1. Foundation for Advanced Robotics and AI

The basic principle of robotics and AI

Artificial intelligence applied to robotics development requires a different set of skills from you, the robot designer or developer. You may have made robots before. You probably have a quadcopter or a 3D printer (which is, in fact, a robot). The familiar world of Proportional Integral Derivative (PID) controllers, sensor loops, and state machines must give way to artificial neural networks, expert systems, genetic algorithms, and searching path planners. We want a robot that does not just react to its environment as a reflex action, but has goals and intent—and can learn and adapt to the environment. We want to solve problems that would be intractable or impossible otherwise. What we are going to do in this book is introduce a problem – picking up toys in a playroom—that we will use as our example throughout the book as we learn a series of techniques for applying AI techniques to our robot. It is important to understand that in this book, the journey is far more important than the destination. At the end of the book, you should gain some important skills with broad applicability, not just learn how to pick up toys.

One of the difficult decisions I had to make about writing this book was deciding if this is an AI book about robotics or a robotics approach to AI—that is, is the focus learning about robotics or learning about AI? The answer is that this is a book about how to apply AI tools to robotics problems, and thus is primarily an AI book using robotics as an example. The tools and techniques learned will have applicability even if you don't do robotics, but just apply AI to decision making to trade on the stock market. What we are going to do is first provide some tools and background to match the infrastructure that was used to develop the examples in the book. This is both to provide an even playing field and to not assume any knowledge on the reader's part. We will use the Python programming language, the ROS for our data infrastructure, and be running under the Linux operating system. I developed the examples in the book with Oracle's VirtualBox software running Ubuntu Linux in a virtual machine on a Windows Vista computer. Our robot hardware will be a Raspberry Pi 3 as the robot's on-board brain, and an Arduino Mega2560 as the hardware interface microcontroller. In the rest of this chapter, we will discuss some basics about AI, and then proceed to develop two important tools that we will use in all of the examples in the rest of the book. We will introduce the concept of soft real-time control, and then provide a framework, or model, for interfacing AI to our robot called the Observe-Orient-Decide-Act (OODA) loop.

What is AI (and what is it not)?

What would be a definition of AI? In general, it means a machine that exhibits some characteristics of intelligence—thinking, reasoning, planning, learning, and adapting. It can also mean a software program that can simulate thinking or reasoning. Let's try some examples: a robot that avoids obstacles by simple rules (if the obstacle is to the right, go left) is not an AI. A program that learns by example to recognize a cat in a video, is an AI. A mechanical arm that is operated by a joystick is not AI, but a robot arm that adapts to different objects in order to pick them up is AI.

There are two defining characteristics of artificial intelligence robots that you must be aware of. First of all, AI robots learn and adapt to their environments, which means that they change behaviors over time. The second characteristic is emergent behavior, where the robot exhibits developing actions that we did not program into it explicitly. We are giving the robot controlling software that is inherently non-linear and self-organizing. The robot may suddenly exhibit some bizarre or unusual reaction to an event or situation that seems to be odd, or quirky, or even emotional. I worked with a self-driving car that we swore had delicate sensibilities and moved very daintily, earning it the nickname Ferdinand after the sensitive, flower loving bull from the cartoon, which was appropriate in a nine-ton truck that appeared to like plants. These behaviors are just caused by interactions of the various software components and control algorithms, and do not represent anything more than that. One concept you will hear around AI circles is the Turing test. The Turing test was proposed by Alan Turing in 1950, in a paper entitled Computing Machinery and Intelligence. He postulated that a human interrogator would question an hidden, unseen AI system, along with another human. If the human posing the questions was unable to tell which person was the computer and which the human was, then that AI computer would pass the test. This test supposes that the AI would be fully capable of listening to a conversation, understanding the content, and giving the same sort of answers a person will. I don't believe that AI has progressed to this point yet, but chat bots and automated answering services have done a good job of making you believe that you are talking to a human and not a robot.

Our objective in this book is not to pass the Turing test, but rather to take some novel approaches to solving problems using techniques in machine learning, planning, goal seeking, pattern recognition, grouping, and clustering. Many of these problems would be very difficult to solve any other way. A software AI that could pass the Turing test would be an example of a general artificial intelligence, or a full, working intelligent artificial brain, and just like you, a general AI does not need to be specifically trained to solve any particular

problem. To date, a general AI has not been created, but what we do have is narrow AI, or software that simulates thinking in a very narrow application, such as recognizing objects, or picking good stocks to buy. What we are not building in this book is a general AI, and we are not going to be worried about our creations developing a mind of their own or getting out of control. That comes from the realm of science fiction and bad movies, rather than the reality of computers today. I am firmly of the mind that anyone preaching about the evils of AI or predicting that robots will take over the world has not worked or practiced in this area, and has not seen the dismal state of AI research in respect of solving general problems or creating anything resembling an actual intelligence.

There is nothing new under the sun

Most of AI as practiced today is not new. Most of these techniques were developed in the 1960s and 1970s and fell out of favor because the computing machinery of the day was insufficient for the complexity of the software or number of calculations required, and only waited for computers to get bigger, and for another very significant event – the invention of the internet. In previous decades, if you needed 10,000 digitized pictures of cats to compile a database to train a neural network, the task would be almost impossible—you could take a lot of cat pictures, or scan images from books. Today, a Google search for cat pictures returns 126,000,000 results in 0.44 seconds. Finding cat pictures, or anything else, is just a search away, and you have your training set for your neural network—unless you need to train on a very specific set of objects that don't happen to be on the internet, as we will see in this book, in which case we will once again be taking a lot of pictures with another modern aid not found in the 1960s, a digital camera. The happy combination of very fast computers; cheap, plentiful storage; and access to almost unlimited data of every sort has produced a renaissance in AI. Another modern development has occurred on the other end of the computer spectrum. While anyone can now have a supercomputer on their desk at home, the development of the smartphone has driven a whole series of innovations that are just being felt in technology. Your wonder of a smartphone has accelerometers and gyroscopes made of tiny silicon chips called microelectromechanical systems (MEMS). It also has a high resolution but very small digital camera, and a multi-core computer processor that takes very little power to run. It also contains (probably) three radios: a WiFi wireless network, a cellular phone, and a Bluetooth transceiver. As good as these parts are at making your iPhone™ fun to use, they have also found their way into parts available for robots. That is fun for us because what used to be only available for research labs and universities are now for sale to individual users. If you happen to have a university or research lab, or work for a technology company with multi-millions dollar development budgets, you will also

learn something from this book, and find tools and ideas that hopefully will inspire your robotics creations or power new products with exciting capabilities.

The example problem – clean up this room!

In the course of this book, we will be using a single problem set that I feel most people can relate to easily, while still representing a real challenge for the most seasoned roboticist. We will be using AI and robotics techniques to pick up toys in my upstairs game room after my grandchildren have visited. That sound you just heard was the gasp from the professional robotics engineers and researchers in the audience. Why is this a tough problem, and why is it ideal for this book? This problem is a close analog to the problem Amazon has in picking items off of shelves and putting them in a box to send to you. For the last several years, Amazon has sponsored the Amazon Robotics Challenge where they invited teams to try and pick items off shelves and put them into a box for cash prizes. They thought the program difficult enough to invite teams from around the world. The contest was won in 2017 by a team from Australia. Let's discuss the problem and break it down a bit. Later, in Chapter 2, we will do a full task analysis, use cases, and storyboards to develop our approach, but we can start here with some general observations. Robotics designers first start with the environment – where does the robot work? We divide environments into two categories: structured and unstructured. A structured environment, such as the playing field for a first robotics competition, an assembly line, or lab bench, has everything in an organized space. You have heard the saying A place for everything and everything in its place—that is a structured environment. Another way to think about it, is that we know in advance where everything is or is going to be. We know what color things are, where they are placed in space, and what shape they are. A name for this type of information is a prior knowledge – things we know in advance. Having advanced knowledge of the environment in robotics is sometimes absolutely essential. Assembly line robots are expecting parts to arrive in exactly the position and orientation to be grasped and placed into position. In other words, we have arranged the world to suit the robot. In the world of our game room, this is simply not an option. If I could get my grandchildren to put their toys in exactly the same spot each time, then we would not need a robot for this task. We have a set of objects that is fairly fixed – we only have so many toys for them to play with. We occasionally add things or lose toys, or something falls down the stairs, but the toys are a element of a set of fixed objects. What they are not is positioned or oriented in any particular manner – they are just where they were left when the kids finished playing with them and went home. We also have a fixed set of furniture, but some parts move – the footstool or chairs can be moved around.

This is an unstructured environment, where the robot and the software have to adapt, not the toys or furniture.

The problem is to have the robot drive around the room, and pick up toys. Let's break this task down into a series of steps: 1. We want the user to interact with the robot by talking to it. We want the robot to understand what we want it to do, which is to say, what our intent is for the commands we are giving it. 2. Once commanded to start, the robot will have to identify an object as being a toy, and not a wall, a piece of furniture, or a door. 3. The robot must avoid hazards, the most important being the stairs going down to the first floor. Robots have a particular problem with negative obstacles (dropoffs, curbs, cliffs, stairs, and so on), and that is exactly what we have here. 4. Once the robot finds a toy, it has to determine how to pick the toy up with its robot arm. Can it grasp the object directly, or must it scoop the item up, or push it along? We expect that the robot will try different ways to pick up toys and may need several trials and error attempts. 5. Once the toy is acquired by the robot arm, the robot needs to carry the toy to a toy box. The robot must recognize the toy box in the room, remember where it is for repeat trips, and then position itself to place the toy in the box. Again, more than one attempt may be required. 6. After the toy is dropped off, the robot returns to patrolling the room looking for more toys. At some point, hopefully, all of the toys are retrieved. It may have to ask us, the human, if the room is acceptable, or if it needs to continue cleaning. What will we be learning from this problem? We will be using this backdrop to examine a variety of AI techniques and tools. The purpose of the book is to teach you how to develop AI solutions with robots. It is the process and the approach that is the critical information here, not the problem and not the robot I developed so that we have something to take pictures of for the book. We will be demonstrating techniques for making a moving machine that can learn and adapt to its environment. I would expect that you will pick and choose which chapters to read and in which order according to your interests and you need, and as such, each of the chapters will be standalone lessons. The first three chapters are foundation material that support all of the rest of the book by setting up the problem and providing a firm framework to attach all of the rest of the material.

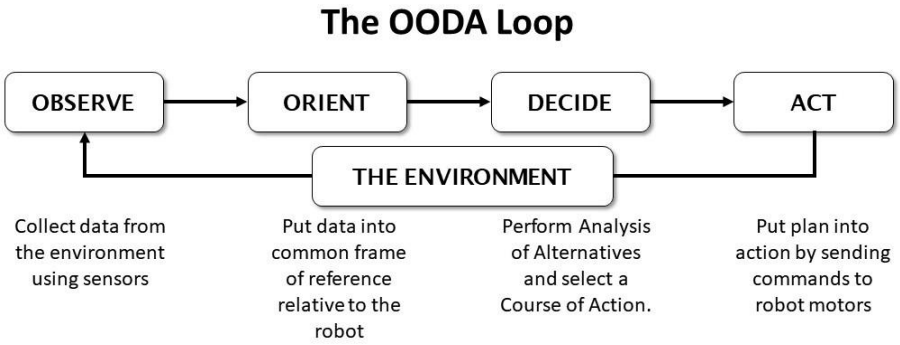
What you will learn

Not all of the chapters or topic in this book are considered *classical* AI approaches, but they do represent different ways of approaching machine learning and decision-making problems.

Building a firm foundation for robot control by understanding control theory and timing. We will be using a soft real-time control scheme with what I call a frame-based

control loop. This technique has a fancy technical name – rate monotonic scheduling—but I think you will find the concept fairly intuitive and easy to understand.

At the most basic level, AI is a way for the robot to make decisions about its actions. We will introduce a model for decision making that comes from the US Air Force, called the OODA (Observe- Orient-Decide- Act) loop. Our robot will have two of these loops: an inner loop or introspective loop, and an outward looking environment sensor loop. The lower, inner loop takes priority over the slower, outer loop, just as the autonomic parts of your body (heartbeat, breathing, eating) take precedence over your task functions (going to work, paying bills, mowing the lawn). This makes our system a type of subsumption architecture in Chapter 2, *Setting Up Your Robot*, a biologically inspired control paradigm named by Rodney Brooks of MIT, one of the founders of iRobot and designer of a robot named Baxter.



The **OODA loop** was invented by Col. John Boyd, a man also called *The Father of the F-16*. Col. Boyd's ideas are still widely quoted today, and his OODA loop is used to describe robot artificial intelligence, military planning, or marketing strategies with equal utility. The OODA provides a model for how a thinking machine that interacts with its environment might work.

Our robot works not by simply doing commands or following instructions step by step, but by setting goals and then working to achieve these goals. The robot is free to set its own path or determine how to get to its goal. We will tell the robot to *pick up that toy* and the robot will decide which toy, how to get in range, and how to pick up the toy. If we, the human robot owner, instead tried to treat the robot as a teleoperated hand, we would have to give the robot many individual instructions, such as move forward, move right, extend arm, open hand, each individually and without giving the robot any idea of why we were making those motions.

Before designing the specifics of our robot and its software, we have to match its capabilities to the environment and the problem it must solve. The book will introduce

some tools for designing the robot and managing the development of the software. We will use two tools from the discipline of systems engineering to accomplish this – use cases and storyboards. I will make this process as streamlined as possible. More advanced types of systems engineering are used by NASA and aerospace companies to design rockets and aircraft – this gives you a taste of those types of structured processes.

Artificial intelligence and advanced robotics techniques

The next sections will each detail a step-by-step example of the application of a different AI approach.

We start with object recognition. We need our robot to recognize objects, and then classify them as either toys to be picked up or not toys to be left alone. We will use a trained artificial neural network (ANN) to recognize objects from a video camera from various angles and lighting conditions.

The next task, once a toy is identified, is to pick it up. Writing a general purpose pick up anything program for a robot arm is a difficult task involving a lot of higher mathematics (google inverse kinematics to see what I mean). What if we let the robot sort this out for itself? We use genetic algorithms that permit the robot to invent its own behaviors and learn to use its arm on its own.

Our robot needs to understand commands and instructions from its owner (us). We use natural language processing to not just recognize speech, but understand intent for the robot to create goals consistent to what we want it to do. We use a neat technique called the "fill in the blank" method to allow the robot to reason from the context of a command. This process is useful for a lot of robot planning tasks.

The robot's next problem is avoiding the stairs and other hazards. We will use operant conditioning to have the robot learn through positive and negative reinforcement where it is safe to move.

The robot will need to be able to find the toy box to put items away, as well as have a general framework for planning moves into the future. We will use decision trees for path planning, as well as discuss pruning for quickly rejecting bad plans. We will also introduce forward and backwards chaining as a means to quickly plan to reach a goal. If you imagine what a computer chess program algorithm must do, looking several moves ahead and scoring good moves versus bad moves before selecting a strategy, that will give you an idea of the

power of this technique. This type of decision tree has many uses and can handle many dimensions of strategies. We'll be using it to find a path to our toy box to put toys away.

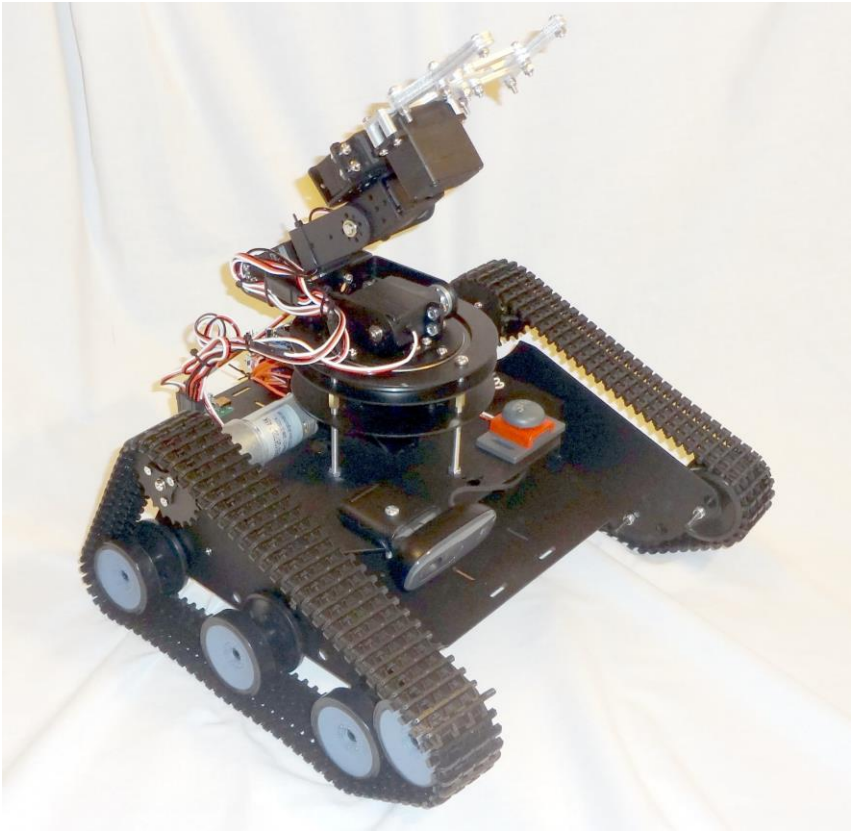
Our final practical chapter brings a different set of tools not normally used in robotics, or at least not in the way we are going to employ them.

I have four wonderful, talented, and delightful grandchildren who love to come and visit. You'll be hearing a lot about them throughout the book. The oldest grandson is six years old, and autistic, as is my grandaughter, the third child. I introduced the grandson, William, to the robot , and he immediately wanted to have a conversation with it. He asked What's your name? and What do you do? He was disappointed when the robot made no reply. So for the grandkids, we will be developing an engine for the robot to carry on a small conversation. We will be creating a robot personality to interact with children. William had one more request of this robot: he wants it to tell and respond to knock, knock jokes.

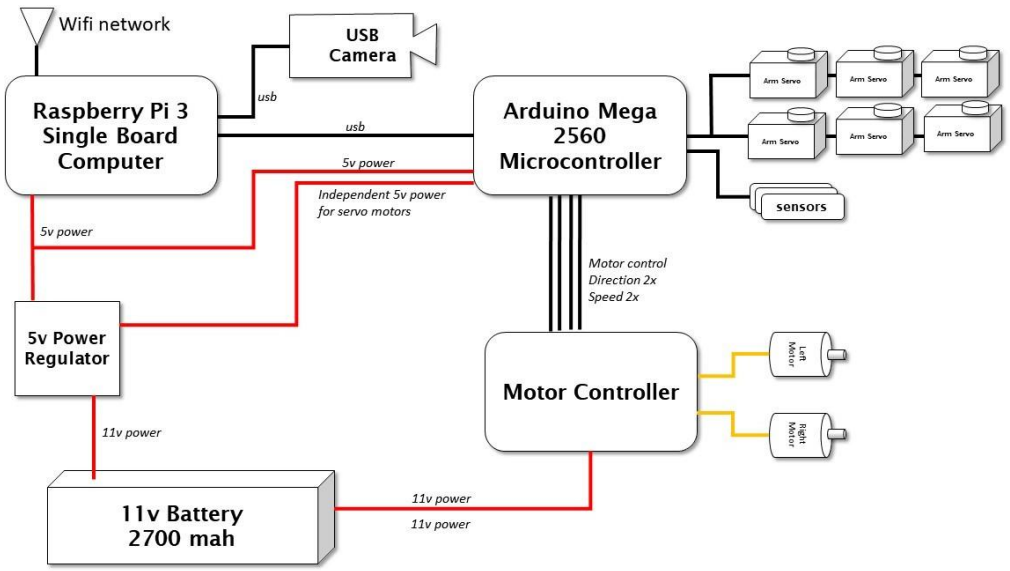
While developing a robot with actual feelings is far beyond the state of the art in robotics or AI today, we can simulate having a personality with a finite state machine and some Monte-Carlo modeling. We will also give the robot a model for human interaction so that the robot will take into account the child's mood as well. I like to call this type of software an artificial personality to distinguish it from our artificial intelligence. AI builds a model of thinking, and AP builds a model of emotion for our robot.

Introducing the robot and our development environment

This is a book about robots and artificial intelligence, so we really need to have a robot to use for all of our practical examples. As we will discuss in Chapter 2 at some length, I have selected robot hardware and software that would be accessible to the average reader, and readily available for mail order. In the Appendix, I go through all of the setup of all of the hardware and software required and show you how I put together this robot and wired up his brain and control system. The base and robot arm were purchased as a unit from AliExpress, but you can buy them separately. All of the electronics were purchased from Amazon. As shown in the photo, our robot has tracks, a mechanical six degree-of-freedom arm, and a computer. Let's call him TinMan, since, like the storybook character in The Wizard of Oz, he has a metal body and all he wants for is a brain.



Our tasks in this book center around picking up toys in an interior space, so our robot has a solid base with two motors and tracks for driving over a carpet. Our steering method is the tank-type, or differential drive where we steer by sending different commands to the track motors. If we want to go straight ahead, we set both motors to the same forward speed. If we want to travel backward, we reverse both motors the same amount. Turns are accomplished by moving one motor forward and the other backward (which makes the robot turn in place) or by giving one motor more forward drive than the other. We can make any sort of turn this way. In order to pick up toys we need some sort of manipulator, so I've included a six-axis robot arm that imitates a shoulder – elbow – wrist- hand combination that is quite dexterous, and since it is made out of standard digital servos, quite easy to wire and program. You will note that the entire robot runs on one battery. You may want to split that and have a separate battery for the computer and the motors. This is a common practice, and many of my robots have had separate power for each. Make sure if you do to connect the ground wires of the two systems together. I've tested my power supply carefully and have not had problems with temperature or noise, although I don't run the arm and drive motors at the same time. If you have noise from the motors upsetting the Arduino (and you will tell because the Arduino will keep resetting itself), you can add a small filter capacitor of 10 µf across the motor wires.



The main control of the TinMan robot is the Raspberry Pi 3 single board computer (SBC), that talks to the operator via a built-in Wi-Fi network. An Arduino Mega 2560 controller based on the Atmel architecture provides the interface to the robot's hardware components, such as motors and sensors. You can refer to the preceding diagram on the internal components of the robot. We will be primarily concerned with the Raspberry Pi3 single board computer (SBC), which is the brains of our robot. The rest of the components we will set up once and not change for the entire book. The Raspberry Pi 3 acts as the main interface between our control station, which is a PC running Linux in a virtual machine, and the robot itself via a Wi-Fi network. Just about any low power, Linux-based SBC can perform this job, such as a BeagleBone Black, Oodroid XU4, or an Intel Edison. Connected to the SBC is an Arduino 2560 Mega microcontroller board that will serve as our hardware interface. We can do much of the hardware interface with the PI if we so desired, but by separating out the Arduino we don’t have to worry about the advanced AI software running in the Pi 3 disrupting the timing of sending PWM (pulse width modulated) controls to the motors, or the PPM (pulse position modulation) signals that control our six servos in the robot arm. Since our motors draw more current than the Arduino can handle itself, we need a motor controller to amplify our commands into enough power to move the robot’s tracks. The servos are plugged directly into the Arduino, but have their own connection to the robot’s power supply. We also need a 5v regulator to provide the proper power from the 11.1v rechargeable lithium battery power pack into the robot. My power pack is a

rechargeable 3S1P (three cells in series and one in parallel) 2,700 ah battery normally used for quadcopter drones, and came with the appropriate charger. As with any lithium battery, follow all of the directions that came with the battery pack and recharge it in a metal box or container in case of fire.

Software components (ROS, Python, and Linux)

I am going to direct you once again to the Appendix to see all of the software that runs the robot, but I'll cover the basics here to remind you. The base operating system for the robot is Linux running on a Raspberry Pi 3 SBC, as we said. We are using the ROS to connect all of our various software components together, and it also does a wonderful job of taking care of all of the finicky networking tasks, such as setting up sockets and establishing connections. It also comes with a great library of already prepared functions that we can just take advantage of, such as a joystick interface.

The ROS is not a true operating system that controls the whole computer like Linux or Windows does, but rather it is a backbone of communications and interface standards and utilities that make putting together a robot a lot simpler. The ROS uses a publish/subscribe technique to move data from one place to another that truly decouples the programs that produce data (such as sensors and cameras) from those programs that use the data, such as controls and displays. We'll be making a lot of our own stuff and only using a few ROS functions. Packt Publishing has several great books for learning ROS. My favorite is Learning ROS for Robotics by Aaron Martinez and Enrique Fernandez.

The programming language we will use throughout this book, with a couple of minor exceptions, will be Python. Python is a great language for this purpose for two great reasons: it is widely used in the robotics community in conjunction with ROS, and it is also widely accepted in the machine learning and AI community. This double whammy makes using Python irresistible. Python is an interpreted language, which has three amazing advantages for us:

- Portability: Python is very portable between Windows, Mac, and Linux. Usually the only time you have to worry about porting is if you use a function out of the operating system, such as opening a file that has a directory name.
- As an interpreted language, Python does not require a compile step. Some of the programs we are developing in this book are pretty involved, and if we write them in C or C++, would take 10 or 20 minutes of build time each time we made a change. You can do a lot with that much time, which you can spend getting your program to run and not waiting for make to finish.

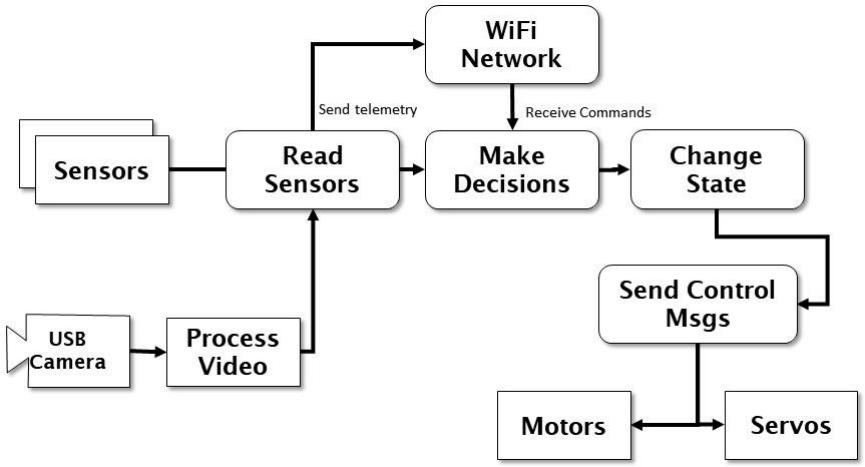
- Isolation. This is a benefit that does not get talked about much, but having had a lot of experience with crashing operating systems with robots, I can tell you that the fact that Python's interpreter is isolated from the core operating system means that having one of your Python ROS programs crash the computer is very rare. A computer crash means rebooting the computer and also probably losing all of your data you need to diagnose the crash. I had a professional robot project that we moved from Python to C++, and immediately the operating system crashes began, which shot the reliability of our robot. If a Python program crashes, another program can monitor that and restart it. If the operating system is gone, there is not much you can do without some extra hardware that can push the reset button for you. (For further information, refer to Python Success Stories https://www.python.org/about/success/devil/).

Robot control systems and a decision-making framework

Before we dive into the coding of our base control system, let's talk about the theory we will use to create a robust, modular, and flexible control system for robotics. As I mentioned previously, we are going to use two sets of tools in the sections: soft real-time control and the OODA loop. One gives us a base for controlling the robot easily and consistently, and the other provides a basis for all of the robot's autonomy.

Soft real-time control

The basic concept of how a robot works, especially one that drives, is fairly simple. There is a master control loop that does the same thing over and over; it reads data from the sensors and motor controller, looks for commands from the operator (or the robot's autonomy functions), makes any changes to the state of the robot based on those commands, and then sends instructions to the motors or effectors to make the robot move:



The preceding diagram illustrates how we have instantiated the OODA loop into the software and hardware of our robot. The robot can either act autonomously, or accept commands from a control station connected via a wireless network.

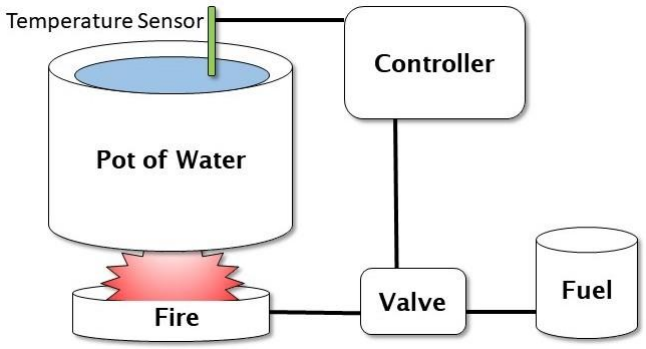
What we need to do is perform this control loop in a consistent manner all of the time. We need to set a base frame rate, or basic update frequency, in our control loop. This makes all of the systems of the robot perform better. Without some sort of time manager, each control cycle of the robot takes a different amount of time to complete, and any sort of path planning, position estimate, or arm movement becomes more complicated.

If you have used a PID controller before to perform a process, such as driving the robot at a consistent speed, or aiming a camera at a moving target, then you will understand that having even-time steps is important to getting good results.

Control loops

In order to have control of our robot, we have to establish some sort of control or feedback loop. Let's say that we tell the robot to move 12 inches (30 cm) forward. The robot has to send a command to the motors to start moving forward, and then have some sort of mechanism to measure 12 inches of travel. We can use several means to accomplish this, but let's just use a clock. The robot moves 3 inches (7.5 cm) per second. We need the control loop to start the movement, and then at each update cycle, or time through the loop, check the time, and see if 4 seconds has elapsed. If it has, then it sends a stop command to the motors. The timer is the control, 4 seconds is the set point, and the motor is the system that

is controlled. The process also generates an error signal that tells us what control to apply (in this case, to stop). The following diagram shows a simple control loop. What we want is a constant temperature in the pot of water:



The Valve controls the heat produced by the fire, which warms the pot of water. The Temperature Sensor detects if the water is too cold, too hot, or just right. The Controller uses this information to control the valve for more heat. This type of schema is called a closed loop control system.

You can think of this also in terms of a process. We start the process, and then get feedback to show our progress, so that we know when to stop or modify the process. We could be doing speed control, where we need the robot to move at a specific speed, or pointing control, where the robot aims or turns in a specific direction.

Let's look at another example. We have a robot with a self-charging docking station, with a set of light emitting diodes (LEDs) on the top as an optical target. We want the robot to drive straight into the docking station. We use the camera to see the target LEDs on the docking station. The camera generates an error, which is the direction that the LEDs are seen in the camera. The distance between the LEDs also gives us a rough range to the dock. Let's say that the LEDs in the image are off to the left of center 50% and the distance is 3 feet (1 m) We send that information to a control loop to the motors – turn right (opposite the image) a bit and drive forward a bit. We then check again, and the LEDs are closer to the center (40%) and the distance is a bit less (2.9 feet or 90 cm). Our error signal is a bit less, and the distance is a bit less, so we send a slower turn and a slower movement to the motors at this update cycle. We end up exactly in the center and come to zero speed just as we touch the

docking station. For those people currently saying "But if you use a PID controller …", yes, you are correct, I've just described a "P" or proportional control scheme. We can add more bells and whistles to help prevent the robot from overshooting or undershooting the target due to its own weight and inertia, and to damp out oscillations caused by those overshoots.

The point of these examples is to point out the concept of control in a system. Doing this consistently is the concept of real-time control.

In order to perform our control loop at a consistent time interval (or to use the proper term, deterministically), we have two ways of controlling our program execution: soft real time and hard real time.

A hard real-time system places requirements that a process executes inside a time window that is enforced by the operating system, which provides deterministic performance – the process always takes exactly the same amount of time.

The problem we are faced with is that a computer running an operating system is constantly getting interrupted by other processes, running threads, switching contexts, and performing tasks. Your experience with desktop computers, or even smart phones, is that the same process, like starting up a word processor program, always seems to take a different amount of time whenever you start it up, because the operating system is interrupting the task to do other things in your computer.

This sort of behavior is intolerable in a real-time system where we need to know in advance exactly how long a process will take down to the microsecond. You can easily imagine the problems if we created an autopilot for an airliner that, instead of managing the aircraft's direction and altitude, was constantly getting interrupted by disk drive access or network calls that played havoc with the control loops giving you a smooth ride or making a touchdown on the runway.

A real-time operating system (RTOS) allows the programmers and developers to have complete control over when and how the processes are executing, and which routines are allowed to interrupt and for how long. Control loops in RTOS systems always take the exact same number of computer cycles (and thus time) every loop, which makes them reliable and dependable when the output is critical. It is important to know that in a hard real-time system, the hardware is enforcing timing constraints and making sure that the computer resources are available when they are needed.

We can actually do hard real time in an Arduino microcontroller, because it has no operating system and can only do one task at a time, or run only one program at a time. We have complete control over the timing of any executing program. Our robot will also have a more capable processor in the form of a Raspberry Pi 3 running Linux. This computer, which has some real power, does quite a number of tasks simultaneously to support the operating system, run the network interface, send graphics to the output HDMI port, provide a user interface, and even support multiple users.

What is a robot?

The word robot entered the modern language from the play R.U.R. by the Czech author, Karel Capek, which was published back in 1920. Roboti is supposed to be a Czech word meaning forced servitude. In the play, an industrialist learns how to build artificial people – not mechanical, metal men, but made of flesh and blood, who emerge from a factory fully grown. The English translation of the name as Rossum's Universal Robots (R.U.R.) introduced the word "robot" to the world.

Subsumption architecture

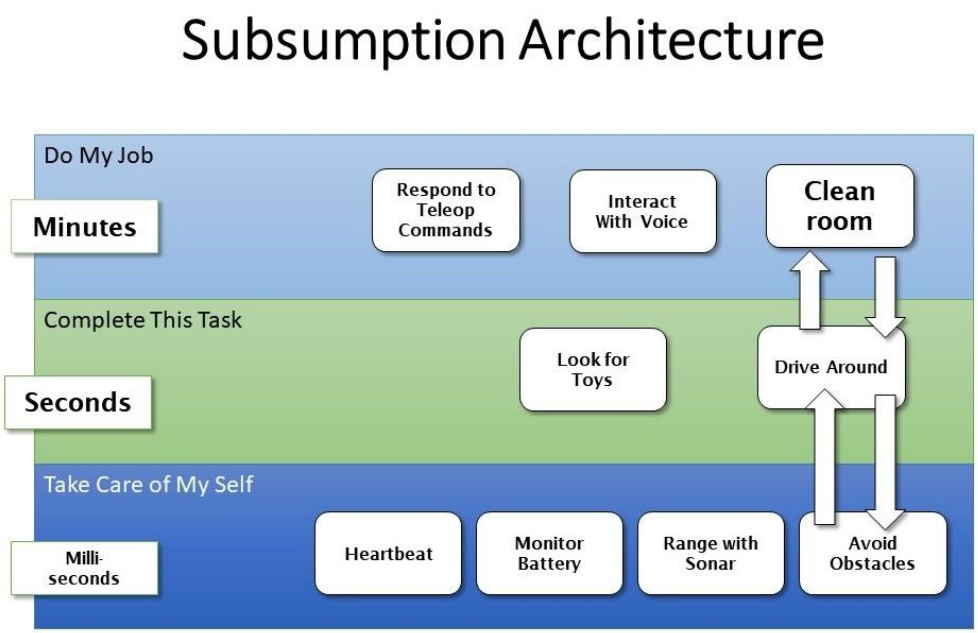
At this point, I want to spend a bit more time on the idea behind the subsumption architecture, and point out some specifics of how we will be using this concept in the design of our robot project. Many of you will be familiar with the concept from school or from study, and so you can look at my diagram and then move on. For the rest of us, let's talk a bit about this biologically inspired robot concept.

Subsumption architecture was originally described by Dr. Rodney Brooks, a professor at MIT, who would later help found iRobot Corporation and invent the Baxter Robot. Rodney was trying to develop analogues of insect brains in order to understand how to program intelligent robots. Robots before this time (1986) were very much single-threaded machines that pretty much only did one thing at a time. They read sensors, made decisions and then acted – and only had one goal at any one time. Creatures such as flies or ants have very simple brains but still manage to function in the real world. Brooks reasoned that there were several layers of closed-loop feedback processes going simultaneously.

The basic concept of subsumption has been around for some time, and it has been adapted, reused, refined, and simplified in the years since it was first introduced. What I am presenting here is my take, or my interpretation, of how to apply the concept of subsumption to a robot in the context of what we are trying to accomplish.

The first aspect to understand is that we want our robot to act on a series of goals. The robot is not simply reacting to each stimulus in total isolation, but is rather carrying out some sort of goal-oriented behavior. The goal may be to pick up a toy, or navigate the room, avoiding obstacles. The paradigm we are creating has the user set goals for the robot and the robot determine how to carry those goals out, even if the goal is simply to move one meter forward.

The problem begins when the robot has to keep more than one goal in mind at a time. The robot is not just driving around, but driving around avoiding obstacles and looking for toys to pick up. How do we arbitrate between different goals, to determine which one has precedence? The answer is found in the following diagram:



We will divide the robot's decision-making systems into three layers. Each layer has a different level of responsibility and operates on a different time scale. At the lowest levels are what we might call the robot's autonomic nervous system – it contains the robot's internal health-keeping and monitoring functions. These processes run very fast – 20 times a second or so, and only deal with what is inside the robot. This would include reading internal

sensors, checking battery levels, and reading and responding to heartbeat messages. I've labeled this level take care of myself.

The next level handles individual tasks, such as driving around, or looking for toys. These tasks are short term and deal with what the sensors can see. The time period for decisions is in the second range, so these tasks might have one or two hertz update rates, but slower than the internal checks. I call this level complete the task – you might call it drive the vehicle or operate the payload.

The final and top level is the section devoted to completing the mission, and it deals with the overall purpose of the robot. This level has the overall state machine for finding toys, picking them up, and then putting them away, which is the mission of this robot. This level also deals with interacting with humans and responding to commands. The top level works on tasks that take minutes, or even hours, to complete.

The rules of the subsumption architecture – and even where it gets its name – have to do with the priority and interaction of the processes in these layers. The rules are as follows (and these are my version):

- Each layer can only talk to the layers next to it. The top layer talks only to the middle layer, and the bottom layer also talks only to the middle layer. The middle layer can communicate with either.
- The layer with the lower level has the highest priority. The lower level has the ability to interrupt or override the commands from higher layers.

Think about this for a minute. I've given you the example of driving our robot in a room. The lowest level detects obstacles. The middle level is driving the robot in a particular direction, and the top layer is directing the mission. From the top down, the uppermost layer is commanded to clean up the room, the middle layer is commanded to drive around, and the bottom layer gets the command left motor and right motor forward 60% throttle. Now, the bottom level detects an obstacle. It interrupts the drive around function and overrides the command from the top layer to turn the robot away from the obstacle. Once the obstacle is cleared, the lowest layer returns control to the middle layer for the driving direction.

Another example could be if the lowest layer loses the heartbeat signal, which indicates that something has gone wrong in the software. The lowest layer causes the motors to halt, overriding any commands from the upper layers. It does not matter what they want; the robot has a fault and needs to stop. This priority inversion of the lowest layers having the highest

priority is the reason we call this a subsumption architecture, since the lower layers can subsume – or take precedence over – the higher layers.

The major benefit of this type of organization is that it keeps procedures clear as to which events, faults, or commands take precedence over others, and prevents the robot from getting stuck in an indecision loop.

Each type of robot may have different numbers of layers in their architecture. You could even have a supervisory layer that controls a number of other robots and has goals for the robots as a team. The most I have had so far has been five, for one of my self-driving car projects.

Software setup

To match the examples in this book, and to have access to the same tools that are used in the code samples, you will have to set up three environments:

- A laptop or desktop computer: This will run our control panel, and also be used to train neural networks. I used a Windows 10 computer with Oracle VirtualBox supporting a virtual machine running Ubuntu 16.14. You may run a computer running Ubuntu or another Linux operating system by itself (without Windows) if you want. Several of the AI packages we will use in the tutorial sections of the book will require Ubuntu 16 or later to run. We will load ROS on this computer. I will also be using a PlayStation-type game controller on this computer for teleoperation (remote control) of the robot.
- Raspberry Pi 3: Also running Ubuntu Linux (you can also run other Linux versions, but you will have to make any adjustments between those OS versions yourself). The Pi 3 also runs ROS. We will cover the additional libraries we need in each section of the text.
- Arduino Mega 256: We need to be able to create code for the Arduino. I'm using the regular Arduino IDE from the Arduino website. It can be run in Windows or in Linux. Installation will be covered regardless.

Laptop preparation

I will just cover creating the Ubuntu Linux virtual machine under VirtualBox, since that is my setup. You can find instructions for installing Ubuntu 16.04 LTS without VirtualBox at https://tutorials.ubuntu.com/tutorial/tutorial-install-ubuntu-desktop#0:

1. Download and install VirtualBox from https://www.virtualbox.org/wiki/

Downloads. Pick the version that matches your computer setup (Windows).

2. Download the Ubuntu system image from https://www.ubuntu.com/download/desktop. It will be an .iso file that is quite large (almost 2 GB). An .iso is a disk image file that is a byte-for-byte copy of another filesystem.
3. Open VirtualBox and select **New.**
4. Make up a descriptive name for this virtual machine and select **Linux** and **Ubuntu (64-bit)** in the **Type** and **Version** fields. Select **Next**.
5. Set a **Base Memory size**. I picked **3** GB.
6. Select a size for your virtual machine partition. I made a **40** GB virtual drive.
7. Click **Next.**
8. Select **Start** (green arrow) and pick your **Media Source** as the .iso we downloaded in Step 2.
9. Finish the installation by following the prompts.
10. Restart the virtual machine once you are finished.

Installing Python

The Linux Ubuntu system will come with a version of Python. I am going to assume that you are familiar with Python, as we will be using it throughout the book. If you need help with Python, *Packt Publishing* has several fine books on the subject.

Once you log on to your Virtual Machine, check which version of Python you have by opening up a terminal window and typing python at the command prompt. You should see the Python version as follows:

```
Python 2.7.12 (default, Dec  4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can see that I have version 2.7.12.

We are going to need several add-on libraries that augment Python and extend its capability. The first thing to check is to see whether you have pip installed. **Pip** is a **Python Installation Package** that helps to load other packages from the internet to extend Python. Check to see whether you have pip by typing the following:

```
pip
```

If you get No command 'pip' found, then you need to install pip. Enter the following:

```
Sudo apt-get install python-pip python-dev build-essential
Sudo pip install –upgrade pip
```

Now we can install the rest of the packages that we need. To begin with, we need the python math packages numpy, the scientific python libraries scipy, and the math plotting libraries matplotlib. I will direct you to install other packages as required in the relevant chapters.

Let's install our other libraries. We need the numerical python library (numpy), the scientific python library (scipy), and matplotlib for making graphs:

```
>>sudo apt-get install python-numpy python-scipy python-matplotlib python-
sympy
```

If you want to use the iPython (interactive Python) and Jupyter Notebook process to test your code (and document it at the same time), you can install those as well. I will not be using them in this book, but I do admit that they are quite useful.

```
>>sudo apt-get install ipython ipython-notebook
```

I'll cover the other Python libraries that we will use later (Open CV, Scikit-Learn, Keras, and so on) as we need them in the appropriate chapters.

Installing ROS on the laptop

We need a copy of ROS to talk to our robot and to execute development. You can find directions on how to do this at the ROS website repository at http://wiki.ros.org/kinetic/Installation/Ubuntu.

I'll give you the quick and dirty method here:

Get on your Linux laptop/desktop computer – virtual or otherwise – and get to a command prompt. We need to establish the source of the software. We do this as follows:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main"
> /etc/apt/sources.list.d/ros-latest.list'
```

We need to set up our key to get access to the software with this command. Remember that you can cut and paste this from the ROS website as well:

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key
421C365BD9FF1F717815A3895523BAEEB01FA116
```

At this point, it is recommended that you check that your Linux installation is up to date. We can do this by entering the following command:

```
sudo apt-get update
```

This is going to take a bit of time depending on your internet connection and computer.

Now, we are finally ready to install the main ROS distribution. Since we are on a desktop or laptop computer, we can get the whole thing, so that we have all of the tools and user interfaces. Note that we are using ROS Kinetic, the latest full release as at the time of publication of this book:

```
sudo apt-get install ros-kinetic-desktop-full
```

Expect that this step can take quite a while – it was about 30 minutes on my computer. Once it finally finishes, we can proceed to setting up the ROS on our computer. We need to set up the ROS dependency manager using these commands:

```
sudo rosdep init
rosdep update
```

rosdep keeps up with which other packages your program depends on, and will download them automatically if they are missing. This is a good feature to have, and we won't be needing it very often – just when we install a new package.

This next part is really important. ROS uses several environment variables to configure some important parts of the system, such as keeping up with which computer is in charge. The default configuration is set by doing the following:

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc source
~/.bashrc
```

The first setup appends the /opt/ros/kinetic/setup.bash to our default .bashrc file so that those environment variables will get automatically created each time we start a new terminal window. The .bashrc script is executed when a new bash shell program is started.

We can check the default setup by looking at the environment variables. Let's get only the ones with the string ROS inside them by using a pipe (|) symbol and the grep command:

```
>>:~$ env | grep ROS
ROS_ROOT=/opt/ros/kinetic/share/ros
ROS_PACKAGE_PATH=/opt/ros/kinetic/shar
e ROS_MASTER_URI=http://localhost:11311

SESSION_MANAGER=local/BookROS:@/tmp/.ICE-
unix/1847,unix/BookROS:/tmp/.ICE- unix/1847

ROS_DISTRO=kinetic
ROS_ETC_DIR=/opt/ros/kinetic/etc/ros
```

The most important variable in this set is the ROS_MASTER_URI. This points to the place where the ROSCORE program is running. Normally, this will be on the robot. Let's say the robot's IP address is 192.168.1.15. Then, we set the ROS_MASTER_URI to be http://192.168.1.15:11311. You can also set up an entry in the /etc/hosts file with the name of the robot and the IP address as follows:

```
>>:sudo nano /etc/hosts

192.168.1.15            tinma
```

Then you can use the hostname tinman in place of the IP address:

```
env ROS_MASTER_URI=http://tinman:11311.
```

Remember to change this in the. bashrc script as well (nano .bashrc).

Setup of Raspberry Pi 3

I ended up doing two different operating systems on the Raspberry Pi 3, depending on whether or not I wanted to use the Google Voice interface. In the chapter on Artificial Personality, we will be using the Google Voice Assistant as one option for providing the robot with the ability to listen and respond to commands, as well as adding some personality and additional conversation capability. The easiest way to perform this is to use the Raspbian operating system image that Google provides with the DIY Voice Kit. You can see https://aiyprojects.withgoogle.com/voice to look at the kit. This very inexpensive piece of hardware ($24.95) includes a very nice speaker, a pair of sensitive microphones, and even a cool LED light-up button. We will cover the use of this hardware add-on in the chapter on Artificial Personality. It is much easier to use the operating system image provided with that kit than try and transplant all that capability onto an already built Raspberry Pi operating system. So, if you want to do the voice part of the robot project, go to that chapter and we will cover the setup of the Raspberry Pi 3 image and the Raspbian operating system there.

Otherwise, if you don't want to use the Google Voice Assistant, you can build up a Raspberry Pi 3 with another operating system image. I happen to prefer to run Ubuntu on my Pi 3 to match my virtual machine.

For this setup, we will use an image provided by Ubuntu. Go to the Ubuntu Wiki website concerning the Raspberry Pi 3 (https://wiki.ubuntu.com/ARM/RaspberryPi).

The basic step, which you can follow on the website, is to prepare an SD card with the operating system image on it. I used Win32DiskImager, but there are several programs available that will do the job. You need an SD card with at least 8 GB of space –and keep in mind you are erasing the SD card in doing this.

Download the disk image from (https://downloads.ubiquityrobotics.com/*)* and pick the version for the Raspberry Pi 3. It is 1.2 GB of data, so it may take a bit. This image already has the ROS on it, so it will save a lot of time later.

Follow the directions with your SD card – the website advises using a Class 10 memory card of at least 8 GB, or preferably 16 GB. Put the SD card in your reader and start up your disk imager program. Double (and triple) check that you are picking the right drive letter – you are erasing the disk in that drive. Select the disk image you downloaded – I used the 16.04.2 version. Hit the **write** button and let the formatter create your Pi 3 disk image on the SD card.

You can follow the usual setup for setting your language and keyboard, as well as setting up the network. I like to use a static IP address for the robot, since we will be using this a lot. Use the same instructions from the preceding section on setting up ROS environment variables. Put your robot's name in the /etc/hosts file and set the ROS_MASTER_URI to the Pi 3's host name – tinman in my case.

The operating system comes with Python already installed, as before when we set up the laptop/desktop virtual machine, so we follow the same procedures as previously to load the python libraries NumPy, SciPy, and a new package, Pyserial. We need this for talking to the serial port:

```
>>: pip install pyserial
```

VNC

One tool that I have added to my Raspberry Pi 3 is Virtual Network Computing, or VNC. This utility, if you are not familiar with it, allows you to see and work with the Pi 3 desktop as if you were connected to it using a keyboard, a mouse, and a montor. Since the Pi 3 is physically installed inside a robot that travels by itself, attaching a keyboard, mouse, and monitor is not often convenient (or possible). There are many different versions of VNC, which is a standard protocol used among many Unix-type – and non-Unix type - operating systems. The one I used is called RealVNC. You need two parts – the server and the client. The server side runs on the Pi 3 and basically copies all of the pixels appearing on the screen and sends them out the Ethernet port. The client catches all of this data and displays it to you on another computer. Let's install the VNC server on the Pi 3 using this command:

```
>>: sudo apt-get install realvnc-vnc-server
```

You can reference the RealVNC website at https://www.realvnc.com/en/raspberrypi/. This will cover configuration items and how to set up the software. The VNC software is generally included in most Pi 3 Raspbian releases.

Load the viewer on your Windows PC, Linux virtual machine, or do like I did, and load VNC on your Apple iPad. You will find the ability to log directly into the robot and use the desktop tools to be very helpful.

Setting up catkin workspaces

We will need a catkin workspace on your development machine –laptop or desktop – as well as on the Raspberry Pi. Follow the instructions at http://wiki.ros.org/catkin/Tutorials/create_a_workspace.

If you are already a user of ROS, then you know what a catkin workspace is, and how it is used to create packages that can be used and deployed as a unit. We are going to keep all of our programs in a package we will call tinman. Let's go ahead and put this package together. It takes just a few steps.

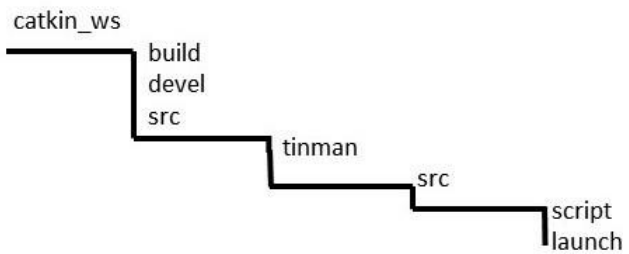
Start in the home directory:

```
mkdir –p catkin_ws/src
cd catkin_ws/src
catkin_make

source devel/setup.bash
catkin_create_pkg tinman
catkin_make

cd src/tinman/src
mkdir script
mkdir launch
```

You'll be left with a directory structure that looks something like this:



Hardware

The TinMan robot is based on a kit I found from a Chinese company named Robo-Soul. It is called the TK-6A Robot base with a 6-DOF Robot arm. The kit arrived in a timely

fashion via post, but arrived with no instructions whatsoever. Also, the pictures on the website did not match the kit, either, so I have basically no guide to putting this thing together, other than trial and error. I will provide an abbreviated version here in the book that will get you through the rough parts. A complete version will be on the website for the book at http:// github.com/fgovers/ai_and_robots.

Beginning at the beginning – knolling

The best way to start with a kit this complex, particularly when you don't have instructions, is by knolling. What is knolling, you ask? **Knolling** is the process of laying out all of the parts in an orderly fashion, all at right angles, so that you can see what you are working with, as demonstrated in the following image:

Knolling was – discovered – by a janitor by the name of Andrew Kromelow, who worked at Frank Gehry's furniture factory. The factory designed and made chairs for

Knoll, a company started by Florence Knoll, who designed furniture with simple geometric forms. Each night, Andrew would arrange all of the tools and workbenches in careful arrays of rectangular forms, which he called *knolling* in honor of the furniture. We use knolling to figure out just what we have and what order to put it together in.

Assembling the tracks

We start by assembling the track components. Grab the big triangular plates. The cut-out part goes in the back at the top, since that is where the motor and drive wheel goes. You will have one of two track configurations – one with either five small metal wheels, or, like me, a setup with four large plastic bogie wheels. I believe that the plastic wheels will be the default going forward. Each wheel goes into the smaller holes on the triangle piece – ignore the big holes. The wheels are held in place by the black hex bolts with bare shafts, and fastened with the nylon lock nuts provided. Leave the frontmost wheel off (or remove it later when we mount the tracks):

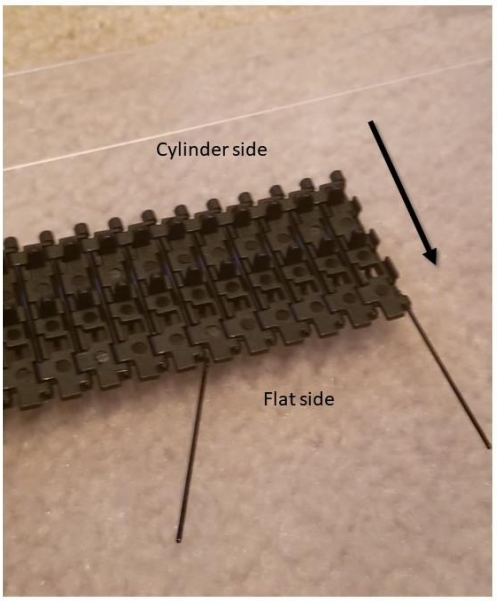
Next, we construct the drive wheels. Each drive wheel is made up of two sprocket wheels (with the rectangular teeth, a motor coupler (thick brass cylinder), three brass spacers (thin hexagonal hollow parts) and a triangular cap with four holes. Note that one of the sprocket wheels has a round opening and one has a hexagonal opening. The brass motor coupler likewise has a hexagonal end and a round end. The motor coupler runs down the center of the drive wheel assembly. Attach the three brass spacers and the triangular end cap to the sprocket wheel with the hexagon opening using small round-

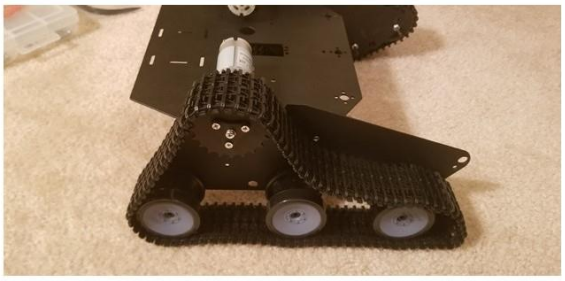
head screws and lock washers. Put the motor coupler into the hexagonal hole and attach with a larger round head screw. Put the other sprocket wheel on the spacers and attach with small screws. Repeat for the other drive wheel. You will notice the small holes in the motor coupler for set screws to be added later – make sure you can get to these holes. There are sets of holes on either side of the motor coupler. I had to disassemble one of my drive wheels to reposition the set- screw holes:

Attach the motor mounts (L-shaped bracket) to one of the robot base plates (largest plate with clipped corners) – both plates are identical. Use four longer screws and lock washers. Next, attach the motor to the motor mount using three screws. Now it gets exciting, as we can attach the two triangular track units at the side of the base plate to the upper pair of holes in the triangular drive plates, leaving the round corner for the motor to poke through. Attach the drive wheels to the motors by sliding the motor shaft into the coupler and attach with set screws (use the regular small round head screws).

Mounting the tracks

The most difficult part of the base assembly is getting the tracks onto the bogie wheels. First, you must assemble the tracks into continuous loops. The tracks are composed of a bunch of small plastic tread units that are connected to each other with small metal pins. If you look at your tracks, one end will have a pin, and the other will not. You have to remove the pin by pulling or pushing it out using a thumbtack (which was helpfully provided in the kit). You want to push the pin away from the side with the small cylinders – it needs to come out the other side. You can see this in the following image. Pull the pin out far enough to engage the two ends of the track, and carefully push it back in to connect the track. I did not need the extra track sections that were provided. Now we have a loop of track. If you put the frontmost bogie wheel on the drive section, remove it now. Loop the track around the bogie wheels and over the drive wheel. You will have to adjust the drive wheel in or out to engage the sprockets with the tracks. Now you have to lever the front drive bogie wheel into place by angling the long screw into its hole and tightening the nut until it all drops into place. This took a fair amount of effort and about an hour of careful wiggling to get it all to line up. You must keep at it until you can move the tracks easily around the bogie wheels and the drive wheel. If one of the bogie wheels is stuck, loosen up the lock nut just a tiny bit to allow it to turn without binding. There is no track tension adjustment on this kit, which can be a problem. You can make the front bogie wheel a slot rather than a hole to get some tension adjustment. I did not have to do this on my version of the robot kit:





Now we can assemble the other large base plate on the bottom two holes in the track plates and our drive base is complete. The second base plate goes upside down relative to the first plate – the bent outer section goes up. I staggered the two plates by rotating the bottom plate 180 degrees, but they can also go exactly parallel if you want. I liked the staggered arrangement for later placement of sensors.

You will need to solder wires onto the drive motors before installing the robot arm. Some very nice *spaghetti* wire was included in the kit for this purpose.

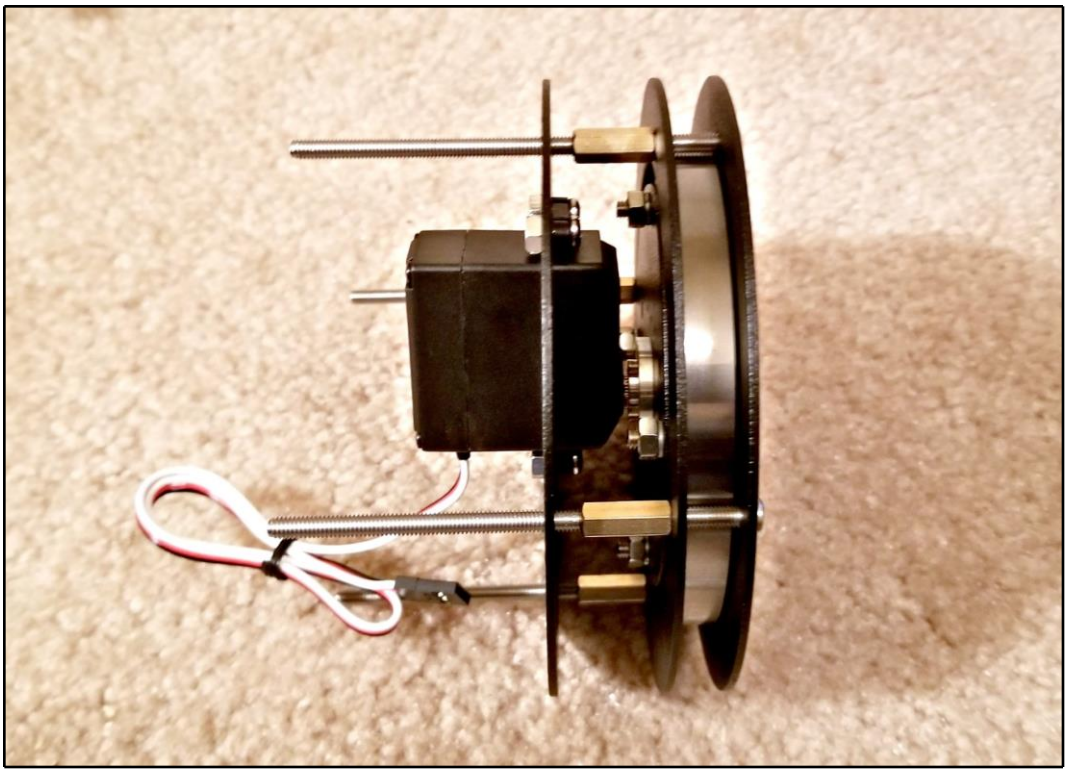
Now I have some good news and some bad news: We are done with the base (good news!) but we still have to do the arm, and it is much more difficult to assemble (bad news!).

An important note regarding Servo installation: you need to install the servos in the arm at the middle of their travel. Each servo can turn its gears through 170 degrees of angle. You need to assemble the arm with the servos in the middle. I ran the servos all the way to the left by turning the gear by hand, and then all the way to the right, and then picked a point half-way before putting them into the arm. I tried to visualize each arm joint in the middle of its travel, and assemble the arm that way.

Arm base assembly (turntable)

The turntable is the rotating assembly that forms the base of the robot arm, and will create the robot shoulder rotate axis. This is a fairly involved part that I approached with a great deal of skepticism. However, we will see that the result is most satisfactory and appears to be a very solid design. We start with two critical steps. You must do the following first, or you will have to disassemble everything and start over. You can guess how I know this. First, pick one of the two smaller circular plates with the small holes in their center. Into one of these, you will attach one of the servo couplers, the aluminum disks that attach to the servos. To the other one you will bolt one of the "universal servo brackets", the U- shaped arm part that has two small arms and one large arm, along with a large number of holes. This will form the shoulder elevate joint later. You have to bolt these on first because you can't do it later:

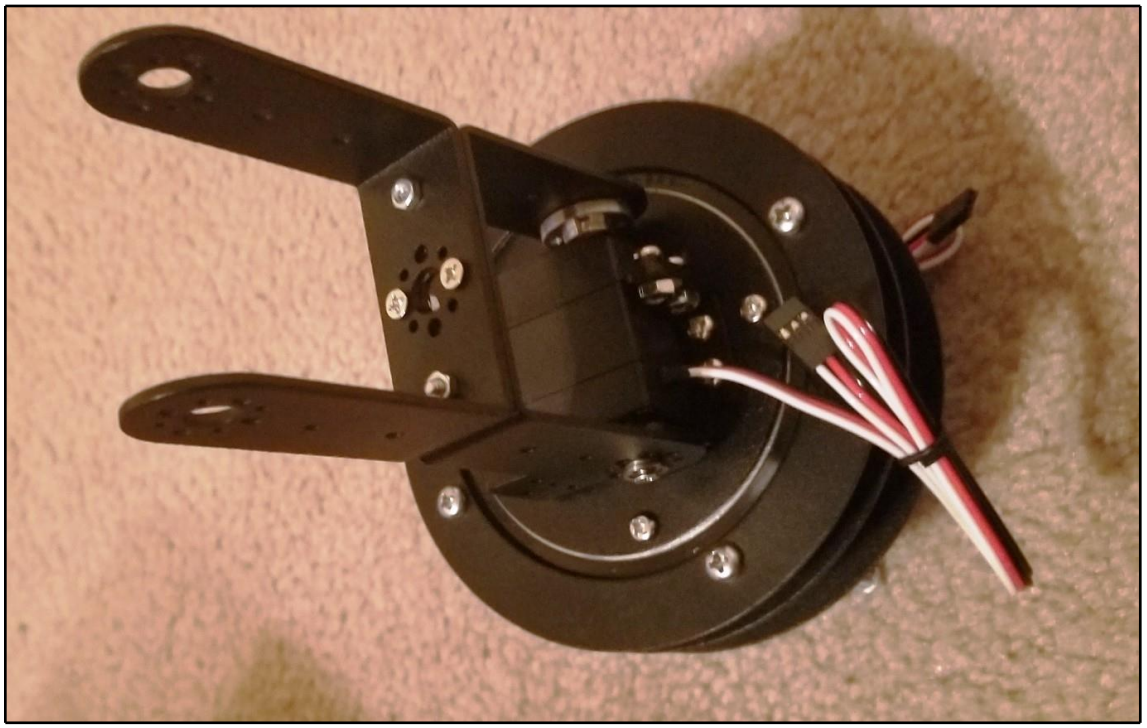
Now we sandwich the large bearing between the two small circular plates and attach them with four long screws and nuts. The servo attaches to the bottom (silver disk) and the arm servo bracket is on the top. The screws must go down so that we don't have anything poking up to interfere with the arm. After taking a moment to admire our work, we now attach to two larger circles with the big cutouts. These are secured with four of the very long screws. Next, we grab the large circle with the servo-shaped hole in the middle, and install one of our servos in that rectangular spot with four screws and nuts. Then, we need to secure the bottom of the larger turntable assembly by putting four of the brass spacers all the way down on the very long screws until they touch the bottom of the metal plate. Now, mate the servo's spline output to the silver disk-shaped servo adapter you placed on the bottom of the turntable. We can now add four nuts to the long screws to secure the bottom plate. Be sure to get them all even and do not tilt the plate with the servo to one side or the other. Our turntable is done, and you can mount it to the robot mobile base if you wish, or keep the arm assembly all together and mount the turntable later after you have assembled the entire arm:



Arm assembly

The robot arm is made out of what is generally called the **servo construction kit** components. You will note that there are several standard parts used over and over. There is what I call the **universal servo bracket**, which is the U-shaped bracket with all of the holes. Most of the arm servos will fit into these brackets. There are three C-shaped brackets with rounded corners. We have one right angle bracket, one robot hand (which is pretty obvious), the five remaining servo motors, and a plastic bag with the servo couplers, which are aluminum disks with four holes around the outside and one hole in the middle. We also have three bearings in that same bag.

Our first step in the arm assembly is to take two of the large C brackets and fasten them back to back to make one long bracket with the curved sections on either end. I used four screws for this, but there are six holes. I picked the two outer and inner holes:

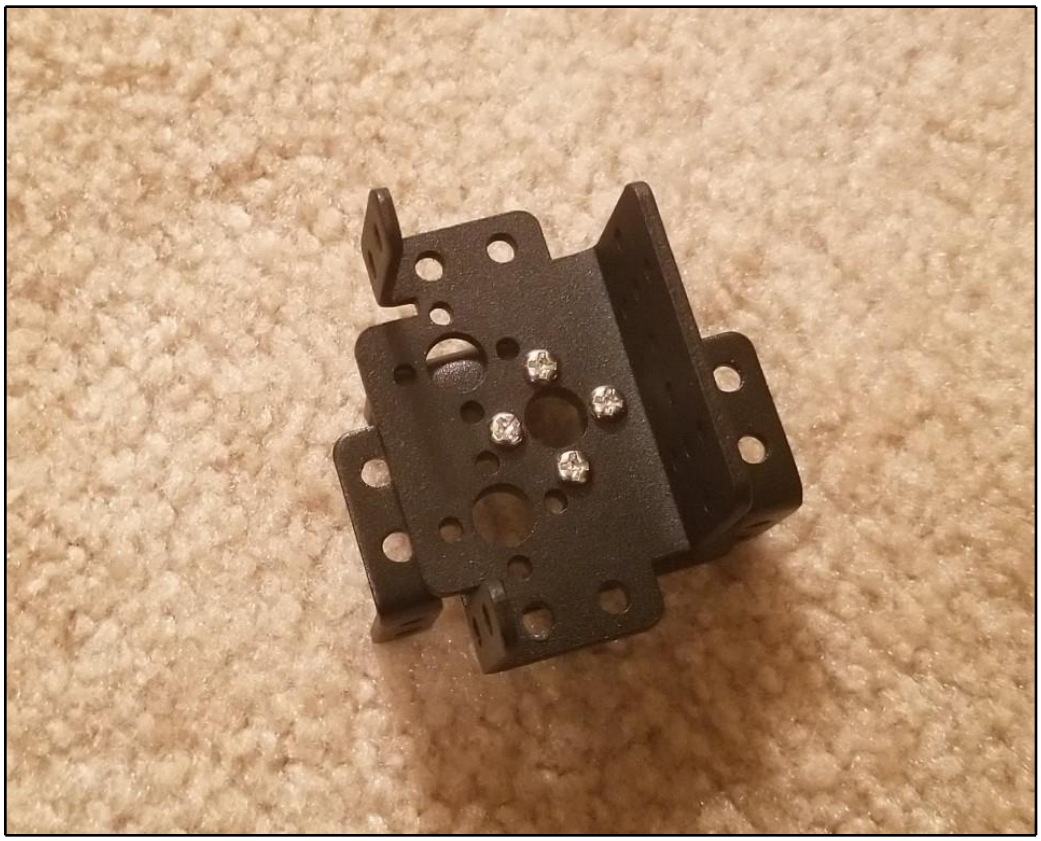


Now, all of the servos that attach to the C-shaped arms will go together the same way. We grab one of the small bearings and a short screw and fasten the bearing into the side of the rounded part of the C bracket away from the side of the servo. Now attach the

bearing and the screw to the universal servo bracket (which I will just call a *US bracket* from now on) and fasten with a nut. If you think about how the servo goes in, you will see which side to attach the bracket. You now install the servo in the US bracket with four screws. Now, you have two pivot points on either side of the servo. Install the servo coupler (silver disk) to the servo and attach the servo coupler to the C bracket to complete the shoulder elevate joint.

Now for the elbow joint: we take one of the loose US brackets and also get the L-shaped bracket. We want to fasten these two together at right angles which is how the servo goes in. The image should explain how this works. Now, we repeat the technique we used to assemble the shoulder joint – we mount the bearing in the other end of the long C bracket, attach the universal bracket to it with a nut, and then install the servo. That completes the elbow, so now we work our way to the wrist. We attach the other C bracket to the end of the L bracket that we just attached to the elbow joint:

This next bit is a bit tricky. We need to fasten two of the US brackets together in the middle and at right angles to one another. We will be making a wrist joint that can both tilt and rotate, and that takes two servo motors perpendicular to each other. Once that is done, we can grab our remaining bearing and mount it in the C bracket that we attached to the elbow. Now use a screw and nut to attach one of the universal brackets to the C bracket, just as we did before, and then install the servo and servo coupler to the wrist tilt joint:



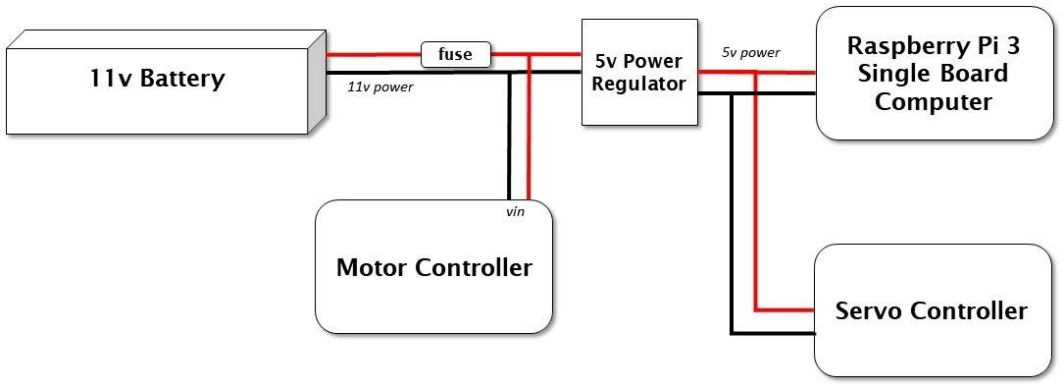
This is as good a time as any to take a break and think through the next steps. Grab the robot hand, and you can directly attach one of your servos. Make sure you first put the servo in the middle, and then incorporate the hand grip into the middle of its travel as well. Then, line things up and install the servo. That was not too bad. Now install the wrist rotate servo to the US bracket on top of the wrist joint. Our final assembly step should be pretty obvious. We put the servo coupler on the wrist with a screw in the center hole, then line up the hand, and put two screws into the matching holes in the hand and the wrist.

Our erector-set construction is complete and we have the mechanical form of the robot.

Take some time now and tidy up all of the servo cables. There are four sets of servo extension cables in the kit, so attach these to the four top servos in the arm. I used cable ties to attach the cables to the side of the arm. There is also a spiral cable organizer – at least I think that is what it is called – in the kit. You can use this to also clean up the arm cables, and it makes everything all the same color.

Wiring

The power wiring diagram is included by way of illustration. We have four main electronic components: the Raspberry Pi 3, our robot's brain; the Arduino Mega; the motor shield; and the servo controller. We will be needing two sets of power – the Pi and servo controller need 5v, while the motor controller needs the full 11 volts from the battery pack. I purchased a 5v power supply to convert the battery to 11v. The motor controller needs power on the screw terminals labeled "EXT_PWR". Hence, we need to create a power harness with two splits – one split that puts 11v into the power supply board and the motor controller. We can then wire 5v from the power supply board to the Pi 3. A second 5v goes to the "servo power" connection on the servo controller board. This is the two-pin connector that is aligned with the six servo three-pin connectors, as can be seen in the following diagram:



The Pi 3 has sufficient power on its USB interface to power the Arduino (that has almost no load on it) and on the servo controller logic circuit, which also takes very little power. We run USB cables from Pi 3 to Arduino, and from Pi 3 to the servo controller. Later, we will plug a USB camera into the Pi 3 as well. If we need to, we can run a separate 5 v to the Arduino at a later date if we are having power problems. The USB lines also take care of control signals to the Arduino and the servo controller. The motor shield is plugged directly onto the top of the Arduino Mega and needs no further connections.

My plan of attack was to put together the battery and power converter and test them for the proper voltages and polarity using a voltmeter before connecting any of the sensitive components. I also loaded the drivers and control software to the Arduino and

Raspberry Pi 3 before plugging them into the battery power supply and connecting with USB cables.

All of the servo cables can be installed in the servo controller, being careful to put the black wires, which are the ground wires (or negative wires), all on the same side. They go to the outside of the servo controller. You will want to double check that all of the black wires on the servos are lined up on the "-" or negative pins.

I've provided connections for teleoperations using a PlayStation/Xbox-type joystick, which is useful for establishing that all motors are running the right way. We'll cover calibration in the software section. If one or other of the drive motors are running backward, you just have to switch the motor wires.