out the voltage fluctuations like a pond smooths out stream flow. These basic circuit devices are used in a common household circuits such as a power supply, Fig. 7.13.

In this circuit, wall power (alternating current at 115 volts) is fed into the left side. S1 is the symbol for the on/off switch. The next device is a 3 Amp (3A) fuse. The high voltage AC is fed into the transformer (T1) and dropped down to 24 volts (still AC). Next comes the bridge circuit which re-routes the current flow so we have rectified (or unidirectional) current flow. Following the bridge is a large capacitor that will smooth the flow. It still has ripples in the flow (and they can be large). So the current is fed into a voltage regulator which significantly smooths the voltage level. The resistors and capacitors surrounding the regulator (LM317) select the output voltage level. Now you understand what is inside those bricks that charge your laptop, phone, camera, etc.

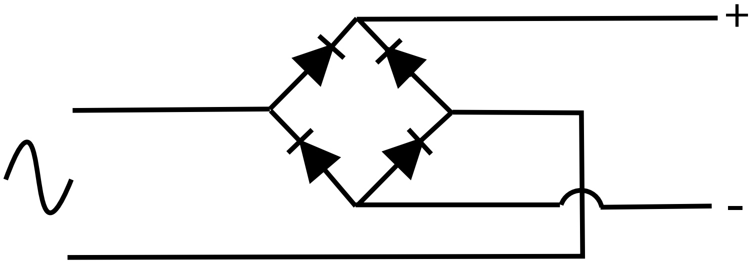


Fig. 7.11: A combination of diodes known as a bridge to convert alternating current into positive current.

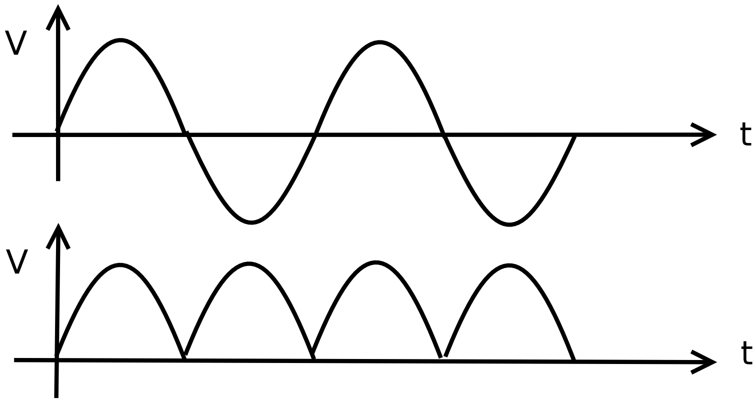


Fig. 7.12: The change in the current flow after the bridge circuit.

7.2 Batteries

Most mobile devices rely on batteries and they will be the main power source for the robots we discuss. [It is possible to have solar powered or inductively powered systems and maybe we will see more of this in the future.] All batteries are relatively slow, controlled, chemical reactions. They are not devices that store electric charge. The closest thing to that is a capacitor, which is comprised of two metal plates with an insulator between them. By applying voltage, you charged the plates, storing energy in the form of actual electric charge. However, capacitors tend to discharge all of their stored energy at once. In addition, the total energy they can store is far less than most batteries. Batteries, on the other hand, store energy in the form of chemical potential energy. This is far more stable than storing raw electric charge, but it does lead to a few problems. The big one is that, since the energy storage relies on chemistry, temperature is important. Being stored in a place that is too hot or too cold can cause a battery to burst, or drain it. Another is that, over time, the chemicals will degrade or react with other chemicals, causing the battery's maximum storage potential to decline.

We normally divide batteries up into Primary (non-rechargeable) and Secondary (rechargable) chemistries. We will briefly discuss secondary chemistries. There are three well known currently used chemistries for rechargeable batteries: Lead-Acid, Nickel Metal Hydride (NiMH), Lithium Polymer (LiPo). Lead-Acid is the type found in automobile, boat, motorcycle batteries.

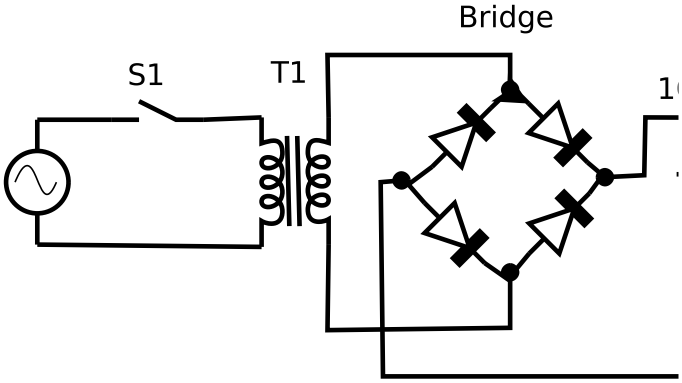


Fig. 7.13: The power supply circuit to provide a fixed 5 volts.

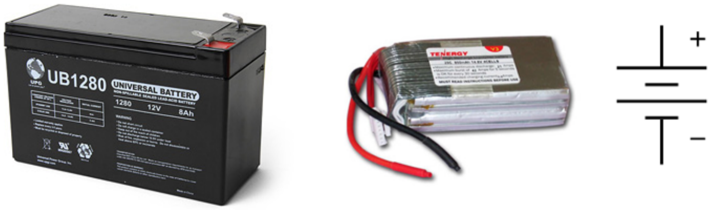


Fig. 7.14: Batteries: a) Lead-Acid cell. b) Li-Po c) Battery circuit symbol.

Table 7.2: Quick comparison of battery chemistries.

Chemistry:	Lead-Acid	NiMH	LiPo
Cell:	2.1V	1.2V	3.7 V
Weight:	Heavy	Light	Light
Energy density:	Moderate	High	Very high
Environment:	Toxic	Relatively green	Relatively green

7.2.1 Battery Voltage

We'll start with voltage because that's the easiest one to handle. You want to make sure the voltage your batteries produces matches the voltage rating for the things you want to power: motors, cpu, sensors, etc. However, there's another consideration to make. You almost never want your batteries directly connected to your sensitive electronics. You always want a regulator of some sort in between. Why? As you drain a battery, the voltage will decline over time. In addition, large loads on the battery can temporarily cause the voltage to fluctuate i.e. powering motors. Motors also cause back-emf, but we'll talk about that later. All of this can cause damage to unshielded electronics. In addition to the power fluctuations, you rarely have motors that you want to drive with the same voltage as the computer. In practice, you'll want to match your main battery voltage to the voltage of the motors you want to power, and then use regulators to smooth and reduce the voltage for the other components.

7.2.2 Battery Capacity - mAh "milli-Amp Hours"

This is a measure of how much actual energy the battery can hold. To put it simply, a 1000mAh battery could sustain a drain of 1A for 1 hour before being depleted. If you draw 2A it will only last half an hour. At 0.5A, two hours.

7.2.3 The C rating

Most batteries have a C rating. This is a somewhat cryptic value that tells you how quickly a battery can discharge without damaging itself. This is not a limit on how much current the battery can draw. Ohm's law will dictate the current that gets drawn from the battery. The C rating just tells you what is safe. The tricky bit is that the actual safe rate of discharge depends on the size of the battery. Here's the simple way to think about it. The amount of continuous current drain a battery can handle without damaging itself is obtained by multiplying the C rating with the battery capacity. For instance, a 1500mAh battery with a 5C rating can handle a continuous drain of 30,000mA or 30A.

7.2.4 Lead Acid Batteries - Pb

Lead acid batteries are very stable which is why we use them in cars. They also have a pretty good capacity, and are able to source a tremendous amount of current at once. This is good, because it takes a lot of force to turn over an engine. For a rugged, outdoor robot that may experience a variety of temperature conditions, lead-acid may be the way to go. However, lead-acid batteries are generally larger and heavier than their counterparts. Charging them is easy. Just hook them up to a power supply, set the power supply to the battery's rated voltage, and limit the current. What you limit the current to depends on the battery. Car batteries can generally handle up to 6A. The real problem is heat. If you charge too fast, the metal plates in the battery heat up, and can cause the acid to boil. This creates potentially toxic vapors and, if the vapors escape the battery housing, reduce the lifespan and charge of the battery. Other than charging too quickly, there isn't much to worry about here. Please charge in a well-ventilated area, just in case. Running a lead-acid battery dead isn't really a big deal as long as you don't leave it dead for a long time, or it doesn't get too cold while dead.

7.2.5 Lithium Polymer - LiPo

Lithium Polymers are light-weight, small, and can store a great deal of power. A LiPo battery generally consists of some number of cells. Each cell has a rated voltage of 3.7V. Batteries with more voltage are built by putting multiple cells in series. This voltage has to do with the internal chemistry. LiPo batteries are not nearly as stable as lead acid. There are three major concerns when dealing with a LiPo. First, only charge using a LiPo charger. There are some extra pins on a LiPo battery that tell the charger important information about the cells within. Second, make sure you look up the rating for charging a LiPo. The general rule of thumb is that a LiPo can be charged at the rate of 1C. Charging it slower is fine. Charging faster can cause the LiPo to heat up, swell, and potentially burst. When the LiPo bursts, the chemicals inside will spontaneously burn, creating fire, pressure, and heat. Third, never ever cut or puncture a LiPo. A ruptured LiPo is not safe and has caused serious fires.

That all said, LiPo batteries are pretty safe if you follow the guidelines for charging. There are two more things to worry about. 3.7V is the rated voltage for a cell, but when you charge, you generally charge to about 4.2V - the charger will handle this. Never overcharge the LiPo. However, unlike a lead-acid battery, letting the charge get too low in a LiPo will permanently damage it. The minimum safe level for a single cell is 3V. Always monitor the voltage of LiPo batteries you are using and ensure they don't drop below this level. The difficult part is, LiPo voltage doesn't drop linearly. Fig. 7.15 shows voltage versus charge for a LiPo. As you can see from the graph, you have to monitor the battery voltage very carefully, because it decreases rapidly once the charge gets low.

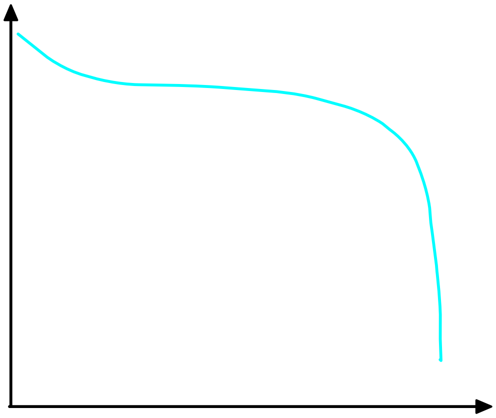


Fig. 7.15: LiPo Voltage VS Charge.

As a side note, this chart is for one specific battery I found on a forum post at Traxxas.com. Individual results may vary in specifics, but the point is the same. Monitor your voltage and don't let it drop below 3.0V per cell. There is one additional caveat. LiPo batteries don't hold their charge forever, and they will, if left sitting on a shelf for long periods of time, eventually degrade. It is recommended to discharge and charge a LiPo battery every few months when it is not in active use. Discharging can be done by using the LiPo while closely monitoring the voltage, or with a dedicated charger.

The final thing to mention is balancing. Because a LiPo battery may be made up of multiple cells, the total voltage isn't enough to tell the health of the battery. Some chargers will also balance while they charge. Balancing slowly bleeds charge from one cell and puts it into another cell. This keeps the cells from becoming unbalanced. This is a good thing since often each cell will discharge differently. This can lead to one cell with a much higher or lower voltage than the others. If one cell gets overcharged, it may swell and/or rupture. If one cell gets too low, it may "die". Dead cells are ones that have dropped to a low enough voltage that they cannot safely be recharged. Most chargers will refuse to charge the battery if there are dead cells. If you know what you're doing, sometimes dead cells can be nursed back to life, but it's a delicate and potentially dangerous procedure. It's usually better to recycle the battery and get a new one.

7.2.6 Other Batteries

There are three more common types of batteries. Lithium Iron Phosphate - LiFePO4 often pronounced "LieFo" - Nickel Metal Hydride - NiMH - and Nickel Cadmium - NiCd pronounced "Nigh-Cad". Roughly, LiFePO4 are similar to LiPo batteries but more stable and more expensive.

7.2.7 Tips For Not Lighting Things On Fire

Forethought and attentiveness are key for keeping the magic smoke inside the components. All electrical components are made using plastic, silicon, some trace metals, and a mystical substance called magic smoke. If you give the component too much voltage, current, or heat the magic smoke will use this extra energy to break free and escape. It is a very clever substance, and even just a momentary spike is enough to free it. At this point the component will no longer work. Nobody is perfect, but here are a few tips picked up over the years to keep the magic smoke locked up tight.

Turn Off The Power

This should be common sense, but you'd be surprised how often people get overconfident about what they're doing and modify a circuit while it's powered. Sure, if you know what you're doing you're theoretically safe but in practice it is foolish and dangerous. The problem arises when one of those tiny wires gets away from you or if you drop a metal piece or if you touch something by accident or … Powered wires appear to be supernaturally attracted to conductive terminals, particularly ones that will causes sparks. Just don't do it. The five seconds it takes to flip off the power could save you three days of waiting for new parts or a visit to the emergency room.

Electrical Tape

If you're working on a circuit and if you aren't immediately dealing with a wire, tape the end. Most of the instances in which folks blew something up was because they forgot about a wire that wasn't connected, and it brushed up against something else, making a short. This is particularly important when dealing with batteries. Remember, you can't turn a battery off. If both terminals of a battery aren't connected to something, the free wires should be taped over. Also, NEVER cut more than one battery cable at a time. This can short the battery and again release the magic smoke.

7.3 Basic Power Delivery

Say that you have a basic electric light circuit (like a flashlight), Fig. 7.16-(a). We are able to turn this on and off using a switch, Fig. 7.16-(b/c).

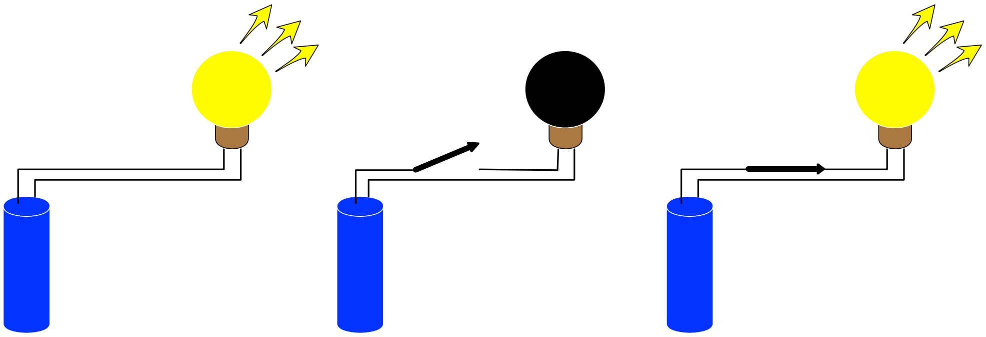


Fig. 7.16: Basic power delivery a) Direct connection b) Open switch c) Closed switch.

This is a great system until someone asks to dim the light. Now we are faced with how to reduce the voltage across the light. One might think that a transformer could be used. The problem is that the transformer is an AC not a DC device. So, the next idea is to limit current by placing a device along the flow of current that will resist the current flow. We can use the component described above called a resistor.

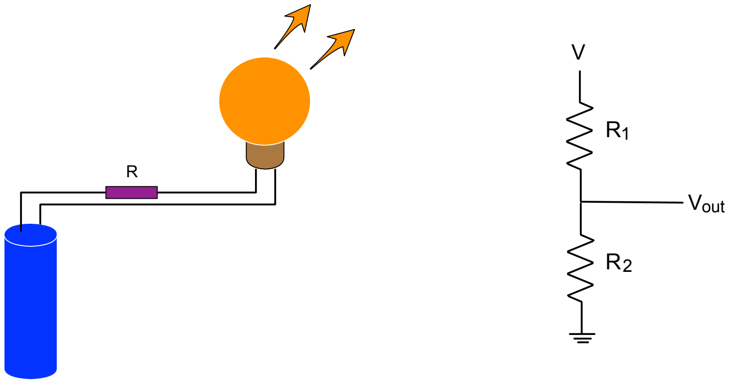


Fig. 7.17: Power control. (a) Resistor to limit current flow and drop voltage. (b) A voltage divider circuit.

The problem with this design is that some of the energy is wasted as heat in the resistor. For low power circuits, this may not be a problem, but for higher power devices like for electric motors, considerable energy is wasted as heat. Current through the resistors in Fig. 7.17-(b) is

$$i = \frac{V}{R_1 + R_2}.$$

Voltage drop across R_1 in Fig. 7.17-(b) is

$$V_{R_1} = \left(\frac{R_1}{R_1 + R_2} \right) V.$$

Power is

$$W = i * V_{R_1} = R_1 \left(\frac{V}{R_1 + R_2} \right)^2$$

7.3.1 A quick example:

Assume we are using a 12V power source and we want to use a voltage divider to provide 9V, Fig. 7.18.

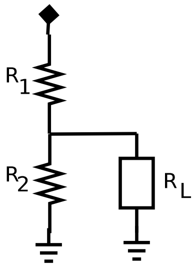


Fig. 7.18: Voltage divider to drop 12V to 9V.

Assume that the load is a simple resistor with resistance 10 ohms. Since R_2 is in parallel with the load, we get an effective resistor for the parallel combination of the load and R_2:

$$R_p = \frac{R_2 R_L}{(R_2 + R_L)} = \frac{10R_2}{(R_2+10)}.$$
$$The total resistance is$$

$$R = R_1 + R_p = R_1 + \frac{10R_2}{(R_2+10)}.$$

The voltage drop across R_1 is $(12-9) = 3$ volts and the current is given by

$$i = V/R = \frac{12}{R_1 + \frac{10R_2}{(R_2+10)}} = 3/R_1$$
$$so,$$

$$\frac{1}{4} = \left(\frac{R_1}{R_1 + \frac{10R_2}{(R_2+10)}}\right)$$

and after some algebra,

$$R_1 = \frac{5R_2}{(R_2+10)}.$$

If $R_2 = 10$ Ohms, then $R_1 = 2.5$. The load uses: $W_L = iV = (9/10)9 = 8.1$ Watts. The whole circuit uses

$$W = V^2/R = \frac{12^2}{R_1 + \frac{10R_2}{(R_2+10)}} = \frac{12^2}{2.5 + \frac{100}{(20)}} = 19.2$$

A waste of 19.2 - 8.1 = 11.1 Watts. For circuits that power larger motors, this can be a significant problem as it can be very difficult to remove the heat. The system will be at risk due to the high temperatures, for example burned components and melted solder, or even fire. For battery based circuits, this approach significantly reduces battery life. Another approach is needed.

One solution is to switch on and off the power very quickly, known as Pulse Width Modulation, PWM. To see what we mean, Fig. 7.19 here is a graph of the voltage though time.

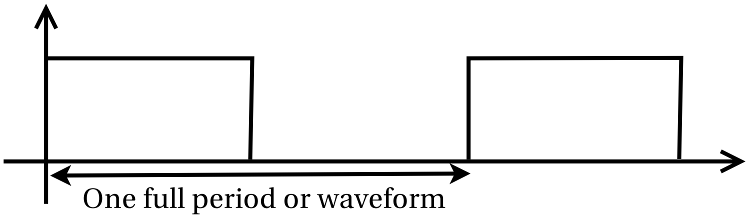


Fig. 7.19: Switching power on and off.

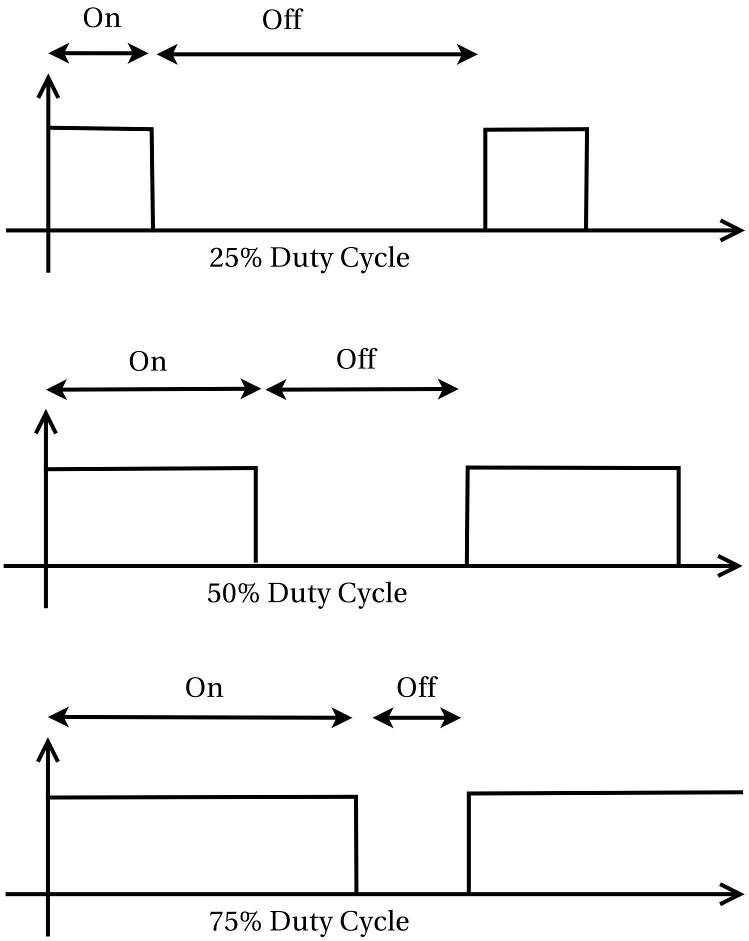


Fig. 7.20: On-Off pulsing known as Pulse Width Modulation - PWM.

The amount of time the pulse is high compared to low is the duty cycle. Duty cycle is often expressed as a percent of the pulse length which is called the period. Why does this matter? By this method, we deliver a fraction of the energy which then makes light dimmer. It does not have the energy waste as compared to using a resistor. If we run the on and off fast enough, our eyes will not see the flicker and it will just appear dimmer.

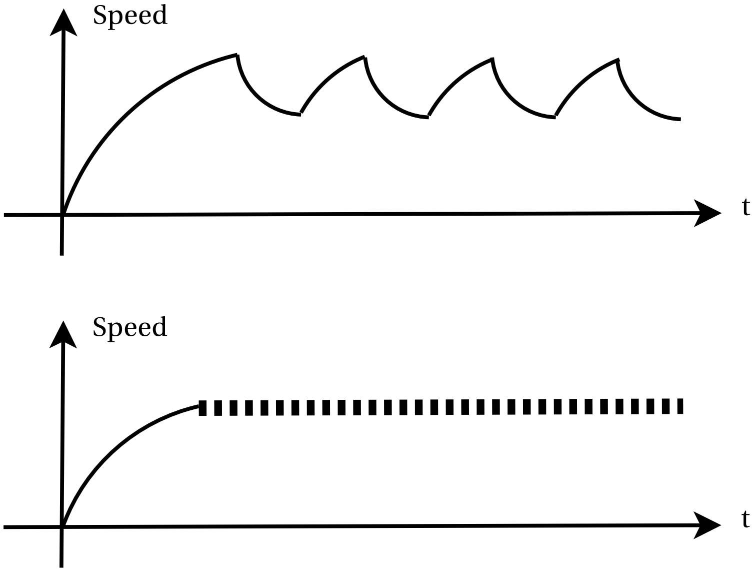


Fig. 7.21: PWM control of an electric motor.

This is also the method by which we control an electric motor. The frequency of this waveform does not change (because the duration of a single waveform is unchanged). The time that the voltage is high compared to the voltage is low does change. During the high part of the waveform an electric motor will start to increase in speed. During the low part the motor will coast and slow down.

You may ask how we switch the power on and off really fast. It is not like we have a little light switch and 87 cups of coffee. Hard for us, trivial for a computer. In fact, this is the basic way computers operate. They switch lines on and off millions or even billions of times per second. A program can be used to switch on and off an output line at a variety of frequencies and duty cycles.

If a computer generates the signal, the computer electronics is probably limited to 0.1 Amps or less. Certainly not enough to drive a large electric motor which might want to draw many amps. Using a pwm to drive a power transistor is the way to get power delivered. One minor problem is that this only runs one way. An H-bridge is a clever way to provide a reverse current, Fig. 7.23. By closing S1 and S4, current will flow from left to right, Fig. 7.24. By closing S3 and S2, current will flow from right to left. Replacing the switches with transistors will provide the switching speed required for PWM operation.

7.4 Electric Motors

Although motors are actuation and not sensing devices, since they are electrically powered, we address them here. An electric motor is in concept a very simple device. Any time current is flowing through a wire, a magnetic field is generated around (orthogonal to) the current flow direction. By wrapping the wire into a coil, the field lines overlap and intensify the magnetic field. This is the basis of an electromagnet. The

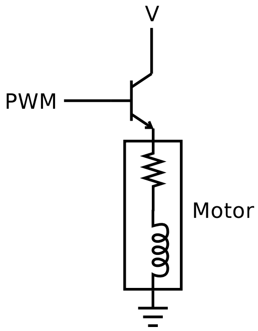


Fig. 7.22: Using a transistor to control power.

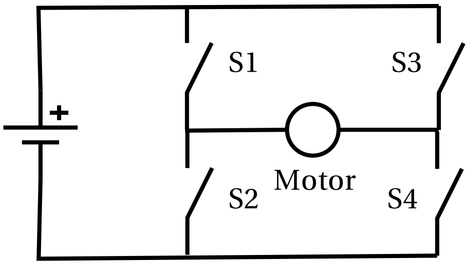


Fig. 7.23: H-Bridge, a way to select the direction of current flow.

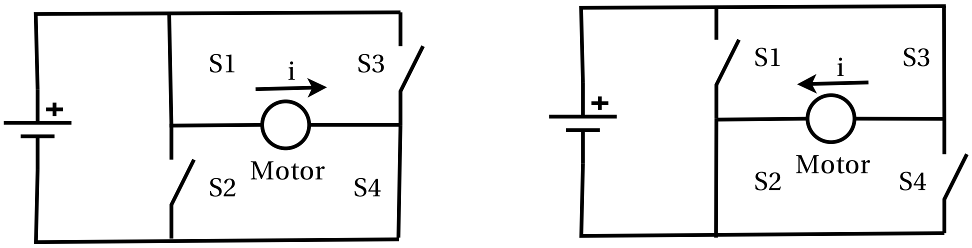


Fig. 7.24: Selecting current direction.

electromagnet will generate a force on a permanent magnet with the direction of the force depending on the magnet pole and current direction on the electromagnet. Placing the electromagnet on a pivoting or lever arm as shown in Fig. 7.25, a rotational force can be generated.

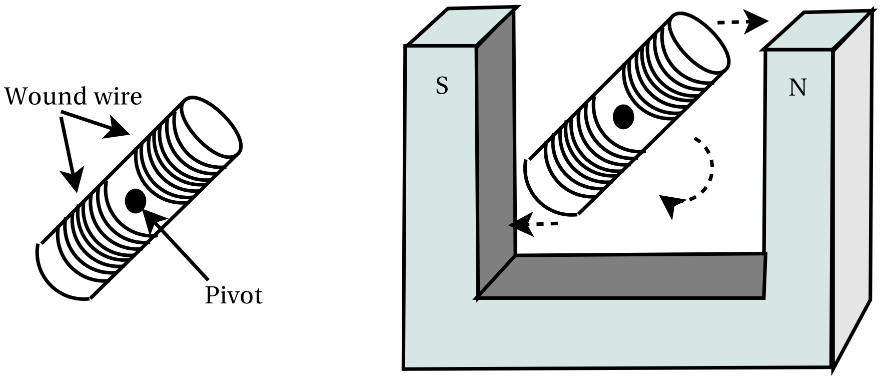


Fig. 7.25: Basic electric motor.

As described, the electromagnet will just align in the permanent magnet field and oscillate to a stop. The magic is to switch the current direction right as the moving electromagnet lines up. Then what was an attractive force switches to a repulsive force. The momentum of the arm will push past the alignment and the repulsive force will accelerate the arm towards the opposite alignment. Then one switches the current again repeating the process. The rotating arm will accelerate until the frictional forces balance with the magnetic forces.

There are many types of electric motors and what was presented is a simple inductive motor. Earlier designs would switch the current flow using contacts on the rotating shaft. Now electronic switching can be used. Motors which have this mechanical switching are referred to as brushed motors (the contact is a metal “brush”) and ones that don’t use them are called brushless motors. Motors can be run off of direct current as described above, a DC motor, and can run off of alternating current, an AC motor. Different designs provide motors with different speeds (revolutions per minute), power requirements and different torque properties.

A servo is an electric motor, some electronics and some gears. A signal is sent to the servo, normally a PWM signal. This signal is modified for the particulars of the servo operation and the specific motor. Normally this means that the PWM encodes a servo angle and this needs to be translated into the correct signals to position the motor. Motor position in low cost servos is read by a potentiometer (a variable resistor) which then by using some control logic adjusts the position according to the servo signal. A rough schematic is given in Fig. 7.26.

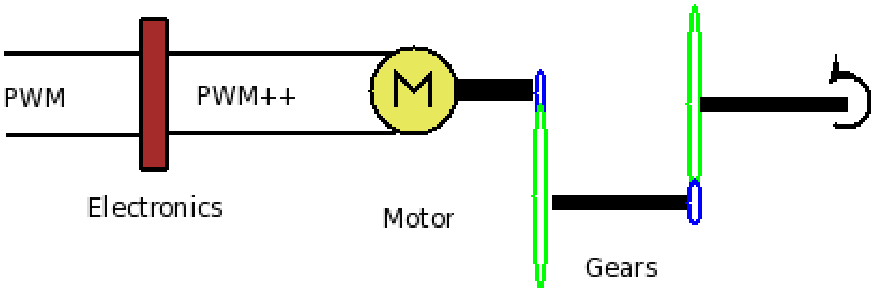


Fig. 7.26: Servo internals.

7.5 Problems

1. Provide the name and circuit diagram for the following:
    1. The circuit element that allows current to flow in one direction.
    2. The device that can boost or reduce AC voltage sources.
    3. The basic elements that store energy in electrical or magnetic fields.
    4. The device that reroutes current from ac to dc.
    5. The device that prevents oscillations in mechanical switch circuits.
2. Provide a labeled circuit/hardware diagram for a system that has a microcontroller driving a brushed DC motor (based on motor encoder output so that the system can control the actual speed and direction of the motor using a simple feedback based control loop). Assume that the stall current of the motor and the motor operating voltages are well in excess of what the microcontroller can source. [Note: Your microcontroller has PWM, GPIO, I2C, UART, lines for this application.] Explain each part of the diagram.
3. Assume that you can provide input for a motor controller in terms of percent of duty cycle (0-100): u. Also assume that at a 10 Hz rate you get a reading from an encoder that provides the output rpm of the wheel. Write a function that controls the rpm (range is 0 to 350), based on the value in u.
4. What is a PWM signal?
5. Can you think of a circuit to accept DC power which could hook up to the batteries either way. [Meaning that if the user hooks up the wires backwards, it automatically still works].
6. In an H bridge, are there switch combinations that cause problems? Why or why not?
7. When robots are rolling down a hill, the electric motors can act as generators. The current generated may damage the motor controllers. Is there a design that might be able to route the generated power to an on-board battery charger? Provide a circuit.

CHAPTER

EIGHT

SENSORS AND SENSING

Robotics is a interdisciplinary subject which relies on mechanical, electrical and software systems. Even though the focus of the text is on the computational aspects of robotics, it is important to have an overall understanding on the core systems and their functions. For this chapter we briefly touch on some of the sensors encountered in current robots.

8.1 Sensing

Sensors are a key tool to perceiving the environment. Our eyes, ears, nose, tongue, and skin are all sensors giving us details about our surroundings. Whether the environment is known or unknown, a robot requires sensors to perceive it as well. A roboticist needs to understand how a sensor functions and what its limitations are in order to use it to its full capacity. These limitations could include noise, bandwidth, data errors, and many other issues that must be accounted for in order to get accurate results. Understanding the physical principles of the sensors available is the key to understanding, modeling, and utilizing this information.

A sensor can be any device that converts energy into a usable signal. Sensors fall into two classes: passive sensors and active sensors. **Passive Sensors** use energy from the environment to power the measurement. A bump or temperature sensor are examples of passive sensing. **Active sensors** inject energy into the environment in a particular manner and measure the reaction. Active sensing can often get better results, but at an increased complexity, cost, and power requirements. It could also require the modification of the surrounding environment, such as the placement of beacons or tags. Laser and ultrasonic ranging are examples of active sensing.

We can further classify sensors by which part of the robot's environment, properioceptive or exterioceptive, they are sensing. **Properioceptive Sensors** measure the internal state of the system (robot), such as motor speeds, wheel loads, turn angles, battery status, temperature and other aspects that are internal to the machine. **Exterioceptive Sensors** measure information from the robot's external environment; external to the robot, such as distances to objects, GPS, ambient light and temperature, magnetic fields, accelerations, etc.

Another way to classify sensors is by the data they return. Sensors can return information in **analog** or **digital** form. Typically an analog sensor will vary voltage (maybe current) as the measured quantity changes value. Normal application is to feed that signal into a device known as an ADC or analog to digital converter. The vast array of microcontrollers on the market offer built in ADC lines. For example Arduino boards can take in analog signals. The voltage level is sampled and converted to a numerical value (hence digital). This means you might need some glue electronics to convert your sensor's voltage range to the full range of the ADC inputs. The ADC will be listed at having a certain number of bits (say 12 bit). This gives the resolution. For a 12 bit device, it means that the sampling will break the signal into 2^{12} discrete values or

4096 different levels. More bits means better resolution. But it takes more hardware and memory inside so there is a tradeoff. A digital sensor is a sensor that returns the measurement already in digitized or packet form and there is no need for an analog to digital conversion.

Digital sensors are often analog devices with built in ADC chips. To save packaging space they will very often communicate via a bus and not have the 12 pins required for a 12 bit sampling. So beyond the ADC, they will have some other device inside to send the signal out on some type of serial line protocol (uart, i2c, spi, . . .). Digital devices can be more accurate than analog devices but not always. It depends on the number of bits used and other factors in fabrication.

The desired qualities of a sensor are high accuracy and resolution, wide range of measurement, low delay times, stability of measurement with respect to the environment (no temperature drift or magnetic field interference for example), and above all very low cost. Sensors will talk to computers using two standard methods: polling and interrupts. For polling, the computer periodically reads the value on the correct register. Simplistically this is implemented via a loop in the software with a delay although better approaches involve setting a cpu timer and using an interrupt. The other approach is to have the sensor generate the interrupt and the cpu's interrupt handler will read the value on the register.

8.1.1 Sensor Metrics

There is a significant range in the quality of the sensed data. Some sensors may be very narrow in the range of sensing (e.g. range or angle of perception) while other are very wide. Sensors have noise which can vary depending on the sensor and the environment. Sensors will measure some quantity over some range, there are maximum and minimum values for inputs. The dynamic range is the ratio of the upper limit to the lower limit. Ranges can be very large and so the decibel is normally used. The formula for expressing the ratio depends on whether the sensed quantity is related to power or a field. Use of the 20 instead of the 10 is based on the standard use of the ratio of the squares and so we have a factor of 2 which comes out front.

Power

$$L_p = 10\log_{10}\left(\frac{P}{P_0}\right)\text{dB}$$

Field

$$L_p = 20\log_{10}\left(\frac{F}{F_0}\right)\text{dB}$$

Note that the ratio makes the decibel a unitless quantity.

Compute the dynamic range in dB for a measurement from 20mW to 50kW.

$$L_p = 10\log_{10}\left(\frac{50000}{.02}\right)\text{dB} = 63.979\text{dB}$$

Compute the dynamic range in dB for a measurement from 0.1V to 12V.

$$L_p = 20\log_{10}\left(\frac{12}{.1}\right)\text{dB} = 41.584\text{dB}$$

We have been using some terms that describe the sensor data and it is worth reviewing these terms.

Resolution In the world of digital sensors, this is often described as the number of bits used. It is the smallest change in the sensed value that can be measured. It is what you see in your science courses on measurement precision.

Accuracy It is how close the reported or measured value is to the actual value.

Range Or measurement range. It is the range of input values the sensor can detect.

Repeatability This describes the changes in the measured parameter over multiple measurements with a fixed value.

Frequency Some sensors produce new values at some clock rate which is given by frequency.

Response time The time delay between the measurement and the output value. Sometimes this will be used as the time delay between when the cpu requests a measurement and when the measurement is the available to the cpu.

Linearity The signal output is a linear function of the input.

Sensitivity The ratio of measured value to sensor output value.

All sensing involves measurement errors. There are standard ways to measure the error. Assume that x is the true value and z is the measured value. The *absolute error* is given by $|x - z|$. The *relative error* is given by $|1 - z/x|$. The reason we might choose relative error over absolute error is based on scale. For example, which of the pairs would you say is a better estimate: $(x, z) = (0.1, 0.2)$ and $(x, z) = (100, 102)$? The absolute error for the first is 0.1 and for the second is 2. Two is larger than 0.1. But intuitively we see that going from 100 to 102 is closer at the scale of 100. The relative error shows this with the first being a relative error of $|1 - 0.2/0.1| = 2$. The relative error on the second one is $|1 - 102/100| = 0.02$. This fits with our intuition about the errors. Relative error removes the scale and can be reported as a percentage which is called the percentage error, $100|1 - z/x|$. The accuracy of a measurement is given by 100 - percentage error.

8.2 Position, Velocity and Orientation Sensors

Navigation and localization is one of the more challenging problems in mobile robotics and any estimate is welcome. If you have an estimate on wheel velocity then you can by integration estimate rotation and through wheel diameter estimate the linear travel (over very short distances). Using the differential drive equations, (2.7), the wheel velocity may be integrated to determine position in the global or inertial reference frame. So the obvious question is… how do we determine wheel position or velocity?

Position sensors measure the absolute displacement of a joint or other sensed item for both linear and rotary joints. Both analog and digital technologies are used for measuring displacement. Variable resistors have been common for years. Either as a rotary device such as found in volume dials or slider (fader) such as found in mixers. The rotary variable resistor is known as a potentiometer or “pot” for short. Typically wired in a voltage divider (discussed later), the voltage across the potentiometer can be measured as an analog signal. It can be converted to a digital signal for use in a microcontroller. Variable resistors have an element which slides over a coil of resistive wire or over carbon base changing the electrical path which varies the resistance.

A direct digital approach is to use encoders. Absolute encoders return a Gray code as a function of position. These also come in linear and rotary designs. Typically one has a collection of sensors which can detect light and dark. The encoder allows light to pass through it in some pattern that correlates to position. The sensed pattern is converted to the output code. Another approach used is to count the number of times a particular pattern occurs or a beam is interrupted. This is explored in detail below with the optical wheel encoders which can be used for velocity in addition to position estimates.

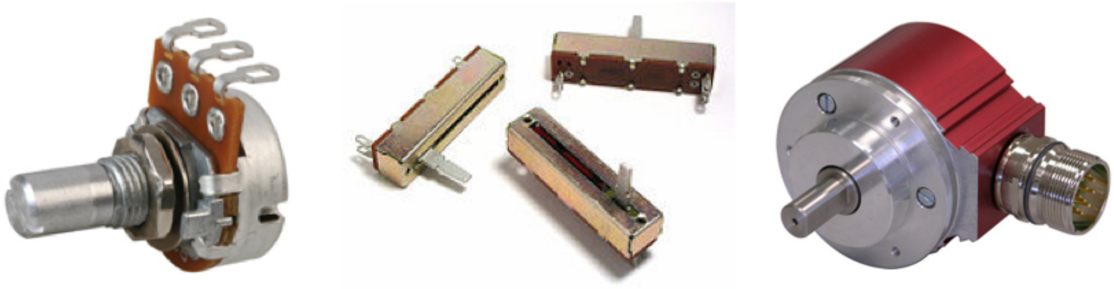


Fig. 8.1: Position sensors: (a) Potentiometer, (b) Fader, (c) Encoder.

8.2.1 Tachometers

An electric motor and a generator are very similar devices which just operate in opposite fashions. Providing electrical power in a motor causes the shaft to turn. Conversely turning the shaft of a generator produces electricity. A tachometer can be built out of a generator (or electric motor). The faster the shaft spins, the greater the voltage or higher the frequency produced. This can be converted to a digital signal and thus provides a measure of rpm.

8.2.2 Optical Wheel Encoders

One option to tackle measurement of rotation is known as an optical wheel encoder. We will discuss two common types, the incremental and absolute encoders. They operate by focusing a beammof light onto a pattern mounting to the rotating surface. That pattern is read and the rotation or rotation increment is computed. The dominant lighting source in electronics and robotics, are Light Emitting Diodes, or LEDs[1] which can run on very low power, are available in many frequencies and can switch on/off quickly. Fig. 8.2.

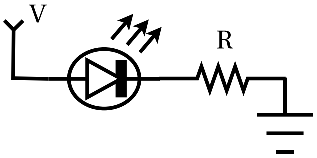


Fig. 8.2: LED

LEDs can emit in non-visible ranges, ultraviolet and infrared. Many of the non-visible frequencies are popular for simple object detection in combination with a phototransistor, Fig. 8.3. In this example, the

[1] LEDs have a variety of operating specs and you have to read the datasheet to find out about the specific voltage-current properties. Normally one is given an operating range and one must work out a suitable way to power the diode. For example, assume we have and LED which operates in the 3-6 volt range and targeted current level is 20mA. If we select $V = 5$, then the resistor should be $R = V/I = 5/.02 = 250$. Since 250 is not a standard value, we select the closest available resistor value which is $R =270$ ohms.

infrared LED shines on some object and is reflected back to the phototransistor. The IR light activates the transistor and causes it to switch on and pull the output to low.

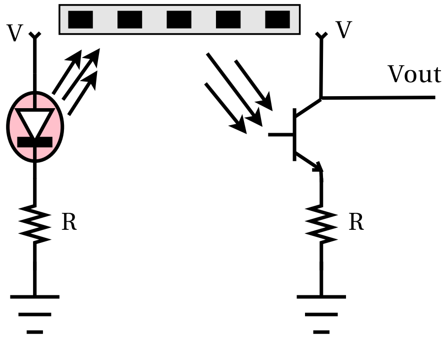


Fig. 8.3: Infrared LEDs used for obstacle detection.

This system can be used for simple occupancy detection or close obstacle detection. We can also use the LED-transistor combination to determine wheel rotation; to measure the speed or position of a wheel or dial. For example the dials on electronic devices like a volume control. In addition, knowing wheel rotation can assist in the process of localizing the robot. The fundamental idea is to generate a radial or linear pattern of black and white stripes (or slits). The IR light is either reflected or not. This is sensed with the phototransistor. Counting the stripes (or lists) can provide an estimate of wheel rotation. Over a fixed interval of time this provides an estimate of wheel velocity. The estimate is clearly improved if more stripes (or slits) per revolution are used.

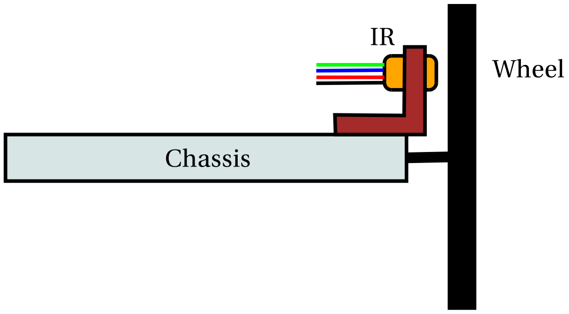


Fig. 8.4: Mounting for the encoder sensor

There are two basic components needed to build your own incremental encoder. First you need the light source and the detector. Second you need a pattern. To read the pattern, you will need an optical sensor. Typically one uses an IR LED (IR light emitting diode) and phototransistor pair, Fig. 8.5. These are packaged in single units, for example the Fairchild QRD1313. This has the LED and the phototransistor packaged into a unit that is 6.1mm x 4.39mm x 4.65mm (height).

An encoder pattern may simply be a pattern printed on paper and attached (glued) to the inside of a robot wheel. Simple encoder patterns are just alternating black and white radial stripes. Two examples are given in Fig. 8.6.

This encoder pattern will generate in idea conditions a square wave if one continuously samples the state of the phototransistor. See Fig. 8.7. Our system is digital and we don't use the raw voltage signal. That signal is sampled by some type of analog to digital hardware. For example, the output of the phototransistor is connected to a general purpose I/O line (GPIO) on a microcontroller. The microcontroller input will

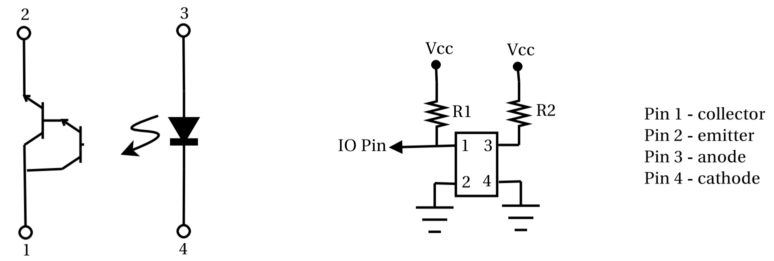


Fig. 8.5: IR LED (IR light emitting diode) and phototransistor pair.

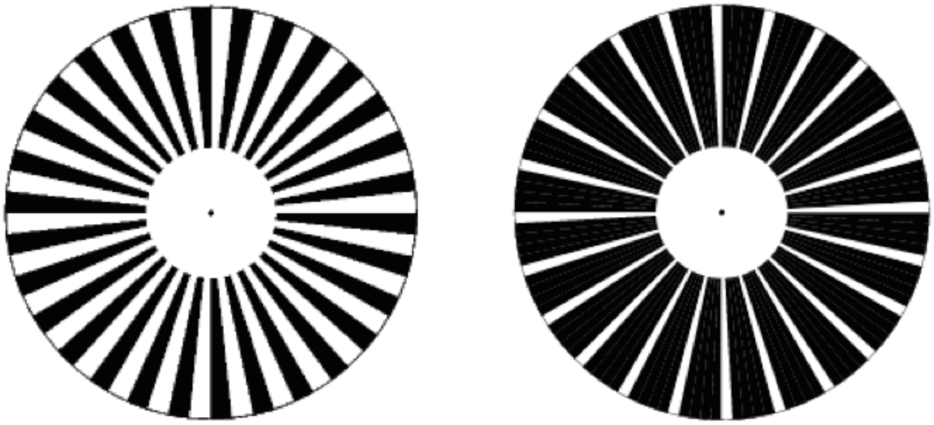


Fig. 8.6: Wheel encoder pattern (a) with 1-1 ratio, (b) with 1-4 ratio.

threshold the signal. Anything below a certain voltage will be considered *low* or zero volts. Above a threshold voltage level it will be considered *high* or max voltage for the input. The low or high on the line is then the 0 or 1 when the value is read in the code.

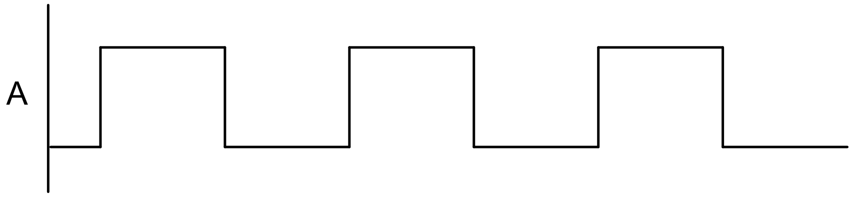


Fig. 8.7: A square wave generated by the phototransistor sensing the rotating pattern.

In reality the wave will not be square. The light source and the sensitivity of the detector as well as the thresholding might trigger high for much greater percentage of the time. Sometimes their is sufficient light that the system never pulls low (or the opposite). This is the reason you may need to use the pattern shown in Fig. 8.7 (b). Hooking the output of the sensor to a oscilloscope can give you an idea of the wave shape. [You will need to run the motor and encoder. Mount them, hook a power supply and then hook to the oscilloscope. If youn have a clean square wave - great.]

There are two common ways that reading the GPIO input can be done. One is known as polling and the other as interrupts. *Polling* requires that the hardware samples the sensor (the GPIO) at a regular rate. In simple applications this is done just by putting the GPIO read in a loop. The loop is not always uniform when it reads, but it can be good enough for some applications.

The digital signal is then converted to a binary stream as seen in Fig. 8.8. When the rotation speed increases, then the number of zeros or ones in the uniform part of the sequence will decrease. By counting, one can gain an estimate of the square wave period and then of rotational velocity. Unless you know that the square wave will always have 50% duty cycle, you should start and end counting at the transition from 0 to 1. The period can be computed from $T = n\Delta t$, where n is the previous count.

Assume that you have k bands (black or white, not both) and the period was measured to be T. The rotational velocity (radians) is

$$\omega = \frac{2\pi}{kT}$$

Since there is finite time between samples, there is an error in the estimate of the rotational velocity. The problem is that you don't know exactly when the transition from low to high or high to low occurred. This will introduce some measurement error. This error is higher for systems with fewer bands (wider bands) or slower rotational velocities.

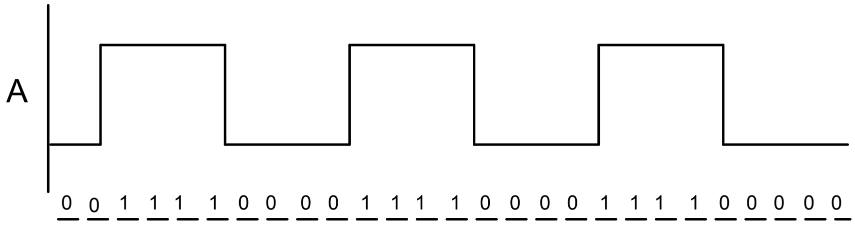


Fig. 8.8: Using polling to sample the square wave.

If we want to get a more accurate sense of the period of the square wave, then we need to implement interrupts. You need hardware that supports interrupts on a GPIO input. *Interrupts* cause a change in the program flow. WHen an interrupt occurs, the current program is suspended and the execution jumps to special code, the interrupt handler. This handler does the necessary work to deal with the situation and then the system returns to the program.

In this case, we set the GPIO to generate an interrupt when the line goes from low to high. This gives us the front edge of the square wave. The interrupt handler can grab the current time and store it in a global variable. It will compute the difference between the current interrupt time and the last one. It can store the period in a global variable as well. The global variable is available to other programs that would like to know the period from which the rotational velocity is estimated.

Although the interrupt approach is more complicated to program, it provides more accurate timings. However, updates only occur when the interrupt occurs so this operates asychronously relative to the loops in the software. It is an example of event based programming.

We may extend either approach by mounting two detectors offset by 1/2 of the width of the band. Since the black and white bands form the full square wave this offset amounts to 1/4 of the full wave which is also 90 degrees if we think of the square wave running a full cycle. The "leading" wave indicates direction. So this

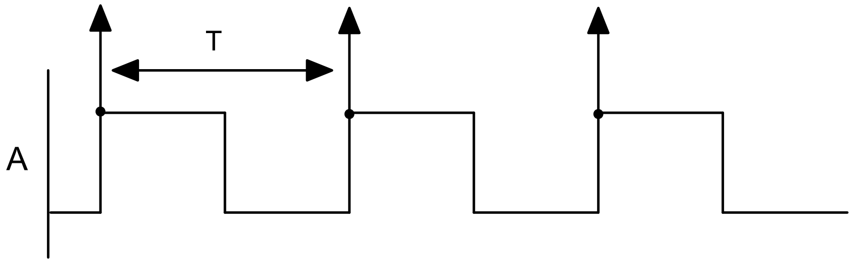


Fig. 8.9: Using interrupts to determine the period and thus the rotational velocity.

dual version can sense speed and direction Fig. 8.10. These are known as phase quadrature encoders. They are also in the category of incremental encoders meaning you still need to poll and count or use interrupts, and, you need to do this on both channels.

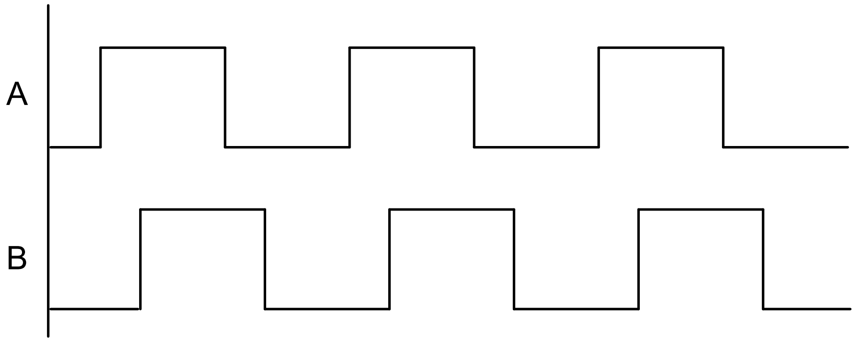


Fig. 8.10: The two square waves that are 90 degrees out of phase.

Using a row of detectors and more complicated patterns, it is possible to create an absolute encoder. These encoders know position as well as velocity. This is important when absolute position is required. Instead of a balck stripe a pattern of dots following the Grey code is used. Grey code is popular because you only have one bit change with each increment or rotation.

A partial absolute system can be easily added to any incremental encoder. A third detector can look for a black or white dot on the outer part of the ring. This can generate a pulse for each complete rotation of the wheel. This will allow corrections to any possible counting or missed interrupts errors.

These designs are not limited certain designs. The black and white pattern can be placed linearly. Encoding can be done on linear actuators as well as the rotational units discussed above.

8.2.3 Doppler Effect

Direct measurement of velocity may be achieved by using the Doppler Effect. Recall when a vehicle passes by, you notice a change in the sound of the machine. The sound waves are compressed as the vehicle approaches and are expanded as the vehicle retreats. This compression results in a higher frequency of the sound and so as the vehicle passes, you hear the drop in frequency. Transmitting a known frequency and listening to the reflected sound, one can estimate the relative velocity.

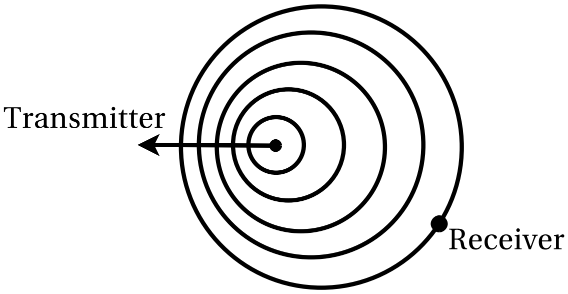


Fig. 8.11: Using the Doppler Effect to estimate velocity.

The formula that describes the change in frequency for a moving sound source (a transmitter) is

$$f_r = f_t(1 + v/c).$$

If the receiver is moving the formula for the frequency change is

$$f_r = f_t/(1 + v/c).$$

If you know the frequency change you can then compute v.

8.2.4 Gyroscopes

The word pose is used for both position and orientation. Measurement of orientation is done through several basic sensing approaches which we discuss here.

A gyroscope is a heading sensor that gives a measure of orientation with respect to a fixed frame. A standard gyro provides an absolute measure of the heading for a mobile robot normally measured in degrees off of some fixed heading. The classical mechanical gyroscope uses a spinning body and the resulting rotational inertia. Optical gyros can use phase shifts of intersecting beams of light to measure changes in orientation. Rate gyros give a measure of angular speeds which is the standard for low-cost MEMS systems. These are the most common units found in mobile robots, UAVs, phones and other portable devices. These gyros will return data in degrees per second (deg/s). Like accelerometers, the MEMS units (microelectromechanical systems) are packaged into 1, 2, 3 sensors to provide rotational rates about the x, y and z axes. Also like the accelerometer, the gyro can have a digital or analog interface.

Drift can be a significant issue. The absolute direction can be lost over a period of time depending on the sensor quality. This is an issue that must be addressed for systems which run for long periods of time without a recalibration.

8.2.5 Compass

The compass or magnetometer is one of the oldest sensors in use dating back 4000 years. Early forms would take a small piece of loadstone (natural magnetite) and suspend it from a silk thread or place it on wood and float that in water. This magnetic rock would orient along the Earth’s field lines and could then

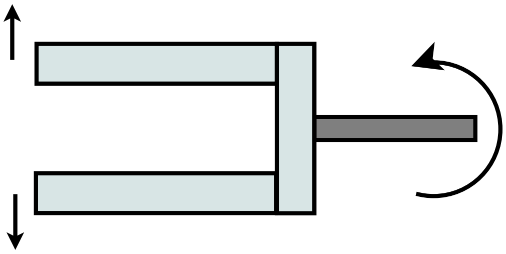


Fig. 8.12: Tuning fork MEMS gyroscope.

be used for navigation. Due to the liquid iron core, the Earth's field is sufficiently strong to measure with portable devices. Although pole reversals do occur, we have relatively stable pole locations for long periods of time, so compass navigation has been a popular orientation tool for thousands of years. This is an absolute measure of orientation in contrast to the relative sensing we saw with the radial encoder.

There are multiple ways to measure a magnetic field. The traditional methods are known as mechanical in that the force of the field lines induces a torque and moves part of the sensor. Other approaches use the Hall-Effect or Magnetoresistive sensors. The earth's magnetic field is still relatively weak. Other magnetic sources such as inductors can disturb and invalidate measurements. It is critical when building the sensing system, the sensor is not placed near a motor, power supply or any other device which can generate magnetic interference. Large amounts of iron can alter the earth's field or even shield it. This prevents a magnetic sensing in certain environments.

Magnetic encoding

It is possible to use Hall-Effect or other similar devices to do encoding. Small Hall-Effect sensors with sub-degreee accuracy are available. Placing a small ceramic magnet on the end of a shaft will generate a rotating magnetic field which can be detected with the Hall-Effect sensor. Figure Fig. 8.13 shows how this is done.

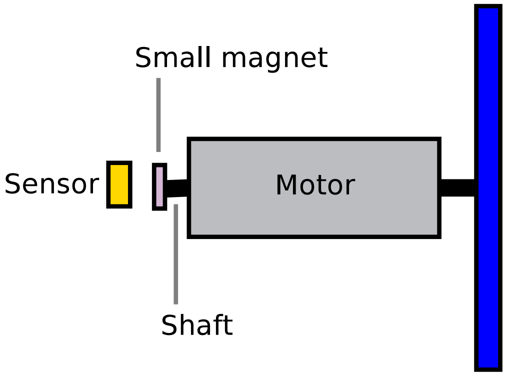


Fig. 8.13: Hall-Effect based shaft rotation sensor.

8.2.6 Accelerometers and Inertial Sensing

An accelerometer measures acceleration in a particular direction. The standard units on acceleration are meters per seconds squared (m/s^2). The sensor is normally a MEMS unit which are often packaged together using two or three sensor units pointed in orthogonal directions. This can provide acceleration information along each of the coordinate axes. Common constructions use two plates with one moveable and attached to a mass, and the other fixed. Acceleration will cause the plate to move and change the capacitance. This change can be measured and related to the acceleration. Output may be a voltage level in which the sensor is known as an analog sensor or the output can be through a digital interface, such as I^{2}c making it a digital sensor.

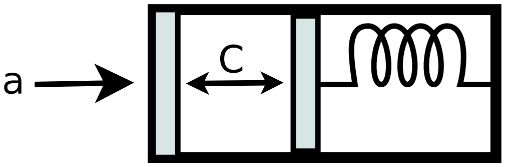


Fig. 8.14: Simple accelerometer structure.

A simple application of an accelerometer is an inclineometer or tilt sensor. These sensors can have a great deal of noise and extracting a good signal can be very challenging. Note that it is temping to think that this device can provide position information. After all, we learn in calculus that if we integrate acceleration twice, we obtain position. The problem is the noise. Even though integration is numerically a smoothing process which can reduce noise by averaging it out, over time small errors build and position accuracy is poor. In practice, using an accelerometer does not provide adequate position or velocity estimates.

8.2.7 Inertial Measurement Unit (IMU)

An Inertial Measurement Unit or IMU packages accelerometers, gyroscopes and possibly a compass together into a single unit. A 6DOF (degrees of freedom) IMU will have a three axis accelerometer and three axis gyroscope. A 9DOF IMU will have the three axis accelerometer, three axis gyroscope and a 3 axis magnetometer. These devices normally provide a digital interface such as USB and return text strings of data at some Hz. IMUs are used as the basis for AHRS: Attitude and Heading Reference System.

8.2.8 Attitude and Heading Reference System (AHRS)

AHRS consist of accelerometers, gyroscopes and magnetometers on all three axes. So, AHRS includes an IMU. In addition to the IMU, the AHRS has the algorithms to provide attitude and heading information as well as the required hardware for the computation. These algorithms include sensor fusion codes which take data from multiple sources and “fuse” them into a hopefully more accurate picture of the measured quantity. A popular estimator known as the Kalman Filter is used to do the fusion and state estimation. A variant of AHRS is an Inertial Navigation System (INS). The difference is that INS estimates attitude, position and velocity.

8.3 Location Sensors

8.3.1 Beacons

While IMUs are amazing devices, they cannot give us accurate position information. Estimating the position in the environment is essential for navigation. One approach is to instrument the environment. An example of this would be placing markers that the robot sensors can detect and reliably interpret for location information. We will explore several of these ideas with beacon systems as our guiding example.

Beacons are probably the simplest approach to localization. A beacon is any type of landmark with a known location. Natural beacons such as stars, sun, moon, mountains, streams and other markers have been used throughout human existence[1]. Manmade beacons include towers, signs, lighthouses and other marked locations.

Indoor beacon systems include using colored or IR lights, RFID tags, ultrasonic transducers, QR codes, colored tags and other forms of environmental instrumentation. Normally this means that the environment is modified in some detectable manner. Similar approaches can be done outdoors, but since the introduction of GPS, it has dominated the localization approaches. GPS, the Global Positioning System, was developed for the US military for their localization and navigation requirements.

GPS uses signals from satellites to triangulate position. Conceptually it is rather simple to use time of flight from four satellites to exactly locate an object. The challenges are that the distances are great, the speed of light is very high and the Earth is often in the way. To address the line of sight requirement, 24 satellites with several spares orbit the earth every 12 hours at an altitude of 20,190 km. They are arranged as four satellites in six planes offset by 55 degrees from the plane of the equator. Knowing the time of flight and the speed of light, distance of the observer from the satellite can be determined.

There are several challenges to be overcome. First is a precise measurement of the time of flight. Time synchronization between satellites and GPS receiver is essential. Secondly, a precise location of the satellite is required. In addition one needs to deal with signal quality and interference.

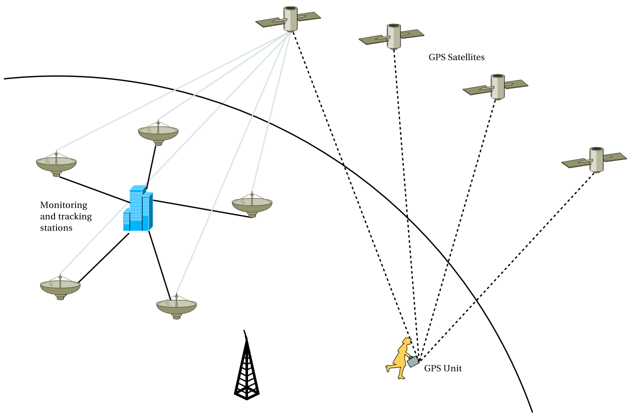


Fig. 8.15: Global Positioning System - GPS

Each satellite has an atomic clock for ultra-precise tracking of time. They are monitored by ground stations. These ground stations also track the location of the satellites. The ground station can perform the location analysis and transmit the position estimate to the satellites.

[1] One would assume that natural beacons are used by animals as well

A GPS receiver will grab a code from the satellite which has time stamp data. Using this information, the distance between the satellite and the receiver is computed. The clock on most receivers is not very accurate so information from more than three satellites are required to adjust for local clock errors. This allows for estimates to be accurate within several meters.

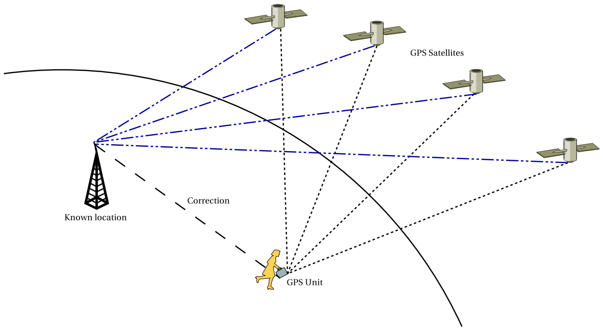


Fig. 8.16: GPS with local correction.

Example

Assume that you have four beacon towers located in roughly a square over a 10km x 10km patch of land. You place a coordinate system on the land and measure the beacon locations. The locations in meters are B1 (0,0), B2 (56, 9752), B3 (9126, 7797), B4 (9863, 218). If the beacons transmit a packet with a time stamp, then a mobile system with an accurate clock can determine its location in the instrumented area. Determine locations if $t_1 = 22793$ ns, $t_2 = 15930$ ns, $t_3 = 20817$ ns, $t_4 = 29793$ ns. The distances are found via $d = ct$: $d_1 = 6838m$, $d_2 = 4779m$, $d_3 = 6245m$, $d_4 = 8938m$. So our object lies on a circle of distance d_1 from beacon one and distance d_2 from beacon two, etc.

One may intersect two circles to provide the location of the two intersecting points and then proceed over all combinations:

$$(x - a_i)^2 + (y - b_i)^2 = r_i^2, \quad (x - a_j)^2 + (y - b_j)^2 = r_j^2.$$

The algebra can be simplified by expanding each circle equation

$$x^2 - 2a_i x + a_i^2 + y^2 - 2b_i y + b_i^2 = r_i^2, \quad x^2 - 2a_j x + a_j^2 + y^2 - 2b_j y + b_j^2 = r_j^2$$

and computing a difference

$$2(a_j - a_i)x + 2(b_j - b_i)y + a_i^2 - a_j^2 + b_i^2 - b_j^2 = r_i^2 - r_j^2.$$

Using three circle equations, you can obtain two linear equations

$$2(a_j - a_i)x + 2(b_j - b_i)y = r_i^2 - r_j^2 - a_i^2 + a_j^2 - b_i^2 + b_j^2$$

$$2(a_k - a_i)x + 2(b_k - b_i)y = r_i^2 - r_k^2 - a_i^2 + a_k^2 - b_i^2 + b_k^2.$$

In a noise free world, the solution would be where the circles intersect exactly such as seen in Fig. 8.17. But this does not happen due to noise and sensor inaccuracies. The circles do not intersect as shown in Fig. 8.18.

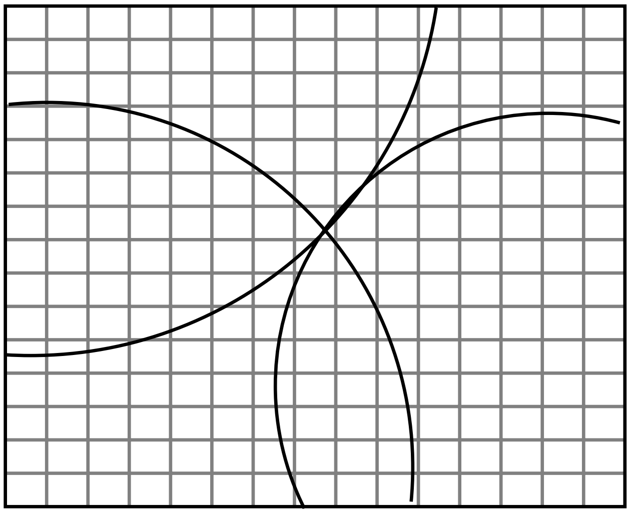


Fig. 8.17: Exact intersection of three circles.

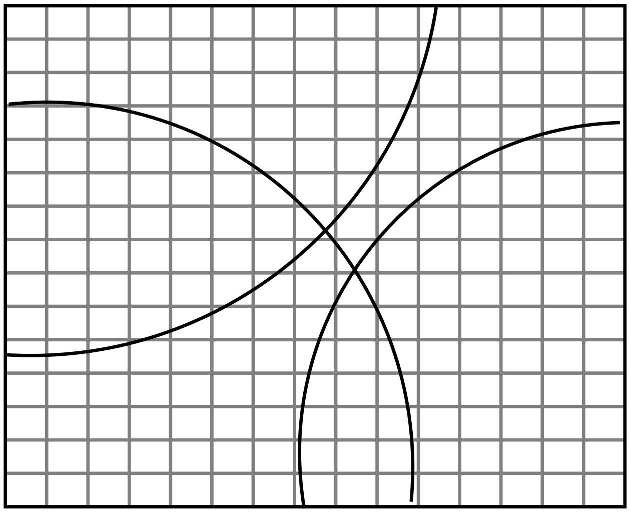


Fig. 8.18: Non-intersection of three circles.

One way to approach this problem is to cast into a optimization problem. If we are a certain distance (in two dimensions) away from a beacon, then we lie on a circle where the radius of the circle is the distance away from the beacon. The object must lie on all of the circles which are have the given distance.

We would like to minimize the distance that our selected point (x, y) lies off of each circle. The distance the point misses the circle from B1 is $|\sqrt{x^2+y^2} - 6838|$. From the individual errors, we can form the total error function by summing up the individual error terms.

$$\begin{aligned} E = \quad & |\sqrt{x^2+y^2} - 6838| + |\sqrt{(x-56)^2+(y-9752)^2} - 4779| \\ & + |\sqrt{(x-9126)^2+(y-7797)^2} - 6245| + |\sqrt{(x-9863)^2+(y-218)^2} - 8938|. \end{aligned}$$

If $E = 0$, then we are at the (x, y) point that matches all four distances.

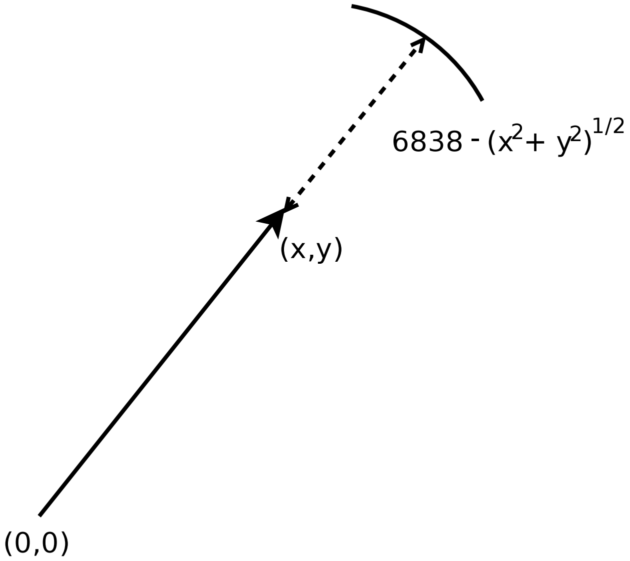


Fig. 8.19: Radial error function.

Since there is measurement error we will have in practice that $E > 0$, so we are looking for the minimum value for E. A traditional multivariate calculus approach is to take partial derivatives and set them to zero. This produces a system of nonlinear equations which must be solved numerically. It is the square root that gives complicated algebra as well as division by zero errors.

One additional problem is the absolute value. The derivative of the absolute $(d/dx)|x| = x/|x|$ is the sign function, $sign(x)$ (not $\sin()$). This is not continuous and will wreak havoc on some optimization codes. In addition, combinations of absolute values can lead to non-single point minimums although unlikely in our case. To address these issues, we change our error function by replacing the absolute value with a square. Indeed this will change the function but will allow for unique mins. Note that for a single component element of the expression, $|f(x, y)|$ the minimum will not move when we move to $[f(x, y)]^2$. For sums, $|f(x, y) + g(x, y)|$ this is no longer true, but not necessarily a bad result.

There are several directions we can head to find the extremal. Many variants of Newton's Method are available. One can imagine custom search algorithms. For simplicity we will leave those approaches to text's on numerical optimization and we will use gradient descent. Recall the definition of the gradient is

$\nabla E = \langle \partial E/\partial x, \partial E/\partial y\rangle$. The updated function to minimize is

$$\begin{aligned} E \quad = \quad & \left(\sqrt{x^2+y^2} - 6838\right)^2 \\ & + \left(\sqrt{(x-56)^2+(y-9752)^2} - 4779\right)^2 \\ & + \left(\sqrt{(x-9126)^2+(y-7797)^2} - 6245\right)^2 \\ & + \left(\sqrt{(x-9863)^2+(y-218)^2} - 8938\right)^2. \end{aligned}$$

Since we are using a numerical method (gradient descent) and thus not an exact method, it makes sense to use a numerical approach to computing the partial derivatives. Recall that the approximation of the derivative is

$$\frac{\partial F}{\partial x_k} \approx \frac{F(x_1, x_2, \ldots, x_k + \Delta x, \ldots, x_n) - F(x_1, x_2, \ldots, x_n)}{\Delta x}$$

for small Δx. For each item in the gradient vector, you can estimate the derivative. This requires two function evaluations, a difference and a multiply. [Precompute $1/\Delta x$ and then multiply.] For the algorithm, if you have rough guess as to location, you can use this for your initial guess for gradient descent. Otherwise you can pick the center or a random point in the search region.

We can use the gradient descent method to find the solution. Set $x_0 = 5000$, $y_0 = 5000$, $k = 0$, $t = 1$:

```
While ($t > t_0$)
    • $u = \nabla E(x_k, y_k)/\|\nabla E(x_k, y_k)\|$
    • $(a, b) = (x_k, y_k) - tu$
    • while $[E(a, b) > E(x_k, y_k)]$
        – $t = t/2$
        – $(a, b) = (x_k, y_k) - tu$
    • $k = k + 1$
    • $(x_k, y_k) = (a, b)$
```

```python
from math import *
# The function definition
def funct(x,y):
   E = (sqrt(x**2 + y**2) - 6838)**2 \
   + (sqrt((x-56)**2 + (y-9752)**2) - 4779)**2 \
   + (sqrt((x-9126)**2  + (y-7797)**2) - 6245)**2 \
   + (sqrt((x-9863)**2 + (y-218)**2) - 8938)**2
   return E
```

```python
# The numerical gradient approximation
def grad(x,y):
    delta = 0.0001
    E = funct(x,y)
    E1 = funct(x+delta,y)
```

(continues on next page)

(continued from previous page)

```
    E2 = funct(x,y+delta)
    dEx = (E1-E)/delta
    dEy = (E2-E)/delta
    return dEx, dEy
```

```
# The size of the vector
def norm(r,s):
    return sqrt(r*r+s*s)

# The step in the direction (u,v)
def step(x,y, u,v,t):
    a = x - t*u
    b = y - t*v
    return a, b
```

```
# Globals
x = 5000
y = 5000
t = 10.0
tsmall = 0.00001

# The descent algorithm
while (t > tsmall):
    dx, dy = grad(x,y)
    size = norm(dx,dy)
    u = dx/size
    v = dy/size
    a,b = step(x,y,u,v,t)
    while (funct(a,b) > funct(x,y)):
        t = 0.5*t
        a,b = step(x,y,u,v,t)
    x,y = a,b

print x, y
```

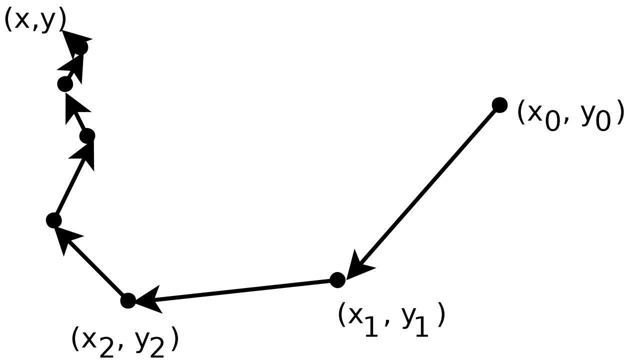


Fig. 8.20: Gradient Descent

The intersection point is $x = 3120$, $y = 6085$. Note that this algorithm is not guaranteed to converge on the solution (the global minimum). It can get trapped in local minima. To address this problem you may re-run the algorithm with different random starting points.

There are plenty of other ways to treat this problem. An image processing approach akin to the Hough Transform (with voting) would also work. It is also possible to lay down a grid and then increment grid cells for each circle that passes through. The cell with the largest value is a candidate for the location. Starting with a course grid and refining the grid is a way to produce a hierarchal method that can have high accuracy but still be fast. See if you can come up with other approaches to this example.

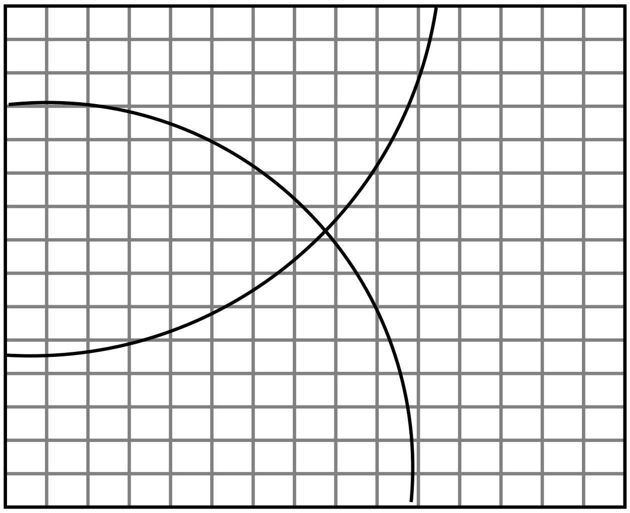


Fig. 8.21: Hough Transform

Footnote

8.4 Proximity Sensors

8.4.1 Switches

The most elementary sensor available is a switch and commonly used as a bump or contact sensor on a robot. Fig. 8.22-(a) gives the schematic for a pull down circuit. When the switch is open, the output (labeled out) is pulled high. This is the connection of the resistor to the 5V line. For many years, digital logic used 5 volts for the high or on and zero volts for low or off. This was the TTL standard. It was based on the bipolar junction transistor technology. With the increasing popularity of CMOS due to lower power consumption, lower voltage devices started to appear. More 3.3 volt system entered common use. Currently, most sensors available at the hobby level are 3.3 volt boards. Many microcontrollers used in hobby class robots have moved down to 1.8 volts.

Lower voltages are used in high performance processors. [Lower voltage means faster switching.] USB and a number of interface circuits still run at 5 volts since this was such a standard for so many years. In terms of the logic it does not matter what voltages are used as long as high and low levels are easily distinguished. It does matter when you are attempting to connect a sensor to controller. We will address this issue later on in the chapter.

When you close a switch (or have a bump sensor contact), it does not behave like you initially expect. The voltage on the output line is given in Fig. 8.22-(b). The problem jumps right out. There is not a single

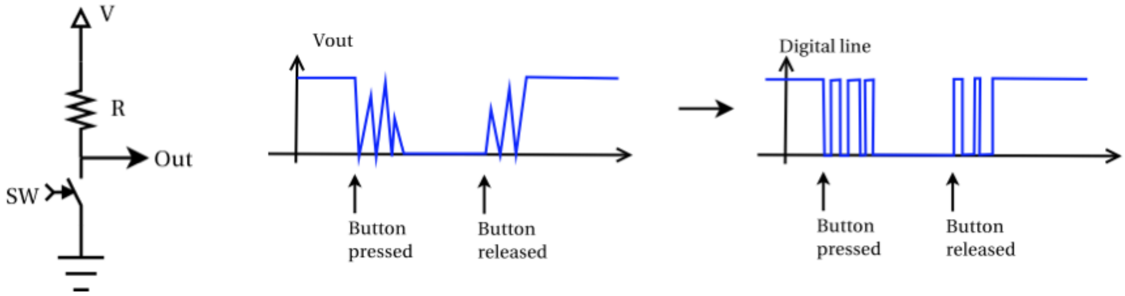


Fig. 8.22: Switch (a) the associated circuit. (b) the generated signal.

close and then open. The problem is that a switch is a mass spring system and will vibrate. At the contact point, the switch is like the basketball player who lowers their dribble hand making the ball bounce faster. The switch vibrates until closed. Since a microcontroller can sample at microsecond intervals, each one of these bounces appears like a button press. So, we don't just generate a single closing of the switch, but we may have hundreds. You can imagine what this text would look like if the keyboard did not address this iiisssssuuueeeeee. The process of removing the false signals, the noise, is called debouncing. There are both hardware and software solutions to the problem.

The first approach we will discuss is given in Fig. 8.23-(a). With the switch open the output again is tied to the high (the 5 volts). The capacitor between the output line and ground will be charged (after a short interval following power-on). When the switch is depressed, the capacitor will discharge through R2. Voltage across a capacitor is the integral of the current flowing. In English this means that the capacitor will smooth the voltage level and cut down on the fast oscillations. It filters out higher frequency noise. The voltage profile is given in Fig. 8.23-(b). The reverse happens when the switch is released. A combination of a resistor and capacitor filters out higher frequencies and is often called an RC filter. Using an RC filter can remove the the alternating voltage levels and appears to solve the problem. However another issue arises.

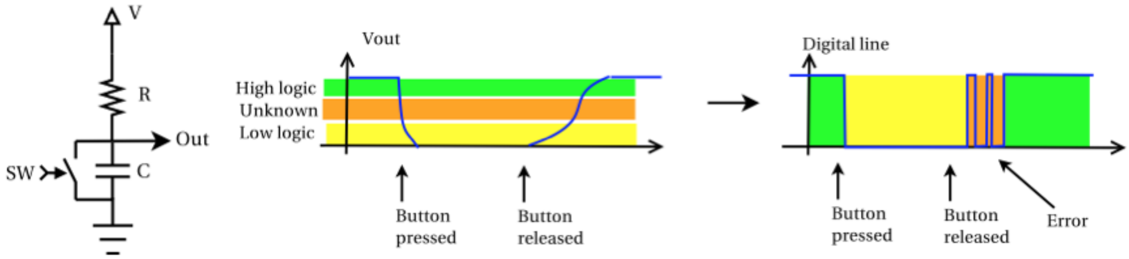


Fig. 8.23: Debounce (a) Basic Hardware, (b) Signal produced.

The system will spend more time in transition; more time in the zone between logic high and logic low. This middle region is not stable for the electronics and can be interpreted by the input of the controller as either logic level, or even jump back and forth. This again produces multiple signals. To solve this aspect, we add another device called a Schmidt trigger. It has a property called hysteresis. Assume for the moment that the input to the Schmidt trigger is currently set at low (close to zero volts). As the voltage increases, the trigger output will stay at low (very close to zero). At some point, the voltage will cross a threshold, V1, for which the trigger will "fire" and the output switches to high. In the other direction, if the input is sitting at high, the output will be high as well (say 5 volts). If the input starts to drop, the ouput will hold at high until the input crosses a threshold, V2. Then the output switches to low. So far we don't have anything that a transistor can't do. However, the magic is in that $V1 > V2$. These values are not the same.

How does that help us? Once the switch is depressed in Fig. 8.24, the voltage across the capacitor starts to drop. But the voltage must drop down to level V2 before the device switches the output to low. Any oscillation above V2 will not change the output. Once the voltage has gone below V2, the device triggers and now the voltage must rise above V1 before another change happens. If the values for V1 and V2 fall outside the indeterminate region for the controller input, we have removed the ambiguous region, and then have removed the mechanical and electrical noise.

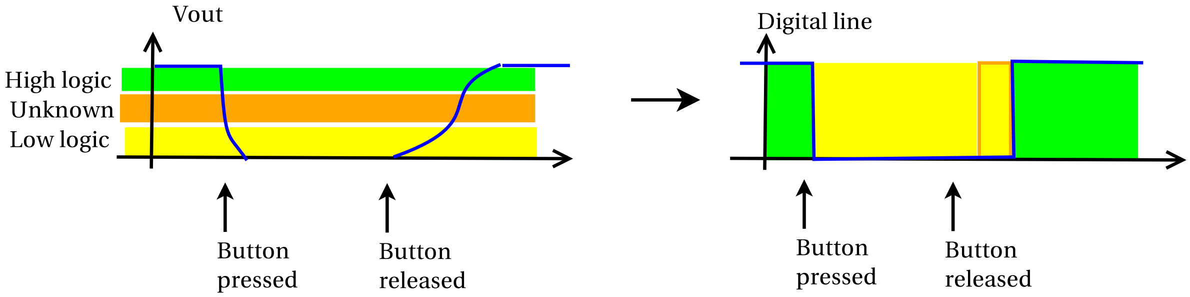


Fig. 8.24: Debounce (a) Improved hardware, (b) Signal produced.

Software solutions are also available and normally approach the problem by introducing delays in the sampling to allow the switch to settle down.

Assume that you have your robot completely surrounded by touch sensors - say 24 sensors. Also assume that your robot has 8 general purpose input-output (GPIO) lines. Seems like you can only use 8 of the 24. This is where multiplexing and demultiplexing integrated circuit chips are really useful, Fig. 8.25. Essentially it is the memory addressing question. The multiplexer unit can select a line to read and make the connection from that line to output. The figure shows 4 input lines, one output line and two select lines. So, one connects the output line on the multiplexer to the GPIO line configured as input. Also needed is connecting the two select line to the GPIO configured as output. With 24 lines, one connects the bottom 5 select lines and the multiplexer output line to six of the GPIO lines. This leaves two GPIO open for other use.

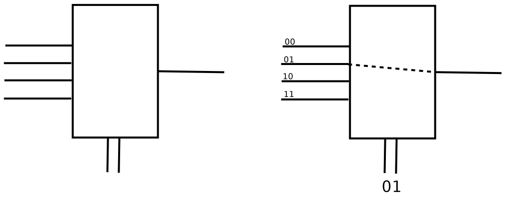


Fig. 8.25: Multiplexers and demultiplexers allow one deal with dozens of devices and a few GPIO.

The only issue is that you might miss a signal because you were looking at a sensor on another line. If you know that the signal will last a minimum amount of time, say 250 ms. Then you need to make sure that you are running an polling loop that takes less than 250 ms to complete. More on multiplexing and encoding can be found in basic texts on digital systems.

8.4.2 Range Sensors

Sensors which estimate the distance are known as range sensors. Range information is one of the main aspects of localization, navigation and mapping. Note that distance sensors which perform short distance measurements are sometimes called proximity detectors. The two main ranging technologies use ultrasound or light. This is a form of active sensing. The device will emit a short pulse and then listen for an echo. The time of the echo provides an estimate of distance using the rate equation. The traveled distance of a sound wave or light wave is given by

$$d = c \cdot t$$

is the distance traveled (round trip), c is the speed of the wave, t is the time of flight.

From this information, we can also indirectly measure velocity by looking at the relative displacement of the fixed object over a short time interval.

Sound and light have vastly different propagation speeds. The speed of sound is roughly 0.3 meters per millisecond where the speed of light is 0.3 meters per nanosecond. This places light at about one million times faster. Off-the-shelf electronics are able to time and process the signals for a ultrasonic basic ranging system. Light is another matter and is much harder to type. Light based rangers, LIDAR or a laser range finder is the preferred ranging hardware. Laser range finders are very accurate, relatively fast and provide a greater number of range points. The downside is that they cost significantly more and can be delicate instruments.

The quality of range sensor data depends on several aspects of the measurement system. Due to discretization, analog to digital conversion, interrupt handling or polling, uncertainties about the exact time of arrival of the reflected signal arise and reduce the accuracy of the estimate. The beam will spread out and makes detection more difficult. The beam may reflect off of the target in a complicated manner. These issues can make it more difficult to detect a reflection. Light will travel in a predictable way as the speed of light does not vary much. The speed of sound is very different however, variations in the density of the air or water can introduce errors in the distance estimation. A more subtle problem can arise if the robot or the target is moving. The Doppler affect can change the frequency of the reflected signal, and again introduce errors.

Sonar

Sonar stands for sound navigation and ranging. The idea is to transmit a packet of ultrasonic pressure waves and listen for the reflection. The time of flight gives the distance. Distance d of the reflecting object can be calculated based on the propagation speed of sound, c, and the time of flight, t:

$$d = \frac{c \cdot t}{2}$$

The speed of sound, c (about 340 m/s), in air is given by

$$c = \sqrt{\gamma R T}$$

where γ is the adiabatic index, R is the gas constant, and T is the gas temperature in Kelvin.

Sonar typically has a frequency: 40 - 180 KHz and so is above most human hearing although some animals may detect the sonar. The pressure waves are normally generated by a Piezo transducer. A transducer is any device that can convert energy in one form to another. In this case, it is a quartz crystal that vibrates when placed in an oscillating electrical current (or generates an electric current when deformed or vibrated).

The sound wave from the transducer will propagate out just like a disturbance in water. Objects will reflect the wave back towards the transducer. Some systems use the same transducer for transmission and reception.

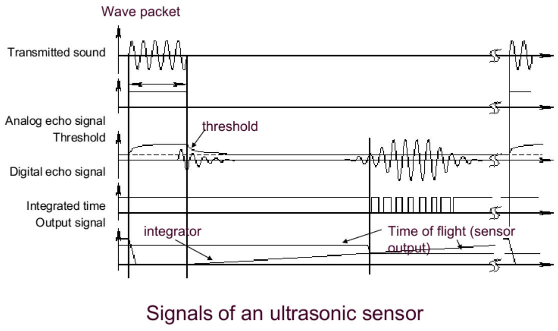


Fig. 8.26: Sonar Echos

Others will have separate transducers. The sound will propagate in a cone shape region with angles varying from 20 to 40 degrees in lower cost units. The vendor will normally provide an intensity cone that shows signal strength in decibels as a function of angle.

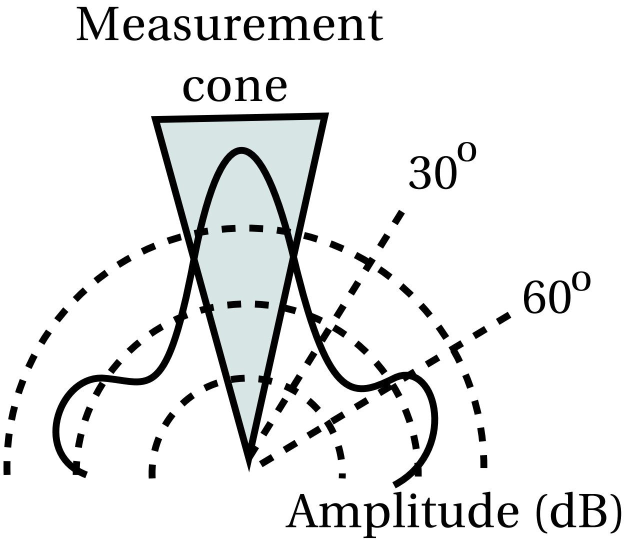


Fig. 8.27: Sonar Cone

One of the obvious problems is with surfaces that absorb a considerable amount of energy. This could be mistaken for no object at all since no bounce is required. Surface properties like surface smoothness and angle of incidence will have a significant impact on the return sign. A surface that has the surface normal not pointed towards the receiver will not deliver as much energy and again may produce incorrect results.

Laser Ranging, aka LIDAR

Laser ranging follows essentially the same ideas that sound ranging does. Light operates at a greater frequency with a much smaller wavelength. This allows for much greater resolution. The downside is the speed of light is so high that it is difficult to measure the return time directly. LIDAR operates by sending a beam out to a target. That beam is reflected back. These two beams are parallel which helps in system design to filter out interference. Once the round trip time is determined, the distance is easily computed. The laser is placed on a panning system which then sweeps the field. This will provide a data set which has angle and distance information from the LIDAR to the targets.

On most systems the round trip time is not timed (since sub nanosecond timers are required). Time of flight measurement can be done by a phase shift technique. An interference pattern between the reflected wave and the emitted wave is setup. The resulting phase shift can be extracted. This allows one to compute the propagation delay and thus the distance traveled. A frequency modulated continuous wave is used and the beat frequency formed by interference between reflected and transmitted waves form the basis of the phase shift. A pulsed laser is often used instead of a continuous beam laser. This can reduce power requirements.

From Fig. 8.28, the beam is split at point s. One branch travels to the object and back, and then up to the measurement unit for a distance of $L + 2D$. The other branch just travels up to the measurement unit for a distance of L. The difference between these two distances is $(L + 2D) - L = 2D$. This difference can be expressed in terms of the phase shift:

$$2D = \frac{\theta}{2\pi}\lambda$$

where θ is the phase shift and λ is the wavelength. If the total beam distance covered is D', c is the speed of light, f is the modulating frequency, we see

$$D' = L + 2D = L + \frac{\theta}{2\pi}\lambda, \qquad \lambda = \frac{c}{f}.$$

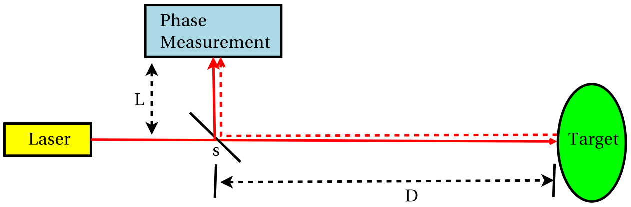
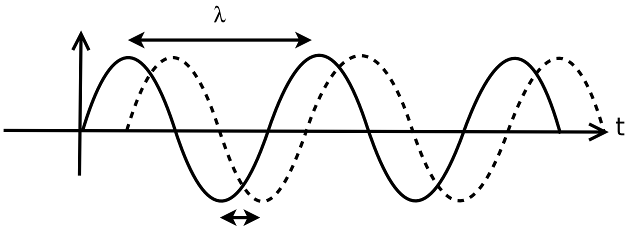


Fig. 8.28: The basic operational diagram for a laser ranger.



For reference, if $f = 5$ Mhz then $\lambda = 60$ meters. This allows us to compute D as a function of θ

$$D = \frac{\lambda}{4\pi}\theta.$$

One problem that is immediately clear is that the range estimate is not unique. This is easy to see. A distance difference of a half wavelength would generate the same phase shift as 1.5 wavelengths and 2.5 wavelengths and 3.5 wavelengths, etc. For example if $\lambda = 60$ then a target at 5, 35, 65, ... meters will give the same phase shift.

Example

Assume you are using a laser diode to build a distance sensor.

- What is the wavelength of the modulated frequency of 12MHz?
- If you measure a 20 degree phase shift, this value corresponds to what distances?
- What other modulation frequency would be a good choice to isolate the value? (show this)
- How would you do the modulation and phase shift measurement?

The wavelength is given by $\lambda = c/f = 3(10^8)/(12(10^6)) = 25$ meters. A 20 degree shift is $(20/360)$ of the wavelength, so we get

$$(20/360) * 25 \approx 1.389m$$

The actual distance is 1/2 of this value since the beam travels to the obstacle and back: $0.6945m$ but we will do our computations on the full trip and then at the very end, cut our number on half. This would correspond to 1.389, 26.389, 51.389, 76.389, 101.389, 126.389, or

$$1.389 + 25n \text{ for } n = 0, 1, 2, 3...$$

If you select different frequencies that are multiplies of each other, say 5MHz and 10MHz, you can see that it does not help much. You need frequencies that are different enough. As long as our values are relatively prime, frequency selection is pretty open. Factors of 12 are 2, 3, 4. So 5 Mhz would work (as would 17 Mhz and many others) for some distance out. Using 5Mhz, we have a wavelength of 60 meters.

Assume that you use the 5Mhz frequency and you measure a phase shift of 158.334 degrees. This must correspond to the distances

$$\begin{aligned}(158.334/360) * 60 + 60m \approx 26.389 + 60m, \quad m = 0, 1, 2...\\ = 26.389, 86.389, 146.389, 206.389, ...\end{aligned}$$

These agree at 26.389. Since we cut the distance in half, the object must be at D = 13.1945. You might wonder if that was the only overlap. We did not go out very far and it could be possible that it repeats.

To find where the two give the same value, set

$$1.398 + 25n = 26.389 + 60m,$$

and obtain

$$m = 5(n - 1)/12.$$

We thus need $5(n - 1)/12$ to be an integer for these two to agree. Inspection tells us that $n - 1 = 12$ or $n = 13$. When $n = 13$ then $m = 5$. If you don't see this, then you can run a simple Python program to check. Step up the values: $n = 0, 1, 2, 3...$ and see when you get an integer for m:

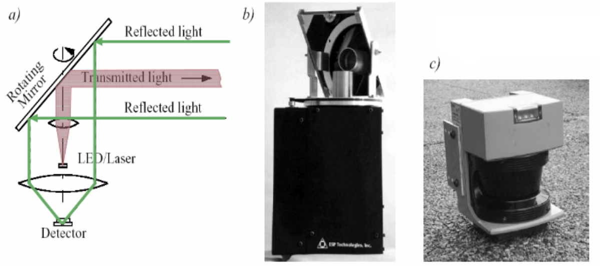
```
>>> for n in range(20):
...    m = 5.0*(n-1)/12.0
...    print "n = ", n, "   m = ", m
...
```

The output becomes

```
n =  0   m =  -0.416666666667
n =  1   m =  0.0
n =  2   m =  0.416666666667
n =  3   m =  0.833333333333
n =  4   m =  1.25
n =  5   m =  1.66666666667
n =  6   m =  2.08333333333
n =  7   m =  2.5
n =  8   m =  2.91666666667
n =  9   m =  3.33333333333
n =  10   m =  3.75
n =  11   m =  4.16666666667
n =  12   m =  4.58333333333
n =  13   m =  5.0
n =  14   m =  5.41666666667
n =  15   m =  5.83333333333
n =  16   m =  6.25
n =  17   m =  6.66666666667
n =  18   m =  7.08333333333
n =  19   m =  7.5
```

So $m = 5$. This gives isolation out to about 26.389 + 60(5) = 326.389 meters using the two frequences. Remember to cut this in half, so uniqueness range is 163 meters.

Just as with sonar, errors can arise based on the hardware construction and the reflected object surface. Confidence in the range (phase/time estimate) is inversely proportional to the square of the received signal amplitude. Dark distant objects do not produce as good of range estimate as closer brighter objects.



8.5 Problems

1. Assume that you are working in a large event center which has beacons located around the facility. Estimate the location of a robot, (a, b, c), if the (x, y, z) location of the beacon and the distance from the beacon to the robot, d, are given in the table below.

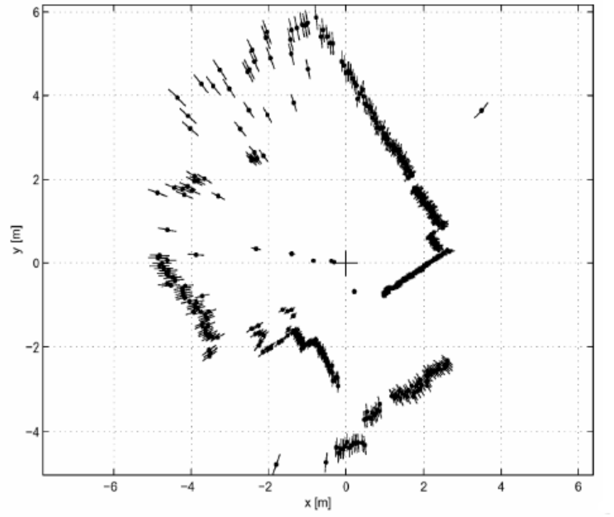


Fig. 8.29: Typical range image of a 2D laser range sensor with a rotating mirror. The length of the lines through the measurement points indicate the uncertainties.

x	y	z	d
884	554	713	222
120	703	771	843
938	871	583	436
967	653	46	529
593	186	989	610

2. If you are using a laser diode to build a distance sensor, you need some method to determine the travel time. Instead of using pulses and a clock, try using phase shifts. What is the wavelength of the modulated frequency of 10MHz? If you measure a 10 degree phase shift, this value corresponds to what distances? What if the phase shift measurement has noise: zero mean with standard deviation 0.1? How does one get a good estimate of position if the ranges to be measured are from 20 meters to 250 meters?

3. Write a Python function to simulate a LIDAR. The simulated LIDAR will scan a map and return the distance array. We assume that the obstacle map is stored in a two dimensional gridmap, call it map. You can use a simple gridmap which uses 0 for a free space cell and 1 for an occupied cell. The robot pose (location and orientation) will be stored in a list called pose which will hold x, y, theta (where these are in map cordinates). Place LIDAR parameters into a list which has total sweep angle, sweep increment angle and range. The function call will be data = lidar(pose, objmap, params) in which data is the 1D array of distance values to obstacles as a function of angle. Test this on a map with more than one obstacle. Appendix A shows how one may generate a map in a bit map editor like GIMP and then export in a plain text format which is easily read into a Python (Numpy) array. [Although you can fill the grid by a python function which sets the values, using the bit map editor will be much easier in the long run.]

CHAPTER NINE

MACHINE VISION AND APPLICATIONS TO ROBOTICS

9.1 Triangulation and Structured Light

The last section explored using time of flight to determine the distance traveled. Distances can also be determined by using geometric properties of the object (or image of the object). The approach is to project a well defined light pattern (points, lines) onto the environment and objects within. The reflected light is then captured by a photo-sensitive line or matrix (camera) sensor device. Simple triangulation then allows the computation of the distance in question.

Kinect and ASUS sensors use arrays of projected IR dots. The size of the dot indicates distance of the dot. If the size of object is known, then triangulation can be done without projecting light. Standard computer vision techniques can recover relative image size.

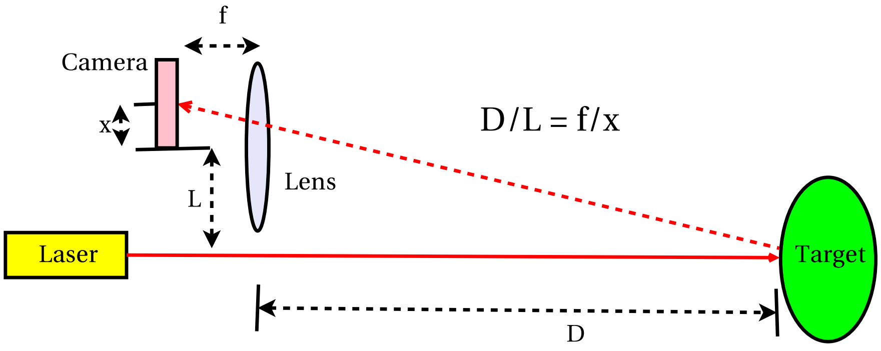


Fig. 9.1: Laser Triangulation.

Laser Triangulation is done by setting the measurement apparatus up so that simple trigonometry can be used to measure distance, Fig. 9.1. In this case, the laser setup uses similar triangles making the mathematics much simpler. The distance D is given by

$$D = \frac{fL}{x}.$$

All sensing involves error. We will address types of errors and how to mitigate errors in the chapter on filtering. To get a feel for how errors affect a result, one can run a simple numerical experiment. Compute the range of the input values based on the error. Plug the high and low values in and you can compute the range on the output value. The following example provides some details.

Example

Assume that for the triangulation setup in Fig. 9.1, we have $f = 8$mm, $L = 3$cm and $x = 2$mm. Using the formula we see that $D = 0.8 * 3/0.2 = 12$cm. What if there is error in the x or L measurement?

What is the error in the distance if we know that x has a max of 20% error? A 20% variation means that our value ranges between $[1.6, 2.4]$. We can plug the two values in and see what the range in D is. This works because D is a monotonic function of both x and L[1] Plugging these values in we have $10 \leq D \leq 15$. Which gives $15/12 - 1 = 0.25$ or 25% error. If we have 10% error in L, it gives $10.8 \leq D \leq 13.2$ or a 10% error off of 12. To combine these, we look for the largest and smallest values possible for D, given the range in input values. For $x = 1.6$ and $L = 3.3$ we get $D = 16.5$. Likewise for $x = 2.4$ and $L = 2.7$ we get $D = 9$. The max of the two is a 37.5% error (from $D = 12$cm).

Is there a way to estimate combined error from the equation? In Calculus, the total derivative for $f(x, y, z)$:

$$df = \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial y}dy + \frac{\partial f}{\partial z}dz$$

can be used to gain an error formula:

$$E = \Delta f \approx \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y + \frac{\partial f}{\partial z}\Delta z.$$

Using the values from the last example, $f = 8$mm, $L = 3$cm and $x = 2$mm and the variations, x has a max of 20% error and 10% error in L, find the error estimate for D. Using $\Delta L = \pm 0.3$, $\Delta x = \pm 0.04$, and variation in f means $\Delta f = 0$ we have

$$E = \frac{L}{x}\Delta f + \frac{f}{x}\Delta L - \frac{fL}{x^2}\Delta x = \frac{3}{0.2}(0) + \frac{0.8}{0.2}(\pm 0.3) - \frac{(0.8)(3)}{(0.2)^2}(\pm 0.04) = \pm 3.6$$

This estimates an error of 25%. This turns out to be not so accurate. A 20% error is a bit too large for the linear approximation to be close, but works as a rough estimate.

A way to modify the previous laser distance example is a common industrial vision setup. Look at the diagram and see what formulas can we derive. Note that:

$$\left(\frac{z}{x}\right) = \left(\frac{f}{u}\right) \qquad \text{and} \qquad \tan(\alpha) = \left(\frac{z}{b-x}\right)$$

Flip the second formula:

$$\cot(\alpha) = \left(\frac{b-x}{z}\right)$$

Then multiply by z:

$$(z)\cot(\alpha) = (b - x)$$

Move the x over:

$$(z)\cot(\alpha) + x = b$$

From the first ratio: $z = \dfrac{fx}{u}$.

[1] Monotonic means that $f'>0$ or $f'<0$ in the interval of interest.

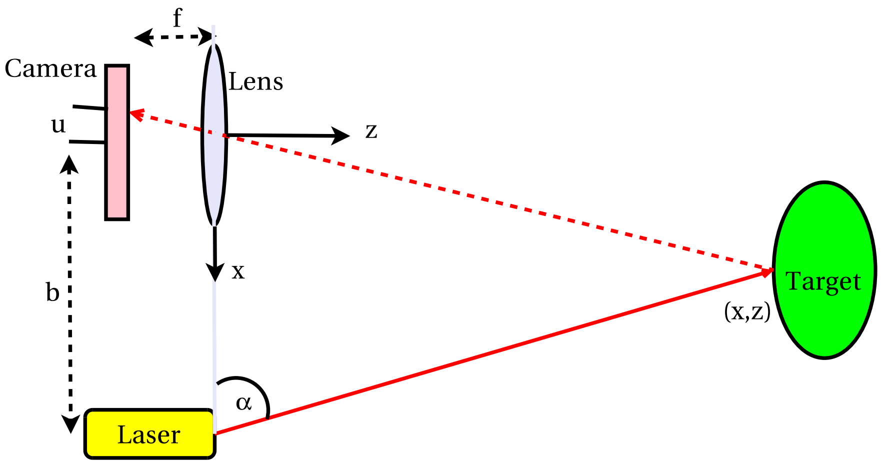


Fig. 9.2: Computer Vision

Plug this in for z:

$$\left(\frac{fx}{u}\right)\cot(\alpha) + x = b.$$

Factor out the x and divide the rest over:

$$x = \frac{b}{\left(\frac{f}{u}\right)\cot(\alpha) + 1}$$

then using

$$z = \frac{fx}{u} = \left(\frac{f}{u}\right)\frac{bu}{\left(\frac{f}{u}\right)\cot(\alpha) + 1}.$$

Summarizing the formulas:

$$x = \frac{bu}{f\cot\alpha + u}, \quad z = \frac{bf}{f\cot\alpha + u} \tag{9.1}$$

What are x and z if b = 20cm, f = 2cm, α = 60deg, and u = 7mm? So, using these formulas:

$$x = 20 * 0.7/(2\cot(60) + 0.7) = 7.55cm,$$

$$z = 20 * 2/(2\cot(60) + 0.7) = 21.57cm.$$

The Sharp distance sensor uses a very similar approach to estimate distances. The displacement of the beam center on the beam detector is used for the distance estimate, see Fig. 9.3. Distance D is given by

$$D = \frac{fb}{2d}.$$

Because the focal length is small, the range of distances are limited by the resolution of the detector (which provides D). The Sharp detector returns the distance estimate as an analog voltage. An analog to digital converter can be used to provide the numerical value. In practice, the relation between voltage and distance is not linear and some calibration in software is required.

Another approach used in machine vision is **Structured Light**. A known pattern of light is projected onto the environment. Common patterns are dots, stripes and grids. A camera will view the instrumented scene and determine the object heights using geometry.

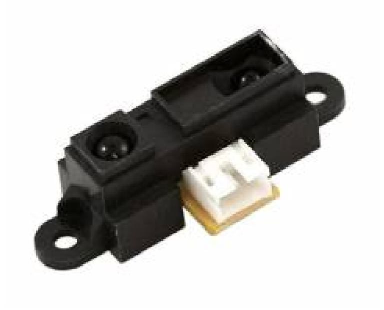


Fig. 9.3: Sensor package.

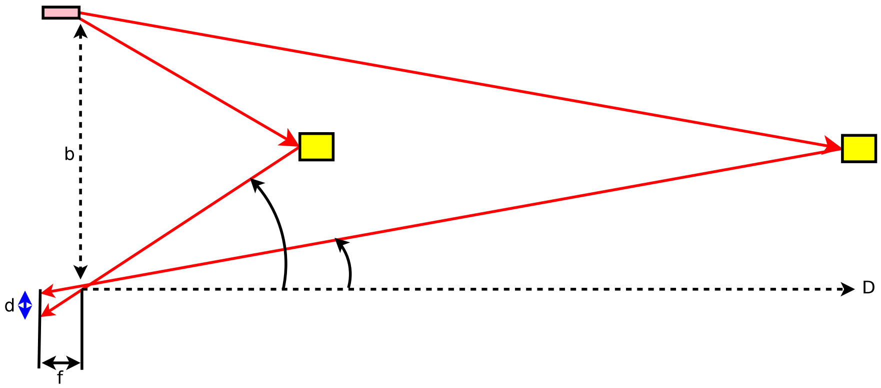


Fig. 9.4: The triangulation used to calculate distance

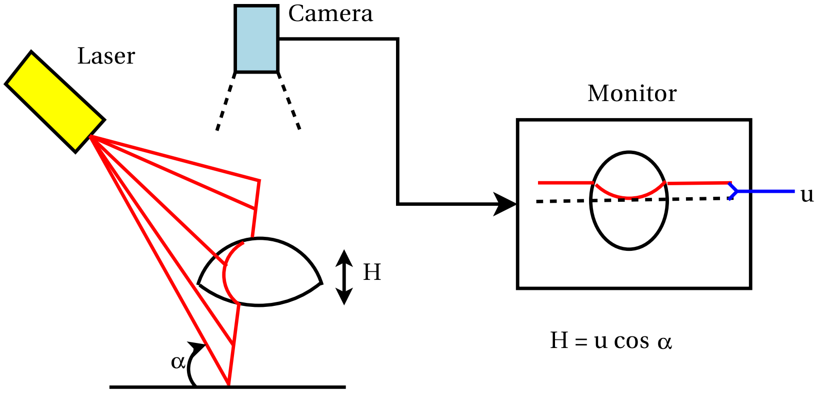


Fig. 9.5: Structured light.

9.2 Stereo Cameras

We can build cameras that are much more accurate than the eye, however, understanding that the image belongs to a particular object is a much harder task. Implicit in the process is our stereo vision. We have the ability to reconstruct the 3D world through our eyes; an ability for which significant effort has been expended to duplicate in computer science. A branch of modern computer vision, stereo vision, uses multiple cameras and algorithms to reconstruct and understand the environment. Some of the tools developed to do this have been widely distributed. For example, the automatic stitching of images that cameras perform when building panoramas.

This is successfully employed using pairs of cameras which do a good job at reconstructing the 3D world we live in. The process requires one to determine the position shift in pixels of a fixed object. The difficult part is automatically identifying common points in the image. Significant effort has been invested in the determination of common features between two images. Several well known algorithms such as SIFT and SURF are available now to simplify this process. Once that is done, it is easy to triangulate the depth of the point, Fig. 9.6.

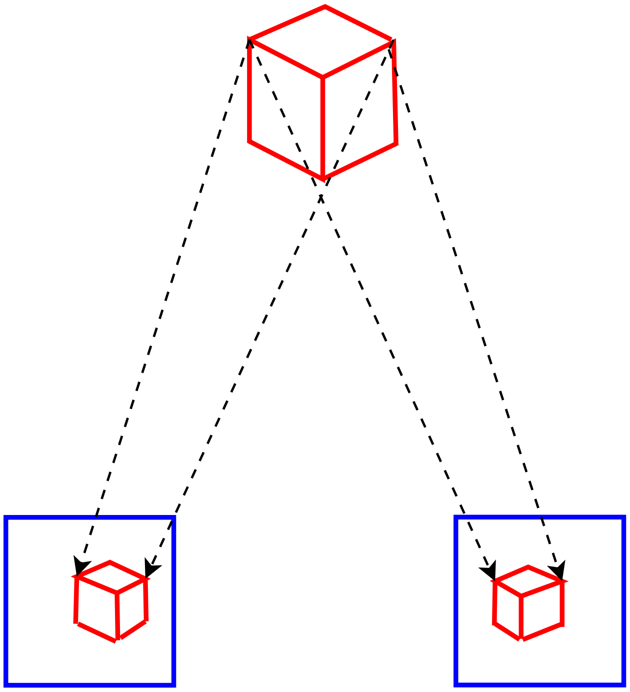


Fig. 9.6: Seeing in three dimensions with a pair of calibrated cameras: determining depth using basic Trigonometry.

Define a coordinate system where the horizontal axis is x and the vertical axis is z. Let the focal point of the left camera be at the origin of the $x-z$ coordinate system. Using both cameras, we would like to find the coordinates (x, z) for the point w. Assume that we are given the focal depth f (positive value) and pixel offsets in image sensor v_1, v_2 all as *unsigned* (positive) quantities. Then

$$\left(\frac{z}{x}\right) = \left(\frac{f}{v_1}\right), \qquad \left(\frac{z}{b-x}\right) = \left(\frac{f}{v_2}\right)$$

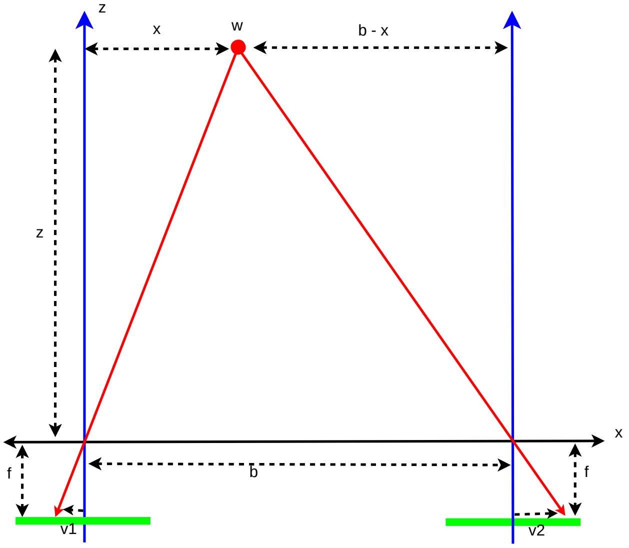


Fig. 9.7: Seeing in three dimensions with a pair of calibrated cameras: determining depth using basic Trigonometry.

Cross multiply and set equal to common fraction; then remove fractions:

$$\left(\frac{v_1}{x}\right) = \left(\frac{f}{z}\right) = \left(\frac{v_2}{b-x}\right) \quad \Rightarrow \quad v_2 x = v_1(b-x) = v_1 b - v_1 x \Rightarrow (v_1+v_2)x = v_1 b$$

Solving for x, we obtain the equation below. Plugging this into $z = fx/v_1$ we obtain the equation for z.

$$x = \frac{v_1 b}{v_1 + v_2}, \quad z = \frac{fb}{v_1 + v_2} \tag{9.2}$$

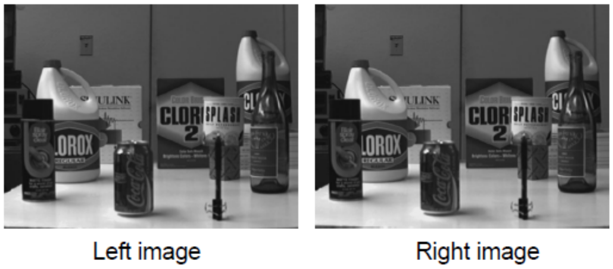


Fig. 9.8: Seeing in three dimensions with a pair of calibrated cameras: building a disparity (depth) map.

Once depth for the collection of feature points are known, depth for surrounding points is inferred. This allows the construction of a disparity map which maps grey scale values to pixel. It is a depth map which is shown in Fig. 9.9. A depth map is a useful tool in object identification. The depth map can be used for segmentation, the process by which we separate an image into distinct components or objects. Once we have the object segmented, then we may lookup in a shape database to determine what the object is, known as object recognition. The depth map is one of several ways to perform object recognition and is a useful tool if we have already computed the depth map.

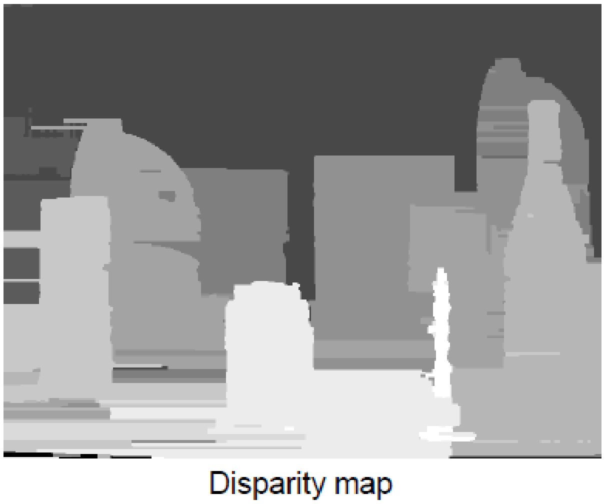


Fig. 9.9: Seeing in three dimensions with a pair of calibrated cameras: reconstructing the 3D world.

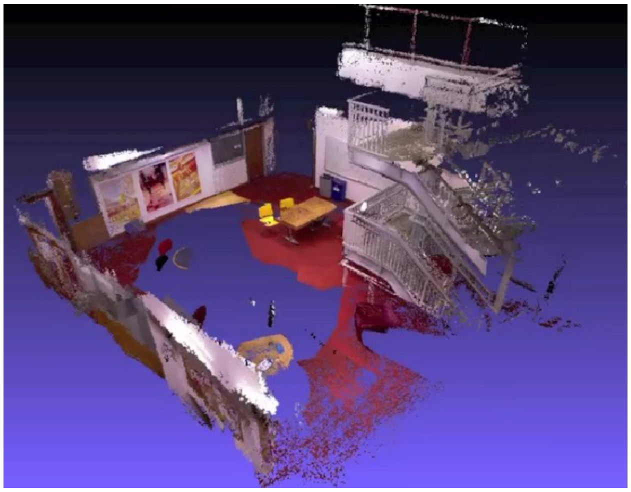


Fig. 9.10: Seeing in three dimensions with RGBD sensors.

Once a depth map is made, reconstruction of the environment can follow. Essentially a 3D CAD type representation of the world surrounding the robot. Thus the environment is mapped in 3D. This is useful for robots which perform remote reconnaissance as well as for robots which need to navigate through the environment according to some plan. The map building process normally places the robot in the map, known as localization. Thus we can compute optimal paths and safe trajectories.

9.2.1 Depth Sensing Cameras

We also have a choice of sensors which can directly measure the depth of field. These are known as time of flight cameras or 3D cameras. The Microsoft Kinect is a common example. The units range significantly in cost depending on accuracy, range and speed of the device. These devices directly provide depth without having to compute a disparity map or some other intermediate data set. They are very helpful in doing 3D reconstructions of the environment.

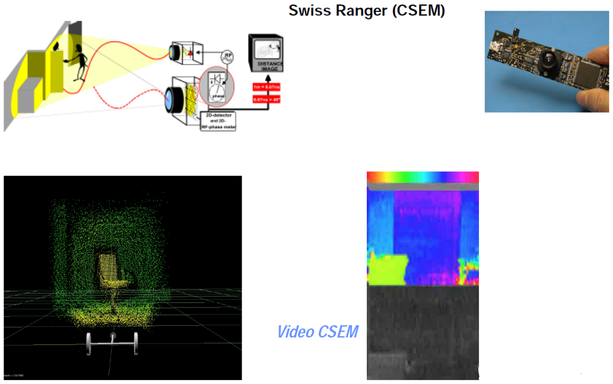


Fig. 9.11: 3D Camera

9.3 Lane Detection

Note: Write the section on lane detection. This will be based on OpenCV.

9.4 Traffic Signals and Traffic Signs

Note: Write the sections on detecting and recognizing standard traffic signals and signs.

9.5 Machine Learning

Note: Write the sections on learning (might need GPU covered separately)

This section should cover learning basics, standard methods (CNNs, RL, ...) and some implementations such as pytorch, ISAAC/Jetpack, CUDA on a Jetson.

9.6 Problems

1. Assume you have a laser triangulation system as shown in Fig. 9.2 given by (9.1) and that $f = 8$mm, $b = 30$cm. What are the ranges for α and u if we need to measure target distances in a region (in cm) $20 < z < 100$ and $10 < x < 30$?
2. Assume you have two cameras that are calibrated into a stereo pair with a baseline of 10cm, and focal depth of 7mm. If the error is 10% on v_1 and v_2, $v_1 = 2$mm and $v_2 = 3$mm, what is the error on the depth measurement z? Your answer should be a percentage relative to the error free number. Hint: If $v_1 = 2$ then a 10% error ranges from 1.8 to 2.2. [Although not required, another way to approach this problem is the total differential from calculus.]
3. Assume you have two cameras that are calibrated into a stereo pair with an estimated baseline of 10cm, and focal depth of 10mm. If the error is 10% on the baseline, what is error on the depth measurement z with $v_1 = 2$mm and $v_2 = 3$mm? Your answer should be a percentage relative to the error free number. See the hint above.
4. With a single camera, explain how a straight line (produced by a laser) can resolve depth information.

Note: Complete this chapter

CHAPTER TEN

MOTION CONTROL

One of the core properties of a robot is the ability to move. In this chapter we discuss moving the robot in a predetermined or controlled manner. Given a particular path or type of motion, we address how to perform it. This falls under a large engineering discipline known as controls and we will just touch on a few concepts here. The main example we will use is the differential drive robot. It is a significant robot in its own right, and presents many of the challenges found in most mobile robotic systems.

10.1 Basic Controls

In the Simulation Chapter, we saw that a parametric curve can be used to generate a path for a robot. Two issues quickly arise. First, noise, errors in velocity and path discretization will cause the vehicle to drift off of the computed path. The formulas continue to drive the robot as if we are still on the correct trajectory. This is known as *Open Loop* control. This means there is no feedback from the sensors to correct for drift, discretization or path error. The drift will accumulate over time and no matter how accurately you compute the path, the robot will continue to stray from the path. The only way to correct the drift is to have some type of feedback in the control code.

The second issue is that a given parametric curve sets velocity as part of the path. Later in this chapter we will separate the path from velocity in the parametric functions, but it still leaves no room to adapt to changes. Just as the position can drift, so can the velocity. We need feedback for velocity as well. In addition, it is important to be able to adapt to the environment, for example if we need to slow down at some point to avoid a collision. So, feedback in our systems turns out to be essential.

As we discussed in the simulation chapter, the approach to minimize the drift is to replace the parametric path with a feedback control system based on sensing the environment. Thus if we missed hitting the target point in the last segment, we don't use next point in the precomputed path, but have the controller provide a modified direction, see Fig. 10.1. This keeps the robot headed towards the desired path by using position feedback. A simple stop, rotate to orient and drive approach will work in very simple applications. How about an approach that corrects as we drive? When we drive a car, we continually monitor our position on the road and correct as needed.[1]

[1] Texting while driving is an example of open loop control. Not looking at the road means you do not use sensor feedback to correct the path resulting in a collision.

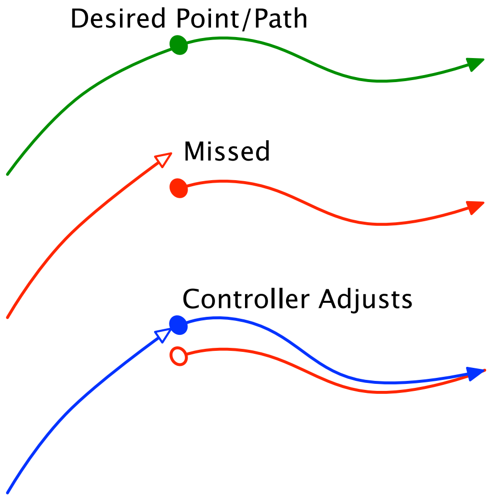


Fig. 10.1: Using position feedback as a way to mitigate drift.

10.1.1 Heading to Goal

After driving a few robots around on less than perfect terrain, it is clear that driving a straight line can be hard. Assuming the goal is actually driving to a goal and not as much staying on the line, how can we correct during the route to keep our orientation towards the goal point? Assume that you know your orientation error, Fig. 10.2.

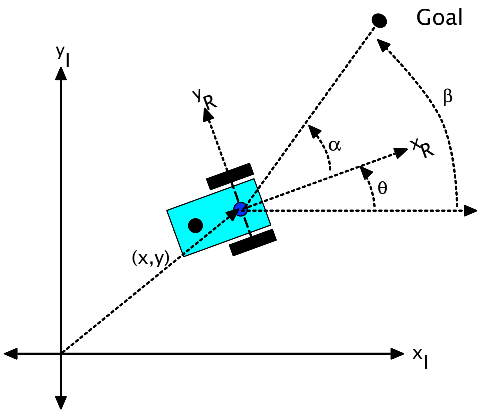


Fig. 10.2: Orientation error of α.

Recall the alternate form of the differential drive equations:

$$v = \frac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)$$

$$\dot{\theta} = \frac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2).$$

We are interested in finding wheel velocities such that v is constant, v_c and $\alpha \to 0$. So we start with wheel velocities of the form

$$\dot{\phi}_1 = \frac{1}{r}(v_c + Lk\alpha)$$

$$\dot{\phi}_2 = \frac{1}{r}(v_c - Lk\alpha)$$

This gives us

$$v = v_c$$

$$\dot{\theta} = k\alpha.$$

Assume that the robot is relatively far from the goal with respect to robot speed. If so, then $\dot{\beta}$ is small. Since $\beta = \omega + \alpha$ it implies that $\dot{\alpha} \approx -\dot{\omega}$ and we have

$$\dot{\alpha} = -k\alpha.$$

The solution to this differential equation is

$$\alpha(t) = \alpha(0)e^{-kt}$$

which has $\alpha(t) \to 0$ as $t \to 0$ if $k > 0$. Large values of k produce fast response times and small values produce slow response. However, in implementation one works with a discrete (digital) time controller. Large values can cause the robot to sweep past the target and oscillate about the goal line. These oscillations can even grow causing unstable motion. Normally we select small values to start and observe how well it controls the robot. Increasing k as needed when the response is insufficient. Selecting good values of k, known as the gain, is discussed in the PID control section.

10.1.2 Classical Controls

Controls is a large interdisciplinary subject. Many engineers have expressed the sentiment that robotics and controls are the same subject. That view is up for debate, however everyone agrees that controls is a very important aspect to robotics. In the next few sections, we briefly touch on the topic. We begin with some terms found in the controls literature.

- Open Loop - device control without using sensor feedback.
- Closed Loop - device control using sensor feedback.
- Bang-Bang (On-off) Control - A control approach that turns the actuator or motor completely on or off without using proportional values.
- P Control - Proportional control, using the error between the desired state and sensed state to control the device.
- PD Control - Proportional-Derivative control, using the error between the desired state and sensed state and rate of change of the error to control the device.
- PID Control - Proportional-Integral-Derivative control, using the error, rate of change of the error and a history of the error to control the device.

PID is one of the most popular control approaches in industry. It has wide application due to ease of use and reasonable effectiveness. Related to PID are PD, PI and P which are just versions of PID where selected terms are set to zero in the formulas. From holding a temperature in your house or oven, to cruise control on your car, to managing flow rates in industrial plants to many more applications, control systems are an essential aspect to an engineered solution. We will be using these algorithms to set position, velocity and force in our robots.

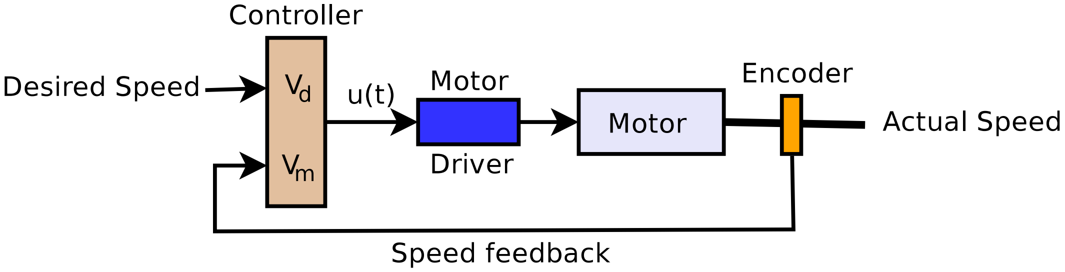


Fig. 10.3: Feedback and Control for motor speed.

Fig. 10.3 shows the basic feedback loop for controlling the speed on a motor. We will assume this encoder returns angular velocity (rpm) although in practice they return a signal which needs to be translated to rpm. This can be easily done as part of the controller block. The controller block is the part that takes the two signals, desired speed and actual speed, and decides what command to give to the motor driver. The motor driver will then power the motor.

Very little must change in the diagram if we wish to build a servo or manipulator control system. For a servo, the encoder returns absolute angle information. If the motor is attached to a mechanical arm, then the encoder is the device that returns the location of the end-effector. Most of our discussion below will use motor speed as an example, but these ideas apply to articulated systems as well.

We will be concerned with three quantities here. First, the desired configuration, which in Fig. 10.3 is v_d. Second, the measured or actual configuration, v_m. And third, the control signal delivered to the actuator control unit, $u(t)$.

10.1.3 Bang-Bang Control

Bang-Bang control or On-Off control attempts move the system to the desired configuration by using a series of on/off signals:

$$u(t) = \begin{cases} u_c & \text{if } v_m(t) < v_d \\ 0 & \text{if } v_m(t) \geq v_d \end{cases}$$

where u_c is a constant. When $v_m < v_d$ a constant power is applied to the motor. Since the system is digital, the signal is sampled every Δt seconds. There are also delays in the time required to run the control algorithm. So, the power will normally be applied past the point where the motor speed exceeds the desired speed.

In this system, a fixed control effort is used and no attempt at scaling it based on measured speeds is done. The obvious result is an oscillation of the actual speed around the set speed, Fig. 10.4. This approach is fine for systems with slow dynamics (significant inertial) where one only wants to be close to the set value; such as house temperature. For higher response systems, this approach can feel rough as it jumps from off to on and back. It can also become unstable with very rapid response times.

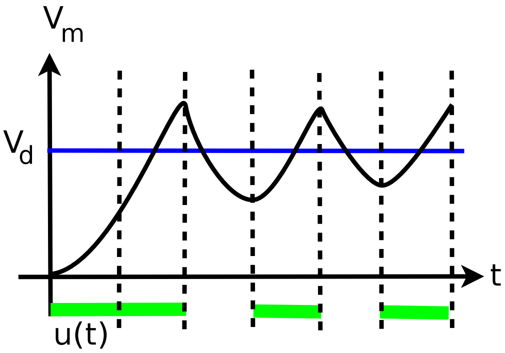


Fig. 10.4: Bang-Bang or On-Off Control.

We can smooth this out a bit by placing a range for turning on and off the motor. Assume that $v_{\text{on}} < v_d < v_{\text{off}}$ then

$$u(t) = \begin{cases} u_c & \text{if } v_m(t) \leq v_{\text{on}} \\ 0 & \text{if } v_m(t) \geq v_{\text{off}} \end{cases} .$$

This is not a good velocity or position control system for the robot. We clearly want to take into account how far off the set value we are and adjust our control effort. Meaning we want to base our control effort on the amount of error. This is the approach used with the family of P, PD and PID controllers described below.

10.1.4 Proportional Control

The idea of proportional control is to set the control effort proportionally to the error, $e(t) = v_d - v_m$:

$$u(t) = K_P e(t).$$

This is an intuitive thing to try and in some applications works well. A proportional control is mathematically more complicated that the Bang Bang control discussed above. It overcomes some of the issues in on-off controllers since they can continuously vary their output. The constant K_P is known as a gain, in the case the proportional gain. A high proportional gain results in a large change in the output for a given change in the error. If the proportional gain is too high, the system can become unstable, oscillation (like we saw with the On-Off controller) can occur. In contrast, a small gain results in a small output response to a large input error, and a less responsive or less sensitive controller.

Example of how to use a p-controller to drive a robot along a path

In this example, we extend the previous example of holding a fixed heading. Assume that you have two points $t_0 : (x_0, y_0)$ and $t_1 : (x_1, y_1)$. If the robot is at the first point with an unknown orientation, how does one drive the robot to the second point in a smooth motion? We can define two error terms

$$e_1(t) = \sqrt{(x_1 - x_m)^2 + (y_1 - y_m)^2},$$

$$e_2(t) = \text{atan2}((y_1 - y_m), (x_1 - x_m)) - \theta_m.$$

e_1 measures the difference between current location and goal location, and e_2 measures the difference between current vehicle direction and the direction to the goal. We can try a proportional control on the orientation of the vehicle by proportionally adjusting the differences in the wheel velocities. Let v be the base speed. Then select the speed for the wheels:

$$\dot{\phi}_1 = v + k_1 e_2(t), \quad \dot{\phi}_2 = v - k_1 e_2(t).$$

The robot can move at a fixed speed until it gets close to goal and then can ramp down the speed by using the e_1 error value. If $e_1(t) < d$ then

$$\dot{\phi}_1 = k_2 e_1(t)(v + k_1 e_2(t)), \quad \dot{\phi}_1 = k_2 e_1(t)(v - k_1 e_2(t)).$$

The value k_2 can be selected so the speed is continuous across the $e_1(t) < d$ jump. Selecting some arbitrary values, $r = 20$, $L = 12$, $\Delta t = 0.01$, $k_1 = 2.0$ and $k_2 = 0.2$. k_2 is selected for continuity on wheel speed. k_1 was derived experimentally. The start point is (0,0) and endpoint is (40,60). The result is given in Fig. 10.5.

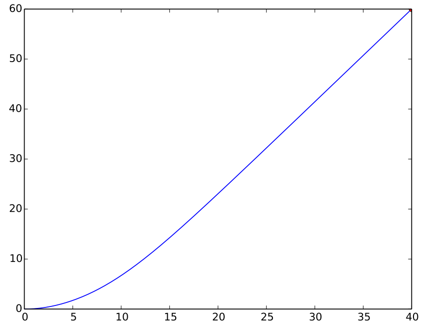


Fig. 10.5: A noise free example.

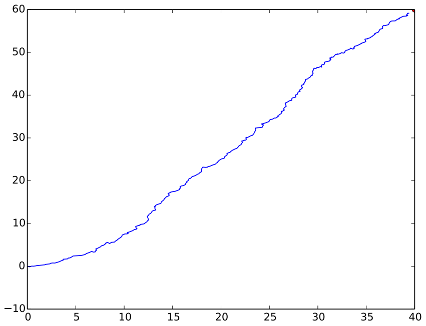


Fig. 10.6: Adding noise to the wheels.

```
r = 20.0
l = 12.0
dt   = 0.01
Tend = 6.0
```

(continues on next page)

(continued from previous page)

```
N = int(Tend/dt)

xend = 40
yend = 60
v = 1.0
k1 = 2.0
k2 = 0.2

x = np.zeros(N)
y = np.zeros(N)
th = np.zeros(N)

i= 0
while(i<N-1):
    th_err = atan2(yend - y[i], xend - x[i]) - th[i]
    d1 = abs(x[i] - xend)
    d2 = abs(y[i] - yend)
    w = v
    d = sqrt(d1*d1+d2*d2)
    if (d<0.5):
        break
    if (d > 100):  break
    w1 = w + k1*th_err
    w2 = w - k1*th_err
    if (d<5):
        w1, w2 = k2*d*(w + k1*th_err), k2*d*(w - k1*th_err)
    dx = (r*dt/2.0)*(w1+w2)*cos(th[i])
    dy = (r*dt/2.0)*(w1+w2)*sin(th[i])
    dth = (r*dt/(2.0*l))*(w1-w2)
    x[i+1] = x[i] + dx
    y[i+1] = y[i] + dy
    th[i+1] = th[i] + dth
    i = i+1
```

A simple modification can take a sequence of points and navigate the robot along the path of points. Place the goal points into an array. Set the counter to the first array index. When the robot is within a small distance of the goal point, increment the counter. The controller will adjust. Fig. 10.7 demonstrates this algorithm.

There are two main issues reported with proportional control. The first is oscillation which can be produced by setting the gain to large. The second is persistent offset error. This is a constant difference between the desired value (set point) and the measured value. In some systems, turning the gain down to avoid oscillations produces higher offset error. Turing the gain up to remove the offset error introduces or increases oscillations. These systems may not have a "sweet spot" or interval of values for which neither issue is presented. So we need additional machinery to correctly control the system.

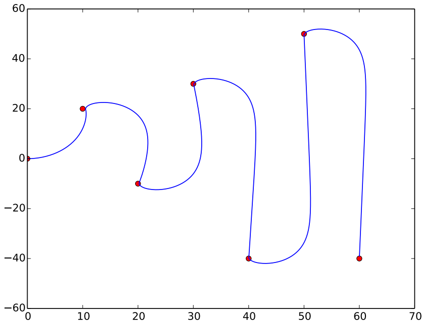


Fig. 10.7: Starting direction $\theta = 0$.

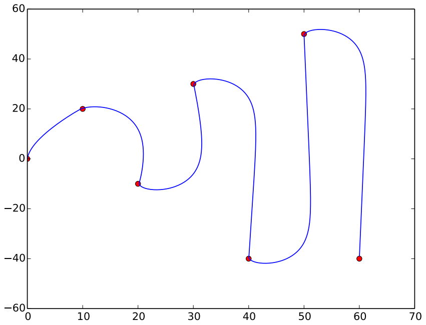


Fig. 10.8: Starting direction $\theta = \pi/2$.

10.1.5 PID Control Overview

To address the oscillations, overshoot and instability, we use a more robust control term, $u(t)$, that includes the error, the change in the error and the error history:

$$u(t) = k_P e(t) + k_D \frac{de(t)}{dt} + k_I \int_0^t e(\tau) d\tau$$

- $e(t)$ - Error $= v_{des}(t) - v_{act}(t)$
- k_P - Proportional gain
- k_I - Integral gain
- k_D - Derivative gain

PID - Proportional Term

Within the PID control, the proportional control contributes in the same manner as it does alone.

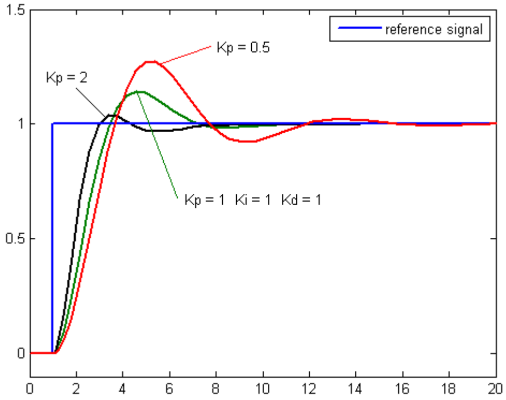


Fig. 10.9: Proportional control.

PID - Integral Term

The contribution from the integral term is proportional to both the magnitude of the error and the duration of the error. The integral in a PID controller is the sum of the instantaneous error over time and gives the accumulated offset that should have been corrected previously. The accumulated error is then multiplied by the integral gain (k_i) and added to the controller output.

The integral term accelerates the movement of the process towards setpoint and eliminates the residual steady-state error that occurs with a pure proportional controller. However, since the integral term responds to accumulated errors from the past, it can cause the present value to overshoot the setpoint value. This is known as integrator windup.

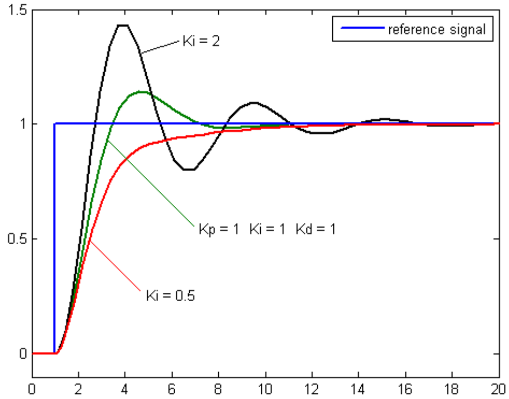


Fig. 10.10: Proportional-Integral control.

PID - Derivative Term

The derivative of the process error is calculated by determining the slope of the error over time and multiplying this rate of change by the derivative gain k_d. The magnitude of the contribution of the derivative term to the overall control action is termed the derivative gain, k_d. The derivative term slows the rate of change of the controller output. Derivative control is used to reduce the magnitude of the overshoot produced by the integral component and improve the combined controller-process stability. However, the derivative term slows the transient response of the controller.

Also, differentiation of a signal amplifies noise and thus this term in the controller is highly sensitive to noise in the error term, and can cause a process to become unstable if the noise and the derivative gain are sufficiently large.

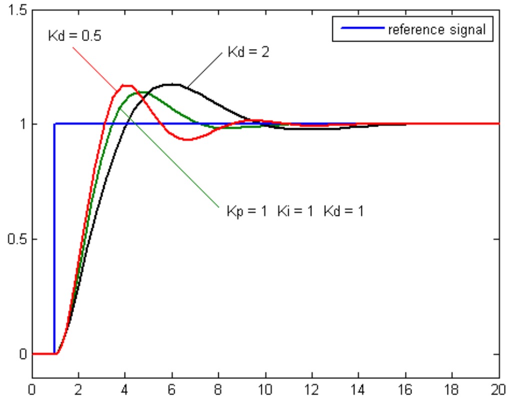


Fig. 10.11: Proportional-Integral-Derivative control.

PI Control Discretization

Set $K_D = 0$

$$u(t) = k_P e(t) + k_I \int_0^t e(\tau) d\tau$$

Use of the controllers in a computer requires discretization. Let t_n be the discrete times, Δt the time step, $e_n = e(t_n)$, and $U_n = u(t_n)$. The discrete form can be converted to a basic recursion:

$$U_n = k_P e_n + k_I \Delta t \sum_{i=1}^{n} \frac{e_i + e_{i-1}}{2}$$

$$\begin{aligned} U_n - U_{n-1} &= k_P(e_n - e_{n-1}) + k_I \Delta t \left(\frac{e_n + e_{n-1}}{2} \right) \\ U_n &= U_{n-1} + K_P(e_n - e_{n-1}) + K_I(e_n + e_{n-1}) \end{aligned} \tag{10.1}$$

where $K_P = k_p$, $K_I = k_I \Delta t/2$.

PD Control Discretization

$$u(t) = k_P e(t) + k_D \frac{de(t)}{dt}$$

The expression can be converted into a recursive relation:

$$U_n = k_P e_n + k_D \frac{e_n - e_{n-1}}{\Delta t}$$

$$\begin{aligned} U_n - U_{n-1} &= k_P(e_n - e_{n-1}) + k_D \frac{e_n - 2e_{n-1} + e_{n-2}}{\Delta t} \\ U_n &= U_{n-1} + K_P(e_n - e_{n-1}) + K_D(e_n - 2e_{n-1} + e_{n-2}) \end{aligned} \tag{10.2}$$

where $K_P = k_p$, $K_D = k_D/\Delta t$.

PID Control Discretization

$$u(t) = k_P e(t) + k_I \int_0^t e(\tau) d\tau + k_D \frac{de(t)}{dt}$$

In a similar fashion as above, the expression can be converted into a recursive relation:

$$U_n = k_P e_n + k_I \Delta t \sum_{i=1}^{n} \frac{e_i + e_{i-1}}{2} + k_D \frac{e_n - e_{n-1}}{\Delta t}$$

$$\begin{aligned} U_n - U_{n-1} &= k_P(e_n - e_{n-1}) + k_I \Delta t \left(\frac{e_n + e_{n-1}}{2} \right) + k_D \frac{e_n - 2e_{n-1} + e_{n-2}}{\Delta t} \\ U_n &= U_{n-1} + K_P(e_n - e_{n-1}) + K_I(e_n + e_{n-1}) + K_D(e_n - 2e_{n-1} + e_{n-2}) \end{aligned} \tag{10.3}$$

where $K_P = k_p$, $K_I = k_I \Delta t/2$, $K_D = k_D/\Delta t$.

10.1.6 PID Application

For this example, we want to control our differential drive robot to follow a path. Assume the path is given by a list of close points (x_n, y_n) for $0 \leq n \leq N$. As before we are interested in finding wheel velocities such that v is constant, v_c, and we are driving from point to point. Recall the differential drive equations:

$$v = \frac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)$$
$$\dot{\theta} = \frac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2).$$

We will use a PID control approach to control the wheel velocities so we can closely track the path defined by the points. Using the same structure we assume wheel velocities of the form

$$\dot{\phi}_1 = \frac{1}{r}(v_c + Lu(t))$$
$$\dot{\phi}_2 = \frac{1}{r}(v_c - Lu(t))$$

where $u(t)$ is obtained from a PID control strategy. In practice on a computer this would use the discrete form of the PID control:

$$\omega_{1,n} = \frac{1}{r}(v_c + LU_n)$$
$$\omega_{2,n} = \frac{1}{r}(v_c - LU_n)$$

Since U_n is a function of the error e_n, we need to figure out what we want for error. The idea is to head to n-th point in the list until we are within some neighborhood of the point, then we increment the counter n meaning target the following point. This requires control on the heading like the example that started our controls journey, but we cannot assume the points are far away, so we need to track position and angles.

Assuming you have your current location as (x, y) and target location (x_n, y_n), the required heading would be $\langle x_n - x, y_n - y\rangle$. If you have a compass on the robot, then you know the current heading. The heading error we used before is

$$e_n = \text{atan2}(x_n - x, y_n - y) - \theta_n.$$

What if you don't have a compass? The current heading can be estimated by $\langle x - x_{n-1}, y - y_{n-1}\rangle$ which is just an estimate of the derivative. And we have

$$e_n = \text{atan2}(x_n - x, y_n - y) - \text{atan2}(x - x_{n-1}, y - y_{n-1}).$$

Up to this point we have computed the heading error without discussion. Specifically the approach has been to take the current heading vector and compute the heading angle via atan2, then subtract from the desired heading (which might have also been computed via atan2). This gives formulas that look like atan2(vy,vx) - atan2(uy,ux).

Example:

- For ux = 0.8, uy = 0.5 and vx = 1.1, vy = -.2
  then atan2(vy,vx) - atan2(uy,ux) ≈ -0.73845 .
- For ux = -0.8, uy = 0.5 and vx = -1.1, vy = -.2

then atan2(vy,vx) - atan2(uy,ux) $\approx$ -5.544732 .

If you graph these, the pair of vectors are reflections about the y-axis and so should give the same result. What is the issue? The sum of those two appears to be -2π. Not surprisingly the problem lies in the ambiguity of which angle is desired. From calculus we know that the angle between two vectors can be determined by the dot product.

$$u \cdot v = \cos(\theta)\|u\|\|v\| \quad \to \quad \theta = \cos^{-1}\left(\frac{u \cdot v}{\|u\|\|v\|}\right).$$

It is worthwhile to write a function that correctly determines the signed angle between vectors. Using the cross product:

```python
def angle(u1, u2, v1, v2):
   n1 = math.sqrt(u1*u1+u2*u2)
   n2 = math.sqrt(v1*v1+v2*v2)
   dot = u1*v1+u2*v2
   cross = u1*v2 - v1*u2
   if cross == 0.0:  return 0.0
   if cross > 0:  sign = 1
   if cross < 0:  sign =-1
   theta = sign*math.acos(dot/(n1*n2))
   return theta
```

Returning to the control problem we have:

$$e_n = \text{angle}(\cos(\theta_n), \sin(\theta_n), x_n - x, y_n - y)$$

$$U_n = U_{n-1} + K_P(e_n - e_{n-1}) + K_I(e_n + e_{n-1}) + K_D(e_n - 2e_{n-1} + e_{n-2})$$

$$\omega_{1,n} = \tfrac{1}{r}(v_c + LU_n)$$

$$\omega_{2,n} = \tfrac{1}{r}(v_c - LU_n)$$

The last aspect is to work out the transition to the next point. Assume you want to get within δ of the point. This means that you want to drive to (x_n, y_n) until

$$d = \sqrt{(x - x_n)^2 + (y - y_n)^2} < \delta$$

then increment n to switch to the new point. If you are using the non-compass formula, you will want save your location when you switch. Using the saved location instead of (x_{n-1}, y_{n-1}) will give you better heading accuracy.

10.1.7 PID Parameter Tuning

Tuning a PID control can be a bit of an art. There are a number of approaches in the literature and we provide two below. The first tuning method has you work the gains like dials starting with the proportional gain, the addressing the derivative gain and finishing with the integral gain.

1. Select a typical operating setting for the desired speed, turn off integral and derivative parts. Increase K_P to maximum or until oscillation occurs.
2. If system oscillates, divide K_P by 2.

3. Increase K_D and observe behaviour when increasing or decreasing the desired speed by 5%. Select a value of K_D which gives a damped response.
4. Slowly increase K_I until oscillation starts. Then divide K_I by 2 or 3.
5. Check overall controller performance under typical conditions.

PID Parameter Tuning: Ziegler-Nichols method

Another heuristic tuning method is formally known as the Ziegler - Nichols method, introduced by John G. Ziegler and Nathaniel B. Nichols in the 1940s. As in the method above, the K_i and K_d gains are first set to zero. The P gain is increased until it reaches the ultimate gain, K_u, at which the output of the loop starts to oscillate. K_u and the oscillation period T_u are used to set the gains as shown in Table 10.1

Table 10.1: Ziegler - Nichols method values

Control Type	K_p	K_i	K_d
P	$0.50K_u$	•	•
PI	$0.45K_u$	$T_u/1.2$	•
PD	$0.8K_u$	•	$T_u/1.2$
PID	$0.60K_u$	$T_u/2$	$T_u/8$

10.1.8 Velocity and Position Control

When working with robot arms and vehicles, it is rarely safe to jump from one speed to another (or one force to another). Safety of the humans and the system requires that changes in the system state be controlled. Clearly a robot arm that jumps from one position to another is dangerous. As would an autonomous vehicle which skidded off from the light. It is also hard on the mechanical systems to have this form of bang-bang control applied to changes in set points. For vehicles, high wheel torques can cause slip and slide which introduces errors in the navigation.

This is addressed by having speed ramp functions to control the transition to new set points. This is no more than what we all do in our cars by slowly pressing down on the accelerator until we reach the desired speed. Fig. 10.12 shows one sample ramp function. There are times when one needs coordinated control between multiple devices. This is necessary with any vehicle that has more than one drive motor, for example a differential drive, Fig. 10.13.

Ramping up each motor to the same speed does not assure straight motion. Variations between ramp ups can cause significant errors in orientation for differential drive. Both motors must be ramped up in the same manner so must be fed the same ramp up function, Fig. 10.14. This can be done with a P, PI or PID control; a PID version is shown in Fig. 10.15.

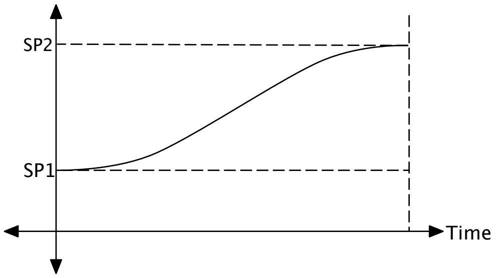


Fig. 10.12: A speed ramp function for a single motor

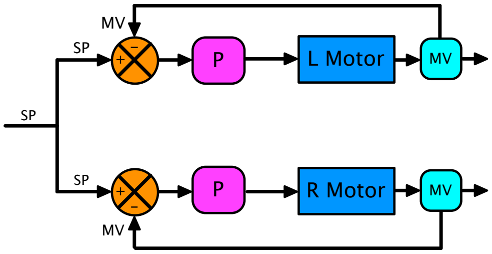


Fig. 10.13: Coordinating two motors separately with a P controller.

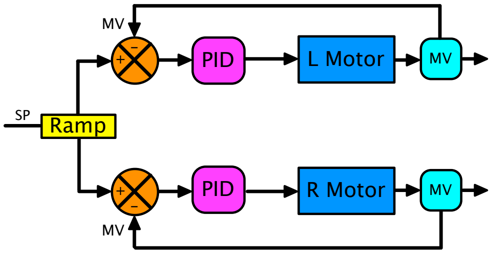


Fig. 10.14: Coordinating two motors with the same ramp function.

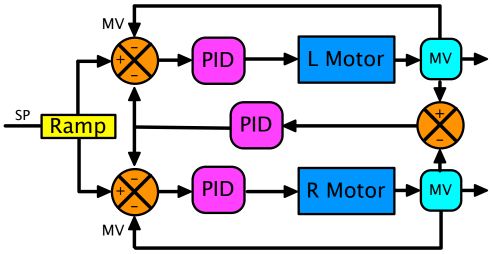


Fig. 10.15: Coordinating two motors with dual P/PI/PID controllers.

10.1.9 Kinematics vs Physics Engine

Kinematics is concerned with the geometry of motion. It describes the geometry based on the constraints of motion without concerns for the causes of the motion such as the forces acting on the system. A physics engine models the forces and the subsequent motion. A kinematics simulation has no knowledge of mass, inertia, friction, momentum and accelerations. It is possible to have jumps in velocity, which would correspond to infinite forces/accelerations. A physics engine will provide a more realistic simulation at the cost of increased computation. This includes the control strategy and so in practice you may not be able achieve the desired control due to limits on forces available.

10.2 Problems

1. Using python, drive the DD robot along the following points at uniform speed with a p-controller: (0,0), (1,1), (2,0), (3,-1). Ramp up at the first point and ramp down to stop at the last point. Plot the points and the robot's path.
2. Create a list of 12 points that trace a figure 8. Using python, drive the DD robot along the list of points. Ramp up at the first point and ramp down to stop at the last point. Plot the points and the robot's path.
3. Write a Python program to navigate a robot using the Wavefront algorithm. Demonstrate on a map with multiple obstacles.
4. Using Veranda, drive the DD robot along the following points at uniform speed with a p-controller: (0,0), (1,1), (2,0), (3,-1). Ramp up at the first point and ramp down to stop at the last point. Plot the points and the robot's path.
5. Create a list of 12 points that trace a figure 8. Using Veranda, drive the DD robot along the list of points. Ramp up at the first point and ramp down to stop at the last point. Plot the points and the robot's path.
6. Using Veranda, navigate a robot via the Wavefront algorithm. Demonstrate on a map with multiple obstacles.
7. In Veranda, drive along $x(t) = 2t - 1$ and $y(t) = 3t + 4$ with unit speed ($r = 1$, $L = 4$) with

   a. Basic bot

   b. DD Robot

   c. Mecanum robot

   Use a video screen capture program to record the results.
8. Assume that your path planner provided 20 points for a path. The x and y arrays are given below.

```
x =
[ -0.01899877   0.50988231   0.39195992   0.49792957   1.19094274
 0.95169032   0.9641402    1.66612687   1.61850455   1.75073195
 1.82176635   2.32907279   2.76304305   2.31714      2.67598521
 3.02594109   3.17585141   3.5780992    3.49887423   4.27148019
 3.8037559    3.75547343   4.67667882   4.37852488   5.03050105
 4.68437205   5.3628283    5.69527227   5.5529013    5.65019384
```

(continues on next page)

(continued from previous page)

```
 6.06986463   6.6702379    6.71739313   6.97436902   7.18467038
 7.26795319   7.07714903   7.67207152   7.90740038   7.77081972
 8.03395401   8.22962525   8.43945312   8.9500287    8.83487244
 9.02244307   9.82322882   9.753483     9.79235576  10.49574545]
y =
[ -6.53957063  -8.40356597  -7.48865588  -6.78476431  -6.78271663
-7.92142123  -6.57019995  -6.68484716  -5.79862184  -5.9067885
-5.23993105  -4.28376187  -2.74857672  -2.92599682  -1.44630301
-0.55009994   0.42216262   1.5820503    1.85426302   4.54071375
 4.93526941   6.84360249   7.56566459   7.93381515   8.88892991
10.11966925  10.8261301   12.03920962  11.37421679  11.94089295
13.0999794   13.06199908  12.75615952  13.33620909  12.57081916
12.56314669  12.50733857  11.7055243   10.1871132   10.41736635
 9.22141768   8.18619806   5.76331083   3.81212651   1.08743568
-1.32495576  -3.92211759  -7.07911689 -10.18134814 -13.76897354]
```

Let $r = 10$, $L = 20$. Use a linear control approach to direct a DD robot to follow the path.

9. Drive the DD robot along the following points at uniform speed using a cubic spline: (0,0), (1,1), (2,0), (3,-1). Plot the points and the robot's path.

10. Assume that you have the four wheel Mecanum system with the 45° roller wheels. Also assume the wheel radius is 5cm, axle distance between wheels (left to right) is 30cm and distance between axles (front to back) is 40cm. What should the wheel velocities be if

    a. you want to drive at an angle of 20 degrees to the right of the forward line for the vehicle at 10 cm/sec (but not change orientation),

    b. you want to head in a straight line at 10cm/sec but rotate the vehicle about the centroid at 1 radian per minute.

11. Assume you have a planner that has provided the following points (2,9), (12,13), (23, 40). Also asume that you start with zero derivative at (2,9) and pass through (12, 13) with slope $m = 2.5$ and have slope $m = 0$ at (23,40). Find the spline.

12. Assume you have a planner that has provided the following points (7,13), (10,20), (25,5). Assume that your previous point was (2,0), you want to pass through (10, 20), with slope $m = -2$, and have slope $m = 3$ at (25,5). Find the spline.

CHAPTER ELEVEN

ROBOTICS DESIGN ELEMENTS

Before we proceed with building robots, we need to address several aspects of robot design which includes human aspects such as safety, human interaction and human environments. Robots can be very helpful, capable even lifesaving devices. However they can pose serious risks which need to be recognized and addressed. In addition, there are complexities in working with humans and in human environments which need to be addressed as well. In this short chapter we examine some of the issues.

11.1 Working with humans

Robert Williams, an American Auto Worker. Mr. Williams worked at a Ford Motor Company factory in Flat Rock, Michigan. He was working on January 25, 1979 with a parts-retrieval system that moved material from one part of the factory to another. When the robot began running slowly, Williams reportedly climbed into the storage rack to retrieve parts manually. He was struck in the head by the arm of a 1-ton production-line robot as he was gathering parts and killed instantly. He would go down in history as the first recorded human death by a robot. William's family successfully sued the manufacturers of the robot, Litton Industries, and was awarded $10 million dollars. The court concluded that there were insufficient safety measures in place to prevent such an accident from happening.

Kenji Urada, a Japanese maintenance engineer. Urada worked at the Akashi Kawasaki Heavy Industries plant. On July 4, 1981, Urada was checking on a malfunctioning robot. He leaped over the protective fence and accidentally hit the on-switch, resulting in the robot pushing him into a grinding machine with its hydraulic arm and crushing him to death. Mr. Urada is the second individual to be listed as a death by a robot.

Wanda Holbrook, an American technician. In July 2015, Wanda Holbrook, a maintenance technician performing routine duties on an assembly line at Ventra Ionia Main, an auto-parts maker in Ionia, Michigan, was "trapped by robotic machinery" and crushed to death.

When OSHA was founded in 1971, there was an estimated 14,000 job related fatalities every year. This is roughly 38 deaths per day. In 2017, the number has dropped to around 12 per day. The vast majority of these deaths are impact related. The deaths are from falling or impact (struck by vehicles or other machinery). Next in line are workplace violence, electrocutions and drowning. Careful design, planning and implementation of the workspace can prevent injuries and fatalities as well as save considerable finances.

Large robots have become common in industrial environments and are now starting to penetrate other markets. Since 1971, OSHA has documented over 300,000 work related US fatalities. The fatality number for industrial robotics is much lower at roughly 30 deaths for a 30 year period, see Table 20.1. [However, this is not a fair comparison since the overall number of workers around industrial robots is much less that the general work population.] A good argument can be made that a number of robotic systems place the robot in the higher risk situation and they have most likely saved more lives than were lost. The point here, though, is that shipments of robots are currently exponentially increasing and as these machines move out of heavy industrial settings, the potential for human injury is exponentially increasing.

Industrial robots deployed in the automotive sector are powerful machines. They can strike a human with great force causing fatal injuries. Even smaller or lower powered systems can cause significant injury. The power-up process can produce unpredictable positioning and movement. This has prompted a series of guidelines for the setup and use of industrial robots. These systems are placed in cages or in blocked off areas. Fenced regions are setup so that opening the fence door shuts down power. Isolation gates are common practice to keep people safe. A standard power-down procedure is required for physical access to the robot and robot workspace.

An examination of OSHA records and supported by studies in Sweden and Japan show that accidents don't occur during the normal operation of the robot. During normal operation, training and safety barriers protect the people working around the machines. Accidents occur during programming, program touch-up or refinement, maintenance, repair, testing, setup, or adjustment. Problems that arise occur when something unexpected has happened. No training procedures for the current problem may exist and the work staff is forced outside their training and expertise. The situation may be confusing with multiple distractions. Human error occurs often which has resulted in terrible accidents.

The reports show multiple instances of individuals circumventing safety systems in place. By jumping fences or crossing safety barriers, individuals placed themselves at grave risk. Accidents occur during maintenance when systems are activated while individuals are inside the robot workspace. Poor decisions to save time, by troubleshooting the system without proper shutdown caused numerous fatalities and injuries. Businesses that place extreme pressure to keep on schedule or not stop the line, setup the culture of skipping normative practices leading to unsafe decisions.

11.1.1 Robotics Industries Association

Founded in 1974, Robotics Industries Association, RIA, is the only trade group in North America organized specifically to serve the robotics industry. Member companies include leading robot manufacturers, users, system integrators, component suppliers, research groups, and consulting firms. https://www.robotics.org/

Safety standards by the RIA are used as the industry standard. The national standard ISO 10218 is based on the RIA standard R15.06. This standard covers hazard identification, risk assessment, actuation control, speed control, stopping control, operation modes, axis limiting and all other aspects of robot design and operation.

Once the decision is made to bring in robots, a full hazard identification and analysis is required. The standard identifies the following aspects:

1. the intended operations at the robot, including teaching, maintenance, setting and cleaning;
2. unexpected start-up;
3. access by personnel from all directions;
4. reasonably foreseeable misuse of the robot;

5. the effect of failure in the control system; and
6. where necessary, the hazards associated with the specific robot application.

Industrial robots need to be physically separated from people and the standard defines the needed infrastructure. These include covering gears, links, toolheads and electrical systems with panels and when not possible placing in physical barriers between human work areas and the robotics hardware. Robots need to have emergency stops and accessible power-off panels. The physical barriers should automatically stop the robot when anyone enters the workspace. The system needs to have speed controls and workspace limit controls. Good software and good interfaces are needed.

There is no substitute for good training and good policy. Many of the accidents could have been avoided if workers followed the access rules. Some accidents arose due to insufficient barriers, markers or space. All of these threats can be addressed by a careful hazard analysis.

11.2 Collaboration with Humans

The National Science Foundation (NSF) announced the second NRI, National Robotics Initiative, Ubiquitous Collaborative Robots (NRI-2.0). The first line of the NSF proposal call reads *The goal of the National Robotics Initiative (NRI) is to support fundamental research that will accelerate the development and use of robots in the United States that work beside or cooperatively with people.* This is not a new research line, but is a very public acceleration of a trend. The goal of the research and expectation for the future is that humans and robots will be working together in a collaborative manner. It should be noted that collaboration is more than remote control. The intent is to have autonomous robots working in collaboration with people, but not fully controlled by those individuals.

To have this collaboration, we need to remove the cages. The system needs to be safe and people need to trust the system. Trust is earned, not just announced. So how can one develop a trusted and safe robotic partner?

11.2.1 Environmental Awareness

Robots need full awareness of their surroundings. They especially need to know if and where the humans are. This requires a number of sensors. These sensors need to be redundant and differentiated. By this we mean that we need multiple sensors which can confirm sensor readings. A particular vantage point might be blocked or subject to interference. For example, reflected light could produce a bogus reading on a light sensor pointed in a particular direction. Noise or other disturbances can produce erroneous readings for vibration or pressure sensors. So redundancy is important for correct errors and filling in gaps.

By differentiation we are asking for diversity of sensor types. We might want to know that we sense something at a particular range, we sense noises and detect heat. The three together gives us more information than having just one sensor type in which each one would be fooled by the same erroneous signal. It is much less likely that three different erroneous signals would occur to fool three different types of sensors.

For a human to work safely and comfortably around the machine, they need to know the machine is aware of their location with the idea that the machine can and will adapt to the human presence. In this case, the robot must avoid collisions. Limiting movement areas, limiting movement speed and force reduction are things that can be employed to enhance safety and confidence. Newer systems will have a zone for which the robot must enforce predetermined limits. Clearly implementation of these policies requires constant environmental awareness. More than just limits, feedback from the robot to the individuals is needed. Using lights, sounds and motions are all approaches that can be used to let the people working around the system know whether or not the system is aware of their presence.

11.2.2 Direct Communication

Work can involve lots of concurrent activities. Humans are easily distracted and this can cause someone to not pay attention to a robot or miss the warning light. Systems need to have very direct and clear communications to avoid potential harm. You see in plenty of industrial machines the system of warning lights and sirens indicating machine activity. In collaborative robotics intended to be with humans, the cages and sirens are not an option. The recent expectation is for a much higher level of cognition in robots we work with. So, we can expect that robots speak to us letting us know what they sense, what they plan, and what they are attempting. Robots with eyes or faces can turn towards the users which helps in gaining human attention and communication.

11.2.3 Indirect Communication

You commonly hear people say things like “non-verbals makes up 93% of communication”[1] (which is now seen as more urban legend or myth). However, non-verbal communication is an important manner in which humans communicate. Giving robots the ability to both understand and present non-verbal elements helps in the overall goal of safety and confidence. Non-verbals such as gestures and expressions can be added in the overall system design at with current technology. Understanding human gestures and expressions is under development seeing advances based on the newer deep learning (neural networks) systems.

It is clear that the non-verbal will go a long way in making people more comfortable around the robots. It is one of the ways we can anthropomorphize the robot leading to better human adoption. Although taking this too far can lead to other problems as we will discuss below.

11.2.4 The real world

One of the reasons we have robots is that they can repeat an action over and over again, exactly the same each time. This is what industrial machines do. Generally people don’t like repetitive jobs, are not good at it and can cause injury. We live in a world of imprecision and randomness. Repeatability and uniformity are abstract concepts not often found in nature. Once the robot leaves the confines of the production line, the environment gets much more complicated. Robots must be tolerant of variation in all dimensions. It must be able to handle this from the outside as well as itself. It must be able to respond to failing sensors, software and actuators in addition to unexpected external events.

Fault tolerance will be an increasingly important aspect to human-robot collaboration. Systems which constantly assess the state of the robot, the progress of the task and the environment are needed to be successful in the dynamic and varied human environment. At first, humans will accept changing their behavior to work with the robot, but for the robot to be accepted it needs to meet the person halfway and not require that behavioral changes are required for the human collaborator; especially those due to safety concerns.

Consider power failure in the robot. The first industrial robot the author saw was one who had a bad reputation. It was in a research lab tended by graduate students. When powered on the arm would jump

[1] In 1971, Albert Mehrabian published a book, Silent Messages, asserting these numbers.

from its resting position to some ready or home position. This action was very fast. When power was removed, the forces used to keep the arm in place were released and the arm would release in another unexpected manner. One of those power cycles caused the arm to strike a student. The student was injured but otherwise was ok.

Power changes can be very scary. If the power hits all systems at once, the servos will receive power at the same time as all of the electronics. Computers and microcontrollers take time to power up or boot. So, this leaves the servos at the mercy of the random signals on the communication channels. Mobile robots jump, jerk and drift. Arms can swing and gantries can move. It is essential that a safe power on process is developed. A well designed system does not allow the random bits on the controllers and buses to cause robot motion. Only when the CPU is up and has verified its state, sensor inputs, executed safe state protocols will it move the robot. The system must validate that only those commands will initiate motion.

Power down is another dangerous time. A heavy robot arm can fall or drop a heavy load. Mechanical locks or resistance must be used to prevent people getting hurt by robot components falling to a rest position. The power up / down problem is part of a larger issue of behavioral expectations and consistency. Software is said to be secure if it behaves like you expect it to.[2] The author believes this is a good definition of a secure or trustworthy robot. One that behaves like you expect it to behave. This is how individuals learn to trust each other.

We started this chapter with three accounts of horrible industrial accidents. The fatality was due to collision between the operator and the robot manipulator. Not all robots working in close proximity will be sufficiently powerful to injure a person, but that is not the point. One must design the system with the assumption that human robot collisions will happen. Having touch sensing through the robot helps the system know when contact is made. This must trigger an interruption of the current task. The task can be stopped, the articulator moved to a safe distance and then wait for human direction.

Having power limits in the robot may also help. So if the robot manipulator collides with a human, it cannot do any real damage. There is current research in soft (flexible) robotic systems. One of the goals is to increase the safety by limiting the possible power delivered to any obstacle. Responding to a collision is important even in these low power cases since there is probably an issue, and it is annoying to get struck by the robot.

11.2.5 Close interactions

For a robot to work with people, it needs to act like people. A concept of personal space needs to be enforced. Beyond awareness its surroundings and of individuals near it, the robot needs to respond like humans do in respecting personal space. Path planning needs to route around heads and limbs. Just like we do when working together. When the path planner can not do this it needs to tell the human in a polite way to adjust. Equally useful will be the ability to understand the human through gestures and verbal commands that the robot needs to adjust. There are times that the robot and the human will need to be in physical contact to perform a task. Careful visual and audio feedback is required to be an effective partner in the collaboration. To be fair, this is a skill that many people struggle with.

[2] This is a definition by Garfinkle and Spafford, Practical Unix and Internet Security.

11.2.6 Appearance

With the innate human tendency towards anthropomorphism, we can build on it by providing the robot with humanlike features. Eyes, faces and arms all work at a psychological level to make the machine seem more human. However, there are clear limits to this increasing humanization which can be seen in our psychological response to certain systems. Take Actroid, Fig. 1.2, which is designed to replace a human receptionist. It has been built to look as human as possible. The idea expressed by Japanese roboticist Masahiro Mori in 1970 is that the more human-like a machine appears, the more endearing it will be. This is not the case, however. As the design becomes more and more similar to the human or animal it is attempting to model, we have a negative response. We use terms like "creepy" or "wrong". It makes us uncomfortable. This is known as *uncanny valley*. Our acceptance of, or comfort with, the machine drops as the design approaches lifelike accuracy. All cultures (that the author is aware of) exhibit this, but varies greatly in the exact boundary of their limits.

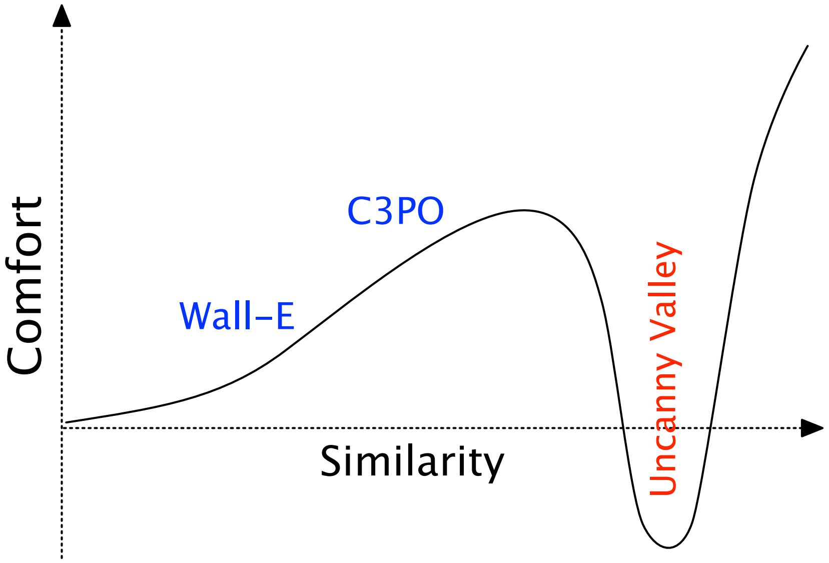


Fig. 11.1: Uncanny Valley, the drop in the comfort graph as a function of human likeness.

11.2.7 A completely different view

Evan Selinger, a Philosophy Professor at RIT, has a completely different take on the utility of anthropomorphic design. He argues that bots, robots and the like should strive to be less or appear less human. That because we have this innate tendency towards anthropomorphism, we make assumptions and mistakes based on those assumptions. Take Siri for example, Siri is based on speech recognition and machine learning technologies. Siri uses a female voice and human speech patterns to present the guise of humanity. Although sophisticated, Siri and Alexa and the like are far from human. Machine Learning is still a mathematical pattern matching tool and not a self-conscience cognitive entity. Placing this technology in a robot, does not then transform the robot into more human than molding it into a human form.

Dr. Selinger argues that the designers should do the opposite. Have the system constantly let everyone know it is a robot; voice its limitations. The system needs drop a gender in the voice or at least vary the one used.

By continually providing feedback that separates the robot or system from anthropomorphism, the system is better able to assist the user since the context is clear. Robotic systems are created to assist us with tasks. Making them increasing human does not necessarily make them better assistants. For example, fidelity to human speech patterns means that, as Dr. Selinger puts it, the "key-board shortcuts" are not available.

To build on this idea, one can argue don't need to create robots that are a partial or substandard human. We have plenty of people on the planet and many are underemployed. We need the robots to focus on the tasks in which we do want to replace human labor. We also know that humans are generalists. We are not the fastest or the strongest or the most robust. We do many many things and in some cases just well enough. Our robots should be tuned and exceptional for the task at hand. They should be specialists and as such not strive to look or act or be like humans.

11.2.8 Human Environments

Human and outdoor spaces are messy. They are random, complicated and dynamic. Operating there is more challenging than in a designed and predictable assembly line. To complete a variety of tasks, robots need to understand their location and orientation in space. They need to sense and understand landmarks, obstacles and free space. In order to do this in the past, the operating environment needed to be augmented or instrumented. For example, lines painted on the floor or conduit in the concrete would be used for directing the robot along paths and hallways. IR sources, RFID tags or other systems are used for landmarks and by using stored maps, landmarks would be used for localization. Orientation could be inferred from the landmarks or if possible a compass.

Systems up to now would instrument the environment to help the robot in the small confines of rooms and hallways found indoors. Outside the system might access GPS which can give a rough estimate but lacks the fidelity needed for indoor navigation. Modifying the environment can be expensive and intrusive. It might not even be possible for some locations. Until robots have a very clear understanding for their surroundings, systems must rely on changing the environment.

To have an effective home robot, the homeowner needs to accept the augmentation costs or not use the robot. Modern deep learning systems may bring changes where it is no longer necessary to instrument the region. Until then, design decisions must include environmental augmentation.

11.3 Cybersecurity Issues

Robotics software is complicated. Current design approaches use multiple cores and cpus. Interprocess communication is done via buses or sockets. Effectively a robot is a collection of networked nodes. As such it is prone to all of the security issues found in any distributed system. It is in effect the IOT (Internet of Things) security problem.

Using Garfinkle and Spafford's definition of security, that the computer should behave as expected, there are a number of security issues.[1] The problems listed above in this chapter are specific examples of security problems.

The most advanced robotics software is based on ROS, the Robot Operating System, which is covered in detail in the next chapter. ROS is not an operating system, but a middleware layer and software collection.

[1] The term security has become associated with preventing system cracking but secure really means that you can trust the system. You may not care about intrusion or data exposure, but you do care that the system operates the way you need it to.

ROS manages socket communication over multiple nodes. ROS currently only runs on Linux. Unix and thus Linux, was not designed with security in mind. It was designed for ease of use, flexibility, extensiblity and an expectation of user sophistication. Much of Unix was developed by students and researchers to facilitate projects and not as production (engineered) code.

So this means that ROS inherits any security issues found in Linux (Ubuntu). The steps, processes and procedures that any security professional working on a distributed network of Linux systems would take are steps that need to be considered in any ROS based system. The principles are similar, Table 11.1 outlines the steps.

Table 11.1: Security Planning

Aspects of Planning	Creating Policies
Risk Assessment	Implementation
Cost-Benefit Analysis	Validation

The first design stage is security planning. Have a complete understanding of the goals or tasks for the robot. This normally means having something like a CONOPS document (concept of operations document). This will allow you to determine the overall planning process. Yes, I just suggested that you plan your planning. The idea is that you don't want to miss any aspect of the system level view. Having system engineering expertise on the team can be quite valuable here.

Based on a concept of use, the next step is to perform a thorough risk assessment. We have discussed a few risks above. One must look at various scenarios such as loss of the robot, loss of data, loss of control, etc. Depending on application, we may be concerned about harming individuals or loss of collected data or lost revenue due to down time. For surveillance robots, loss of data integrity or real time feeds can shut down a mission, but for deep space robots there might not be any expectation of having a continuous communication. You may want the data to be confidential as with military robots or continuously available interaction. There are some core questions you should ask:

- What am I trying to protect?

  [Human life and limb, data, control, expensive hardware, . . .]

- What do I need to protect against?

  [The elements, environmental variation, hostile humans or software, . . .]

- How much money, time and effort am I willing to spend to obtain adequate protection?

  [Cost of loss vs cost of protection.]

Immediately after you can perform the following steps:

1. Identify assets
2. Identify threats
3. Calculate risks

Threats come in many forms. Most of them are not related to malicious humans, Table 11.2. The media will latch onto a DEF CON report about "hacking" into the bluetooth on a tire pressure monitor and then accessing some of the car's control system. This leads the media to report that hackers can break into your car and drive you over a cliff. Although this is a very real concern in the future, currently there are much more pressing issues. The likely causes for robot failures for the near future will be lack of risk analysis and poorly tested software.

Table 11.2: System Failures

Power loss or surges and battery life	Equipment failure, EMF noise, static
Sensor failure or obstruction	Upgrades to underlying software
Loss of network service	Viruses and poorly tested software
Loss of human input	Third party (crackers)
Water, dust and chemical damage	Misconfigured software

You may be able to remove a risk by changing the way a feature is implemented but in some cases it requires removing a feature (or ability in the case of robots). Maybe it is the decision on physical barriers instead of the sensors and controls required to work safely around the robot. Keep in mind that engineering is not about providing the coolest and newest technology. Good engineering like good design is about solving problems. Sometimes the best solution is not the highest tech solution.

Cost-Benefit analysis takes the risks and converts them to cost by estimating the cost of the threat if it occurs. In the cases you can estimate the cost and compare this to the cost of building a solution which avoids that particular threat, Table 11.3. Clearly some of the numbers are very rough. Estimating the time and parts to build some system comes from experience. If multiple systems are built then the design and testing costs can be prorated over all of the production units leaving just the parts and assembly costs for the units. If development costs are x and per unit cost is y then the cost of the threat mitigation is $x + Ny$ where the number of units is N. The threat risk is multiplied over the number of deployed units, N. If the threat probability per unit for the lifetime of the device is p and the cost of that threat is z, then you are comparing Npz to $x + Ny$. This is the most simplistic way to view the analysis and more detailed studies should be done.

Table 11.3: Cost Benefit Analysis

Cost of loss	*Cost of prevention*
Short/Long term lack of availability	Additional design and testing
Permanent loss (accidental or deliberate)	Equipment (hardware and software)
Unauthorized disclosure (to some or all)	User training
Replacement or recovery cost	Performance

Although companies will assign a cost to loss of life and limb (based on litigation and settlement amounts), we will assume this cost is higher than the cost to prevent or avoid the risk. In this case you have hard limits on the requirements that need to be enforced. Once the cost-benefit analysis is complete, you will have an updated set of requirements. In addition you can set guidelines for how the software system will be designed and managed. Some of this will be implemented in a set of security policies. Often these are very simple tasks like making sure software is configured correctly. The last stages of the planning process involve a careful design with clear test cases at each stage to validate the design.

11.3.1 Network Security

ROS based robots are a collection of networked nodes. Many systems have wifi or bluetooth access. This opens the door for unauthorized access. We strongly suggest getting a network security expert to advise the team on design before the system goes to production. This is not a security text, but the issues you are addressing are common security problems. There are two types of access one can have: passive and active. Passive access is worried about intercepting data. Active access is about modifying machine behavior and is a direct host attack.

Passive:

- Network wiretapping
- Port scans and Idle scans

Active:

- Denial-of-service attack
- Spoofing
- Man in the middle
- ARP spoofing
- Smurf attack
- Buffer overflow
- Heap overflow
- SQL injection

Careful design, attention to details and good testing can go a long way to prevent security issues. In many cases it is just a matter of just getting it on the "to do" list and not difficult or expensive.

11.3.2 Adversarial Machine Learning and other attacks

New robots will need to address a whole new generation of attacks. These attacks will be presented against the sensors and software in novel ways. Several possible attacks are outlined below to illustrate the vast array of issues the roboticist must address.

Insecure Embedded Devices

In 2008, the National Highway Transportation Safety Administration mandated direct tire pressure monitoring. Indirect systems measure the rotation speed of the wheel. Direct monitors have a pressure sensor built into the wheel and transmit a tire pressure to the vehicle electronics. In 2010, it was demonstrated that it was possible to hack into the tire pressure monitor system for automobile tires. The study showed that from this entry point, vehicle systems could be disrupted or even controlled. Examples of shutting down brakes selectively, stopping the engine and other hacks were described.

Like many IOT or other embedded devices, security is not implemented. Classically for embedded devices it made sense. Embedded systems are were isolated from other systems. But with the advent of bluetooth, wifi and other wireless communications appearing on embedded hardware, they become open to intrusion and manipulation. To address this, all wireless communications should be encrypted. Even simple systems like an outside temperature monitor. The point there is that the sensor engineer cannot predict how the data will be used in an autonomous system. The hackers may find that the right combination of false sensor readings causes the vehicle software to make a catastrophic decision. The encryption will also help in terms of a direct attack to load malicious code into any vehicle system in a manner similar to the cyberattacks discussed above.

Computer Vision Vulnerabilities

Computer vision is an active area of research which has shown great progress in the last decade. Since 2012, we are seeing the transition of computer vision systems from feature based approaches to deep learning approaches. Deep learning (or machine learning) algorithms are not well understood. In 2016, CMU showed they could defeat state of the art face recognition algorithms. It is clear that the neural network based vision system could be confused or mislead by correctly constructed patterns. Neural network approaches are trained in a manner that means the resulting decision system is not transparent. Testing is harder, often statistically based, and systems can be shipped with significant issues in vision accuracy or object recognition.

Sensor Compromise

In addition to vision, many autonomous vehicles currently use lidar and gps. Lidar, or laser ranging, uses reflected laser light to determine the distance of objects. Interference from other light sources can causes errors in distance estimation. Use of laser pointers or other sources overlapping the same frequency as the lidar could blind the device. GPS spoofing can be done by sending false signals to the GPS satellite receivers. Currently spoofed signals are hard to detect and so false readings for position (and so velocity) are possible.

Motivation

Who are the actors? Consider the fear and anger with the vision of the future that eliminates so many jobs. Autonomous delivery vehicles, autonomous long distance trucking and transport all have very real economic consequences for a number of people.[2] Angry over job loss has in the past led some to strike out at employers. Fear of a new technology can lead to preemptive strikes. Bored kids or anti-technology zealots as well as all forms of terrorists can find ways to exploit the autonomous systems. The angry unemployed Teamster can cause financial harm to a company by wrecking some of the fleet. The Luddite can cause vehicles to go astray to make robotics tech seem dangerous in an attempt to sway public opinion. The terrorist can take over the navigation remotely and drive the truck into the crowd; even coordinating a fleet for a large impact and very deadly attack.

It is important that robotics organizations provide options and retraining for displaced workers. Public education on the Luddite fallacy is important.[3] It is easy for politicians to vilify groups for their own gain and so countering this behavior will require constant effort for the near future. The root cause in many cases is inequity in economics, corruption and unemployment. Addressing these issues will go a long way in solving the security problems as well as many problems facing us.

[2] There are roughly 1.7 million trucking jobs in the U.S.

[3] The fallacy is that new technology eliminates jobs overall. New tech just displaces jobs to new sectors.

11.4 The Impact of Robotics - Human Cost

Although robots in some form have been around for decades, robotics is in many ways still an emerging technology - especially robots with the newer AI systems included. Even with the optimistic predictions made about the positive economic impacts, robotics has the potential to displace workers and cause unemployment. We will briefly touch on a few thoughts given by economists who study new technology. This is a politically charged conversation and there are plenty of opinions. And as a disclaimer, since this is a robotics text, probably sees more benefits than risks with robotics.

We start with the concern of many Americans. "I am worried that I will be replaced by a robot and then be un-employed." This is a valid concern. There are plenty of examples of industrial jobs being replaced by automation. It is often met with unsupported optimism that this new age of robotics will provide a multitude of benefits for which we cannot foresee. All of this is happening against a backdrop discussion of the dangers found in modern AI. Given that this tech can render a percentage of the workforce unemployed, or that there is profound concern of the dangers of super intelligent AIs built into mobile robits - why would we allow robots? It is a complicated question for which there is not a one size fits all answer.

We start by describing two basic forms of technology entry: Enabling Technologies and Replacing Technologies. Enabling technology is one that assists a worker to be faster, more accurate, lower cost work. When view at the system level, it is possible this tech will replace a person, but overall allows for lower cost production. Replacing technologies do just that. The worker is replaced by the new technology.

In real applications, elements of both will appear. The idea behind enabling technologies is that the production line becomes more efficient, profits increase and value increases. In the case where people lose jobs, the idea is that the economic impact is sufficiently strong to bring everyone up. Meaning the increased revenues then generate other opportunities and those that had lost there jobs will find good jobs elsewhere due to the economic impact. This is to be contrasted to replacing technologies which remove individuals and do not have an impact strong enough to counter act the economic effects of job loss. Those individuals may find lower wage work or may be unemployed which leads to dimished economic growth.

History has examples of both types of market entry. Replacing technologies are seen to lower overall productivity and leave economic picture worse off. This is the current trend. It does have historical precedent and we have witnessed long periods of decline before. This is when new tech enters, jobs are lost but new opportunity has yet to be found.

It is hard to say where robotics lies currently - athough you will hear plenty of experts take a stand. Either way, we can make some choices as we move forward. We can make choices on what directions to increase automation so that we create more opportunity to offset job loss.

An example of this is the following. Assume that an employer can reduce costs for a stage of a process by replacing a person with automation. But no other aspect of the process changes. So number of unit produced does not change although there might be a drop in price. Most likley this is an example of a replacing technology. Stopping here would lead to an overall community productivity loss. The opportunity would be to look at bottlenecks in other parts of the production line. If automation can be added, then now it may be possible to increase the production rate. A drop in price and an ability to source more product hopefully means the ability to enter an additional market and look at scaling up.

The opportunity now is that if the factory scales (new or larger markets), it can add new jobs based on the scaling. Additional marketing, sales, supply chain people can be hired. So even though the new automation has replaced a job just like the replacing technology did, it acts as an enabling technology for the company. Of course this is just a thought experiment and for actual situations, the truth may be a combination of the different enabling tech, replacing tech and business specific details. The point is that it is important on a larger scale to look for enabling technologies which can open new markets and effectively continue to grow overall productivity.

So, can we answer whether or not robotics will be good or bad for society. Yes, we can answer. We don't mean we can provide a yes or no now. We mean that we as a society can guide the process to achieve either answer. So, our conclusion is that the impact of robotics is up to us. Whether or not this is true for AI is altogether another unknown.

11.5 Problems

1. What are the pros and cons to sharing address space?
2. How does a system call differ from a function call?
3. How can a message passing system be used to coordinate two processes? Show a pseudo-code example.
4. What are some pros and cons with centralized server architectures for coordinating multiprocess communication?
5. Many industrial systems have fences or cages that contain the robot which prevents human-robot collisions. The cage system is designed to shut power if the cage door is open. Some of these systems have a key and lock that prevents restarting. This has been defeated by a worker closing the cage door after entry, not taking the key and another worker by accident powering up the robot. Describe some additions to this system which could prevent this breach of safe operation.
6. Name two very simple things that can be done to make industrial robots safer.
7. List some periods of robot operation that are very risky for humans. What can be done to mitigate the risk?
8. What needs to be addressed for humans to comfortably work with robots?
9. What is the cause of uncanny valley?
10. Provide an example not in the text of an exploit of a robotics system.

Note: Missing basic design material. Could move some of the soft eng content here as well as core mechanical/electrical design content.

CHAPTER

TWELVE

SYSTEM INTEGRATION

Robotics

12.1 Assembling the system

Starting . . .

Note: Write this chapter

Note: This section is about System Engineering as it applies to robotics and autonomous systems. There might be overlap with the design chapter. Will need to write and then sort out.

CHAPTER

THIRTEEN

GENERAL KINEMATIC MODELING

Differential drive is a popular approach for lower cost and smaller robots. When the weight or terrain demand four drive wheels, other drive systems are better options. In this section, we approach the general modeling kinematic problem more formally to address different wheel and drive systems.

13.1 Fixed Wheel

We begin with the axle and wheel. Let P be the chassis center, L is the distance from P to the wheel contact point. Define α as the angle between the wheel axle and X_R and β as the angle between the wheel axis A and the axle. We also define A as the vector in the wheel axle direction and v as the orthogonal vector (the vector in the travel direction of the wheel). See Fig. 13.1.

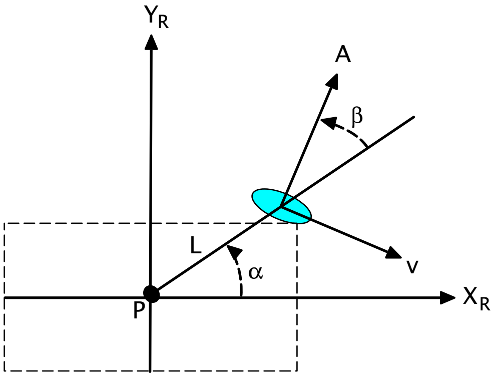


Fig. 13.1: Basic wheel and axle configuration.

$$A = \langle \cos(\alpha+\beta), \sin(\alpha+\beta)\rangle \tag{13.1}$$

$$v = \langle \sin(\alpha+\beta), -\cos(\alpha+\beta)\rangle \tag{13.2}$$

We will examine the linear and rotational aspects of motion separately. They can be combined in the following manner:

$$\langle \dot{x}_I, \dot{y}_I, 0\rangle + \left\langle 0, 0, \dot{\theta}\right\rangle = \left\langle \dot{x}_I, \dot{y}_I, \dot{\theta}\right\rangle = \dot{\xi}_I$$

These equations are in the inertial or global coordinate system. Our derivations will start in the robot or local coordinate system and so the velocity vectors will need to be rotated into the robot coordinates: $\dot{\xi}_R = R(\theta)^{-1}\dot{\xi}_I$.

Recall earlier we introduced the *no slip* and *no slide* constraints. Here we will apply the two constraints to derive the fundamental kinematic equations for each wheel. Those equations will be combined to build the equations of motion for the particular drive system and hence the vehicle.

We begin with the no slip constraint. All of the motion of the vehicle must be accounted for by wheel motion. This means that the total motion of the craft must be equal to the wheel travel and in the wheel direction. This seems obvious and can be translated into a mathematical constraint using the no slip condition. First, we resolve the linear motion of the craft and then address the rotational aspect. Recall that magnitude of the linear motion from the wheel (in the direction of the wheel) is $r\dot{\phi}$. The direction of motion is given by v. Since we have both linear and rotational motion built into $\dot{\xi}$, and to keep the dimensions matching up, the vector v is extended to $\langle v_1, v_2, 0\rangle$. If v is a unit vector, $\|v\| = 1$, then the projection onto v is

$$P_v^1(u) = \frac{v\cdot u}{\|v\|^2} = v\cdot u = \langle \sin(\alpha+\beta), -\cos(\alpha+\beta), 0\rangle \cdot u.$$

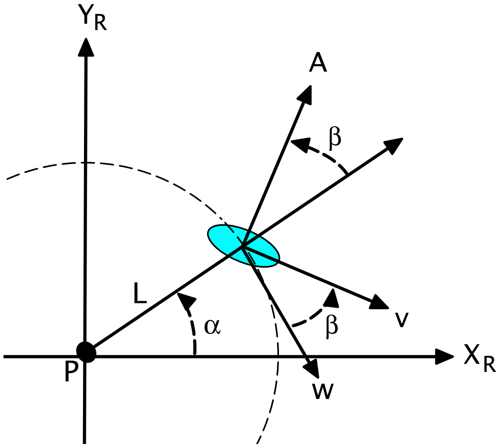


Fig. 13.2: Motion in the angular direction is shown by the vector w.

For angular motion, we can break the motion of the wheel vector (v) into radial and angular components, Fig. 13.2. The radial component is in the direction of the L vector. The angular component is w. It is the angular and not the radial component which will contribute to $\dot{\theta}$. The angular component must have $-L\dot{\theta}$ for the angular speed in the w direction. [The negative comes from the direction of w.] Projecting that speed onto v gives $-L\cos(\beta)\dot{\theta}$ which means our projection component is $P_v^2 = \langle 0, 0, -L\cos(\beta)\rangle$. Combining the projections $P_v = P_v^1 + P_v^2$:

$$P_v = \langle \sin(\alpha+\beta), -\cos(\alpha+\beta), -L\cos(\beta)\rangle$$

and recall

$$P_v[\dot{\xi}_R] = P_v[R(\theta)^{-1}\dot{\xi}_I].$$

So we obtain:
$P_v[R(\theta)^{-1}\dot{\xi}_I]$

$$= \langle \sin(\alpha+\beta), -\cos(\alpha+\beta), -L\cos(\beta)\rangle \cdot R(\theta)^{-1}\left\langle \dot{x}_I, \dot{y}_I, \dot{\theta}\right\rangle. \tag{13.3}$$

For *No Slip* we have:

$$P_v[R(\theta)^{-1}\dot{\xi}_I] = r\dot{\phi}$$

$$\Rightarrow \langle \sin(\alpha+\beta), -\cos(\alpha+\beta), -L\cos(\beta)\rangle\, R(\theta)^{-1}\dot{\xi}_I = r\dot{\phi}$$

For *No Slide*, we want the projection in the direction of A and L to be zero (a similar derivation as above):

$$P_A[R(\theta)^{-1}\dot{\xi}_I] = 0$$

$$\Rightarrow \langle \cos(\alpha+\beta), \sin(\alpha+\beta), L\sin(\beta)\rangle \cdot R(\theta)^{-1}\dot{\xi}_I = 0$$

13.1.1 Steered Wheel

The only difference for steered wheels compared to fixed wheels is that the angle β varies over time. This does not have an effect on the form of the equations at an instanteous time, but will when integrated over time.

For *No Slip* we have:
$P_v[R(\theta)^{-1}\dot{\xi}_I]$

$$= \langle \sin(\alpha+\beta(t)), -\cos(\alpha+\beta(t)), -L\cos(\beta(t))\rangle\, R(\theta)^{-1}\dot{\xi}_I = r\dot{\phi}$$

For *No Slide*, as before we want the projection to be zero:
$P_A R(\theta)^{-1}\dot{\xi}_I$

$$= \langle \cos(\alpha+\beta(t)), \sin(\alpha+\beta(t)), L\sin(\beta(t))\rangle \cdot R(\theta)^{-1}\dot{\xi}_I = 0$$

13.1.2 Castor Wheel

For the castor wheel, the no slip condition is the same (as the castor offset, d, plays no role in the motion in the direction of the wheel). The offset, d, does change the equations in the no slide aspect.

For *No Slip*:

$$\langle \sin(\alpha + \beta(t)), -\cos(\alpha + \beta(t)), -L\cos(\beta(t)) \rangle \, R(\theta)^{-1}\dot{\xi}_I = r\dot{\phi}$$

For *No Slide*:

$$\langle \cos(\alpha + \beta(t)), \sin(\alpha + \beta(t)), d + L\sin(\beta(t)) \rangle \cdot R(\theta)^{-1}\dot{\xi}_I + d\dot{\beta} = 0$$

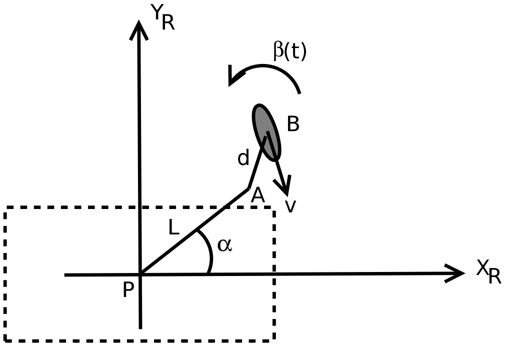


Fig. 13.3: Castor Wheel

13.1.3 Omni, Swedish, or Mecanum Wheels

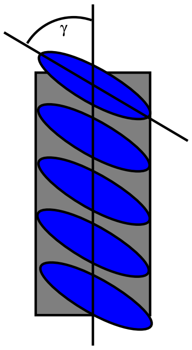


Fig. 13.4: Swedish Wheel

Let γ be the angle between the roller axis and wheel plane (plane orthogonal to the wheel axis) For *No Slip*:

$$\langle \sin(\alpha + \beta + \gamma), -\cos(\alpha + \beta + \gamma), -L\cos(\beta + \gamma) \rangle \, R(\theta)^{-1}\dot{\xi}_I$$

$$= r\dot{\phi}\cos(\gamma)$$

For *No Slide*:

$$\langle \cos(\alpha+\beta+\gamma), \sin(\alpha+\beta+\gamma), L\sin(\beta+\gamma)\rangle \cdot R(\theta)^{-1}\dot{\xi}_I$$

$$= r\dot{\phi}\sin(\gamma) + r_{sw}\dot{\phi}_{sw}$$

Note that since ϕ_{sw} is free (to spin), the no slide condition is not a constraint in the same manner as the fixed or steered wheels.

13.2 Multiple Wheel Model and Matrix Formulation

Since nearly all the robots we will work with have three or more wheels. The equations we derived above can be combined to build a complete kinematic model. We begin with some basic variables that define the system.

- Let N denote the total number of wheels
- Let N_f denote the number of fixed wheels
- Let N_s denote the number of steerable wheels
- Let $\phi_f(t)$ and β_f be the fixed wheel angular velocity and wheel position.
- Let $\phi_s(t)$ and $\beta_s(t)$ be the steerable wheel angular velocity and wheel position.

We bundle the latter two values in a vector for notational ease:

$$\phi(t) = (\phi_{f,1}(t), \phi_{f,2}(t), \phi_{f,3}(t), ..., \phi_{s,1}(t), \phi_{s,2}(t), ...)$$

$$\beta(t) = (\beta_{f,1}(t), \beta_{f,2}(t), \beta_{f,3}(t), ..., \beta_{s,1}(t), \beta_{s,2}(t), ...)$$

Next we collect the no slip constraints, the equations derived above for the various drive types and place them in a matrix:

$$J_1 R(\theta)^{-1}\dot{\xi}_I = \begin{bmatrix} J_{1f} \\ J_{1s} \end{bmatrix} R(\theta)^{-1}\dot{\xi}_I = J_2\dot{\phi}$$

where J_1 is the matrix with rows made up of the rolling constraints and J_2 is a diagonal matrix made from wheel diameters. In a similar manner we can bundle up the no slide constraints (fixed and steered):

$$C_1 R(\theta)^{-1}\dot{\xi}_I = \begin{bmatrix} C_{1f} \\ C_{1s} \end{bmatrix} R(\theta)^{-1}\dot{\xi}_I = 0.$$

This is matrix shorthand to address the kinematic models for a variety of systems.

$$\begin{bmatrix} J_1 \\ C_1 \end{bmatrix} R(\theta)^{-1}\dot{\xi}_I = \begin{bmatrix} J_2 \\ 0 \end{bmatrix} \dot{\phi}$$

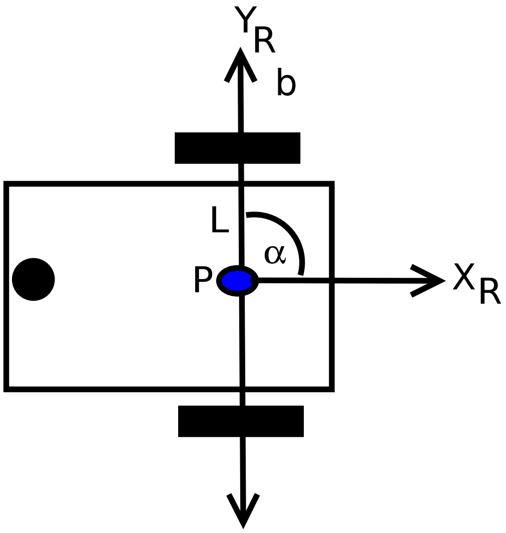


Fig. 13.5: The differential drive robot dimensions and variables.

13.3 Differential Drive - Rederivation

To get a feel for the more general approach above, it is worthwhile to rederive the equations for the differential drive robot. This allows us to check our result as well as see how the matrices are constructed. From Fig. 13.5 we have for the left wheel: $\alpha = \pi/2$, $\beta = 0$; and for the right wheel: $\alpha = -\pi/2$, $\beta = \pi$ (to be consistent with the coordinate system).

Recall the left wheel rolling constraint is given by

$$\langle \sin(\alpha+\beta), -\cos(\alpha+\beta), -L\cos(\beta)\rangle = \langle 1, 0, -L\rangle$$

and the right wheel rolling constraint is

$$\langle \sin(\alpha+\beta), -\cos(\alpha+\beta), -L\cos(\beta)\rangle = \langle 1, 0, L\rangle .$$

From these two equations we can form the rolling constraint matrix:

$$J_1 = \begin{bmatrix} 1 & 0 & -L \\ 1 & 0 & L \end{bmatrix}$$

In a similar manner, recall that the left wheel sliding constraint is

$$\langle \cos(\alpha+\beta), \sin(\alpha+\beta), L\sin(\beta)\rangle = \langle 0, 1, 0\rangle ,$$

and the right wheel sliding constraint is

$$\langle \cos(\alpha+\beta), \sin(\alpha+\beta), L\sin(\beta)\rangle = \langle 0, 1, 0\rangle .$$

Again in a similar manner we can form the sliding constraint matrix:

$$C_1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} .$$

Since the two rows are linearly dependent, we only need to keep one row, so the matrix looks like

$$C_1 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

The two matrices are stacked to form a single matrix model:

$$\begin{bmatrix} J_1 \\ \\ C_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -L \\ 1 & 0 & L \\ 0 & 1 & 0 \end{bmatrix}.$$

The same can be done with the right hand side arrays

$$\begin{bmatrix} J_2 \\ 0 \end{bmatrix} = \begin{bmatrix} r & 0 \\ 0 & r \\ 0 & 0 \end{bmatrix}.$$

The resulting motion model is

$$\begin{bmatrix} 1 & 0 & -L \\ 1 & 0 & L \\ 0 & 1 & 0 \end{bmatrix} R(\theta)^{-1}\dot{\xi}_I = \begin{bmatrix} r & 0 \\ 0 & r \\ 0 & 0 \end{bmatrix} \dot{\phi}$$

Expanding

$$\begin{bmatrix} 1 & 0 & -L \\ 1 & 0 & L \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \dot{\xi}_I = \begin{bmatrix} r & 0 \\ 0 & r \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{\phi}_2 \\ \dot{\phi}_1 \end{bmatrix}$$

To be consistent with the previous example, we had the left wheel as (2) and the right wheel as (1) - hence the reverse ordering on the ϕ terms.

This is the system to solve. Invert the left hand array first, then invert the rotation matrix.

Working out the details:

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \dot{\xi}_I = \begin{bmatrix} 1 & 0 & -L \\ 1 & 0 & L \\ 0 & 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} r & 0 \\ 0 & r \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{\phi}_2 \\ \dot{\phi}_1 \end{bmatrix}$$

$$\dot{\xi}_I = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & 0 & -L \\ 1 & 0 & L \\ 0 & 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} r & 0 \\ 0 & r \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{\phi}_2 \\ \dot{\phi}_1 \end{bmatrix}$$

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1/2 & 1/2 & 0 \\ 0 & 0 & 1 \\ -1/(2L) & 1/(2L) & 0 \end{bmatrix} \begin{bmatrix} r\dot{\phi}_2 \\ r\dot{\phi}_1 \\ 0 \end{bmatrix}$$

and finally ....

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{r}{2}\dot{\phi}_1 + \frac{r}{2}\dot{\phi}_2 \\ 0 \\ -\frac{r}{2L}\dot{\phi}_2 + \frac{r}{2L}\dot{\phi}_1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{r}{2}\left(\dot{\phi}_1 + \dot{\phi}_2\right)\cos\theta \\ \frac{r}{2}\left(\dot{\phi}_1 + \dot{\phi}_2\right)\sin\theta \\ \frac{r}{2L}\left(\dot{\phi}_1 - \dot{\phi}_2\right) \end{bmatrix}$$

(and you didn't think this was going to work out, did you.) You may apply this machinery to other systems as well.

13.4 Omniwheel Example

For this example we look at a Swedish three wheel robot, Fig. 13.6. We use an unsteered 90° Swedish wheel, so $\beta_i = 0$ and $\gamma_i = 0$ for all i. Going counterclockwise in the figure, we have $\alpha_1 = \pi/3$, $\alpha_2 = \pi$ and $\alpha_3 = -\pi/3$. You will note that the C_1 matrix is of zero rank and so the sliding constraint does not contribute to (nor is needed for) the model. The equations for motion then are

$$\dot{\xi}_I = R(\theta) J_{1f}^{-1} J_2 \dot{\phi}$$

where

$$J_{1f} = \begin{bmatrix} \sqrt{3}/2 & -1/2 & -L \\ 0 & 1 & -L \\ -\sqrt{3}/2 & -1/2 & -L \end{bmatrix}, \quad J_2 = \begin{bmatrix} r & 0 & 0 \\ 0 & r & 0 \\ 0 & 0 & r \end{bmatrix}$$

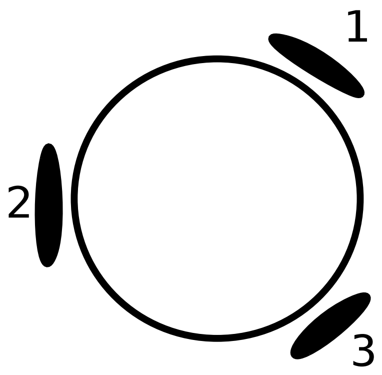


Fig. 13.6: The Omniwheel can be configured in a three wheel system.

13.5 Dual Differential Drive

This section derives the kinematics for a robot with a single axle. This will be used to extend the differential drive to the dual differential drive. All results are with respect to the local robot coordinate system, with y the forward direction, z up, and x defined according to the right hand rule. The total length of the axle is given by $2L$, the robot angle by θ, and the angle of the axle with respect to the robot given by α, with $\alpha = 0$ aligning the axle with the x axis Fig. 13.7. The points at the end of the axle are denoted by A and B, with A corresponding to the point in the positive x direction when $\alpha = 0$.

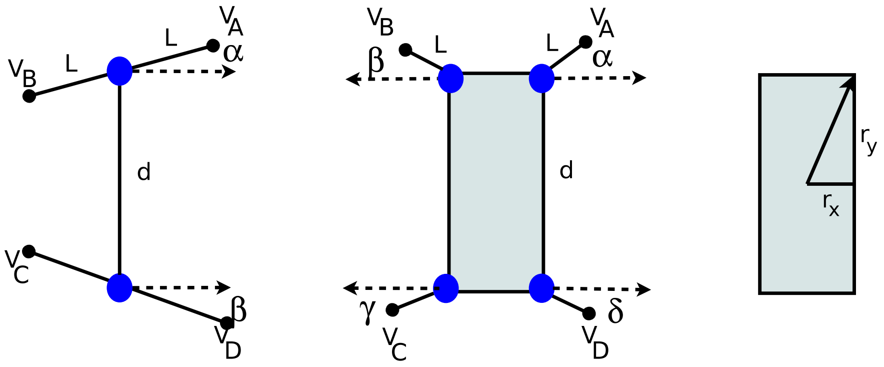


Fig. 13.7: The axis angles.

Simple planar kinematics gives the following relationships between the velocities at points A and B and the robot motion. Let x, y denote the center of the axle.

$$v_{A_x} = \dot{x} - L\dot{\alpha}\sin\alpha, \quad v_{A_y} = \dot{y} + L\dot{\alpha}\cos\alpha$$

Incorporating the non-holonomic constraint on wheel velocity directions yields

$$V_A \sin\alpha = L\dot{\alpha}\sin\alpha - \dot{x}, \quad V_A\cos\alpha = \dot{y} + L\dot{\alpha}\cos\alpha$$

where V_A is the magnitude of the axle tip velocity. Similarly, for point B

$$V_B \sin\alpha = -L\dot{\alpha}\sin\alpha - \dot{x}, \quad V_B\cos\alpha = \dot{y} - L\dot{\alpha}\cos\alpha$$

Combining the equations for points A and B results in

$$\dot{y} = \frac{V_A + V_B}{2}\cos\alpha, \quad \dot{x} = -\frac{V_A + V_B}{2}\sin\alpha, \quad \dot{\alpha} = \frac{V_A - V_B}{2L}$$

The major difference with this current derivation and our previous version in the Terms Chapter is that the coordinate system is rotated by $90°$ compared to what we use.

The analysis now can be easily extended to the case of two axles. Let the pivots for each of the two axles be separated from the robot centroid by distance d in the y direction. Let A and B denote the velocities of wheel for the axle offset in the positive y direction from the centroid and C and D denote the velocities of wheel for the axle offset in the negative y direction from the centroid. The angle of the front axle with respect to the robot is given by α, whereas the angle of the rear axle with respect to the robot is given by β. Then

$$V_A \sin\alpha = L\dot{\alpha}\sin\alpha - \dot{x} + d\dot{\theta}, \quad V_A\cos\alpha = \dot{y} + L\dot{\alpha}\cos\alpha$$

$$V_B \sin\alpha = -L\dot{\alpha}\sin\alpha - \dot{x} + d\dot{\theta}, \quad V_B\cos\alpha = \dot{y} - L\dot{\alpha}\cos\alpha$$

for the front axle and

$$V_C \sin\beta = L\dot{\beta}\sin\beta - \dot{x} - d\dot{\theta}, \quad V_C\cos\beta = \dot{y} + L\dot{\beta}\cos\beta$$

$$V_D \sin\beta = -L\dot{\beta}\sin\beta - \dot{x} - d\dot{\theta}, \quad V_D\cos\beta = \dot{y} - L\dot{\beta}\cos\beta$$

for the rear axle.

Combining equations for the dual differential drive case results in

$$\dot{y} = \frac{V_A + V_B}{2}\cos\alpha = \frac{V_C + V_D}{2}\cos\beta$$

Note that this equation places a constraint on the relationship between front and rear axle velocities.

$$\begin{aligned}\dot{\theta} &= \frac{(V_A + V_B)\sin\alpha - (V_C + V_D)\sin\beta}{4d} \\ \dot{x} &= -\frac{(V_a + V_B)\sin\alpha + (V_C + V_D)\sin\beta}{4} \\ \dot{\alpha} &= \frac{V_A - V_B}{2L}, \quad \dot{\beta} = \frac{V_C - V_D}{2L}\end{aligned}$$

Implementation of the forward kinematics is easily done and can be simulated for sample wheel speeds without use of the brake. Fig. 13.8, shows the resulting path for sample wheel inputs which demonstrate the ability to steer the craft. The wheel speeds for this figure are

$$\begin{aligned}V_A, V_B &= 5t - t^2 + 1.5 \mp \sin(t), \quad 0 \le t \le 5 \\ V_C, V_D &= (5t - t^2)\cos(\alpha)/\cos(\beta) \pm \sin(t), \quad 0 \le t \le 5.\end{aligned}$$

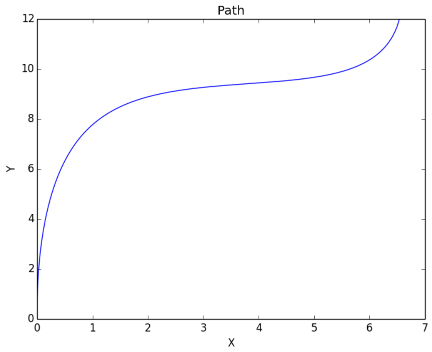


Fig. 13.8: Path for the DDD system demonstrating the ability to steer and control the vehicle with free axle pivots.

13.6 Four Axle Robot or the Four Wheel Steer Robot

The case of a four axle robot is very similar to the dual differential drive case. The angles of the four axles are α, β, γ, and δ, with α representing the angle of the axle in the first quadrant, β the angle of the axle in the second quadrant, γ the angle of the axle in the fourth quadrant, and δ the angle of the axle in the third quadrant. Let the hinge point be located by vector $\vec{r}$ with components of magnitude r_x and r_y with respect to the centroid of the robot, and have the wheel located at distance L from the hinge. Then the velocities of the ends of the axles are given below. The constraints for the front two axles are:

$$V_{A_x} = \dot{x} - r_y\dot{\theta} - \dot{\alpha}L\sin\alpha = -V_A\sin\alpha,$$

$$V_{A_y} = \dot{y} + r_x\dot{\theta} + \dot{\alpha}L\cos\alpha = V_A\cos\alpha,$$

$$V_{B_x} = \dot{x} - r_y\dot{\theta} + \dot{\beta}L\sin\beta = -V_B\sin\beta,$$

$$V_{B_y} = \dot{y} - r_x\dot{\theta} - \dot{\beta}L\cos\beta = V_B\cos\beta,$$

and the constraints for the rear two axles are:

$$V_{C_x} = \dot{x} + r_y\dot{\theta} + \dot{\gamma}L\sin\gamma = -V_C\sin\gamma,$$

$$V_{C_y} = \dot{y} - r_x\dot{\theta} - \dot{\gamma}L\cos\gamma = V_C\cos\gamma,$$

$$V_{D_x} = \dot{x} + r_y\dot{\theta} - \dot{\delta}L\sin\delta = -V_C\sin\delta,$$

$$V_{D_y} = \dot{y} + r_x\dot{\theta} + \dot{\delta}L\cos\delta = V_C\cos\delta.$$

These equations reduce to the DDD case when the offset is removed, i.e., when pivots are located in the center of the robot. The consequence is that the constraint these equations present is $\alpha = \beta$ and $\gamma = \delta$. For any other angular relationships the wheels' kinematic constraints would conflict and the robot would be locked in place. In the general case, we must have a relation $\alpha = \beta + \epsilon_1$ and $\gamma = \delta + \epsilon_2$ where ϵ_1, ϵ_2 are the corrections due to the offset.

However, clearly there are admissible motions, such as the case in which

$$V_{A_y} = V_{B_y} = V_{C_y} = V_{D_y} = \dot{y},$$

$$V_{A_x} = V_{B_x} = V_{C_x} = V_{D_x} = 0,$$

$$\dot{\theta} = \alpha = \beta = \gamma = \delta = \dot{x} = 0.$$

In other words, a vehicle that already has forward motion could maintain it with all brakes unlocked. Given the constraint that the angles must remain equal, the kinematics of the FWS robot are identical to those of the DDD robot as expected.

The system that emerges is one where the split axles are connected to the center of the robot as shown in Fig. 13.9. The locking mechanism will lock the axles in line, but leave them free to pivot with respect the frame. This produces a robot which has DDD motion normally. When the pivot brakes are released, then the axles can separate and the wheels move to a configuration that allows in place rotation.

So, based on the kinematics, we see that linear motion is possible for the both vehicles when the pivot brakes are locked or free. The DDD vehicle can also turn without locks on the pivots. The kinematic constraint induced by the body connection between front and rear axles places constraints on wheel motion (as expected). Violating these will cause wheel slip and slide. You can think of DDD motion as simply two differential drive robots moving in tandem.

The FWS system is more complicated and the dynamics do allow unlocked pivots during a turn as long as not all are unlocked. So, dynamic turns can be performed by acting on axles sequentially. One may employ motion sequences such as

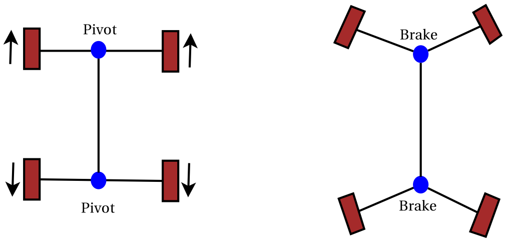


Fig. 13.9: Hybrid between the DDD and FWS designs. This places the pivots at the center allowing different axle angles. This design also holds costs by only using two brakes.

1. Unlock rear axle pivots
2. Change rear wheel velocities
3. Lock rear axle pivots
4. Unlock front axle pivots
5. Change front wheel velocities
6. Lock front axle pivots

to turn the robot without performing a complete stop. This configuration works very much like an Ackerman drive other than the ability to stop and rotate in place. A simulation is shown of the DDD-FWS hybrid in Fig. 13.10.

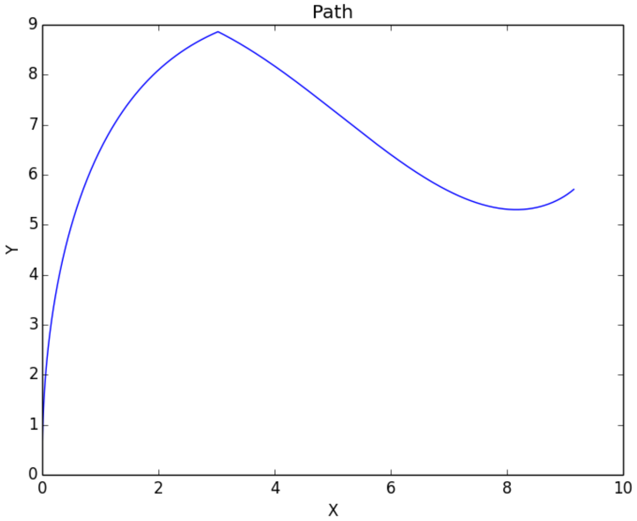


Fig. 13.10: Path for the DDD-FWS hybrid system demonstrating the ability to steer and control the vehicle with free axle pivots. The system stops halfway and resets pose.

13.7 Maneuverability

In this section we study the ability of a particular wheel configuration to move around in the environment. Each wheel must respect its sliding constraint which limits the motion in the workspace. We will see that the ability to move in the environment is a combination of the mobility provided by the sliding constraint and the available steering.

The basic wheel constraint prevents motion in the direction of the axle. Recall that

$$\begin{bmatrix} C_{1f} \\ C_{1s} \end{bmatrix} R(\theta)^{-1}\dot{\xi}_I = 0$$

which means that $R(\theta)^{-1}\dot{\xi}_I$ is in the nullspace of the array $C_1 = \begin{bmatrix} C_{1f} \\ C_{1s} \end{bmatrix}$. The *Nullspace* of the matrix A is the collection of vectors v such that $Av = 0$.

We will see that this idea can be related to the ICR (instantaneous center of rotation). The axle of a standard wheel can be extended into what is called the zero motion line. For multiple fixed wheels (this can be extended to steered wheels), the zero motion lines intersect at the ICR. Recall that if the zero motion lines do not intersect at a single point then the craft does not move. This was argued in the case of the standard automobile design, four wheels, two fixed and two steered. We argued that the four wheels must all line up on two concentric circles and the wheel direction must be tangent to the circle it was on. If the ICR is a finite value in the plane, then we have a circle with finite radius. If the ICR is out at infinity, then the radius of the circle is infinite and we are traveling on a straight line.

To explore this connection, we delve more into the linear algebra of the constraint matrices. The kinematic formulas we derived are a function of the wheel constraints. It is the number of independent constraints that are important are contribute to the robot kinematics. This is connected to the rank of the constraint matrix. The rank of C_1

is the number of independent constraints or the number of linearly independent rows. The greater the rank, the more constrained the vehicle. Clearly

$$0 \leq \text{rank}(C_1) \leq 3.$$

Each constraint is related to the wheel's zero motion line. The zero motion line is the axle direction which from :eq: *eq:axledirection*, $A = \langle \cos(\alpha+\beta), \sin(\alpha+\beta)\rangle$. Contrary to what you may expect, adding steerable wheels increases the number of constraints. Keep in mind adding omniwheels or Mecanum wheels does not increase the number of constraints since those do not enforce a no slide condition.

Example for the differential drive: $\alpha_1 = \pi/2$, $\beta_1 = 0$, $\alpha_2 = -\pi/2$, $\beta_2 = 0$

$$C_1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & -1 & 0 \end{bmatrix}, \quad \text{rank}(C_1) = 1.$$

The two wheels in this case share the zero motion line. And so we only have one zero motion line. Using the formula for A above we gain $A = \langle 0, 1\rangle$ which is clearly the first two components of the constraint. Not surprising as this is by construction. The zero motion line is then the vertical line spanned by A.

Next we examine the bicycle. For the bike, we have $L_1 = L_2 = L$, $\beta_1 = \beta_2 = \pi/2$, $\alpha_1 = 0$, $\alpha_2 = \pi$. From these two wheels we obtain the two constraints which are loaded into C_1.

$$C_1 = \begin{bmatrix} 0 & 1 & L \\ 0 & -1 & L \end{bmatrix}, \quad \text{rank}(C_1) = 2.$$

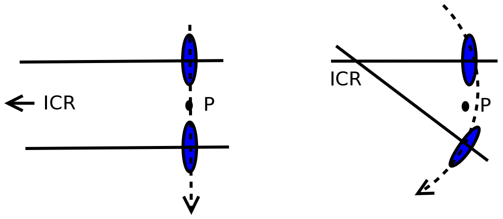


Fig. 13.11: A fixed turn bike wheel.

The first two components of the constraints agree with the differential drive example, but we have a different result for the constraint matrix.

An Ackerman drive can be modeled as two bicycles attached together. To drive the front wheels must respect the Ackerman angle constraint, Fig. 13.12. Because of the Ackerman steering angle constraint, with the back wheels and one front free, you have selected the location of the ICR. This means that the other steered wheel zero motion line is prescribed. So, it must mean that the four rows in the constraint matrix are linearly dependent. We know that the two rows for the fixed wheels are the same line. This tells us one fixed wheel and one steered wheel are sufficient. Thus we have two linearly independent rows for the the constraint matrix.

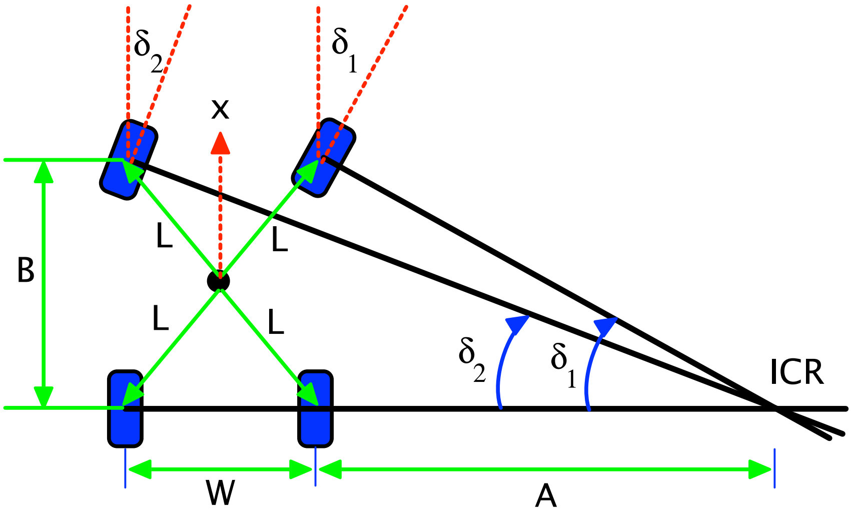


Fig. 13.12: The Ackerman Design

The algebra for the general case is difficult, however, we can put some values on this diagram. Let right front wheel (steered) be wheel 1, left front wheel be wheel 2 and the left rear wheel be wheel 3. We don't need the fourth wheel. The diagram has wheels 1 and 2 fully labeled and for wheel 3, the same conventions are followed. Assume that $A = 5$, $B = 3$ and $W = 2$. This means that

$$\delta_1 = \text{atan2}(B, A) = \text{atan2}(3, 5), \quad \delta_2 = \text{atan2}(B, A + W) = \text{atan2}(3, 7), \quad \delta_3 = 0$$

and

$$\alpha_1 = -\text{atan2}(W/2, B/2) = -\text{atan2}(1, 3/2)$$

$$\alpha_2 = -\alpha_1, \quad \alpha_3 = \pi - \alpha_2$$

This provides us with

$$\beta_1 = -\pi/2 - \alpha_1 - \delta_1, \quad \beta_2 = \pi/2 - \alpha_2 - \delta_2, \quad \beta_3 = -\pi/2 - \alpha_3 - \delta_3$$

we can plug each into the constraint equation

$$\langle \cos(\alpha + \beta), \sin(\alpha + \beta), L\sin(\beta) \rangle$$

to build the matrix C. This is done with the following program.

```
import numpy as np
import numpy.linalg as lin
import math

A = 5
B = 3
W = 2
d1 = math.atan2(B,A)
d2 = math.atan2(B,A+W)
d3 = 0.0
a1 = -math.atan2(W/2.0,B/2.0)
a2 = ath.atan2(W/2.0,B/2.0)
a3 = math.pi - a2

b1 = -math.pi/2.0 - a1 - d1
b2 = math.pi/2.0 - a2 - d2
b3 = -math.pi/2.0 - a3 - d3
L = math.sqrt(W*W+B*B)/2.0

C = np.array([
[math.cos(a1+b1) , math.sin(a1+b1), L*math.sin(b1)],
[math.cos(a2+b2) , math.sin(a2+b2), L*math.sin(b2)],
[math.cos(a3+b3) , math.sin(a3+b3), L*math.sin(b3)]])

print C

r = lin.matrix_rank(C)
print r
```

The output

```
[[ -5.14495755e-01  -8.57492926e-01  -1.80073514e+00]
 [  3.93919299e-01   9.19145030e-01   9.84798246e-01]
 [ -3.82856870e-16  -1.00000000e+00   1.50000000e+00]]
2
```

In general, if the rank of C_1 is greater than one then the vehicle at best can only travel a line or a circle. Rank = 3 means no motion at all. We can define the *degree of mobility* = δ_m, also known as *DDOF - differential degrees of freedom*,

$$\delta_m \equiv \dim \mathcal{N}(C_1) = 3 - \text{rank}(C_1)$$

This is the robot's degrees of freedom or a measure of the local mobility of the robot.

For a differential drive the degree of mobility is $\delta_m = 2$. We define the *degree of steerability*, δ_s as

$$\delta_s \equiv \text{rank}(C_{1,s}).$$

Note that increasing this rank increases steerability, but since C_1 contains $C_{1,s}$, it will decrease mobility. We can also define DOF, *the degrees of freedom*, which is based on the workspace dimension which is two or three.

We have $N_f = 2$ and $N_s = 2$.

$$\text{rank}(C_{1f}) = 1$$

(since they share an axle). Since all axle lines must intersect in a point for the vehicle to move (example above), once you prescribe on wheel, you have prescribed both wheels.

$$\text{rank}(C_{1s}) = 1$$

So:

$$\text{rank} \begin{bmatrix} C_{1f} \\ C_{1s} \end{bmatrix} = 2$$

Thus $\delta_m = 1$ and $\delta_s = 1$.

We can contrast this with the equal steer angle vehicle. This has $N_f = 2$ and $N_s = 2$ just like the Ackerman. However the three rows are linearly independent (rank is 3). This provides us with $\delta_m = 0$ and $\delta_s = 1$.

An important concept is the Degree of Maneuverability, δ_M,

$$\delta_M = \delta_m + \delta_s.$$

This measures the degrees of freedom the robot can operate in a global sense. So even if the robot does not have full mobility in a local sense, the robot can operate through a series of movements in this larger sense. A differential drive robot for example $\delta_M = \delta_m + \delta_s = 2$.

Degree of Maneuverability is equivalent to control degrees of freedom. A *holonomic* robot is a robot with ZERO nonholonomic constraints. A holonomic kinematic constraint can be expressed as an explicit function of position variables alone. A robot is holonomic if and only if DDOF = DOF. A robot is said to be omnidirectional if it is holonomic and DDOF = 3. This means that the robot can *Maneuver* and *Orient*.

Table 13.1: Summary of some common configurations.

Configuration	Maneuverability	Mobility	Steerability
A. Omniwheel	$\delta_M = 3$	$\delta_m = 3$	$\delta_s = 0$
B. Differential	$\delta_M = 2$	$\delta_m = 2$	$\delta_s = 0$
C. Omni-Steer	$\delta_M = 3$	$\delta_m = 2$	$\delta_s = 1$
D. Tricycle	$\delta_M = 2$	$\delta_m = 1$	$\delta_s = 1$
E. Two-Steer	$\delta_M = 3$	$\delta_m = 1$	$\delta_s = 2$

13.8 Problems

1. Derive (5.4).
2. Assume that you have a square robot which is 50 cm per side and uses four 15 cm diameter omniwheels with$\gamma = 0$ configuration. The wheels are mounted at each corner at 45° to the sides. Find the kinematic equations of motion.
3. What are the motion equations for the Syncro Drive System as a function of wheel velocity and wheel turn angle? Use r for wheel radius.
4. Derive the equations of motion for the three wheel omniwheel robot, Fig. 13.6.

CHAPTER

FOURTEEN

MANIPULATORS: ARMS AND LEGS

Note: Write this chapter. Currently these are converted slides.

There is a wealth of literature on robotic manipulators especially in the context of fixed industrial robots. This is an important area of study but will not be duplicated here. This text is focused on mobile systems. Manipulators enter as arms and legs. It is possible to mount a robotic manipulator on the robot chassis. The group of manipulators which are only involved in transporting the robot we will call legs, and the rest will be called arms. The control of these might not be at all different and strictly rely on use.

In the first section we lay down some basic terms required for the methods presented on forward kinematics derivation. The actual forward and inverse kinematics will be treated in the following section on Denavit-Hartenberg Parameters.

14.1 Representation of objects in space

Note: These are from slides. Need to get a more expository version (with images).

Before we can derive the formwad kinematics for a serial chain manipulator, we need to be able to describe in a very general manner, the changes in position or orientation of a solid object. So, we begin by reviewing how we represent a rigid body in space. For this text all of our systems will be combinations of rigid elements and we will leave the flexible arms for other texts.

The direction the end effector faces is the approach direction or the principle direction. We will use the vector a to indicate the unit vector pointing in the approach direction. A second orthogonal direction to a can be found and will be called n. A third direction, o, selected using the cross-product $o = a \times n$.

Load the three vectors row-wise into a matrix

$$F = \begin{pmatrix} n_x & o_x & a_x \\ n_y & o_y & a_y \\ n_z & o_z & a_z \end{pmatrix}$$

and since these are mutually orthogonal vectors, we can see that this acts like a coordinate system. Let $c = [c_1, c_2, c_3]$

$$c' = Fc = c_1 n + c_2 o + c_3 a$$

and F transforms from one coordinate system to another. Note that F can generate scalings, rotations, reflections, shears. Thus it is a transformation matrix. It is a linear transformation and so soes not translate (since $F0 = 0$). To translate we need to augment by a shift vector

$$c' = Fc + T$$

This coordinate transformation and translation is known as an affine map. Although the affine map works well as a way to shift coordinate systems, the linear transformation quality will turn out to be important and so to gain rotations, scalings as well as the translation, but keeping the linearity property, we inflate our matrix and introduce homogeneous coordinates.

14.1.1 Homogeneous Coordinates

The translation will be appended as a final column in the matrix and a unit basis vector is added to the last row giving us

$$F = \begin{pmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

This turns out to be a rotation (scale, reflection) followed by a translation. A translation matrix can be formed by

$$T = \begin{pmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Successive motion can be computed by matrix multiplication. Let R be a rotation and T be a translation. Then

$$M = TR$$

is the matrix that describes the rotation by R followed by translation by T.

$$\begin{pmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} n_x & o_x & a_x & 0 \\ n_y & o_y & a_y & 0 \\ n_z & o_z & a_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Homogeneous coordinates are defined by appending a "1" at the bottom of a normal 3 component position vector giving

$$\xi = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Allows for general transforms: $\xi' = A\xi$, which are linear transforms. In most of our applications, we will be interested in a rotation and then a translation. Shear and reflection are not an issue here since these changes in coordinates will apply to rigid robot hardware which (for now) does not experience reflection and shear. So for example a rotation about the z axis and then a translation of (t_1, t_2, t_3) would have the following tansformation matrix.

$$\xi' = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & t_1 \\ \sin\theta & \cos\theta & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \xi$$

It is useful to review the basic rotations about the three axes:

- About z

$$R_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- About x

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- About y

$$R_y = \begin{pmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Assume that you are given the following motions: Rotate about the x-axis 30 degrees, translate in y by 3cm, and rotate about the z axis 45 degrees. Find the coordinate transformation.

$$R_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos 30 & -\sin 30 & 0 \\ 0 & \sin 30 & \cos 30 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad R_2 = \begin{pmatrix} \cos 45 & -\sin 45 & 0 & 0 \\ \sin 45 & \cos 45 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Then the transformation is $M = R_2 T R_1$

$$= \begin{pmatrix} \cos 45 & -\sin 45 & 0 & 0 \\ \sin 45 & \cos 45 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos 30 & -\sin 30 & 0 \\ 0 & \sin 30 & \cos 30 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos 45 & -\sin 45 & 0 & 0 \\ \sin 45 & \cos 45 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos 30 & -\sin 30 & 3 \\ 0 & \sin 30 & \cos 30 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos 45 & -\sin 45 \cos 30 & -\sin 45 \sin 30 & -3 \sin 45 \\ \sin 45 & \cos 45 \cos 30 & -\cos 45 \sin 30 & 3 \cos 45 \\ 0 & \sin 30 & \cos 30 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

14.1.2 RPY Angles and Euler Angles

Roll-Pitch-Yaw (RPY) angles provide the position and orientation of a craft by using a translation to body center and then three rotation matrices for craft pose.

- Rotation about a (z axis) - Roll
- Rotation about o (y axis) - Pitch
- Rotation about n (x axis) - Yaw

$$M = R_n R_o R_a T$$

Euler angles provide the position and orientation of a craft by using a translation to body center and then three rotation matrices for craft pose. However - reference is with respect to the body, not the world coordinates.

- Rotation about a (z axis) - Roll
- Rotation about o (y axis) - Pitch
- Rotation about a - Roll

$$M = R_a R_o R_a T$$

14.1.3 Combined Transforms

Begin with a point x in space. An application of a transformation, T_1, with respect to the global frame carries this point to a new point x':

$$x' = T_1 x$$

We can think of the new point x' as movement of the original point x. This can be repeated. Apply another transformation T_2 to the new point x':

$$x" = T_2 x' = T_2(x') = T_2(T_1 x) = T_2 T_1 x$$

Note that each transform was done with respect to the fixed frame.

Again, begin with a point x in space. If we view the transformation, T from the perspective of the point (which will be fixed), then it appears that the “fixed” frame is moving AND that the motion is in the *opposite* direction of the fixed frame transformation. Opposite here would be the inverse transformation: T^{-1}. Thus combined transformations from the point’s “point of view”:

$$T^{-1} = T_2^{-1}T_1^{-1}, \quad \text{or} \quad T = \left(T_2^{-1}T_1^{-1}\right)^{-1}$$

$$T = T_1 T_2$$

This places the list of operations in reverse order. Successive transformations relative to the global frame are left multiplied:

$$T = T_n T_{n-1} \ldots T_1 T_0$$

For example, take a rotation about z of 30 degrees, R_1, followed by a rotation about x by 60 degrees, R_2:

$$R = R_2 R_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos 60 & -\sin 60 & 0 \\ 0 & \sin 60 & \cos 60 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos 30 & -\sin 30 & 0 & 0 \\ \sin 30 & \cos 30 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Successive transformations relative to the moving frame are right multiplied:

$$T = T_0 T_1 \ldots T_{n-1} T_n$$

For example, take a rotation about x by 45 degrees, R, followed by a translation in z by 4 cm, T:

$$M = TR = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos 60 & -\sin 60 & 0 \\ 0 & \sin 60 & \cos 60 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The formula for inverting transformation matrices is given by

$$T^{-1} = (T_n T_{n-1} \ldots T_1 T_0)^{-1} = T_0^{-1} T_1^{-1} \ldots T_{n-1}^{-1} T_n^{-1}$$

How does one invert the transformations? For us this is simplified since we are restricted to rotations and translations which are easily inverted. Rotation matrices are orthogonal and so

$$R^{-1} = R^T$$

For example, the inverse of the 60 degree rotation mentioned above:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos 60 & -\sin 60 & 0 \\ 0 & \sin 60 & \cos 60 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos 60 & \sin 60 & 0 \\ 0 & -\sin 60 & \cos 60 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translation matrices are simple as well. One just negates the translation components.

Thus:

$$\begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 & 0 & -a \\ 0 & 1 & 0 & -b \\ 0 & 0 & 1 & -c \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Thus we can just undo the transformations individually.

14.2 Denavit-Hartenberg Parameters

Note: Again, these are converted slides and need to be written up properly. The slow part for these sections is getting good images created.

Given joint angles and actuator lengths one can compute the end effector position. Thus it is possible to compute the effector path as a function of arm movements.

$$\big(\theta_1(t), ..., \theta_n(t)\big) \to p(t)$$

However it is MUCH harder to find the angle functions if you are given the end effector path:

$$p(t) \to \big(\theta_1(t), ..., \theta_n(t)\big)$$

A simple way to relate the end effector to the base for a serial chain manipulator is to see each link as a transformation of the base coordinate system. This is the approach suggested by Denavit and Hartenberg.

14.2.1 Denavit-Hartenberg formalism

Provides a standard way to build kinematic models for a robot. It is a simple concept and the complexity is hidden in the composition of linear transformations. This provides a powerful way to generate the kinematic equations in a consistent form.

As suggested above, we follow out the links of the manipulator, and see them as rotations and translations of the coordinate system:

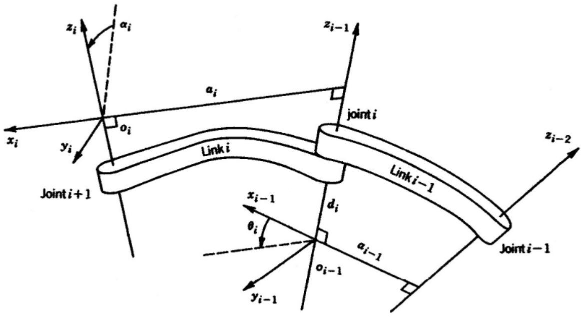
$$P = P_0 P_1 ... P_{n-1} P_n$$

where $P_k = R_z T_z T_x R_x$

- Each link is assigned a number. Normally start with the base and work towards the effector.
- All joints are represented by the z axis, z_i where the z axis is the axis of revolution (right hand rule for orientation).
- θ_i will represent the rotation about the joint.
- The x axis, x_i is in the direction that connects the links. [Well, connects the z axes of each joint.]
- a_i is link length.
- α_i will be the angles between z axes (if they are not parallel).
- d_i will represent the offset along the z axis.

Thus, the translation from one joint to the next involves a rotation, translation, translation and a rotation:

- Rotate about the local z axis angle θ.
- Translate (offset) along the z axis amount d.



- Translate (link length) along x amount a.
- Rotate about the new x axis (the joint twist) amount α.

This set of transformations will then change the coordinate system to the next link in the serial chain.

$A_{n+1} =$

$$\begin{pmatrix} \cos\theta_{n+1} & -\sin\theta_{n+1} & 0 & 0 \\ \sin\theta_{n+1} & \cos\theta_{n+1} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_{n+1} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & a_{n+1} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_{n+1} & -\sin\alpha_{n+1} & 0 \\ 0 & \sin\alpha_{n+1} & \cos\alpha_{n+1} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$A_{n+1} =$

$$\begin{pmatrix} \cos\theta_{n+1} & -\sin\theta_{n+1}\cos\alpha_{n+1} & \sin\theta_{n+1}\sin\alpha_{n+1} & a_{n+1}\cos\theta_{n+1} \\ \sin\theta_{n+1} & \cos\theta_{n+1}\cos\alpha_{n+1} & -\cos\theta_{n+1}\sin\theta_{n+1} & a_{n+1}\sin\theta_{n+1} \\ 0 & \sin\alpha_{n+1} & \cos\alpha_{n+1} & d_{n+1} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

A parameter table keeps track for each link, the values of θ, d, a and α.

Starting from the base of the robot, we can built the transformation that defines the kinematics:

$$A = A_1 A_2 \dots A_n$$

14.2.2 D-H Two Link Example

Link	θ	d	a	α
1	θ_1	0	a_1	0
2	θ_2	0	a_2	0

$$A_1 = \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & a_1\cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & a_1\sin\theta_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_2 = \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & a_2\cos\theta_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & a_2\sin\theta_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

So,

$$A = A_1A_2 = \begin{pmatrix} \cos(\theta_1+\theta_2) & -\sin(\theta_1+\theta_2) & 0 & a_2\cos(\theta_1+\theta_2)+a_1\cos\theta_1 \\ \sin(\theta_1+\theta_2) & \cos(\theta_1+\theta_2) & 0 & a_2\sin(\theta_1+\theta_2)+a_1\sin\theta_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

14.3 DH Inverse Kinematics

How can we use this technology to solve the inverse kinematics problem?

$$T^{-1} = T_0^{-1}T_1^{-1}\ldots T_{n-1}^{-1}T_n^{-1}$$

In each matrix one can solve algebraically for θ_i in terms of the orientation and displacement vectors. What does this look like for the two link manipulator?

Link	θ	d	a	α
1	θ_1	0	a_1	0
2	θ_2	0	a_2	0

$$A_1 = \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & a_1\cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & a_1\sin\theta_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_2 = \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & a_2\cos\theta_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & a_2\sin\theta_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

So,

$A = A_1A_2 =$

$$\begin{pmatrix} \cos(\theta_1+\theta_2) & -\sin(\theta_1+\theta_2) & 0 & a_2\cos(\theta_1+\theta_2)+a_1\cos\theta_1 \\ \sin(\theta_1+\theta_2) & \cos(\theta_1+\theta_2) & 0 & a_2\sin(\theta_1+\theta_2)+a_1\sin\theta_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Then we have that the transformation carries the frame to some frame description $A = F$:

$$A = \begin{pmatrix} \cos(\theta_1+\theta_2) & -\sin(\theta_1+\theta_2) & 0 & a_2\cos(\theta_1+\theta_2)+a_1\cos\theta_1 \\ \sin(\theta_1+\theta_2) & \cos(\theta_1+\theta_2) & 0 & a_2\sin(\theta_1+\theta_2)+a_1\sin\theta_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = F$$

Then the location of the end effector $(x, y, z) = (p_x, p_y, p_z)$:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a_2\cos(\theta_1+\theta_2)+a_1\cos\theta_1 \\ a_2\sin(\theta_1+\theta_2)+a_1\sin\theta_1 \\ 0 \end{pmatrix}$$

How can we use this technology to solve the inverse kinematics problem?

$$T^{-1} = T_0^{-1}T_1^{-1}\dots T_{n-1}^{-1}T_n^{-1}$$

In each matrix one can solve algebraically for θ_i in terms of the orientation and displacement vectors. What does this look like for the two link manipulator?

Recall that

$$A_1 = \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & a_1\cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & a_1\sin\theta_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_2 = \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & a_2\cos\theta_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & a_2\sin\theta_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Thus

$$A = A_1(\theta_1)A_2(\theta_2) = \begin{pmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Right multiply to decouple: $A_1 = AA_2^{-1}$

$$= \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & a_1\cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & a_1\sin\theta_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & -a_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Note that $a_1 \cos\theta_1 = p_x - a_2 n_x$ and $a_1 \sin\theta_1 = p_y - a_2 n_y$

This provides us with

$$\theta_1 = \text{atan2}\left(\frac{p_y - a_2 n_y}{a_1}, \frac{p_x - a_2 n_x}{a_1}\right)$$

From $\cos\theta_1 = \cos\theta_2 n_x - \sin\theta_2 o_x$ and $-\sin\theta_1 = \sin\theta_2 n_x + \cos\theta_2 o_x$
we can solve for θ_2.

$$\begin{pmatrix} \cos\theta_1 \\ -\sin\theta_1 \end{pmatrix} = \begin{pmatrix} n_x & -o_x \\ n_x & o_x \end{pmatrix} \begin{pmatrix} \cos\theta_2 \\ \sin\theta_2 \end{pmatrix}$$

$$\begin{pmatrix} \cos\theta_2 \\ \sin\theta_2 \end{pmatrix} = \frac{1}{2n_x o_x} \begin{pmatrix} o_x & o_x \\ -n_x & n_x \end{pmatrix} \begin{pmatrix} \cos\theta_1 \\ -\sin\theta_1 \end{pmatrix}$$

So … $\theta_2 = \text{atan2}\left(o_x(\cos\theta_1 - \sin\theta_1), -n_x(\cos\theta_1 + \sin\theta_1)\right)$

There is a problem. The two link example has two degrees of freedom. The assumption here is that you have four variables to input (four degrees of freedom): p_x, p_y, n_x, n_y. You may not know n_x, n_y.[2] For general systems this approach will succeed if you have enough degrees of freedom in your robot.

The general approach is to form matrix A analytically and set to final pose matrix. Then by applying inverses A_k^{-1}, examine intermediate results looking for terms which provide one of the angle variables: θ_j.

Producing actual robot motion means moving the end effector along some path $(x(t), y(t), z(t))$.

One really wants

$$\left(\theta_1(t), ..., \theta_n(t)\right) = f^{-1}(p(t), n(t), o(t), a(t))$$

There is no reason to expect that there exists a solution, that you can find the solution, or that the solution is unique.

Kinematic equations are derived by the developer of the robot. Inverse kinematic formulas are derived in an "ad hoc" manner.

How?

$$p(t) \to \left(\theta_1(t), ..., \theta_n(t)\right)$$

Assume that you have $(\theta_1, ..., \theta_n) = f(p, n, o, a)$.

For each t, solve

$$\begin{bmatrix} \theta_{1k} \\ \theta_{2k} \\ \vdots \\ \theta_{nk} \end{bmatrix} = \begin{bmatrix} \theta_1(t_k) \\ \theta_2(t_k) \\ \vdots \\ \theta_{nk} \end{bmatrix} = \begin{bmatrix} f_1(p(t_k), n(t_k), o(t_k), a(t_k)) \\ f_2(p(t_k), n(t_k), o(t_k), a(t_k)) \\ \vdots \\ f_n(p(t_k), n(t_k), o(t_k), a(t_k)) \end{bmatrix}$$

[2] We will address the specific situation in a few slides.

CHAPTER FIFTEEN

FILTERING AND STATE ESTIMATION

Consider the two robots given in Fig. 15.1. How should we approach designing robots to function in conjunction with humans or instead of humans? For example:

- Robotic system to move goods through a distribution center.
- Robotic system to care for the elderly.
- Robotic system to perform tasks in hostile environments (deep sea, reactors, space, underwater caves, etc).
- Robotic systems for assistive technology.

To do this we need to have perception of a changing environment, we need to make decisions about how to respond based on the robot function (goals) and we need to control effectors to carry out the intended functions. For robot perception alone, we scan the environment, segment out objects and then recognize the objects. There many types of sensors used to just understand the surrounding environment:

- Contact sensors
- Internal Sensors
- Accelerometers
- Gyroscopes
- Compasses
- Proximity Sensors
- Sonar
- Radar
- Laser range finders
- Infrared
- Cameras
- GPS

One of the basic functions, localization, is completely dependent on the sensors. Decisions about future actions are made based on the sensors. Feedback from the actuators is also based on the sensors. Many aspects of the system ride on the sensors. The Sensors Chapter presented a number and variety of sensing systems. There is a vast array of sensors which can sense or measure physical quantities. Accuracy on sensors varies greatly with some very accurate and others having considerable errors.

Fig. 15.1: Examples of two robotics systems that interact with humans.

Problem: How can we design a system which can function without human input but has random elements in the environment? How can the system determine its location accurately enough to navigate? How can the robot use manipulators around humans safely if manipulator location, feedback and control are uncertain. In manufacturing systems, we are able to instrument objects of interest and highly constrain the environment. The robot might not be local and the object to be manipulated may have a known location. Clearly variation and noise occur, but not to the degree found in robots that are intended to go into the world and operate outside the confines of a factory.

15.1 Sensor Noise and Measurement

The main concern of this chapter is the errors involved with the measurements. There are plenty of ways error can enter. A brief list of error types is given below.

- Intrinsic ability of the sensor
- Connection of the sensor to the world
- Connection of the sensor to the electronics
- Sampling and Aliasing
- Interference
- Precision
- Accuracy
- Signal to noise ratio

The current values of the variables in the system is called the *state*. Often this is represented as x_k (a vector of real values), the state at time step k (an integer). By x_k we mean the system's configuration which includes pose, dynamics, and internal measurements that are relevant to the problem. The temperature of the CPU may be an important variable in the system, but probably has little to do with localization. So x_k does not contain everything. All of the sensors have uncertainty, i.e. they are very noisy. For example, smooth surfaces cause sonar and lasers to reflect less back to the source and thus give wrong ranging results. This is known as specular reflection. Normally this results in estimating the object as much further away. So we don't know the robot's state. We can only estimate it. To gain an accurate estimate, we must model the type of error present in the system. Clearly we design the system to minimize the errors, but one cannot design them all away. The greatest error normally encountered is with the sensor itself. This will be our focus.

How can we model this error or uncertainty? Error is modeled using probabilistic tools. Typically we ask "what is the error of this measurement for a particular state"? We can define $p(z_k|x_k)$ as the probability or likelihood of getting the measurement value z_k given we are reading state x_k. Can we determine or model $p(z_k|x_k)$? Again, the question we ask here is … what is the current status of the robot? What are all the values of all the relevant parameters for the robot's relation to the environment?

If we happen to know something about the environment, say we have a map and a set of expectations based on that map, does this change our probability? This is pretty intuitive. If my previous location was near San Francisco and as a ground robot can only travel at some speed, then the probability of seeing the Eiffel tower should be low. In this case can we write down $p(z_k|x_k, m_k)$ where m_k is a map of the environment?

In many sensing systems we may have redundant measurements of some quantities. We may actually measure combinations of components and possibly miss some. For example, I might be able to measure velocity but not position. I might know speed but not the components of velocity. This means that the measurement z is some function of the state x with errors: $z = h(x) + \delta$. Is it possible to determine h and δ? We will normally have an array of values $z_k = \{z_k^1, z_k^2, \dots, z_k^k\}$. We will assume the measurements are independent:

$$p(z|x, m) = \prod_{k=1}^{N} p(z_k|x, m).$$

What is involved in a measurement? What can activate the sensor? Measurement can be caused by:

- a known obstacle
- interference
- other obstacles

- random events
- maximum range
- sensor or software errors

The error or noise enters in the measurement of known item, the position of known item, the position of other obstacles and as a missing item.

15.2 State Estimation

15.2.1 State variables and Errors

Recall that for any state variable we have

- True value: y
- Measured value: $\tilde{y}$
- Estimated value: $\hat{y}$

The true value is not known. It is what we seek. The measured value comes from the sensor which is subject to error as listed above. Estimated value comes from measurement and system model.

Basic errors that are used:

- Measurement error: $v = y - \tilde{y}$
- Residual error: $e = \tilde{y} - \hat{y}$

We don't know v obviously. The residual error, e, is based on a model of the system and is known explicitly. Basic error types: we are concerned with two fundamental error types

- Systematic error - deterministic
- Random error - non-deterministic

Systematic errors are errors of design or implementation:

- Incorrectly mounted sensor
- Blocked sensor
- Sensor biased by hardware
- Sampling issues
- Resolution issues
- Incomplete measurements
- Sensitivity
- Nonlinearity

Random errors

- Based on white or Gaussian noise

- Actually could be any distribution, but Gaussian is standard.

15.2.2 Filters

The idea of filtering is to use the measurement PLUS the model to provide a better estimate of the state. Finding the model may be the hardest part but the part that makes the process effective. For example, it is where the Kalman Filter enters. The Kalman filter uses a linear time stepping model and environmental data to improve state estimation.

What does one mean by filter? In this case we are attempting to filter out noise. Simple filters in signal processing often filter in the frequency domain. For example filtering out high frequencies since this is often noise. We can filter out noise by fitting the data to a model. We assume that the data represents a constant and so we can compute the mean of the data. This model can also be represented by the distribution that the data appears to have come from, e.g. a normal distribution.

The distribution that the data comes from can change over time. If two data items come from the same distribution then we have some reason to believe a mean is a good filter value. If not, how do we balance data items which have different reliability?

15.2.3 High and Low pass filters

Assume that you have digitized signal, meaning the analog sensed values have been converted to numerical values sampled at regular times. Call that signal $\{z[n]\}$. If that signal has high frequency noise (static or white noise), how can you eliminate or filter out that noise? If the signal has low frequency noise (like mechanical oscillations or other forms of bias), can this be filtered out. The answer is yes. Two common filters are low and high pass filters. The low pass refers to the filter allowing low frequencies through but filtering out the higher frequencies like the static. [And similarly for the high pass filter.]

Since integration tends to smooth out signals, we use an integration formula that has an exponential decay built in. This removes the high frequencies (the static) and leaves the core signal. The algorithm is given below. Sample output may be found in Fig. 15.3.

Low Pass Filter (integration based)

```
for i from 1 to n
       y[i] := y[i-1] + a * (z[i] - y[i-1])
```

The Python code

```
import numpy as np
import pylab as plt

N = 1000
sigma = 1.0
n = np.random.normal(0,sigma,N)
t = np.linspace(0,12,N)
x = np.sin(t)
z = x + n
```

(continues on next page)

(continued from previous page)

```
y = np.zeros(N)

y[0] = z[0]
i = 1
while(i<N):
    y[i] = y[i-1] + 0.075*(z[i] - y[i-1])
    i = i+1
```

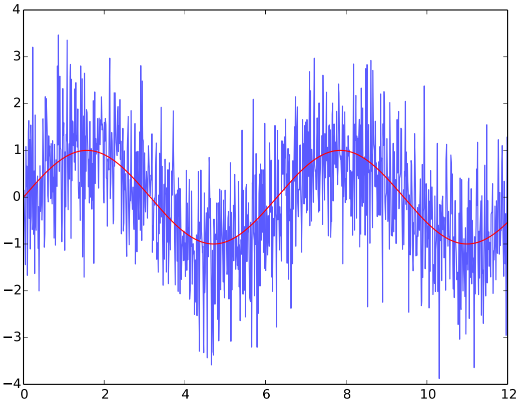


Fig. 15.2: Signal in red, noisy version of the signal in blue.

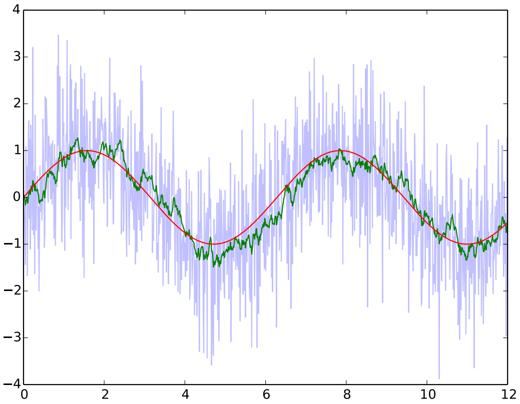


Fig. 15.3: Noisy signal in blue, filtered signal in green.

Differentiation will set constants to zero and attenuate low frequencies, filters based on differentiation formulas are employed. One such formula is given below. The output of this filter is given in Fig. 15.5.

High Pass Filter (differentiation based)

```
for i from 1 to n
     y[i] := a * (z[i] - z[i-1])
```

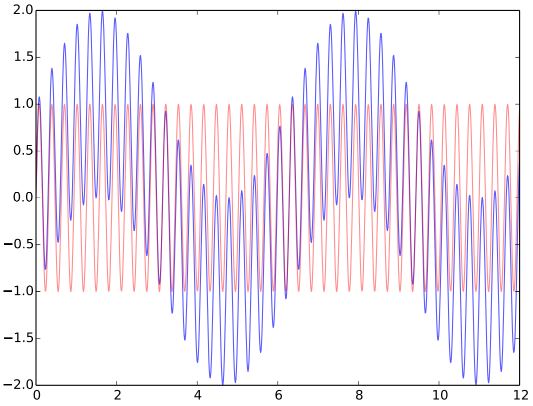


Fig. 15.4: Signal in red, noisy version of the signal in blue.

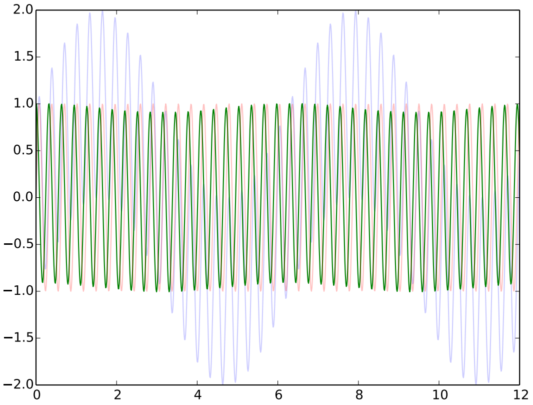


Fig. 15.5: Noisy signal in blue, filtered signal in green.

A variation of the high pass filter is

```
for i from 1 to n
     y[i] := a * (y[i-1]  + z[i] - z[i-1])
```

The band pass filter is a filter which allows a range of frequencies to pass through. One may simply try applying both a low and high pass filter. Although filters are easy to understand and to implement, designing them for a specific application can be challenging.

15.2.4 Complementary Filter

Assume that you have two different sensors (measurements from two different sources) in which one sensor has high frequency noise and the other sensor has low frequency noise. A complementary filter exploits this situation by applying a low pass filter to the first sensor data and a high pass filter to the second sensor. The two signals "complement" each other in terms of information.

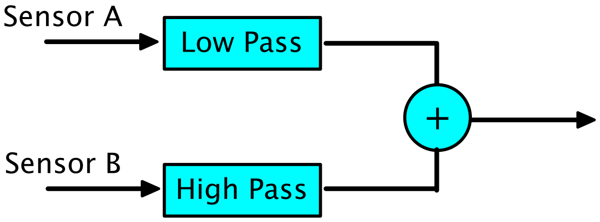


Fig. 15.6: Complementary Filter

15.3 Sensor Fusion

15.3.1 Simple Example of Sensor Fusion

Consider a system with n sensors each making a single measurement:

$$z_i, \quad i = 1, \ldots, n$$

of some unknown quantity x. The measurements really are described by

$$z_i = x + v_i, \quad i = 1, \ldots, n.$$

We seek an optimal estimate of x based on a linear combination of these measurements:

$$\hat{x} = \sum_{i=1}^{n} k_i z_i.$$

How should we proceed? It is a matter of writing down an expression for the error in the estimate and minimizing the error.

We will assume that the noise v_i is zero mean white noise (normally distributed) and independent. Thus we have that

$$E(v_i) = 0, \quad \text{and} \quad E(v_i v_j) = 0, \quad i \neq j,$$

where $E(x)$ is the expected value of x. We want the estimate to be unbiased which means that $E(\hat{x}-x) = 0$. We define optimality as minimizing the mean square error:

$$E[(\hat{x} - x)^2].$$

Lemma 1 $E[\hat{x} - x] = 0$ implies $\sum_{i=1}^{n} k_i = 1$

An unbiased estimate means that $E(\hat{x} - x) = 0$,

$$E[\hat{x}-x] = E\left[\sum_{i=1}^{n} k_i z_i - x\right] = E\left[\sum_{i=1}^{n} k_i(x+v_i) - x\right]$$

$$= E\left[\sum_{i=1}^{n} k_i x - x + \sum_{i=1}^{n} k_i v_i\right] = \sum_{i=1}^{n} k_i E[x] - E[x] + \sum_{i=1}^{n} k_i E[v_i] = 0$$

since $E(v_i) = 0$ and $E(x) = x$ we have that

$$\sum_{i=1}^{n} k_i = 1.$$

Lemma 2 $E[(\hat{x}-x)^2] = \sum_{i=1}^{n} k_i^2 \sigma_i^2$

where σ_i are the standard deviations for v_i, $E((v-E(v))^2) = \sigma^2$.

$$E[(\hat{x}-x)^2] = E\left[\left(\sum_{i=1}^{n} k_i z_i - x\right)^2\right] = E\left[\left(\sum_{i=1}^{n} k_i(x+v_i) - x\right)^2\right]$$

$$= E\left[\left(\sum_{i=1}^{n} k_i x - x + \sum_{i=1}^{n} k_i v_i\right)^2\right] = E\left[\left(\sum_{i=1}^{n} k_i v_i\right)^2\right]$$

$$= E\left[\sum_{i=1}^{n}\sum_{j=1}^{n} k_i k_j v_i v_j\right] = \sum_{i=1}^{n}\sum_{j=1}^{n} k_i k_j E[v_i v_j] = \sum_{i=1}^{n} k_i^2 \sigma_i^2. \tag{15.1}$$

Optimal Estimate

The main goal is to minimize the mean square error subject to the constraint of having the unbiased estimate:

- Minimize $\sum_{i=1}^{n} k_i^2 \sigma_i^2$ (minimize mean square error),
- Subject to $\sum_{i=1}^{n} k_i = 1$ (unbiased estimate).

We proceed using Lagrange Multipliers which will allow us to optimize a constrained function. Expressing as the Lagrangian

$$L = \sum_{i=1}^{n} k_i^2 \sigma_i^2 - \lambda\left(\sum_{i=1}^{n} k_i - 1\right)$$

we must solve

$$\nabla_k L = 0 \quad \text{with} \quad \sum_{i=1}^{n} k_i = 1.$$

Thus

$$\nabla L = \left[2k_1\sigma_1^2 - \lambda, 2k_2\sigma_2^2 - \lambda, \ldots, 2k_n\sigma_n^2 - \lambda\right] = \vec{0}$$

$$\sum_{i=1}^{n} k_i = 1$$

Solve for k_i in each gradient equation and sum

$$\sum_{i=1}^{n} k_i = \sum_{i=1}^{n} \frac{\lambda}{2\sigma_i^2} = 1$$

So, we have that

$$\lambda = \left(\sum_{i=1}^{n} \frac{1}{2\sigma_i^2}\right)^{-1}$$

This provides k_i

$$k_i = \frac{1}{\sigma_i^2}\left(\sum_{i=1}^{n} \frac{1}{\sigma_i^2}\right)^{-1}$$

and we obtain the estimate

$$\hat{x} = \frac{\displaystyle\sum_{i=1}^{n} \frac{z_i}{\sigma_i^2}}{\displaystyle\sum_{i=1}^{n} \frac{1}{\sigma_i^2}}.$$

From (15.1) we can also gain an estimate of the variance for the estimate, $\hat{x}$ above:

$$\begin{aligned}\sigma^2 = \sum_{i=1}^{n} k_i^2\sigma_i^2 &= \sum_{i=1}^{n}\left(\frac{1}{\sigma_i^2}\left(\sum_{i=1}^{n}\frac{1}{\sigma_i^2}\right)^{-1}\right)^2 \sigma_i^2 \\ &= \left(\sum_{i=1}^{n}\frac{1}{\sigma_i^2}\right)^{-2}\sum_{i=1}^{n}\left(\frac{1}{\sigma_i^2}\right) = \left(\sum_{i=1}^{n}\frac{1}{\sigma_i^2}\right)^{-1}\end{aligned} \tag{15.2}$$

15.3.2 Simple example using uniform variance

If the variances are the same, $\sigma_i = \sigma$, then

$$\sum_{i=1}^{n}\frac{1}{\sigma_i^2} = \frac{1}{\sigma^2}\sum_{i=1}^{n} 1 = \frac{n}{\sigma^2}$$

and so

$$\hat{x} = \frac{\frac{1}{\sigma^2}\displaystyle\sum_{i=1}^{n} z_i}{\frac{n}{\sigma^2}} = \frac{1}{n}\sum_{i=1}^{n} z_i$$

which is the average.

15.3.3 Example with different variances

Say you measure something three different ways and you want to merge these measurements into a single estimate. How does one specifically go about it. Assume that the three devices do return normally distributed measurements. But what is the actual distribution? Keep in mind for normal distributions, we only need to track the mean and standard deviation and those are complete descriptors for the distribution.

Recall that the mean and the standard deviation are

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i, \qquad \sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2}$$

Assume that you sample three sensors with 20 measurements each for some experiment. The data you gain is

```
2.28333   1.87365    2.12419
2.26493   1.77675    1.80968
2.33626   1.85706    2.00608
2.13676   1.83520    2.12145
... (middle removed to fit)
2.14289   1.86792    1.86616
2.21151   1.88855    2.20027
2.17112   1.95257    1.77513
2.19798   1.82083    2.25617
Means:
2.20548   1.85962    2.04204
Standard Deviations:
0.08698   0.04282    0.17674
```

The normal curves for the three sensors are

$$P_i(x|\mu,\sigma) = \frac{1}{\sigma_i\sqrt{2\pi}}\, e^{-\dfrac{(x-\mu_i)^2}{2\sigma_i^2}}$$

and are given in Fig. 15.7.

Assume the experimental setup was such that the true measurement was 2.0. The difference between the true measurement and the sensor average constitutes the systematic error. It is a constant bias term which can be removed. You need to compute the difference between the true value and the dataset average. This provides the amount you need to shift your measurement value:

```
Shift data
x shift (add) =  -0.205476607108
y shift (add) =  0.140376647675
z shift (add) =  -0.0420388951565
```

Once you have the standard deviations, we can perform a single measurement using the three sensor and then merge the three into a single estimate of the state. Assume you get the following measurements for sensors A, B and C respectively: 2.22685 1.90326 2.17253. Then the corrected measurements for sensors A, B and C are $z_1 = 2.02137$, $z_2 = 2.04363$, $z_3 = 2.13049$.

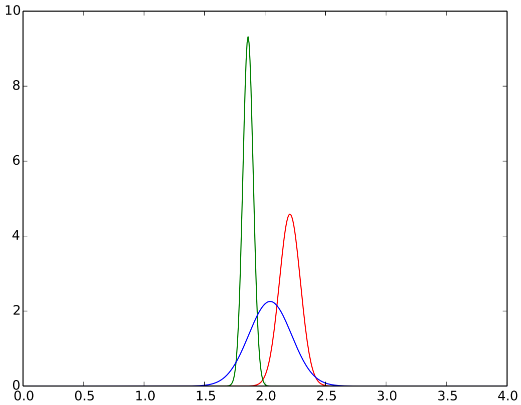


Fig. 15.7: The normal curves for the three sensors. Sensor A is shown in red, sensor B in green and sensor C in blue.

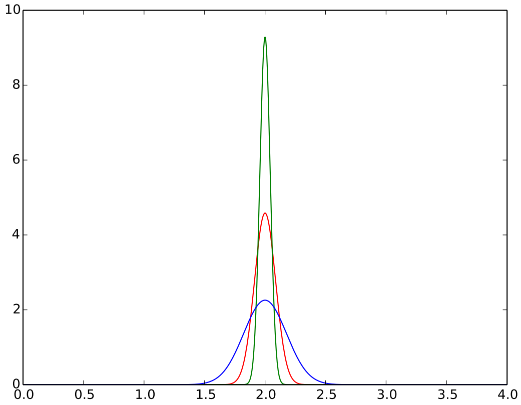


Fig. 15.8: The shifted curves for the three sensors. Sensor A is shown in red, sensor B in green and sensor C in blue.

Using the weighted sum derived above, we can fuse the measurements based on standard deviations.

$$\hat{x} = \frac{\displaystyle\sum_{i=1}^{n} \frac{z_i}{\sigma_i^2}}{\displaystyle\sum_{i=1}^{n} \frac{1}{\sigma_i^2}} = \frac{\dfrac{2.02137}{0.08698^2} + \dfrac{2.04363}{0.04282^2} + \dfrac{2.13049}{0.17674^2}}{\dfrac{1}{0.08698^2} + \dfrac{1}{0.04282^2} + \dfrac{1}{0.17674^2}} = 2.0434.$$

The variance for this measurement is given by $\sigma^2 =$

$$\left(\sum_{i=1}^{n} \frac{1}{\sigma_i^2}\right)^{-1} = \left(\frac{1}{0.08698^2} + \frac{1}{0.04282^2} + \frac{1}{0.17674^2}\right)^{-1} \approx 0.0375404^2$$

Note that the standard deviation is lower than all three of the estimates, meaning the fused measurement is more accurate than any of the measurements alone.

The code to implement the data fusion is given below. We assume we already have three Numpy arrays (the sensor data arrays) filled with the 20 sensor test readings.

```
a_shift = 2.0 - np.mean(sensor_a_data)
b_shift = 2.0 - np.mean(sensor_b_data)
c_shift = 2.0 - np.mean(sensor_c_data)

a_std = np.std(sensor_a_data)
b_std = np.std(sensor_b_data)
c_std = np.std(sensor_c_data)

x = sensor_a + a_shift
y = sensor_b + b_shift
z = sensor_c + c_shift

print "Measurement: "
print '{0:.5f}   {1:.5f}    {2:.5f}'.format(sensor_a, sensor_b, sensor_c)
print "Corrected measurement: "
print '{0:.5f}   {1:.5f}    {2:.5f}'.format(x, y, z)

cdarray = np.array([x, y, z])
sdarray = np.array([a_std, b_std, c_std])
sdarray2 = sdarray*sdarray
top = np.dot(sdarray2,cdarray)
bottom = np.dot(sdarray2,np.ones((3)))
print "Estimate = ", top/bottom
```

Assume you have two sensors, one good one and one that is no accurate at all. Does it really make sense to always merge them? Seems like the better sensor will always produce a more accurate measurement.

Given two sensors, does it always make sense to combine their measurements? Assume that you have two variances: $\sigma_1^2 = 1$, $\sigma_2^2 = 5$. The first sensor is clearly better than the second. The variance formula for the combined measurement is

$$\frac{1}{\sigma^2} = \frac{1}{1} + \frac{1}{5} = 1.2 \quad \Rightarrow \quad \sigma^2 \approx 0.833.$$

The example showed a lower variance on the combined measurement. This is true in general as the next result demonstrates. The fused measurement is more accurate than any individual measurement.

For the weighted averaging process, we have that $\sigma^2 < \sigma_i^2$ for all measurements i.

$$\sigma^2 = \left(\sum_{i=1}^{n} \frac{1}{\sigma_i^2}\right)^{-1} \quad \Rightarrow \quad \frac{1}{\sigma^2} = \sum_{i=1}^{n} \frac{1}{\sigma_i^2}$$

$$\frac{1}{\sigma^2} = \frac{1}{\sigma_k^2} + \sum_{i=1, i\neq k}^{n} \frac{1}{\sigma_i^2} > \frac{1}{\sigma_k^2}$$

$$\frac{1}{\sigma^2} > \frac{1}{\sigma_k^2} \quad \Rightarrow \quad \sigma^2 < \sigma_k^2$$

So there is value in including measurements from lower accuracy sensors.

15.3.4 Recursive Filtering

Say that you have computed an average over a dataset and another value is added to the dataset. Using the previous formula, you need to repeat the summation. However, it is clear that you are repeating much of the work done before. We can rewrite the expression to simply update the formula and build a running average formula. This is the first step to recursive filtering. The average is given by

$$\hat{x}_n = \frac{1}{n}\sum_{i=1}^{n} z_i$$

A new data point provides a new estimate:

$$\hat{x}_{n+1} = \frac{1}{n+1}\sum_{i=1}^{n+1} z_i$$

Pull the last value out of the sum and rework the weight in front of the sum:

$$\begin{aligned}
\hat{x}_{n+1} &= \frac{n}{n+1}\left(\frac{1}{n}\sum_{i=1}^{n} z_i\right) + \frac{1}{n+1} z_{n+1} \\
&= \frac{1}{n+1}\left(n\hat{x}_n + z_{n+1}\right) \\
&= \frac{1}{n+1}\left((n+1)\hat{x}_n + z_{n+1} - \hat{x}_n\right) \\
&= \frac{n+1-1}{n+1}\hat{x}_n + \frac{1}{n+1} z_{n+1} \\
&= \hat{x}_n - \frac{1}{n+1}\hat{x}_n + \frac{1}{n+1} z_{n+1} \\
&= \hat{x}_n + \frac{1}{n+1}\left(z_{n+1} - \hat{x}_n\right).
\end{aligned}$$

Thus we have

$$\hat{x}_{n+1} = \hat{x}_n + K_n\left(z_{n+1} - \hat{x}_n\right), \quad K_n = \frac{1}{n+1}.$$

Take the first column of the data set in Section 15.3.3. Assume that you want to do this as a running average over the N points contained in the file.

```
x = 0
n  = 1

f = open('data2.txt','r')
for line in f:
  item = line.split()
  z = eval(item[0])
  x = x + (z - x)/(n)
  n = n+1

print x
```

Note that you did not need to know how many points were in the file to get the average. It was built into the iteration formula.

This process can be weighted to produce a running weighted average. We will rework the previous derivation for the case where the weighting is not uniform. The running average will be denoted by $\hat{x}_n$ and the running variance will be denoted by P_n

$$\hat{x}_n = P_n \sum_{i=1}^{n} \frac{z_i}{\sigma_i^2}, \qquad P_n = \left(\sum_{i=1}^{n} \frac{1}{\sigma_i^2} \right)^{-1}$$

A new data point provides a new estimate:

$$\hat{x}_{n+1} = P_{n+1} \sum_{i=1}^{n+1} \frac{z_i}{\sigma_i^2}, \qquad P_{n+1} = \left(\sum_{i=1}^{n+1} \frac{1}{\sigma_i^2} \right)^{-1}$$

As with the uniform weighting, pull the last value out of the sum and rework the sum:

$$\begin{aligned}
\hat{x}_{n+1} &= \frac{P_{n+1}}{P_n} \left(P_n \sum_{i=1}^{n} \frac{z_i}{\sigma_i^2} \right) + P_{n+1} \frac{z_{n+1}}{\sigma_{n+1}^2} \\
&= \frac{P_{n+1}}{P_n} \hat{x}_n + P_{n+1} \frac{z_{n+1}}{\sigma_{n+1}^2} \\
&= \frac{P_{n+1}}{P_n} \hat{x}_n + \frac{P_{n+1}\hat{x}_n}{\sigma_{n+1}^2} + P_{n+1} \frac{z_{n+1}}{\sigma_{n+1}^2} - \frac{P_{n+1}\hat{x}_n}{\sigma_{n+1}^2} \\
&= P_{n+1} \left(\hat{x}_n \left(\frac{1}{P_n} + \frac{1}{\sigma_{n+1}^2} \right) + \frac{z_{n+1}}{\sigma_{n+1}^2} - \frac{\hat{x}_n}{\sigma_{n+1}^2} \right)
\end{aligned}$$

Since $1/P_{n+1} = 1/P_n + 1/\sigma_{n+1}^2$

$$\begin{aligned}
&= \hat{x}_n + \frac{P_{n+1} z_{n+1}}{\sigma_{n+1}^2} - \frac{P_{n+1}\hat{x}_n}{\sigma_{n+1}^2} \\
&= \hat{x}_n + K_{n+1} \left(z_{n+1} - \hat{x}_n \right),
\end{aligned}$$

with

$$K_{n+1} = \frac{P_{n+1}}{\sigma_{n+1}^2} = \frac{1}{\sigma_{n+1}^2} \left(1/P_n + 1/\sigma_{n+1}^2 \right)^{-1} = \frac{P_n}{\left(P_n + \sigma_{n+1}^2 \right)}.$$

Using K we can write a recursive formula for P_{n+1}:

$$P_{n+1} = (1 - K_{n+1})P_n$$

This provides us with a recursive weighted filter:

$$\begin{aligned} K_n &= P_{n-1}\left(P_{n-1} + \sigma_n^2\right)^{-1} \\ \hat{x}_n &= \hat{x}_{n-1} + K_n\left(z_n - \hat{x}_{n-1}\right) \\ P_n &= (1 - K_n)P_{n-1}, \end{aligned} \tag{15.3}$$

where $P_0 = \sigma_0^2$ and $\hat{x}_0 = z_0$.

You have now seen two important aspects to the Kalman Filter. The concept of sensor fusion, data from different distributions, and the concept of recursive filtering.

Assume that you get successive measurements from three sensors which are already corrected for deterministic errors. The data is $\{(z,\sigma)\} = \{(1.5, 0.1), (1.3, 0.05), (1.4, 0.15)\}$. Find the recursive fused estimate. For comparison, we first compute using the non-recursive (regular) formula.

$$S = \frac{1.0}{0.1^2} + \frac{1.0}{0.05^2} + \frac{1.0}{0.15^2}, \quad y = \frac{1.5}{0.1^2} + \frac{1.3}{0.05^2} + \frac{1.4}{0.15^2}$$

$$\hat{x} = \frac{y}{S} \approx 1.34489795918$$

The recursive approach is given in the code listing below:

```
z=np.array([1.5,1.3,1.4])
sigma=np.array([0.1,0.05,0.15])
p = sigma[0]**2
xhat = z[0]

for i in range(1,3):
  kal = p/(p + sigma[i]**2)
  xhat = xhat + kal*(z[i] - xhat)
  p = (1-kal)*p

print xhat
```

The result of running the code: 1.34489795918

15.3.5 Multivariate Recursive Filtering

Let W_i is the variance for the sensor. The previous algorithm extends to multiple variables as

- Set $x_0 = z_0$, $P_0 = W_0$
- Let $n = 1$ and repeat:
  - $K_n = P_{n-1}\left(P_{n-1} + W_n\right)^{-1}$
  - $\hat{x}_n = \hat{x}_{n-1} + K_n\left(z_n - \hat{x}_{n-1}\right)$
  - $P_n = (I - K_n)P_{n-1}$

```
while (i<n):
  y = z[i] - x
  S = P + W[i]
  kal = np.dot(P,linalg.inv(S))
  x = x + np.dot(kal,y)
  P = P - np.dot(kal,P)
  i = i+1
```

Sample Data Fusion

Assume that you are given the two measurements $z_1 = (0.9, 2.1, 2.8)$ and $z_2 = (1.1, 2.0, 3.1)$. Also assume the variance-covariance matrices for z_1 and z_2 are

$$W_1 = \begin{pmatrix} 0.2 & 0.02 & 0.002 \\ 0.02 & 0.3 & 0.01 \\ 0.002 & 0.01 & 0.4 \end{pmatrix}, \quad W_2 = \begin{pmatrix} 0.1 & 0.01 & 0.001 \\ 0.01 & 0.16 & 0.008 \\ 0.001 & 0.008 & 0.2 \end{pmatrix}$$

How can you merge these into a single estimate?

```
import numpy as np
from scipy import linalg
z1 = np.array([0.9,2.1,2.8])
z2 = np.array([1.1, 2.0,3.1])
w1 = np.array([[0.2,0.02,0.002],[0.02, 0.3, 0.01],[0.002,0.01,0.4]])
w2 = np.array([[0.1,0.01,0.001],[0.01, 0.16, 0.008],[0.001,0.008,0.2]])
x = z1
P = w1
y = z2 - x
S = P + w2
kal = np.dot(P,linalg.inv(S))
x = x + np.dot(kal,y)
P = P - np.dot(kal,P)
```

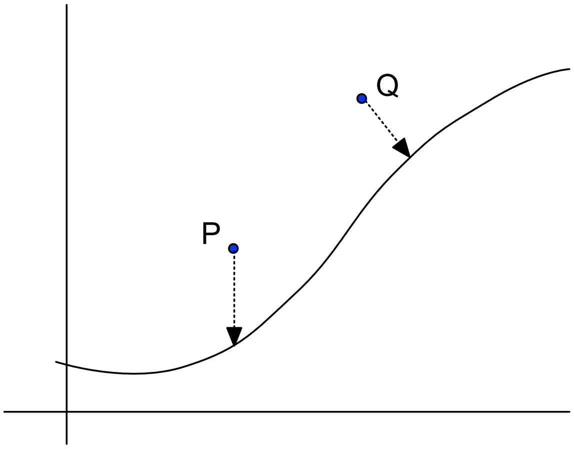
$$x = \begin{pmatrix} 1.03333333 & 2.03420425, & 3.00056428 \end{pmatrix}$$

$$P = \begin{pmatrix} 0.06666667 & 0.00666667 & 0.00066667 \\ 0.00666667 & 0.10434213 & 0.00463772 \\ 0.00066667 & 0.00463772 & 0.13332457 \end{pmatrix}$$

Model based filtering

You have seen two important aspects to the Kalman Filter which we will derive later. The concept of sensor fusion which is merging data from different distributions and the concept of recursive filtering which follows a Markov formulation. Next we look at how projecting data onto a model or fitting to a model can act as a filter.

Assume that you have a model and data points P and Q. We can "filter" by projecting the data onto the model (curve).



Say that you have a data set: $(x_i, y_i), \quad i = 1, \ldots, k$. and you want to fit a model to it (project onto a model):

$$y = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0, \qquad k >> n$$

or in general

$$y = a_n\phi_n(x) + a_{n-1}\phi_{n-1}(x) + \cdots + a_0\phi_0(x)$$

How does one use the data to find the coefficients of the model?

Plug the data into the model:

$$\begin{aligned} y_1 &= a_n x_1^n + a_{n-1}x_1^{n-1} + \cdots + a_1 x_1 + a_0 \\ y_2 &= a_n x_2^n + a_{n-1}x_2^{n-1} + \cdots + a_1 x_2 + a_0 \\ &\vdots \\ y_{k-1} &= a_n x_{k-1}^n + a_{n-1}x_{k-1}^{n-1} + \cdots + a_{k-1}x_{k-1} + a_0 \\ y_k &= a_n x_k^n + a_{n-1}x_k^{n-1} + \cdots + a_1 x_k + a_0 \end{aligned}$$

This can be rewritten in the language of linear algebra:

Plug the data into the model:

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix}}_{y} = \underbrace{\begin{bmatrix} x_1^n & x_1^{n-1} & \ldots & x_1 & 1 \\ x_2^n & x_2^{n-1} & \ldots & x_2 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_k^n & x_k^{n-1} & \ldots & x_k & 1 \end{bmatrix}}_{X} \underbrace{\begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_1 \\ a_0 \end{bmatrix}}_{a}$$

The problem is that this system is not usually square and so one cannot just invert the matrix X to find the coefficients a_j. Expressing our system as

$$y = Xa$$

We assume that we have many data points but wish a low degree polynomial to fit the data points, $k >> n+1$ where k is the number of points and n is the degree of the polynomial. This is an overdetermined problem and presents us with a non-square matrix A. We form the normal equations

$$X^T y = X^T X a$$

we obtain a solvable system. If $X^T X$ is of full rank, then we can invert

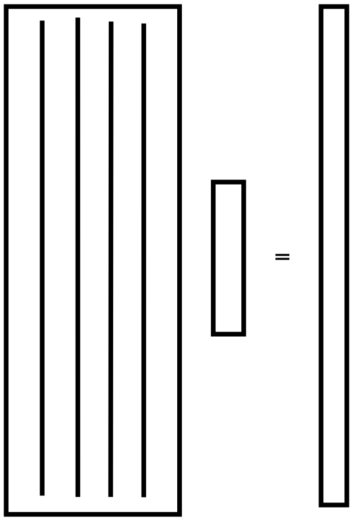
$$a = \left(X^T X\right)^{-1} X^T y$$

Once a is found then we may use

$$\hat{y} = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

as the "fit" to the data.

Before we put this to use, we should address the question "is $X^T X$ of full rank?" What does this mean? Here it means that the columns must be linearly independent. The geometric structure of the system looks like:

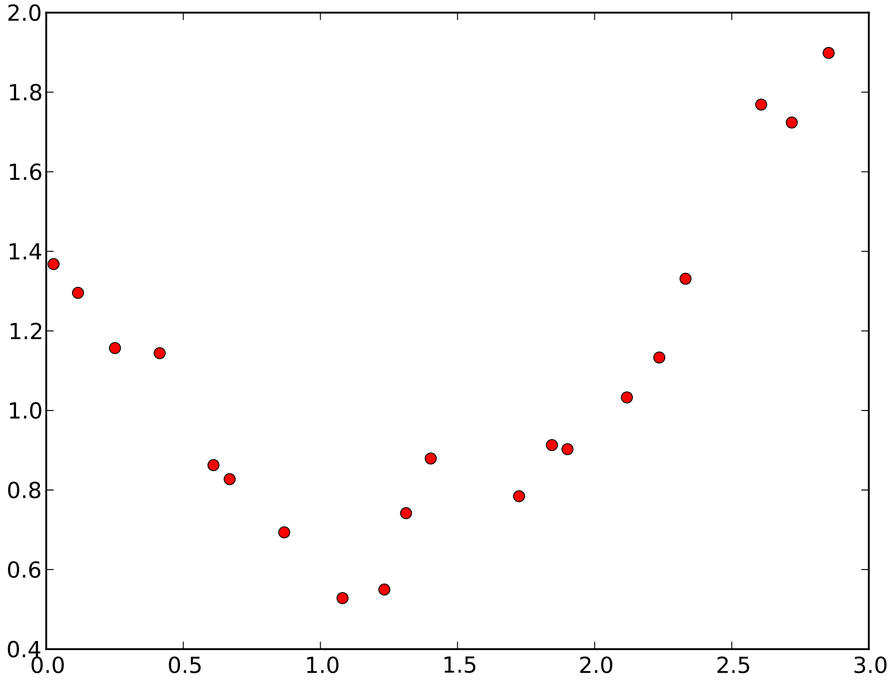


It is clear that if we have more rows than columns, the rows cannot be linearly independent. The columns might be L.I.. If they are not then two of the basis elements $\phi_i(x)$ and $\phi_j(x)$ are the same and we have repeated one.

For this example, we have 20 points for which we would like to fit a quadratic equation. Assume the data is contained in a file named "data.txt" (with the same formatting), we can plot this using:

x_i y_i

```
0.026899  1.367895
0.115905  1.295606
0.250757  1.156797
0.413750  1.144025
0.609919  0.862480
0.669044  0.827181
0.868043  0.693536
1.080695  0.528216
1.233052  0.549789
1.312322  0.741778
1.402371  0.879171
1.724433  0.784356
1.844290  0.912907
1.901078  0.902587
2.117728  1.032718
2.235872  1.133116
2.331574  1.331071
2.607533  1.768845
2.719074  1.723766
2.853608  1.898702
```



Assume that the model for the data is $y = a_2 x^2 + a_1 x + a_0$. Find a_2, a_1, a_0. Note that the system arises:

$$\begin{aligned}
(0.026899, 1.367895) \rightarrow \quad & 1.367895 = a_2(0.026899)^2 + a_1(0.026899) + a_0 \\
(0.115905, 1.295606) \rightarrow \quad & 1.295606 = a_2(0.115905)^2 + a_1(0.115905) + a_0 \\
(0.250757, 1.156797) \rightarrow \quad & 1.156797 = a_2(0.250757)^2 + a_1(0.250757) + a_0 \\
& \vdots
\end{aligned}$$

which can be written as

$$\begin{bmatrix} (0.026899)^2 & 0.026899 & 1 \\ (0.115905)^2 & 0.115905 & 1 \\ (0.250757)^2 & 0.250757 & 1 \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} a_2 \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} 1.367895 \\ 1.295606 \\ 1.156797 \\ \vdots \end{bmatrix}$$

The Normal Equations can be formed

$$\begin{bmatrix} (0.026899)^2 & (0.115905)^2 & (0.250757)^2 & \dots \\ 0.026899 & 0.115905 & 0.250757 & \dots \\ 1 & 1 & 1 & \dots \end{bmatrix} \begin{bmatrix} (0.026899)^2 & 0.026899 & 1 \\ (0.115905)^2 & 0.115905 & 1 \\ (0.250757)^2 & 0.250757 & 1 \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} a_2 \\ a_1 \\ a_0 \end{bmatrix}$$

$$= \begin{bmatrix} (0.026899)^2 & (0.115905)^2 & (0.250757)^2 & \dots \\ 0.026899 & 0.115905 & 0.250757 & \dots \\ 1 & 1 & 1 & \dots \end{bmatrix} \begin{bmatrix} 1.367895 \\ 1.295606 \\ 1.156797 \\ \vdots \end{bmatrix}$$

One can solve $X^T X a = X^T y$: $a = (X^T X)^{-1} X^T y$

$$\begin{bmatrix} 286.78135686 & 122.11468009 & 55.44347326 \\ 122.11468009 & 55.44347326 & 28.317947 \\ 55.44347326 & 28.317947 & 20. \end{bmatrix} \begin{bmatrix} a_2 \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} 72.4241925 \\ 33.380646 \\ 21.534542 \end{bmatrix}$$

$$\begin{bmatrix} a_2 \\ a_1 \\ a_0 \end{bmatrix} \approx \begin{bmatrix} 0.4930957 \\ -1.212858 \\ 1.42706 \end{bmatrix}$$

The curve is approximately $y = 0.49x^2 - 1.21x + 1.42$, Figure Fig. 15.9

```
N = len(xl)
x = np.array(xl)
y = np.array(yl)
xx = x*x
A = np.array([xx, x, np.ones((N))]).T
AT = np.array([xx, x, np.ones((N))])
AA = np.dot(AT,A)
ATy = np.dot(AT,y)

c = linalg.solve(AA,ATy)
```

(continues on next page)

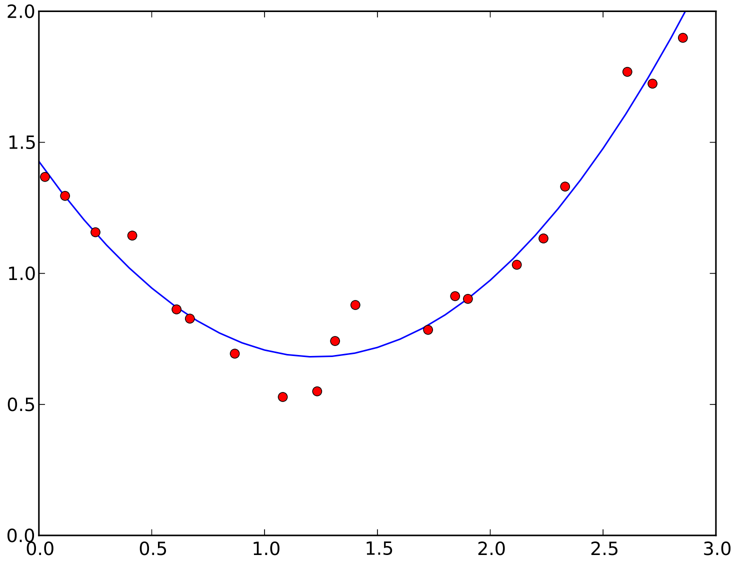


Fig. 15.9: The plot of $y = 0.49x^2 - 1.21x + 1.42$.

(continued from previous page)

```
t = np.arange(0,3, 0.1)
tt = t*t
B = np.array([tt,t,np.ones(len(t))]).T
s = np.dot(B,c)
plt.plot(t,s, 'b-', x,y, 'ro')
plt.xlim(0,3)
plt.ylim(0,2)
plt.show()
```

A couple of figures can help. For the following, we generate a segment of a curve $y = x^2 - 2x + 1$ and add some noise. In Fig. 15.10 the points with the added noise are show in red and the least squares quadratic fit is shown in blue.

Going a bit further, the noise is extracted and shown in yellow. The blue curve is the least squares fit and the green curve is the original polynomial.

15.3.6 Moore-Penrose Pseudo-Inverse

Recall from linear algebra that we have two types of pseudo-inverses. One that acts on the left one that acts on the right. They each address complementary problems in least squares solutions to non-square systems. These are reproduced here for convenience.

1. Left Moore-Penrose Pseudo-Inverse: $H^+ = \left(H^T H\right)^{-1} H^T : H^+ H = I$

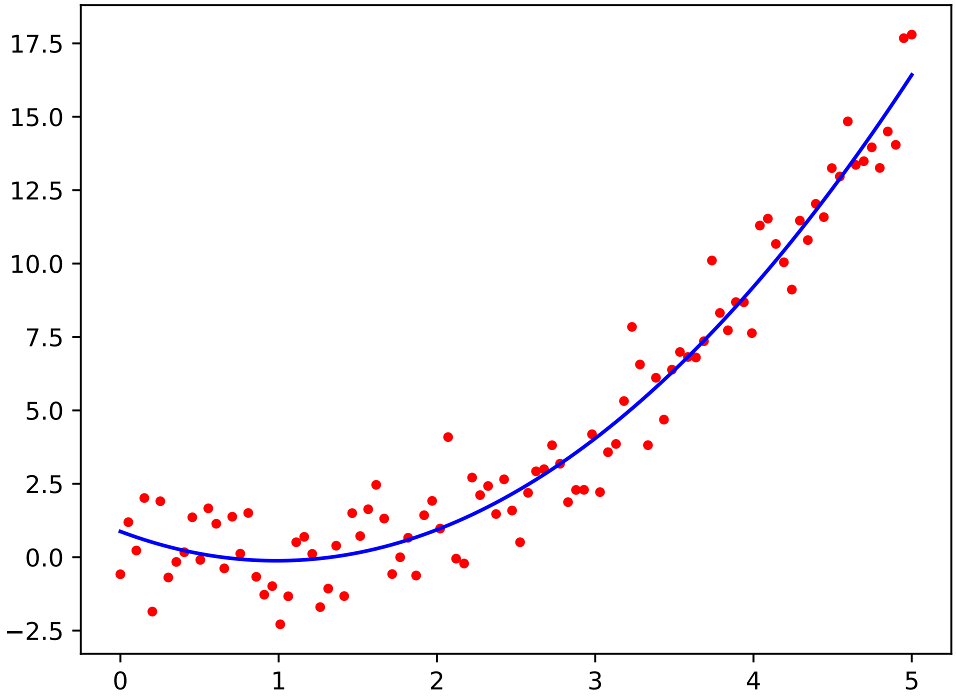


Fig. 15.10: A data set with noise shown in red and the least squares fit is shown in blue.

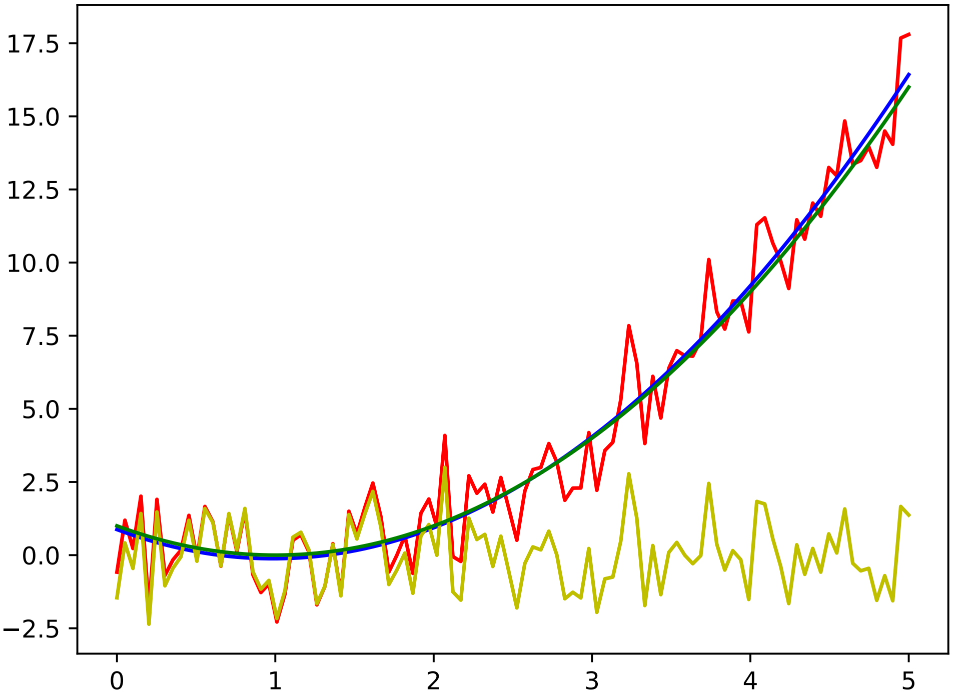
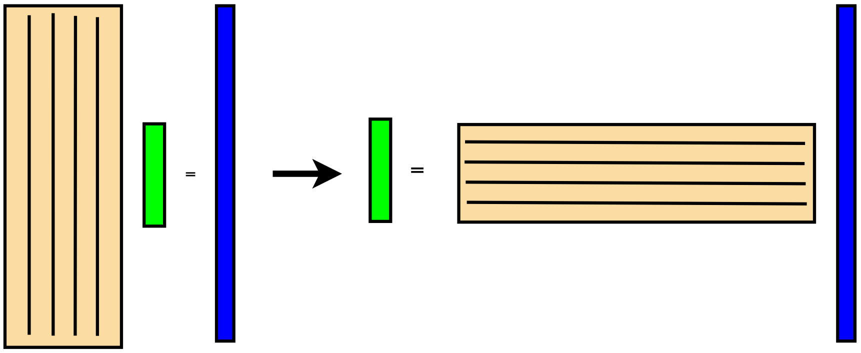
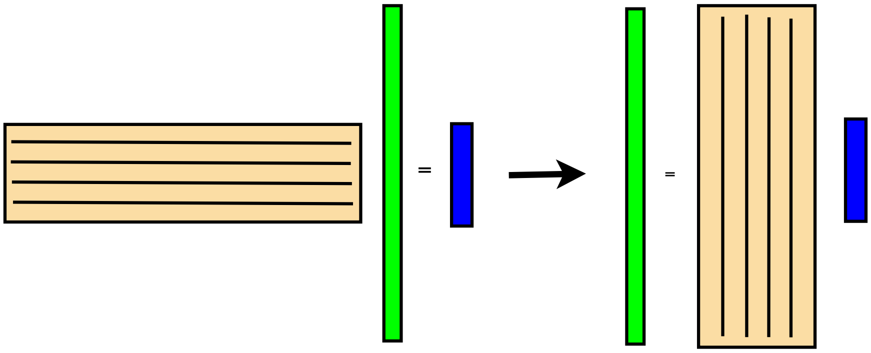


Fig. 15.11: The red curve is the sample or noisy set. The blue curve is the least squares interpolant. Subtracting the interpolant from the original set gives us the noise curve shown in yellow. The original data is shown in green.



2. Right Moore-Penrose Pseudo-Inverse: $H^+ = H^T \left(HH^T\right)^{-1} : HH^+ = I$



15.3.7 Least Squares Observer

Least Squares is used because there is noise in the data collection or the observations. Here we will summarize the material above and use a notation closer to what is used in the Kalman Filter. Let's start with a familiar example. Assume that you have a collection of similar sensors (equal standard deviations for now) that you gather measurements from: $z_1, z_2, \ldots, z_n$. You know that they are noisy versions of a hidden state x, with noise w meaning that $z = Hx + w$, the observation of x subject to noise w. Given k observations z of state $x \in \mathbb{R}^n$, $k >> n$, with noise w:

$$z = Hx + w.$$

As before, we aim to find $\hat{x}$ which minimizes the square error:

$$\|z - H\hat{x}\|.$$

So, we are seeking the least square solution to $z = H\hat{x}$ which is

$$\hat{x} = \left(H^T H\right)^{-1} H^T z.$$

The difference between the estimate and the actual value

$$\hat{x} - x = \left(H^T H\right)^{-1} H^T(Hx + w) - x = \left(H^T H\right)^{-1} H^T w$$

If w has zero mean then $\hat{x} - x$ has zero mean and $\hat{x}$ is an unbiased estimate (as we had before).

Example

In this example we observe just the state variable and without noise we would just have $z = x$. Using this as our model we obtain a set of equations:

$$\begin{gathered} z_1 = x + w_1 \\ z_2 = x + w_2 \\ \vdots \\ z_n = x + w_n. \end{gathered}$$

We have solved this problem earlier, but this time we will rewrite it in a matrix form. Bear with me since it is a lot of machinery for a simple problem, but it will help lead us to the more general case which follows. It can be written as

$$z = Hx + w$$

where

$$z = \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix}, \quad w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}, \quad H = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}.$$

Write out the estimate to see how it compares to the previous one:

$$\begin{aligned} \hat{x} = \left(H^T H\right)^{-1} H^T z &= \left(\begin{bmatrix} 1 & 1 & \dots & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \right)^{-1} \left(\begin{bmatrix} 1 & 1 & \dots & 1 \end{bmatrix} \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix} \right) \\ &= \frac{1}{n} \sum_{i=1}^{n} z_i \end{aligned}$$

which agrees with our earlier work (and below we will show that the weighted one works out as well). The strength of this approach is in the ease of generalization[1].

Weighted Least Squares Observer

Traditional least squares is formulated by minimizing using the normal innerproduct:

$$x^T y = \sum_i x_i y_i.$$

If the inner product is weighted:

$$x^T y = \sum_i x_i y_i q_i = x^T Q y$$

[1] Generalization is not our goal, we have a specific problem to address.

then the weighted least squares solution to

$$z = Hx + w$$
$$is$$

$$\hat{x} = \left(H^TQH\right)^{-1} H^TQz.$$

The matrix Q is any matrix for which the innerproduct above is a valid. However, we will select Q as a diagonal matrix containing the reciprocals of the variances (the reason shown below in the covariance computation). We can rework our simple example:

$$z = \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix}, \quad w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}, \quad H = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

and

$$Q = \begin{bmatrix} \sigma_1^{-2} & 0 & \dots & 0 \\ 0 & \sigma_2^{-2} & \dots & 0 \\ 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \sigma_n^{-2} \end{bmatrix}.$$

The estimate, $\hat{x}$ is then $\hat{x} = \left(H^TQH\right)^{-1} H^TQz$,

$$\hat{x} = \left(\begin{bmatrix} 1 & 1 & \dots & 1 \end{bmatrix} \begin{bmatrix} \sigma_1^{-2} & 0 & \dots & 0 \\ 0 & \sigma_2^{-2} & \dots & 0 \\ 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \sigma_n^{-2} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \right)^{-1}$$

$$\times \left(\begin{bmatrix} 1 & 1 & \dots & 1 \end{bmatrix} \begin{bmatrix} \sigma_1^{-2} & 0 & \dots & 0 \\ 0 & \sigma_2^{-2} & \dots & 0 \\ 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \sigma_n^{-2} \end{bmatrix} \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix} \right),$$

$$\hat{x} = \frac{\displaystyle\sum_{i=1}^{n} \frac{z_i}{\sigma_i^2}}{\displaystyle\sum_{i=1}^{n} \frac{1}{\sigma_i^2}}$$

The covariance of this estimate is

$$= \left(H^TQH\right)^{-1} H^TQ\,W\,QH \left(H^TQH\right)^{-1}$$

Often one selects the weighting to be inversely proportional to W (the matrix of reciprocal variances) which is what we did above:

$$Q = W^{-1}.$$

A smaller standard deviation means better data, and thus we weigh this more. Substituting in

$$\hat{x} = \left(H^T W^{-1} H\right)^{-1} H^T W^{-1} z$$

with covariance

$$P = \left(H^T W^{-1} H\right)^{-1}$$

Given an observation z of state x with noise w:

$$z = Hx + w$$

the $\hat{x}$ which minimizes the square error

$$\|z - H\hat{x}\|$$

$$\hat{x} = H^{+} z = W^{-1} H^T \left(H W^{-1} H^T\right)^{-1} z$$

with W the covariance of w and error covariance

$$P = \left(H W^{-1} H^T\right)^{-1}$$

if we take the same weighting as before.

Example

Assume that we have two state variables x_1 and x_2 and we are able to observe the first directly (with noise) and the sum of the two (with noise). The model will be two constants we are observing through a noisy observation process. This means:

$$z = Hx \quad \Rightarrow \quad \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

Multiple observations give:

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ \vdots \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 0 \\ 1 & 1 \\ \vdots & \vdots \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ \vdots \end{bmatrix}$$

The least square solution to $z = H\hat{x}$ is

$$\hat{x} = \left(H^T H\right)^{-1} H^T z$$

Assume we have data:

```
0.874328560532
3.25683958794
0.859486711669
2.86834487616
1.25271217589
2.95373764186
0.881013871661
3.09066238259
0.971121996741
3.03754386081
```

Compute Normal Equation:

$$H^T H = \begin{bmatrix} 10 & 5 \\ 5 & 5 \end{bmatrix} \qquad H^T z = \begin{bmatrix} 20.04579167 \\ 15.20712835 \end{bmatrix}$$

Solve $H^T H x = H^T z$: Then: $x_1 = 0.96773266, \ \ x_2 = 2.07369301$

Note that the actual values were $x_1 = 1, x_2 = 2$

Example

Recall that there are two forms of the Least Squares Inverse (the Pseudoinverse). The examples above used the left inverse. That applied when we had more equations than unknowns (or variables), the problem was overdetermined. There will be times for which the reverse is true; that we will have more unknowns than equations. For the underdetermined problem we use the right inverse. The following illustrates this idea.

Say that the system can observe two of three variables: (u, v) from (u, v, θ),

$$z_k = Hx_k \quad \Rightarrow \quad \begin{bmatrix} \xi_k \\ \eta_k \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} u_k \\ v_k \\ \theta_k \end{bmatrix}$$

For this problem we solve using the right inverse:

$$x_k = H^+ z_k.$$

The reason can be seen by looking at the object to be inverted in the two pseudo-inverse formulas:

$$H^T H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad HH^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

The left matrix is not invertable. A right pseudo-inverse

$$\begin{bmatrix} u_k \\ v_k \\ \theta_k \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} \xi_k \\ \eta_k \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \xi_k \\ \eta_k \end{bmatrix} = \begin{bmatrix} \xi_k \\ \eta_k \\ 0 \end{bmatrix}$$

Effectively we have produced a projection. This projection restricted our variables to the relevant observational data. It can then be used in sensor fusion applications.

Example 3

Assume that we have a noisy data set (x_i, y_i) which we know lies on a line:

```
[[  0.          -5.65520482]
 [  0.10204082   4.53774258]
 [  0.20408163   3.71191423]
 [  0.30612245   1.44760549]
 [  0.40816327   0.88024529]
 [  0.51020408   4.25592703]
 [  0.6122449    0.81475181]
 [  0.71428571   0.9275501 ]
 [  0.81632653   2.70301802]
 [  0.91836735   5.74002313]
 [  1.02040816   1.27503184]
 [  1.12244898   3.82976944]
 [  1.2244898    2.34108935]
 [  1.32653061   6.44934519]
 [  1.42857143   6.10025845]
 [  1.53061224   2.0450073 ]
 [  1.63265306   8.08201653]
 [  1.73469388   3.79104473]
 [  1.83673469   5.40629739]
 [  1.93877551   4.15556209]
 [  2.04081633   4.49578503]
 [  2.14285714   7.48854739]
 [  2.24489796   5.07750616]
 [  2.34693878   4.29701526]
 [  2.44897959   7.20452521]
 [  2.55102041   6.72492257]
 [  2.65306122   7.56408995]
 [  2.75510204   7.2419468 ]
 [  2.85714286   3.45946936]
 [  2.95918367   3.54635642]
 [  3.06122449   5.54792305]
 [  3.16326531   8.60804178]
 [  3.26530612   5.41562294]
 [  3.36734694  10.3737351 ]
 [  3.46938776   7.89065344]
 [  3.57142857   6.86298534]
 [  3.67346939   7.81332673]
 [  3.7755102    8.55556688]
 [  3.87755102   9.56774192]
 [  3.97959184   8.10000457]
 [  4.08163265   8.98656353]
 [  4.18367347   6.34429316]
 [  4.28571429   4.62596754]
 [  4.3877551    5.46160224]
 [  4.48979592  11.6944026 ]
 [  4.59183673   9.44392528]
 [  4.69387755   8.49333718]
 [  4.79591837  12.5121096 ]
 [  4.89795918   7.59781085]
```

(continues on next page)

(continued from previous page)

```
 [  5.             9.60759719]]
```

If we know the formula for the line we can project onto the line. For this example, we will assume we don't have the formula and are attempting to deduce the line. Meaning the model is that the data has a linear relation, we just lack the parameters. [So we are doing a parametric curve fit.] We use the same approach as with previous datasets. The model is $y = a_1x + a_0$. Application of the data set and we have an overconstrained system of equations. Using the left pseudoinverse as before we can determine a_1, a_0. We may get something like $a_1 = 2.2231$, $a_0 = 1.0124$, see Fig. 15.12 for data and plot. How would this be a filter? You can project points onto the line via the line projection formula found in calculus: $a = a_1$, $b = -1.0$, $c = a_0$,

$$\begin{aligned} d &= a^2 + b^2 \\ px &= \frac{b(bx - ay) - ac}{d} \\ py &= \frac{a(-bx + ay) - bc}{d} \end{aligned}$$

The application of this as a filter is shown in Fig. 15.13.

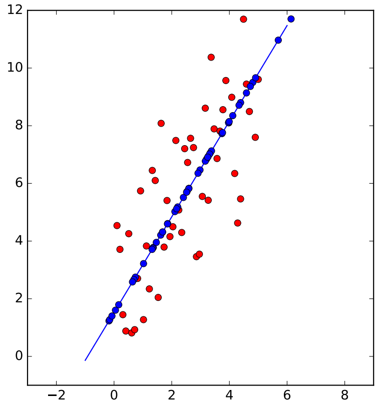


Fig. 15.12: Dataset and least square fit. The data is in red, the curve fit is the solid blue line and the projection of the data is the blue dots.

15.4 Problems

1. Let $z_1 = 284$, $z_2 = 257$, $z_3 = 295$, be measurements from sensors which have normally distributed errors with standard deviations $\sigma_1 = 10$, $\sigma_2 = 20$, $\sigma_3 = 15$, respectively. What is the best estimate for the measured state?
2. You are given three distance sensors which all measure the same distance. To determine the accuracy of each you run repeated measurements of an object which you setup so that the sensor will return 2 meters. Running 40 measurements you obtain the data listed below. Next you measure the distance of an object ahead of the robot. Sensor A gives 2.4577696, sensor B gives 1.8967743, and sensor C

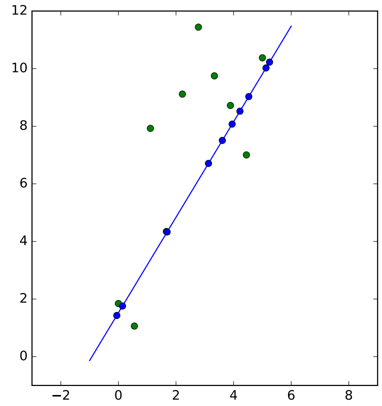


Fig. 15.13: Projecting data onto the line as a filter. Green dots are new data, the curve fit is the solid blue line and blue dots are their projections.

gives 2.1352561. Combine these readings to make a more accurate estimate of the object distance. Hint: you need to figure out the distributions and the data according to the variances.

```
2.2411549   1.8286673    2.3295015
2.3366108   1.9243295    2.4867167
1.9687234   1.8972737    2.5489412
2.1240351   2.0961534    1.9834876
2.3984044   1.7985819    2.6153805
2.3523899   1.8377782    1.8132444
2.1074266   1.8358201    2.5563951
2.4711542   1.8875839    2.2031291
2.1998000   1.8116113    2.3117542
2.2710086   1.8701890    2.3495262
2.3530473   1.7646824    2.0109293
2.4391559   1.9499153    2.5030771
2.2066306   1.9243432    2.3561112
2.3000099   1.8309696    2.3097754
2.2235766   1.8453219    2.2940692
2.1396901   1.8390955    2.1904604
2.0929719   1.7978329    2.5693897
2.3154159   1.8217245    1.9332188
2.3716302   1.9558670    2.3002433
2.2611420   1.8654487    2.5508342
2.1415088   1.7836290    2.6884786
2.2088487   1.9245743    2.5037028
2.2714614   1.8918415    2.7112663
2.3345816   1.8275421    2.1656644
2.3052296   1.8494488    2.1940472
2.1600041   1.7632971    2.2703708
2.0630943   1.8396972    2.6488544
2.0997821   1.8412331    2.1828831
2.3037175   1.7761007    2.2959535
2.4536524   1.8542271    2.0446945
```

(continues on next page)

(continued from previous page)

```
2.3909478   1.8649815    2.7852822
2.1195966   1.9533324    2.5700007
2.0205112   1.8857815    2.1113650
2.1708006   1.7115595    2.1215336
2.0800748   1.9403332    2.3126032
2.3332722   1.8530670    2.4687277
2.0826115   1.8279041    2.6104026
2.2652480   1.9058054    2.3165716
2.3734464   1.9632258    2.0907554
2.0563260   1.9367908    2.2130578
```

3. Perform a numerical study on the three sensor problem. The claim is that if you fuse the three sensor measurements using the formula derived in the text, the fused value (the estimate) is a better estimate than that any of the measurement values even if one measurement comes from a sensor with a very low error (small standard deviation). Generate sample values from three distributions with the same mean (select mean = 5) but very different sigmas (sigma1 = 0.05, sigma2 = 0.25, sigma3 = 0.5). Run 1000 experiments and compute the percentage for which this is true.

4. Let $x = (x_1, x_2)$, $y = (y_1, y_2)$ and define $d(x, y) = \|x - y\|_P$ where $\|x\|_P^2 = x^T P x$ for

   $$P = \begin{pmatrix} 3 & 0.1 \\ 0.1 & 1 \end{pmatrix}.$$

   Find the closest point on the line $x_2 = 10 - 5x_1$ to the origin with respect to $d(x, y)$.

5. Model determination

   1. Assume that you run an experiment on a single step. Starting from $(x_0, y_0, z_0) = (1, 1, 0.5)$, you obtain x, y, w:

      ```
      2.608832, 5.055857, 6.189379
      2.925827, 5.256055, 6.377555
      2.741887, 5.012025, 6.225253
      2.808115, 5.277323, 6.412870
      2.604396, 4.942732, 6.143021
      2.715381, 5.048058, 6.151169
      2.785934, 5.153957, 6.457948
      2.731107, 5.157646, 6.312867
      2.741480, 5.052214, 6.327102
      2.738335, 5.172248, 6.372636
      2.790870, 5.152972, 6.270782
      2.690942, 4.867113, 6.448155
      2.788157, 4.831810, 6.151857
      3.005297, 5.476095, 6.538915
      2.778656, 5.085782, 6.314246
      2.759511, 5.271102, 6.469469
      2.633871, 4.915128, 6.243359
      2.845448, 5.256687, 6.464442
      2.736627, 5.146030, 6.300301
      2.767497, 5.250046, 6.464192
      2.860662, 4.980395, 6.294793
      2.878436, 5.082964, 6.374364
      ```

(continues on next page)

(continued from previous page)

```
2.825564, 5.114201, 6.288422
2.818848, 4.974110, 6.158882
2.844205, 5.102877, 6.354154
```

This data is repeated experiments and NOT iteration data. The means that each row is generated by starting from the initial condition and taking on step of your machine. Determine the parameters, a, b, c, and covariance V for the kinematic model with zero mean Gaussian noise based on dynamics:

$$x_k = x_{k-1} + (x_{k-1}^2 + y_{k-1}^2)\cos(w_{k-1}) + a,$$

$$y_k = y_{k-1} + (x_{k-1}^2 + y_{k-1}^2)\sin(w_{k-1}) + b,$$

$$w_k = w_{k-1} + (x_{k-1}^2 + y_{k-1}^2 + w_{k-1}^2)^{1/2} + c.$$

Approach this by computing the mean of each column. Using the means you can estimate a,b,c. Then using the covariance estimation given in the notes, you can find the covariance matrix.

2. Assume that you have zero mean Gaussian data. Find a standard deviation that produces data where you observe that 20% of the time you have three correct digits (meaning three zeros). This is not unique. Can you also find a sigma that gives you the previous observation but also 80% of the time you see two correct (or zero) digits. Can you write an observational model for this?

CHAPTER

SIXTEEN

ADVANCED FILTERING TECHNIQUES

Observation is not the whole story. The robot will be out observing the world, even though the observations often include random noise, you normally have additional information that allows you to remove significant noise. For example, if we are observing motion, we know that not all types of motion will be possible. The basic laws of physics clearly play a role. In addition the design and details of the robot also play a role. These constrain the possibilities and in doing so, help improve our estimates of the robot state.

16.1 Models, Dynamical Systems and Filters

In this section we learn how to model the dynamics of the system. For our purposes, it is sufficient to use a simplified model of the dynamics. We will assume that the system dynamics are linear and the variations in the dynamics (the noise) is normal or Gaussian. We will immediately see that this assumptions leaves out the most basic of our robot models, the differential drive. Our resolution will be to develop the theory with the linear Gaussian systems and then develop ways to adapt it to real systems.

16.1.1 Creating Filters from data and models

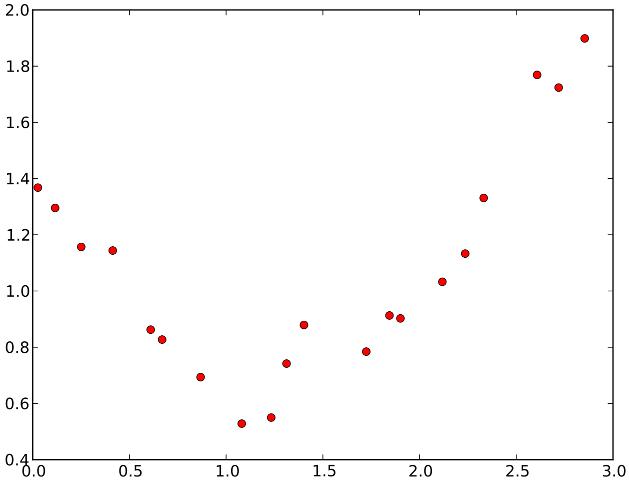
Say that you have a lidar at the side a road or railway which you intend to track velocity of the passing vehicle.[1] The lidar will estimate the distance at the various sweep angles. This can be converted from the polar coordinates of the lidar to the rectangular coordinates natural for tracking the vehicle. Because of measurement error in (r, θ), you have a distribution of possible values in (x, y) of the actual vehicle state (location). If you did not have any additional information, you would have to report the measurement and the standard deviations (assuming you believe the error is normally distributed), and be done.

However, in this case you do have some additional information. Focusing on the train, you know the vehicle is restricted to the rail. This information can be used to reduce or filter out some of the error. The idea then would be to project the estimate of location onto the track since you know (well actually assume) the restriction of the vehicle. The rail here is our proxy for kinematic constraints. The kinematic constraints describes the restrictions on motion just like the rail restricts the train motion. It makes sense then to use the kinematics to help filter out the noise in our measurements.

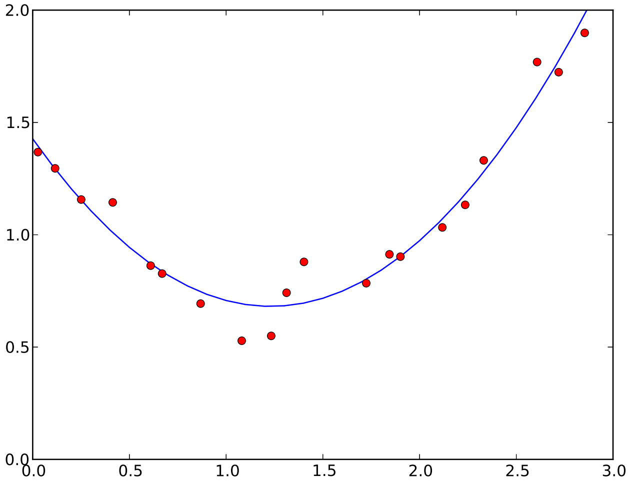
[1] Ignoring for the moment that this is a ridiculously expensive way to track velocity.

Measurement → Model → Estimate/Predict

The work we did earlier in fitting curves is one form of this idea. We have a model in mind and we then "project" the data onto the model.



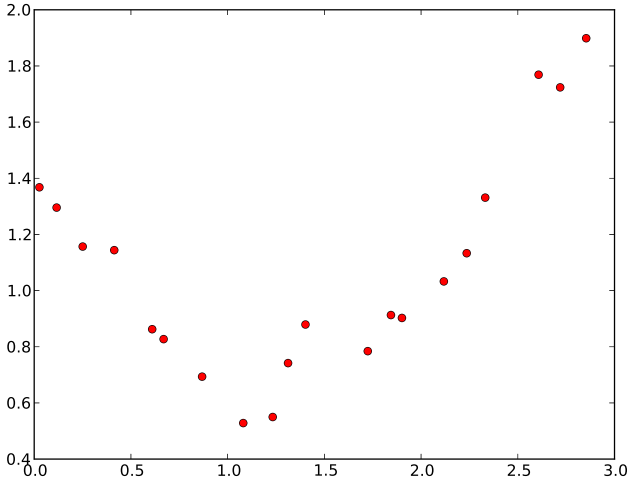
⇒

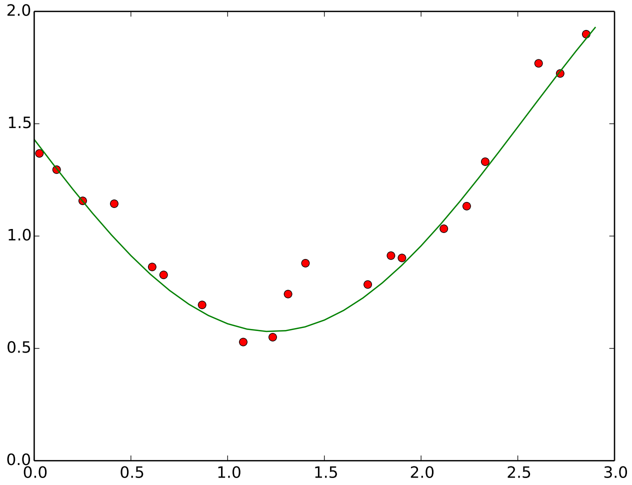


Using quadratic model, $y = c_2x^2 + c_1x + c_0$, and least squares we find $y = 0.49x^2 - 1.21x + 1.42$. So we can use this to extrapolate at $x = 5$ we have $y = 6.5$.

⇒

Note that this is very different from using a sinusoidal model and gaining the fit $y = -0.95\sin(1.2x + 0.1) + 1.525$, where for $x = 5$ we have $y = 1.698$


2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.0
0.5
1.0
1.5
2.0
2.5
3.0


2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

A Physics Example

In this example, we go one step further. Using observational data of motion and knowing the motion is restricted in some manner, can we then use the data + model to predict a missing piece of information. Formally this is known as interpolation or extrapolation depending on the dataset. We use the data plus the model to answer the question. Physical systems have equations describing the behavior of the objects in the system. For example one may know about the forces acting on an object of mass m, F_x and F_y which will give the equations of motion

$$\ddot{x} = \frac{F_x}{m} \qquad \ddot{y} = \frac{F_y}{m}.$$

For this example, assume that these forces are constant and so we may easily integrate them

$$x(t) = \frac{F_x}{2m}t^2 + v_{x,0}t + x_0 \quad \text{and} \quad y(t) = \frac{F_y}{2m}t^2 + v_{y,0}t + y_0.$$

These equations restrict the possible values of x and y while still being general enough to allow for a variety of starting conditions. Assume for the moment that we do not have knowledge of the initial data for a particular application (but understand the forces involved) and would like to use some observed data to determine those values. The observations of the moving object are probably very noisy. Thus you would obtain (t_i, x_i, y_i) data $(i = 1 \ldots k)$. Each data item should satisfy the equations

$$x_i = \frac{F_x}{2m}t_i^2 + v_{x,0}t_i + x_0 \quad \text{and} \quad y_i = \frac{F_y}{2m}t_i^2 + v_{y,0}t_i + y_i, \quad i = 1 \ldots k.$$

There are two unknowns for each equation. Two data points would allow an exact answer (two equations and two unknowns). But what if you had a dozen observations? With noisy data, getting many observations should provide a better estimate of the initial values than just picking two observed values. So, how do we do this?

We assume that we have obtained some data on the motion of an object and wish to compute its equations of motion. Plugging the data in gives the equations:

$$x_i = \frac{F_x}{2m}t_i^2 + v_{x,0}t_i + x_0 \quad \text{and} \quad y_i = \frac{F_y}{2m}t_i^2 + v_{y,0}t_i + y_i, \quad i = 1 \ldots k.$$

Rewrite the expression as

$$\xi_i = x_i - \frac{F_x}{2m}t_i^2 = v_{x,0}t_i + x_0 \quad \text{and} \quad \eta_i = y_i - \frac{F_y}{2m}t_i^2 = v_{y,0}t_i + y_i, \quad i = 1 \ldots k.$$

Then

$$\begin{pmatrix} \xi_1 \\ \xi_2 \\ \ldots \\ \xi_k \end{pmatrix} = \begin{pmatrix} t_1 & 1 \\ t_2 & 1 \\ \vdots & \vdots \\ t_k & 1 \end{pmatrix} \begin{pmatrix} v_{x,0} \\ x_0 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} \eta_1 \\ \eta_2 \\ \ldots \\ \eta_k \end{pmatrix} = \begin{pmatrix} t_1 & 1 \\ t_2 & 1 \\ \vdots & \vdots \\ t_k & 1 \end{pmatrix} \begin{pmatrix} v_{y,0} \\ y_0 \end{pmatrix}$$

The pseudo-inverse is $M = (X^T X)^{-1} X^T$

$$= \left[\begin{pmatrix} t_1 & t_2 & \ldots & t_k \\ 1 & 1 & \ldots & 1 \end{pmatrix} \begin{pmatrix} t_1 & 1 \\ t_2 & 1 \\ \vdots & \vdots \\ t_k & 1 \end{pmatrix} \right]^{-1} \begin{pmatrix} t_1 & t_2 & \ldots & t_k \\ 1 & 1 & \ldots & 1 \end{pmatrix}$$

which gives the least squares estimate

$$\begin{pmatrix} v_{x,0} \\ x_0 \end{pmatrix} = M \begin{pmatrix} \xi_1 \\ \xi_2 \\ \dots \\ \xi_k \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} v_{y,0} \\ y_0 \end{pmatrix} = M \begin{pmatrix} \eta_1 \\ \eta_2 \\ \dots \\ \eta_k \end{pmatrix}$$

Physics example with numbers

Assume that we had the following data: $(t, x, y) = (1, 10, 22), (2, 19, 60), (3, 32, 51)$ and that $F_x = 0$ and $F_y = -2$, $m = 0.25$. So first we gain: $t = [1, 2, 3]$, $\xi = [10, 19, 32]$, $\eta = [26, 76, 87]$. We first compute

$$\left[\begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{pmatrix}\right]^{-1} = \left[\begin{pmatrix} 14 & 6 \\ 6 & 3 \end{pmatrix}\right]^{-1} = \frac{1}{6}\begin{pmatrix} 3 & -6 \\ -6 & 14 \end{pmatrix}$$

$$= \begin{pmatrix} 1/2 & -1 \\ -1 & 7/3 \end{pmatrix}$$

$$\begin{pmatrix} v_{x,0} \\ x_0 \end{pmatrix} = \begin{pmatrix} 1/2 & -1 \\ -1 & 7/3 \end{pmatrix}\begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \end{pmatrix}\begin{pmatrix} 10 \\ 19 \\ 32 \end{pmatrix} = \begin{pmatrix} 11 \\ -1.666667 \end{pmatrix}$$

and

$$\begin{pmatrix} v_{y,0} \\ y_0 \end{pmatrix} = \begin{pmatrix} 1/2 & -1 \\ -1 & 7/3 \end{pmatrix}\begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \end{pmatrix}\begin{pmatrix} 26 \\ 76 \\ 87 \end{pmatrix} = \begin{pmatrix} 30.5 \\ 2.0 \end{pmatrix}$$

So we have that the start location is $(-1.666667, 2.0)$ with initial velocity of $(11, 30.5)$.

16.1.2 Linear Dynamical System

An operator, L, is said to be linear if for scalars a, b and vectors x, y we have

$$L(ax + by) = aLx + bLy$$

A dynamical system

$$x_k = Lx_{k-1} \quad \text{(discrete)}$$

or

$$\dot{x} = Lx \quad \text{(continuous)}$$

is said to be linear if L is a linear operator. Linearity means we may construct solutions using simple addition.

Example of linear operators

Some examples of linear operators are matrices, $A(\alpha x + \beta y) = \alpha Ax + \beta Ay$, and derivatives, $(d/dx)[\alpha u + \beta v] = \alpha du/dx + \beta dv/dx$.

Example: Nonlinear Kinematic Models

The dynamics of a differential drive robot is **NOT** linear:

$$\dot{x} = \frac{r}{2}(\dot{\phi_1} + \dot{\phi_2})\cos(\theta)$$

$$\dot{y} = \frac{r}{2}(\dot{\phi_1} + \dot{\phi_2})\sin(\theta)$$

$$\dot{\theta} = \frac{r}{2L}(\dot{\phi_1} - \dot{\phi_2}).$$

This follows from noting that $\cos(\theta)$ and $\sin(\theta)$ are nonlinear functions of the state variable θ.

16.1.3 Dynamics with Noise

Let x_k be the current state and z_k be the observation. We study the linear system with noise:

$$\begin{aligned} x_k &= Fx_{k-1} + Gu_k + v_k \\ z_k &= Hx_k + w_k \end{aligned}$$

where v_k, w_k are assumed to be zero mean Gaussian noise with covariance matrices V_k and W_k respectively.

We are interested in tracking not just the estimate of the state, but the state's distribution as well since the addition of noise produces random variations in values. The simplest distribution to track is a Normal or Gaussian distribution. Using the Bayes Filter terminology, we have three elements:

1. The state transition probability $p(x_k|u_k, x_{k-1})$ must arise from

   $$x_k = Fx_{k-1} + Gu_k + v_k$$

   where x_k, x_{k-1} are state vectors, u_k controls, v_k is the noise, F and G are matrices. v_k is a mean zero normally distributed random variable with covariance matrix V_k. This is linear system dynamics. Thus the mean of the posterior state is

   $$E(x_k) = Fx_{k-1} + Gu_k,$$

$p(x_k|u_k, x_{k-1})$

$$= \frac{1}{\sqrt{\det(2\pi V_k)}} e^{-\frac{1}{2}(x_k - Fx_{k-1} - Gu_k)^T V^{-1}(x_k - Fx_{k-1} - Gu_k)}.$$

1. The measurement probability $p(z_k|x_k)$ must also be linear

   $$z_k = Hx_k + w_k$$

   where H is a $m \times n$ matrix and w_k is Gaussian mean zero random variable (noise) with covariance matrix W_k. The mean of the observation

   $$E(z_k) = Hx_k,$$

   $$p(z_k|x_k) = \frac{1}{\sqrt{\det(2\pi W_k)}} e^{-\frac{1}{2}(z_k - Hx_k)^T W^{-1}(z_k - Hx_k)}$$

2. Initial belief, bel(x_0) must be normally distributed, say with mean $\hat{x}_0$ and covariance P_0

   $$\text{bel}(x_0) = \frac{1}{\sqrt{\det(2\pi P_0)}} e^{-\frac{1}{2}(x_0 - \hat{x}_0)^T P_0^{-1}(x_0 - \hat{x}_0)}$$

   If assumptions 1,2,3 hold then bel(x_k) is also a Gaussian distribution.

Terminology

We will introduce some fairly common notation used in state estimation. As stated before, we cannot observe the actual value of the quantity x, and so we will indicate with a "hat" the estimate of the value, $\hat{x}$.

- Let $\hat{x}_{k-1|k-1}$ be the current state estimate at time step $k-1$.
- Let $\hat{x}_{k|k-1}$ be the prediction of the next state using a model of the dynamics.
- Let $P_{k|k-1}$ be the covariance of $\hat{x}_{k|k-1}$ ($E[(x_k - \hat{x}_{k|k-1})(x_k - \hat{x}_{k|k-1})^T]$)
- Let z_k be the observation or measurement of x_k.
- Let $\hat{x}_{k|k}$ be the update based on the observation. $\hat{x}_{k|k}$ is our best estimate of x_k
- Let $P_{k|k}$ be the covariance of $\hat{x}_{k|k}$ ($E[(x_k - \hat{x}_{k|k})(x_k - \hat{x}_{k|k})^T]$)

At the risk of being redundant, we need to address a common misunderstanding. The state vector x is not something that normally can be observed. We would not need to do any type of filtering if we could observe it. The observation of x is z. It differs from x in two primary manners. First there is noise in the observation. Meaning that x and z differ by the added noise. Second, we don't observe all of the components of x. Some are missing. This means that the lengths of the vectors for x and z are often different.

For the algorithms in the next couple of sections, we will be estimating x by using z. So z is an input variable. To develop code, we need to test on actual data sets, so we will need to create some fake z to run our tests. The creation of the z data is not part of any of the filters. This is no different than when you create unit tests. They are essential to the development process, but not part of the primary codebase.

16.1.4 Scalar Kalman Filter

For the moment assume that x, F, G, u are scalars. Also assume we have a starting value for the state x_0 and some estimate of the error in that starting value, σ_0^2. The error in the process is measured and has variance σ_v^2, meaning v_k is drawn from a zero mean Gaussian distribution with variance σ_v^2 which gives us the process:

$$x_k = Fx_{k-1} + Gu_k + v_k.$$

The estimate of state based on the process is simply

$$\tilde{x}_k = F\hat{x}_{k-1} + Gu_k.$$

Prior to the process, the variance estimate for x_{k-1} is σ_{k-1}^2. What happens? It is transformed via

$$\tilde{\sigma}_k^2 = (F\sigma_{k-1})^2 + \sigma_v^2 = F^2\sigma_{k-1}^2 + \sigma_v^2.$$

The next thing required is to merge the process prediction with the observation data, z_k (scalar), this observation has quality σ_w^2. These are fused using (15.3) into

$$S_k = \frac{1}{\tilde{\sigma}_k^2} + \frac{1}{\sigma_w^2}$$

$$\begin{aligned} K_k = \left[S_k\sigma_w^2\right]^{-1} = \left[\sigma_w^2\left(\frac{1}{\tilde{\sigma}_k^2} + \frac{1}{\sigma_w^2}\right)\right]^{-1} &= \left[\sigma_w^2\left(\frac{\tilde{\sigma}_k^2 + \sigma_w^2}{\tilde{\sigma}_k^2\sigma_w^2}\right)\right]^{-1} \\ &= \frac{\tilde{\sigma}_k^2}{\tilde{\sigma}_k^2 + \sigma_w^2} \end{aligned}$$

$$\hat{x}_k = \tilde{x}_{k-1} + K_k\left(z_k - \tilde{x}_{k-1}\right)$$

$$\sigma_k^2 = (1 - K_k)\tilde{\sigma}_k^2$$

We can summarize the process

$$\begin{aligned} x_k &= Fx_{k-1} + Gu_k + v_k \\ z_k &= x_k + w_k \end{aligned}$$

in the standard notation of the Kalman Filter. Let the process noise v_k have variance $V = \sigma_v^2$ and the observation noise w_k have variance $W = \sigma_w^2$. We track the estimate (or mean) $\hat{x}_{k|k}$ and the variance $p_{k|k}$. We will also make the following substitutions: $P_{k-1|k-1} = \sigma_{k-1}^2$, $P_{k|k-1} = \tilde{\sigma}_k^2$ and $P_{k|k} = \sigma_k^2$.

The Scalar Kalman Filter Algorithm

In the case where the state vector to be estimated is a scalar, the derivation is much easier and sets the stage for the multivariate version shown in the next section.

- Predicted state: $\hat{x}_{k|k-1} = F\hat{x}_{k-1|k-1} + Gu_k$
- Predicted estimate error: $P_{k|k-1} = F^2P_{k-1|k-1} + V$
- Optimal Kalman gain: $K_k = P_{k|k-1}/(P_{k|k-1} + W)$
- Updated state estimate $\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(z_k - \hat{x}_{k|k-1})$
- Updated estimate variance: $P_{k|k} = (1 - K_k)P_{k|k-1}$

Example

Assume that you are given a simple scalar process on $0 \leq k < N$:

$$x_k = x_{k-1} + u_k$$

where the control input is

$$u_k = 0.5 * (1 - 1.75k/N).$$

Also assume that you have process noise with standard deviation of 0.2 and observation noise with standard deviation of 0.75.

When we don't have actual experimental data, we need to simulate the data. To illustrate the filter, we will create a noisy dataset; we pretend to run the dynamical system and get the observations. Later on in the multivariate content, much greater detail is given to the creation of noisy data. For now, focus on the filter aspect and not on the creation of z.

```
N = 100
mu1, sigma1 = 0.0, 0.2
mu2, sigma2 = 0.0, 0.75
process_noise = np.random.normal(mu1,sigma1, N)
observation_noise = np.random.normal(mu2,sigma2, N)
x_sim = np.zeros(N)
z_sim = np.zeros(N)
u = np.arange(N)
k = 1
while (k<N):
  x_sim[k] = x_sim[k-1] + 0.5*(N-1.75*u[k])/N + process_noise[k-1]
  z_sim[k] = x_sim[k] + observation_noise[k-1]
  k = k+1
```

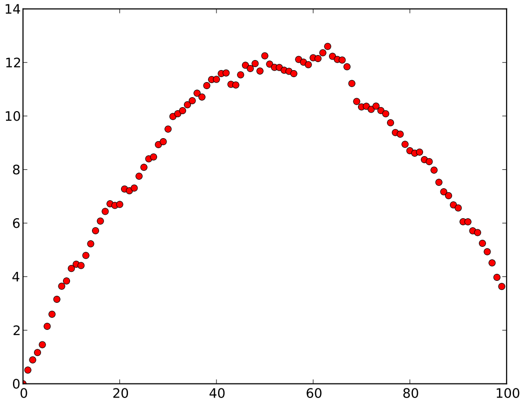


Fig. 16.1: Plot of x_0.

Using the fake observations, we can test the filter.

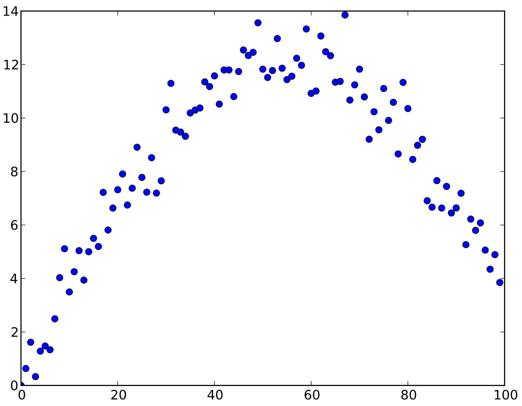


Fig. 16.2: Noisy observation of x_0.

```
x_filtered = np.zeros(N)
covariance_filtered = np.zeros(N)
k = 1
while (k<N):
  x_process_update = x_filtered[k-1] + 0.5*(N-1.75*u[k])/N
  variance_update = pf[k-1] + sigma1*sigma1
  kal_gain = variance_update/(variance_update + sigma2*sigma2)
  x_filtered[k] = x_process_update + kal_gain*(z_sim[k-1] - x_process_update)
  covariance_filtered[k] = (1-kal_gain)*variance_update
  k = k+1
```

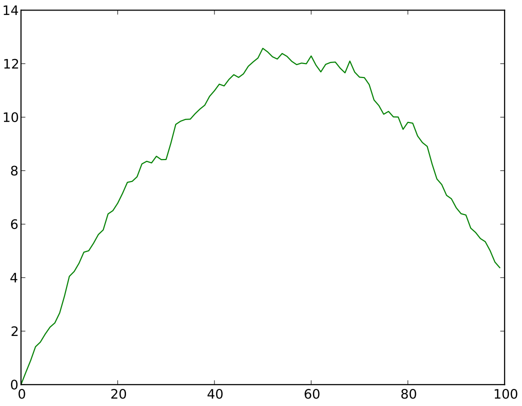


Fig. 16.3: Kalman estimate of x_0.

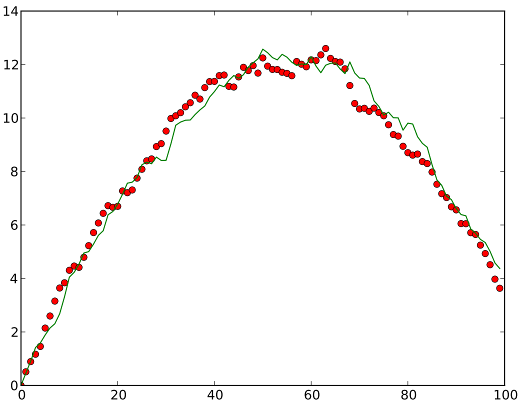


Fig. 16.4: Comparison of state estimate to real state.

16.2 The Kalman Filter

In this section we develop the multivariate version of the Kalman Filter. The steps are basically the same as the scalar version described in the last section, but the derivation is more involved.

16.2.1 The Multivariate Version

The Kalman Filter has two stages. A predictive step based on the system dynamics and an update based on observations or measurements.

The full Kalman Filter has the following objects to track:

- *Prediction*: $\hat{x}_{k|k-1}$, $P_{k|k-1}$
- *Update*: $\hat{x}_{k|k}$, $P_{k|k}$
- $P_{k|k} = \text{cov}(x_k - \hat{x}_{k|k})$
- $P_{k|k-1} = \text{cov}(x_k - \hat{x}_{k|k-1})$
- $S_k = \text{cov}(z_k - H\hat{x}_{k|k-1})$

The prediction step uses the system dynamics, the linear dynamical model, to predict where the system should be. This prediction is for both the state estimate $\hat{x}$ and the covariance of $\hat{x}$. This stage is also known as the *a priori* since it occurs before the observation.

The update step takes the observation at that step and compares it to the prediction. The difference between the two is known as the innovation. It is what is new compared to the system dynamics. Using a weighted least squares approach, the two are merged. This is done by determining how reliable the new information is based on the innovation covariance. The weight term is known as the Kalman gain. The weighted innovation is added to the prediction of the state estimate to obtain the Kalman estimate. As before, this stage is also known as the *a posteriori* because it occurs after the observation. Repeated steps or iterations of the Kalman filter allow the filter to track sequential stages of a process. These sequential steps make this a recursive linear gaussian state estimator.

Formally we have a dynamical process

$$x_{k+1} = F_k x_k + G u_k + v_k \tag{16.1}$$

where F_k is the state transition matrix, Gu_k is the input control and and observation

$$z_k = H x_k + w_k \tag{16.2}$$

where H is the observation matrix. The random variables v_k, w_k are drawn from Gaussian distributions with covariance models given by

$$V = E[v_k v_k^T], \qquad W = E[w_k w_k^T].$$

The error covariance of the estimate is

$$P_k = E[e_k e_k^T] = E[(x_k - \hat{x}_k)(x_k - \hat{x}_k)^T]. \tag{16.3}$$

The state estimate will be denoted $\hat{x}_k$ and the process update to the state is denoted $\tilde{x}_k$

Before we go into the details on the filter design, a couple of comments about the matrices given in the dynamical process.

> The matrix F is given by the model of the physical process. It is a square matrix with dimension $n \times n$ where n is the number of state variables (the length of x). When you are given a continuous dynamical system, make sure you first discretize the problem. Only then can you extract the correct matrix F.
>
> The matrix G is more of a placeholder for now. We assume that we have some type of control input Gu_k but for our discussion you don't need to write this in any special form as long as you add the control values into the process update. Meaning you don't need to figure out matrix G to do the process update step.
>
> The matrix H is the observation matrix. This acts to relate the observed variables to the state variables. For example, say that you have a state vector of (x_1, x_2, x_3) and can observe all three as $z = (z_{x_1}, z_{x_2}, z_{x_3})$. Then
>
> $$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$
>
> However, if we observe x_1 and x_3 as $z = (z_{x_1}, z_{x_3})$ then
>
> $$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
>
> or if we only observe x_2 as $z = (z_{x_2})$ then
>
> $$H = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$
>
> Note what the matrix H does in the following product HAH^T for the observation $z = (z_{x_1}, z_{x_3})$:
>
> $$HAH^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & c \\ d & f \\ g & i \end{bmatrix} = \begin{bmatrix} a & c \\ g & i \end{bmatrix}$$

Moving on to the derivation, we assume that we can write our estimate as a combination of the process update and the observation

$$\hat{x}_k = \tilde{x}_k + K_k(z_k - H\tilde{x}_k) \tag{16.4}$$

The optimal choice of the Kalman gain parameter is to select K_k to minimize the mean square error $E[\|x_k - \hat{x}_{k|k}\|^2]$. You will notice that

$$E[\|x_k - \hat{x}_{k|k}\|] = E\left[\sum_i (x_k^i - \hat{x}_{k|k}^i)^2\right] = Tr(P_{k|k})$$

where $Tr(P_{k|k})$ is the trace of $P_{k|k}$. So, we need an expression for $P_{k|k}$ in terms of the Kalman gain.

We can plug in the observation, (16.1) into (16.4)

$$\hat{x}_k = \tilde{x}_k + K_k(Hx_k + w_k - H\tilde{x}_k)$$

This form of the estimate can be substituted into the error covariance

$$P_{k|k} = E[e_k e_k^T] = E[[(I - K_kH)(x_k - \tilde{x}_k) - K_k w_k][(I - K_kH)(x_k - \tilde{x}_k) - K_k w_k]^T].$$

Since observation or measurement noise is not correlated to process noise we can rewite

$$P_{k|k} = (I - K_kH)E[(x_k - \tilde{x}_k)(x_k - \tilde{x}_k)^T](I - K_kH)^T - K_kE[w_k w_k^T]K_k^T.$$

Since $P_{k|k-1} = E[(x_k - \tilde{x}_k)(x_k - \tilde{x}_k)^T]$ we obtain

$$P_{k|k} = (I - K_kH)P_{k|k-1}(I - K_kH)^T - K_kWK_k^T.$$

Expanding the expression and using $S_k = HP_{k|k-1}H^T + W_k$ we have

$$P_{k|k} = P_{k|k-1} - K_kHP_{k|k-1} - P_{k|k-1}H^TK_K^T + K_kS_kK_k^T$$

As stated above, we want to minimize $Tr(P_{k|k})$ with respect to K_k:

$$\frac{\partial Tr(P_{k|k})}{\partial K_k} = -2(HP_{k|k-1})^T + 2K_kS_k = 0,$$

solving for the Kalman gain gives

$$K_k = P_{k|k-1}H^TS_k^{-1}.$$

We can collect the results into the following algorithm:

Kalman Filter

Predict: Prediction or a priori stage

- Predicted state: $\hat{x}_{k|k-1} = F_k\hat{x}_{k-1|k-1} + G_ku_k$
- Predicted estimate covariance: $P_{k|k-1} = F_kP_{k-1|k-1}F_k^T + V_k$

Update: Update or a posteriori stage

- Innovation residual or measurement residual: $y_k = z_k - H_k\hat{x}_{k|k-1}$
- Innovation (or residual) covariance: $S_k = H_k P_{k|k-1} H_k^{\mathrm{T}} + W_k$
- Optimal Kalman gain: $K_k = P_{k|k-1} H_k^{\mathrm{T}} S_k^{-1}$
- Updated state estimate $\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k y_k$
- Updated estimate covariance: $P_{k|k} = (I - K_k H_k) P_{k|k-1}$

The control input is the current control input and depends on how you index it as to being u_k or u_{k-1}. You can think of this control being injected between k and $k-1$. So it is not critical how you index the term and will be clear from the process equations.

If the model is accurate, and the values for $\hat{x}_{0|0}$

and $P_{0|0}$ accurately reflect the distribution of the initial state values, then the following invariants are preserved: (all estimates have mean error zero)

- $\mathrm{E}[x_k - \hat{x}_{k|k}] = \mathrm{E}[x_k - \hat{x}_{k|k-1}] = 0$
- $\mathrm{E}[z_k] = 0$

where $E[\xi]$ is the expected value of ξ.

Assume that you have the following Gaussian process and observation:

$$\begin{aligned} x_k &= Fx_{k-1} + Gu_k + v_k \\ z_k &= Hx_k + w_k \end{aligned}$$

then

Kalman Algorithm

```
**Input** $x_0$, $P_0$
**Output** Estimates of $x_k$, $P_k$
$k = 0$
**while** (not terminated) **do**
    $k = k + 1$
    $x_k = F_k x_{k-1} + G_k u_k$
    $P_k = F_k P_{k-1} F_k^T + V_k$
    $y_k = z_k - H_k x_k$
    $S_k = H_k P_k H_k^{\mathrm{T}} + W_k$
    $K_k = P_k H_k^{\mathrm{T}} S_k^{-1}$
    $x_k = x_k + K_k y_k$
    $P_k = (I - K_k H_k) P_k$
**end while**
```

The Kalman code generally looks like

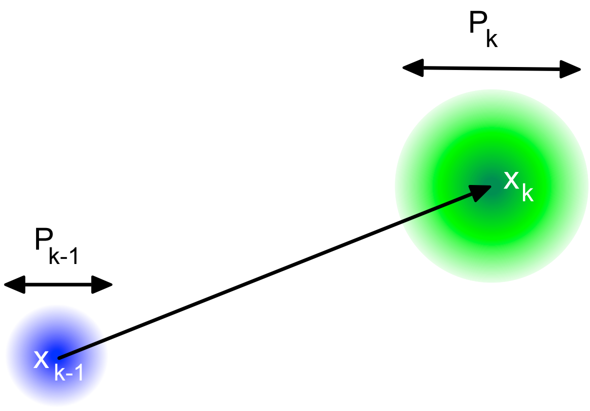


Fig. 16.5: Single Step of Kalman process.

```
k = 1
while (k<N):
  x_process_update = np.dot(F,x_estimate[k-1]) + G[k]
  P_variance_update = np.dot(F,np.dot(P_variance[k-1],FT)) + V
  innovation = z_observation[k] - np.dot(H,x_process_update)
  Innovation_covariance = np.dot(H,np.dot(P_variance_update,HT)) + W
  Kal_gain = np.dot(np.dot(P_variance_update,HT), linalg.inv(Innovation_
↪covariance))
  x_estimate[k] = x_process_update + np.dot(Kal_gain,y)
  P_variance[k] = P_variance_update - np.dot(Kal_gain,np.dot(H,P_variance_
↪update ))
  k = k+1
```

16.2.2 Simple Example of a Single Step

Let

$$x = \begin{bmatrix} a \\ b \end{bmatrix}, \quad F = \begin{bmatrix} 0.9 & -.01 \\ 0.02 & 0.75 \end{bmatrix}, \quad G = \begin{bmatrix} 0.1 \\ 0.05 \end{bmatrix}, \quad H = \begin{bmatrix} 1 & 0 \end{bmatrix},$$

$$V = \begin{bmatrix} 0.005265 & 0 \\ 0 & 0.005265 \end{bmatrix}, \quad W = 0.7225, \quad z_1 = 0.01$$

$$u_k = \sin(7 * k/100), \quad x_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad P_0 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

Apply the Kalman Filter process and compute $\hat{x}_{1|1}$ and $P_{1|1}$.

Process update:

$$\hat{x}_{1|0} = \begin{bmatrix} 0.9 & -.01 \\ 0.02 & 0.75 \end{bmatrix} \hat{x}_{0|0} + \begin{bmatrix} 0.1 \\ 0.05 \end{bmatrix} u_k = \begin{bmatrix} 0.9 & -.01 \\ 0.02 & 0.75 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.05 \end{bmatrix} \sin(7/100)$$

$$\approx \begin{bmatrix} 0.0069942847 \\ 0.0034971424 \end{bmatrix}$$

Process covariance update:

$$P_{1|0} = FP_{0|0}F^T + V =$$

$$P_{1|0} = \begin{bmatrix} 0.9 & -.01 \\ 0.02 & 0.75 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0.9 & 0.02 \\ -.01 & 0.75 \end{bmatrix} + \begin{bmatrix} 0.005265 & 0 \\ 0 & 0.005265 \end{bmatrix}$$

$$= \begin{bmatrix} 0.005265 & 0 \\ 0 & 0.005265 \end{bmatrix}.$$

Innovation and innovation covariance:

$$y_1 = 0.01 - \begin{bmatrix} 1 & 0 \end{bmatrix} \hat{x}_{1|0} = 0.01 - \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 0.0069942847 \\ 0.0034971424 \end{bmatrix}$$

$$= 0.0030057153$$

$$S_1 = HP_{1|0}H^{\mathrm{T}} + W = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 0.005265 & 0 \\ 0 & 0.005265 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 0.7225$$

$$= 0.728125$$

Kalman Gain

$$K_1 = P_{1|0}H_1^{\mathrm{T}}S_1^{-1} = \begin{bmatrix} 0.005265 & 0 \\ 0 & 0.005265 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} /0.728125$$

$$= \begin{bmatrix} 0.00772532 \\ 0.0 \end{bmatrix}$$

Updated state variables

$$\hat{x}_{1|1} = \hat{x}_{1|0} + K_1 y_1 = \begin{bmatrix} 0.0069942847 \\ 0.0034971424 \end{bmatrix} + \begin{bmatrix} 0.00772532 \\ 0.0 \end{bmatrix} (0.00300572)$$

$$= \begin{bmatrix} 0.007017504813 \\ 0.0034971424 \end{bmatrix}$$

State variable covariance:

$$P_{1|1} = (I - K_1 H_1)P_{1|0} = \begin{bmatrix} 0.99227468 & 0.0 \\ 0.0 & 1.0 \end{bmatrix} P_{1|0}$$

$$= \begin{bmatrix} 0.005224326 & 0.0 \\ 0.0 & 0.005265 \end{bmatrix}$$

It is useful to visualize the effects of a single Kalman step. The images are provided in Fig. 16.8 - Fig. 16.10 and the numbers used are not the same as the example above[1]. The system we use is Let

$$x_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad P_0 = \begin{bmatrix} 0.01 & 0 \\ 0 & 0.001 \end{bmatrix}, \quad F = \begin{bmatrix} 0.85 & -.1 \\ 0.02 & 0.75 \end{bmatrix},$$

$$G = \begin{bmatrix} 0.025 \\ 0.05 \end{bmatrix}, \quad H = I, V = \begin{bmatrix} 0.0075^2 & 0 \\ 0 & 0.0075^2 \end{bmatrix},$$

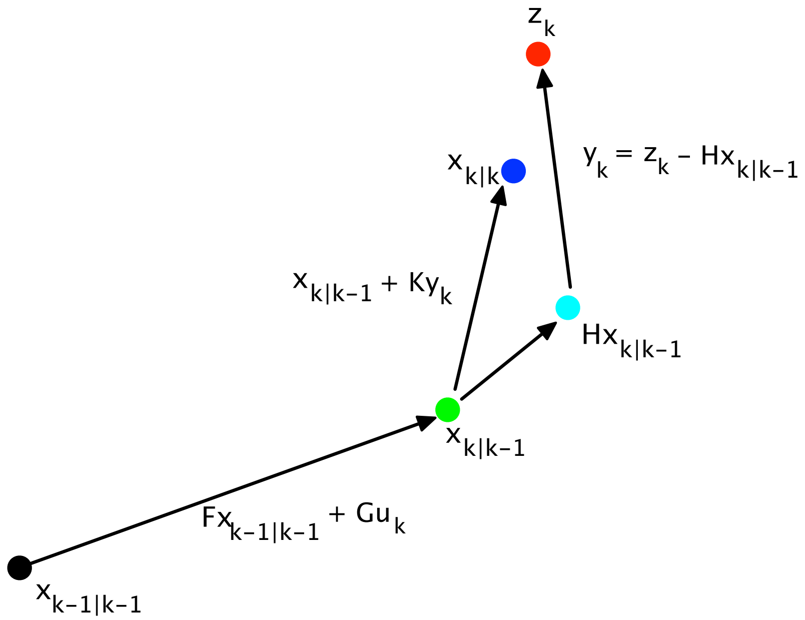


Fig. 16.6: Parts of the single Kalman step - estimate.

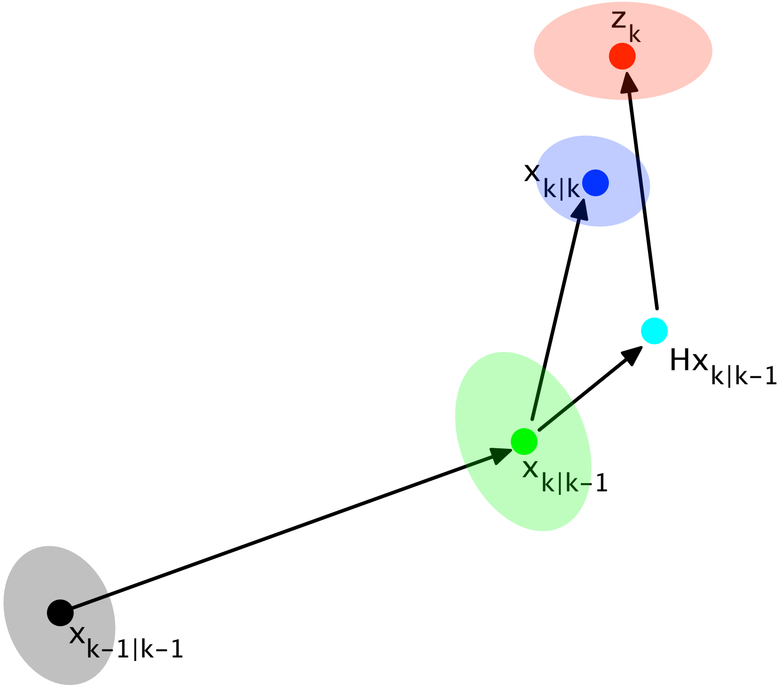


Fig. 16.7: Parts of the single Kalman step - covariances.

$$W = \begin{bmatrix} 0.035^2 & 0 \\ 0 & 0.035^2 \end{bmatrix}, \quad a = \begin{bmatrix} 0.01 \\ 0.02 \end{bmatrix}, \quad z = \hat{x} + a + w_k.$$

Starting with a single point, we move this forward using the process update. From the same starting point we run each forward with the process update, $\hat{x}_{k|k-1}$ many times to generate a distribution. The resulting points are different since the process update has noise. Fig. 16.8 shows the point cloud (in blue). This process does not have a great deal of noise so the cloud is tightly clustered. Fig. 16.9 shows the observation z_k. Fig. 16.10 shows the observation update, the fusion of the observation with the state update.

```
for i in range(M):
    x_process_update = np.dot(F,x_initial) + G + np.random.normal(mu1,sigma1,␣
↪2)
    P_variance_update = np.dot(F,np.dot(P_initial,FT)) + V
    z_test_data = np.dot(F,_initial) + G + a + np.random.normal(mu2,sigma2, 2)
    innovation = z_test_data - x_process_update
    Innovation_variance = P_variance_update + W
    kal_gain = np.dot(pp,linalg.inv(Innovation_variance))
    x_filter = x_process_update + np.dot(kal_gain,innovation)
```

You will notice that it is not circular. The covariance matrix really trusted the y process estimate and so weighted the process more than the observation. In the x estimate, much more of the observation was used. So the resulting point cloud has lower variation in y than x. Fig. 16.11 graphs the error ellipses for the previous point clouds. It is easier to see the changes from this than looking at the raw data.

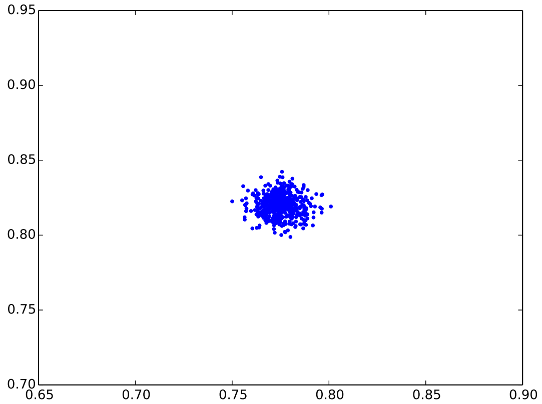


Fig. 16.8: Point distribution after process update.

16.2.3 Kalman Code and Generation of Testing Data

The development of filtering software needs to have datasets to test the software. The early stages of software development are about removing simple errors such as syntax errors. In the absence of a real robot producing actual data, how do we develop and test our code? This can be done using pure simulation. We can simulate the motion of a robot. In practice we just compute the location and orientation of the robot based on

[1] The numbers were selected to help visualize the process.

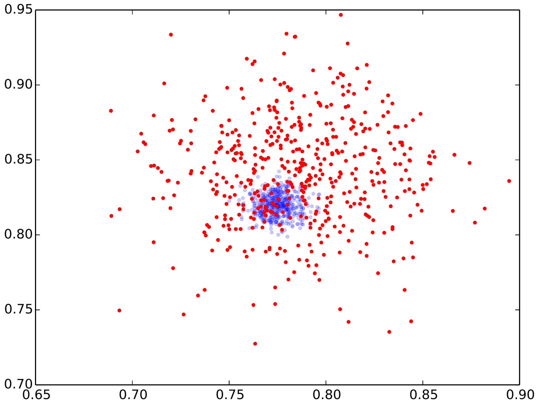


Fig. 16.9: Observed point distribution.

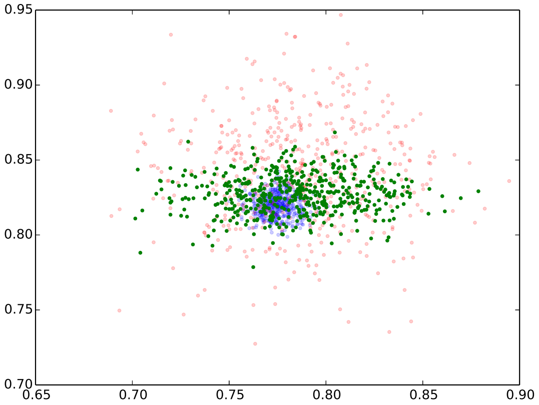


Fig. 16.10: Final distribution after update step.

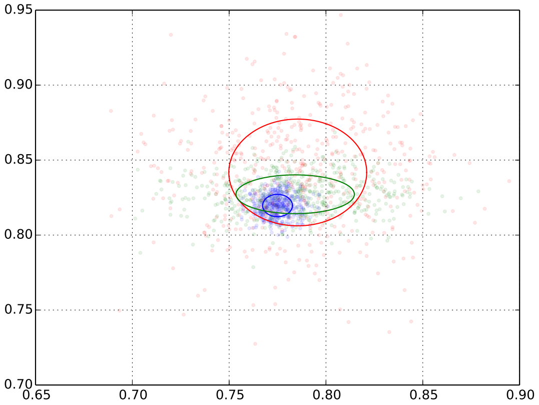


Fig. 16.11: The standard deviation based ellipses.

Z
Kalman Filter
^
X

Fig. 16.12: Kalman Code as a black box.

the motion equations or kinematics derived in the Motion chapter. For example, for the differential drive robot, we can send control signals (the wheel speeds) and compute the location of the robot. Each step of the simulation produces a small motion and a small amount of error. That error will accumulate which is consistent with what we see in actual systems. Assume that the robot moves along according to the kinematic model F and G plus the noise, we have

$$x_{k+1} = Fx_k + Gu_k + v_k$$

This produces the robot path as a vector of values $\{x\}$.

At each step along the computed path, we can make an observation (z_k) which is noise added to the exact values $x_k + v_k$ where v_k is Gaussian noise. Since z_k is not added back into the computation for x_{k+1}, the observation noise, w_k, does not accumulate. The process is the following:

$$x_{k+1} = Fx_k + Gu_k + v_k$$

$$z_{k+1} = Hx_k + w_k$$

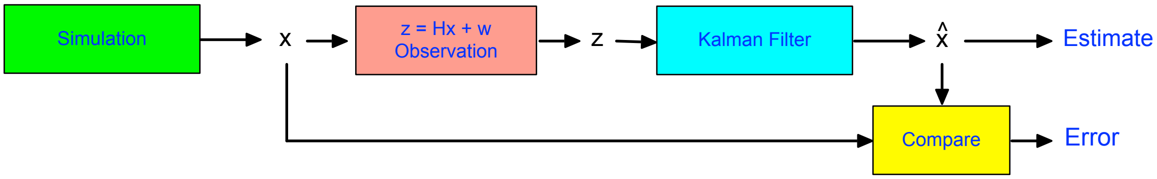


Fig. 16.13: Simulation and testing.

The point is that the observations z can be computed after we compute the x values or they can be computed together in the loop. It does not matter in this case.

For this next example we modify the last example in a couple of ways. We will observe both variables. This will have the effect of making the innovation covariance S a matrix and we will need to compute a matrix inverse. Next we will use a non-diagonal noise covariance for V and W.

We use these values to run a simulation which then produces the observations we need to feed into the Kalman filter. The code block below will generate a list of values which can be used as the observations for a run of a Kalman filtering algorithm. Let

$$x = \begin{bmatrix} a \\ b \end{bmatrix}, \quad F = \begin{bmatrix} 0.85 & -.01 \\ 0.02 & 0.65 \end{bmatrix}, \quad G = \begin{bmatrix} 0.1 \\ 0.05 \end{bmatrix}, \quad H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

$$V = \begin{bmatrix} 0.2 & 0.02 \\ 0.02 & 0.35 \end{bmatrix}, \quad W = \begin{bmatrix} 0.4 & 0.0 \\ 0.0 & 0.4 \end{bmatrix}.$$

$$Gu_k = \begin{bmatrix} 0.2\sin(0.025k) \\ 0.075\cos(0.025k) \end{bmatrix}, \quad x_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad P_0 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

The includes …

```
from math import *
import numpy as np
import pylab as plt
from scipy import linalg
```

The simulation variables …

```
#  Create fake dataset for experiment
N = 200
t = np.linspace(0, 10, N)  # for control input
u1 = 0.75*np.sin(0.5*t)
u2 = 0.5*np.cos(0.5*t)
mu1 = [0.0,0.0]
mu2 = [0.0,0.0]
x_sim = np.zeros((N,2))
z_sim = np.zeros((N,2))
F = np.array([[0.85,-0.01],[0.02,0.65]])
FT = F.T
G = np.array([u1, u2]).T
```

The filter variables

```
H = np.array([[1,0],[0,1]])
HT = H.T
V = np.array([[0.2,0.02],[0.02,0.35]])
W = np.array([[0.4,0.0],[0.0,0.4]])
P = np.zeros((N,2,2))
x_estimate = np.zeros((N,2))
```

The simulation …

```
k = 1
while (k<N):
  process_noise = np.random.multivariate_normal(mu1,V,1)
  observation_noise = np.random.multivariate_normal(mu2,W, 1)
  x_sim[k] = np.dot(F,x_sim[k-1]) + G[k] + process_noise
  z_sim[k] = np.dot(H,x_sim[k]) + observation_noise
  k = k+1
# done with fake data
```

The code block above provides the array z which is then piped into the Kalman Filter

```
k = 1
while (k<N):
  x_process_update = np.dot(F,x_estimate[k-1]) + G[k]
```

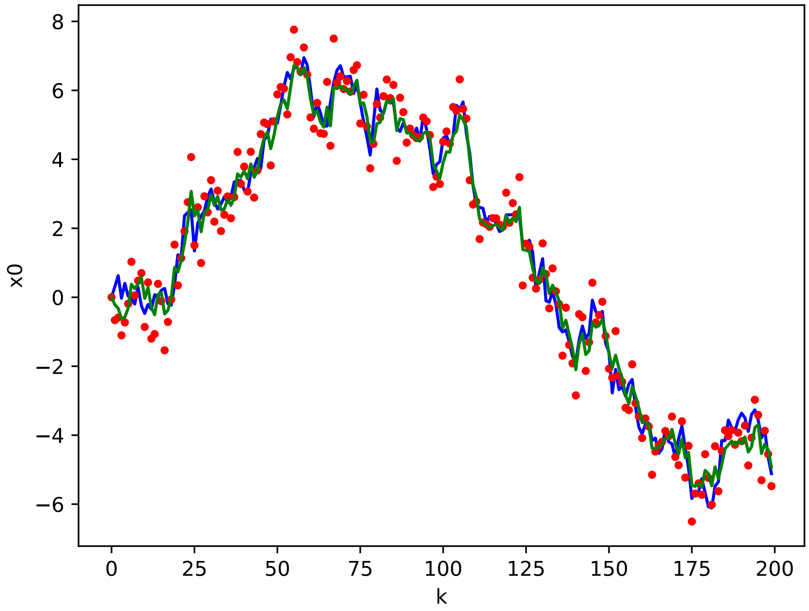
(continues on next page)

(continued from previous page)

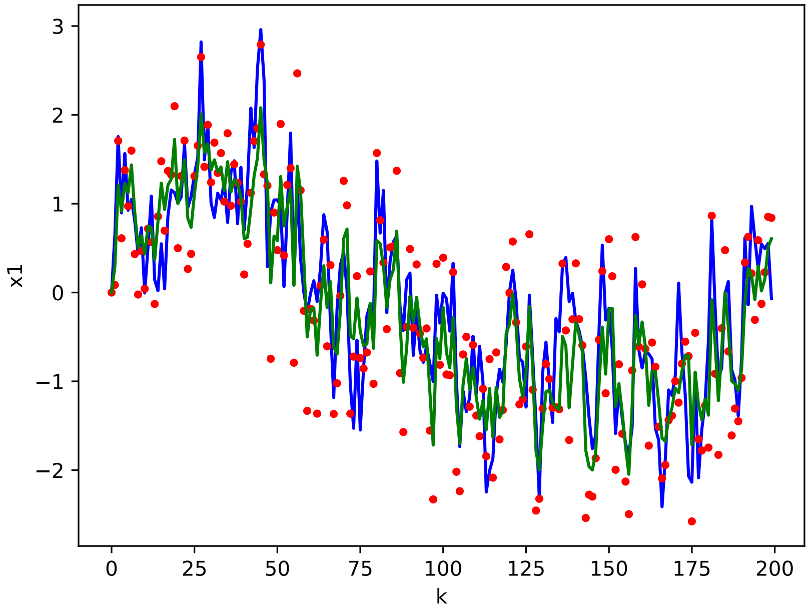
```
  P_variance_update = np.dot(F,np.dot(P_variance[k-1],FT)) + V
  innovation = z_observation[k] - np.dot(H,x_process_update)
  Innovation_covariance = np.dot(H,np.dot(P_variance_update,HT)) + W
  Kal_gain = np.dot(np.dot(P_variance_update,HT), linalg.inv(Innovation_
↪covariance))
  x_estimate[k] = x_process_update + np.dot(Kal_gain,y)
  P_variance[k] = P_variance_update - np.dot(Kal_gain,np.dot(H,P_variance_
↪update ))
  k = k+1
```

```
t = np.arange(0,N,1)
plt.xlabel('k')
plt.ylabel('x0')
plt.plot(t, x_sim[:,0], 'b-', t,z_sim[:,0],'r.', t, x_estimate[:,0],'g-')
plt.savefig("kalmandemo2_x.pdf",format="pdf")
plt.show()

plt.xlabel('k')
plt.ylabel('x1')
plt.plot(t, x_sim[:,1], 'b-', t, z_sim[:,1],'r.', t, x_estimate[:,1],'g-')
plt.savefig("kalmandemo2_y.pdf",format="pdf")
plt.show()
```

The blue dots are a graph of $(x_0)_k$, the red dots are the observations z_k, and the green dots are the Kalman estimate of the state.



The blue dots are a graph of $(x_1)_k$, and the green dots are the Kalman estimate of the state.



16.2.4 How to inject noise

You may have noticed that we have added noise to the end of the expression. Why add? Why not multiply? Assume that we have two signals

$$a(t) = \cos(t), \quad b(t) = 20\cos(t)$$

and to them we add mean zero Gaussian noise with standard deviation $\sigma = 0.25$, v:

$$a_1(t) = \cos(t) + v, \quad b_1(t) = 20\cos(t) + v$$

or we multiply that noise

$$a_2(t) = v\cos(t), \quad b_2(t) = 20v\cos(t)$$

We then subtract off the signal and compute the standard deviations. For a_1 and b_1, it is mathematically clear that you would get $\sigma = 0.25$ back - if the sample size large enough.

```
>>> c = np.cos(t)
>>> a1 = c + np.random.normal(0, 0.25,100)
>>> b1 = 20*c + np.random.normal(0, 0.25,100)
>>> a2 = np.random.normal(0, 0.25,100)*c
>>> b2 = 20*np.random.normal(0, 0.25,100)*c
>>> a1sub = a1 - c
>>> b1sub = b1 - 20*c
>>> a2sub = a2 - c
>>> b2sub = b2 - 20*c
>>> np.std(a1sub)
```

(continues on next page)

(continued from previous page)

```
0.26168514491592509
>>> np.std(b1sub)
0.20957486503563907
>>> np.std(a2sub)
0.73517338736953186
>>> np.std(b2sub)
14.687819454616823
```

The multiplication by the signal will amplify the noise by the signal strength and this changes the effective standard deviation. We will for this text focus on adding noise via addition. One issue we will address later in this chapter is the difference between process noise and control noise. By process noise we mean the addition of noise in the process step, the addition of v:

$$x_{k+1} = Fx_k + Gu_k + v_k.$$

Noise in the control would appear as $u_k + r_k$ where r_k is some zero mean noise term. This would get changed by the term G

$$x_{k+1} = Fx_k + G(u_k + r_k) + v_k = Fx_k + Gu_k + Gr_k + v_k = Fx_k + Gu_k + v_k'.$$

For now, we just assume we can lump the two together with a modified process noise term.

16.2.5 The Classic Vehicle on Track Example

Consider a mobile robot along a track. Let the state $x = [x_r, s_r]$

where x_r and s_r are the vehicle position and speed. Let m

denote the mass of the vehicle and u be the force acting on the vehicle. Note that

$$\frac{ds_r}{dt} = \frac{u}{m}$$

Discretize

$$\frac{s_r(t+T) - s_r(t)}{T} \approx \frac{ds_r}{dt}$$

T is the sample rate. Thus

$$s_r(k+1) = s_r(k) + \frac{T}{m}u(k).$$

From calculus we know that

$$\frac{dx_r}{dt} = s_r.$$

Discretizing this equation

$$\frac{dx_r}{dt} \approx \frac{x_r(k+1) - x_r(k)}{T} = s_r(k)$$

and rewriting gives

$$x_r(k+1) = x_r(k) + T s_r(k).$$

This gives the pair of equations

$$\begin{aligned} x_r(k+1) &= x_r(k) + T s_r(k) \\ s_r(k+1) &= s_r(k) + \tfrac{T}{m}\, u(k) \end{aligned}$$

Load the variables into an array

$$x_{k+1} = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} 0 \\ T/m \end{bmatrix} u_k + v_k$$

Assume that you have some sensors

$$z_{k+1} = \begin{bmatrix} 0 & 1 \end{bmatrix} x_k + w_k$$

where v and w are zero mean Gaussian noise. Thus

$$F_k = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}, \quad G_k = \begin{bmatrix} 0 \\ T/m \end{bmatrix}, \quad H_k = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

For this example take $m = 1$ and $T = 0.5$. Assume the covariance of v_k

$$V_k = \begin{bmatrix} 0.2 & 0.05 \\ 0.05 & 0.1 \end{bmatrix}$$

Assume the covariance for w_k is $W_k = [0.5]$, and at $k = 0$, $u(0) = 0$ and $\hat{x}_{0|0} = \begin{bmatrix} 2 & 4 \end{bmatrix}^T$,

$$P_{0|0} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

Next we compute one iteration of the Kalman Filter.

- State estimate prediction:

$$\hat{x}_{1|0} = F_1 \hat{x}_{0|0} + G_1 u_1 = \begin{bmatrix} 1 & 0.5 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.5 \end{bmatrix} 0 = \begin{bmatrix} 4 \\ 4 \end{bmatrix}$$

- Covariance prediction

$$P_{1|0} = F_1 P_{0|0} F_1^T + V_1$$

$$= \begin{bmatrix} 1 & 0.5 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0.5 & 1 \end{bmatrix} + \begin{bmatrix} 0.2 & 0.05 \\ 0.05 & 0.1 \end{bmatrix} = \begin{bmatrix} 1.7 & 1.05 \\ 1.05 & 2.1 \end{bmatrix}$$

Assume that you measure and obtain

$$z_1 = 3.8$$

- Innovation:

$$y_k = z_1 - H\hat{x}_{1|0} = 3.8 - \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 4 \end{bmatrix} = -.2$$

- The matrix S

$$S_1 = HP_{1|0}H^{\mathrm{T}} + W_1 = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 1.7 & 1.05 \\ 1.05 & 2.1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 0.5 = 2.6$$

- The matrix K (Kalman Gain)

$$K_1 = P_{1|0}H^{\mathrm{T}}S_1^{-1} = \begin{bmatrix} 1.7 & 1.05 \\ 1.05 & 2.1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} (2.6)^{-1} = \begin{bmatrix} 0.404 \\ 0.808 \end{bmatrix}$$

- The estimate update:

$$\hat{x}_{1|1} = \hat{x}_{1|0} + K_1 y_1 = \begin{bmatrix} 4 \\ 4 \end{bmatrix} + \begin{bmatrix} 0.404 \\ 0.808 \end{bmatrix} (-.2) = \begin{bmatrix} 3.9192 \\ 3.8384 \end{bmatrix}$$

- The covariance estimate update:

$$P_{1|1} = (I - K_1 H)P_{1|0}$$

$$= \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 0.404 \\ 0.808 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix} \right) \begin{bmatrix} 1.7 & 1.05 \\ 1.05 & 2.1 \end{bmatrix} = \begin{bmatrix} .4242 & .8484 \\ .8484 & 1.6968 \end{bmatrix}$$

16.2.6 The Kalman Gain

The Kalman Gain selects the amount of process update to be used compared to the amount of observation to be used. It is weighting each one to produce the best possible estimate of state based on the current understanding of the errors on both.

The Kalman Gain can be written as

$$K_k = P_k H_k^{\mathrm{T}} \left(H_k P_k H_k^{\mathrm{T}} + W_k \right)^{-1}.$$

If all of these variables were *scalars*, we can get a feel for the bounds on the Kalman Gain:

$$K_k = P_k H_k / \left(H_k^2 P_k + W_k \right)$$

When $W_k = 0$ then $K_k = 1/H_k$ and as $W_k \to \infty$ then $K_k = 1$, so $1/H_k < K_k < 1$.

16.2.7 Some issues to address

Because the Kalman filter is trying to estimate the state, and determine the process as well as the observation quality, the initial iterations may be very inaccurate. Assuming you have a convergent process, it can still take some time for the filter to converge and provide a good state estimate. What the filter is doing is figuring out the errors for the state estimate (the covariance P). Many robotics applications will have the robot sit still for a few seconds to allow the filter to converge.

A common question is what should the initial values be? For the state estimate, one clearly uses starting information that one has. The problem is that maybe not all the data is known. For unknown variables, setting to zero is about all you can do. The corresponding entry in the covariance matrix should be infinity (or a very large value). Another approach for the covariance is to set it to zero and let the first dozen iterations figure out the covariance or one can populate it with values. One could even store the covariance after the filter settles and use that to initialize the filter.

For matrix W, we use the sensor datasheets which can provide standard deviations for sensor readings. The squares of those can be placed on the diagonal of W. The matrix V is harder to determine and may require some experimentation. A simplistic approach would be to run the robot for a single step and measure the end state. Repeat this process for a large enough times as possible. That endstate measurement data can be used to determine the variances of the process as well as can be used to adjust the process in case of parameter issues.

A variation on this approach for V is to run the robot in for multiple time steps and do the statistics on the end state as before. Another method is to compare the Kalman estimation with the actual state (done by hand measurement and not onboard sensing). The tune the parameters. You can then optimize to gain good choices for V, W. It should be noted that V is the estimate for a given Δt. It needs to be scaled for time steps other than the one it was developed for. So, if V was developed for a time step of Δt and the Kalman estimation loops are using a time step of T , then $V' = (T/\Delta t)V$ would scale the covariance.

One concern follows from unreliable sensor connections. What happens when a sensor is down or is not sending data? The Kalman gain is the term that selects the relative amount of the model verses the sensor to use in the estimate. Lacking a sensor, the Kalman gain will after some iterations shut off that sensor. It will do this even if the sensor is operational. It the sensor is giving readings that don't make sense given the physical model, the Kalman gain will reset to where only the physical model is used.

$$K_k = P_k H_k^T S_k^{-1} \to 0$$

This can be a problem for sensors that have drift or some type of uncorrected deterministic error (DC bias).

The Kalman filter does not correct for drift that occurs in gyros and other instruments. The common fix is to periodically reset (zero) the sensor when in a known configuration - for example when the vehicle is stopped and you know it is not turning. The issue of course is that after a period of time the Kalman estimate becomes just the process update step. The Kalman Gain parameter can be monitored. When it falls below some threshold, then the sensor needs to be reset.

Work Estimates

If you have n equations, the work (multiplications) in the filter is:

1. $\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + G_k u_k : O(n^2)$

2. $P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + V_k : \ O(n^3)$
3. $K_k = P_{k|k-1} H_k^{\mathrm{T}} \left[H_k P_{k|k-1} H_k^{\mathrm{T}} + W_k \right]^{-1} : \ O(m!) + O(n^2 m)$
4. $\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \left(z_k - H_k \hat{x}_{k|k-1} \right) : \ O(n^2)$
5. $P_{k|k} = (I - K_k H_k) P_{k|k-1} : \ O(n^3)$

The largest work is in step 3. By using an LU factorization, we can move this down to $\max(O(m^3), O(n^2 m))$ work. Step 2 can exploit symmetry to reduce work as only 1/2 the matrix needs to be computed. For small matrices, explicit formulas for the inverse can be used.

16.2.8 Different Sensor Types

Now that we have the basic Kalman Filter process, we can look at some variations on how it is applied. One question that arises is "What should we do if we have multiple sensors?" Currently, the update stage runs a single measurement fusion. The solution is to run the update loop for each sensor. This is equivalent to running the full Kalman loop but skipping the prediction step between the different sensors. The algorithm follows.

Predict:

- $\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + G_k u_k$
- $P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + V_k$

Update:

- foreach sensor i:
  - $y_k = z_k^i - (H^i)_k \hat{x}_{k|k-1}$
  - $S_k = (H^i)_k P_{k|k-1} (H^i)_k^{\mathrm{T}} + W_k^i$
  - $K_k = P_{k|k-1} (H^i)_k^{\mathrm{T}} S_k^{-1}$
  - $\hat{x}_{k|k-1} = \hat{x}_{k|k-1} + K_k y_k$
  - $P_{k|k-1} = (I - K_k (H^i)_k) P_{k|k-1}$
- $\hat{x}_{k|k} = \hat{x}_{k|k-1}$
- $P_{k|k} = P_{k|k-1}$

From this algorithm we notice that we have the ability to fuse multiple different sensors; meaning you have multiple sensors measuring a single state x_k. Using the update steps we can fuse sensor measurements without the need to perform the prediction step. Sensor fusion can be done using a simplification of the Kalman Filter. Since we only have observations, $F = I$, $G = 0$, $V = 0$ and so the apriori stage of the filter drops out: So, we can just skip the apriori step. This means we can define $\hat{x}_k = \hat{x}_{k|k}$ and $P_k = P_{k|k}$ and we have a basic formula to merge the sensed data. Since we don't have the time loop (in k), we can redefine k to loop over the sensors. This reduces to exactly the sensor fusion algorithm given in Section 15.3.5.

In the last section we discussed the issue regarding unreliable sensor readings in the situation where the data is occasionally not available. This brings up a concern about having the data ready when the update step is done. The assumption so far was that the Kalman loop is run at the same frequency that the data is arriving.

However, there are several situations for which this is a problem. One such situation is when several different classes of sensors are being used. For example, your magnetometer may run at 80 Hz and your Lidar might operate at 10 Hz. One solution is to run at 10Hz and just skip the extra measurements from the magnetometer. Another possible problem arises when the time between the sensor readings are very long giving a Δt that is very large. A large Δt can make the predictive step inaccurate.

Predict:

- $\hat{x}_{k|k-1} = F_k\hat{x}_{k-1|k-1} + G_k u_k$
- $P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + V_k$

Update:

- Loop over available sensor data during Δt :
  - $y_k = z_k^i - (H^i)_k \hat{x}_{k|k-1}$
  - $S_k = (H^i)_k P_{k|k-1} (H^i)_k^{\mathrm{T}} + W_k^i$
  - $K_k = P_{k|k-1} (H^i)_k^{\mathrm{T}} S_k^{-1}$
  - $\hat{x}_{k|k-1} = \hat{x}_{k|k-1} + K_k y_k$
  - $P_{k|k-1} = (I - K_k (H^i)_k) P_{k|k-1}$
- $\hat{x}_{k|k} = \hat{x}_{k|k-1}$
- $P_{k|k} = P_{k|k-1}$

16.3 Extended Kalman Filter

Recall that when we started the section on linear dynamical systems the first example was the differential drive dynamics. These equations are not linear due to the cos and sin terms. What does one do in the case where the process or the observation is a nonlinear function? For a nonlinear process, let $x_k \in R^n$, $u_k \in R^p$, $v_k \in R^n$, $w_k \in R^m$,

$$x_k = f(u_k, x_{k-1}) + v_k,$$

$$z_k = h(x_{k-1}) + w_k$$

where v_k has variance V_k and w_k has variance W_k, and let

$$F_k = \partial f / \partial x = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \dots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}, H_k = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \frac{\partial h_1}{\partial x_2} & \dots & \frac{\partial h_1}{\partial x_n} \\ \frac{\partial h_2}{\partial x_1} & \frac{\partial h_2}{\partial x_2} & \dots & \frac{\partial h_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \\ \frac{\partial h_m}{\partial x_1} & \frac{\partial h_m}{\partial x_2} & \dots & \frac{\partial h_m}{\partial x_n} \end{bmatrix}$$

This is a Taylor expansion approach to dealing with nonlinear mappings. For more information about Jacobians, https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant.

1. Predicted state:

$$\hat{x}_{k|k-1} = f(\hat{x}_{k-1|k-1}, u_k)$$

2. Predicted estimate covariance:

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + V_k$$

3. Optimal Kalman gain:

$$K_k = P_{k|k-1} H_k^{\mathrm{T}} \left(H_k P_{k|k-1} H_k^{\mathrm{T}} + W_k\right)^{-1}$$

4. Updated state estimate:

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \left(z_k - h(\hat{x}_{k|k-1})\right)$$

5. Updated estimate covariance:

$$P_{k|k} = (I - K_k H_k) P_{k|k-1}$$

Summed up into a procedure, we have:

EFK Algorithm

```
Input $x_0$, $P_0$
Output Estimates of $x_k$, $P_k$
$k = 0$
while (not terminated) do
    $k = k + 1$
    $x_k = f(\hat{x}_{k-1|k-1}, u_k)$
    $F_k = \text{Jacobian}(f)|_{x_k}$
    $P_k = F_k P_{k-1} F_k^T + V_k$
    $H_k = \text{Jacobian}(h)|_{x_k}$
    $y_k = z_k - H_k x_k$
    $S_k = H_k P_k H_k^{\mathrm{T}} + W_k$
    $K_k = P_k H_k^{\mathrm{T}} S_k^{-1}$
    $x_k = x_k + K_k y_k$
    $P_k = (I - K_k H_k) P_k$
end while
```

16.3.1 Short Example

What is the Extended Kalman Filter formulation of the motion model:

$$\begin{array}{l}\dot{x} = y \\ \dot{y} = -\cos(x) + 0.4\sin(t)\end{array},$$

and observation

$$h(x, y) = \begin{bmatrix} x \\ y \end{bmatrix}$$

with step size $\Delta t = 0.1$, and noise

$$V = \begin{bmatrix} 0.1 & 0.01 \\ 0.01 & 0.1 \end{bmatrix}, \quad , W = \begin{bmatrix} 0.05 & 0 \\ 0 & 0.05 \end{bmatrix}.$$

Assume that the initial values, $t_0 = 0$,

$$x_0 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

and we have a reasonably accurate start

$$P_0 = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}$$

Using a basic Euler formulation we replace the derivative:

$$\begin{aligned} \frac{x_{k+1} - x_k}{0.1} &= y_k \\ \frac{y_{k+1} - y_k}{0.1} &= -\cos(x_k) + 0.4\sin(t_k) \end{aligned}$$

This gives the discrete form:

$$\begin{aligned} x_{k+1} &= x_k + 0.1 y_k \\ y_{k+1} &= y_k - 0.1\cos(x_k) + 0.04\sin(t_k) \end{aligned}$$

and so

$$F = \begin{bmatrix} 1 & 0.1 \\ 0.1\sin(x_k) & 1 \end{bmatrix}, \quad H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

EKF:

1. Predicted state:

$$\hat{x}_{1|0} = f(\hat{x}_{0|0}, u_1) = \begin{pmatrix} 1 + 0.1(1) \\ 1 - 0.1\cos(1) + 0.04\sin(0) \end{pmatrix} = \begin{pmatrix} 1.1 \\ 0.45969769 \end{pmatrix}$$

2. Predicted estimate covariance:

$$P_{1|0} = F_1 P_{0|0} F_1^T + V_1$$

$$= \begin{bmatrix} 1 & 0.1 \\ 0.1\sin(1) & 1 \end{bmatrix} \begin{bmatrix} 0.05 & 0 \\ 0 & 0.05 \end{bmatrix} \begin{bmatrix} 1 & 0.1\sin(1) \\ 0.1 & 1 \end{bmatrix} +$$

$$\begin{bmatrix} 0.1 & 0.01 \\ 0.01 & 0.1 \end{bmatrix} = \begin{bmatrix} 0.605 & 0.10207355 \\ 0.10207355 & 0.60354037 \end{bmatrix}$$

3. Optimal Kalman gain:

$$K_1 = P_{1|0} H_1^{\mathrm{T}} \left(H_1 P_{1|0} H_1^{\mathrm{T}} + W_1 \right)^{-1}$$

$$= \begin{bmatrix} 0.605 & 0.10207355 \\ 0.10207355 & 0.60354037 \end{bmatrix}$$

$$\times \left(\begin{bmatrix} 0.605 & 0.10207355 \\ 0.10207355 & 0.60354037 \end{bmatrix} + \begin{bmatrix} 0.05 & 0 \\ 0 & 0.05 \end{bmatrix} \right)^{-1}$$

$$= \begin{bmatrix} 0.92175979 & 0.01221999 \\ 0.01221999 & 0.92158505 \end{bmatrix}$$

4. Updated state estimate:

$$\hat{x}_{1|1} = \hat{x}_{1|0} + K_1 \left(z_1 - h(\hat{x}_{1|0}) \right)$$

$$= \begin{pmatrix} 1.1 \\ 0.45969769 \end{pmatrix} + \begin{bmatrix} 0.92175979 & 0.01221999 \\ 0.01221999 & 0.92158505 \end{bmatrix}$$

$$\times \left(\begin{pmatrix} 1.15 \\ 0.5 \end{pmatrix} - \begin{pmatrix} 1.1 \\ 0.45969769 \end{pmatrix} \right) = \begin{pmatrix} 1.14658048 \\ 0.4974507 \end{pmatrix}$$

5. Updated estimate covariance:

$$P_{1|1} = (I - K_1 H_1) P_{1|0}$$

$$= \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{bmatrix} 0.92175979 & 0.01221999 \\ 0.01221999 & 0.92158505 \end{bmatrix} \right)$$

$$\times \begin{bmatrix} 0.605 & 0.10207355 \\ 0.10207355 & 0.60354037 \end{bmatrix} = \begin{bmatrix} 0.04608799 & 0.000611 \\ 0.000611 & 0.04607925 \end{bmatrix}$$

16.3.2 Differential Drive Example

In Terms Chapter, we derived the equations for the motion of the differential drive robot. In that chapter we also simulated the motion of the robot based on wheel velocity data. Small amounts of noise in the wheel velocity data could cause significant errors in position estimation. Using the Extended Kalman Filter, we can improve the location estimate as well as gain estimates for the uncertainty of the location. Fig. 16.14 recalls the variables and equations that were derived.

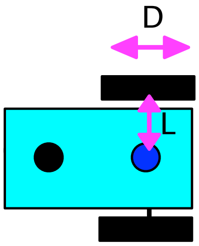


Fig. 16.14: The variables used in the DD model.

$$\dot{x} = \tfrac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\cos(\theta)$$

$$\dot{y} = \tfrac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\sin(\theta)$$

$$\dot{\theta} = \tfrac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2)$$

As with the linear continuous models, both the Kalman and Extended Kalman filters act on discrete dynamics. So as before, we need to discretize the equations.

$$\frac{x(t+\Delta t) - x(t)}{\Delta t} \approx \dot{x} = \frac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\cos(\theta)$$
$$\frac{y(t+\Delta t) - y(t)}{\Delta t} \approx \dot{y} = \frac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\sin(\theta)$$
$$\frac{\theta(t+\Delta t) - \theta(t)}{\Delta t} \approx \dot{\theta} = \frac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2)$$

The discretized variables are

$$t_k \equiv k\Delta t, \quad t_{k+1} = (k+1)\Delta t$$

$$x_k \equiv x(t_k), \quad y_k \equiv y(t_k)$$

$$\omega_{1,k} \equiv \dot{\phi}_1(t_k), \quad \omega_{2,k} \equiv \dot{\phi}_2(t_k)$$

The discrete approximations to the differential drive equations are:

$$x_{k+1} = x_k + \frac{r\Delta t}{2}(\omega_{1,k} + \omega_{2,k})\cos(\theta_k)$$
$$y_{k+1} = y_k + \frac{r\Delta t}{2}(\omega_{1,k} + \omega_{2,k})\sin(\theta_k)$$
$$\theta_{k+1} = \theta_k + \frac{r\Delta t}{2L}(\omega_{1,k} - \omega_{2,k})$$

The next step is to linearize the process dynamics. This means that we must compute the matrix F from the nonlinear model f.

$$x_k = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix}, \quad u_k = \begin{bmatrix} \omega_{1,k} \\ \omega_{2,k} \end{bmatrix},$$

$$f(x_k,u_k) = \begin{bmatrix} x_k + \frac{r\Delta t}{2}(\omega_{1,k}+\omega_{2,k})\cos(\theta_k) \\ y_k + \frac{r\Delta t}{2}(\omega_{1,k}+\omega_{2,k})\sin(\theta_k) \\ \theta_k + \frac{r\Delta t}{2L}(\omega_{1,k}-\omega_{2,k}) \end{bmatrix}$$

$$F_k = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\frac{r\Delta t}{2}(\omega_{1,k}+\omega_{2,k})\sin(\theta_k) \\ 0 & 1 & \frac{r\Delta t}{2}(\omega_{1,k}+\omega_{2,k})\cos(\theta_k) \\ 0 & 0 & 1 \end{bmatrix}$$

Assume that you start the robot with pose $[0,0,0]$ and you know this is exact so

$$P_{0|0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Let the process noise and measurement noise covariances be

$$V = \begin{bmatrix} 0.2 & 0.01 & 0.1 \\ 0.01 & 0.2 & 0.01 \\ 0.1 & 0.01 & 0.3 \end{bmatrix}, \quad W = \begin{bmatrix} 0.25 & 0 & 0.1 \\ 0 & 0.25 & 0.1 \\ 0.1 & 0.1 & 0.4 \end{bmatrix}$$

and the control inputs be $\omega_{1,0} = 1$, $\omega_{2,0} = 2$. Take $\Delta t = 0.1$, $r = 4$, $L = 6$.

Take

$$h_k(x_k) = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix}, \quad H_k = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and so we plug in H into our process and express:

1. $\hat{x}_{k|k-1} = f(\hat{x}_{k-1|k-1}, u_k)$
2. $P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + V_k$
3. $K_k = P_{k|k-1}\left(P_{k|k-1} + W_k\right)^{-1}$
4. $\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k\left(z_k - \hat{x}_{k|k-1}\right)$
5. $P_{k|k} = (I - K_k)P_{k|k-1}$

Note that the steps above ONLY apply to when you can observe all three variables making H the identity matrix.

$$\hat{x}_{1|0} = f(\hat{x}_{0|0}, u_0) = \begin{pmatrix} \frac{4(0.1)}{2}(1+2)\cos(0) \\ \frac{4(0.1)}{2}(1+2)\sin(0) \\ \frac{4(0.1)}{12}(1-2) \end{pmatrix} = \begin{pmatrix} 0.6 \\ 0 \\ -0.333 \end{pmatrix}$$

$$F = \begin{bmatrix} 1 & 0 & -\frac{r\Delta t}{2}(\omega_{1,k}+\omega_{2,k})\sin(\theta_k) \\ 0 & 1 & \frac{r\Delta t}{2}(\omega_{1,k}+\omega_{2,k})\cos(\theta_k) \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0.6 \\ 0 & 0 & 1 \end{bmatrix}$$

so . . .

$$P_{1|0} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0.6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0.6 & 1 \end{bmatrix} + \begin{bmatrix} 0.2 & 0.01 & 0.1 \\ 0.01 & 0.2 & 0.01 \\ 0.1 & 0.01 & 0.3 \end{bmatrix}$$

$$= \begin{bmatrix} 0.2 & 0.01 & 0.1 \\ 0.01 & 0.2 & 0.01 \\ 0.1 & 0.01 & 0.3 \end{bmatrix}$$

$$K = \begin{bmatrix} 0.2 & 0.01 & 0.1 \\ 0.01 & 0.2 & 0.01 \\ 0.1 & 0.01 & 0.3 \end{bmatrix} \left[\begin{bmatrix} 0.2 & 0.01 & 0.1 \\ 0.01 & 0.2 & 0.01 \\ 0.1 & 0.01 & 0.3 \end{bmatrix} + \begin{bmatrix} 0.25 & 0 & 0.1 \\ 0 & 0.25 & 0.1 \\ 0.1 & 0.1 & 0.4 \end{bmatrix} \right]^{-1}$$

$$= \begin{bmatrix} 0.2 & 0.01 & 0.1 \\ 0.01 & 0.2 & 0.01 \\ 0.1 & 0.01 & 0.3 \end{bmatrix} \begin{bmatrix} 2.552 & 0.126 & -0.749 \\ 0.126 & 2.317 & -0.400 \\ -0.749 & -0.400 & 1.705 \end{bmatrix}$$

$$= \begin{bmatrix} 0.437 & 0.008 & 0.017 \\ 0.043 & 0.461 & -0.070 \\ 0.032 & -0.084 & 0.433 \end{bmatrix}$$

Assume we have the observation: $z_k = [0.5, 0.025, -0.3]^T$ then the innovation

$$z_k - \hat{x}_{k|k-1} = \begin{pmatrix} -.1 \\ 0.025 \\ 0.033 \end{pmatrix}$$

So,

$$\hat{x}_{1|1} = \hat{x}_{1|0} + K_k \left(z_k - \hat{x}_{k|k-1} \right)$$

$$= \begin{pmatrix} 0.6 \\ 0 \\ -0.333 \end{pmatrix} + \begin{bmatrix} 0.437 & 0.008 & 0.017 \\ 0.043 & 0.461 & -0.070 \\ 0.032 & -0.084 & 0.433 \end{bmatrix} \begin{pmatrix} -0.1 \\ 0.025 \\ 0.033 \end{pmatrix}$$

$$\hat{x}_{1|1} = \begin{pmatrix} 0.557 \\ 0.005 \\ -0.324 \end{pmatrix}$$

$$P_{1|1} = (I - K)P_{1|0} = \begin{bmatrix} 0.563 & -0.008 & -0.017 \\ -0.043 & 0.539 & 0.070 \\ -0.032 & 0.084 & 0.567 \end{bmatrix} \begin{bmatrix} 0.2 & 0.01 & 0.1 \\ 0.01 & 0.2 & 0.01 \\ 0.1 & 0.01 & 0.3 \end{bmatrix}$$

$$P_{1|1} = \begin{bmatrix} 0.111 & 0.004 & 0.051 \\ 0.004 & 0.108 & 0.022 \\ 0.051 & 0.022 & 0.168 \end{bmatrix}$$

16.3.3 EKF Python Example

We will take a similar setup as before, with a few values modified, and generate the Python code required. For this simulation, we place the noise only in the process equations and the observation. It is also reasonable to consider placing the noise in the control inputs as well. Assume that you start the robot with pose $[0, 0, 0]$ and you know this is exact so

$$P_{0|0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Let the process noise and measurement noise covariances be

$$V = \begin{bmatrix} 0.025^2 & 0 & 0 \\ 0 & 0.025^2 & 0 \\ 0 & 0 & 0.025^2 \end{bmatrix}, \quad W = \begin{bmatrix} 0.85^2 & 0 & 0 \\ 0 & 0.85^2 & 0 \\ 0 & 0 & 0.85^2 \end{bmatrix}$$

and the control inputs be $\omega_1 = 1.5\sin(t/10)$, $\omega_2 = \cos(t/10)$. Take $\Delta t = 0.1$, $r = 4$, $L = 6$, and

$$h_k(x_k) = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix}, \quad H_k = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

To create the observation data we have a simulation:

```
N = 100
mu1, sigma1 = 0.0, 0.025
mu2, sigma2 = 0.0, 0.85
var1 = sigma1*sigma1
var2 = sigma2*sigma2
dt = 0.1
r = 4
dd = r*dt/2.0
L = 6
x = np.zeros((N,3))
z = np.zeros((N,3))
t = np.linspace(0, 10, 100)
w1 = 1.5*np.sin(t)
w2 = 1.0*np.cos(t)


k = 1
while (k<N):
  q = np.random.normal(mu1,sigma1,3)
  r = np.random.normal(mu2,sigma2, 3)
  x[k,0] = x[k-1,0] + dd*(w1[k]+w2[k])*cos(x[k-1,2]) + q[0]
  x[k,1] = x[k-1,1] + dd*(w1[k]+w2[k])*sin(x[k-1,2]) + q[1]
  x[k,2] = x[k-1,2] + dd*(w1[k]-w2[k])/L + q[2]
  z[k,0] = x[k,0] + r[0]
  z[k,1] = x[k,1] + r[1]
  z[k,2] = x[k,2] + r[2]
  k = k+1
```

The code to implement the Extended Kalman Filter is very similar to the regular Kalman filter. The only difference is the inclusion of the Jacobians for the process and observations. The observation is a linear relation, so we just use the Jacobian from the last example. The first plot the code generates is the time plots of simulation pose (blue line), observation of the pose (red dots) and the pose estimate via Kalman (green dots). The second plot is a workspace domain plot of x values against y values, with θ ignored.

```
H = np.array([[1,0,0],[0,1,0],[0,0,1]])
HT = H.T
V = np.array([[var1,0,0],[0,var1,0],[0,0,var1]])
W = np.array([[var2,0,0],[0,var2,0],[0,0,var2]])
P = np.zeros((N,3,3))
xf = np.zeros((N,3))
xp = np.zeros(3)
sp = np.zeros(3)

k = 1
while (k<N):
  xp[0] = xf[k-1,0] + dd*(w1[k]+w2[k])*cos(xf[k-1,2])
  xp[1] = xf[k-1,1] + dd*(w1[k]+w2[k])*sin(xf[k-1,2])
  xp[2] = xf[k-1,2] + dd*(w1[k]-w2[k])/L
  F1 = [1.0,0.0, -dd*(w1[k]+w2[k])*sin(xf[k-1,2])]
  F2 =[0,1,dd*(w1[k]+w2[k])*cos(xf[k-1,2])]
  F = np.array([F1,F2,[0,0,1]])
  FT = F.T
  pp = np.dot(F,np.dot(P[k-1],FT)) + V
  y = z[k] - np.dot(H,xp)
  S = np.dot(H,np.dot(pp,HT)) + W
  SI = linalg.inv(S)
  kal = np.dot(pp,np.dot(HT,SI))
  xf[k] = xp + np.dot(kal,y)
  P[k] = pp - np.dot(kal,np.dot(H,pp))
  k = k+1

t = np.arange(0,N,1)
plt.plot(t, x, 'b-', t,z,'r.', t, xf,'go')
plt.show()

plt.plot(x[:,0], x[:,1], 'b-',z[:,0], z[:,1] ,'r.', xf[:,0], xf[:,1],'go')
plt.show()
```

16.3.4 Mecanum EKF Example

Developing the Extended Kalman Filter for the Mecanum drive is basically the same process. The only thing to derive is the matrix F. Recalling (5.5):

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} + \frac{r\Delta t}{4} \begin{bmatrix} A\cos(\theta_k) - B\sin(\theta_k) \\ A\sin(\theta_k) + B\cos(\theta_k) \\ \frac{2}{(L_1+L_2)}C \end{bmatrix}$$

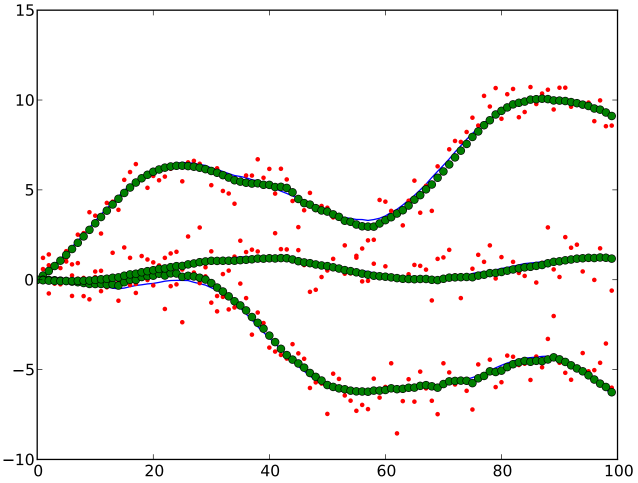


Fig. 16.15: The Extended Kalman Filter applied to the motion of a differential drive robot. Domain axis is time and vertical axis are the state variables. The simulation pose is given by the blue line, the observation of the pose given by the red dots and the pose estimate is given by the green dots.

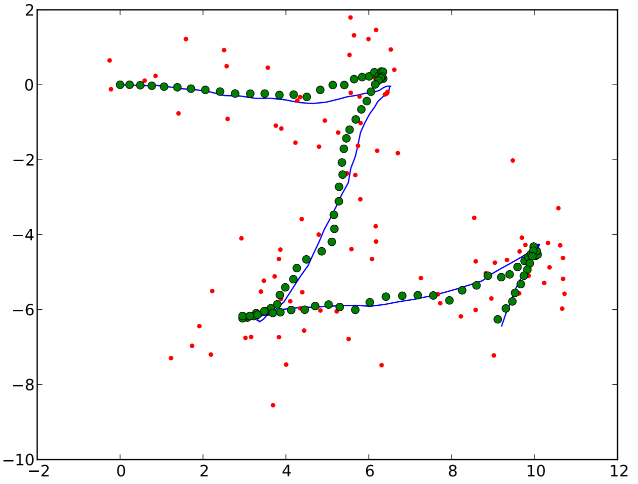


Fig. 16.16: The Extended Kalman Filter applied to the motion of a differential drive robot. This figure plots the y state variable against the x state variable with θ ignored. The simulation pose is given by the blue line, the observation of the pose given by the red dots and the pose estimate is given by the green dots.

where $A = (\omega_{FL,k} + \omega_{FR,k} + \omega_{BL,k} + \omega_{BR,k})$, $B = (-\omega_{FL,k} + \omega_{FR,k} + \omega_{BL,k} - \omega_{BR,k})$, and $C = (-\omega_{FL,k} + \omega_{FR,k} - \omega_{BL,k} + \omega_{BR,k})$. If we define $\xi_k = (x_k, y_k, \theta_k)^T$, $u_k = (\omega_{FL,k}, \omega_{FR,k}, \omega_{BL,k}, \omega_{BR,k})^T$ and reduce the k index by one, then the process can be written compactly as

$$\xi_k = f(\xi_{k-1}, u_k).$$

Computing the Jacobian of f:

$$F = \begin{bmatrix} 1 & 0 & \frac{r\Delta t}{4}\left[-A\sin(\theta_{k-1}) - B\cos(\theta_{k-1})\right] \\ 0 & 1 & \frac{r\Delta t}{4}\left[A\cos(\theta_{k-1}) - B\sin(\theta_{k-1})\right] \\ 0 & 0 & 1 \end{bmatrix}.$$

The rest of the process is identical to the differential drive examples.

16.3.5 Process Noise

We return to our original nonlinear process,

$$x_k = f(u_k, x_{k-1}) + v_k,$$

$$z_k = h(x_{k-1}) + w_k$$

where, $x_k \in R^n$, $u_k \in R^p$, $v_k \in R^n$, $w_k \in R^m$, v_k has variance V_k and w_k has variance W_k, and let

$$F_k = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \dots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}, H_k = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \frac{\partial h_1}{\partial x_2} & \dots & \frac{\partial h_1}{\partial x_n} \\ \frac{\partial h_2}{\partial x_1} & \frac{\partial h_2}{\partial x_2} & \dots & \frac{\partial h_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \\ \frac{\partial h_m}{\partial x_1} & \frac{\partial h_m}{\partial x_2} & \dots & \frac{\partial h_m}{\partial x_n} \end{bmatrix}$$

How to model the noise? The noise in the controls is the input and it drives the process noise. We assume here that we are going to gain all of our noise from the control noise and develop the model. We first assume that the control noise is drawn from a zero mean normal distribution with a covariance matrix R_k: $N(0, R_k)$. We also assume that the process noise depends on control noise: $f(u_k + N(0, R_k), x_k)$. The details are outside the scope of this text, but we have that a change of coordinates can relate the resulting process noise V_k to control noise R_k. The transformation that relates the noise term V_k to the covariance R_k is

$$V_k = G_k R_k G_k^T$$

where G_k is the Jacobian of g_k with respect to the control variables.

$$G_k = \begin{bmatrix} \frac{\partial g_1}{\partial u_1} & \frac{\partial g_1}{\partial u_2} & \dots & \frac{\partial g_1}{\partial u_p} \\ \frac{\partial g_2}{\partial u_1} & \frac{\partial g_2}{\partial u_2} & \dots & \frac{\partial g_2}{\partial u_p} \\ \vdots & \vdots & \vdots & \\ \frac{\partial g_n}{\partial u_1} & \frac{\partial g_n}{\partial u_2} & \dots & \frac{\partial g_n}{\partial u_p} \end{bmatrix}$$

Example with the DD model: The linearization of g with respect to the control:

$$G = \frac{\partial g}{\partial u_k} = \begin{pmatrix} \frac{r\Delta t}{2}\cos\theta_k & \frac{r\Delta t}{2}\cos\theta_k \\ \frac{r\Delta t}{2}\sin\theta_k & \frac{r\Delta t}{2}\sin\theta_k \\ \frac{r\Delta t}{2L} & -\frac{r\Delta t}{2L} \end{pmatrix}$$

We can map the control noise into process space via

$$V_k = \begin{pmatrix} \frac{r\Delta t}{2}\cos\theta_k & \frac{r\Delta t}{2}\cos\theta_k \\ \frac{r\Delta t}{2}\sin\theta_k & \frac{r\Delta t}{2}\sin\theta_k \\ \frac{r\Delta t}{2L} & -\frac{r\Delta t}{2L} \end{pmatrix} \begin{pmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{pmatrix} \begin{pmatrix} \frac{r\Delta t}{2}\cos\theta_k & \frac{r\Delta t}{2}\sin\theta_k & \frac{r\Delta t}{2L} \\ \frac{r\Delta t}{2}\cos\theta_k & \frac{r\Delta t}{2}\sin\theta_k & -\frac{r\Delta t}{2L} \end{pmatrix}$$

16.4 Particle Filters

Note: Lots to add for this section.

Particle filters are a form of sequential Monte Carlo methods used to address the hidden Markov chain and nonlinear filtering problems. Normally known as a non-parametric filter and separated from Kalman Filters, we will treat them together as forms of dynamic state estimation. It is essentially a form of a genetic algorithm applied to state estimation and is a popular form of non-parametric alternative to the Gaussian filters (Kalman Filter). Particle filters use a finite number of discrete variables to approximate the posterior. The basic algorithm for one step of the filter is given Algorithm *alg:particlefilter*. We will use the notation that x_k is a particle, X_k is the set of paticles, u_k is the control input and z_k is the observation at time step k.

Particle Filter. [thrun2005probabilistic]

```
Input $X_{k-1}$, $u_k$, $z_k$
Output $X_k$
Initialize $Y$, $X_k$ to zero.
for $m = 1$ to $M$ do
    Sample $x_k^m$ from $p(x_k|u_k, x_{k-1}^m)$
    $w_k^m = p(z_k|x_k^m)$
    $Y = Y + [x_k^m]$
end for
for $m = 1$ to $M$ do
    Select $x_k^i$ from $Y$ with probability $w_k^i$
    $X_k = X_k + [x_k^i]$
```

```
endfor
```

The standard algorithm for resampling selects the particle based on some probability which is proportional to the probability of the observation given the particle, often called the weight of the particle, w_k. One approach to this is in algorithm *alg:probsample*. In the Evolutionary Computation literature this is a form of Roulette selection. A low variance resample approach which numerically faster can be found in algorithm *alg:lowvariancesample*. To obtain good state estimates, it may be necessary to use many particles which can require significant computation. This is the tradeoff when compared to Gaussian filters (Kalman Filter).

Resample algorithm. Uses the particle weights to set the sampling probability.

```
Input $Y$ (particle set), $w^i$, $w = \sum w^i$
Output $X$ (resampled particle set)
Initialize $X$ to zero
for $m = 1$ to $M$ do
    Generate $r$, random number from uniform distribution $(0, w)$
    $s = 0, i = 0$
    while $s < r$ do
        $s = s + w^i$
        $i = i + 1$
    end while
    $X = X + \{x_i\}$
end for
```

```python
k = 1
while (k<N):
    j=0
    ws = 0
    while (j < M):
        q = np.random.normal(mu1,sigma1,3)
        pftemp[j,0] = pf[j,k-1,0] + dd*(w1[k]+w2[k])*cos(x[k-1,2]) + q[0]
        pftemp[j,1] = pf[j,k-1,1] + dd*(w1[k]+w2[k])*sin(x[k-1,2]) + q[1]
        pftemp[j,2] = pf[j,k-1,2] + dd*(w1[k]-w2[k])/L + q[2]
        dist = distance(pftemp[j],z[k])
        weight[j] = (1.0/sqrt(2.0*pi))*exp(-(dist/(2*sigma2**2)))
        ws = ws + weight[j]
        j = j+1
    j = 0
    while (j < M):
        i = 0
        wsum = weight[0]
        rval = ws*rnd.ranf()
        while (wsum < rval):
            i = i+1
            wsum = wsum + weight[i]
```

(continues on next page)

(continued from previous page)

```
        pf[j,k,:] = pftemp[i,:]
        mean[k,:] = mean[k,:] + pf[j,k,:]
        j = j+1
    mean[k,:] = (1.0/M)*mean[k,:]
    k = k+1
```

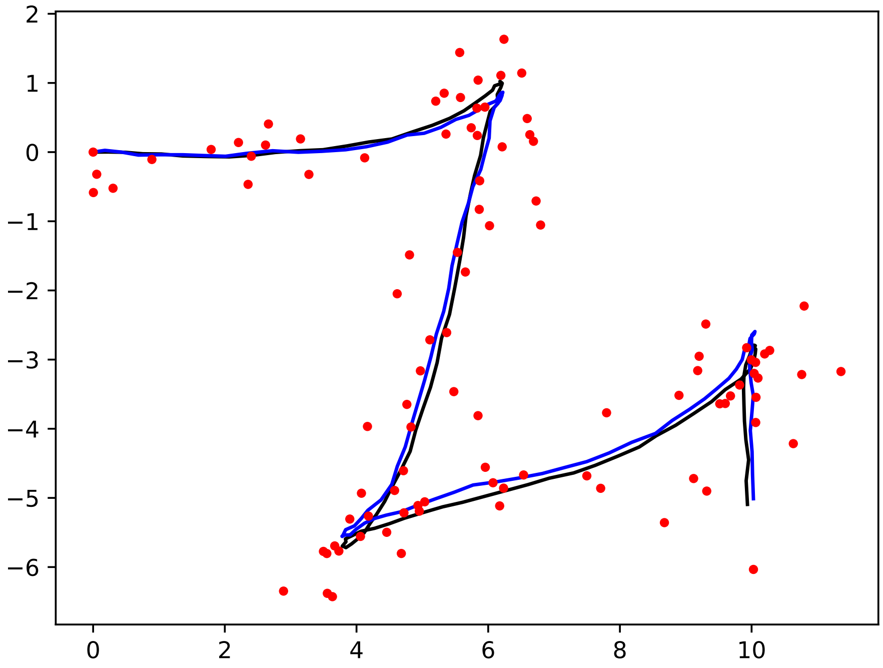


Fig. 16.17: The Particle Filter applied to the motion of a differential drive robot using the same dynamics as EKF example above. The simulation pose is given by the blue line, the observation of the pose given by the red dots and the pose estimate is given by the black line. 50 particles are used and the average is the pose estimate.

Low variance resample algorithm. This algorithm runs faster than the previous resample approach.

```
**Input** $Y$ (particle set), $w^i$, $w = \sum w^i$
**Output** $X$ (resampled particle set)
Initialize $X$ to zero
$r = rand(0, 1/M)$
$c = w^0$
$i = 1$
**for** $m = 1$ to $M$ **do**
```

```
    $u = r + (m-1)/M$
    while $c < u$ do
        $i = i + 1$
        $c = c + w^i$
    end while
    $X = X + \{x_i\}$
end for
```

The example particle filter above *alg:particlefilter* uses a fixed population size. Since particle filters are closely related to evolutionary algorithms, we can adapt them to state estimation. The particle filter here has two stages:

1. **Dynamics Update** Sample from the particle set to produce a temporary particle set. This advances the dynamics like the first step in the Kalman Filter. In the first stage, one can produce any number of sample particles.
2. **Observation Update** Resample based on the measurement to produce final particle set. This stage, the observation is used to select particles. The particles are selected based on the probability of the observation based on the particle. This stage can reduce the number of particles if needed. For example, this step can downsample to keep a fixed population size.

```
k = 1
while (k<N):
    for i in range(P):
        j = 0
        ws = 0
        while (j < M):
            q = np.random.normal(mu1,sigma1,3)
            pftemp[j+i*M,0] = pf[j,k-1,0] + dd*(w1[k]+w2[k])*cos(x[k-1,2]) + q[0]
            pftemp[j+i*M,1] = pf[j,k-1,1] + dd*(w1[k]+w2[k])*sin(x[k-1,2]) + q[1]
            pftemp[j+i*M,2] = pf[j,k-1,2] + dd*(w1[k]-w2[k])/L + q[2]
            weight[j+i*M] = distance(pftemp[j],z[k])
            ws = ws + weight[j+i*M]
            j = j+1
    j = 0
    while (j < M):
        ind = np.argsort(weight)
        pf[j,k,:] = pftemp[ind[j],:]
        mean[k,:] = mean[k,:] + pf[j,k,:]
        j = j+1
    mean[k,:] = (1.0/M)*mean[k,:]
    k = k+1
```

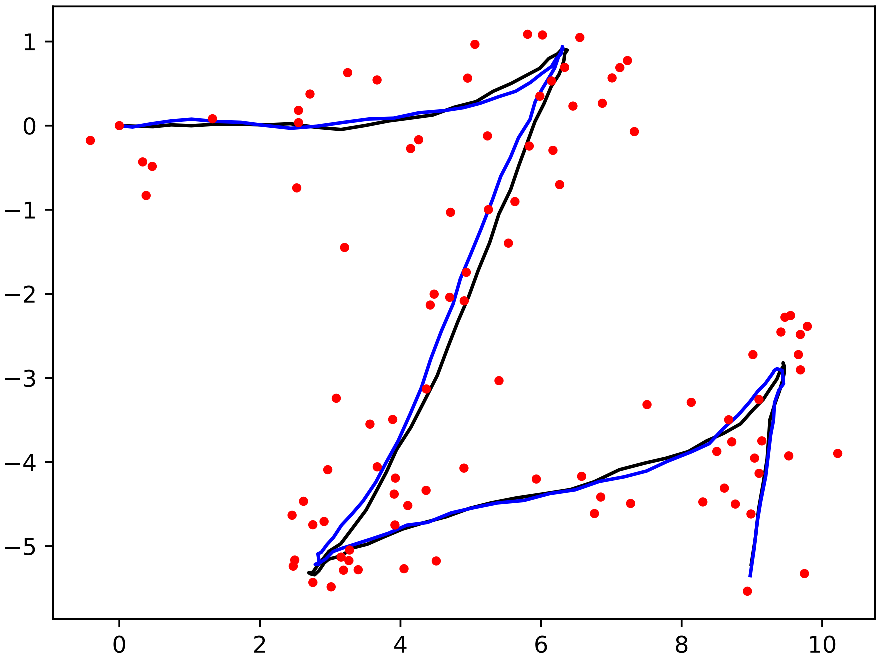


Fig. 16.18: The second Particle Filter applied to the motion of a differential drive robot as above. This filter double samples the physics, sorts the candidate particles and enforces a rank selection to reduce to required population size. The simulation pose is given by the blue line, the observation of the pose given by the red dots and the pose estimate is given by the black line. 50 particles are used and the average is the pose estimate.

16.5 Problems

1. Basic Kalman Filter. Let

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad F = \begin{bmatrix} 0 & 0.1 \\ -0.02 & 0.2 \end{bmatrix}, \quad G_k u_k = \begin{bmatrix} 0 \\ 2 * \sin(k/25) \end{bmatrix},$$

$$H = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad V = \begin{bmatrix} 0.05^2 & 0 \\ 0. & 0.05^2 \end{bmatrix}, \quad W = 0.25^2,$$

$$x(0) = \begin{bmatrix} 0.025 \\ 0.1 \end{bmatrix}, \quad P(0) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

   Apply the Kalman Filter process to compute 100 iterations and plot them. Hint: run the simulation to create your observation data z and then run your Kalman Filter.

2. Assume that one has three different measurements for the location of some object. The three measurements with the covariances are

$$(10.5, 18.2), \quad \begin{pmatrix} 0.1 & 0.01 \\ 0.01 & 0.15 \end{pmatrix}; \quad (10.75, 18.0), \quad \begin{pmatrix} 0.05 & 0.005 \\ 0.005 & 0.05 \end{pmatrix};$$

$$(9.9, 19.1), \quad \begin{pmatrix} 0.2 & 0.05 \\ 0.05 & 0.25 \end{pmatrix}.$$

   Fuse this data into one measurement and provide an estimate of the covariance.

3. Run a simulation on

$$\begin{aligned} \dot{x} &= y \\ \dot{y} &= -\cos(x) + 0.5\sin(t) \end{aligned}$$

   adding noise to the x and y components (with variance = 0.2 on each). Let $\Delta t = 0.1$. Assume that you can observe the first variable, x, with variance 0.25. Record the observations. Write a program to run the EKF on the observed data and compare the state estimate to the original values.

4. Differential Drive - EKF. The motion equations for a differential drive robot are given below. Assume that the wheels are 5cm in radius and the wheelbase is 12cm. Recall that the kinematics for this is (r = radius, L = wheelbase):

$$\dot{x} = \tfrac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\cos(\theta)$$

$$\dot{y} = \tfrac{r}{2}(\dot{\phi}_1 + \dot{\phi}_2)\sin(\theta)$$

$$\dot{\theta} = \tfrac{r}{2L}(\dot{\phi}_1 - \dot{\phi}_2)$$

   Select $\Delta t = 0.2$ (time increment) and convert to discrete equations. After conversion, assume the covariance of the state transition is V. Also assume that you have a local GPS system that gives (x, y) data subject to Gaussian noise with covariance W. The units on the noise are given in cm. If you want to use meters then you will need to divide your noise by 100.

   a. Starting at $t = 0$, $x = 0$, $y = 0$, $\theta = 0$, predict location when wheel velocities are:

```
t=0 -> 5:  omega1 = omega2 = 3 (rads/time),
t=5 -> 6:  omega1 = - omega2 = 1,
t=6 -> 10: omega1 = omega2 = 3,
t=10 -> 11:  - omega1 = omega2 = 1,
t=11 -> 16: omega1 =  omega2 = 3,
```

assuming that you have Gaussian noise in the process that is described by:

$$‘V = \begin{bmatrix} .05 & .02 & 0.01 \\ .02 & .05 & 0.01 \\ 0.01 & 0.01 & .1 \end{bmatrix}‘$$

b. Write out the formulas for the Extended Kalman Filter.

c. Apply an Extended Kalman filter to the motion simulation above to track the location of the vehicle. Observations can be simulated by using previous simulation data as actual data, i.e. use this as the observed data (z_k). Parameters:

$$x_{0|0} = (0,0,0), \quad V = \begin{bmatrix} .05 & .02 & 0.01 \\ .02 & .05 & 0.01 \\ 0.01 & 0.01 & .1 \end{bmatrix},$$

$$W = \begin{bmatrix} .08 & .02 \\ .02 & .07 \end{bmatrix}, \quad P_{0|0} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0.5 \end{bmatrix}.$$

d. Output the x-y locations on a 0.5 sec grid and compare in a plot.

e. The covariance matrix P gives the uncertainly ellipse for the location of the robot. Plot 5 ellipses along the path. This ellipse has major and minor axes given by the eigenvectors of P and the axes lengths are given by the associated eigenvalues. Matplotlib can plot an ellipse, click here.

5. Assume that you have a differential drive robot located in a lab with special landmarks placed in various locations around the lab. Also assume that your robot has a forward looking stereo vision system which can determine the distance and angle off the forward direction (relative to the robot) of the landmark. You don’t know the locations of the landmarks and the stereo system can only see the landmarks if the angle off of the front is less than 75°.

   a. Write the equations for the apriori EKF step (f_k) for some process noise covariance V_k.

   b. Assuming that the error of the angular measurement is 2 degrees in standard deviation - when you can observe the landmark, and the distance measurement error is 5 percent; what is the observation formula (h_k) and the error W_k?

   c. What are the linearizations of f and h?

   d. What are the aposteriori formulas? Don’t forget about the conversions from the robot (sensor) coordinates to the global or map coordinates.

   e. Write out the EKF process to track the location of the robot and the discovered landmarks. You should assume that you start at (0,0,0).

CHAPTER SEVENTEEN

PLANNING

Comment here about motion planning . . .

17.1 Potential Functions

A potential function is a differentiable real-valued function $U : \mathbb{R}^m \to \mathbb{R}$. We may think of it as an energy and thus the gradient is a force. The gradient

$$\nabla U(q) = \begin{bmatrix} \frac{\partial U}{\partial q_1} \\ \frac{\partial U}{\partial q_2} \\ \cdots \\ \frac{\partial U}{\partial q_n} \end{bmatrix} = \vec{F}$$

The gradients can be used to act on the robots like forces do on charged particles.

The vector fields (gradients) may be used to pull a robot to a particular goal or push a robot away from an obstacle.

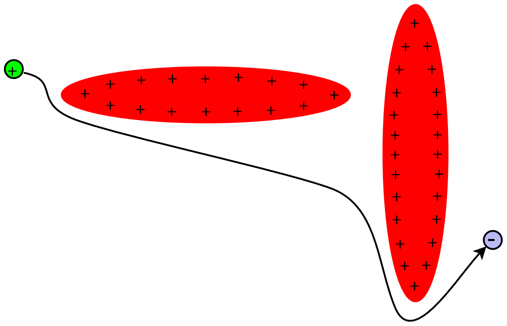


Fig. 17.1: Potential function navigation.

Vectors are seen as velocity not forces so this is a first order system.

The robot can move downhill using gradient descent:

$$\dot{c}(t) = -\nabla U(c(t)),$$

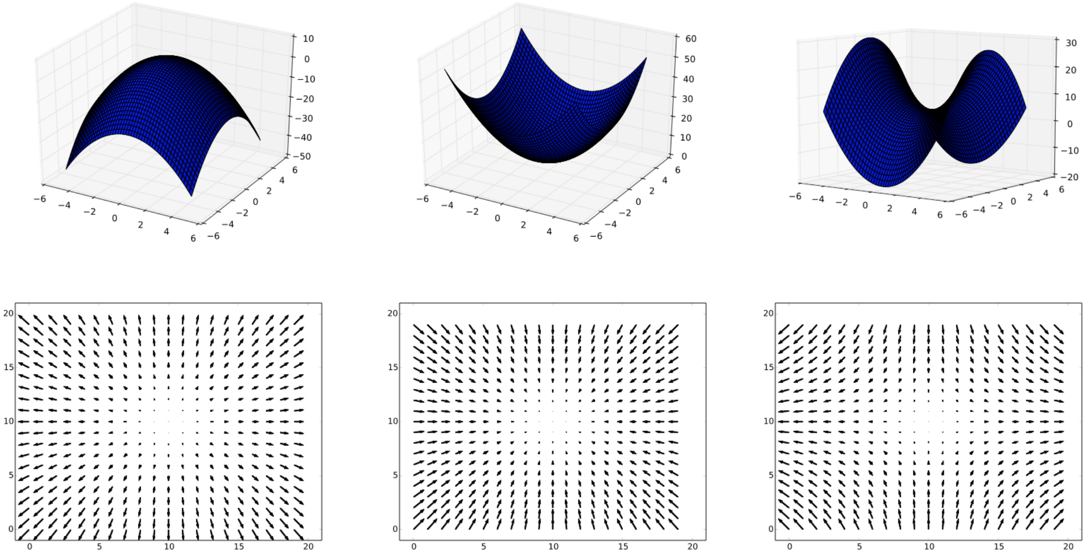
$$\frac{dx}{dt} = -\frac{\partial U}{\partial x}$$
$$\frac{dy}{dt} = -\frac{\partial U}{\partial y}$$

$\nabla U(q)$, q_{start} Sequence $q_1, q_2, q_3, \ldots, q_n$ $q(0) = q_{\text{start}}$ $i = 0$ $q(i+1) = q(i) - \alpha(i)\nabla U(q(i))$ $i++$

It will stop when it reaches a critical point, q^*: $\nabla U(q^*) = 0$. This point is a maximum, minimum or saddle point. It depends on the eigenvalues of the Hessian

$$H(U) = \begin{bmatrix} \dfrac{\partial^2 U}{\partial q_1^2} & \cdots & \dfrac{\partial^2 U}{\partial q_1 \partial q_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial^2 U}{\partial q_n \partial q_1} & \cdots & \dfrac{\partial^2 U}{\partial q_n^2} \end{bmatrix}$$

The Hessian is symmetric so the eigenvalues are real. Thus we get:



17.1.1 Example Potential Functions

Provide an example of an attractive potential function to the point (5,6).

$$U_a = (x-5)^2 + (y-6)^2$$

The gradient can be found: $\nabla U =< 2x - 10, 2y - 12 >$. Does $-\nabla U$ point to (5,6)? Pick a random point, (2,3). The vector from (2,3) to (5,6) is $< 3, 3 >$. The negative of the gradient, $-\nabla U$ at (2,3) is $< 6, 6 >$ which is $2 < 3, 3 >$ which works for this point and is easy to show in general. The graph is shown in Fig. 17.2.

Next we write down a repulsive potential function for an ellipse. The general equation of an ellipse is $(x-h)^2/a^2 + (y-k)^2/b^2 = 1$, and for this example we will select $a = 1$, $b = 2$, $h = 3$, $k = 4$. A repulsive function would be one that the gradient points away from.

An example of a repulsive potential:

$$U_r = \frac{1}{(x-3)^2 + (y-4)^2/4 - 1}$$

The graph of this function is shown in Fig. 17.3.

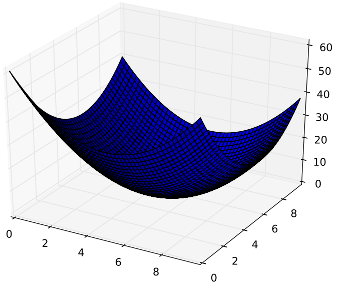


Fig. 17.2: Attractive potential function.

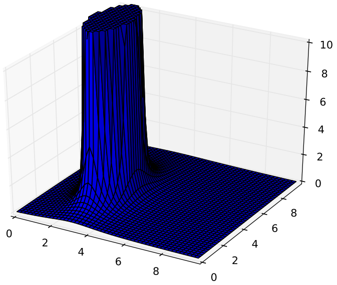


Fig. 17.3: Repulsive potential function.

17.1.2 Constructing Potentials

As suggested above, we will construct the potential functions from two basic types (Fig. 17.2, Fig. 17.3):

- Attractive Potential, denoted by $U_{\text{att}}(q)$, and
- Repulsive Potentials, denoted by $U_{\text{rep}}(q)$.

The full potential function will then be a combination of the two basic types. We will begin by just summing the potentials. This is the easiest approach but as you will see does not scale to multiple objects effectively. Using just addition, simple potential functions may be constructed from these:

$$U(q) = U_{\text{att}}(q) + U_{\text{rep}}(q)$$

And more complicated functions may be constructed via

$$U(q) = U_{\text{att}}(q) + \sum_i U_{\text{rep}\, i}(q)$$

We also assume that the outer boundary is not critical and so we ignore outer boundary effects. Later we will be able to include the boundary.

Attractive Potential A very simple function to use for the attractive potential is

$$U_{\text{att}} = k_0 \left[(x - x_0)^2 + (y - y_0)^2\right]$$

where (x_0, y_0) is the location of the goal. The value k_0 selects how steep the function walls are and thus changes the magnitude of the resulting gradient. This is the force pushing the object to the goal. We will balance k_0 with the constants of the other functions to gain an effective potential function surface. See Fig. 17.2.

Repulsive Potential A single repulsive potential can be formed by modifying the attractive potential. Assume that you can enclose the obstacle in a circle $(x - x_0)^2 + (y - y_0)^2 = r^2$ for some radius r and some center (x_0, y_0). Let

$$\rho = (x - x_0)^2 + (y - y_0)^2 - r^2$$

The function ρ is zero on the boundary of the circle and is positive outside the circle. It is a paraboloid that opens up. Then the repulsive potential is can be formed from γ/ρ or

$$U_{\text{rep}} = \frac{\gamma}{(x - x_0)^2 + (y - y_0)^2 - r^2}$$

U_{rep} is a function that goes to infinity at you approach the circle. See Fig. 17.3. The term γ is the strength of the field. It is a parameter which can be varied to adjust the relative force exerted by the repulsive field. It can shape the robot path an sometimes avoid local extremals.

Some authors like to shut down the repulsive potential by subtracting off a constant so it is zero outside a larger circle:

$$U_{\text{rep}} = \begin{cases} \dfrac{\gamma}{(x - x_0)^2 + (y - y_0)^2 - r^2} - \sigma & \text{for } \rho < \frac{\gamma}{\sigma} \\ 0 & \text{for } \rho \geq \frac{\gamma}{\sigma}. \end{cases}$$

Because there are quadratic functions involved, the growth can excessive. One way to deal with large values is to use conic potentials instead of quadratic potentials. Let $q = (x, y)$ and $q_{\text{goal}} = (x, y)_{\text{goal}}$. The conic potential:

$$U_{\text{att}} = \gamma d(q, q_{\text{goal}})$$

The gradient is then

$$\nabla U(q) = \frac{\gamma}{d(q, q_{\text{goal}})}(q - q_{\text{goal}})$$

This presents numerical issues due to the discontinuity, so normally one uses $U(q) = \gamma d^2(q, q_{\text{goal}})$

$$\nabla U(q) = \gamma(q - q_{\text{goal}})$$

Velocity is too large far away and will overwhelm other fields. We use a linear velocity for far field and quadratic velocity for near field. The switch over point is at distance d^*_{goal}:

$$U_{\text{att}}(q) = \begin{cases} (1/2)\gamma d^2(q, q_{\text{goal}}), & d(q, q_{\text{goal}}) \leq d^*_{\text{goal}}, \\ d^*_{\text{goal}}\gamma d(q, q_{\text{goal}}) - (1/2)\gamma (d^*_{\text{goal}})^2, & d(q, q_{\text{goal}}) > d^*_{\text{goal}}, \end{cases}$$

which gives

$$\nabla U_{\text{att}}(q) = \begin{cases} \gamma(q - q_{\text{goal}}), & d(q, q_{\text{goal}}) \leq d^*_{\text{goal}}, \\ d^*_{\text{goal}}\gamma \frac{(q - q_{\text{goal}})}{d(q, q_{\text{goal}})}, & d(q, q_{\text{goal}}) > d^*_{\text{goal}}, \end{cases}$$

The repulsive potential is the same as the one above. We rewrite the expression in slightly different notation where the γ/σ term is replaced by $1/Q^*$ which is a measure of distance away from the obstacle boundary. Essentially Q^* is the cutoff distance for when we no longer express the repulsive potential field. The formula in the new notation is

$$U_{\text{rep}}(q) = \begin{cases} (1/2)\eta \left(\frac{1}{D(q)} - \frac{1}{Q^*}\right), & D(q) \leq Q^*, \\ 0, & D(q) > Q^* \end{cases}$$

This becomes a very complicated formula when the obstacles are no longer circles. It is very difficult to arrive at a formula for the closest obstacle. Finding equidistance lines is a whole issue alone. We will address this when we discuss Voronoi decomposition.

Note that placing repulsive potentials in can change the location of the minimum that you have setup through the attractive potential. This is one reason we go to the trouble of placing a cutoff on the obstacle potentials. A simple one dimensional example can demonstrate. Assume you want your minimum to be at $x = 0$, so you try $U_a = x^2$. Next you place in a repulsive potential at $x = 5$, $U_r = |x - 5|^{-1}$. Combining we have

$$U = x^2 + \frac{1}{|x-5|}.$$

Compute the derivative and set to zero:

$$\frac{dU}{dx} = 2x - \frac{\text{sign}(x-5)}{|x-5|^2} = 0.$$

For $x < 5$ we have

$$\frac{dU}{dx} = 2x - \frac{1.0}{|x-5|^2} = 0$$

which can solved: $2x(x-5)^2 = 1$ or $x \approx 0.02$. No longer at $x = 0$.

Summary:

$$U(q) = U_{\text{att}}(q) + U_{\text{rep}}(q)$$

$$U_{\text{att}}(q) = \begin{cases} (1/2)\gamma d^2(q, q_{\text{goal}}), & d(q, q_{\text{goal}}) \leq d^*_{\text{goal}}, \\ d^*_{\text{goal}}\gamma d(q, q_{\text{goal}}) - (1/2)\gamma (d^*_{\text{goal}})^2, & d(q, q_{\text{goal}}) > d^*_{\text{goal}}, \end{cases}$$

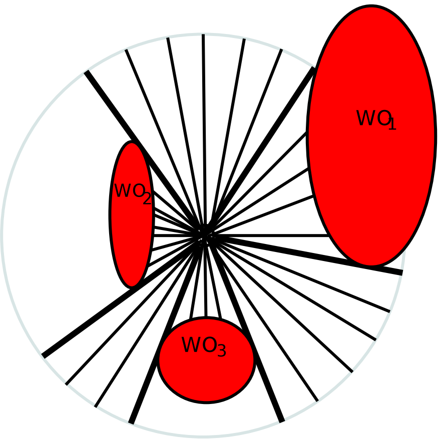


Fig. 17.4: LIDAR Range map.

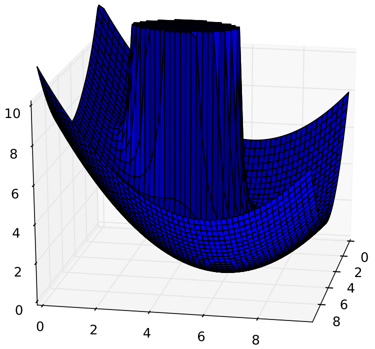
$$U_{\text{rep}}(q) = \begin{cases} (1/2)\eta\left(\frac{1}{D(q)} - \frac{1}{Q^*}\right), & D(q) \leq Q^*, \\ 0, & D(q) > Q^* \end{cases}$$

The distance $D(q)$ can be determined from a LIDAR sweep if the robot is located at q.

To compute the potential function, you need to know all of the distances, not just from a single point q.

Often the environment is represented on a grid which can simplify the planning process in some cases. Our first step is to remove the analytic repulsive potential and replace it with a discrete method known as the Brushfire algorithm. This can remove the problems related to finding repulsive potentials that don't overwhelm the attractive potential.

$$U = (x-5)^2 + (y-6)^2 + \frac{\gamma}{(x-3)^2 + (y-4)^2/4 - 1}$$



The equations of motion that generate the path are

$$\frac{dx}{dt} = -\frac{\partial U}{\partial x} = -2(x-5) + \frac{2\gamma(x-3)}{[(x-3)^2+(y-4)^2/4-1]^2}$$
$$\frac{dy}{dt} = -\frac{\partial U}{\partial y} = -2(y-6) + \frac{\gamma(y-4)/2}{[(x-3)^2+(y-4)^2/4-1]^2}$$

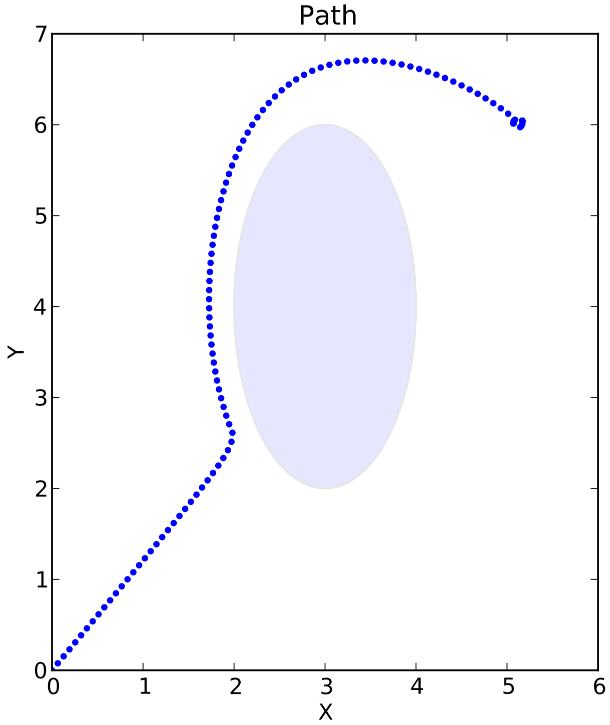
This is solved by using a discrete approach which is known as steepest descents.

$$x_{n+1} = x_n - \eta\left\{2(x_n-5) - \frac{2\gamma(x_n-3)}{[(x_n-3)^2+(y_n-4)^2/4-1]^2}\right\}$$
$$y_{n+1} = y_n - \eta\left\{2(y_n-6) - \frac{\gamma(y_n-4)/2}{[(x_n-3)^2+(y_n-4)^2/4-1]^2}\right\}$$

Note that γ is a measure of field strength and η is a step size parameter. Moving these two around is useful to adjust for better computed paths.

```
import numpy as np
import scipy as sp
import pylab as plt
from matplotlib.patches import Ellipse

NP = 200
t = np.arange(0,NP,1)
x = np.zeros((NP))
y = np.zeros((NP))
x[0] = 0.0
y[0] = 0.0
gamma = 1.0
zeta = 0.1

for i in range(1,NP):
  v = gamma/(((x[i-1]-3.0)**2 + ((y[i-1]-4.0)**2)/4 -1.0)**2)
  vx = 2.0*(x[i-1]-5.0) - 2*(x[i-1]-3)*v
  vy = 2.0*(y[i-1]-6.0) - 0.5*(y[i-1]-4)*v
  vn = np.sqrt(vx*vx+vy*vy)
  vx2 = vx/vn
  vy2 = vy/vn
  print v, -vx2, -vy2
  x[i] = x[i-1] - zeta*vx2
  y[i] = y[i-1] - zeta*vy2

ell = Ellipse((3.0,4.0),2,4,0)
a = plt.subplot(111, aspect='equal')
ell.set_alpha(0.1)
a.add_artist(ell)

plt.plot(x,y, 'b.')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Path')
plt.show()
```

Let the domain be the square $0 \le x \le 10$, $0 \le y \le 10$.



- Place the start position at (1,1)
- Place the goal position at (9,8)
- Obstacle 1: disk centered at (4,3) of radius 2.5.
- Obstacle 2: disk centered at (7,8) of radius 1.

What is the potential function?

Obstacles in red…

What is the attractive potential? Let $q = (x, y)$,

$$U_a(q) = (x-9)^2 + (y-8)^2.$$

What is the repulsive potential?

$$U_r(q) = \frac{\gamma_1}{(x-4)^2 + (y-3)^2 - 2.5^2} + \frac{\gamma_2}{(x-7)^2 + (y-8)^2 - 1^2}$$

The resulting potential is the sum:

$$U = U_a(q) + U_r(q) = (x-9)^2 + (y-8)^2 +$$

$$\frac{\gamma_1}{(x-4)^2 + (y-3)^2 - 2.5^2} + \frac{\gamma_2}{(x-7)^2 + (y-8)^2 - 1^2}$$

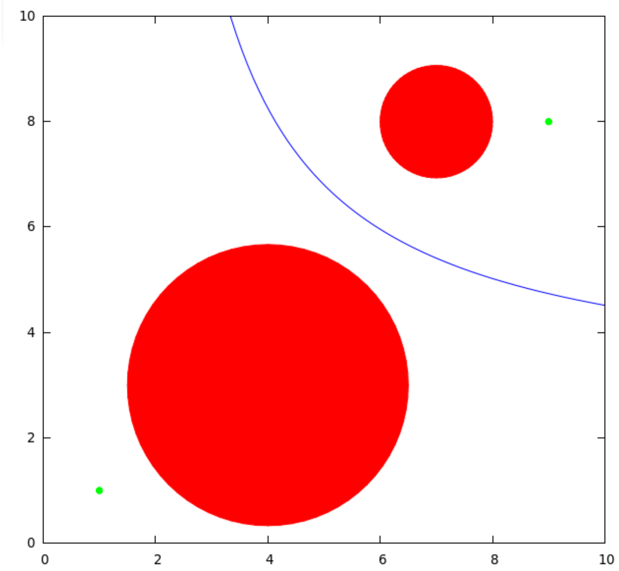


Fig. 17.5: Two obstacles and the resulting equal distance line.

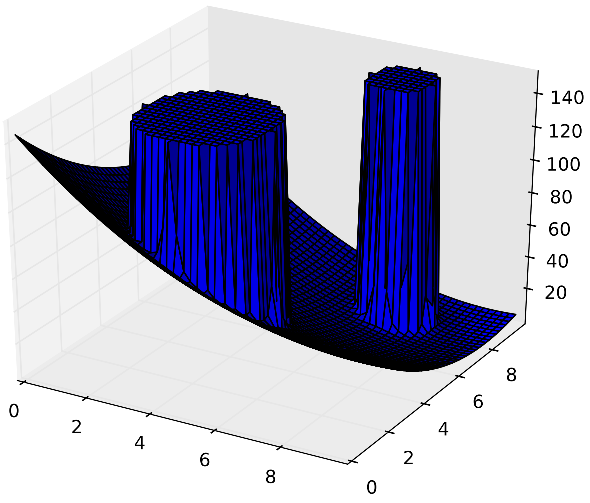


Fig. 17.6: Potential function surface.

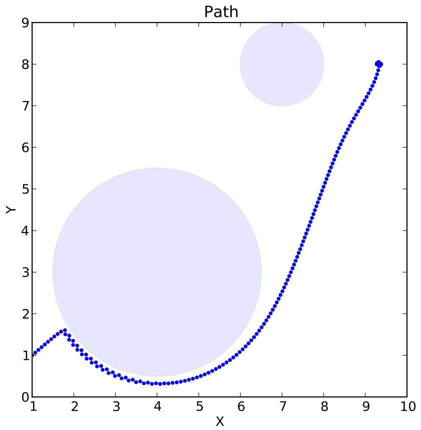


Fig. 17.7: Resulting navigation.

These simple functions work well for simple domains. However, when the obstacles increase, then the simple potentials cease to be effective. A more methodical approach is needed.

If you looked carefully at the path in Fig. 17.7, you will notice that the path appears to oscillate when it gets near the large obstacle. Indeed this is what is happening. This oscillation is a direct result of the steepest descent algorithm is appears in many numerical optimization routines. The numerics will follow the steepest gradient and will oscillate back and forth along the steep walls. It will slowly average out traversing the mean path which will trace the valley floor, Fig. 17.8.

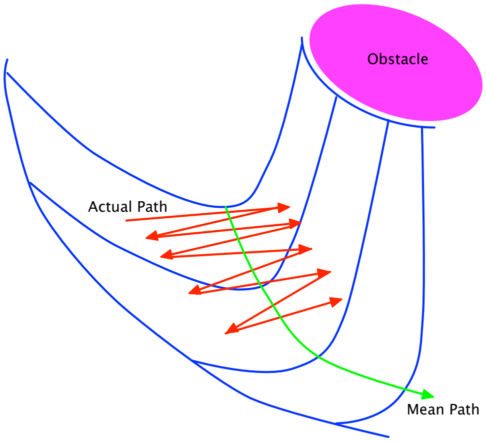


Fig. 17.8: Numerical Oscillation near steep gradients.

There is nothing particularly special regarding the functions we have presented. Our goal is to find a potential surface which can "navigate" a vehicle from start to finish. Getting familiar with the shapes and level sets of graphs can be very helpful. This can help one in the construction process. Typically we want our level set to track an obstacle boundary.

Construct a function which directs the craft onto the line $y = 2x + 3$. Then $U = (2x + 3 - y)^2$ will suffice. This function has a minimum along $y = 2x + 3$ and increases as you move away from the line.

Keep in mind that you must be very careful combining the functions since they can interact in very complex ways. You may have to have cutoff distances from obstacles to keep them from corrupting each other.

17.1.3 Higher Dimensions

One of the advantages of potential functions is that they scale to higher dimensions in a very efficient manner. We will start with three dimensions. The attractive and repulsive potentials follow the same pattern as we saw in two dimensions.

Construct an attractive potential for the the point x_0, y_0, z_0.

$$U_{att} = (x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2,$$

and

$$\nabla U_{att} = \langle 2(x - x_0), 2(y - y_0), 2(z - z_0)\rangle$$

Construct a repulsive potential for a spherical obstacle centered x_0, y_0, z_0 of radius R.

$$U_{rep} = \frac{\gamma}{(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - R^2}$$

and

$$\nabla U_{rep} = \frac{-2\gamma\langle (x - x_0), (y - y_0), (z - z_0)\rangle}{((x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - R^2)^2}$$

Build a function that can direct a drone to a landing pad. Assume the landing pad is at (0,0,0). We construct a cone centered at the landing pad which will "pull" the drone in. We can take a simple attractive function

$$U_{att} = x^2 + y^2 + \alpha z^2 = r^2 + \alpha z^2, \quad r = \sqrt{x^2 + y^2}$$

and then a vertical squeeze function

$$U_{att2} = (z - r)^2.$$

The resulting potential is $U = U_{att} + \gamma U_{att2}$,

$$U = r^2 + \alpha z^2 + \gamma (z - r)^2 = (1 + \gamma) r^2 + (\gamma + \alpha) z^2 - 2\gamma r z$$

$$= (1 + \gamma)(x^2 + y^2) + (\gamma + \alpha) z^2 - 2\gamma z\sqrt{x^2 + y^2}.$$

For the attractive function, the parameter α can be used to vary the relative strength in the z direction. In the squeeze function, the parameter γ can be used to adjust the strength of that field component.

17.2 Brushfire Algorithm

The Brushfire algorithm is used to compute the distances which are used in the repulsive potential. *It is not a planner!*. It will replace the $d(p, q)$ function above and can be part of other planning algorithms. Assume that your map is represented on a grid domain.

Four point neighbors have a distance which is the same as the Euclidean distance. Eight point connectivity includes diagonals, greater connectivity, but ignores diagonal distance.

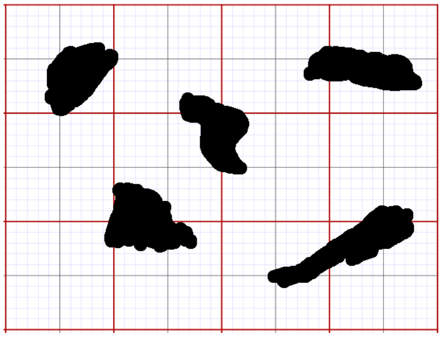


Fig. 17.9: A grid domain with obstacles. Normal practice is to label the cell as occupied if any part of the physical obstacle overlaps the cell.

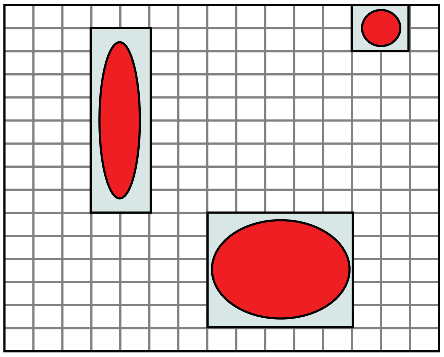


Fig. 17.10: Example of obstacle overlab with grid domain.

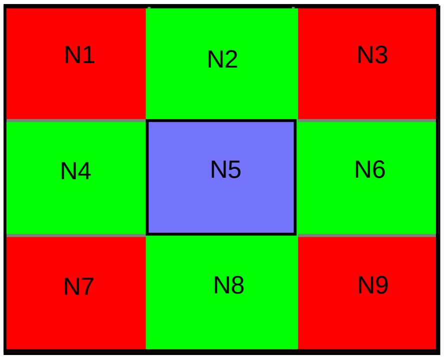


Fig. 17.11: Four and eight point connectivity to determine neighbors.

- Set free space pixels to zero.
- Set pixels which are occupied, even partially, by objects to 1
- Neighbor pixels (containing 0) to object pixels are set to 2
- Neighbors (containing 0) to pixels containing 2 set to 3, etc

Gradient map made by looking at the smallest value in your connectivity.

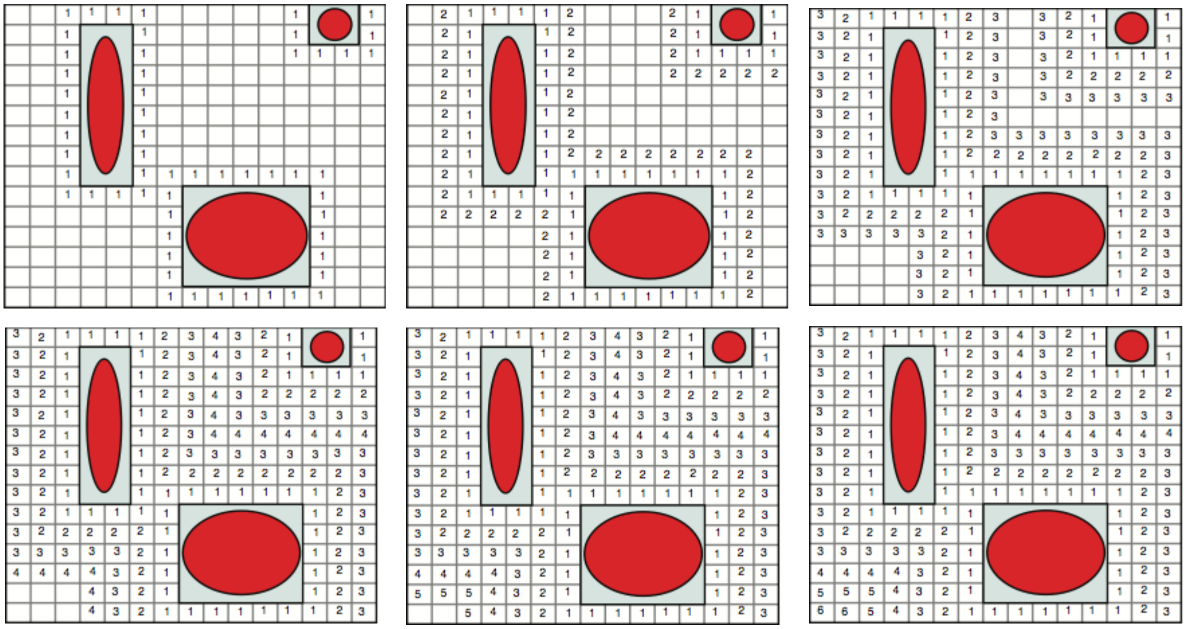


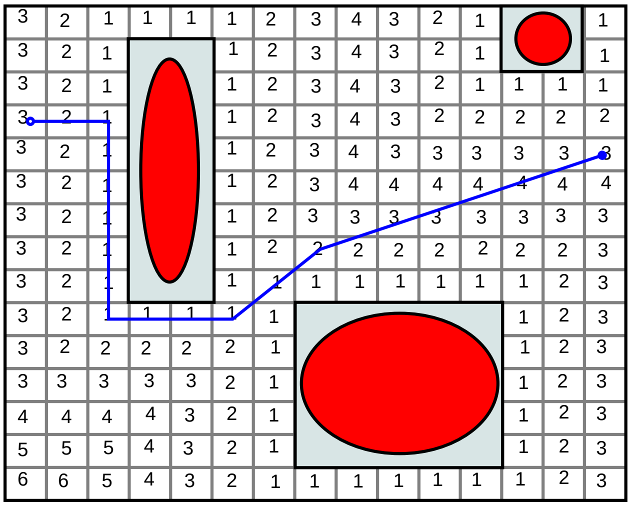
Fig. 17.12: Brushfire example

A gradient map can be produced at each pixel by finding the neighbor pixel with the largest value. Both distance and gradient are now available and thus a planning algorithm can use this to determine a path. Fig. 17.13 shows the Steepest Descent Path. This path may not be unique due to the discrete nature of the domain map. At each step, there can be multiple cells with the same value. One must have a selection process and different selection choices lead to different descent paths. Note that this process will generalize to any dimension.

17.2.1 Potentials and Brushfire

The Brushfire algorithm may be used to replace the repulsive potential. The attractive potential is still required to complete the routing. One may use any form of attractive potential. The idea is to use the atttractive potential to direct the robot to the goal. The Brushfire algorithm can be used to keep the robot from colliding with an obstacle. Since the Brushfire map includes distance to an obstacle, then the cells of greatest increase are in the direction away from the obstacle. This is a discrete negative gradient. In combination with the attractive potential can be used to route.

One approach to planning is given in Algorithm *alg:brushfire*. Assume that the domain is discretized and the current location of the robot is indexed by $q = (i, j)$. Also assume the goal location is $q_{\text{goal}} = (i^*, j^*)$. Call the Brushfire cell number for cell $q = (i, j)$, $b(q)$. The attractive potential in grid coordinates is



3	2	1	1	1	1	2	3	4	3	2	1			1
3	2	1			1	2	3	4	3	2	1			1
3	2	1			1	2	3	4	3	2	1	1	1	1
3	2	1			1	2	3	4	3	2	2	2	2	2
3	2	1			1	2	3	4	3	3	3	3	3	3
3	2	1			1	2	3	4	4	4	4	4	4	4
3	2	1			1	2	3	3	3	3	3	3	3	3
3	2	1			1	2	2	2	2	2	2	2	2	3
3	2	1			1	1	1	1	1	1	1	1	2	3
3	2	1	1	1	1	1						1	2	3
3	2	2	2	2	2	1						1	2	3
3	3	3	3	3	2	1						1	2	3
4	4	4	4	3	2	1						1	2	3
5	5	5	4	3	2	1						1	2	3
6	6	5	4	3	2	1	1	1	1	1	1	1	2	3

Fig. 17.13: Steepest Descent Path

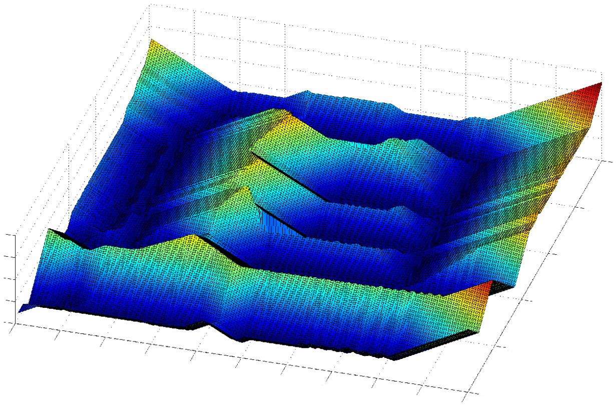


Fig. 17.14: Brushfire Surface

$U = [(i-i^*)^2 + (j-j^*)^2]/2$, so the gradient $\nabla U = (i-i^*, j-j^*) = q - q_{\text{goal}}$. We can combine Brushfire with the discrete potential function to obtain the Algorithm *alg:brushfire*.

Discrete potential function planner

```
Input A point robot with a tactile sensor and $D_{\text{min}}$.
Output A path to the goal.
while true do
    repeat
        Compute $q_{\text{goal}} - q = (h, k)$.
        Compute $z = \text{max}(h, k)$ and $\Delta q = (\text{int } h/z, \text{int } k/z)$
        Compute $q_{\text{new}} = q + \Delta q$
        Set $q_{\text{new}} \to q$
    until $q = q_{\text{goal}}$ or $b(q) = D_{\text{min}}$
if Goal is reached
then exit
Set $L$ equal to list of unvisited neighbor cells with $b(i, j) = D_{\text{min}}$
if L is empty, then conclude there is no path to goal.
repeat
    Select $q = (i, j) \in L$
    Compute $q_{\text{goal}} - q = (h, k)$.
    Compute $z = \text{max}(h, k)$ and $\Delta q = (\text{int } h/z, \text{int } k/z)$
    Compute $q_{\text{new}} = q + \Delta q$
    Set $L$ equal to list of unvisited neighbor cells with $b(i, j) = D_{\text{min}}$
    if L is empty, then conclude there is no path to goal.
until $q_{\text{goal}}$ is reached or $b(q_{\text{new}}) > D_{\text{min}}$
```

17.2.2 Dealing with discrete functions

How do we modify the potential function approach? Recall that we have

$$U(q) = U_{\text{att}}(q) + U_{\text{rep}}(q)$$

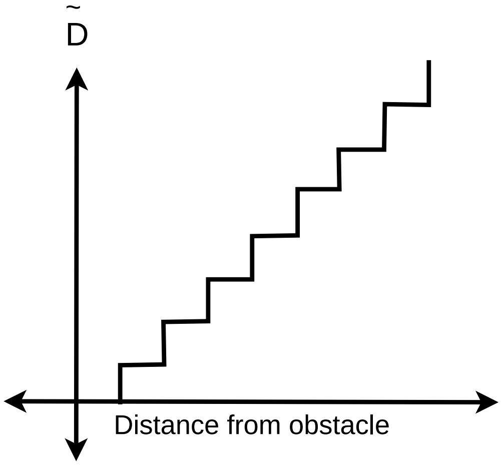
with the attractive potential as

$$U_{\text{att}}(q) = \begin{cases} (1/2)\gamma d^2(q, q_{\text{goal}}), & d(q, q_{\text{goal}}) \le d^*_{\text{goal}}, \\ d^*_{\text{goal}}\gamma d(q, q_{\text{goal}}) - (1/2)\gamma (d^*_{\text{goal}})^2, & d(q, q_{\text{goal}}) > d^*_{\text{goal}}, \end{cases}$$

and the repulsive potential as

$$U_{\text{rep}}(q) = \begin{cases} (1/2)\eta \left(\frac{1}{\tilde{D}(q)} - \frac{1}{Q^*}\right), & \tilde{D}(q) \le Q^*, \\ 0, & \tilde{D}(q) > Q^* \end{cases}$$

where $\tilde{D}$ is found from the Brushfire Map. The issue is that $\tilde{D}$ is not a continuous function. It is a piecewise constant function and so $\nabla\tilde{D}$ is zero on all of the interiors of the cells.[1]



The gradient can be estimated as the difference in cell values. Thus

$$\nabla U_{rep} = \left\langle \Delta\tilde{D}/\Delta x, \Delta\tilde{D}/\Delta y \right\rangle$$

Because the discrete distance function jumps, it can cause the path to oscillate back and forth along the normal direction to the obstacle.[2] Tuning the potential function can also be challenging. One may need to adjust weights in the sum:

$$aU_{\text{att}}(q) + bU_{\text{rep}}(q)$$

What one wants is motion orthogonal to the boundary of the obstacle.

Motion towards the obstacle is in the direction of the repulsive potential gradient, ∇U_{rep}, so we select motion orthogonal to the gradient, $\nabla U_{\text{rep}}^{\perp}$:

$$\text{Heading} = \lambda(1-d)\nabla U_{\text{att}} + \lambda d \nabla U_{\text{rep}}^{\perp}$$

where $d = D/Q^*$ and λ is positive "tunable" value. This gives a smooth transition to orthogonal motion. We still need to understand $\nabla U_{\text{rep}}^{\perp}$.

The orthogonal subspace $\nabla U_{\text{rep}}^{\perp}$ is a line. From this we need to select a direction. We can do this by projecting the gradient of the attractive potential onto the subspace:

$$\text{proj}(\nabla U_{\text{att}})_{\nabla U_{\text{rep}}^{\perp}} = \frac{\left(\frac{\partial U_{\text{att}}}{\partial x}\right)\left(\frac{\partial U_{\text{rep}}}{\partial y}\right) - \left(\frac{\partial U_{\text{att}}}{\partial y}\right)\left(\frac{\partial U_{\text{rep}}}{\partial x}\right)}{\|\nabla U_{\text{rep}}\|^2}\nabla U_{\text{att}}.$$

17.2.3 Local Minima Problem

Gradient descent will move towards a local minimum, but not necessarily the global minimum. Take the map with goal given by Fig. 17.15.

[1] Known as zero "almost everywhere".

[2] This is due to the switching on and off a large repulsive potential.

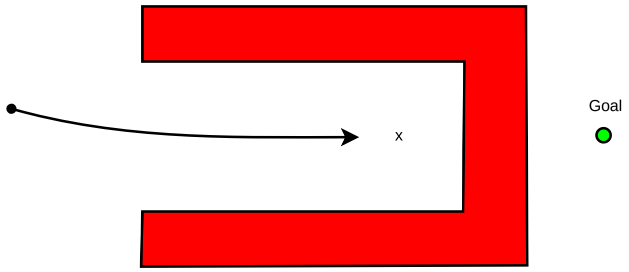


Fig. 17.15: Using an attractive potential function centered at the goal and a repulsive potential function based on distance from the obstacle, the robot will be attracted to some point *x* where it will stop. This point is a local minimum in the combined potential function (attractive and repulsive potentials combined).

There is a point, *x* inside the horseshoe where the attractive forces and repulsive forces balance giving rise to a local min for the combined potential function. The robot will stall at this point. There is not a simple fix here.

17.2.4 Maximum obstacle distance path

Some routing problems require the vehicle to keep a maximum distance from obstacles. For example, quadrotors are effected by ground and wall effects which can cause collisions and vehicle damage. Using the Brushfire and Wavefront algorithms together can be used to produce safe paths. The idea is to use the Brushfire algorithm to do the skeletalization of the domain. Then the Wavefromt algorithm searches the reduced path.

- Use Brushfire to find equidistance points or ridges and label ridge pixels
- Compute shortest path between start point and the ridge: start path
  - Use a Wavefront planner.
  - Set the start point as the wave start.
  - Stop when the wave hits the ridge.
  - Label start path pixels
- Compute shortest path between end point and the ridge: end path
  - Use a Wavefront planner.
  - Set the end point as the wave start.
  - Stop when the wave hits the ridge.
  - Label end path pixels.
- Back track along different segments in the path list to find global path
  - Starting at end pixel.
  - Apply wavefront to labeled pixels.
  - Stop wavefront when start pixel is found.

17.3 Problems

1. Let the domain be the rectangle $0 \leq x \leq 15$ and $0 \leq y \leq 10$. Place the start position at (0,5). Place the end position at (15,5). Assume you have a disk centered at (6,4) with radius 2 and a disk centered at (8,6) with radius 3. Find a potential function which can navigate the robot from the start to the end position.

   1. Plot the resulting path in Python with obstacles included in the map.
   2. Using STDR, move a robot along the path. Use a video screen capture program to record the results.
   3. Compare to the Wavefront approach.

2. Rework the previous problem with a Poisson navigation function.

3. Write a Python program to implement the Brushfire algorithm and extract the equidistance pixels. Display the equidistance pixel map. Demonstrate on a map with multiple obstacles.

Note: add all content from the old 416 notes (visibility maps, RRTs, etc).

CHAPTER EIGHTEEN

LOCALIZATION AND MAPPING

This chapter focuses on determining the location of a robot on a known map, the converse problem of building a map given a location, and when neither the location or map are involved known as SLAM - Simultaneous Localization and Mapping.

Note: these sections are a mix of converted slides and notes. Reorganization of the topics is also needed.

18.1 What is localization?

Using sensory information to locate the robot in its environment is the most fundamental problem to providing a mobile robot with autonomous capabilities. [Cox 1991]

Given a map of the environment, a sequence of actions and sensor measurements, our goal is to estimate the robot's position and error of the position estimate. There are two problem classes or types for localization we will study: **Position Tracking** and **Global Localization**. They emphasize two different but important aspects of localization.

Position tracking is one of the most common issues facing the beginning roboticist. Also called ego-motion determination is the task which given a known starting location, keeps track of the path or position of the robot over time. This can be a local method in that the absolute location of the robot may not be known and only the relative position from the starting point can be determined. Methods may employ motion sensing such as wheel encoders, inertial navigation sensors and optical flow algorithms. The focus can be purely on relative changes in position and not at all related to a map.

Global localization is the task of determining the position in a global coordinate system. This can arise from determination of initial pose for the position tracking problem or in recovery from localization system failure (also known as the kidnapped robot problem). Global localization may want to place the robot on a known map or begin the mapping process with a known location.

As we learned in the Filtering chapter, measurements of pose data is error prone. The system can be uncertain about global landmarks. [We observe we are 30 meters down the hall, but don't know which hallway.] All of this introduces error into both the position tracking and the global position of a robot. This requires some type of error reduction or filtering approach to be employed. In addition to noise, the landscape can change. One must take into account the dynamic environment and any changes that occur due to the robot.

Some of the localization approaches interact with maps. The structure of the map and the information stored in the map can strongly influence the results. Some approaches are metric in that they track pose data for all the actors. Other approaches are feature based which may only know the robot is near something. One may instrument the environment with tags and beacons. Or one may just learn interesting features and infer location.

Localization may employ active or passive approaches. Passive approaches just observes where active approaches means the robot moves to minimize some variable of the pose and goal. Multiple robots can be used to enhance localization by exchange of information as well as use the other robots for relative positioning.

18.2 Localization Belief

Note: This section under development.

The process of localization is stating a belief that the robot is found at a given location. There is ambiguity in our knowledge which implies this belief can be represented as a probability distribution. We can arbitrarily decide that this belief, the probability distribution, is a normal distribution or something more exotic. Having a single "bump" means we have one hypothesis of location, a *single hypothesis*. Having a distribution with multiple bumps means we have *multiple hypotheses* of the robot location.

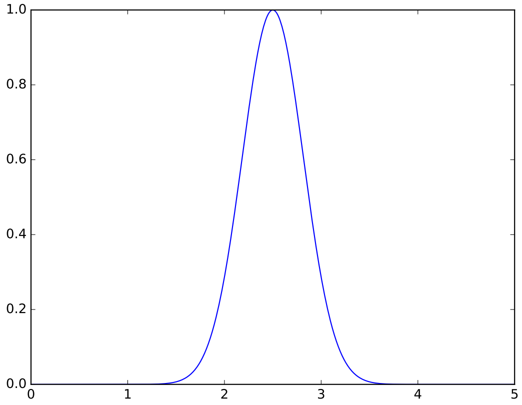


Fig. 18.1: Single Hypothesis

Having multiple hypotheses seems a bit odd at first, but actually arises. Imagine you have a Starbucks map - a map of a city that just shows Starbucks. Also assume you drive up to a Starbucks. Now compare to the map. You can now isolate your position to one of the n Starbucks locations on the map. This is an example of multiple hypotheses. Only until you receive additional information are you able to break the ambiguity. With the reduced information available to a robot, this situation arises when faced with vision system that use corners and walls to generate landmarks.

18.3 Maps

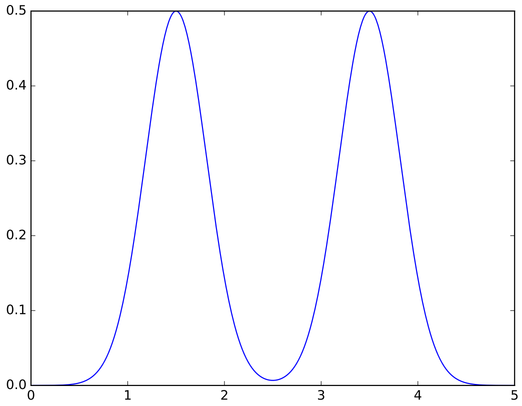


Fig. 18.2: Multihypothesis

Note: This section under development.

The representation of the environment is reflected in the map. We have discussed several methods earlier such as metric or continuous descriptions, discrete or grid descriptions and topological descriptions. Each choice has advantages and disadvantages.

Continuous descriptions can have high accuracy and lower data storage requirements. The trade-off is the increased complexity of representation of obstacles. Conversely discrete descriptions have high storage requirements but reduced complexity of obstacle representation. Discrete maps have limits on accuracy imposed by the basic grid cell size. The precision of the map influences how precisely one can store map features and is a factor in the computational complexity of the algorithms using the map.

- **Uniform** Cells based on a uniform grid. In 2d these are often called pixels. In 3d some call them voxels. Simple storage arrays. Not adaptive and can be storage intensive.
- **Tree** Use of a tree representation for space. Hierarchical decomposition of space. In 2d, one uses a quadtree and in 3d one uses an octree. Can save storage if most of space is not partitioned.
- **BSP** Binary Space Partitioning tree. Each cell is divided in half with some condition about area or partition line.

Much of the machinery used are the tools from computer graphics. Items like points, cells, voxels, polyhedra, splines, patches, etc make up the geometric primitives.

A map will then consist of the representation of space and the objects inside that space. The maps we have discussed so far are usually known as metric maps, ones which have an absolute reference frame and numerical values as to object location.

Another successful approach is using *topological* representations. Here space is represented as a graph. Connectivity between objects (vertices) is done via the edges. The edges can have length as well as be oriented. A vertex represents a unique landmark and can hold orientation data (wrt to the graph).

For unexplored areas of the map one should place “Here there be dragons”…

18.4 Correlation-Based Models

Given a global map, one type of sensor model is to correlate a local sensor based map with a global map. *Map Matching*. It is tied with map building and localization. This is described later in detail. Preliminaries

- Assume that you have a occupancy grid map m.
- Assume that this is a simple map with grid cells marked as occupied or not - binary map.
- Store the map in an array $m[i][j]$.
- Let $x_t = (x, y, \theta)$ be the robot's pose.
- Let z_t^k be the range value of a sensor reading.
- Let $x_{k,\text{sens}}, y_{k,\text{sens}}$ be the location of the sensor in the local coordinates.
- Let $\theta_{k,\text{sens}}$ be the angle of the beam from the local (robot) coordinate system.
- Use sensors to build a local map $m_{\text{local}}[i_L][j_L]$
- Correlate local and global coordinate systems: $\begin{pmatrix} x & y & \theta \end{pmatrix}^T$

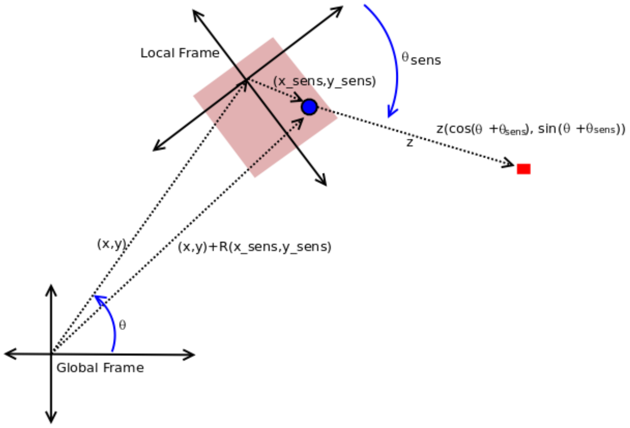


Fig. 18.3: Coordinate transforms to relate observed obstacle to global map.

$$\begin{pmatrix} x_{z_t^k} \\ y_{z_t^k} \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x_{k,\text{sens}} \\ y_{k,\text{sens}} \end{pmatrix} + z_t^k \begin{pmatrix} \cos(\theta + \theta_{k,\text{sens}}) \\ \sin(\theta + \theta_{k,\text{sens}}) \end{pmatrix}$$

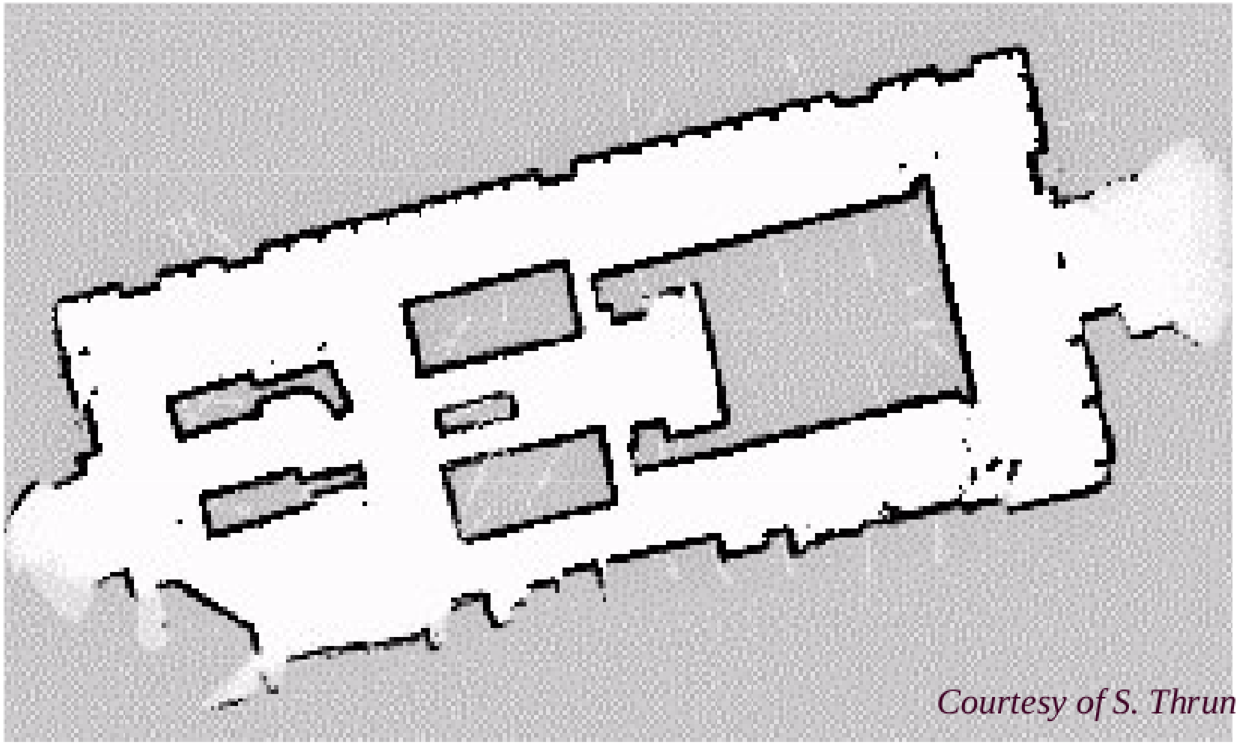
- Find the correlation between the two spatially aligned maps on the common regions of definition.
  - List out the map as a vector $v[k] = m[i][j]$ where $k = n * j + i$.
  - Plot them as vectors and compare, how close?
  - Using the average of the two, show you can get a better comparison by subtracting off the average.
  - Find the angle between the two differenced vectors.
  - Thus $\overline{m} = \frac{1}{2N} \sum \left(m[i][j] + m_{\text{local}}[i][j]\right)$.

- Define

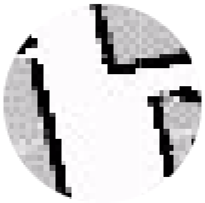
$$\rho = \frac{(m - \overline{m}) \cdot (m_{\text{local}} - \overline{m})}{\|m - \overline{m}\| \|m_{\text{local}} - \overline{m}\|}$$

- Define $p(m_{\text{local}}|x_t, m) = \max\{\rho, 0\}$

Can you do template matching on this? How about ICP?



Where does this fit:



Extract features from measurements.

This is similar to what is done in computer vision.

- Identify features which correspond to distinct objects, call them landmarks.
- Assume you can obtain a range and bearing for the landmark.
- Call the unique identifier for a landmark, a signature.
- For the i^{th} measurement at time t, denote range by r_t^i, bearing ϕ_t^i and signature s_t^i.

Feature based map: $m = \{m_1, m_2, \dots\}$. The j^{th} map feature be defined by $m_j = (m_{j,x}, m_{j,y}, s_j)^T$. The i^{th}

feature then can be correlated to the j^{th} landmark.

Let the robot pose given by $x_t = (x, y, \theta)^T$. Then we have:

$$r_t^i = \sqrt{(m_{j,x} - x)^2 + (m_{j,y} - y)^2} + \epsilon_{\sigma_r^2}$$

$$\phi_t^i = \tan^{-1} \frac{m_{j,y} - y}{m_{j,x} - x} - \theta + \epsilon_{\sigma_\phi^2}$$

$$s_t^i = s_j + \epsilon_{\sigma_s^2}$$

Data association problem A key problem is the association of features to landmarks.

- Introduce a *correspondence variable* between feature f_t^i and landmark m_j: $c_t^i \in \{1, 2, 3, \dots, N+1\}$ where N is the number of landmarks in the map.
- If $c_t^i = j \le N$ then the i^{th} feature observed at time t

  corresponds to the j^{th} landmark in the map. [c_t^i is the true identity.]
- If $c_t^i = N + 1$ then the feature does not correspond to a landmark in the map.

To compute the probability of a feature corresponding to known landmark:

1. $j = c_t^i$
2. $\hat{r}_t^i = \sqrt{(m_{j,x} - x)^2 + (m_{j,y} - y)^2}$
3. $\hat{\phi}_t^i = \text{atan}\left(\frac{m_{j,y} - y}{m_{j,x} - x}\right) - \theta$
4. $q = \text{Gauss}(r_t^i - \hat{r})\text{Gauss}(\phi_t^i - \hat{\phi})\text{Gauss}(s_t^i - \hat{s})$

CHAPTER

NINETEEN

BIBLIOGRAPHY

CHAPTER TWENTY

APPENDIX

20.1 Robotics Accident Data

Table 20.1: Table was compiled from US Government office OSHA, 30 years, 1987-2017, of OSHA Robotics related fatal accident reports.

Date	Description of fatality in OSHA report
6/29/87	Employee Crushed And Killed By Robot Arm
11/28/87	Employee Crushed To Death While Working On A Lathe
5/17/89	Employee Crushed And Killed By Industrial Robot
10/1/92	Employee Killed When Crushed By Robot Arm
3/13/93	Employee Killed When Crushed By Robot
4/8/94	Employee Dies From Coronary Insufficiency
9/27/94	Employee Crushed To Death By Activated Robot
2/15/96	Employee Killed When Burned By Robotic Hot Metal Pourer
1/27/97	Crushed By Robot Arm During Machine Cycle
4/29/97	Employee Struck By Material Handling Equipment
5/4/99	Employee Killed When Crushed By Robot Arms
6/8/99	Employee Killed When Crushed By Robot Against Conveyor
8/27/99	Employee Killed In Robotic Weld Cell
12/29/01	Employee Killed When Robot Pinned His Neck
7/28/03	Employee Is Killed When Crushed By Equipment
12/13/03	Employee Is Killed When Caught In Robotic Arm
3/30/04	Employee Was Killed By Industrial Robots
3/22/06	Employee Dies When Struck By Robotic Equipment
7/24/06	Employee Is Killed When Crushed By Robot
10/31/06	Worker Is Killed, Lockout Procedures Not Followed
5/13/07	Employee Dies After Being Struck By Robotic Arm
7/21/09	Employee Is Killed By Robotic Palletizer
8/2/11	Employee Is Killed When Caught In Equipment
12/15/12	Robot Crushes And Kills Worker Inside Robot Work Cell
3/7/13	Maintenance Worker Is Struck And Killed By Robot
6/16/13	Employee Is Struck By Axis Arm, Later Dies
7/7/15	Employee Is Killed When Head Is Crushed By Robot Arm
6/19/16	Employee is Killed While Making Repairs

20.2 Linux Command Environment

We will survey basic operations which are common to all Linux systems. For the casual user of the command line, Linux, Unix and even OSX systems are the same.

Linux is an operating system, i.e., a language by which you communicate with the computer. Linux is one the most popular true operating systems. This means you are likely to encounter it in your own environment. Linux is written in C/C++, and so not essential, knowledge of C/C++ is very useful when working on all the variants of Linux.

20.2.1 Windowing

The graphical display system on top of Linux is referred to as X Windows (although it should be called the X windowing system). The Ubuntu is currently running the X window system with the Gnome window manager. Type Ctrl + Alt + T to bring up a terminal.

The more commonly used commands are

ls This command stands for list. It asks that the contents of the directory be listed. It is similar to the dos command dir. In the terminal:

```
> ls
```

cp This is the copy file command. So if you have a file named foo.c and you want a copy, say as a backup, you would just type

```
> cp foo.c foo.backup
```

mv This command stands for move. The primary use is to rename files. For example, if you have a file named foo and you want to call it foo.bar the you would type

```
> mv foo foo.bar
```

rm This command stands for remove. It is the delete command. Usage is normally

```
> rm foo.bar
```

mkdir This command stands for make directory (a directory is a folder). From your current directory (if you are in your login directory), you can create a subdirectory named foodir by

```
> mkdir foodir
```

rmdir This command stands for remove directory. You can remove a subdirectory named foodir by

```
> rmdir foodir
```

The directory must be empty to do this.

cd This command stands for change directory. You would use this command to connect to the directory you just created above by

```
> cd foodir
```

Test your knowledge by creating a robotics directory.

20.2.2 Creating a file

Using gedit, you can create a simple program. Type the following into the editor:

```
#include <stdio.h>
int main()
{
   printf("Hello World\n");
   return 0;
}
```

Save this as hello.c: click File and then Save, navigate to your robotics directory, but do not exit.

20.2.3 Compiling

To compile the Hello file, go back to your terminal window and type

```
> cd robotics
> gcc -o hello  hello.c
```

(note that this assumed you created the robotics directory.) To get used to how things work, make some errors in the hello.c file so that you can see what messages you get.

If you made errors, you should see error messages displayed. Fix these errors and recompile. Don't forget to save your program after correcting the errors. To recompile, go back to the shell window and hit the up arrow key. This should bring back the previous command so you don't have to retype it. You can hit the up arrow key repeatedly to bring back past commands. Also, !g will run the last command starting with the letter 'g'. If your program compiles correctly, nothing is printed by the compiler.

There is one odd behaviour that you should know about. If your program uses math.h and you want to compile this on a Linux/Unix system you need to use:

```
> gcc -o hello  hello.c -lm
```

The C language (and Unix) began before math coprocessors were available. Math functions were done in software and they took significant space. The decision was made to not automatically include them to save space and compile time. And so you needed to add the -lm to the compile (gcc) line. At this point, all of this is history, but convention has not changed.

To run any executable (program) under Linux, simply type its name (and any required command-line arguments) at the command prompt in a terminal window. If Linux complains that it cannot find the file, and you see it in the directory listing, try typing:

```
./filename
```

To run your program, just enter the name of the executable as a shell command. In this case, just type hello and hit enter. With the pcDuino as installed, the system will likely say that there is no such command. This is because the current directory, referred to as . (just one dot), is not in your path by default. The easy way to run hello right now is to type the command ./hello which specifies to run the file hello in the current directory.

Note: if you don't like putting "./" in front of your command, you can have the system do this. To put the current directory in your default path, do the following:

- In Leafpad, click file, open, and select ubuntu from the list on the left.
- In the upper left is what looks like a pencil. It should say "Type a file name" when you cursor over it. Click on that.
- Type in the file name .bashrc (including the dot) in the box marked "Location"
- Hit return (or click open in the lower right)
- The file has a lot of stuff in it you do not need to worry about.
- After the three comment lines (starting with #) at the start of the file, insert the line

  ```
  export PATH=$PATH:.
  ```

- Save the file and exit Leafpad. Note that if you did not make a change, "save" is gray, you cannot select save if you have not made any changes. This is true in general for Leafpad.

In general, to compile a C file, type

```
gcc -Wall -g filename.c -o filename
```

The -Wall switch turns on all warnings, and the -g switch adds debugging information. The o filename specifies the name of the resulting executable (the default is a.out). If you get compiler errors, go back to your editor and fix them. Save your file and recompile until you can compile successfully.

If you type only gcc foo.c you will get a default name of a.out. If you only want to compile a function, not link it with anything else, you type

```
gcc -c foo.c
```

This compiles the source to object code, foo.o. For programs with several external functions (meaning multiple files), a longer sequence is required. Say you have a main routine prog.c, and this calls external functions, funct1.c, funct2.c, and requires the library (static linking) lib.a. You would have to compile each function and the main:

```
cc -c prog.c
cc -c funct1.c
cc -c funct2.c
```

Then you can link these together into an executable:

```
gcc -o prog prog.o funct1.o funct2.o lib.a
```

For further practice, try the following:

Type “cd /ho<tab>” the shell should expand the tab so that you see “home/”. The shell will expand a tab to a file name if there is only one way to expand it or expand it until there is a difference. Try “cd /ho<tab>/ubu<tab>”. Do a cd to /usr/include to see the include files available. This can be handy if you are trying to find a function. For example, try

```
grep random *.h | more
```

This will show all the lines that have the word random in them. Use a space to tell the command more to display the next screen.

Use the cd command with no parameters to get back to your home directory. Note that this command (cd) in DOS prints the current working directory like pwd in Linux, but cd in Linux takes you to your current working directory.

```
cd                    Change back to home directory
```

Redirection: At times you may want to capture the output of a command into a file. There is a redirection command available. For example, if you want to list the contents of a directory and save this in a file, type ls > foo and the file foo will be created and output of the ls command entered. Another example is the intended use of the cat command. This command will allow you to concatenate two files (glue them together). If you have two files: file1 and file2 and want to connect them into file3, then cat file1 file2 > file3. In a similar vein is the append command. If you want to append the output of the ls command to the end of file3, then ls >> file3.

20.2.4 Less commonly but still useful commands

cat This command stands for concatenate. It is often used to send the contents of a file display although this is not the main purpose of the command. cat foo will send the contents of the file foo to the display. See the section below on redirection on other uses of cat.

diff File compare. To compare two files, say file1 and file2, type diff file1 file2.

df Report the free and used disk (partition) blocks.

du Report disk usage for a directory or file system (use with the k option du -k).

ps Process status. This lists the processes you have from your current shell. The command line options here depend on which flavor of unix you run. Normally ps -aux will list out all of the processes on the machine.

kill This command will send a signal to a process. Normal usage is to terminate a process kill -9 pid where the pid is the process ID and is given by ps.

nice This command is used to change the priority of processes. For example, you wish to run a program called gnubeast. Say that this program attempts to disprove Fermat’s conjecture by finding a counterexample. You have a nagging feeling that this could run for a long long time and use lots of system resources. Then you should run the program at a lower priority so that other users are not adversely affected by your computations. This is done by nice gnubeast and this will lower the priority by 10 units (a system measure). You can be even nicer by typing nice +15 gnubeast.

grep Search a file for a pattern (think Get Regular ExPression). Usage is grep pattern file. Example: grep foo bar.c will print out the lines in the file bar.c which have the string foo.

tar Tape Archive. A utility to dump a directory or list of files into one file. Usage is tar -cf foo.tar foo where foo is a directory name or a list of files. To recover this, try tar -xf foo.tar.

gzip File compression. Usage is gzip foo and output will be foo.gz.

gunzip File uncompression. Usage is gunzip foo.gz.

ssh Remote secure shell. Will allow a command to be run on a remote computer. Usage is ssh linux101

scp Remote copy. Usage: scp linux101: /home/users/amy/foo.c . This will copy the file foo.c out of your remote home directory to your local directory.

20.3 Installation of Python and Gazebo

Install the SciPy Stack: a collection of open source libraries for scientific computing in Python

```
sudo apt-get install python-numpy python-scipy python-matplotlib ipython
ipython-notebook python-pandas python-sympy python-nose
```

Install the Latex typesetting language through TexLive (libraries) and TexMaker (editor)

```
sudo apt-get install texlive-full
sudo apt-get install texmaker
```

Install Gazebo (a robot simulator):

Setup your computer to accept software from packages.osrfoundation.org.

```
sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu
`lsb_release -cs` main" > /etc/apt/sources.list.d/gazebo-latest.list'
```

Setup keys

```
wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
```

Install Gazebo

```
sudo apt-get update
sudo apt-get install gazebo5-build-deps
sudo apt-get install gazebo5
sudo apt-get install libgazebo5-dev
```

Check your installation (initial execution takes extra time)

```
gazebo
```

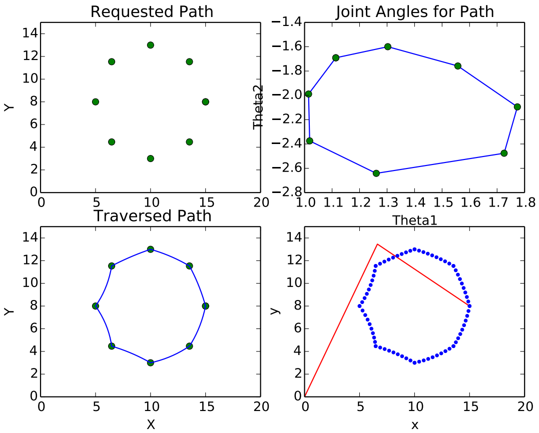


Fig. 20.1: The Two Link Manipulator code results with a larger step in computation of the angle values. The servo movement will connect the vertices, but the path is not straight (you end up with something similar to a polygon, but with curved segments). The simulation will show a stop and go like behavior.

```
import numpy as np
import matplotlib.pyplot as plt
import time
from math import *

a1 = 15
a2 = 10
step = np.pi/4
N = 10

t = np.arange(0, 2*np.pi+step, step)
x = 5*np.cos(t) + 10
y = 5*np.sin(t) + 8
xsim1 = np.zeros((t.size-1)*N)
ysim1 = np.zeros((t.size-1)*N)
xsim = np.zeros((t.size-1)*N)
ysim = np.zeros((t.size-1)*N)

a1 = 15.0
a2 = 10.0
d = (x*x + y*y - a1*a1 - a2*a2)/(2*a1*a2)
t2 = np.arctan2(-np.sqrt(1.0-d*d),d)
t1 = np.arctan2(y,x) - np.arctan2(a2*np.sin(t2),a1+a2*np.cos(t2))

for i in range(t.size-1):
  t1step = np.linspace(t1[i],t1[i+1], N)
  t2step = np.linspace(t2[i],t2[i+1], N)
  for k in range(N):
    xsim1[N*i+k] = a1*np.cos(t1step[k])
    ysim1[N*i+k] = a1*np.sin(t1step[k])
```

(continues on next page)

(continued from previous page)

```
    xsim[N*i+k] = a2*np.cos(t1step[k]+t2step[k]) + xsim1[N*i+k]
    ysim[N*i+k] = a2*np.sin(t1step[k]+t2step[k]) + ysim1[N*i+k]

#  Plot code removed for space
#  Same as previous example (from Plots to Animation)
for k in range((t.size-1)):
  for j in range(N):
      i = k*N+j
      x1 = xsim1[i]
      y1 = ysim1[i]
      x2 = xsim[i]
      y2 = ysim[i]
      plt.setp(arm,xdata = [0,x1,x2], ydata = [0,y1,y2])
      plt.draw()
      plt.plot([x2],[y2],'b.')
  time.sleep(0.15)

plt.ioff()
#plt.savefig("twolinkexample.pdf",format="pdf")  # if you want to save
plt.show()
```

20.3.1 Mobile Example

We provide a similar example to what was done in the Python Chapter.

$$\left(\frac{dx}{dt},\frac{dy}{dt}\right) = \begin{cases} (0.5t, 0.0), & 0 \le t < 2, \\ (0.25t, 0.65t), & 2 \le t < 5, \end{cases}$$

$$andstartingat : math : `t = 0`, : math : `(x, y) = (1, 1)`.$$

```
line, = plt.plot([],[],'bo')
plt.xlim(0, 6)
plt.ylim(0, 8)
plt.xlabel('x')
plt.ylabel('y')
plt.draw()
x = 1.0
y = 1.0
dt = 0.1

for t in np.arange(0,5,dt):
    if t < 2:
        x = x + 0.5*t*dt
    if (t>=2):
        x = x + 0.25*t*dt
        y = y + 0.65*t*dt
    line.set_xdata([x])
    line.set_ydata([y])
    plt.draw()
```

(continues on next page)

(continued from previous page)

```
    plt.plot([x],[y],'bo')

plt.ioff()
plt.show()
```

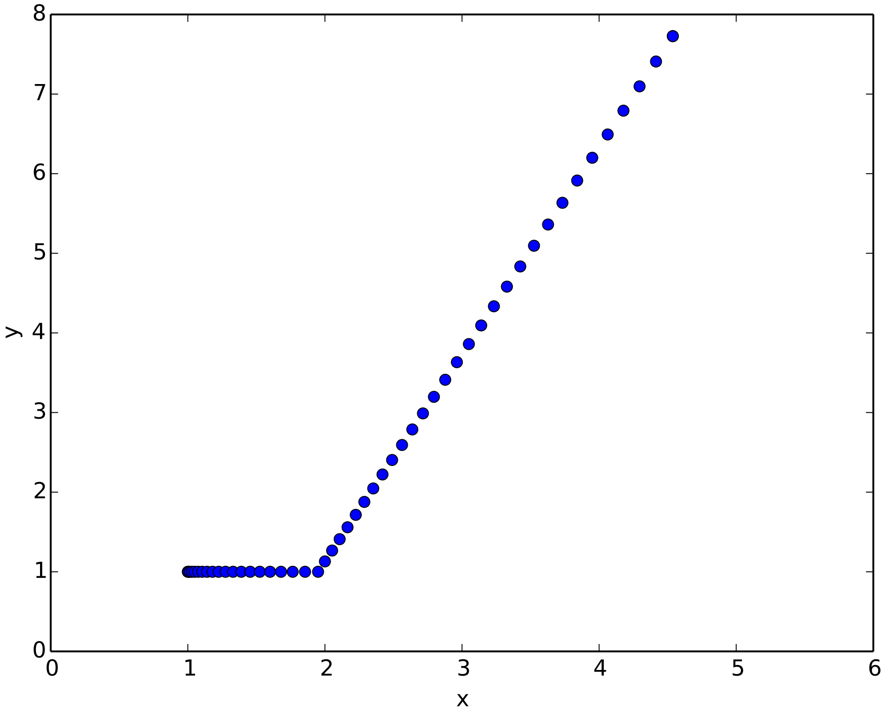


Fig. 20.2: A plot of the simple mobile example.

20.4 Client Code in C

The following code implements the client control code in C. It is a minor modification of the free download at http://www.linuxhowtos.org/data/6/client.c.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(const char *msg)
{
```

(continues on next page)

(continued from previous page)

```
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    if (argc < 3) {
       fprintf(stderr,"usage %s hostname port\n", argv[0]);
       exit(0);
    }
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
         (char *)&serv_addr.sin_addr.s_addr,
         server->h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR connecting");

    while(1) {
     printf("> ");
     bzero(buffer,256);
     fgets(buffer,255,stdin);
     n = write(sockfd,buffer,strlen(buffer));
      if (n < 0)
        error("ERROR writing to socket");
      bzero(buffer,256);
          n = read(sockfd,buffer,255);
          if (n < 0)
              error("ERROR reading from socket");
          printf("%s\n",buffer);
    }
    close(sockfd);
    return 0;
}
```

CHAPTER

TWENTYONE

MATHEMATICAL AND COMPUTATIONAL BACKGROUND

Robotics as an academic subject can be very mathematical in nature. This text does not attempt to place the subject on a rigorous mathematical foundation, but it is necessary to use the mathematical formalism to avoid confusion. So, we jump in with some common notation and then proceed to the standard vocabulary that all robotics professionals use.

Note: This is under significant development.

21.1 Mathematical Notation

This text will use italicized letters for the variables in formulas: x. We will not distinguish scalar (single) variables from vector (array) variables. Scalar and vector quantities will normally be lower case (unless we need to be consistent with a well known formula). When possible, elements of a vector will be indexed using a subscript x_k. Matrices will be indicated using italicized uppercase, M. Discrete time steps will be indicated by superscripts, x^k.

Notation related to robotics:

- Real line: $\mathbb{R}$
- Plane: $\mathbb{R}^2$
- Space: $\mathbb{R}^3$
- Workspace: $\mathcal{W}$
- Workspace Obstacles: $\mathcal{WO}_i$
- Free space: $\mathcal{W} \setminus \bigcup_i \mathcal{WO}_i$
- Point in space: $x = (x_1, x_2, x_3)$
- Vector: $\vec{v} \in \mathbb{R}^n$, $v = [v_1, v_2, \ldots, v_n]^T$.
- Bounded workspace: $\mathcal{W} \subset B_r(x) \equiv \{y \in \mathbb{R}^n | d(x,y) < r\}$ for some $0 < r < \infty$.

We will use several forms of notation for derivatives. Let $f(x)$ be a differentiable function, both Newton's notation, $f'(x)$, and Leibniz's notation, df/dx, will be employed. Specifically for derivatives of functions of time, we will also use the superscript dot notation, $\dot{g}$. The dot product and cross product of two vectors will be denoted by $x \cdot y$ and $x \times y$ respectively. Matrix vector and matrix matrix products will be denoted by Ax

and AB. The normal matrix multiplication is implied. Rarely will we need to do element-wise matrix and vector operations. Those will be written out since they don't occur often enough to warrant custom notation.

There is a temptation for some authors to present material in the most abstract setting possible. It can be argued that this provides the most general case (widest application). In addition, one should remove the specifics of any application so the abstract formulation represents the core concept at its purest form. In many cases this is true. At times it is also true that the author is trying to impress his or her audience with their mathematical talent using an assault of symbols. At times I am sure you will feel this way, but we have made significant effort to keep the mathematical level in the first two years of an engineering curriculum (Calculus, Differential Equations, Linear Algebra, Probability).

21.2 Parametric Form

Say you want to traverse a path C, shown in Figure Fig. 21.1

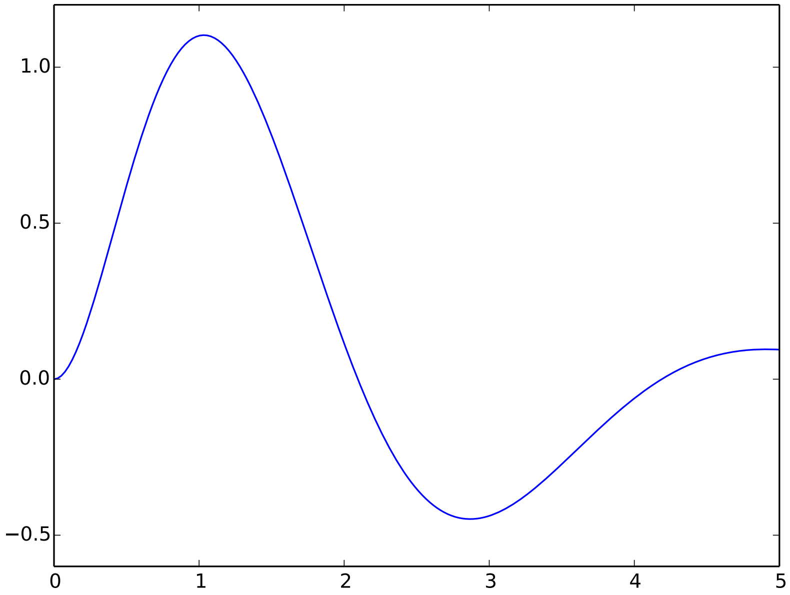


Fig. 21.1: A path for an explicitly defined function.

The path C often will come from some function description of the curve $y = f(x)$. This type of description will work for many paths, but fails for a great number of interesting paths like circles: $x^2 + y^2 = 1$. We want to be able to wander around in the plan crossing our own path which certainly is not the graph of a function. So, we must move to a parametric description of the path (actually a piecewise parametric description). You want to prescribe $x(t), y(t)$ and obtain $\dot{\phi}_1, \dot{\phi}_2$. Clearly if you have $x(t), y(t)$, differentiation will yield $\dot{x}(t), \dot{y}(t)$, so we may assume that we know $\dot{x}(t), \dot{y}(t)$. Using $\dot{x}$ and $\dot{y}$ we may drive the robot along the curve of interest. How does one follow an arbitrary curve?

The first step is to write in parametric form: $x(t)$, $y(t)$. Example: convert $y = x^2$ to parametric

$$\text{Let } x = t \quad \rightarrow \quad y = x^2 = t^2$$

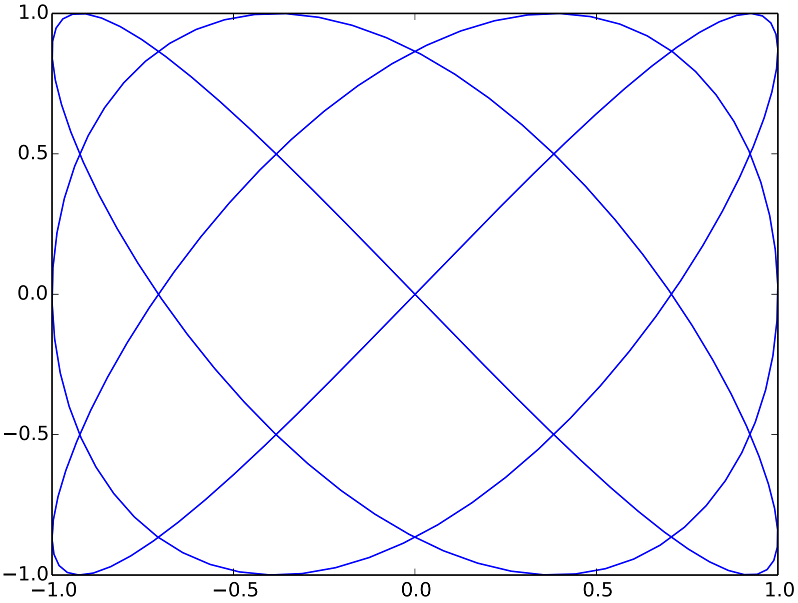


Fig. 21.2: A path for a parametric function.

Note that there are an infinite number of choices :

Let

$$\begin{array}{lll} x = 2t & \to & y = x^2 = 4t^2 \\ x = e^t & \to & y = x^2 = e^{2t} \\ x = \tan(t) & \to & y = x^2 = \tan^2(t) \end{array}$$

and so forth.

All the parametric forms provide the same curve, same shape, same geometry. They vary in the speed. Think of the function form telling you the shape, like the shape of a road, but not the velocity. The parametric form gives you both path shape and velocity. We will assume that you can find parametric functions $x = \phi(t)$ and $y = \psi(t)$ such that the graph is $y = f(x)$ which generates the path C of interest.

21.2.1 Example Functions

Some examples of parametric forms may help in getting good at writing these down.

Line $x(t) = t$, $y(t) = mt + b$, where m is the slope and b is the intercept.

Circle $x(t) = R\cos(t) + h$, $y(t) = R\sin(t) + k$, where the radius is R and the center is (h, k).

Ellipse $x(t) = A\cos(t) + h$, $y(t) = B\sin(t) + k$, where A and B describe the major and minor axes and the center is (h, k).

Lissajous $x(t) = A\sin(at)$, $y(t) = B\sin(bt)$ (Fig. 21.2 $A = 1$, $B = 1$, $a = 3$, $b = 4$). Infinity: $A = 1$, $B = 0.25$, $a = 1$, $b = 2$

Root $x(t) = t^2$, $y(t) = t$.

Heart $x(t) = 16\sin^3(t)$, $y(t) = 13\cos(t) - 5\cos(2t) - 2\cos(3t) - \cos(4t)$

21.3 Vectors, Matrices and Linear Systems

A vector is a list of numbers. It can be used to represent physical quantities like force and direction. It can be expressed as

$$\vec{x} = \langle x_1, x_2, x_3, \ldots, x_n \rangle .$$

The notation for a point in n-dimensional space and a n-dimensional vector are similar: $\vec{x} \in \mathbb{R}^n$: $\vec{x} = (x_1, x_2, ...x_n)$, and also written as

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} .$$

If the context is understood, the small arrow above the variable is left off, so $\vec{x}$ becomes x. The basic datatype used in scientific computing is the array. Arrays are used to store points, vectors, matrices and other mathematical constructs. The basic operations defined on vectors are listed below. Let $c \in \mathbb{R}$ and $x, y \in \mathbb{R}^n$, then

- Sum: $x + y = \{x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n\}$
- Scalar multiplication: $cx = \{cx_1, cx_2, \ldots, cx_n\}$
- Inner product (related to angle): $x \cdot y = \sum_{i=1}^n x_i y_i$
- Norm (length): $\|x\| = \sqrt{\sum_{i=1}^n x_i^2}$
- Norm as multiplication: $\|x\|^2 = x^T x$

We will make use of matrix algebra and will follow the normal conventions. Let $A, B \in \mathbb{R}^{n\times n}$,

$$A = \begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \ldots & \ldots & \ldots \\ a_{n1} & \ldots & a_{nn} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & \ldots & b_{1n} \\ \ldots & \ldots & \ldots \\ b_{n1} & \ldots & b_{nn} \end{pmatrix} .$$

Matrix addition and multiplication are defined in the standard manner as

- $A + B = \begin{pmatrix} a_{11} + b_{11} & \ldots & a_{1n} + b_{1n} \\ \ldots & \ldots & \ldots \\ a_{n1} + b_{n1} & \ldots & a_{nn} + b_{nn} \end{pmatrix}$
- $AB = \begin{pmatrix} c_{11} & \ldots & c_{1n} \\ \ldots & \ldots & \ldots \\ c_{n1} & \ldots & c_{nn} \end{pmatrix}$,

where the entries are $c_{ij} = \sum_k a_{ik}b_{kj}$

Matrix vector multiplication occurs often and is given by

- $Ax = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \sum_k a_{1k}x_k \\ \sum_k a_{2k}x_k \\ \vdots \\ \sum_k a_{nk}x_k \end{pmatrix}$

The identity element and the matrix transpose are given by

- $I = \begin{pmatrix} 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix}$
- Transpose: A^T: $\{a_{ij}\}^T = \{a_{ji}\}$ Example: If $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ then $A^T = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$

Some additional matrix terms and properties:

- The matrix determinant is indicated by det(A)
- The transpose formula is given by $(AB)^T = B^T A^T$
- The determinant formula is given by det(AB) = det(A)det(B)
- A symmetric matrix is defined by $A^T = A$
- A symmetric positive definite matrix satisfies $x^T Ax > 0$ for $x \neq 0$.

21.4 Linear Systems

One of the most common mathematical operations is solving simultaneous linear equations:

$$\begin{gathered} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n \end{gathered}$$

Using the matrix notation defined above we may write this in a very compact form:

$$\Rightarrow \quad Ax = b$$

where

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{n1} & \dots & a_{nn} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$$

One approach to solve the equations is Gaussian Elimination. The industry version of Gaussian Elimination is the LU factorization. An LU factorization decomposes the matrix A into the product of a lower triangular matrix, L, and an upper triangular matrix, U. The strength of this approach is that the LU factorization is done for A once. Once done, solving $Ax = b$ for different b's can be done relatively easily. You don't actually have to know how to do this, only how to call the system solvers.

21.4.1 Inverses

The inverse of A is notated A^{-1}:

$$A(A^{-1}) = I = (A^{-1})A$$

Given the inverse:

$$Ax = b \rightarrow x = A^{-1}b$$

Is this a good approach to solving $Ax = b$?

No. The fast multiplication algorithms are not numerically stable. Best to use a Gauss-Jordan based approach like the LU factorization. LU can also make good use of matrix structure. Possible that an algorithm may list an inverse, but this can often be converted to a linear solve. For example if the formula lists $y^* = y + BC^{-1}x$, then solve $Cz = x$ first and then find $y^* = y + Bz$.

Finding curves from data

Say that you have a data set:

$$(x_i, y_i), \quad i = 1, \ldots, k$$

and you want to fit a model to it:

$$y = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

or in general

$$y = a_n\phi_n(x) + a_{n-1}\phi_{n-1}(x) + \cdots + a_0\phi_0(x).$$

How does one use the data to find the coefficients of the model?

Plug the data into the model:

$$\begin{aligned}
y_1 &= a_n x_1^n + a_{n-1}x_1^{n-1} + \cdots + a_1 x_1 + a_0 \\
y_2 &= a_n x_2^n + a_{n-1}x_2^{n-1} + \cdots + a_1 x_2 + a_0 \\
&\vdots \qquad\qquad\qquad\qquad\qquad\qquad . \\
y_{k-1} &= a_n x_{k-1}^n + a_{n-1}x_{k-1}^{n-1} + \cdots + a_{k-1} x_{k-1} + a_0 \\
y_k &= a_n x_k^n + a_{n-1}x_k^{n-1} + \cdots + a_1 x_k + a_0
\end{aligned}$$

This can be rewritten in the language of matrix algebra.

Plug the data into the model:

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix}}_{y} = \underbrace{\begin{bmatrix} x_1^n & x_1^{n-1} & \dots & x_1 & 1 \\ x_2^n & x_2^{n-1} & \dots & x_2 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_k^n & x_k^{n-1} & \dots & x_k & 1 \end{bmatrix}}_{X} \underbrace{\begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_1 \\ a_0 \end{bmatrix}}_{a}.$$

$$y = Xa \tag{21.1}$$

If $k = n + 1$, then this system is square, meaning it has the same number of equations as unknowns. For the polynomial, the matrix will be invertible and we can solve the system. For larger values of n, the system becomes ill-conditioned and has some numerical accuracy problems, but can be solved in the theoretical sense. Specific formulas have been created to avoid the numerical errors. These are the Lagrange formulas presented in the next section.

21.4.2 Simple Example

Fit a quadratic to (0,1), (1,2), (2,5). The quadratic is $y = a_2x^2 + a_1x + a_0$. The matrix model is

$$\underbrace{\begin{bmatrix} 1 \\ 2 \\ 5 \end{bmatrix}}_{y} = \underbrace{\begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 4 & 2 & 1 \end{bmatrix}}_{X} \underbrace{\begin{bmatrix} a_2 \\ a_1 \\ a_0 \end{bmatrix}}_{a}.$$

Then we must row reduce

$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 2 \\ 4 & 2 & 1 & 5 \end{bmatrix} \to \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 4 & 2 & 0 & 4 \end{bmatrix} \to \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} \to \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

You can read off the coefficients here: $a_2 = 1$, $a_1 = 0$ and $a_0 = 1$. Thus we obtain $y = x^2 + 1$ which checks with the data. The next section gives you a way to do this without a matrix solve.

21.5 Interpolation

21.5.1 Lagrange Interpolation

There are times when you would like to just prescribe the points and generate the polynomial without the worry of having numerical issues in the linear solver. There are a couple of approaches to finding the polynomial, one popular method is known as Lagrange Interpolation. For a set of N points, we can find a polynomial of degree $N-1$ that can interpolate the points. A well known approach is to use Lagrange polynomials:

$$x(t) = \sum_{i=0}^{N} x_i q_i(t), \quad y(t) = \sum_{i=0}^{N} y_i q_i(t) \quad \text{where} \quad q_i(t) = \prod_{\substack{j=0 \\ j\neq i}}^{N} \frac{t-t_j}{t_i-t_j}$$

Since this is a parametric form, we have freedom to select the $\{t_i\}$ values.

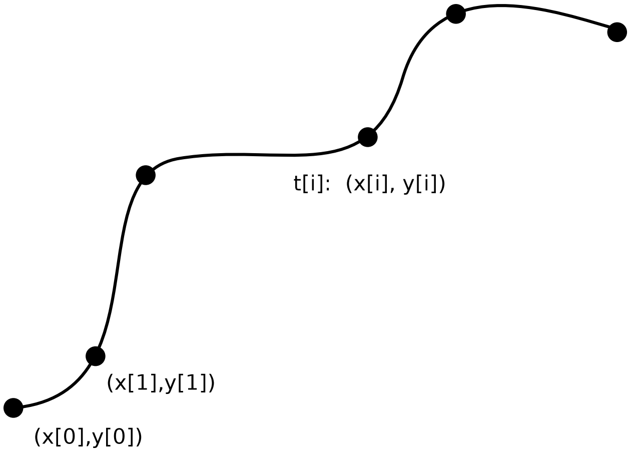


Fig. 21.3: Polynomial Interpolant of data.

Assume that you are given the points (0,1), (1,2), (2,5). Find the Lagrange interpolant. First we define $t_i = i$ and compute the Lagrange polynomials:

$$q_0(t) = \left(\frac{t-1}{0-1}\right)\left(\frac{t-2}{0-2}\right) = \frac{1}{2}(t-1)(t-2)$$

$$q_1(t) = \left(\frac{t-0}{1-0}\right)\left(\frac{t-2}{1-2}\right) = -(t)(t-2)$$

$$q_2(t) = \left(\frac{t-0}{2-0}\right)\left(\frac{t-1}{2-1}\right) = \frac{1}{2}(t)(t-1)$$

Then using the interpolation formula:

$$x(t) = -t(t-2) + t(t-1) = t,$$

$$y(t) = \frac{1}{2}(t-1)(t-2) - 2t(t-2) + \frac{5}{2}(t)(t-1) = t^2+1.$$

This process can be used for arbitrary many points. However, the greater the number of points, the higher degree polynomial and several problems arise. Clearly the formulas get more complicated as well as the computation effort. The central problem is that the interpolant can oscillate between data points. Although the polynomial includes the data points, a poor path emerges. Another approach is to fix the degree of polynomial and attempt a least squares approximation. In this case, the path will have less oscillation, but could miss many or possibly all of the data points.

So, why not just limit the number of points used? Say we pick two or three points at a time? Two points will give rise to a linear interpolant and three will give rise to a quadratic interpolant. We just take two or three at a time computing the interpolants as we travel. This would have the added benefit that we don't even need to know all of them when we start. And this idea takes us to a tool known as cubic splines - which can be done in an iterative fashion as well as having smooth connections.

21.5.2 Cubic Splines

The straight line connection between two points discussed above uses a linear polynomial. To gain the smoothness in the transition from point to point, we need a higher degree polynomial. At minimum for matching at a point requires both the location and direction. Direction is prescribed by the derivative. This is four data items: left position, left derivative, right position and right derivative. A quadratic only has three degrees of freedom which would result in some points not having a smooth transition, so we move to a cubic polynomial.

The method of Cubic Splines is one of the most popular interpolation methods. There are several methods that can be used to find the cubic spline given the endpoint data. In addition to fitting the data, it also will minimize the curvature along the interpolant. This is exactly the tool we need. It can be used iteratively as data points arrive in the path queue and can be used iteratively to produce wheel velocities. Assume that you have two points $t_0 : (x_0, y_0)$ and $t_1 : (x_1, y_1)$. Also assume that you have a derivative at each point $t_0 : (\dot{x}_0, \dot{y}_0)$ and $t_1 : (\dot{x}_1, \dot{y}_1)$. The cubic spline is

$$x(t) = (1-z)x_0 + zx_1 + z(1-z)\left[a(1-z) + bz\right]$$

$$y(t) = (1-z)y_0 + zy_1 + z(1-z)\left[c(1-z) + dz\right]$$

where

$$a = \dot{x}_0(t_1 - t_0) - (x_1 - x_0), \quad b = -\dot{x}_1(t_1 - t_0) + (x_1 - x_0)$$

$$c = \dot{y}_0(t_1 - t_0) - (y_1 - y_0), \quad d = -\dot{y}_1(t_1 - t_0) + (y_1 - y_0)$$

$$z = \frac{t - t_0}{t_1 - t_0}$$

When we are working with signal filters we end up with a large number of sample points. One of the filter techniques is to "fit" a polynomial to the points. However, we will want to limit the degree of the polynomial and this gives rise to non-square systems (more equations and unknowns). This problem is addressed below in the least squares section.

21.6 Linear Algebra Concepts

When we attempt to integrate multiple sensors or when we compute paths from discrete points, the methods use the tools from linear algebra. No attempt here is made to be complete or expository. This is intended to review the language and concepts only. The reader who is unfamiliar with Linear Algebra as a subject is strongly encouraged to explore it,[1]. Calculus, Linear Algebra and Probability are three legs to the mathematical stool every engineer should have.

If $x, y \in \mathbb{R}^n$ are vectors, and $a, b \in \mathbb{R}$ are real numbers, we say that $ax + by$ is a *linear combination* of x and y. Over all possible values for a and b, we say $ax + by$ is a span of x and y. Spanning sets arise in all sort of

[1] Gilbert Strang - Linear Algebra, see the online text.

applications. It is a way to decompose sets into basic components. For example, the span of $x = \langle 1, 0\rangle$ and $y = \langle 0, 1\rangle$ is the plane and the vectors x and y are a known as a basis. The term basis is a minimal spanning set and the number of linear independent basis elements is the dimension. More information on these ideas can be found in most linear algebra textbooks.

We can represent a line through the origin by $t\,\langle a, b\rangle$

where $t \in \mathbb{R}$ (t is the scale factor). Geometrically we are scaling the vector into spanning a line. The vector we are using is $\langle a, b\rangle$. Another example is the collection of all 2×2 matrices:

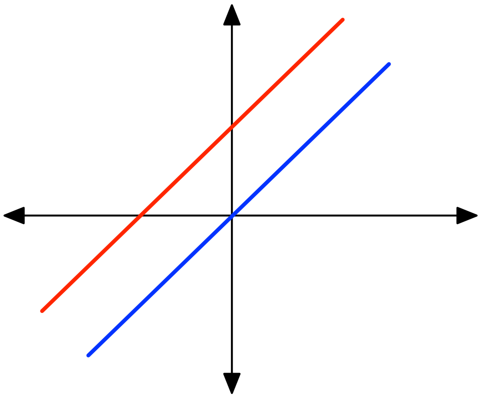
$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

which is the linear combination of

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

One consequence of these ideas is that of a vector space. It is the span of a collection of vectors (or all linear combinations of the vectors). More formally, V is a vector space if $x, y \in V$ are vectors, and $a, b \in \mathbb{R}$, then $ax + by \in V$.

The two examples above are vector spaces: the line through the origin and the collection of 2×2 matrices. Note that in the figure below, the solid line is a vector space is, and the dotted is not. A vector space must include the zero element and the dotted line does not.



A subspace is a subset of a vector space V that is also a vector space. For example, a line through the origin is a subspace of the plane. Also, a plane through the origin is a subspace of three space, such as the span of

$$\left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \right\}.$$

The reason these concepts are discussed is that when solving linear systems or doing least squares (optimization), you are often working with vector spaces and subspaces. The literature uses this terminology and the concepts have a very rich geometric structure which can be helpful in understanding the problems.

A very well studied subspace is the *Nullspace* of a matrix, N. It is defined as all w such that $Aw = 0$. Note that if $Au = 0$ and $Av = 0$ then

$$A(cu + dv) = cAu + dAv = c(0) + d(0) = 0$$

thus it is correctly called a subspace. Also, $u = 0$ is trivially in the nullspace. If a matrix has a nullspace, then the associated linear systems problem $Ax = b$ will not have a unique solution which is important to know if you need a solution to your problem.

An example of this issue is if you wanted to solve $Ax = b$ where

$$A = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.$$

Can this be solved for x? In this trivial example you can see that it can be and $x = \langle 1, 0, 0 \rangle$ works. However the solution is not unique. Without going into the details, we see that there are two vectors which span the Nullspace:

$$v_1 = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \quad v_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

i.e. $Av_1 = 0$ and $Av_2 = 0$. So we actually gain a two dimensional family of solutions (meaning a plane)

$$x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + c_1 \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} + c_2 \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

Another popular subspace is known as the *Column Space*. It is the span of the columns (treated as vectors) of A. This tells you the range space of the matrix. Using the last A as the working example:

$$A = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

the range is given by the span of the columns. So we have

$$\left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \right\}$$

Note that a similar notion is the span of the rows, called the *Row Space*.

21.6.1 Eigenvalues and Eigenvectors

Let x solve $Ax = \lambda x$ (the invariant directions problem).

$$Ax - \lambda x = 0 \quad \Rightarrow \quad (A - \lambda I)x = 0 \quad \Rightarrow \quad x \in \mathcal{N}(A - \lambda I)$$

The latter saying that x must be in the Nullspace of $A - \lambda I$. This implies the following polynomial equation which is solved for roots λ.

$$\det(A - \lambda I) = 0 \quad \Rightarrow \quad \lambda$$

We can numerically solve for (λ, x) and these are known as an eigenvalue, eigenvector pair. An example of the SciPy eigenvalue solver is given below.

Eigenvalues for Symmetric Matrices

Assume that A is a real symmetric matrix and that (λ, v) is an eigenvalue, eigenvector pair. If v is complex valued then $\|v\|^2 = v \cdot \bar{v}$ where $\bar{v}$ is the complex conjugate of v. Then we have

$$\lambda\|v\|^2 = \lambda v \cdot \bar{v} = Av \cdot \bar{v} = v \cdot A\bar{v} = v \cdot \overline{Av} = v \cdot \overline{\lambda v} = \bar{\lambda} v \cdot \bar{v} = \bar{\lambda}\|v\|^2$$

So this implies that $\lambda = \bar{\lambda}$ or that λ is real valued.

21.6.2 Orthogonal

The last concept we will review is orthogonality. The basic term means perpendicular. Two vectors, x and y are said to be orthogonal if their dot product is zero:

$$x \cdot y = 0.$$

A matrix, Q, is said to be orthogonal if its columns treated as vectors are mutually orthogonal and of unit length. This turns out to be mathematically equivalent to a matrix satisfying

$$QQ^T = I$$

where I is the identity matrix. We will see orthogonal matrices later when we compute rotations in space. These matrices will be the foundations of the coordinate transformations used in robotic arms.

21.6.3 The Pseudo-Inverse

We will at several occasions run into the problem of solving what is known as the *overdetermined* problem. This is the linear systems problem for which there are more equations than there are unknowns (variables).

The problem is then

$$\begin{array}{c} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n = b_m \end{array} \quad , m > n \tag{21.2}$$

Just as before we can use the matrix notation to write this in a very compact form:

$$\Rightarrow \quad Ax = b$$

where

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}.$$

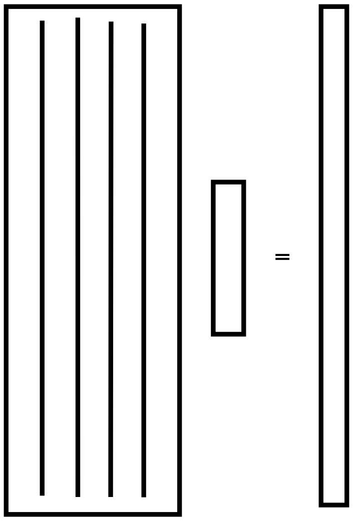


Fig. 21.4: Overdetermined System of Equations

This leads to a non-square matrix which is not invertible. There is no exact solution: $Ax \neq b$ for all possible x in this case. So instead of trying to solve the problem exactly, we ask about getting as close as possible. In other words, this problem is not solvable by regular methods such as the LU factorization or Gauss-Jordan elimination, but can be addressed by minimizing the error using the method of least squares.

The columns must be linearly independent for this method to succeed so we assume that for now. With the columns linearly independent, the core issue geometrically is that the vector b is not in the span of the columns of A. The best we can ask is to get as close as possible. Thus we optimize:

$$\min \|Ax - b\|$$

where we will call the minimizer $\hat{x}$. To minimize we express the norm as a matrix multiply:

$$\|Ax - b\|^2 = (Ax - b)^T(Ax - b) = (Ax)^T(Ax) - b^T(Ax) - (Ax)^T b + b^T b.$$

Note that $b^T Ax = (Ax)^T b$, and $(Ax)^T = x^T A^T$, so we have

$$\|Ax - b\|^2 = x^T A^T Ax - 2x^T A^T b + b^T b.$$

Next we form the gradient of the norm with respect to x. We leave to a homework to show $\nabla[x^T A^T Ax] = 2A^T Ax$ and $\nabla[x^T A^T b] = A^T b$. Then we have

$$\nabla\|Ax - b\|^2 = 2A^T Ax - 2A^T b.$$

To find the minimizer, set $\nabla\|Ax - b\|^2 = 0$ so we obtain

$$A^T A\hat{x} = A^T b.$$

These are known as the *Normal Equations*.

The matrix $A^T A$ is symmetric and if the columns of A are linearly independent, then $A^T A$ is invertible. This yields the solution

$$\hat{x} = \left(A^T A\right)^{-1} A^T b.$$

This formula is known by several names. It is called the Pseudo-Inverse or Moore-Penrose Pseudo-Inverse. It is also called the left-sided pseudo-inverse (because it acts on the left side).

Example Find the least squares solution to

$$\begin{pmatrix}1 & 0\\ 1 & 1\\ 0 & 2\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix} = \begin{pmatrix}1\\ 2\\ 1\end{pmatrix}$$

Forming the normal equations

$$\begin{pmatrix}1 & 1 & 0\\ 0 & 1 & 2\end{pmatrix}\begin{pmatrix}1 & 0\\ 1 & 1\\ 0 & 2\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix} = \begin{pmatrix}1 & 1 & 0\\ 0 & 1 & 2\end{pmatrix}\begin{pmatrix}1\\ 2\\ 1\end{pmatrix}$$

and multiplying out

$$\begin{pmatrix}2 & 1\\ 1 & 5\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix} = \begin{pmatrix}3\\ 4\end{pmatrix}.$$

Solving the two by two system, we obtain

$$\begin{pmatrix}x_1\\ x_2\end{pmatrix} = \begin{pmatrix}\frac{11}{9}\\ \frac{5}{9}\end{pmatrix}.$$

Does this actually solve the problem?

$$\begin{pmatrix}1 & 0\\ 1 & 1\\ 0 & 2\end{pmatrix}\begin{pmatrix}\frac{11}{9}\\ \frac{5}{9}\end{pmatrix} = \begin{pmatrix}\frac{11}{9}\\ \frac{16}{9}\\ \frac{10}{9}\end{pmatrix} \neq \begin{pmatrix}1\\ 2\\ 1\end{pmatrix}$$

It does not solve the problem. What about residual (error)?

$$\| \begin{pmatrix}\frac{11}{9}\\ \frac{16}{9}\\ \frac{10}{9}\end{pmatrix} - \begin{pmatrix}1\\ 2\\ 1\end{pmatrix} \| = \sqrt{(2/9)^2 + (2/9)^2 + (1/9)^2} = 1/9$$

Can we do any better? For any value $x = \langle x_1, x_2\rangle$, is it possible for

$$\| \begin{pmatrix}1 & 0\\ 1 & 1\\ 0 & 2\end{pmatrix} u - \begin{pmatrix}1\\ 2\\ 1\end{pmatrix} \| < 1/9?$$

We will minimize the square of the norm to avoid issues with the square root. The first derivatives must be zero and we apply the second derivative test if the error is a minimum.

$$f(x_1, x_2) = (x_1 - 1)^2 + (x_1 + x_2 - 2)^2 + (2x_2 - 1)^2$$

$$f_{x_1} = 2(x_1 - 1) + 2(x_1 + x_2 - 2), \quad f_{x_2} = 2(x_1 + x_2 - 2) + 4(2x_2 - 1)$$

We see that

$$f_{x_1}(11/9, 5/9) = 0, \quad f_{x_2}(11/9, 5/9) = 0$$

and

$$f_{x_1x_1} = 4, \quad f_{x_2x_2} = 10, \quad f_{x_1x_2} = 2$$

The second derivative test gives $D = 40 - 4 = 36$ which means our surface is curved up at the critical point and thus $(11/9, 5/9)$ is a local min. The function f is a parabolic surface and so $(11/9, 5/9)$ is the global min. Meaning it is the best that we can do.

The other variation of the non-square linear system is the *underdetermined* problem. In this case we have more columns than rows and so has the structure shown in Figure Fig. 21.5

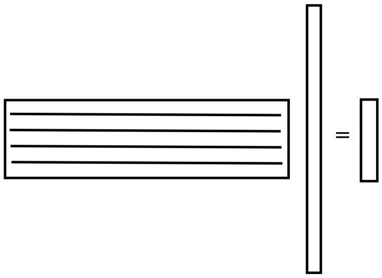


Fig. 21.5: An underdetermined system

The columns cannot be linearly independent and so A^TA is not invertible which means the left sided pseudo-inverse $\left(A^TA\right)^{-1}$ does not exist. So, we need to go another route.

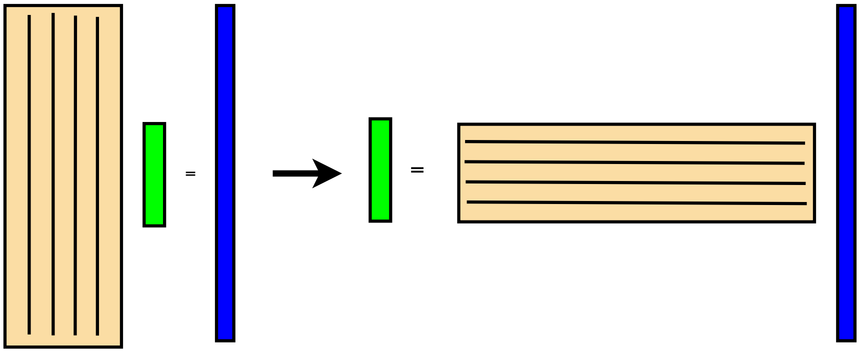
This time instead of assuming the columns are linearly independent we will assume the rows are linearly independent. So although A^TA is not invertible, we have that $\left(AA^T\right)$ is of full rank, or invertible. Using $\left(AA^T\right)$ on the right side gives us the result. Admittedly this version is less intuitive.

$$Ax = b \quad \Rightarrow \quad Ax = Ib$$

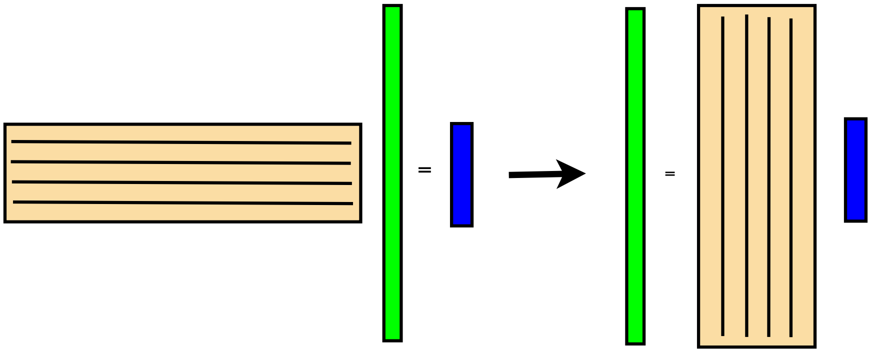
$$Ax = \left(AA^T\right)\left(AA^T\right)^{-1} b$$
$$Ax = AA^T\left(AA^T\right)^{-1} b$$
$$\hat{x} = A^T\left(AA^T\right)^{-1} b$$

Pseudo-Inverse Formulas

1. Left Moore-Penrose Pseudo-Inverse (A has linearly independent columns): $A^+ = \left(A^TA\right)^{-1}A^T$: $A^+A = I$



2. Right Moore-Penrose Pseudo-Inverse (A has linearly independent rows): $A^+ = A^T\left(AA^T\right)^{-1}$: $AA^+ = I$



21.6.4 Singular Value Decomposition

For the normal equations to be invertible the columns of the matrix A must be linearly independent, meaning as vectors they point in different directions. This is fine in the theoretical context, but in practice a data set can produce columns which point in similar directions. This can cause problems with the accuracy of the solution to the normal equations. In addition, the product of A times the transpose of A can increase the ill-conditioning of the matrix.

The standard method to address numerical problems such as this is to compute the pseudo-inverse through the Singular Value Decomposition (SVD). We will present the SVD first and then show how it applies to the pseudo-inverse.

(details needed here) The SVD of $A = U\Sigma V^T$. U, V are orthogonal. Σ is diagonal.

The pseudo-inverse of A is $A^+ = V\Sigma^+U^T$.

Note that the SVD pseudo-inverse has one formulation which makes it a nice for applications which may be deficient in both row and column rank.

21.6.5 Weighted Least Squares

Traditional least squares is formulated by minimizing using the normal inner product:

$$x^T y = \sum_i x_i y_i.$$

Let $x, y \in R^n$. No weights are referred to as uniform weighting. Non-uniform weights are just termed as weights. If the inner product is weighted:

$$\langle x, y\rangle = \sum_{i=1}^{n} x_i y_i q_i = x^T Q y$$

where Q is a $n \times n$ square matrix then what is

least squares solution to $Ax = b$? One simple modification to the previous least squares process is required. We multiply both sides by the weight matrix Q:

$$QAx = Qb$$

then follow the earlier derivation:

$$A^T Q A x = A^T Q b.$$

Assuming that $A^T Q A$ is full rank,

$$x = \left(A^T Q A\right)^{-1} A^T Q b.$$

The matrix Q is any matrix for which the inner product above is a valid. However, we will often select Q as a diagonal matrix containing the reciprocals of the variances (the reason shown below in the covariance computation):

$$Q = \begin{pmatrix} q_1 & 0 & \dots & 0 & 0 \\ 0 & q_2 & \dots & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & 0 & q_{n-1} & 0 \\ 0 & 0 & 0 & 0 & q_n \end{pmatrix} = \begin{pmatrix} 1/\sigma_1^2 & 0 & \dots & 0 & 0 \\ 0 & 1/\sigma_2^2 & \dots & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & 0 & 1/\sigma_{n-1}^2 & 0 \\ 0 & 0 & 0 & 0 & 1/\sigma_n^2 \end{pmatrix}.$$

Assume that you have an x-y data set, Figure Fig. 21.6. Using the process above we compute the uniformly weighted least squares fit to a line, shown in blue, and the weighted least squares fit to a line, shown in green, Figure Fig. 21.7. The weight function weights more heavily towards the origin (using $w_i = 1.0/i^3$). In this example, the weights are scaled so the sum of the weights is one.

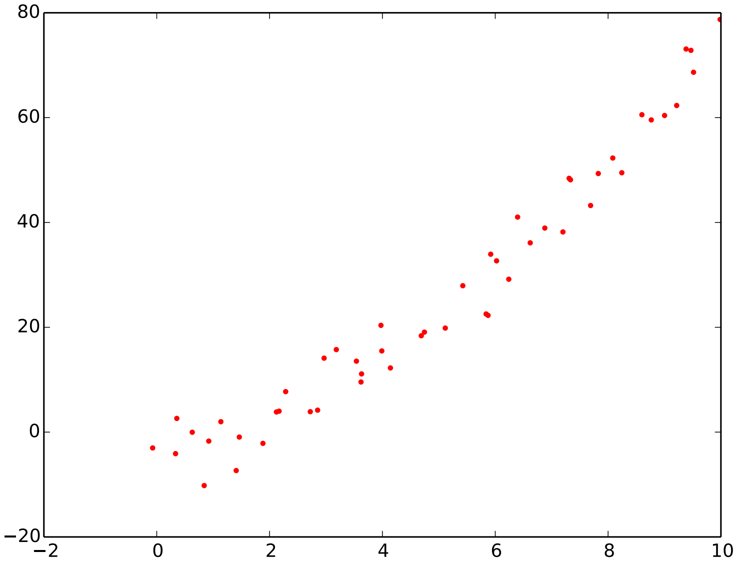


Fig. 21.6: Sample noisy data to fit a line.

21.7 Probability

Let X denote a random variable. Let P denote the probability that X takes on a specific value x: $P(X = x)$. If X takes on discrete values we say that X is a discrete variable. If X takes on continuous values we say that X is a continuous variable.

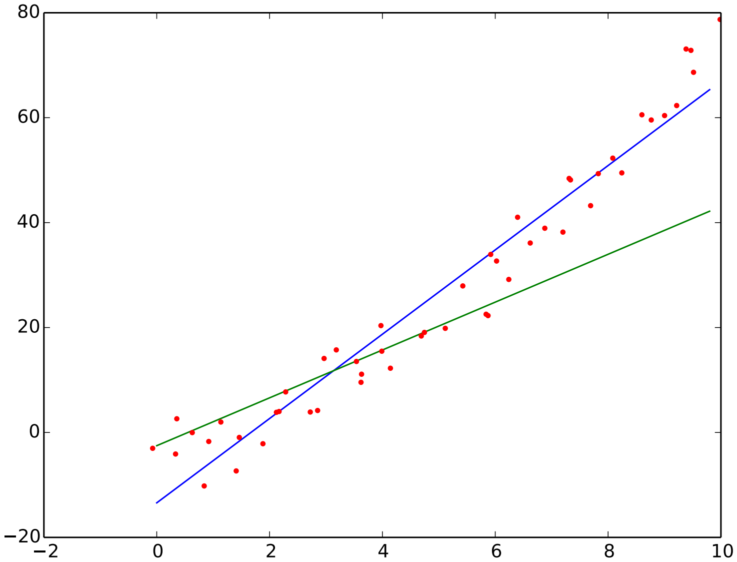


Fig. 21.7: Least squares line fit. Uniform weighting in blue and weighted to the origin in green.

Normally $P(X = x)$ makes sense for discrete spaces and we use $P(x_1 < X < x_2)$ for continuous spaces. For continuous spaces we define the probability density function (pdf) $p(x)$ in the following manner:

$$P(x_1 \leq X \leq x_2) = \int_{x_1}^{x_2} p(x)\, dx$$

21.7.1 Uncertainty and Distributions

Recall that random variables are drawn from some probability distribution function. Often these are the normal distributions seen in many areas of the sciences, but can be any shape as long as the area under the curve is one. Specifically, the normal distribution is given by

$$p(x) = \frac{1}{\sqrt{2\pi}\,\sigma} e^{-(x-\mu)^2/2\sigma^2}$$

where μ is the mean and σ^2 is the variance (σ is the standard deviation). For multivariate distributions (vector valued random variables) we can exend to

$$p(x) = \frac{1}{(2\pi)^{n/2}\sqrt{\det(\Sigma)}} e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)}$$

where μ - mean vector, Σ - covariance matrix (symmetric positive definite).

Let X, Y be two random variables, the joint distribution is

$$P(x, y) = P(X = x \text{ and } Y = y).$$

We say the the variables are independent if

$$P(x, y) = P(x)P(y)$$

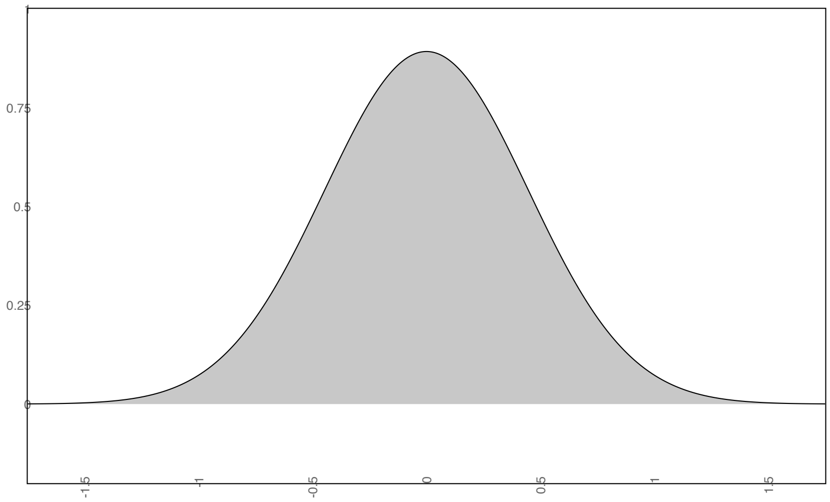


Fig. 21.8: Probability Distribution Function

Conditional probability: what is the probability of x if we know y has occurred? Denoted $P(x|y)$,

$$P(x|y) = \frac{P(x,y)}{P(y)}$$

If they are independent

$$P(x|y) = \frac{P(x,y)}{P(y)} = \frac{P(x)P(y)}{P(y)} = P(x)$$

Total probability (relax the uppercase formalism)

$$p(x) = \sum_y p(x|y)p(y) \quad \left[= \int_Y p(x|y)p(y)\,dy \right]$$

Bayes Rule (way to invert conditional probabilities)

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)}$$

Expectation or the mean or average for a distribution is given by

$$E(x) = \sum xp(x) \quad \left[= \int_X xp(x)\,dx \right]$$

Moments for a distribution are given by

$$\tilde{\mu_r} = E(x^r) = \int_X x^r p(x)\,dx$$

$$\mu = \tilde{\mu_1} = \quad \text{Mean - expected value}$$

Moments about the mean

$$\mu_r = \int_X (x-\mu)^r p(x)\,dx$$

Second moment about the mean is called the *Variance*: $\mu_2 = \sigma^2$, where σ is called the *Standard Deviation*. Note that variance $= E[(x-\mu)^2]$ and covariance $E(X \cdot Y) - \mu\nu$

where μ, ν are the means for X and Y.

The **Covariance** Matrix is given by $\Sigma =$

$$\begin{pmatrix} E[(x_1-\mu_1)(x_1-\mu_1)^T] & \dots & E[(x_1-\mu_1)(x_n-\mu_n)^T] \\ \dots & \ddots & \dots \\ E[(x_n-\mu_n)(x_1-\mu_1)^T] & \dots & E[(x_n-\mu_n)(x_n-\mu_n)^T] \end{pmatrix}$$

$$= E[(x-\mu)(x-\mu)^T]$$

There are many terms to describe the variance of a set of random variables. Variance, covariance and cross-variance, variance-covariance are a few example terms. We will use variance for scalar terms and covariance for vector terms.

Sample covariance

If you know the population mean, the covariance is given by

$$Q = \frac{1}{N}\sum_{k=1}^{N}(x_k - E(x))(x_k - E(x))^T$$

and if you don't know the mean the covariance is given by

$$Q = \frac{1}{N-1}\sum_{k=1}^{N}(x_k - \overline{x})(x_k - \overline{x})^T$$

Note: $(x_1 - \overline{x})$, $(x_2 - \overline{x})$, $(x_2 - \overline{x})$ has $n-1$ residuals (since they sum to zero).

CHAPTER TWENTYTWO

ROS - THE ROBOT OPERATING SYSTEM

The official ROS website defines ROS as follows:

ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more.

We present a brief summary of ROS, specifically ROS2. It should be noted that this introduction is in no way complete, as it does not convey the sheer power and complexity of ROS/ROS2, and the ways in which it goes about doing so. The reader is strongly encouraged to look at the ROS2 tutorials . ROS2 was in beta for most of 2017 until Ardent was released in Dec 2017. Currently, there is not a significant collection of ROS2 information. ROS (ROS1) does have some good references and there is some overlap in concepts. Because of the size of the install base, ROS1 is not leaving anytime soon. A couple of very good recent texts on ROS are [OKane14] and [QGS15]. For the remainder of this text we will only write "ROS" but unless explicitly stated we will mean ROS2.

It is slightly misleading that ROS includes the phrase *operating system* in the title. ROS itself is not an operating system in the traditional sense, but it is much more than just a piece of software. The many components combine to form an ecosystem of software which warrants its title but is best thought of as middleware. While on the "not" topic, ROS is not a programming language, not an IDE (integrated development environment) and not just a library of robotics codes.

As mentioned, package management and hardware abstraction are just a couple of features under the ROS umbrella, which support the communication framework around which ROS is based. The intent is to create a universal system which promotes code reuse and sharing among multiple robotic platforms, operating systems, and applications as well as small program footprints and efficient scaling. These pillars form the core goals of ROS as a whole.

Next, we introduce the simulation tool, Veranda, and cover the basic elements to simulate robot motion. Veranda is a two dimensional simulator. It was designed to have a low barrier to entry and overall ease of use. Veranda uses Box2D, a 2D physics engine for more realistic interactions between objects. It is not intended for very high precision work (and does not support 3D). For those applications, we suggest Gazebo (see appendix).

22.1 ROS2

ROS1 and 2's native habitat is Ubuntu. ROS2 does install and run on on Windows and Mac, and, there have been efforts to port ROS1 to Windows or OSX. The main simulation tool we will introduce later, Veranda, has not been built for Mac or Windows as of June, 2018, so in the next chapter we must use Linux but can use any OS for this chapter. Using Ubuntu might be the best choice for the long run, however.

There are several ways to approach getting a Linux install of ROS2. A standalone Linux system is the easiest. The author has had good success with a virtual machine (Parallels on OSX). You can also use for this chapter the OSX or Windows distribution. Whatever you select, the next step is to install ROS2.

Installation instructions can be found at ROS2 Install. Please do this now if not already completed. We will review the instructions here. The final authority on ROS (installation and other) is OSRF. The instructions below can and will become out of date. They are included here so we can discuss the steps.

22.1.1 Installing ROS2 via Debian Packages

Copied from the ROS2 install page

As of Beta 2 we are building Debian packages for Ubuntu Xenial. They are in a temporary repository for testing. The following links and instructions reference the latest release - currently ardent.

Resources: - Jenkins Instance - Repositories - Status Pages (amd64, arm64)

22.1.2 Setup Sources

To install the Debian packages you will need to add our Debian repository to your apt sources. First you will need to authorize our gpg key with apt like this:

```
sudo apt update && sudo apt install curl
curl http://repo.ros2.org/repos.key | sudo apt-key add -
```

And then add the repository to your sources list:

```
sudo sh -c 'echo "deb [arch=amd64,arm64] http://repo.ros2.org/ubuntu/main␣
↪xenial main" > /etc/apt/sources.list.d/ros2-latest.list'
```

22.1.3 Install ROS 2 packages

The following commands install all `ros-ardent-*` package except `ros-ardent-ros1-bridge` and `ros-ardent-turtlebot2-*` since they require ROS 1 dependencies. See below for how to also install those.

```
sudo apt update
sudo apt install `apt list "ros-ardent-*" 2> /dev/null | grep "/" | awk -F/ '
↪{print $1}' | grep -v -e ros-ardent-ros1-bridge -e ros-ardent-turtlebot2- |␣
↪tr "\n" " "`
```

22.1.4 Environment setup

```
source /opt/ros/ardent/setup.bash
```

If you have installed the Python package argcomplete (version 0.8.5 or higher, see below for Xenial instructions) you can source the following file to get completion for command line tools like ros2:

```
source /opt/ros/ardent/share/ros2cli/environment/ros2-argcomplete.bash
```

22.1.5 (optional) Install argcomplete >= 0.8.5 on Ubuntu 16.04

If you need to install argcomplete on Ubuntu 16.04 (Xenial), you'll need to use pip, because the version available through apt-get will not work due to a bug in that version of argcomplete:

```
sudo apt install python3-pip
sudo pip3 install argcomplete
```

Choose RMW implementation

By default the RMW implementation FastRTPS is being used. By setting the environment variable RMW_IMPLEMENTATION=rmw_opensplice_cpp you can switch to use OpenSplice instead.

22.1.6 Additional packages using ROS 1 packages

The ros1_bridge as well as the TurtleBot demos are using ROS 1 packages. To be able to install them please start by adding the ROS 1 sources as documented here

If you're using Docker for isolation you can start with the image ros:kinetic or osrf/ros:kinetic-desktop This will also avoid the need to setup the ROS sources as they will already be integrated.

Now you can install the remaining packages:

```
sudo apt update
sudo apt install ros-ardent-ros1-bridge ros-ardent-turtlebot2-*
```

22.1.7 Fundamental ROS Entities

Most of the concepts in ROS are based around a set of fundamental entities, which will be discussed in this section. Understanding ROS, the challenges which ROS attempts to overcome and the challenges of using ROS are not possible without a firm understanding of these materials.

22.1.8 Package

Pieces of software are, as the name suggests, known as packages in ROS. Each package carries out a single or group of tightly related functions. Packages need not include code at all; some packages are simply meta-packages for the purpose of ensuring that other packages are present on the system, while other packages include startup routines for robots or 3-D physical definitions which are used to render robots in simulation.

Packages may be placed in different ROS environments, and a number of these environments may be exposed simultaneously, allowing developers to switch between different groups of packages with ease.

22.1.9 Node

The node is quite possibly the single most important concept to understand when discussing the use of ROS. Nodes are essentially vertices in the computation graph that is implemented in ROS, and all of the computation in ROS occurs in a node.

It is considered to be the best form in ROS for a single node to carry out a single task. This helps to promote code reuse, as the node could then be used to perform the same task as part of a completely different system, ideally without any modification of the code.

It is quite common to see hundreds of nodes as part of a single ROS environment, and it is also common of many of them to be active simultaneously. For instance:

22.1.10 Master

The ROS master provides a registration system for the nodes on a ROS system, among other services. Think of it as the operator of a phone network. When a node requests information, it asks the ROS master to connect it to someone who can provide that information. The ROS master doesn't actually give the information to the node, it simply tells it where it may be found. This communication happens over an XMLRPC protocol.

A node does not typically communicate with the master once it has finished initializing and is sending and receiving data. It does, however, talk to the master whenever it needs a new data stream or parameter information.

Worth noting is that while communication between the nodes and the master is sparse, loss of communication with the master can be devastating to a ROS system. If the master were to crash or become otherwise unavailable, the entire ROS system would likely fail if any master communication were to be attempted. The node which tried to communicate with the master would fail, likely causing a domino effect in nodes trying to request data streams that are sequentially becoming unavailable.

In general, every ROS system must have exactly one master. There exist methods of inter-master communication, but there is no built-in methodology for this.

22.1.11 Message

Any data or information that is exchanged between nodes is known as a message, which is defined as a combination of primitive data types or other messages. Some messages include a common header, which includes a sequence number, time stamp and a physical origin known as a frame ID. For example, a *Twist* message contains 6 Float64 values; a 3-D vector of linear velocities as well as a 3-D vector of angular velocities. This message is widely used to describe the velocity of a body in ROS.

Any message defined in ROS is available in any of the supported language in ROS. Once a node sends a message over ROS, the message can be interpreted by another node even if the nodes are not written in the same language or are running on the same operating system. On that note, messages could be considered to be "data contracts" among nodes.

22.1.12 Topic

While a node may request a certain type of data from the master, it is possible that multiple nodes could provide data of that type. The use of "topics" is necessary to uniquely identify a data source to other nodes. Therefore, when a node notifies the master of available data, it must provide a topic name for that data. A connection between nodes is only ever established if the nodes agree on a data type and a topic.

Topics can be thought of as being similar to a telephone number. When a node registers its "number" with the master (a process known as advertising), the master notes the message type that the "number" corresponds to as well as the network address of the node that is providing it. When another node "calls" that number (a process known as subscribing), the master looks to see if there is a registered node providing the requested message type, and tells the node what the address of the other node is. The direct connection between the nodes is then established and the data transfer begins.

It should be noted that while this seems to indicate that a topic corresponds to a single server-client relationship, the topic system allows for multiple subscribers as well as multiple publishers. Therefore the relationship is generally referred to as publisher-subscriber, or "pub-sub."

22.1.13 Service

The publisher-subscriber (pub-sub) model is not always appropriate for all types of data, and the service system exists in ROS to fill in the gap. Services are, like pub-sub messages, exposed in ROS over topics. The group of topic names is separate from the pub-sub topics, but the structure remains.

Services are unique in that they are based on a call-and-response model instead of pub-sub. A node can not only request information from another node, but it can include a message to the other node containing information about the request. The remote node then responds with a single message back to the node that initiated the service call.

Services are useful in many ways, but should not be over-used. Each time a service call is made, the node must request the address of the service provider from the master. If service calls are made frequently, a bottleneck could form in the computation graph at the master.

22.2 ROS Communication

OSRF provides tutorials on ROS, http://wiki.ros.org/ROS/Tutorials. Some of that material is repeated here and much greater detail can be found in the texts referenced earlier. After installing ROS and setting up the environment, one can get started using ROS if you plan to use Python. One can use several other languages for development. Some of those languages may require additional setup. For example, C/C++ require additional setup and so you will need to learn about the build environment, ROS packages, etc. These commands are covered in the Beginner Level Tutorials online. Our goal for this section is to illustrate basic ROS communications which requires some infrastructure. We will return to the administrative side of ROS after some simple coding examples. Some experience with Linux and the command line is useful here.

The terminal or command window brings up the shell or command interpreter. For those not familiar with linux, this is like DOS. The shell program is called bash. There are good online references for bash. The appendix has a brief introduction.

As mentioned above the basic form of ROS communication is the Publish-Subscribe mechanism. To see this in action, you need to do two things: (1) run a subscriber, (2) run a publisher. The "pubsub" communication will be shown in Python. ROS2 is only available for Python3, so if you don't have Python3, please load it now.

22.2.1 Simple Publisher-Subscriber Example

Our first example is going to send a single text message from one program to another. This material is adapted from the basic examples on the ROS2 Github site. Bring up two more terminal windows and type python in each:

```
alta:~ jmcgough$ python3
Python 3.6.5 (default, Apr 25 2018, 14:26:36)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

In one window type:

```
>>> import rclpy
>>> from std_msgs.msg import String
>>> rclpy.init(args=None)
>>> pnode = rclpy.create_node('minimal_publisher')
>>> publisher = pnode.create_publisher(String, 'topic')
```

The first step imports ROS library. The second step brings in the ROS message type *String*. Next we start up ROS2. The fourth line creates a ROS node and names it. The fifth line establishes ourself as a publisher and sets the topic name and datatype. In this example, the topic name is *topic* and the topic datatype is the ROS standard message type *String*. So, your python shell is now a ROS node that can publish on the established topic.

In the second window, type:

```
>>> import rclpy
>>> from std_msgs.msg import String
>>> rclpy.init(args=None)
>>> node = rclpy.create_node('min_sub')
>>> def chatter_callback(msg):
...   global node
...   node.get_logger().info('This is what I heard: "%s"' % msg.data)
...
>>> subscription = node.create_subscription(String, 'topic', chatter_callback)
```

(continues on next page)

(continued from previous page)

```
>>> while rclpy.ok():
...   rclpy.spin_once(node)
```

The first four steps are the same as above. The fifth line defines the callback function. This function is called when a message is published on the topic that our node has subscribed to. Following the callback function, we subscribe to the topic and define the callback function to handle the message that has arrived. The while at the end places the node into an event loop to capture the message.

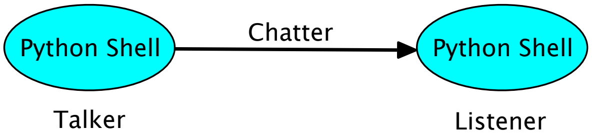


Fig. 22.1: Simple PubSub example

Now the fun step. In the first python window (the one that has the Publisher line), type:

```
>>> msg = String()
>>> msg.data = "Hello"
>>> publisher.publish(msg)
```

You should see on the Subscriber window:

```
[INFO] [min_sub]: This is what I heard: "Hello"
```

You have successfully sent a message from one process (program) to another. There is a similarity between writing to a topic and writing to a file. The line

```
publisher = pnode.create_publisher(String, 'topic')
```

is similar to opening a file named topic and returning the file descriptor `publisher`. The full power of Python is available; a simple extension can produce multiple messages. He is a sample of a loop containing a publish.

```
>>> for i in range(5):
...   msg.data = "Message number " + str(i)
...   publisher.publish(msg)
...
>>>
```

This results with the text in the other window:

```
[INFO] [min_sub]: This is what I heard: "Message number 0"
[INFO] [min_sub]: This is what I heard: "Message number 1"
[INFO] [min_sub]: This is what I heard: "Message number 2"
[INFO] [min_sub]: This is what I heard: "Message number 3"
[INFO] [min_sub]: This is what I heard: "Message number 4"
```

We can extend this example so that our talker is talking to two listening programs. First we modify our talker to *talk* on two topics, by adding the line:

```
>>> publisher2 = pnode.create_publisher(String, 'topic2')
```

Next we create a new program to listen to the new optic. Create a new terminal window and enter:

```
import rclpy
from std_msgs.msg import String
rclpy.init(args=None)
node = rclpy.create_node('min_sub2')
def chatter_callback(msg):
...   global node
...   node.get_logger().info('This is what I heard: "%s"' % msg.data)
...

subscription = node.create_subscription(String, 'topic2', chatter_callback)
>>> while rclpy.ok():
...   rclpy.spin_once(node)
...
```

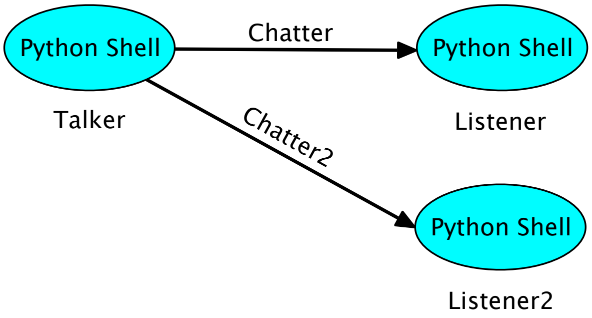


Fig. 22.2: Simple PubSub example cont.

From the publisher python process, setup the new topic

```
>>> publisher2 = pnode.create_publisher(String, 'topic2')
```

and now you can send to the new node:

```
>>> msg.data = "Second topic Hello"
>>> publisher2.publish(msg)
```

or you can send to the old node:

```
>>> msg.data = "First topic Hello"
>>> publisher.publish(msg)
```

You should see the output on the two separate listener programs. One more modification will illustrate these ideas. The previous examples got us up and running. At this point, it is easy to make small changes and run brief experiments in the command interpreter.

22.2.2 Python ROS Programs

There is a limit to how convenient it is using the interpreter directly. The Python interpreter is very handy for developing code and experimenting with parameters. However, as the code base grows it makes sense to move over to placing the code in a file and running it from the bash terminal. For the rest of the examples, we switch to a more traditional programming style. This means the code is in a file which will be executed as a script and not as individual commands. A bit more like what you do with C, Java or normal Python usage.

The main difference it makes at this stage is that you no longer have the event loop which the Python command interpreter gave you. You will need to supply some type of event loop or have all the commands entered and timed as needed. We will focus on the former. So the last example above will be modified with a small loop added and the three programs will be listed below. If you are reading this from an electronic version, you can then cut and paste into your editor. Otherwise the code can be obtained from CODE REPO LINK HERE!!!

Place the code in a file and at the top of the file enter

```
#!/usr/bin/env python3
```

The #! (called shebang) in the first two bytes tells the operating system to use the python interpreter for the file. One new issue is that the process will terminate after the last command. We did not need to worry about this when we were running in the interpreter since it was running an event loop (waiting for our input). So we need to have something to keep the process going. A simple open loop has been added to the publisher for the demonstration. On the subscriber side, we also need a way to keep the process running. ROS provides some commands that allow us to set up the event loop. We will combine a while loop with `rclpy.spin_once(node)` which gives us an infinite loop and waits for an event like a message published on a topic.

Based on the couple of modifications above, the simple publisher and subscriber example can be written as the following Python programs, Listing 22.1, Listing 22.2.

Listing 22.1: Two topic publisher example

```
#!/usr/bin/env python3
import rclpy
from std_msgs.msg import String

rclpy.init(args=None)

node = rclpy.create_node('publisher')
pub1 = node.create_publisher(String, 'topic1')
pub2 = node.create_publisher(String, 'topic2')
msg = String()


while True:
  message = input("> ")
  if message == 'exit':
     break
  msgarr = message.split(',')
```

(continues on next page)

(continued from previous page)

```
  ch = int(msgarr[1])
  msg.data = msgarr[0]
  if ch == 1:
     pub1.publish(msg)
  if ch == 2:
     pub2.publish(msg)


node.destroy_node()
rclpy.shutdown()
```

Listing 22.2: Subscriber 1

```
#!/usr/bin/env python3
import rclpy
from std_msgs.msg import String

def chatter_callback(msg):
   global node
   node.get_logger().info('This is what I heard: "%s"' % msg.data)

rclpy.init(args=None)
node = rclpy.create_node('min_sub1')
subscription = node.create_subscription(String, 'topic1', chatter_callback)
while rclpy.ok():
   rclpy.spin_once(node)
```

Listing 22.3: Subscriber 2

```
#!/usr/bin/env python3
import rclpy
from std_msgs.msg import String

def chatter_callback(msg):
   global node
   node.get_logger().info('This is what I heard: "%s"' % msg.data)

rclpy.init(args=None)
node = rclpy.create_node('min_sub2')
subscription = node.create_subscription(String, 'topic2', chatter_callback)
while rclpy.ok():
   rclpy.spin_once(node)
```

Cut and paste these into three different files, pub.py, sub1.py and sub2.py, and run in three different terminals. In pub.py one can type your message, then comma, then the topic number (1 or 2): *message, number* .

Don't forget to make the two files executable by

```
chmod +x <filename>
```

One can have multiple communication lines between nodes. We will add a third topic to the publisher and

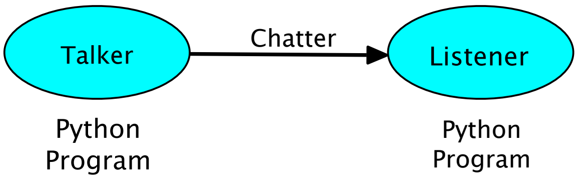


Fig. 22.3: Simple PubSub Program example. Computing the wheel velocties in one program and sending the commmands to another program to implement.

have sub1 subscribe to it. The new versions of the publisher and sub1 are given below.

Listing 22.4: Multi-topic publisher

```
#!/usr/bin/env python3
import rclpy
from std_msgs.msg import String
from std_msgs.msg import Int16

rclpy.init(args=None)

node = rclpy.create_node('publisher')
pub1 = node.create_publisher(String, 'topic1')
pub2 = node.create_publisher(String, 'topic2')
pub3 = node.create_publisher(Int16, 'topic3')
msg = String()
var = Int16()

while True:
  message = input("> ")
  if message == 'exit':
     break
  msgarr = message.split(',')
  ch = int(msgarr[1])
  msg.data = msgarr[0]
  if ch == 1:
     pub1.publish(msg)
  if ch == 2:
     pub2.publish(msg)
  if ch == 3:
     var.data = int(msgarr[0])
     pub3.publish(var)


node.destroy_node()
rclpy.shutdown()
```

and for sub1.py we modify

Listing 22.5: Multi-topic subscriber

```
#!/usr/bin/env python3
import rclpy
from std_msgs.msg import String
from std_msgs.msg import Int16

def chatter_callback(msg):
   global node
   node.get_logger().info('This is what I heard: "%s"' % msg.data)

def chatter_callback2(msg):
   global node
   node.get_logger().info('This is what I heard: "%s"' % msg.data)


rclpy.init(args=None)
node = rclpy.create_node('min_sub1')
subscription = node.create_subscription(String, 'topic1', chatter_callback)
subscription = node.create_subscription(Int16, 'topic3', chatter_callback2)

while rclpy.ok():
   rclpy.spin_once(node)
```

Then on the publisher enter: *42, 3* . You should see the number 42 echoed on the terminal running sub1.

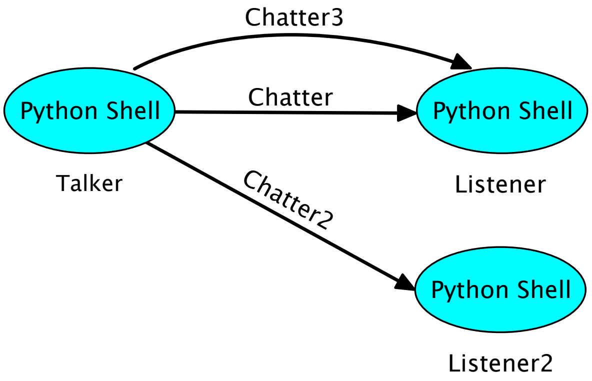


Fig. 22.4: Simple PubSub example cont.

To see what topics are defined, you can get a list of them:

```
alta:Desktop jmcgough$ ros2 topic list
/topic1
/topic2
/topic3
```

As of early 2018, the topic list command was under development. This tool is only accurate for nodes and topics on a single computer. Current development by OSRF is to make the topic list work on distributed nodes.

You can listen in on a topic using the rostopic command.

```
alta:Desktop jmcgough$ ros2 topic echo /topic1
```

Into the publisher python window type:

```
> Hello, 1
```

and you will see in the rostopic command window:

```
data: Hello
```

Table 22.1: Data Types

Bool	Byte	ByteMultiArray
Char	ColorRGBA	Duration
Empty	Float32	Float32MultiArray
Float64	Float64MultiArray	Header
Int16	Int16MultiArray	Int32
Int32MultiArray	Int64	Int64MultiArray
Int8	Int8MultiArray	MultiArrayDimension
MultiArrayLayout	String	Time
UInt16	UInt16MultiArray	UInt32
UInt32MultiArray	UInt64	UInt64MultiArray
UInt8	UInt8MultiArray	...

Often we need to publish a message on a periodic basis. It is possible to place a delay via python sleep in the publishing loop:

```
while True:
  message = input("> ")
  if message == 'exit':
     break
  time.sleep(delay)
```

The sleep command will introduce a delay. This approach will enforce at least that time interval, but not exactly that time interval. The process shares the cpu and longer delays can arise when other processes slow down the system. Some robotics applications require that the time interval is accurate within some constraint.

To increase the timing accuracy, ROS supports an interrupt based method. This approach sets a timer which raises an interrupt. That interrupt causes a function to be called, known as an interrupt handler. Sample code is provided below (adapted from the ROS2 example programs).

```
import rclpy
from std_msgs.msg import String
def timer_callback():
    global i
    msg.data = 'Hello World: %d' % i
    i += 1
```

(continues on next page)

(continued from previous page)

```
    node.get_logger().info('Publishing: "%s"' % msg.data)
    publisher.publish(msg)


rclpy.init(args=None)
node = rclpy.create_node('publisher')
publisher = node.create_publisher(String, 'topic1')

msg = String()
i = 0
timer_period = 0.5  # seconds
timer = node.create_timer(timer_period, timer_callback)

rclpy.spin(node)

node.destroy_timer(timer)
node.destroy_node()
rclpy.shutdown()
```

22.2.3 Publisher - Subscriber for the Two Link Kinematics

Assume that you want to control a two link manipulator using ROS. To do this you will need to describe the path you want to travel in the workspace. So, the first step is to produce the workspace domain points. The you want to ship those points to the inverse kinematics to find the corresponding angles that set the manipulator end effector in the workspace points you desire.

For this example, we are going to create the workspace data, and then publish it with the first node. The next node will subscribe and convert (x, y) data to angle data. That node will then publish to a node that will run the forward kinematics to check the answer. To make this look like a stream of points, a delay is placed

The node that creates the workspace points is given in Listing 22.6. We illustrate with the curve $x(t) = 5\cos(t) + 8$, $y(t) = 3\sin(t) + 10$. The interval is discretized into intervals of 0.1. The (x, y) points are published on the topic named /WorkspacePath.

Listing 22.6: Workspace Points

```
#!/usr/bin/env python
import rclpy
from std_msgs.msg import Float32
from std_msgs.msg import Int8
import math

def timer_callback():
  global t, pubx, puby
  x = 5.0*math.cos(t) + 8.0
  y = 3.0*math.sin(t) + 10.0
  xval.data = x
  yval.data = y
  node.get_logger().info('Publishing: "%f" , "%f" ' % (x,y) )
  pubx.publish(xval)
```

(continues on next page)

(continued from previous page)

```
  puby.publish(yval)
  t = t+step

rclpy.init(args=None)
node = rclpy.create_node('Workspace')
pubx = node.create_publisher(Float32, 'WorkspacePathX')
puby = node.create_publisher(Float32, 'WorkspacePathY')
step = 0.1
t = 0.0
xval = Float32()
yval = Float32()

timer_period = 0.5  # seconds
timer = node.create_timer(timer_period, timer_callback)

rclpy.spin(node)

node.destroy_timer(timer)
node.destroy_node()
rclpy.shutdown()
```

The next stage of the process is to convert the points from the workspace to the configuration space using the inverse kinematic equations. The program performs the inverse kinematics and then publishes the results on the topic /ConfigspacePath. The code is given in Listing 22.7.

Listing 22.7: Inverse Kinematics Code

```
#!/usr/bin/env python
import rclpy
from std_msgs.msg import Float32
from std_msgs.msg import Int8
import math

def callbackX(data):
    global x, y
    x = data.data

def callbackY(data):
    global x, y
    y = data.data
    convert(x,y)

def convert(x,y):
    global pub, a1, a2
    d = (x*x + y*y - a1*a1 - a2*a2)/(2*a1*a2)
    t2 = math.atan2(-math.sqrt(1.0-d*d),d)
    t1 = math.atan2(y,x) - math.atan2(a2*math.sin(t2),a1+a2*math.cos(t2))
    xval.data = t1
    yval.data = t2
    node.get_logger().info('Publishing: "%f" , "%f" ' % (t1,t2) )
    pubcx.publish(xval)
```

(continues on next page)

(continued from previous page)

```
    pubcy.publish(yval)


global x, y, a1, a2, pub
rclpy.init(args=None)
node = rclpy.create_node('InverseK')
subx = node.create_subscription(Float32, 'WorkspacePathX', callbackX)
suby = node.create_subscription(Float32, 'WorkspacePathY', callbackY)
pubcx = node.create_publisher(Float32, 'ConfigspacePathX')
pubcy = node.create_publisher(Float32, 'ConfigspacePathY')
xval = Float32()
yval = Float32()


#Initialize global variables
a1, a2 = 10.0, 10.0
x, y = 0.0, 0.0
while rclpy.ok():
   rclpy.spin_once(node)
```

Finally we would like to check our answer. The angle values from the last node are evaluated by the forward kinematics producing $(\tilde{x}, \tilde{y})$ values. These values are compared to the original (x, y) values. The two sets of values should agree closely. The code for the verification is given in Listing 22.8.

Listing 22.8: Inverse Kinematics Verification

```
#!/usr/bin/env python
import rclpy
from std_msgs.msg import Float32
from std_msgs.msg import Int8
import math


def callbackX(data):
    global t1, t2
    t1 = data.data

def callbackY(data):
    global t1, t2
    t2 = data.data
    convert(t1,t2)


 def convert(t1,t2):
     global a1, a2
     x = a1*math.cos(t1) + a2*math.cos(t1+t2)
     y = a1*math.sin(t1) + a2*math.sin(t1+t2)
     print (x, y)

global a1, a2
rclpy.init(args=None)
```

(continues on next page)

(continued from previous page)

```
node = rclpy.create_node('InverseKcheck')
subx = node.create_subscription(Float32, 'ConfigspacePathX', callbackX)
suby = node.create_subscription(Float32, 'ConfigspacePathY', callbackY)

#Initialize global variables
a1, a2 = 10.0, 10.0
t1, t2 = 0.0, 0.0

while rclpy.ok():
   rclpy.spin_once(node)
```

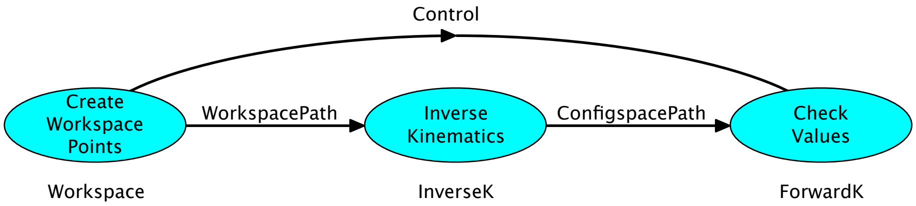


Fig. 22.5: Two Link Manipulator ROS example.

Although many devices produce data in a sequential manner, there are times when you have blocks of data. ROS provides a number of datatypes in both scalar and array form as well as some specialized messages for sending common data blocks such as position and pose updates. When it is possible, one can often get better performance out of sending arrays. This next example demonstrates how to send arrays. For this example we will send a block of 32bit integers which is the datatype `Int32MultiArray`.

Listing 22.9: Example of the MultiArray - Publisher

```
#!/usr/bin/env python
import rclpy
from std_msgs.msg import Int32MultiArray
rclpy.init(args=None)
node = rclpy.create_node('Talker')
pub = node.create_publisher(Int32MultiArray, 'Chatter')

a=[1,2,3,4,5]
myarray = Int32MultiArray(data=a)
pub.publish(myarray)
```

Listing 22.10: Example of the MultiArray - Subscriber

```
#!/usr/bin/env python
import rclpy
from std_msgs.msg import Int32MultiArray

def callback(data):
   print(data.data)
   var = data.data
   n = len(var)
```

(continues on next page)

(continued from previous page)

```
   for i in range(n):
     print(var[i])


rclpy.init(args=None)
node = rclpy.create_node('Subscriber')
sub = node.create_subscription(Int32MultiArray, 'Chatter', callback)

while rclpy.ok():
   rclpy.spin_once(node)
```

22.3 Tutorial: Driving a Robot in Veranda

This section of the book will walk you through the entire process of designing your own robot and programming it to drive and produce sensor feedback. The tutorial assumes you have ROS installed in the default location: `/opt/ros/ardent`.

22.3.1 Part 0: Install and Run Veranda

Veranda can be installed from Roboscience.org. Download and run the install script to set it up. The default install location will be `~/veranda`. This installer script will add the command `veranda` as a bash alias that will start Veranda using ROS.

```
> veranda
```

Note: The default RMW Implementation on linux is FastRTPS; however, the version of it included with ROS Ardent appears to have some issues, so this script will automatically switch to OpenSplice by doing `export RMW_IMPLEMENTATION=rmw_opensplice_cpp` before running the application.

22.3.2 Part 1: Build a Turtle

The first step to simulating robots is having a robot to simulate. When you are greeted with the Veranda application, select the ‘Editor’ button to open the editor.

In the editor, you can place robot components together to build your very own robot! Let’s start by adding a circular body…

Next, we’ll add a couple of wheels…

Now, you may be noticing that my robot looks much more square than yours; if you want to make sure the wheels are exactly where you want them, you can set their position properties to the exact coordinates you want. I made the wheels be exactly 0.6m to the left/right of the center, and 0m above it.

Now that you have a robot built, we need to load it into the simulation. Choose the ‘save’ button, and save your robot as `Turtle.json`. Don’t forget the `.json`! It will not be added automatically if you forget it.

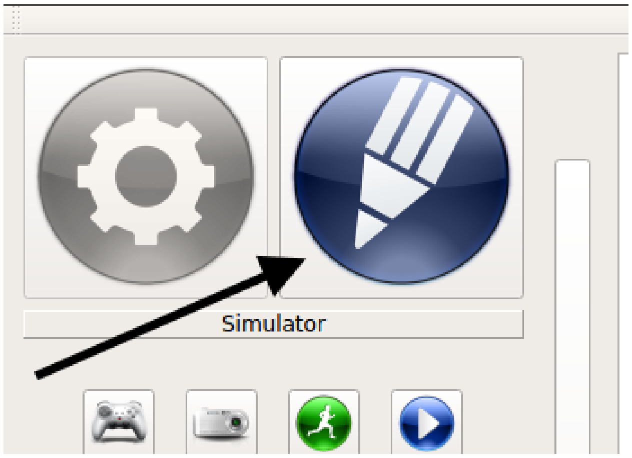


Fig. 22.6: The editor button

Next we have to switch back to simulator mode.

Now, we can load the robot into our toolbox on the right.

And once your robot is in the toolbox, you can add it to the simulation and position it wherever you want!

Finally, we can start the simulation with the 'play' button on the left.

Congratuations! You just simulated your first robot; it sat there, and did nothing. Next, we're going to write some code to make it move.

Tip: If you don't want to go through the trouble of saving your robot in a file and then loading it again, you can use the 'quick-add' button on the editor to put it directly in the toolbox, but beware, if you close Veranda, the robot will be lost forever!

22.3.3 Part 2: Drive your robot in a circle

Now that we have a robot designed, we need to write some code to control it and then connect that code to the simulation using ROS. First, we will pick names for the ROS topics we want to use. Select your turtle robot in the simulator, and then search through its properties for the topic settings for the wheels. Since I left my wheels named 'Fixed Wheel', I am looking for the properties called 'Fixed Wheel1/channels/input_speed', and 'Fixed Wheel2/channels/input_speed'. In my turtle, 'Fixed Wheel1' is on the left, and 'Fixed Wheel2' is on the right, so I named the ROS topics 'robot0/left_wheel' and 'robot0/right_wheel', respectively.

We also need to indicate that the wheels can be driven. Find the properties 'Fixed Wheel1/is_driven' and Fixed Wheel2/is_driven' and set them both to be 'true'

Tip: Having issues telling your wheels apart? They have a 'Name' property that can be changed in the editor to differentiate them better.

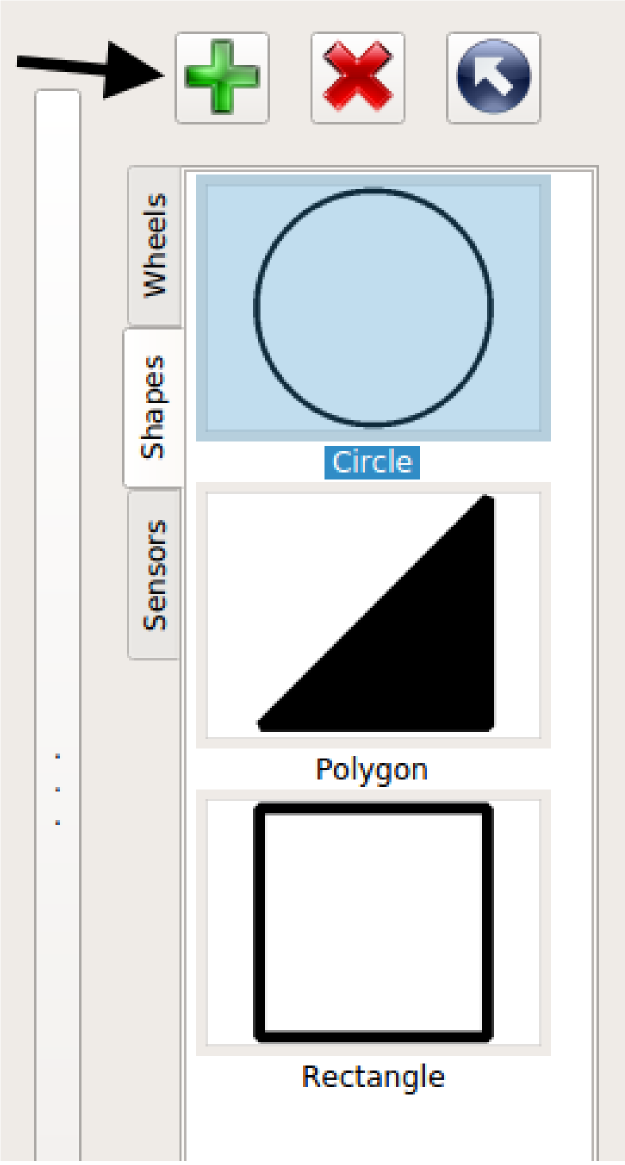


Fig. 22.7: To select the circle from the shapes tab, and press the green plus to add it

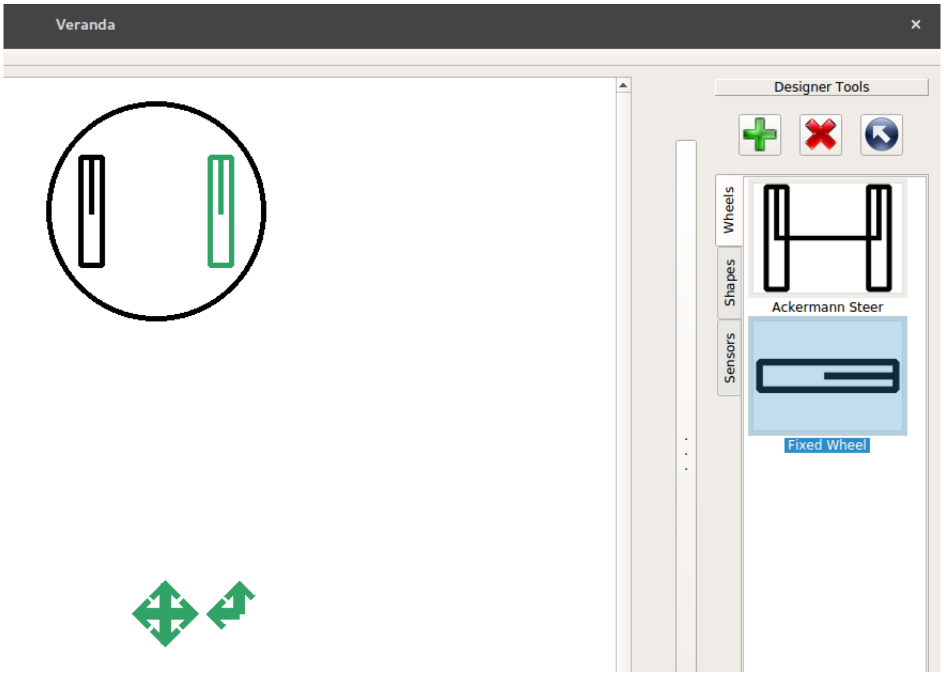


Fig. 22.8: Completed turtle bot

Tip: Don't want to have to set properties every time you start Veranda? You can set many properties in the editor and save their values along with the rest of the robot.

Now that the channels are set, we need to write some code to start driving the robot. To drive a differential robot in a circle, all we need to do is send a different speed command to each wheel; then they will drive that speed forever.

First, we need our python to import the `rclpy` module, and the Node type from that module

```
import rclpy
from rclpy.node import Node
```

Next, we need to import the message type that should be used to communicate to the wheels.

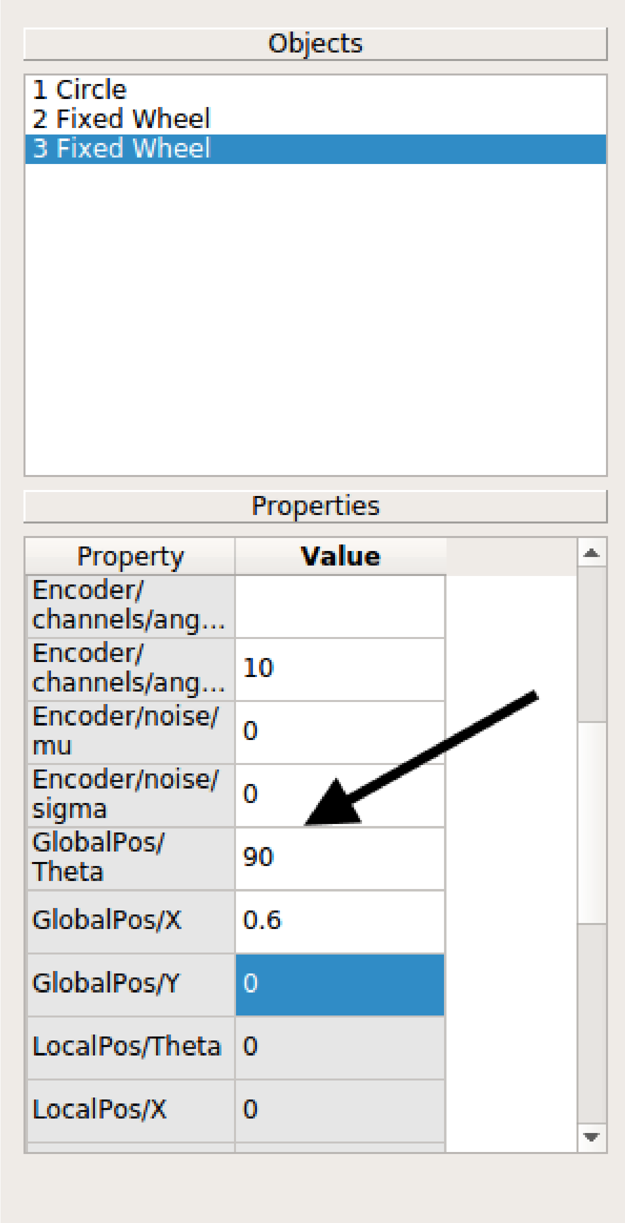
```
from std_msgs.msg import Float32
```

Now, we can initialize ROS and create a Node to publish from

```
rclpy.init()
node = Node("circle")
```

Once the node is created, we can create two publishers; one for each of the wheel topics

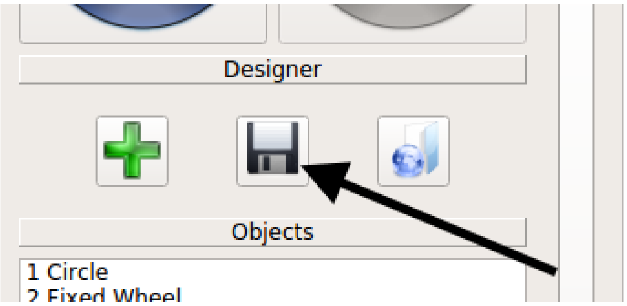
```
publeft = node.create_publisher(Float32, 'robot0/left_wheel')
pubright = node.create_publisher(Float32, 'robot0/right_wheel')
```



1 Circle
2 Fixed Wheel
3 Fixed Wheel

Property	Value
Encoder/ channels/ang…	
Encoder/ channels/ang…	10
Encoder/noise/ mu	0
Encoder/noise/ sigma	0
GlobalPos/ Theta	90
GlobalPos/X	0.6
GlobalPos/Y	0
LocalPos/Theta	0
LocalPos/X	0

Fig. 22.9: With a wheel selected, you can set properties for it



1 Circle
2 Fixed Wheel

Fig. 22.10: Save your robot by pressing the save button in the editor mode

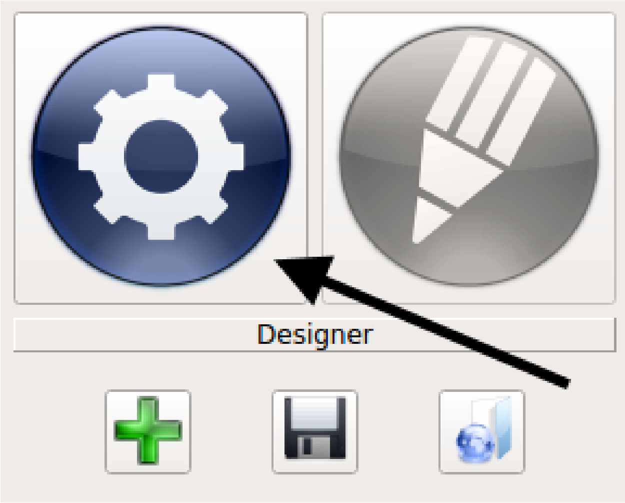


Fig. 22.11: The simulator button

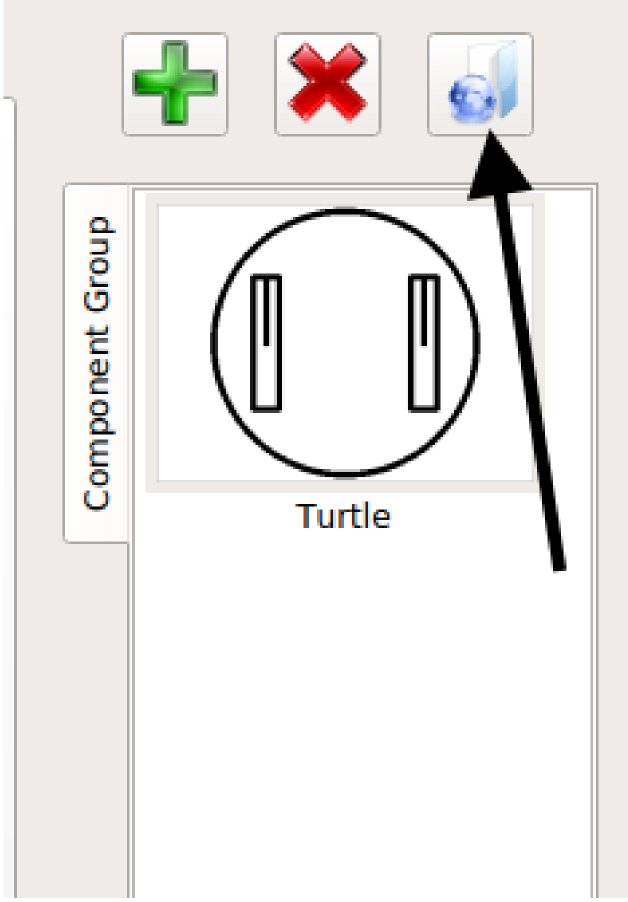


Fig. 22.12: Press the load button on the simulator toolbox to load a robot file

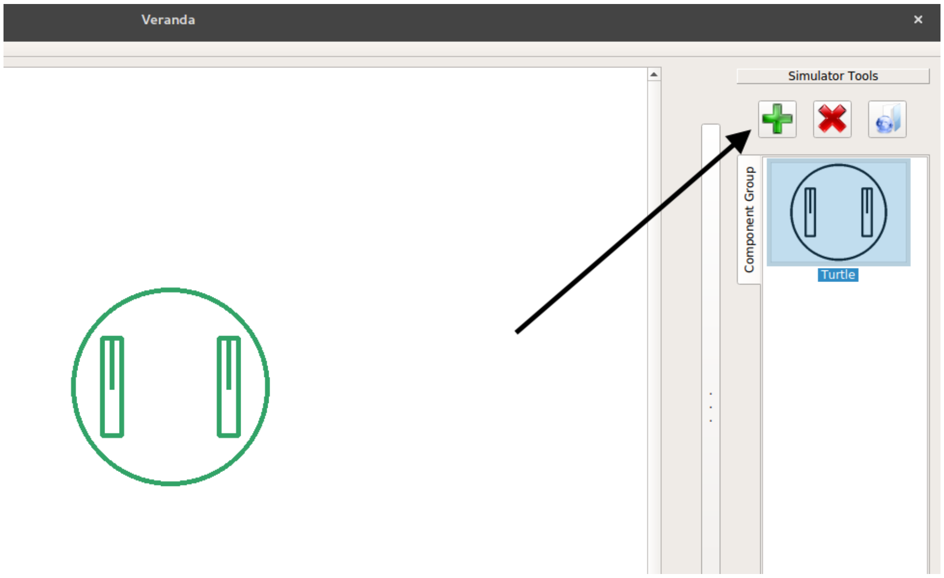


Fig. 22.13: Add robots from the toolbox by selecting them and pressing the green plus

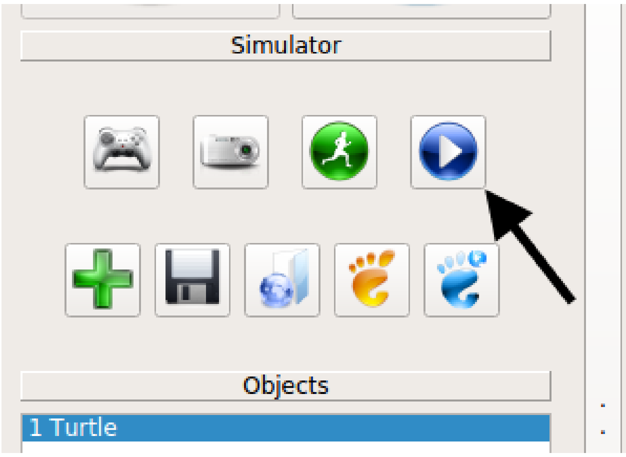


Fig. 22.14: The play button to start a simulation

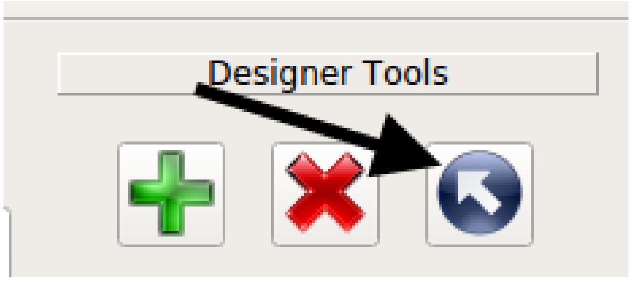
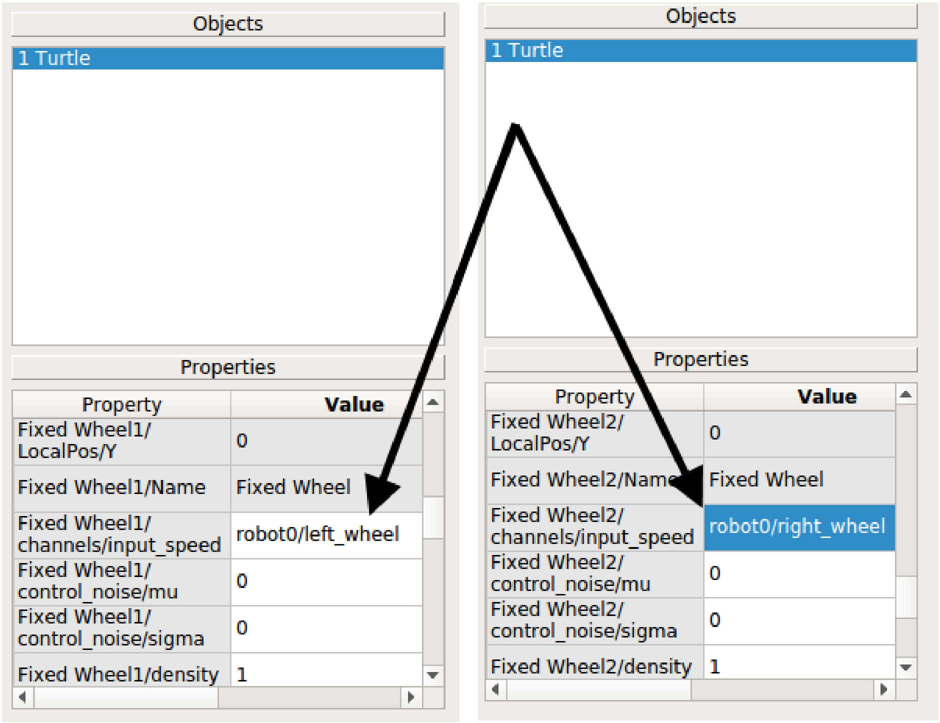


Fig. 22.15: The designer quick-add button



Objects

1 Turtle

Properties

Property	Value
Fixed Wheel1/LocalPos/Y	0
Fixed Wheel1/Name	Fixed Wheel
Fixed Wheel1/channels/input_speed	robot0/left_wheel
Fixed Wheel1/control_noise/mu	0
Fixed Wheel1/control_noise/sigma	0
Fixed Wheel1/density	1

Objects

1 Turtle

Properties

Property	Value
Fixed Wheel2/LocalPos/Y	0
Fixed Wheel2/Name	Fixed Wheel
Fixed Wheel2/channels/input_speed	robot0/right_wheel
Fixed Wheel2/control_noise/mu	0
Fixed Wheel2/control_noise/sigma	0
Fixed Wheel2/density	1

Fig. 22.16: Setting the wheel control topics

Finally we can send a command to each of the wheels. Let's create a Float32 message, and send it with different values to each wheel.

```
msg = Float32()

msg.data = 5.0
publeft.publish(msg)

msg.data = 10.0
pubright.publish(msg)
```

Note: This will command the wheels to drive 5 radians/second and 10 radians/second respectively.

However, if we run the code right now, the messages will not be sent; they have only been queued for publishing. To send them out of the application, we need to 'spin' the ROS node. Once we spin it, ROS will enter an infinite loop which sends queued messages and receives incoming ones.

```
rclpy.spin(node)

node.destroy_node()
rclpy.shutdown()
```

And there we have it! One python program to start driving a robot in a circle. Let's call it 'circle.py'

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import Float32

rclpy.init()
node = Node("circle")

publeft = node.create_publisher(Float32, 'robot0/left_wheel')
pubright = node.create_publisher(Float32, 'robot0/right_wheel')

msg = Float32()

msg.data = 5.0
publeft.publish(msg)

msg.data = 10.0
pubright.publish(msg)

rclpy.spin(node)

node.destroy_node()
rclpy.shutdown()
```

Now, all that's left is to run it. First, we need to start the simulation in Veranda because messages are not published or received while the simulation is stopped. Once the simulation is running, we can run our script to send a command to the wheels to start driving. This is a three-command step, because we need to set up the ROS environment first.

```
> source /opt/ros/ardent/setup.bash
> source ~/veranda/local_setup.bash
> export RMW_IMPLEMENTATION=rmw_opensplice_cpp
> python3 circle.py
```

If all has gone well, the robot in your simulation will now be driving in a circle! Your code will be in an infinite loop waiting to send and receive messages, you can stop it with `Ctrl-C`

Tip: You don't need to do the two `source [path]` commands and the `export RMW_IMPLEMENTATION` every time you run your code, just the first time. After you have sourced the environment for a specific terminal, those environment variables will stay set up!

Important: Your robot might look a little goofy driving this circle. That's because of the way the simulation handles relative mass; the body of the robot is much larger than the wheels, so the wheels have a difficult time moving it. Both wheels have a *density* property that you can use to give them more oomph; I've found that setting the density of the wheels in this demo robot to 5 works well. When you are building your own robot, this is something you will have to adjust so that it drives correctly.

Tip: Want to reset the simulation? Instead of removing the robot and putting it in again, you can use the quicksave before starting the simulation and quickload to reset to the saved version.

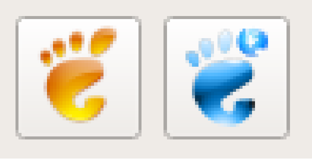


Fig. 22.17: Quicksave (left) and Quickload (right)

22.3.4 Part 3: Drive a more complex path

Driving in a circle is easy, but what if we want to make the robot drive along some path that requires changing wheel speeds? Lets make it drive a wiggle; first driving one wheel, then the other.

Once we call `rclpy.spin()`, our program goes into a loop, so how do we send more commands? We use Timers with callbacks. A Timer in ROS can be created to call a specific function every X seconds.

This is done with the function `node.create_timer(seconds, callback)`. The call returns a Timer Handle, which can be used later to cancel the timer with `node.destroy_timer(handle)`.

So, let's set up some functions to drive a wiggle, they will both work the same way, but one will drive the left wheel, and the other will drive the right.

After we have created our `publeft` and `pubright` publishers, we'll define our function

```
def wiggle_left():
    msg = Float32()

    msg.data = 5.0
    publeft.publish(msg)

    msg.data = 0.0
    pubright.publish(msg)
```

This will stop the right wheel, and start the left wheel. Once we do that, we need to start a timer. When the timer ends, we should call `wiggle_right` to stop the left wheel and start the right one.

```
def wiggle_left():
    msg = Float32()

    msg.data = 5.0
    publeft.publish(msg)

    msg.data = 0.0
    pubright.publish(msg)

    node.create_timer(1, wiggle_right)
```

This will have a 1 second gap between commands. But wait! Timers in ROS go for forever, so if we do this, we'll end up with a bunch of timers starting and stopping the wheels, so we need to save the timer handle, and be able to destroy the timer after it goes off.

```
def wiggle_left():
    global timer_handle
    node.destroy_timer(timer_handle)

    msg = Float32()

    msg.data = 5.0
    publeft.publish(msg)

    msg.data = 0.0
    pubright.publish(msg)

    timer_handle = node.create_timer(1, wiggle_right)
```

If we do the same thing in the `wiggle_right` function, then they can share the timer handle and pass it between themselves. Finally, we need to start the first timer before we spin the node.

```
timer_handle = node.create_timer(0.1, wiggle_left)
rclpy.spin(node)
```

And there we have it! Now `wiggle.py` will drive the wheels alternately. Go ahead and run it to see what it looks like.

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import Float32

rclpy.init()
node = Node("wiggle")

publeft = node.create_publisher(Float32, 'robot0/left_wheel')
pubright = node.create_publisher(Float32, 'robot0/right_wheel')

def wiggle_left():
    global timer_handle
    node.destroy_timer(timer_handle)

    msg = Float32()

    msg.data = 5.0
    publeft.publish(msg)

    msg.data = 0.0
    pubright.publish(msg)

    timer_handle = node.create_timer(1, wiggle_right)

```

(continues on next page)

(continued from previous page)

```
def wiggle_right():
    global timer_handle
    node.destroy_timer(timer_handle)

    msg = Float32()

    msg.data = 0.0
    publeft.publish(msg)

    msg.data = 5.0
    pubright.publish(msg)

    timer_handle = node.create_timer(1, wiggle_left)

timer_handle = node.create_timer(0.1, wiggle_left)
rclpy.spin(node)

node.destroy_node()
rclpy.shutdown()
```

22.3.5 Part 4: Hooking into the Simulation Clock

Now that you have a couple of scripts running, let's take a look at what happens when we use the time-warp capabilities of Veranda. Click the time-warp button while your `wiggle.py` is driving a robot.

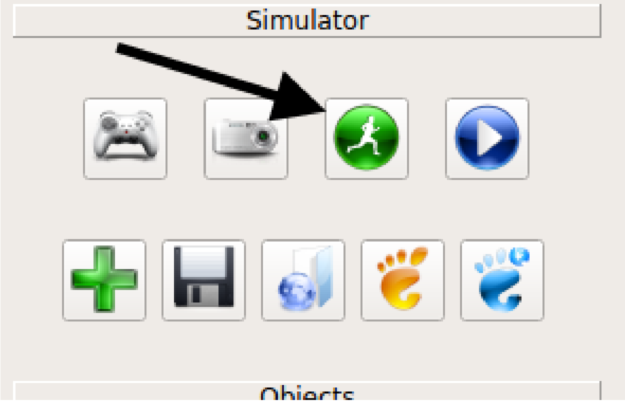


Fig. 22.18: The time warp button; press it multiple times to cycle through 2x, 3x, and 0.5x speeds

That probably didn't do what you expected, did it? The issue here is that, in the simulation, time started moving faster, but the clock in your control script didn't! So for every 1 second of wiggling that the control code thought it was doing, the simulator was actually driving the robot for more than 1 second.

This can be accounted for by using the Veranda SimTimer. The SimTimer listens to the clock message coming from Veranda, and uses those to determine how much time has passed, instead of the sytem clock.

First, we need to include the SimTimer module

```
from veranda.SimTimer import SimTimer
```

Next, after we create our ROS node, we create a timer object which uses that node.

```
simTime = SimTimer(True, "veranda/timestamp", node)
```

Note:

The parameters for the SimTimer are

- Boolean - Should it use the Simulation Timer? If False, the regular system clock is used
- String - ROS Topic that the timestamp is published to. This is currently always the same
- Node - The ROS Node that should be used to listen for time messages

Now, everywhere that we have `node.create_timer` and `node.destroy_timer`, we can replace with `simTime.create_timer` and `simTime.destroy_timer`. It's that easy! Go ahead and run your new wiggle code, and test out how it works with the time-warp feature.

Important: While the create and destroy functions behave similarly, the SimTimer does not return the same dataType as the ROS Node. If the SimTimer is using the timestamp message, it will return integer values as the timer handles, but if it is using the regular ROS timer functionality, (Param 1 is False), it will return the Timer type that `Node.create_timer()` yields.

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import Float32

from veranda.SimTimer import SimTimer

rclpy.init()
node = Node("wiggle")

simTime = SimTimer(True, "veranda/timestamp", node)

publeft = node.create_publisher(Float32, 'robot0/left_wheel')
pubright = node.create_publisher(Float32, 'robot0/right_wheel')

def wiggle_left():
    global timer_handle
    simTime.destroy_timer(timer_handle)

    msg = Float32()

    msg.data = 5.0
    publeft.publish(msg)
```

(continues on next page)

(continued from previous page)

```
    msg.data = 0.0
    pubright.publish(msg)

    timer_handle = simTime.create_timer(1, wiggle_right)

def wiggle_right():
    global timer_handle
    simTime.destroy_timer(timer_handle)

    msg = Float32()

    msg.data = 0.0
    publeft.publish(msg)

    msg.data = 5.0
    pubright.publish(msg)

    timer_handle = simTime.create_timer(1, wiggle_left)

timer_handle = simTime.create_timer(0.1, wiggle_left)
rclpy.spin(node)

node.destroy_node()
rclpy.shutdown()
```

22.4 Simulation of a Differential Drive Robot

Veranda is the simulation application we will use to introduce basic concepts in robotics simulation. The ROS community has used Stage and Gazebo. Stage is no longer supported and one must use either STDR or Veranda. a ROS based two dimensional physics simulator. Gazebo will be discusssed later in this text. Veranda uses the physics engine, Box2D, to determine both motion and interactions (collisions). Essentially this is a 2D game engine and the game players are robots.

Veranda is very general in scope. Robots are a collection of masses which are connected by joints and are subject to forces. Forces are contolled by the user through external programs. All of the communication between the collection of programs is done using ROS messages.

To install Veranda, goto to Roboscience.org. Under software, click on *read more* and you will see the link for Veranda. Follow the link and then follow the instructions on the page. You will download the installer and it will download the application for you. It will then setup the paths and the environment variables.

To load a prebuilt robot, click on the folder symbol in the panel under simulator tools and select one of the Differential Drive robots in the Veranda/Robots subdirectory. Click on the plus symbol under simulator tools to place this in the simulation world. You can zoom in and out using the “q” and “e” keys. You can start the simulation by clicking on the run icon in the simulation panel.

In the Veranda/Scripts directory, you will find some example programs to drive the robot. The first step is to source the setup file:

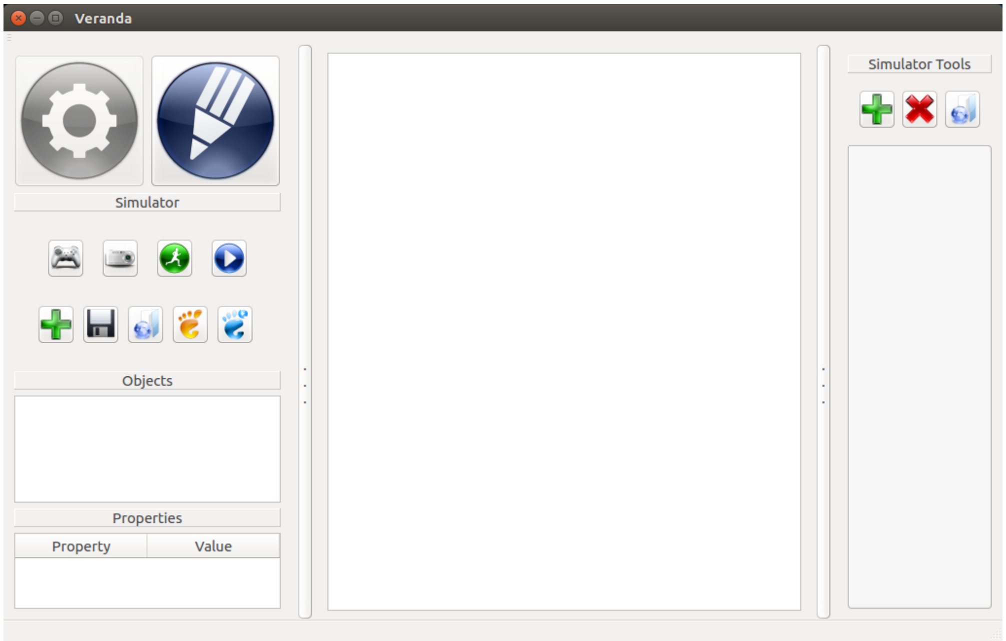
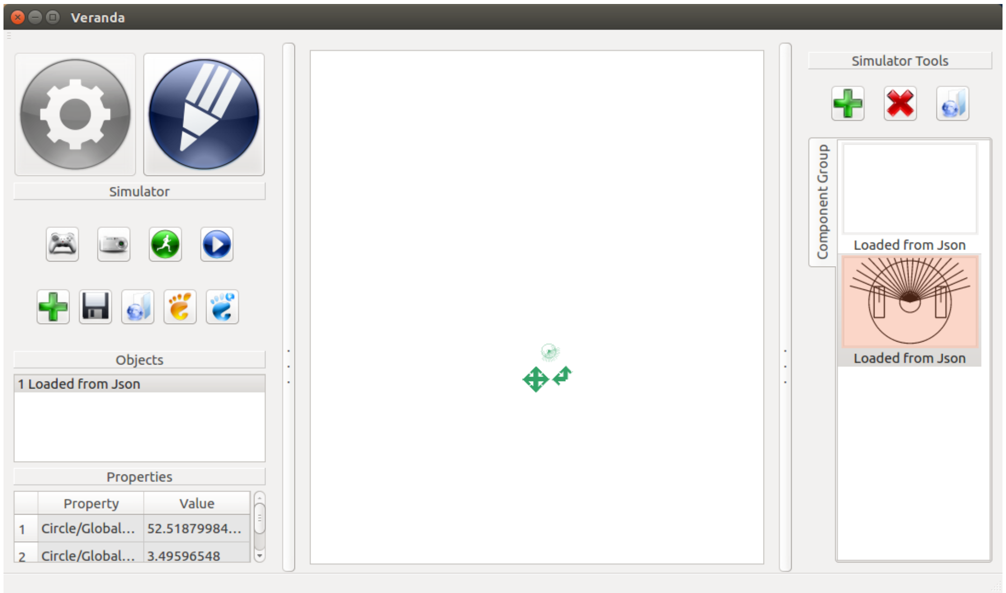


Fig. 22.19: Veranda at launch



1 Loaded from Json

	Property	Value
1	Circle/Global...	52.51879984...
2	Circle/Global...	3.49596548

Fig. 22.20: Differential Robot loaded.

```
source /opt/ros/ardent/setup.bash
cd <veranda directory>
source local_setup.bash
python3 Scripts/fig8_differential.py
```

This should drive the robot in a figure 8 shaped path. You will see other examples in the directory. First we will run the commands in the interpreter *by hand*.

```
import rclpy
from rclpy.node import Node
from veranda.SimTimer import SimTimer
from std_msgs.msg import Float32
import math

rclpy.init()
node = Node("talker")
publeft = node.create_publisher(Float32, 'robot0/left_wheel')
pubright = node.create_publisher(Float32, 'robot0/right_wheel')

msg = Float32()
msg.data = 5.0
publeft.publish(msg)
pubright.publish(msg)
```

This will move the robot. Note that the behavior of the simulator is to keep the motors running on the last received wheel commands. So in the code above, the robot will continue to drive. You will need to set msg.data to zero to stop the bot. This is one common way that motor control systems will operate. This is a design decision which will affect the way a vehicle operates when communications are disrupted. Having a motor control system that automatically slows the vehicle down after some time interval if no communciations have been received is another way to design the system.

Joystick (fill in details)

To drive a predetermined path, a precise sequence of commands must be sent. Relative to the motion and timescale of a robot (real or correctly simulated) the Python commands arrive very quicky. Delays need to be added to insure that certain movements have time to complete. One can code the logic directly as:

```
msgl.data = left_speed ; msgr.data = right_speed
publeft.publish(msgl); pubright.publish(msgr)
sleep(val)

msgl.data = left_speed ; msgr.data = right_speed
publeft.publish(msgl); pubright.publish(msgr)
sleep(val)

...

msgl.data = left_speed ; msgr.data = right_speed
publeft.publish(msgl); pubright.publish(msgr)
sleep(val)
```

The left, right wheel speeds and time delay values can be made into arrays. Then the preceeding commands can be called in a simple loop:

```
for i in range(n):
   msg.data = left_speed[i]
   publeft.publish(msg[i])
   msg.data = right_speed[i]
   pubright.publish(msg[i])
   sleep(val[i])
```

To use the code above to drive a simple shape like a square that is composed of line segments, you need to practice with the timing to get the distances and angles set correctly. To travel straight, you set the two wheel speeds equal. One can figure the delay time out from the differential drive formulas, but for here we will assume it is done experimentally. The harder part is the turns. To turn in place (differential drive robots can pivot in place), you will set the wheel speeds with opposite sign (same magnitude).

In a real robot, the variations in the hardware and environment will cause the robot to drift over time. This accumulates and at some point the error can get large enough to render the system useless. This will be addressed when we discuss using sensor feedback to correct motion.

We will end the section with the figure 8 example code. This example contains many of the ideas discussed in the text so far.

```
import rclpy
from rclpy.node import Node
from veranda.SimTimer import SimTimer
from std_msgs.msg import Float32
import math


# Robot parameters
R = 0.75
L = 1.5

# Location Functions to form a figure 8 of the necessary size
def x_t(t):
    return 12.0*math.sin(t)

def y_t(t):
    return 6.0*math.sin(2*t)

# Differential drive inverse kinematics
def DD_IK(x_t, y_t, t):
    # Calculate xdot and xdotdot at current time
    x_dot_t = 12*math.cos(t)
    x_dotdot_t = -12*math.sin(t)

    # Calculate ydot and ydot dot at current time
    y_dot_t = 12*math.cos(2*t)
    y_dotdot_t = -24*math.sin(2*t)

    #Calculate phi1, phi2
    v = math.sqrt(x_dot_t * x_dot_t + y_dot_t * y_dot_t)
    k = (x_dot_t * y_dotdot_t - y_dot_t * x_dotdot_t)/(v*v*v)
```

(continues on next page)

(continued from previous page)

```
    phi1 = v/R*(k*L+1)
    phi2 = v/R*(-k*L + 1)

    return (phi1, phi2)


# Publishes a set of wheel velocities
# in the format required by the STDR
def publishWheelVelocity(publeft, pubright, phi1, phi2):
    msg = Float32()
    msg.data = phi1
    publeft.publish(msg)
    msg.data = phi2
    pubright.publish(msg)


def main():
    rclpy.init()
    node = Node("talker")
    publeft = node.create_publisher(Float32, 'robot0/left_wheel')
    pubright = node.create_publisher(Float32, 'robot0/right_wheel')
    simTime = SimTimer(True, "veranda/timestamp", node)

    # Factor to scale down speed by
    speedScale = 1

    # Tick time at 10 hz
    dt = 0.1

    def cb():
        # Calculate wheel velocities for current time
        phi1, phi2 = DD_IK(x_t, y_t, simTime.global_time() + 2*math.pi)
        print(phi1, phi2)
        # Publish velocities
        publishWheelVelocity(publeft, pubright, phi1, phi2)

    simTime.create_timer(dt, cb)
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

22.5 Running the Demos

The Veranda project comes with a couple of demo robots and control scripts pre-made. The robots are specifically configured to work with the control scripts; you should be able to load the demo robots right into the simulation, start the script, press play on the simulation, and watch them go. Robots and Scripts can be found in the `Demo` folder of the Veranda repository.

22.5.1 The Demo Robots

There are three robots included with the project:

- Differential-w-GPS

  This is the classic turtle robot. It is a circular body with two wheels, which can be controlled as a differential drive system. The robot has a GPS attached to it, and will publish its absolute location in the simulation.

  Input Topics

Topic	Datatype	Purpose
robot0/left_wheel	Float32	Sets the velocity of the left wheel in radians per second
robot0/right_wheel	Float32	Sets the velocity of the right wheel in radians per second

  Output Topics

Topic	Datatype	Purpose
robot0/pose2d	Pose2D	Reports the current X, Y, Theta positioning of the robot

- Differential-w-Lidar-Touch

  This robot is the classic turtle robot with a twist; attached to the robot are a 360-degree Lidar sensor and a Bump sensor which will detect contact with any part of the robot body.

  Input Topics

Topic	Datatype	Purpose
robot0/left_wheel	Float32	Sets the velocity of the left wheel in radians per second
robot0/right_wheel	Float32	Sets the velocity of the right wheel in radians per second

  Output Topics

Topic	Datatype	Purpose
robot0/laser	LaserScan	Reports what is seen by the robot's Lidar sensor
robot0/touches	ByteMultiArray	Reports the state of all the buttons spaced around the robot

- Ackermann-w-Lidar

This robot is like a car! It has a rectangular base with 4 wheels; the back wheels are fixed and produce thrust, the front wheels cannot produce thrust, but are used to steer. The front wheels turn following the Ackermann constraint. This robot also has a lidar on it, but it only covers a 90-degree area in front of the robot.

Input Topics

Topic	Datatype	Purpose
robot1/left_wheel	Float32	Sets the velocity of the left rear wheel in radians per second
robot1/right_wheel	Float32	Sets the velocity of the right rear wheel in radians per second
robot1/steer	Float32	Sets the angle in radians for the robot to steer towards

Output Topics

Topic	Datatype	Purpose
robot1/laser	LaserScan	Reports what is seen by the robot's Lidar sensor

22.5.2 The Demo Scripts

The Demo/Scripts folder contains 6 different python scripts which can be run with the default robots. All of them can be run with the command `python3 [scriptname]`, but some of them require the command line arguments described below. Remember to source the setup file for ROS2 and the Veranda workspace before running these scripts!

- `fig8_differential.py`

  Usage: `python3 fig8_differential.py`

  Publishes left and right wheel velocities which will drive a robot in a figure-8 shape. It uses the topics `robot0/left_wheel` and `robot0/right_wheel`. As you might expect, it can be used to control either of the demo differential drive robots. If multiple of them are in the simulation, they will all be controlled.

- `joystick_differential.py`

  Usage: `python3 joystick_differential.py {input-topic} [output-topic]`

  Listens for messages from a 2-axis joystick and publishes left/right wheel commands on the topics `[output-topic]/left_wheel` and `[output-topic]/right_wheel` to drive a differential robot. The topic listened to for the joystick is `[input-topic]/joystick`. If no output topic is given, it will be the same as the input topic. (For example, if the `[input-topic]` were 'banana', then the topics used would `banana/joystick`, `banana/left_wheel`, and `banana/right_wheel`.

- `joystick_ackermann.py`

Usage: python3 joystick_ackermann.py {input-topic} [output-topic]

Similarly to joystick_differential.py, this script listens for messages from a 2-axis joystick; but it uses them to produce messages on three topics, in order to drive the Ackermann-w-lidar demo robot. The rules for determining the topic names are the same as for the differential joystick script; but there is an extra topic: [output-topic]/steer to control the steering wheels.

- joystick_listener.py

  Usage: python3 joystick_listener.py {topic}

  Listens to the specified topic for joystick messages and writes the joystick information to stdout. It is similar to using the ros2 topic echo command, and is a good example of how to listen for joystick messages in a script.

- lidar_listener.py

  Usage: python3 lidar_listener.py {topic}

  Listens to the specified topic for LaserScan messages and writes the lidar information to stdout. It is similar to using the ros2 topic echo command, and is a good example of how to listen for lidar messages in a script.

- linux_joy_reader.py

  Usage: python3 linux_joy_reader.py [device]

  This script is only usable on Linux systems! It uses the plug-n-play joystick drivers available in the OS to listen for input from hardware joysticks. It has only been tested with Playstation controllers plugged into a USB port.

  If the script is run with no arguments, it will look through the available devices and print a list of all of the ones which are joysticks (they start with 'js'). When the script is run with one of these devices as its argument, it will listen to the input from that device and publish joystick messages to the topic [device]/joystick.

22.5.3 Enough Talking, Lets Do Some Demos!

Here we've outlined a number of demos that you can run right after installing Veranda. Each one will require that you have multiple command line terminals open, and will number them 1-n. The first time you encounter a terminal number, you should cd into the Veranda workspace and source the setup file for both ROS2 and the Veranda workspace immediately before continuing the demo. All commands are given in terms of Linux, so you may need to make adjustments!

Demo 1: Driving GPS Turtle in a Figure-8

- Terminal 1: ros2 run veranda veranda
- Load the Differential-w-GPS robot into the toolbox
- Add that robot to the simulation

- Start the simulation
- Terminal 2: `python3 ./src/veranda/veranda/Demo/Scripts/fig8_differential.py`
- Bask in the glory of your achievement

Demo 2: Driving Lidar Turtle with the Virtual Joystick

- Terminal 1: `ros2 run veranda veranda`
- Load the Differential-w-Lidar-Touch robot into the toolbox
- Add that robot to the simulation
- Start the simulation
- Create a Virtual Joystick
- Set the joystick topic to `demo/joystick`
- Terminal 2: `python3 ./src/veranda/veranda/Demo/Scripts/joystick_differential.py demo robot0`
- Use the virtual joystick to drive the robot
- Take over the world

Demo 3: Driving Ackermann Bot with the Virtual Joystick

- Terminal 1: `ros2 run veranda veranda`
- Load the Ackermann-w-Lidar robot into the toolbox
- Add that robot to the simulation
- Start the simulation
- Create a Virtual Joystick
- Set the joystick topic to `demo/joystick`
- Terminal 2: `python3 ./src/veranda/veranda/Demo/Scripts/joystick_ackermann.py demo robot1`
- Use the virtual joystick to drive the robot
- Steal the moon

Bonus Demo! Driving Ackermann Bot with a Real, Live Joystick

- Acquire a USB joystick controller
- Plug that joystick into your Linux machine

- Terminal 1: `python3 ./src/veranda/veranda/Demo/Scripts/linux_joy_reader.py`
- Terminal 1: `python3 ./src/veranda/veranda/Demo/Scripts/linux_joy_reader.py [insert device here]`
- Try devices until one works, and the terminal prints stuff when you move the joystick
- Terminal 2: `python3 ./src/veranda/veranda/Demo/Scripts/joystick_ackermann.py [device] robot1`
- Terminal 3: `ros2 run veranda veranda`
- Load the Ackermann-w-Lidar robot into the toolbox
- Add that robot to the simulation
- Start the simulation
- Use the joystick to drive the robot
- Step 3: Profit

Before we do more complicated motion planning, it is important to get a feel for how the simulations are done and to do a few computations directly. This helps the roboticist understand the errors and limitations of the simulations.

22.6 Tutorial: Placing Obstacles and Sensing in Veranda

Assuming you've finished *Tutorial 1*, you have a little robot that can wander around aimlessly in an empty world. But one of the most important parts of robotics is being able to take input from sensors on your robot and react to your environment. This tutorial will show you how to put some obstacles in the world with your robot and some ways the sensors provided with Veranda can be used.

22.6.1 Part 1: Making obstacles

Having a robot that can drive around is fun, but eventually, you may want to try to write code to make the robot avoid things it might run into. The first step to doing that is having things for the robot to hit. Veranda can load image files as a simulation, and turn them into obstacles that robots can hit. To do this, choose the 'load simulation' button, and find your image file. Make sure that all the obstacles you want are in the image, because loading an image will clear your simulation. This example image has a square that we can keep the robot in, and some shapes in the middle for it to avoid.

Tip: Loading images in Veranda works best if they contain only black and white pixels, with no other colors (including grey). If you do try to load other images, you can play with the black/white threshold to get it to turn out better.

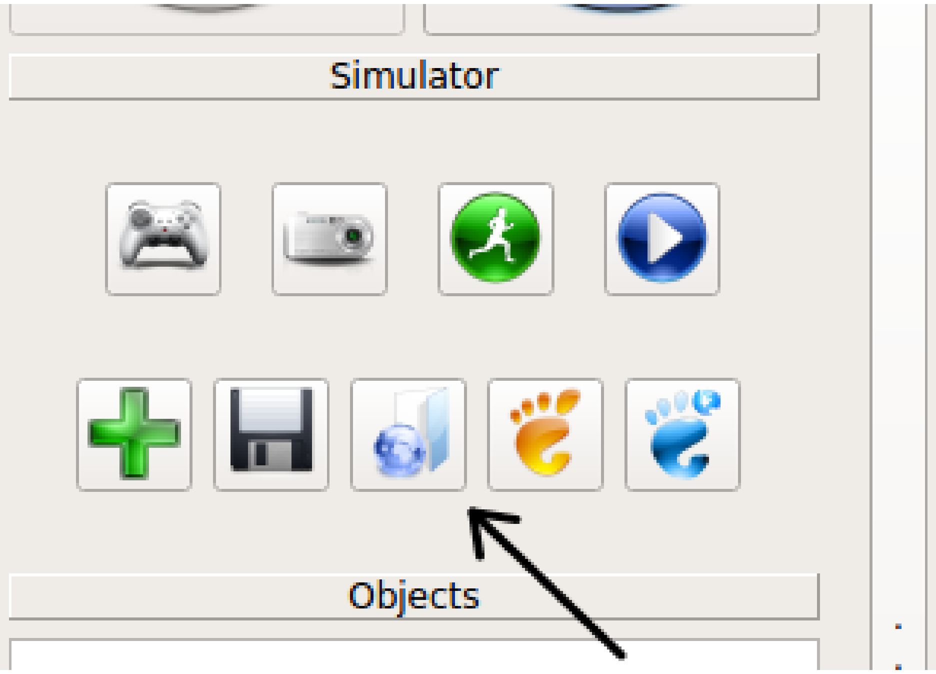


Fig. 22.21: Use the 'Load Simulation' button to to load images as obstacles

Important: Veranda can load a number of different files as full simulations, make sure you pick the correct file type in the file-choose dialog so that you are able to select the file you want.

Once you choose an image, you will be presented with some import options. The most important will be the size options, followed by the threshold options. Veranda will report the size of the image, in pixels, and you will have the option to set the pixel/m ratio, or and the image size (in meters). Our little roomba has a radius of 2m, and the circle obstacle in our image is 60 px in diameter, so if we set 30 px/m, the robot will be the same size as that circle. Let's make it 10 px/m so the robot is smaller than the circle.

22.6.2 Part 2: Did it crash?

Now that there's something to hit, we want to know when the robot hits it. To do this, we'll add a touch sensor to the robot; it will send messages to the control script whenever it touches something.

In the editor, add a Touch Ring to your turtle bot. If you kept your robot at the default size, you will not be able to see any difference, because the touch ring is also a circle, and it defaults to 1m radius.

The touch ring represents a ring of bump sensors evenly spaced around the robot; by default, the 'angle_start' and 'angle_end' properties, which specify which part of the robot has the sensors, encompass the entire chassis. Let's make there be 20 buttons them by setting the property 'sensor_count' to 20. Don't forget to set the ROS topic property 'channels/output_touches' to 'robot0/touches'.

Tip: Don't have your robot loaded in the editor anymore? You can load it into the editor from file!

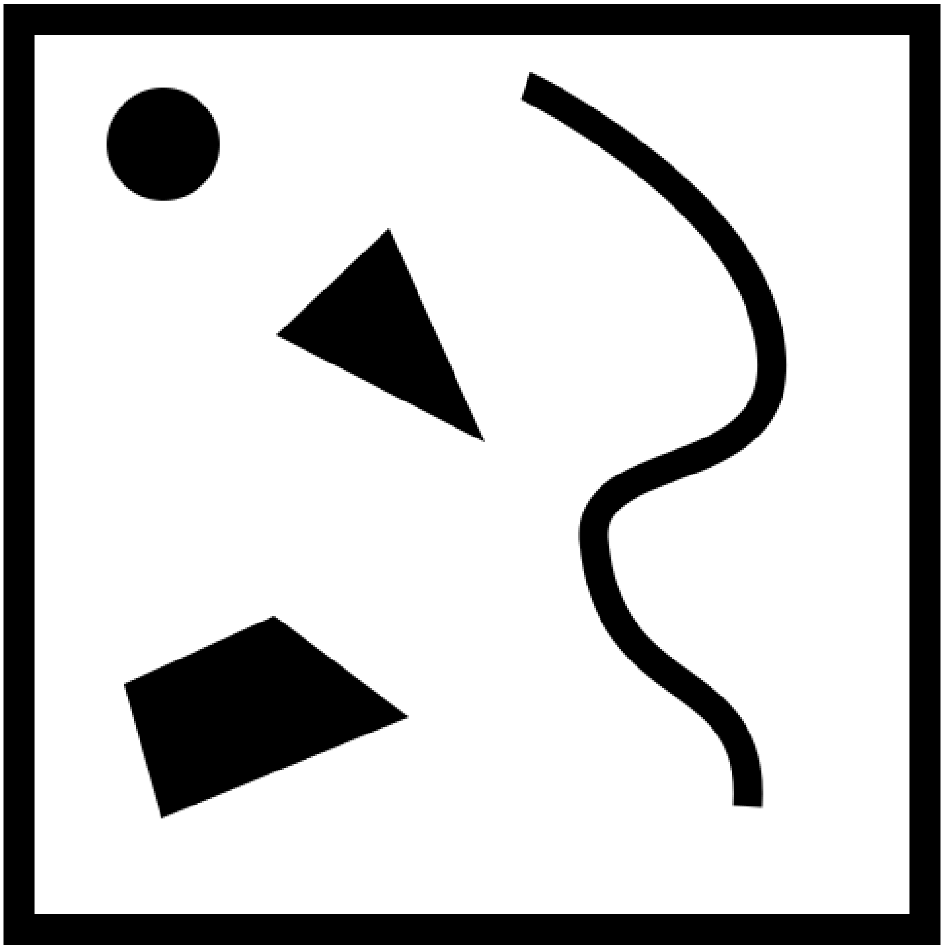
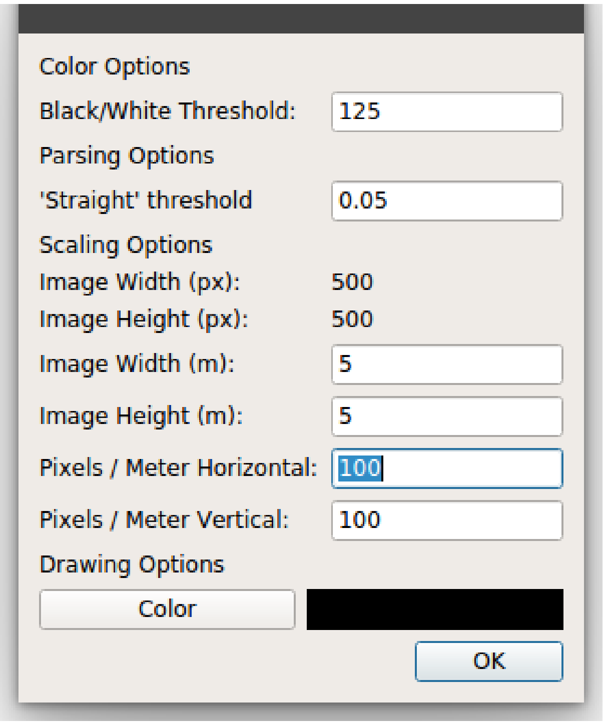


Fig. 22.22: Example of the kind of image you might load. Make sure to get all your obstacles in one picture!



Color Options

Black/White Threshold: 125

Parsing Options

'Straight' threshold 0.05

Scaling Options

Image Width (px): 500

Image Height (px): 500

Image Width (m): 5

Image Height (m): 5

Pixels / Meter Horizontal: 100

Pixels / Meter Vertical: 100

Drawing Options

Color ■

Fig. 22.23: The image importing options

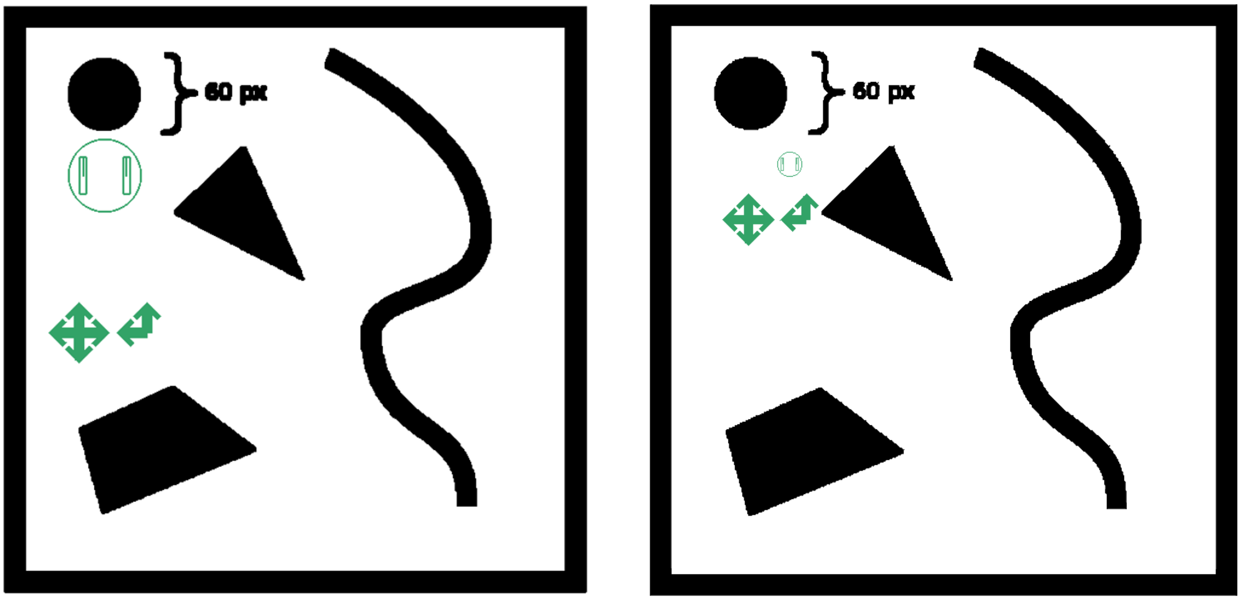


Fig. 22.24: Scaling the image to 30x30 px/m (left) and 10x10 px/m (right)

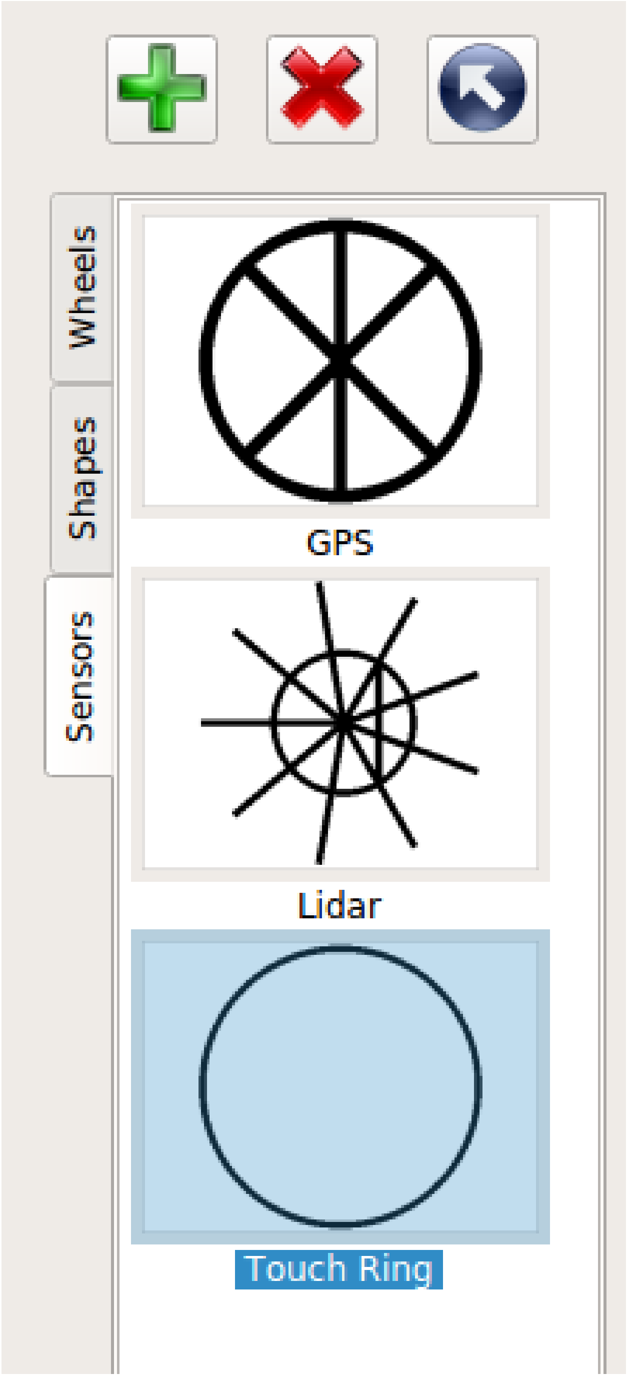
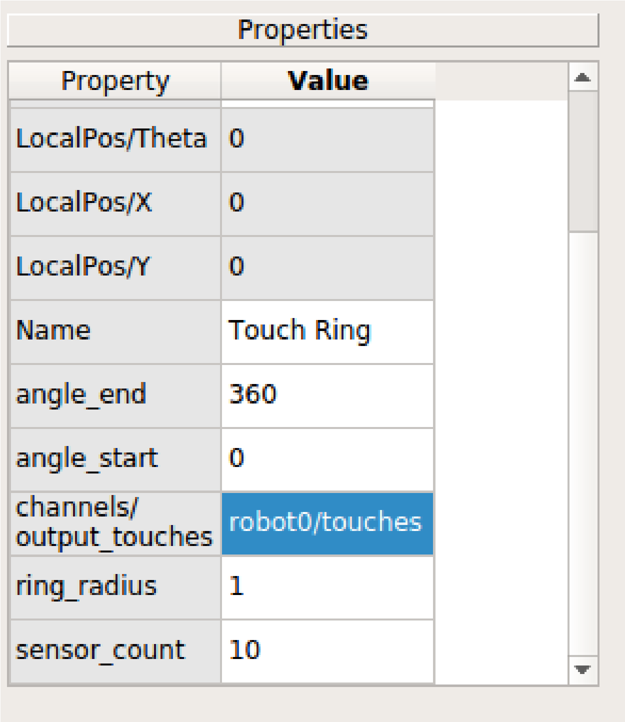


Fig. 22.25: Touch Ring is found under the sensors tab of the editor toolbox



Properties

Property	Value
LocalPos/Theta	0
LocalPos/X	0
LocalPos/Y	0
Name	Touch Ring
angle_end	360
angle_start	0
channels/ output_touches	robot0/touches
ring_radius	1
sensor_count	10

Fig. 22.26: The properties we want to set for the touch ring

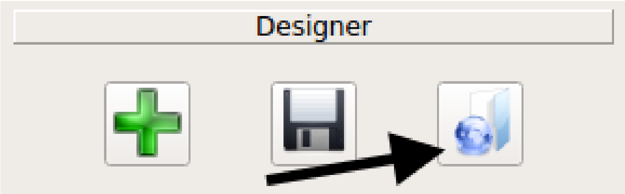


Fig. 22.27: The load button in the editor

Now, when your robot runs into a wall, you'll see a little circle appear on the simulation representing the location of the touch sensor that was triggered.

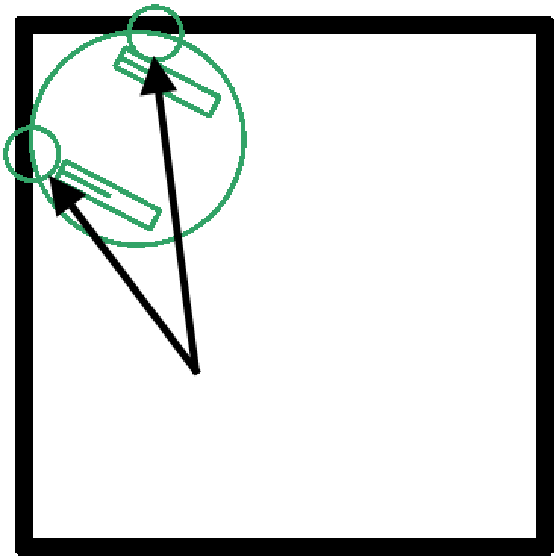


Fig. 22.28: The indicators that your touch ring is sensing something

The last step is to set up a callback in your script to respond to this stimulus. Let's modify `circle.py` for this one.

First, we have to import the message type that the touch ring publishes: ByteMultiArray

```
from std_msgs.msg import ByteMultiArray
```

Next, we create our callback function to handle this data. ROS callbacks always have 1 parameter by default, and that is the message that was sent. In the ROS std_msg messages, each message has a `.data` element which contains the actual information sent. Let's make a callback that outputs the indexes of the buttons that were touched. Because of how ROS handles the ByteMultiArray type in python, we have to use the `struct::unpack()` function to get the data as a char type.

```
from struct import *
def get_hit(message):
    hits = message.data

    for i in range(len(hits)):
        hit = unpack('b', hits[i])[0]

        if hit != 0:
            print("Touched on", i)
    print("----------------")
```

Lastly, we set up a subscriber on the node which will listen to the `robot0/touches` topic for ByteMultiArray messages and call the callback function whenever a message comes in.

```
subtouches = node.create_subscription(ByteMultiArray, 'robot0/touches', get_
↪hit)
```

Now, if you load your little robot into that box and run this code, it will hit the wall, and you'll see something like the following output

```
Touched on 1
----------------
Touched on 0
----------------
Touched on 0
Touched on 3
----------------
```

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import Float32

from std_msgs.msg import ByteMultiArray
from struct import *

def get_hit(message):
    hits = message.data

    for i in range(len(hits)):
        hit = unpack('b', hits[i])[0]

        if hit != 0:
            print("Touched on", i)
    print("----------------")

rclpy.init()
node = Node("circle")

publeft = node.create_publisher(Float32, 'robot0/left_wheel')
pubright = node.create_publisher(Float32, 'robot0/right_wheel')

subtouches = node.create_subscription(ByteMultiArray, 'robot0/touches', get_
↪hit)

msg = Float32()

msg.data = 5.0
publeft.publish(msg)

msg.data = 10.0
pubright.publish(msg)

rclpy.spin(node)

node.destroy_node()
```

(continues on next page)

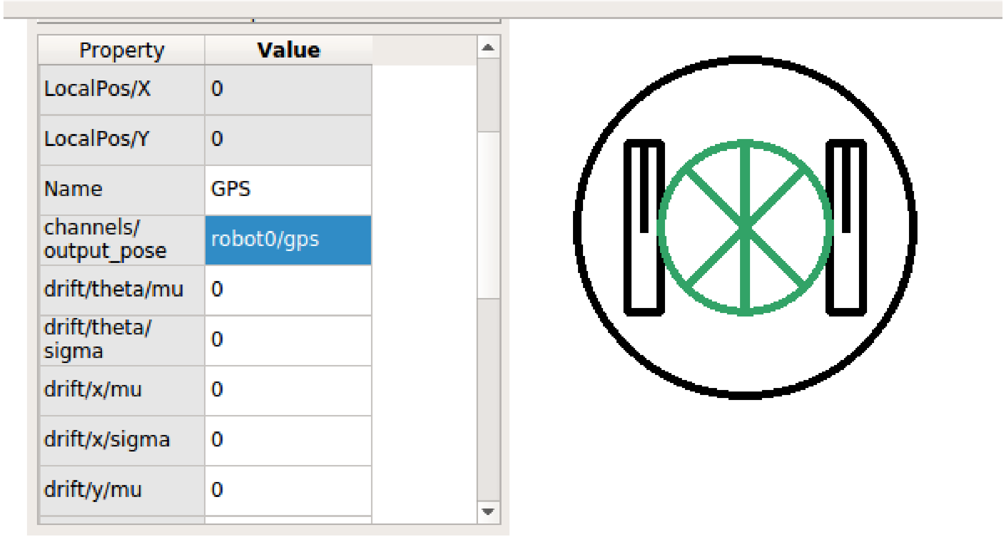
(continued from previous page)

```
rclpy.shutdown()
```

22.6.3 Part 3: Where is it?

One of the most valuable pieces of information you can get is the location of your robot. If you don't have a GPS, or some other positioning system available, your robot will have to estimate it's location based on what it sees. Fortunately, Veranda comes equipped with a GPS sensor that you can use to get the absolute location of your robot.

For this example, I added a GPS to my turtle robot, and set its output channel to be robot0/gps.



Property	Value
LocalPos/X	0
LocalPos/Y	0
Name	GPS
channels/ output_pose	robot0/gps
drift/theta/mu	0
drift/theta/ sigma	0
drift/x/mu	0
drift/x/sigma	0
drift/y/mu	0

Fig. 22.29: Turtle bot upgraded with a gps.

Now all we need to do to start listening to the robot locations is subscribe to that topic and write a function to handle the ROS Pose2D message

Note: This link goes to the original ROS documentation; that's ok, a lot of the built-in messages are the same as they were in ROS 1, just placed under a different header directory

The Pose2D message contains 3 pieces of information: x, y, and theta - the robot's location in the world and direction. Let's observe the turtle's location as it drives in a circle.

First, we need to change our import statement to get the Pose2D message, then we need to change our subscription to use that message.

```
from geometry_msgs.msg import Pose2D
...
gps = node.create_subscription(Pose2D, 'robot0/gps', get_position)
```

We also need to update our callback to handle the message. I set it up to print the angle in degrees. Make sure you modulus the angle to get it into the range you want, because it will just count up or down forever if your robot spins.

```
import math
def get_position(message):
    print("Robot is at (" + str(message.x) + "," + str(message.y) + ") facing
↪" + str((message.theta*180/math.pi) % 360) + " degrees")
    print("----------------")
```

Other than those changes, our code is exactly the same as the code used to print when the robot ran into something. This is what it outputs.

```
Robot is at (-3.6408586502075195,-0.10235483199357986) facing 163.
↪32569095773033 degrees
----------------
Robot is at (-3.8731796741485596,-0.5048621296882629) facing 179.
↪23729964819177 degrees
----------------
Robot is at (-3.9862496852874756,-0.9556393027305603) facing 195.
↪1489083386532 degrees
----------------
Robot is at (-3.971405267715454,-1.4201442003250122) facing 211.
↪06051702911464 degrees
----------------
Robot is at (-3.8297839164733887,-1.8627822399139404) facing 226.
↪97212571957607 degrees
----------------
Robot is at (-3.572237491607666,-2.2496349811553955) facing 242.
↪88373441003932 degrees
----------------
Robot is at (-3.2185018062591553,-2.551058292388916) facing 258.
↪79534310050076 degrees
----------------
Robot is at (-2.795682191848755,-2.743954658508301) facing 274.7069517909622␣
↪degrees
----------------
Robot is at (-2.336179733276367,-2.8135430812835693) facing 290.
↪61856048142363 degrees
```

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import Float32

from geometry_msgs.msg import Pose2D
from struct import *

import math

def get_position(message):
    print("Robot is at (" + str(message.x) + "," + str(message.y) + ") facing
↪" + str((message.theta*180/math.pi) % 360) + " degrees")
```

(continues on next page)

(continued from previous page)

```
    print("----------------")

rclpy.init()
node = Node("circle")

publeft = node.create_publisher(Float32, 'robot0/left_wheel')
pubright = node.create_publisher(Float32, 'robot0/right_wheel')

gps = node.create_subscription(Pose2D, 'robot0/gps', get_position)

msg = Float32()

msg.data = 5.0
publeft.publish(msg)

msg.data = 10.0
pubright.publish(msg)

rclpy.spin(node)

node.destroy_node()
rclpy.shutdown()
```

Note: The GPS seems like a simple sensor, but it has a lot of options. In the gps properties, you can properties for x, y, and theta to specify…

- Drift: How much error can accumulate on each time step
- Noise: How far away from the drifted position can the reported position be
- Probability: What is the probability [0, 1] that the value will not be invalid

For both Drift and Noise, you can specify the Sigma and Mu of the Gaussian distribution used to pick values.

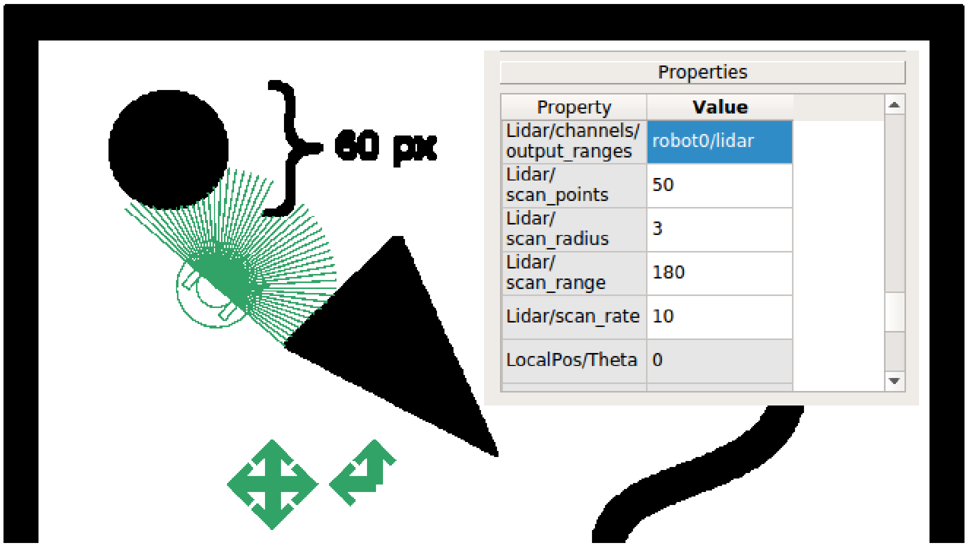
22.6.4 Part 4: What's nearby?

It's great that we can use a bump sensor to know when we hit something, but wouldn't it be great if we could avoid crashing in the first place? The LIDAR sensor allows for just that! It can simulate bouncing rays of light across a range of angles to report how far away things are from your robot. The message that the lidar publishes is the LaserScan message.

Let's upgrade our turtle again, and put it somewhere that the lidar will sense something.

Note: In that image, the lines for the lidar had been updated during simulation. Right after you place the robot, they won't change to reflect what's around them until you press 'play'.

Once again, changing our existing code to use the new message is pretty easy; the hard part is understanding the LaserScan message. This code will make the robot spin slowly in place, and it will print the message



Properties

Property	Value
Lidar/channels/output_ranges	robot0/lidar
Lidar/scan_points	50
Lidar/scan_radius	3
Lidar/scan_range	180
Lidar/scan_rate	10
LocalPos/Theta	0

Fig. 22.30: Turtle bot upgraded with a lidar, sensing some obstacles. I set my lidar to report on the robot0/lidar channel. It is sensing 180 degrees in front of it, with 50 rays that go 3 meters at max.

as-is when it arrives.

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import Float32

from sensor_msgs.msg import LaserScan
from struct import *

import math

def get_position(message):
    print(message)
    print("----------------")

rclpy.init()
node = Node("circle")

publeft = node.create_publisher(Float32, 'robot0/left_wheel')
pubright = node.create_publisher(Float32, 'robot0/right_wheel')

gps = node.create_subscription(LaserScan, 'robot0/lidar', get_position)

msg = Float32()

msg.data = 0.5
publeft.publish(msg)

msg.data = -0.5
pubright.publish(msg)
```

(continues on next page)

(continued from previous page)

```

rclpy.spin(node)

node.destroy_node()
rclpy.shutdown()
```

Let's take a look at one of the LaserScan messages

```
sensor_msgs.msg.LaserScan(
    header=std_msgs.msg.Header(
        stamp=builtin_interfaces.msg.Time(sec=0, nanosec=0),
        frame_id=''),
    angle_min=-1.5707963705062866,
    angle_max=1.5707963705062866,
    angle_increment=0.06411413848400116,
    time_increment=0.0,
    scan_time=0.10000000149011612,
    range_min=2.1359217166900635,
    range_max=2.844834089279175,
    ranges=[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf,
            2.2153570652008057, 2.213566303253174, 2.2417705059051514, 2.
↪280193567276001,
            2.3296966552734375, 2.391442060470581, 2.4669628143310547, 2.
↪5582637786865234,
            2.7111518383026123, 2.844834089279175, inf, inf, inf, inf, inf,␣
↪inf, inf, inf,
            inf, inf, inf, inf, inf, inf, 2.3573288917541504, 2.
↪236565351486206, 2.1359217166900635,
            inf, inf, inf, inf, inf, inf, inf, inf, 2.6468541622161865, 2.
↪440544605255127,
            2.3114850521087646, 2.2470221519470215, 2.1885221004486084],
    intensities=[])
```

There's a lot here to unpack, so let's go one item at a time

- header: Every ROS message has a header stating the message time and the message's id. These are not populated by the Veranda Lidar.
- angle_min/maximum_angle: Bounding range (radians) of the scan, relative to the lidar. Our lidar has a range of 180 degrees, so it goes from -90 to +90, or -pi to +pi.
- angle_increment: Number of radians between each scan point
- time_increment: Time taken between each scan point. Since Veranda is a simulation, we can pause the world and scan it, resulting in instantaneous information
- scan_time: Total time taken to do the scan. This lidar is set to output at 10hz, so that's what it reports.
- range_min/range_max: Minimum and Maximum distance (meters) seen by the lidar.
- ranges: The actual distances seen by the lidar, 1 per scan point. They are reported from minimum angle to maximum. Locations where nothing was seen report infinity.
- intensities: Some lidars (not Veranda's simulation) report the intensity of the light at each point

22.6.5 Part 5: How fast is it going?

The last sensor we're going to discuss here is the encoder. Encoders are devices that can be used to measure the angular velocity of an axle. While real encoders might report frequency of a spinning stripe in front of a sensor, the encoders included in Veranda just report angular velocity. They are attached by default to both the fixed wheel type and Ackermann steering weels type. Just add a wheel to get an encoder. However, until you set the output topic for an encoder, it will do nothing.

Encoders return a single value, the angular velocity of the wheel in radians/second. If we set the output channels for our encoders, and add a little bit of noise, we can see how the noise affects the output while the robot drives in a circle.

```
import rclpy
from rclpy.node import Node

from std_msgs.msg import Float32

from struct import *

import math

left_speed, right_speed = 0, 0

def output():
    print("Wheel speeds: " + str(left_speed) + " - " + str(right_speed))
    print("----------------")

def get_left(message):
    global left_speed

    left_speed = message.data
    output()

def get_right(message):
    global right_speed

    right_speed = message.data
    output()

rclpy.init()
node = Node("circle")

publeft = node.create_publisher(Float32, 'robot0/left_wheel')
pubright = node.create_publisher(Float32, 'robot0/right_wheel')

subleft = node.create_subscription(Float32, 'robot0/left_encoder', get_left)
subright = node.create_subscription(Float32, 'robot0/right_encoder', get_
↪right)

msg = Float32()

msg.data = 5.0
```

(continues on next page)

(continued from previous page)

```
publeft.publish(msg)

msg.data = 10.0
pubright.publish(msg)

rclpy.spin(node)

node.destroy_node()
rclpy.shutdown()
```

```
----------------
Wheel speeds: 2.6641697883605957 - 7.652131080627441
----------------
Wheel speeds: 2.6641697883605957 - 10.325849533081055
----------------
Wheel speeds: 5.337887287139893 - 10.325849533081055
----------------
Wheel speeds: 3.404207706451416 - 10.325849533081055
----------------
Wheel speeds: 3.404207706451416 - 8.392169952392578
----------------
```

22.7 Problems

1. Using ROS and Python, write a chat program (call it *chat.py*). First prompt the user for their name. Write to all members in the chat group that this person has entered the chat. In a loop, grab user inputs and broadcast to the chat with format: name: <user input> . Echo to the terminal all strings sent to the chat.

2. Using ROS and Python, modify the example programs in the text on the kinematics of the two link manipulator.

    1. Write a program that creates a list of 100 equally spaced points along the path $y = 15 - x$ for $0 \leq x \leq 10$ and publishes those points on the topic /physData using a multiarray floating data type, i.e. values x and y are floats. Publish the data at 5Hz.

    2. Write a program that subscribes to topic /physData, plugs the values in, computes the serial two link inverse kinematics to gain the servo angles (pick one of the +/-) and publishes the angles to the topic /thetaData. You may assume the link arms are $a_1 = a_2 = 10$. Format will be the same as the previous topic.

    3. Write a program that subscribes to both /physData and /thetaData. The program should plug the angles into the forward kinematics and check against the data in /physData. It should plot the original curve in green and the "check" in blue.

3. Assume that you have a parallel two link manipulator with $L_0 = 10$cm, $L_1 = 15$cm and $L_2 = 20$cm.

    1. Write a ROS program that creates a list of 100 equally spaced points along the path $x = 7\cos(t) + 10$, $y = 5\sin(t) + 15$ and publishes those points on the topic /physData using a multiarray floating data type, i.e. values x and y are floats. Publish the data at 5Hz.

   2. Write a ROS program that subscribes to topic /physData, plugs the values in, computes the serial two link inverse kinematics to gain the servo angles and publishes the angles to the topic /thetaData. Format will be the same as the previous topic.

   3. Write a ROS program that subscribes to both /physData and /thetaData. The program should plug the angles into the forward kinematics and check against the data in /physData. It should plot the original curve in green and the "check" in blue.

4. Using ROS and python write a program that will add padding to obstacles while shrinking the footprint of the robot to a point. Assume that you have a circular robot with radius 10 and starting pose (15,15,90).

   1. Write a program that will publish the pose of the robot on the topic `/robot/pose` and the footprint type of the robot on `/robot/footprint` as a string (For example circle or polygon). Also publish the radius of the robot on `/robot/radius` as a uint16 message type.

   2. Write a program that will publish a list of obstacles as polygons on the topic `/obstacles`. For this program let the obstacles be the following:

      1. Rectangle with the vertices (40,30), (50,5), (50, 30) (40,30).

      2. Rectangle with the vertices (40,5), (50,5), (50,0), (40,5).

   3. Write a program that subscribes to `/robot/pose`, `/robot/footprint`, and `/obstacles`. Based on the footprint string, this program should be able to subscribe to either the robot radius or dimension topics for circular and rectangular robots. This program will reduce the robot footprint to a point, add padding to the obstacles, and plot the robot as a point and padded obstacles with the maximum x and y values being 70 and 30.

5. Rework the previous problem assuming that you have a rectangular robot with $width = 10$ and $length = 20$ and initial pose (0,10,0).

6. Plot the padding of obstacles using the ros nodes in the previous problem with the initial pose of the robot being (a) (5,10,30), (b) (5,10,70), and (c) (5,25,-90).

7. Write a program that will publish the changing poses of a rectangular robot over time from to in increments of . Assume the inital pose is (5,10, -90) and $width = 10$ and $length = 20$. Use the programs from problem 6 to publish the obstacles and display the padding.

CHAPTER TWENTYTHREE

SIMULATION TOOLS

23.1 Simple Python Tk Two Link Simulator

Note: Check to see if this will work with ROS2. Provide both.

You can download the TwoLink simulator by following the links on `roboscience.org`. To get started, again you need to be in your Ubuntu session and run the ROS Master:

```
>  roscore
```

You can run the Two Link Manipulator simulator we will use by typing

```
>  python twolinksimple.py
```

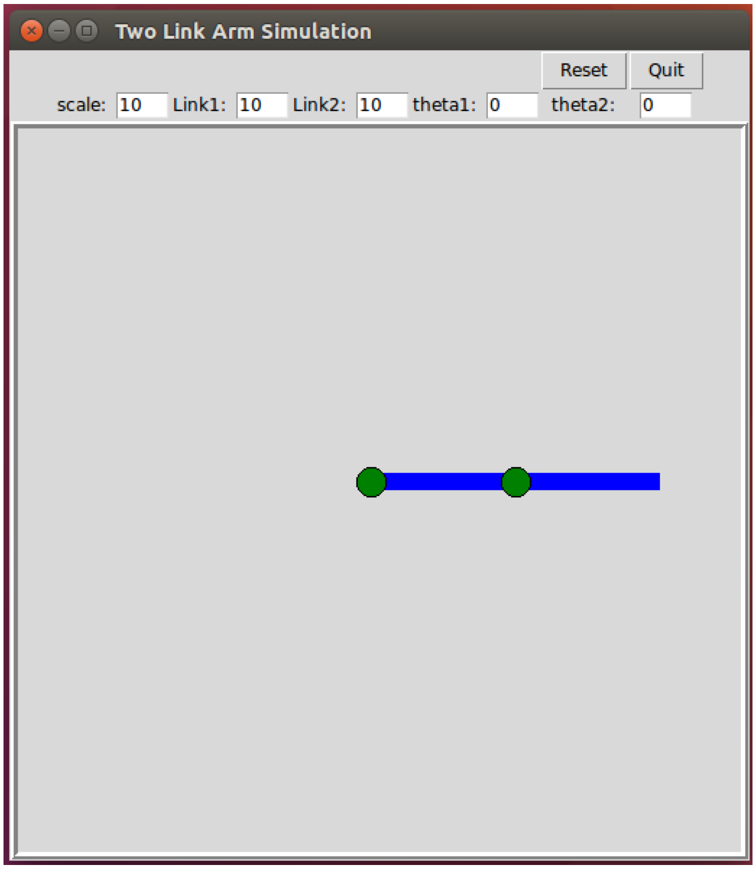
and you should see what is indicated in Fig. 23.1-(a). In another terminal, run Python and type

```
>>> import rospy
>>> from std_msgs.msg import String
>>> pub = rospy.Publisher('TwoLinkTheta', String, queue_size=10)
>>> rospy.init_node('talker', anonymous=True)
>>> message = "20:10:0"
>>> pub.publish(message)
```

You should see the link arm move as shown in Fig. 23.2. The API is very simple. You need to publish a string formatted as “theta1:theta2:pen”. The values theta1 and theta2 are in degrees (int or float), and pen is an int. Pen is set to 1 to draw and 0 to not draw. The program DialCntrl.py is an example of a Tk widget that uses two sliders to set the angle, Fig. 23.4 (a). To gain an understanding of the ROS Node structure, one may list out the ROS nodes (example, your numbers will vary):

```
rosnode list
/DialController_5943_1473004072330
/TwoLinkSimulation_5785_1473004028541
/rosout
```

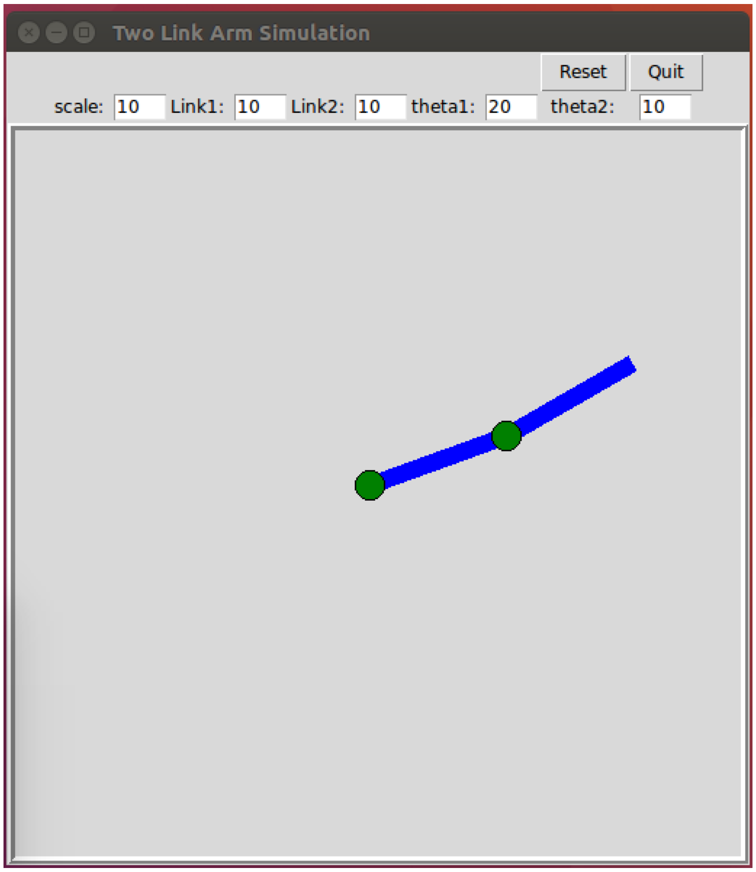
To view the resulting node graph we can use the ROS tool rqt_graph:



Two Link Arm Simulation

scale: 10 Link1: 10 Link2: 10 theta1: 0 theta2: 0

Fig. 23.1: The two link simulator.



scale: 10 Link1: 10 Link2: 10 theta1: 20 theta2: 10

Fig. 23.2: Published angle to the simulator.

```
rosrun rqt_graph rqt_graph
```

In this case it produces Fig. 23.3.

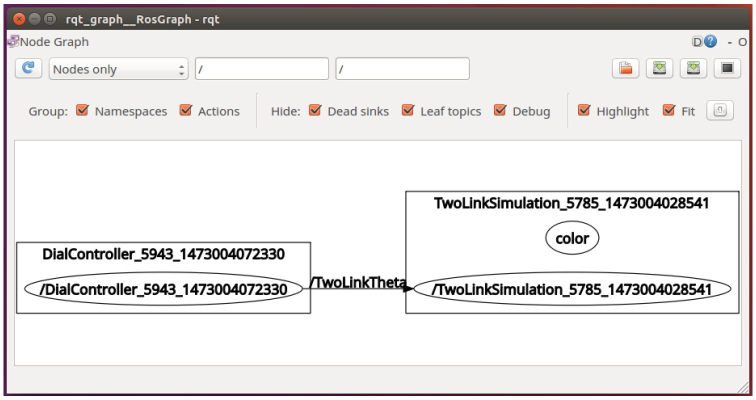

rqt_graph__RosGraph - rqt
Node Graph
Nodes only
/
/
Group:
☑ Namespaces
☑ Actions
Hide:
☑ Dead sinks
☑ Leaf topics
☑ Debug
☑ Highlight
☑ Fit
TwoLinkSimulation_5785_1473004028541
color
DialController_5943_1473004072330
/DialController_5943_1473004072330
/TwoLinkTheta
/TwoLinkSimulation_5785_1473004028541

Fig. 23.3: The ROS Node Graph Tool rqt_graph.

If you are curious about the messages flowing on a topic, recall ROS can echo those to a terminal for debugging purposes. In a free terminal, type

```
rostopic echo /TwoLinkTheta
```

The move one of the sliders. You will see the message on the TwoLinkTheta topic echoed. If you have source code you can clearly print out the messages. It is nice to see what is actually going across. If you don't have source code, then this tool is very handy.

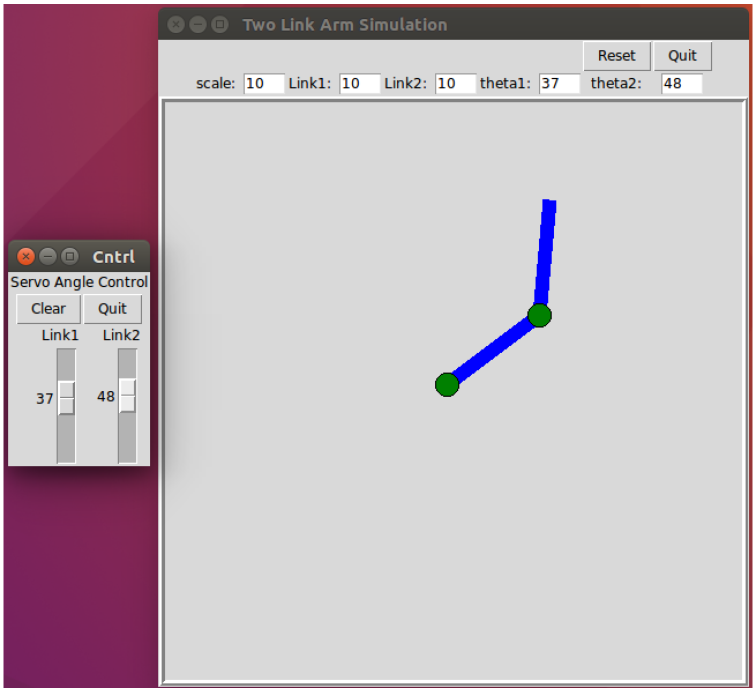
A Tk control that can set position is given in the next example PositionCntrl.py and shown in Fig. 23.4 (b). The widget PositionCntrl.py publishes (x, y) coordinates. An intermediate node IK.py is used to convert the (x, y) values to (θ_1, θ_2) and these values are published to the Two Link Simulator.

```
# Libraries
from math import *
import rospy
from std_msgs.msg import String
```

```
# Call back function
def capture(data):
    var = data.data.split(":")
    x = float(var[0])
    y = float(var[1])
    a1 = float(var[2])
    a2 = float(var[3])
    pen = int(var[4])
    inverse(x,y,a1,a2,pen)
```

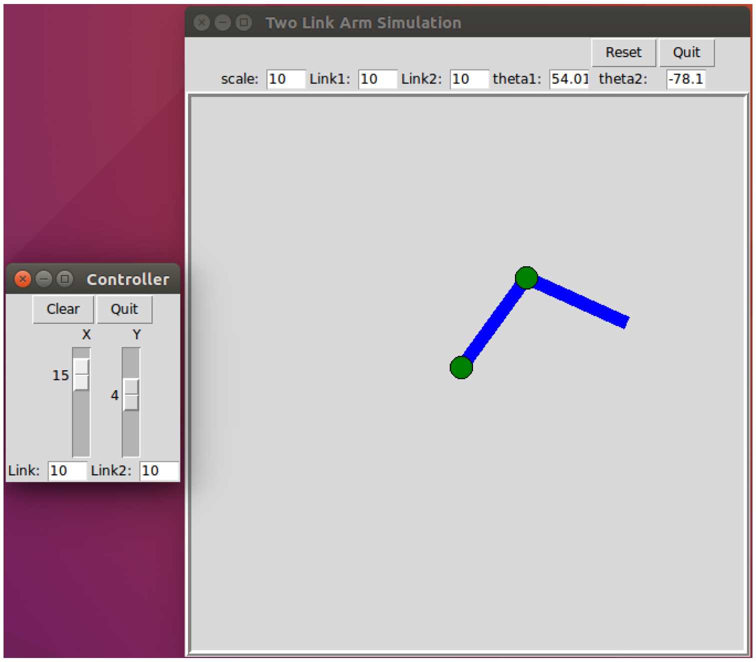
```
# Compute IK and send to simulator
def inverse(x,y,a1,a2,pen):
    if (sqrt(x*x+y*y) > a1+a2):
      print "(x,y) out of reach for links"
```

(continues on next page)



scale: 10 Link1: 10 Link2: 10 theta1: 37 theta2: 48

Fig. 23.4: The servo angle control widget



scale: 10 Link1: 10 Link2: 10 theta1: 54.01 theta2: -78.1

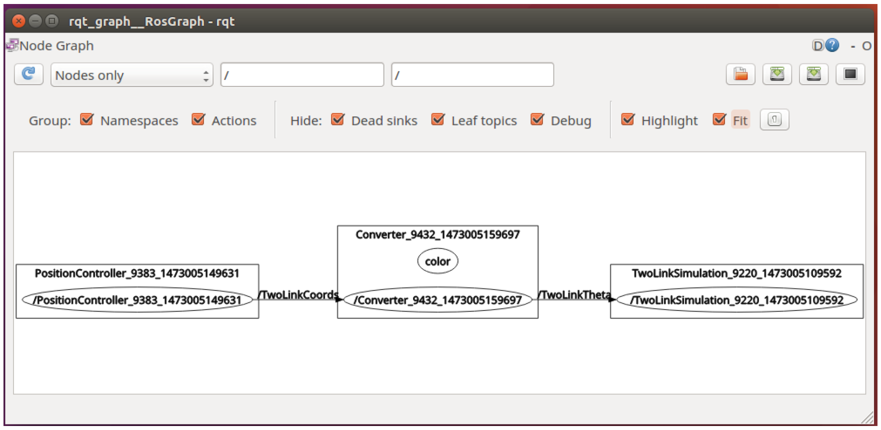
Link: 10 Link2: 10

Fig. 23.5: The position control widget

(continued from previous page)

```
    else:
      d =  (x*x+y*y-a1*a1-a2*a2)/(2.0*a1*a2)
      t2 = atan2(-sqrt(1.0-d*d),d)
      t1 = atan2(y,x) - atan2(a2*sin(t2),a1+a2*cos(t2))
      dt1 = (180.0*t1/pi)
      dt2 = (180.0*t2/pi)
      print x,y, dt1, dt2
      sliders = str(dt1) + ':' + str(dt2) + ':' + str(pen)
      pub.publish(sliders)
```

```
# ROS management
pub = rospy.Publisher('TwoLinkTheta', String, queue_size=10)
rospy.init_node('Converter', anonymous=True)
rospy.Subscriber("TwoLinkCoords", String, capture)
rospy.spin()
```



rqt_graph__RosGraph - rqt

Node Graph

Nodes only / /

Group: ☑ Namespaces ☑ Actions Hide: ☑ Dead sinks ☑ Leaf topics ☑ Debug ☑ Highlight ☑ Fit

PositionController_9383_1473005149631
/PositionController_9383_1473005149631
/TwoLinkCoords
Converter_9432_1473005159697
color
/Converter_9432_1473005159697
/TwoLinkTheta
TwoLinkSimulation_9220_1473005109592
/TwoLinkSimulation_9220_1473005109592

Fig. 23.6: The ROS Node Graph Tool rqt_graph.

23.2 Animation of the Two Link Manipulator

Two Link Manipulator Example

For the arm in the two link example, determine the joint angles to trace out a circle centered at (10,8) of radius 5. The circle can be parametrized by $x(t) = 5\cos(t) + 8$, $y(t) = 3\sin(t) + 10$, $-\pi \leq t \leq \pi$. Generate an array of points on the circle and plug them into the inverse kinematics.

Bring up the two link simulator. Then run the following code in Python. You should see an animation of the two link arm drawing a circle. The final position is given in Fig. 23.7.

```
# Bring in libraries
import rospy
from std_msgs.msg import String
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import time
from math import *
```

```
#Setup Arrays
step = 0.1
t = np.arange(-pi, pi, step)
x = 5.0*np.cos(t) + 8.0
y = 3.0*np.sin(t) + 10.0
```

```
#Initialize variables
a1 = 10.0
a2 = 10.0
d = (x*x + y*y - a1*a1 - a2*a2)/(2*a1*a2)
t2 = np.arctan2(-np.sqrt(1.0-d*d),d)
t1 = np.arctan2(y,x) - np.arctan2(a2*np.sin(t2),a1+a2*np.cos(t2))
```

```
# Setup ROS and publish joint data
pub = rospy.Publisher('TwoLinkTheta', String, queue_size=10)
rospy.init_node('talker', anonymous=True)

for i in range(t.size):
   print t1[i], "  ", t2[i]
   m = str(180*t1[i]/np.pi) + ":" + str(180*t2[i]/np.pi) + ":" + str(1)
   time.sleep(0.25)
   pub.publish(m)
```

In this example, we generate an array named t which is used for the parametric equations of the circle to generate the x and y arrays. We may use the inverse kinematic formulas to determine the arrays for θ_1 and θ_2 called t1 and t2. The θ_1 and θ_2 would be the values sent to the joint actuators. Fig. 23.7 shows the results.

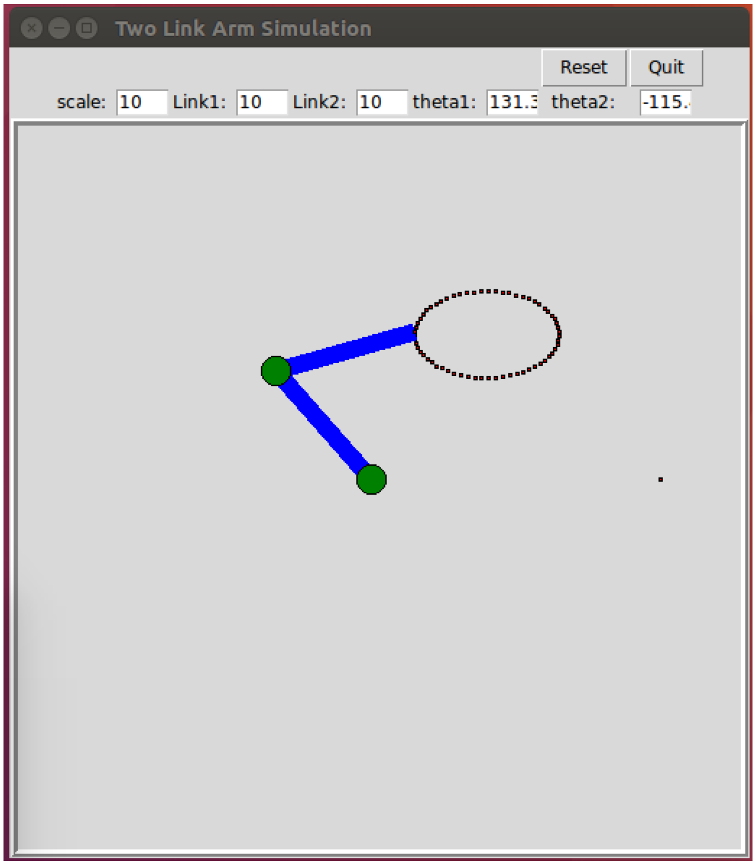
You can modify the data arrays to plot a line:

```
#Setup Arrays
t = np.arange(-5, 8, step)
x = t
y = x + 5
```

The inverse kinematics can be placed into a separate ROS node. The driving program follows (same headers as before). To connect to the simulation program, we use the inverse kinematics node as before

```
#Setup Arrays
a1 = 10
a2 = 10
step = 0.1
t = np.arange(-pi, pi, step)
x = 5.0*np.cos(t) + 8.0
y = 3.0*np.sin(t) + 10.0

pub = rospy.Publisher('TwoLinkCoords', String, queue_size=10)
rospy.init_node('talker', anonymous=True)
```

(continues on next page)



scale: 10 Link1: 10 Link2: 10 theta1: 131.3 theta2: -115.

Fig. 23.7: The output of the circle inverse kinematics code.

(continued from previous page)

```
for i in range(t.size):
   locs = str(x[i]) + ":" + str(y[i]) + ":" + str(10) + ":" + str(10)
                +":" + str(1)
   time.sleep(0.25)
   pub.publish(locs)
```

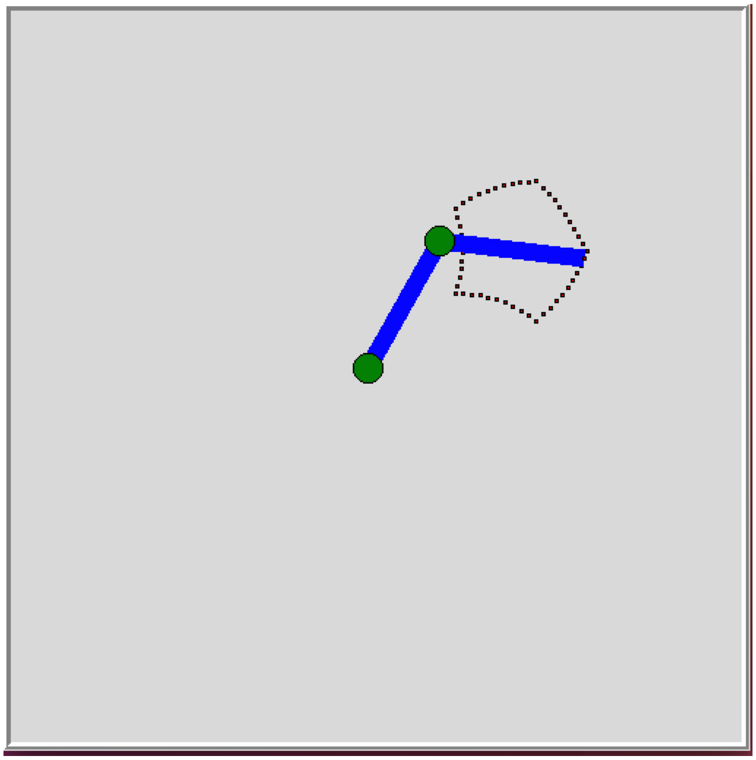


Fig. 23.8: Movement between the points - moving both linearly.

This simulation gives an idea about how to move the robotic arm and the path is correct. The motion however is not smooth. This is because we are moving the arm from position to position. This is known as position control. If you look at the curve produced, it is not a smooth curve but is a curve made of of connected segments like a polygon, Fig. 23.8. Note that the output is not actually a polygon; the sides are not straight line segments.

In between the control points, the system moves according to how the controllers are programmed. They will move the joint angles in a linear fashion. If they are moved together you will see Fig. 23.8. If they are moved one at a time you will see Fig. 23.9.

23.3 STDR - Simple Two Dimensional Robot Simulator

Note: Update for new text. Keep as ROS 1 material.

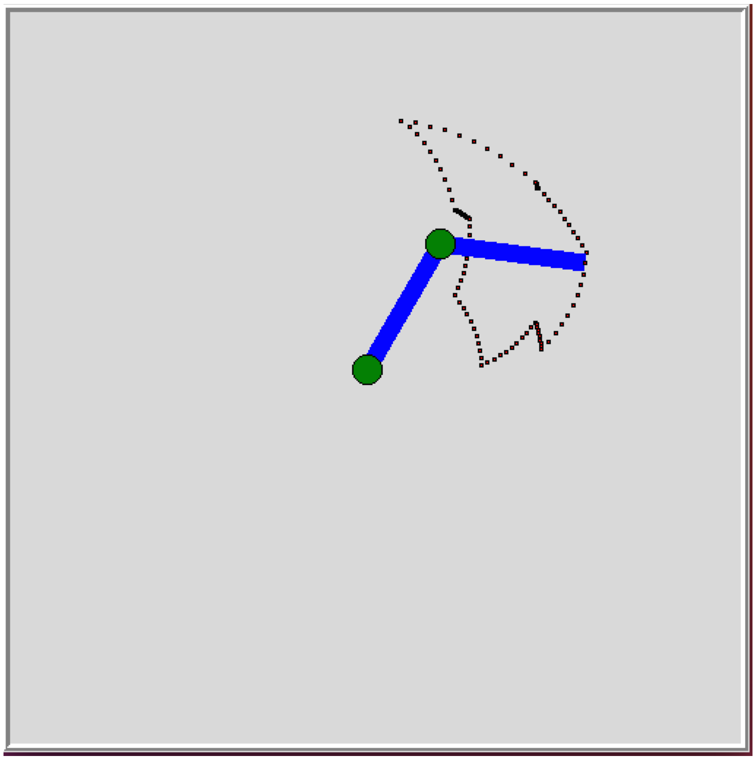


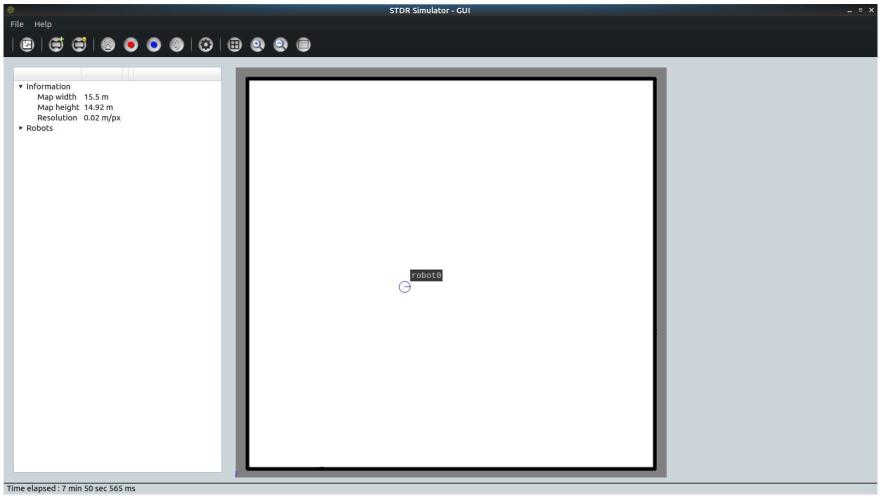
Fig. 23.9: Movement between the points - moving the servos sequentially.

Earlier you installed and tested the STDR simulator. Now we will use it to simulate a robot moving around in plane. The same ROS publish-subscribe interface is used here. As before, you write a control program that publishes motion commands to the STDR Simulator that you loaded earlier. The simulator currently implements a “magic round robot” that can move freely in any direction. This is not like your automobile which can only move forward (and turn in a specific manner). [1] The robot can be driven by using the examples in chapter one with the joy and keyboard teleop nodes when the STDR simulator was initially downloaded and built. There are other ways the robot can be driven around which we demonstrate in this section; using either a multiarray message or the twist message that contains kinematic parameters.

The simulator takes velocity commands in the x and y directions and moves the robot with those velocities. This allows for any sort of robot to be simulated by having an external node handle the specific robot’s kinematics. So then the sim does not need to preprogram all the different popular styles of robots. The sim subscribes to a Twist message (discussed below in the messages section) containing the robot velocities. It will perform the time steps (integrations) to move the robot. It is important that the user provides accurate velocity commands based on the wheels and vehicle design.

To get you up and running, we have provided a differential drive robot node which will convert wheel commands to correct robot velocities based on the differential drive kinematics. First, we show you how to run the simulator. Following that we demonstrate how to move the wheels (to move the robot).

[1] Although this may seem completely made up, we will see in later chapters that there are robots that have this type of motion.



Information
Map width 15.5 m
Map height 14.92 m
Resolution 0.02 m/px
Robots

Time elapsed : 7 min 50 sec 565 ms

Fig. 23.10: STDR Simulator.

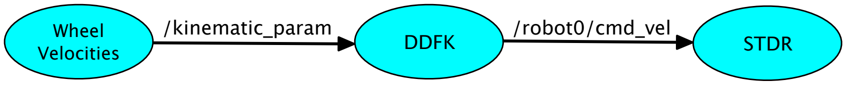


Fig. 23.11: STDR Communications

23.3.1 Running STDR

In order to run the STDR simulator the user will need to run roslaunch in order for it to be started with both the map and robot. For example, to start the simulator with the robot and a map containing no obstacles one would run the following inside of a terminal:

```
roslaunch stdr_launchers no_obst_sim.launch
```

The roslaunch command does use tab completion so other launch files are also accessible that will start up the simulator and all required nodes to start simulation of the robot. There are other launch files that include different maps and robots. These launch files will be named so that the user can easily tell which map and kinematic model that the robot will be using.

For example:

```
roslaunch stdr_launchers omni_wheeled_no_obst_sim.launch
roslaunch stdr_launchers diff_drive_no_obst_sim.launch
```

Once roslaunch executes the user will be greeted by an application looking similar to the one in Fig. 23.10. Roslaunch also starts up the ros master if there isn't one already running on the machine. It also starts up all the nodes and they can be viewed in Fig. 23.12. This graph shows the ROS nodes running for just the STDR simulator with the DDFK node and not any control code you may wish to run. So the actual node complexity is a bit more than what Fig. 23.11 implies since the STDR node is really a placeholder for the graph shown in Fig. 23.12.

The message topics also get started and can be viewed by doing a rostopic list. It is an extensive list and

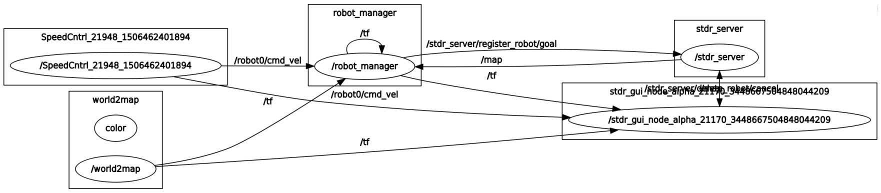

SpeedCntrl_21948_1506462401894
/SpeedCntrl_21948_1506462401894
/robot0/cmd_vel
robot_manager
/tf
/robot_manager
/robot0/cmd_vel
/stdr_server/register_robot/goal
/map
/tf
stdr_server
/stdr_server
/stdr_server/delete_robot/cancel
stdr_gui_node_alpha_21170_3448667504848044209
/stdr_gui_node_alpha_21170_3448667504848044209
world2map
color
/world2map
/tf
/tf

Fig. 23.12: Nodes running after STDR Simulator launch but before you launch your control code.

provides a look under the hood for the simulator. While getting started you will not need to interact with these topics, but later when we are working with sensors, you will need to subscribe to some of the sensor topics.

```
/map
/map_metadata
/robot0/cmd_vel
/robot0/dt
/robot0/laser_0
/robot0/odom
/robot0/pose2D
/robot0/sonar_0
/robot0/sonar_1
/robot0/sonar_2
/robot0/sonar_3
/robot0/sonar_4
/rosout
/rosout_agg
/stdr_server/active_robots
/stdr_server/co2_sources_list
/stdr_server/delete_robot/cancel
/stdr_server/delete_robot/feedback
/stdr_server/delete_robot/goal
```

```
/stdr_server/delete_robot/result
/stdr_server/delete_robot/status
/stdr_server/register_robot/cancel
/stdr_server/register_robot/feedback
/stdr_server/register_robot/goal
/stdr_server/register_robot/result
/stdr_server/register_robot/status
/stdr_server/rfid_list
/stdr_server/sound_sources_list
/stdr_server/sources_visualization_markers
/stdr_server/spawn_robot/cancel
/stdr_server/spawn_robot/feedback
/stdr_server/spawn_robot/goal
/stdr_server/spawn_robot/result
/stdr_server/spawn_robot/status
/stdr_server/thermal_sources_list
/tf
/tf_static
```

23.3.2 Driving the Robot - ROS STDR Messages

Once the simulator is up and running, you can drive the robot as before using the teleop or joystick controls. As mentioned above, we can write our own node to control the robot. This node needs to publish to either the differential drive forward kinematics or directly to the simulator. For simulating a differential drive, you will need to write a wheel control node such as the example below which publishes left and right wheel velocities. That node then coverts those to robot velocities and sends the information to the STDR simulator. For your own custom robot, you will need to write a forward kinematics node which connects to the simulator. You would then send wheel velocities to your custom FK node.

MultiArray

In order to drive the robot around in the simulator for a differential drive robot, the wheel velocities, wheel radius, and the axle length are needed to be published on the `/kinematic_param` topic as an tuple containing four values.

The Python MultiArray is implemented as a tuple. A tuple is similar to a list but not mutable like lists. They are distinguished from lists by the use of parenthesis instead of brackets.

```
>>> # tuple
...
>>> a = (1,2,3)
>>> a[0]
1
>>> a[1]
2
>>> a[1] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

```
import rospy
from math import *
import numpy as np
from std_msgs.msg import Float64MultiArray
from std_msgs.msg import MultiArrayLayout
from std_msgs.msg import MultiArrayDimension
r = 2.0
l = 3.0
def talker(w1, w2, r, l):
    pub = rospy.Publisher('kinematic_params', Float64MultiArray, queue_size=1)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    layout = MultiArrayLayout()
    layout.dim.insert(0, [MultiArrayDimension()] )
    while not rospy.is_shutdown():
        data = Float64MultiArray(data=[])
        data.layout = MultiArrayLayout()
        data.layout.dim = [MultiArrayDimension()]
```

(continues on next page)

(continued from previous page)

```
        data.layout.dim[0].label = "Parameters"
        data.layout.dim[0].size = 4
        data.layout.dim[0].stride = 1
        data.data = [w1,w2,r,l]
        pub.publish(data)
        rate.sleep()

if __name__ == '__main__':
        try:
            talker(1.5,1.0,r,l)
        except rospy.ROSInterruptException:
            pass
```

Similarly for a omni wheel robot the four wheel velocities would be published followed by the wheel radius, front axle length, and lastly the back axle length.

Twist Message

Communication with the simulator is through a ROS topic using the Twist message type. The twist message is a compact array format that can be more efficient than the string format used in the Two Link Manipulator. The Twist format is

```
# This expresses velocity in free space broken into its  linear and angular␣
↪parts.
Vector3  linear
Vector3  angular
```

The twist message is contained in the geometry package:

```
from geometry_msgs.msg import Twist
```

To set twist values on the publishing side, you can set the

```
mytwist = Twist()
mytwist.linear.x = x_vel
mytwist.linear.y = y_vel
mytwist.linear.z = z_vel
```

```
mytwist.angular.x = x_ang_vel
mytwist.angular.y = y_ang_vel
mytwist.angular.z = z_ang_vel
pub.publish(mytwist)
```

For the subscriber, you can access the data via:

```
def callback(msg):
    rospy.loginfo("Received a /cmd_vel message!")
    rospy.loginfo("Linear Components: [%f, %f, %f]"%(msg.linear.x, msg.
↪linear.y, msg.linear.z))
```

(continues on next page)

(continued from previous page)

```
    rospy.loginfo("Angular Components: [%f, %f, %f]"%(msg.angular.x, msg.
↪angular.y, msg.angular.z))
```

Note: Draft - Feedback: Jeff McGough jeff.mcgough@sdsmt.edu Most of the content is loaded. The chapters on *Example Systems* and *System Integration* still need to be developed. The *Computer Vision* and *Localization and Mapping* chapters need significant rework. We will continue to update as material is added. This material is currently used for CSC/CENG 415/515 Introduction to Robotics taught at the South Dakota School of Mines and Technology. There is a slide deck for the course which roughly tracks this text (the text is ahead in edits and organization). When the current course ends, the slide deck will be made available as well. We are currently evaluating running this as an online course. Thoughts on this are welcome.

CHAPTER TWENTYFOUR

INDEX

- genindex
- search

BIBLIOGRAPHY

[Braunl06] Thomas Bräunl. *Embedded Robotics: Mobile Robot Design and Applications with Embedded Systems*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 3540343180.

[Bra18] English Language Wikipedia BradBeattie. Repliee. https://en.wikipedia.org/wiki/Actroid#/media/File:Repliee_Q
2018. Creative Commons License.

[CLH+05] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George A Kantor, Wolfram Burgard, Lydia E. Kavraki, and Sebastian Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Cambridge, MA, June 2005.

[Com18a] Wikimedia Commons. Ied robot. https://en.wikipedia.org/wiki/Improvised_explosive_device#/media/File:IED_d
2018. Wikimedia Commons.

[Com18b] Wikimedia Commons. Keepon. https://en.wikipedia.org/wiki/Keepon#/media/File:KeeponTophatNextfest2007.j
2018. Wikimedia Commons.

[Com18c] Wikimedia Commons. Robocup. https://commons.wikimedia.org/wiki/File:RUNSWift_Naos_2010.jpg, 2018. Wikimedia Commons.

[Com18d] Wikimedia Commons. Roomba. https://commons.wikimedia.org/wiki/File:Roomba_Discovery.jpg, 2018. Wikimedia Commons.

[Com18e] Wikimedia Commons. Teleoperation. https://commons.wikimedia.org/wiki/File:Suitable_Technologies_Beam_t
2018. Wikimedia Commons.

[Com18f] Wikimedia Commons. Underwater robot. https://commons.wikimedia.org/wiki/File:ROV_working_on_a_subsea
2018. Wikimedia Commons.

[Com09] Houghton Mifflin Company. *The American Heritage Dictionary of the English Language*. Houghton Mifflin Company, 2009.

[Dub57] L.E. Dubins. On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents. *American Journal of Mathematics*, 79(3):497–516, July 1957.

[DJ00] Gregory Dudek and Michael Jenkin. *Computational principles of mobile robotics*. Cambridge University Press, New York, NY, USA, 2000. ISBN 0-521-56876-5.

[Eft18] Tilemahos Efthimiadis. Antikythera. https://commons.wikimedia.org/wiki/File:The_Antikythera_Mechanism_(3
2018. Athens, Greece, Creative Commons License - Public Domain.

[Kel13] A. Kelly. *Mobile Robotics: Mathematics, Models, and Methods*. Cambridge University Press, 2013. ISBN 9781107031159. URL: http://books.google.com/books?id=laxZAQAAQBAJ.

[LaV06] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[LS87] V. Lumelsky and A. Stepanov. Path planning strategies for point mobile automation moving amidst unknown obstacles of arbirary shape. *Algorithmica*, pages 403–430, 1987.

[MP18] English language Wikipedia Matti Paavola. Security robot. https://commons.wikimedia.org/wiki/File:Justus_robot_in_Krakow_Poland_Aug2009.jpg, 2018. Creative Commons License.

[Mun00] J.R. Munkres. *Topology*. Prentice Hall, Incorporated, 2000. ISBN 9780131816299. URL: http://books.google.com/books?id=XjoZAQAAIAAJ.

[Mur00] Robin R. Murphy. *Introduction to AI Robotics*. MIT Press, Cambridge, MA, USA, 1st edition, 2000. ISBN 0262133830.

[NAS18a] NASA. Drilling rover. https://www.nasa.gov/centers/glenn/multimedia/imagegallery/if034_scarab_rover.html#.V
2018. Public Domain Image.

[NAS18b] NASA. Nasa robonaut. https://en.wikipedia.org/wiki/Robonaut#/media/File:Robonaut2_-_first_movement_aboard_ISS.jpg, 2018. Wikimedia Commons.

[Nik10] S. Niku. *Introduction to Robotics*. John Wiley & Sons, 2010. ISBN 9780470604465. URL: http://books.google.com/books?id=2V4aGvlGt7IC.

[Nim18a] English language Wikipedia Nimur. Davinci. https://commons.wikimedia.org/wiki/File:Laproscopic_Surgery_R
2018. Creative Commons License.

[Nim18b] English language Wikipedia Nimur. Willow garage pr2. https://commons.wikimedia.org/wiki/File:PR2_Tabletop.jpg, 2018. Wikimedia Commons.

[OKane14] Jason M. O'Kane. *A Gentle Introduction to ROS*. CreateSpace Independent Publishing Platform, 1st edition, 2014. ISBN 978-14-92143-23-9.

[QGS15] Morgan Quigley, Brian Gerkey, and William Smart. *Programming Robotics with ROS*. O'Reilly Media, Sebastopol, CA, USA, 1st edition, 2015. ISBN 978-1449323899.

[RS90] J.A. Reeds and L.A. Shepp. Optimal Paths for a Car that Goes Both Forward and Backwards. *Pacific Journal of Mathematics*, 145(2):367–393, 1990.

[SN04] Roland Siegwart and Illah R. Nourbakhsh. *Introduction to Autonomous Mobile Robots*. Bradford Company, Scituate, MA, USA, 2004. ISBN 026219502X.

[Str88] Gilbert Strang. *Linear Algebra and Its Applications*. Brooks Cole, 1sr edition, 1988. ISBN 0155510053.

[ST91] H. Sussman and G. Tang. Shortest Paths for the Reed-Shepp Car: A Worked Out Example of the Use of Geometric Techniques in Nonlinear Optimal Control. Technical Report SYNCON 91-10, Dept. of Mathematics, Rutgers University, Piscataway, NJ, 1991.

[TBF05] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. Intelligent robotics and autonomous agents series. Mit Press, 2005. ISBN 9780262201629. URL: http://books.google.com/books?id=k\T1\textbackslash{ }_yOQgAACAAJ.

[Wik18a] Wikipedia. Athlete. https://en.wikipedia.org/wiki/ATHLETE#/media/File:ATHLETE_robot_climbing_a_hill.jpg 2018. Public Domain Image.

[Wik18b] Wikipedia. Aljazari. https://en.wikipedia.org/wiki/Ismail_al-Jazari#/media/File:Al-Jazari_-_A_Musical_Toy.jpg, 2018. Public Domain.

[Wik18c] Wikipedia. Asimo. https://en.wikipedia.org/wiki/ASIMO#/media/File:2005_Honda_ASIMO_01.JPG, 2018. Public Domain Image.

[Wik18d] Wikipedia. Kuka. https://en.wikipedia.org/wiki/Industrial_robot#/media/File:Automation_of_foundry_with_rob 2018. Public Domain Image.

[Wik18e] Wikipedia. Rhex. https://en.wikipedia.org/wiki/Rhex, 2018. Public Domain.

[Wik18f] Wikipedia. Robocup. https://en.wikipedia.org/wiki/RoboCup_Standard_Platform_League#/media/File:RUNSW 2018. Public Domain.

INDEX