Table of Contents:

Preface:

My Silicon Valley Adventure

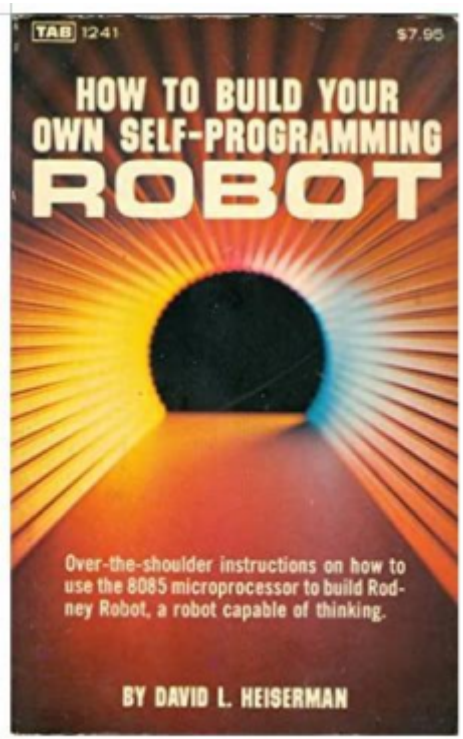
I want to take you on a journey forty years in the making. I graduated two years late from Florida State University in 1982 with a degree in marketing. If I had graduated on time, I would have worked for the local newspaper in advertising sales, and my life would be very different.

Camp Peavy and Loco Jo.

It's funny how little things early in life make a big difference. As it turned out, my last college project was a report and presentation on Apple Computer, which was coming out with the Apple III and going public. That looks interesting, I thought. I want to get involved. So I got a job at the local Radio Shack Computer Center in Tallahassee, Florida, and thus began my adventure.

The Radio Shack Computer Center had classes in BASIC programming, word processing, spreadsheets, databases, and general computer operations. I found that instead of paying for

school, they were paying me! At this point, I decided to take the microcomputer product far into the future.



Heiserman's "How to Build Your Own Self-Programming Robot."

Like everyone at the time, I was obsessed with the movie "Star Wars" and especially loved the robot characters R2D2 and C3PO. I realized mobile robots were microcomputers with eyes, arms, and legs. I soon discovered a book by David L. Heiserman entitled "How to Build Your Own Self-Programming Robot" (TAB BOOKS No. 1241). It changed my life.

I hope this book does the same for some young or young-at-heart robot builders. In Heiserman's book, he explains how to build an 8085 microprocessor-controlled device named Rodney, which featured "machine intelligence," a theory of reinforced learning relevant to this day.

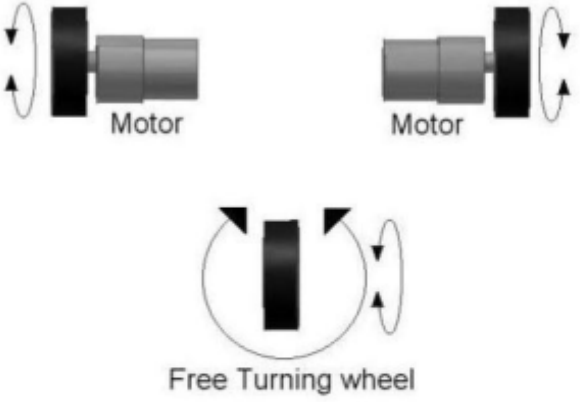
Rodney's primary mode of expression was movement. It had a differential drive system balanced by two caster wheels. In a differential drive system, there are nine possible motion patterns: forward, reverse, clockwise, counterclockwise, forward left, forward right, reverse left, reverse right, and stop (yes, "stop" is a motion pattern). Heiserman enumerated three levels of machine

intelligence (Alpha, Beta, and Gamma) based on these motion patterns.

The third project in this book is a recreation of the Rodney experiment with a popular microcontroller board called the Arduino.

The Big Sale

In 1984, at 26, I made a hefty $100,000 sale of TRS-80 (Tandy Radio Shack) computers to the local community college. I got a $4,000 bonus check, so I packed my bags, my cat (Shalimar), and Rodney (my robot) and headed to Silicon Valley, seeking great wisdom and adventure!



Differential drive is where two separately driven wheels propel and steer the robot; generally, casters are added for balance.

When I arrived in Silicon Valley, I got a job with an electronics distributor called "Zack Electronics" in Palo Alto. After two years of selling microcomputers at the Radio Shack Computer Center in Tallahassee and building robots as a hobby, I expected EVERYTHING in Silicon Valley to be fully computerized and automated . . . it wasn't.

Zack Electronics was a place out of time. Established in 1931 as "Zack Radio Supply," they had an old 1960s-era check posting machine that I swear was made of cast iron; they kept their inventory on a manual "flip file" organizer. You could even test your vacuum tubes! All while selling electronics and test equipment to Apple, Digital Equipment, Hewlett Packard, Stanford University, and NASA Ames . . . Silicon Valley's elite!

Zack won an IBM PC in an RCA promotional sales contest, and no one knew what to do with it. So I took a pirated copy of dBase II and created their first computerized inventory control system. One day old man Zack (Victor Zacharia), strolled through the store and said, "I don't know what you're doing, young man, but keep on doing it." After leaving Zack's, they eventually bought an IBM minicomputer and hired a full-time programmer.

I had plenty of wonderful experiences in Silicon Valley, including selling a Motorola "Brick" cell phone to "The Woz" (Apple co-founder Steve Wozniak) when working at Quement Electronics (a friendly competitor to Zack's) in San Jose. One day Woz was strolling through the computer/phone department (which I managed), and the department phone rang. Woz picks up the phone and says, "Quement Electronics, how can I help you?" - "Yeah, we have plenty of those! Come on down!" I have no idea what the customer was asking for... but an actual example of Woz, the prankster.

I also sold Steve Jobs a Macintosh. No, really, I did. This was when he had left Apple and was running NeXT Computer. I was selling Macintoshes at the Computer Attic, an Apple retailer on University Avenue in Palo Alto, just down from Stanford University. Jobs would periodically roller-blade in downtown Palo Alto and stroll through the store. It wasn't a big deal, as we were the only NeXT retailer in the country. Jobs needed a computer for his daughter, Lisa, before she went to college, so I sold him a Macintosh Portable . . . must've been $5,000. It was in the days of

the "zip-zap" credit card machine using carbon-paper tickets. I will never forget I had to call in for approval . . . the lady on the phone asked, "Did you look at his ID?" I replied confidently, "It's him."

For a time, I was Scott Cook's personal technician. Scott Cook founded Intuit (makers of Quicken and Turbo Tax). While at Intuit, I shook hands with Bill Gates when Microsoft attempted to acquire Intuit. Cook introduced my boss and me as "the guys who keep the company running." I hate to tell stories out of school, but what struck me was how wimpy Gate's handshake was.

Burningman '99 with my trusty robot sidekick "Springy Thingy."

This one time, when businesses were transitioning from overhead transparencies to PowerPoint, I got to fly on a private jet. Cook was scheduled to do a QuickBooks presentation in Mtn. View, San Diego, and Tucson all in one day! My job was plugging the computer into the data projector and ensuring it worked. This was in the early days of PowerPoint. <sarcasm> No pressure, in front of a large group of people at an important presentation,

ensuring the computer synchronized with the data projector. No pressure at all. </sarcasm>

The HomeBrew Robotics Club

A big part of my Silicon Valley life has been the HomeBrew Robotics Club of Silicon Valley (HBRC) www.hbrobotics.org. The HomeBrew Robotics Club is a branch off of the original HomeBrew **Computer** Club of Apple fame.

When the HomeBrew Computer Club disbanded in 1984 because you no longer had to build your own computer, a Special Interest Group (SIG) within the HomeBrew Computer Club interested in robotics formed a new club, the HomeBrew Robotics Club, which is active to this day.

In 2003, some of us in the HomeBrew Robotics Club were lamenting the club's lack of building activity. We knew members had built several robots over the years; most were demo'ed once and relegated to a shelf. We observed that most building activities coincided

The table, the robot, the block, and the box. All the components of the TABLEBot Challenge.

with contests. We recently had a line-following contest and a maze-busting contest, but these were contests and involved stages that had to be set up and stored. Plus, it had "winners" and "losers" . . . someone was always unhappy. The tape was shinier than I used at home, or I expected the maze gates to be wider, etc.

We wanted to create a new contest that didn't have an elaborate setup. At the time, we met at the Castro Middle School library in Campbell, Ca. There were tables all over. We thought, let's do tabletop soccer! We could set up nets to catch the robots if they fell. But no, that would be set up. And that's when it was born: the TABLEBot!

A TABLEBot is a robot that survives, lives, and plays on a table or pays the price. But how do you get to tabletop soccer? A non-trivial task. . . It was decided we would get there in "Phases," so we made three phases and called it the "TABLEBot Challenge," specifically pointing out this was not a contest but a Show or a "Challenge."

The three phases are as follows:

- Phase I: Have the robot drive from one end of the table to the other and back.
- Phase II: Push a block off the edge of the table.
- Phase III: Push the block into a shoebox mounted at the end of the table.

We also did not specify the size, color, or anything about the block, the table, or the robot, so folks could use the robots they had sitting on their shelves and whatever table, block, and goal they wanted. After the first year, we renamed the "TABLEBot Challenge" the "HBRC Challenge" and expanded it with other three-phase events (FloorBot, LineBot, ArmBot, etc.) https://www.hbrobotics.org/index.php/challenges/.

We've held these Challenges, all Show-and-Tell meetings, for over twenty years. The TABLEbot will be featured as the second project in this book.

Robots at Burning Man

I even brought my robots to Burning Man from 1999 to 2005 with the goal of leaving the event with a functioning robot. Burning Man is an extreme art festival in Nevada's Black Rock Desert.

The Black Rock Desert is a playa. Ten thousand years ago, it was a giant salt lake that has since dried up. Consequently, it is the flattest, most barren, God-forsaken land you've ever seen. Perfect for wheeled vehicles of all kinds!

The unique thing about Burning Man is there is no vending; as mentioned, it is an extreme art event. You bring all the food, water, and shelter you need to survive and whatever art you do. While your money is good to buy supplies, RVs, toys, food, etcetera, once you get to Black Rock City (Burning Man), your money is no good. **There is no vending at Burning Man!** My art is robots, so we (my wife and a friend) went and ran robots the whole week, charging batteries with solar panels. And if that wasn't enough, we'd crank up the generator. I went again in 2018, when the theme was "I, Robot," because I did not want to be asked why I didn't go, but let's not get ahead of the story.

RoboGames

Another big part of my Silicon Valley life was RoboGames http://robogames.net/index.php from 2005–2018, where I won three Bronze medals and one Silver. The silver medal was for my Burning Man ArtBot "Springy Thingy." However, I'm most proud of my Bronze medal from 2012 for my RoboMagellan robot, "Blue Dog." RoboMagellan is a contest where you're given a GPS coordinate where a 19" OSHA orange cone is placed, and your robot must traverse approximately a hundred yards and touch the goal orange cone.

RoboGames RoboMagellan 2018. Silver, Gold, and Bronze medal ceremony.

There are bonus cones along the way that you can elect to touch, they give you a fractional multiplier that reduces your overall game time, but if you do not touch the goal cone, your score is your distance from the goal cone. In 2012 I touched the goal cone without even using a camera to guide me to the cone. I

will discuss how to build a RoboMagellan robot in the fourth project chapter of this book.

The PROTOBot

In 2005, I got laid off from my nice cushy network administrator job at Intuit, and at age forty-seven, it was time to decide what I wanted to do with the rest of my life. Fortunately, I had been given a good severance package for having been at Intuit for over twelve years. Most folks would have gone back to being a PC technician or network administrator; those jobs were plentiful and paid well, but I came to Silicon Valley to build robots! So I put together a kit called the PROTOBot based on the Parallax BASIC Stamp, wrote an article for SERVO magazine (a magazine for robot enthusiasts), and started my own business: HomeBrewed Robots!

The PROTOBot on the cover of SERVO Magazine.

I sold a few dozen kits before realizing I wouldn't make a living doing this. I eventually went to work for Jameco Electronics as an electronics technician (as opposed to PC tech). Jameco had just acquired "www.robotstore.com" the previous year, so I was the "robot guy" at Jameco. I even went back to school (College of San Mateo) and studied electronics, earning a diploma in

soldering and electronic assembly. That was an eye-opener! After reading many books on electronics, let me say there is nothing like interacting with a good professor to help one understand a subject.

Eventually, I made my way into a stealth startup called "Home Robots, Inc," located right across from my house (no kidding). I was employee #16. When hired, Linda Pouliot (one of the founders) asked if I had any questions. I asked if I could see a robot run. The engineers scurried to the four corners of the building. There was only one functioning robot, and Vlad (the software engineer) was working on it, so it could not be demo'ed. That's okay, I thought . . . but for the record, there were no functioning robots when I started at Home Robots, Inc. Eventually, "Home Robots, Inc" became "Neato Robotics," which has sold millions of intelligent robot vacuum cleaners worldwide. https://neatorobotics.com/.

The secret to the Neato robot vacuum cleaner is its "lidar" or laser radar, making it a perfect candidate for learning mapping and navigation with ROS (Robot Operating System). The last two projects in this book utilize the Neato robot and ROS.

Let's Build Robots!

I've built hundreds of robots using a variety of computers and microcontrollers. This book will take you from the very beginning of programming an Arduino, building a TABLEBot, machine intelligence, navigating outdoors with a GPS-guided robot, and mapping and navigating using ROS, the current standard framework for robotics.

This book is not for everybody, but it could be for anybody. You see, I am not a formally trained engineer. Much of my knowledge is learned through experimentation. So think of this as

a "right-brain" approach; more arts-and-crafts than engineering, a lot of Velcro, 5-minute epoxy, Shoe Goo, and heuristics. While some methods might have a better look or a more finished appeal, the homebrewed approach is more fun and personal. I want YOU to build YOUR own one-of-a-kind robot. Some parts of this book are advanced and require knowledge of hardware and software or a desire to explore and learn. When you have questions, ask Google.

What makes me qualified to write this book? Well, you write what you know, and I've known a lot of homebrewed robots and robot-builders. I've published several articles in SERVO magazine, and this book represents the culmination of my efforts.

Mobile robotics will be the most significant technological revolution in history. During the computer revolution, the hardware consisted mainly of the computer itself, keyboard, monitor, and printer. Everything else, for the most part, was software or virtual. Robotics is physical, plus electronics, and lastly, software. The Robotics industry will be more significant by several orders of magnitude. Robots come in every shape, size, type, and variety: air, land, and sea. Robotics will eventually dwarf the computer and the automotive industries, if for no other reason than the computer and automotive industries will become the robotics industry. So welcome aboard, and let's enjoy the journey together.

I do not guarantee the accuracy, completeness, and timeliness of the content provided. The use of the contents of this book are done at the user's own risk. In other words, I am not responsible for what you do. Let me know if you have problems, questions, or comments: camp@camppeavy.com

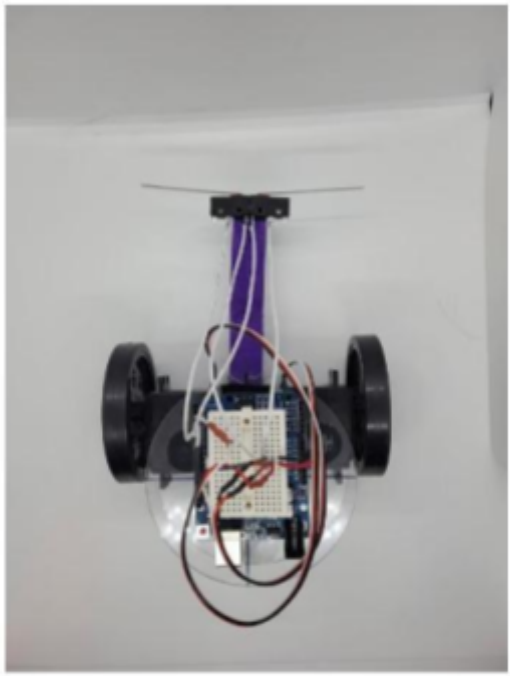
And finally, thanks! Thanks to Michael Ferguson, Alan Federman, Ralph Gnauck, Ross Lunan, James Nugent, Michael Wimble, Marco Walther, Tony Pratkanis, Brian Higgins and Rohan Agrawal. Thanks to my colleagues at Neato and all the folks with

the HomeBrew Robotics Club, including but not limited to Al Margolis, Wayne Gramlich, Bill Benson (deceased), Dan Tuchler, ‘dillo, and Chris Mayer. But most of all, thanks to Dr. Ed Katz for working with me for over a year to complete this work. His numerous suggestions, ideas, and encouragement were invaluable.

Chapter 1:

The PROTOBot

Before we start, I recommend purchasing a soldering iron, solder, safety glasses, electrical tape, and a multimeter. While you're at it, get a battery tester; it might save you hours down the road. You're not going as a robot-builder without some tools and materials.



The fully assembled PROTOBot with servos, wheels, Arduino, solderless breadboard, and popsicle stick snap-action switch bumper mount.

ALWAYS wear safety glasses when soldering! All you need is one slip of the wire to flick hot solder into your eye and ruin your vision. Again, never solder without eye protection! You have been warned. While we're on the subject, try some solder wick and flux. Google it! Soldering is good for the soul.

You'll also want some heat-shrink tubing and the prerequisite heat gun, but short of that, have electrical tape to cover the exposed wire. In the Tools section at the end of the

book, I mention the “Weller WSTA6 Pyropen,” which is cool because it is not only a portable soldering iron, but you can change the tip and make it a hot air tool.

Remember to use what you have! Don't let the struggle for perfection destroy good enough. As mentioned, you'll want a multimeter. It doesn't need to be a fancy high-precision meter, but I recommend something with an audible continuity check and auto-ranging if you can afford it.

Along the way, I will mention various building materials; Instamorph, Sugru, 5-minute epoxy, E6000, hot glue, etc. I encourage you to try different things. Experiment! Find what works for you; find your own way, but most of all, have fun! MacGuyver solutions! This is not a science; it is an art, and you are the artist! You will also hear words and terms with which you will not be familiar; again, Google them. Think of this as a journey. A parts list will be at the end of the book, but don't let what you do not have stop you from starting. Time is the valuable thing!

"This is a football"

On the first day of training camp, Vince Lombardi famously uttered these words to the 1961 Green Bay Packers. The point was to start with a clean slate and focus on the fundamentals. The other Lombardi

The Arduino is the most popular microcontroller due to its easy-to-use and interactive IDE. It has a large following and many projects, kits, and applications developed around it.

quote is, “It’s not whether you get knocked down; it’s whether you get up.” Robot building is hard. Never, never, never give up. I want to start with a clean slate and focus on the fundamentals taking you from the beginning of using the Arduino to control a robot's movements, if you fall, get back up! Settle for nothing less than total world domination!

Our simple PROTOBot will move with hobby servos. A servo is a small motor with a controller initially designed for radio-controlled airplanes. Initially, we will read input from a snap-action switch and, in the next chapter, an ultrasonic (sound) sensor. We'll eventually have a robot moving and staying on a table, finding a block, and putting that block into a shoebox or goal mounted on the table.

This is the starting place for a functioning mobile robot, which will move (output) and react (input) to its environment. This will be elementary if you're already familiar with the Arduino, but I have included building and prototyping tips learned through the years that you might find helpful and fun.

The hobby servo includes motor, gearbox, and controller board. There are 3 wires: Power (red and black) and signal (white). Apply 5 volts power to the power wires pulse the signal 1ms to one direction and 2ms to go the other. 1.5ms to stop.

- First, we will use the Arduino to move a servo motor.
- Then, we will initially read input from a sensor, a switch.
- Finally, we will put the movement and sensing together to react.

Be advised; these are not meant to be literal step-by-step instructions about building a particular robot (however, you could use them that way) but as a guide to building your own personal homebrewed robot, a robot-building lifestyle if you will.

The techniques explored here will help you learn quick prototyping skills and techniques. As mentioned, don't let the struggle for perfection destroy good enough; perfect is the enemy of good enough. Use what you have; don't complain about what you do not have! Don't wait for just the right screw, fastener, or building material. Tape it, glue it! Make it work first, and then make it better. Iterate early and often! Time is the valuable thing! Use whatever you have: popsicle sticks, old toys, cardboard boxes, etc. And lighten up, and have some fun!

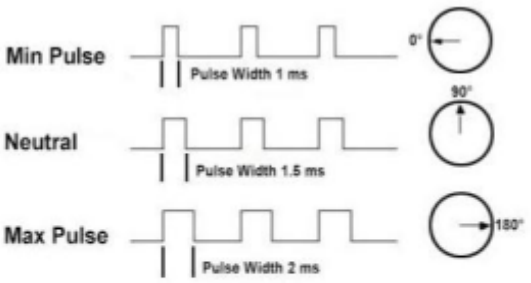
If you have trouble starting, I'd recommend entering a contest or just start telling people you're building a robot. There is nothing like a deadline or people asking how it's going to make something happen. In fact, for the most part, nothing happens without a deadline or some pressure to get it done. If there are no local robot clubs, join one on Facebook or start one. I've heard it said and find it true; the best way to learn is to teach; there's always someone who knows less about this than you. Also, many robot clubs now meet online, including the HomeBrew Robotics Club (www.hbrobotics.org). That having been said, let's get started.

Wire Wiggling

Everyone intuitively understands traditional computer programming for alpha-numerics. For example, to create an address list, you would type in code that requests input from a user (first name, last name, address, city, state, zip). Then, you would sort the data alphabetically by last name and finally print out

your report or mailing list depending on what you wanted as a final result. While you might not know the exact commands or syntax (the way commands are put together), generally, folks have some idea of how alpha-numerics might be used to yield this kind of result. But how does one program a computer to actuate a motor or read a sensor?

To write a program that moves a motor, you need to "wiggle" a wire. To write a program that reads a sensor, you need to "look" at a wire to see whether or not it is "wiggling." Now you don't actually wiggle a wire; you pulse an electronic signal through a wire to actuate the motor or "listen" to a wire to see whether or not it is pulsing electronically.



Standard servos go back-and-forth from 0° to 180° while continuous rotation servos rotate 360° like a drive wheel.

There are basically two ways to actuate motors with the Arduino: H-Bridges and RC servos. I like to use RC (radio control) hobby servos as motors in my robots because they are inexpensive and relatively easy to use. Plus, you can scale up your robot as we do in the RoboMagellan chapter by using Electronic Speed Controls instead of the servo to drive large DC motors.

Hobby servos were originally developed for RC planes to actuate flaps and rudders to control the plane, so traditionally, they actuate between 0° and 180°. With the popularity of tabletop

robotics, it has only been recently that you have been able to get servos that function as gearmotors; that is, they rotate continuously a full 360°. These are known as "continuous rotation" servos. Servos that move back and forth from 0° to 180° are known as "standard" servos.

RC servos have three wires. Two for power, red and black (+ and -), and one for the signal (usually yellow or white). Pulse the signal wire 1ms the servo will move in one direction, pulse it 2ms, and the motor drives in the other direction; a 1.5ms pulse will make it stop.

Arduino Programming

Install the Arduino software onto your computer, either laptop or desktop. The Arduino software is called an "IDE" or "Integrated Development Environment" because it includes everything one needs to create software. An IDE consists of an editor, a compiler, and a debugger. Without the "IDE," you would have to use three different tools. Hence the "integrated" element. You'll essentially use the IDE to create and make the code so the Arduino

Arduino initial screen including setup and loop functions.

understands it. That is, the code is compiled. It is then downloaded to the Arduino, where it will run every time you power up the Arduino.

The "Arduino" is arguably the most popular microcontroller in the world because it's easy to use and interactive. A microcontroller is a single-board computer generally used to actuate motors and monitor sensors. It usually has no monitor or keyboard, so a computer is used to create the program, which is then downloaded into the Arduino microcontroller.

The Arduino has a large following and many projects, kits, and applications. Its programming language is based on Wiring, which is a "C"-like language popular in professional programming (a lot of semicolons and curly brackets (; and {} 's)). This C-like program is compiled into Arduino code. Arduino programs are called "sketches."

Plug the 22 AWG solid wire into the .1" spaced sockets on the Arduino. 5V and GND is a power source. D13-D0 are digital I/O ports.

You can find the Arduino IDE here: https://www.arduino.cc/en/Main/Software. Download the program for whatever operating system you are using. I've mentioned this before, but it bears repeating; any questions about the Arduino or

anything else, ask Google. Type the question as if you were asking another person. When you get an error message, literally type the error message into Google. If Google doesn't give you the answer, it will generally provide links to get the information you seek.

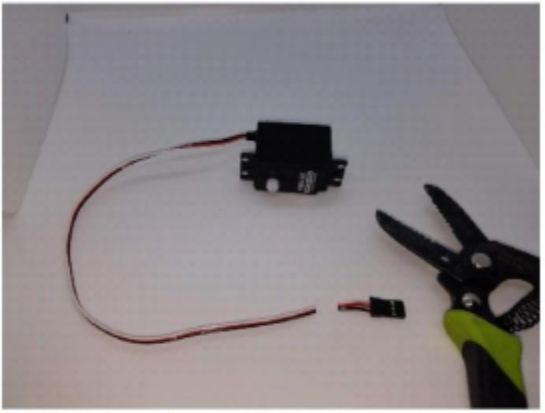
After downloading . . . launch the Arduino program.

How to Move a Servo

Apply five volts to power the servo (red and black wires) and pulse the signal wire with one of the digital pins (D0-D13) on the Arduino or whatever microcontroller you use. Don't know how to pulse a wire? Don't worry, I'll explain. On the Arduino, attach the red wire to the five-volt (5V) power socket; connect the black wire to the ground socket (GND). The signal line is pulsed by the microcontroller, one of the digital header sockets; D0-D13. There are various versions of the Arduino; in this case, we are using the Arduino Uno, but they all operate similarly.

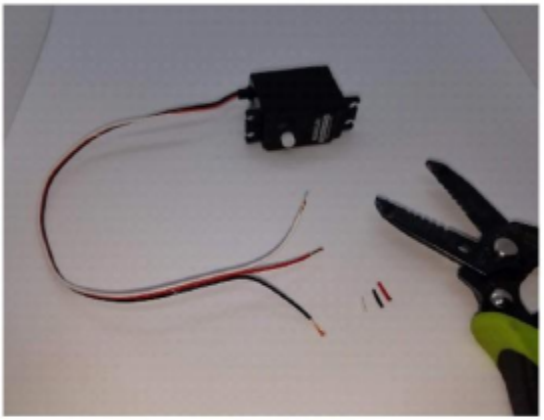
If you pulse the voltage on a signal wire (D2, for example) at 1ms (one millisecond or one thousand times a second), the servo goes in one direction, and at 2ms (two milliseconds or two thousand times a second), it goes the other direction... at 1.5ms, the servo stops. In addition to being inexpensive and easy to use, I like to use servos because if you learn how to actuate a motor with a servo, you can replace the servo with an "ESC" (Electronic Speed Controller), which would allow you to control large DC motors as if they were servos and build bigger robots. Theoretically, you could use the same software developed on your tabletop robot on a large full-size robot, as we do in the RoboMagellan chapter.

The Homebrewed Plug



The first thing to do is cut off the connector. Be careful not to cut too short.

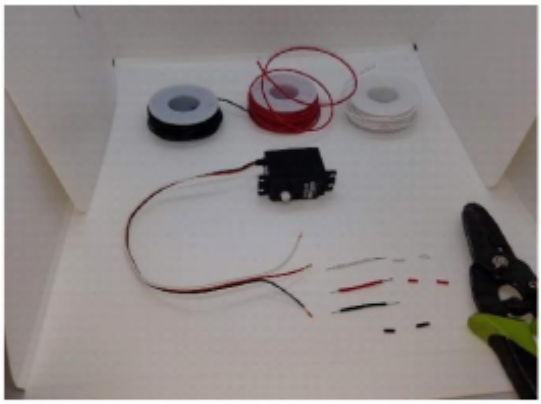
Now let's get that servo motor running. As mentioned, the RC or hobby servo has three wires: power (red and black) and signal (white or yellow usually). The power plugs into "5V" and "GND." The red wire into 5v and the black wire into GND. Finally, plug the signal wire into one of the digital ports (D0-D13). Servos come with a 3-position .1" connector designed to plug into an RC receiver (.1" male header). This is fine for an RC (Radio Control) receiver, but the problem is with the Arduino; the "5v, GND, and signal are not placed together. One of my favorite techniques is to cut off the connector and replace each wire end with a two-inch length of 22 AWG (American Wire Gauge) solid core wire (NOT STRANDED!). You're making plugs for each wire (plugs are male; sockets are female). If possible, try to use red, black, and yellow (or at least a



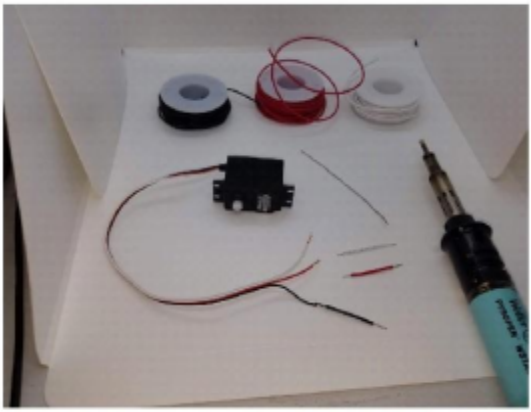
Next spread the wires and strip off ~.25 inches from each wire.

third color) for the three wires.

Be careful not to cut the connector off too short! Better too long than too short. I shall refer to this technique as the "homebrewed plug." You will see this technique used throughout the book. These solid core



Cut three pieces 22GA solid core wire (black, red and white if possible) wire 1.5 inches and strip ~.25" from each end.



Wrap stranded wire around one end of 22GA solid wire and solder.

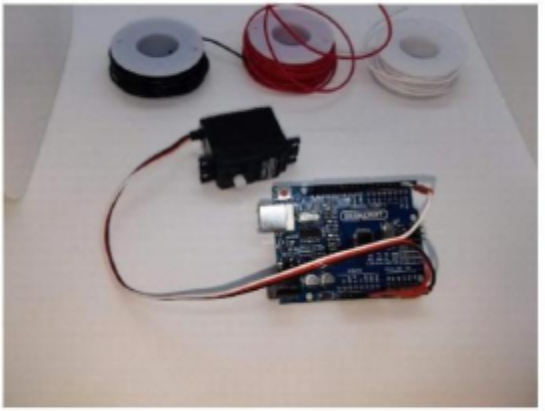
wires plug into sockets on the Arduino and, eventually, a solderless breadboard.

Solderless breadboards are used for prototyping and are known as prototyping boards. Since, for the most part, ALL homebrewed robots are prototypes, I've used them exclusively and with good (not perfect) results over the years. If you take it further, the next step would be to create an actual circuit board in a Gerber file (look it up). Folks hardly make their own boards today (nasty chemicals). They generally send the Gerber file to one of the numerous PCB manufacturers that make them cheaply and in small quantities.

The Servo Library

Finally, wrap and solder the other two wires and use heat-shrink tubing or electrical tape to cover the solder joints.

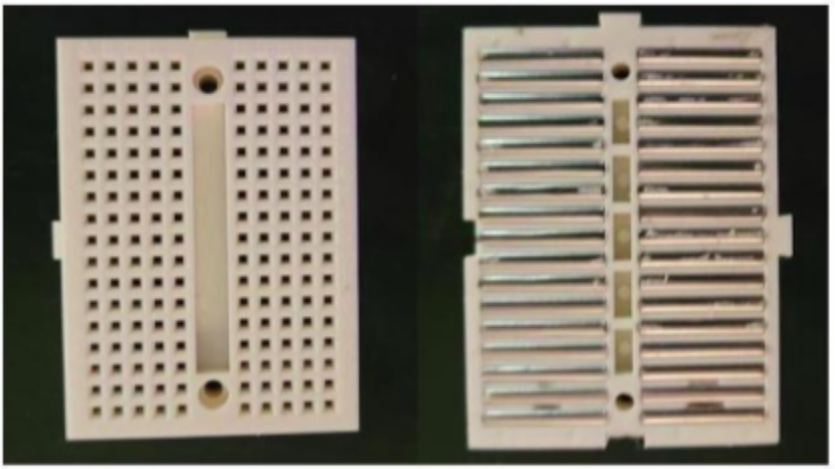
There is more than one way to run a servo on the Arduino. You could do it with the "pulse" command, but it's easier to use the "servo" library https://www.arduino.cc/reference/en/libraries/servo/. A library is a specific bit of code that provides particular functionality. The servo library allows you to control the direction and speed of the servo by saying, "servo.write(0), servo.write(180), or servo.write(90)" to turn the servo clockwise, counterclockwise, or stop. On a continuous rotation servo, this will set the speed of the servo (with 0 being full speed in one direction, 180 being full speed in the other, and a value near 90 being no movement).



This will make 3 Homebrew plugs that plug into the power (5V and GND) and signal (D2) sockets on the Arduino microcontroller.

To load the servo library, go to "sketch" in the menu bar and scroll down to "include library," and select "servo" on the right

column. Then you will need to declare the servo in the header with a particular name; "Servo servomotor," for example. Next, in the setup subroutine or function, attach the servo to a particular pin. In this case, "servomotor.attach(2);".



This is the top of the solderless breadboard. Use 22 gauge solid core wire to jump or make connections.

This is the bottom of the board with the insulation sticker removed. Notice the common electronic connections (metal) between the 5-point ties on each side of the board.

Note: In general, with the Arduino, start with digital pin 2 (D2) and work up, as D0 and D1 are used for communication with the computer when you monitor the Arduino (Serial Monitor).

In the main body or loop section of the code (where the program continuously repeats), say, "servo.write(180);" to make it move in a particular direction (clockwise or counterclockwise, depending on how you look at it). Remember to end your commands with a semicolon!

Go to the Tool menu and select the Arduino you are using (there are various types); in this case, I am using the Arduino Uno and port (for Windows, this will be a "COM" port; for Linux, it will

be "ttyUSB0" or "ttyACM0"). Press the right arrow or Upload button, and your code should compile and download. If you've connected it correctly, the servo should start to turn.

Motor Output Code Snippet

```
-------------------- snip ---------------------
#include <Servo.h>              // Load "Servo" library
Servo servomotor;               // Define servo name
void setup() {
 servomotor.attach(2);          // Set servomotor to Pin 2
}
void loop() {
servomotor.write(180);          // run the motor. "180" makes the
                                // servo turn one direction
                                // "0" the other direction. "90" makes
                                // the motor stop.
}
-------------------- snip --------------------
```

Now change your code in the loop from “servomotor.write(0);” to “servomotor.write(90);”. Download it, and the servo should stop. Next, change the code to "servomotor.write(180);" download it, and the servo will turn in the opposite direction. Congratulations! Using the Servo library, you can move the servo forward, stop, and reverse; from "0" to "90" to "180," and you can control the speed with the values in between! You are on the way to being a robot builder!

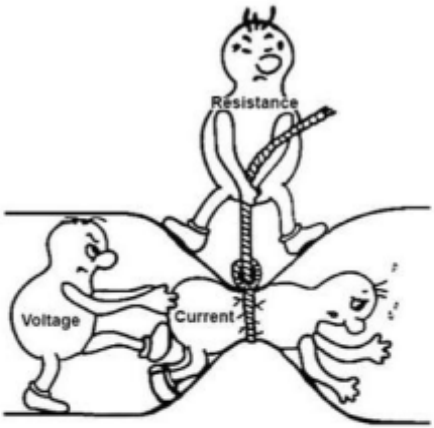
Need INPUT!

Now that we have output (motor running) let's get some input. First, we will learn how to read a switch. There are a variety of switches out there: rocker, slide, toggle, etc., but for the tabletop robot, we like to use snap-action switches because they're useful to detect when our robot bumps into something.

The first thing is to determine the NO (Normally Open) terminals on your snap-action switch. Put your multimeter in "continuity" mode, place your multimeter probes on two terminals, and listen for the "beep." If it beeps when you touch the terminals and stops when you press the switch lever, those are the normally closed (NC) terminals. Once you've discovered which terminals are normally open (beeps when you press the switch), solder a 6-inch length of 22 AWG solid core wire to each terminal and strip .5-inch of insulation off the end of each wire. Notice you are making another set of "homebrewed plugs" for the switch like you did for the servo.

Ohm's Law

Plug one wire into "D2" on the Arduino and the other into one of the GND sockets. Note that all the "GND" sockets are common; they are all connected to each other. You can test that with a continuity check on your multimeter. By the way, you know why it's called a "multimeter," right? Because it measures multiple things like resistance, voltage, and current. If you are new to electronics, I'd encourage you to



Voltage or "potential difference" pushes electrons through a conductor. Resistors "resist" or impede electrons from flowing through a conductor. Electrons moving through a conductor are called “current.” It is measured in Amperes or “Amps.” (image from http://www.sengpielaudio.com/calculator-ohmslaw.htm)

meditate on Ohm's Law, which explains the relationship between voltage, current, and resistance.

There is a hydraulic comparison to electricity that might provide some insight. The water flow is caused by pressure. This water pressure is comparable to voltage. When you turn the faucet on and off, it opens and closes an aperture analogous to resistance. When fully open, there is little resistance. When closed, the resistance is very high. And finally, the water corresponds to the current or the physical electrons flowing through the wire.

Serial Monitor

After connecting the switch, let's do some more coding and give our robot input. Above the setup section of the Arduino sketch, you can include comments, libraries (like we did with the servo), and global variables definitions. For the snap-action switch, we will want to define a constant, "bumper," for example, as the snap-action switch will be used as a bumper "int bumper = 0;". Next, in the setup section, we'll want to turn on the Serial Monitor with "Serial.begin(9600);". The Serial Monitor is one of the coolest features of the Arduino. It allows you to see what is happening with your sensor. Remember the wire-wiggling thing? While witnessing a motor move is obvious, you cannot just look at a sensor and know whether or not it is working. "Serial Monitor" allows us to look at the wire and see whether or not it is moving. In this case, the switch reading will either be a "1" or a "0" (high or low). Serial Monitor lets us look at or monitor the wire and see what it reads.

Arduino Tip: INPUT_PULLUP

Next, in the setup area, we'll set the pin mode as input with a special option called "PULLUP." This initiates an internal pull-up current limiting resistor. Otherwise, a physical resistor would have to be used on the circuit board.

Download the below code (copy and paste) and select "Serial Monitor" under the Tools menu. You should see a stream of "1s." Press the snap-action switch, and the stream on Serial Monitor should change to "0s." Congratulations, you now have input!

Switch Input Code Snippet

```
------------------------------------ snip ------------------------------------
const int bump = 4;                 // Bumper Pin 5
int bumpstate = 0;                  // Set bump State value
void setup() {
        Serial.begin(9600);     // Setup Serial Monitor for monitoring
                                // switch
        pinMode (bump, INPUT_PULLUP);   // Set bump to input
with pullup resistor
}
void loop() {
 bumpstate = digitalRead(4);    // Read bumper state
 Serial.println (bumpstate, DEC);       // Print bumperstate in Serial
                                        // Monitor
}
------------------- snip --------------------
```

```
motor_input_and_output §

// Motor Output Code Snippet

#include <Servo.h>                // Load "Servo" library
Servo servomotor;                 // Define servo name
const int bump = 4;               // Bumper Pin 4
int bumpstate = 0;                // Set Bump State value

void setup() {
  servomotor.attach(2);           // Set servomotor to pin 2
  Serial.begin(9600);             // Setup serial monitor for monitoring switch
  pinMode (bump, INPUT_PULLUP);   // Set bump to input with pullup resistor
}
void loop() {
  bumpstate = digitalRead(bump);     // Read bumper state
  Serial.println (bumpstate, DEC);   // Print bumperstate in Serial Monitor
  if (bumpstate == 0) {
    stop();
  } else {
    servomotor.write(180);
  }
}
void stop() {
  servomotor.write(90);
}

Done compiling.
```

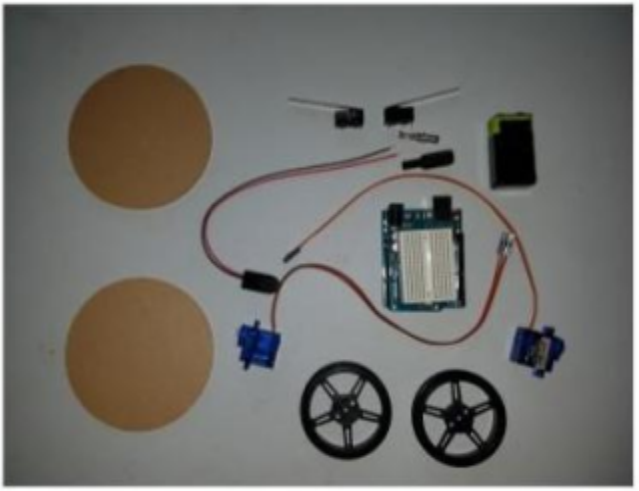
Here we insert an "if" statement so if the switch variable (Bumper) is equal to "0," jump to the "Stop" subroutine and stop the servo (servoleft.write(90);). The program will then go back to the top of the loop. If the switch still reads "0," the motor will remain stopped. When released, the value of "Bumper" goes back to "1," and the servo will start again.

Putting it All Together!

Let's put it together and have the servo change directions when the snap-action switch is pressed.

Notice in the Serial Monitor when the switch is not pressed, the pin reads "1." Then, when you press the switch, Serial Monitor changes to "0." Technically, the Arduino pulls the pin “low” or to ground. Otherwise, it's 5 volts "high" if you check it with a multimeter. Next, we want to create a subroutine that stops the servo when the switch is pressed. To do this, we’ll add the command:

```
if (bumpstate == 0) {
stop();
```



These are the basic parts of our table top robot. The Arduino w/ small solderless breadboard, two micro-servos w/ wheels, 9v battery with snaps and barrel plug, snap-action switches and two 4" acrylic rounds for the body. Your parts might vary but these are the basic components.

This command looks at the variable "bumper" and asks if “Bumper” is the same as "0." It is a comparison **not to be confused with Bumper = 0, which would mean Bumper is equal to 0**. Again, "==" is a comparison operation instead of saying something is equal to something. So IF the variable "Bumper" is the same as "0," go to the subroutine "stop."

Next, we need to create the subroutine "stop," which tells the servo to stop or "servomotor.write(90);". After the program jumps to the stop subroutine, the servo will stop. Notice there is a short 20ms delay for the command to take hold, then the program will go back to "loop." If you let off the switch, the servo will continue to move. Press it again, and it will stop; that is, output, input, and reaction code.

So now you have motion. You can make a motor move by "wiggling" a wire. That is sending a digital pulse down a wire. You can monitor a sensor. That is, read a wire to see if a switch has

been depressed. If the switch has been depressed, stop the motor. There you have it! The essence of robotics: output, input, and reaction.

You can apply this to a mobile robot driving two wheels forward with two bumpers in the front. When the left bumper bumps something, the robot backs up, turns right, and goes forward. When it bumps into something on the right bumper, the robot backs up, turns left, and goes forward.

All you do is run the main loop (void loop()), as the robot senses obstacles, create subroutines telling the robot what to do when it senses something. You can now disconnect the Arduino from the computer and power it from a battery. From here out, when you power the Arduino, it will run that program until you reprogram it.

Output, Input, Reaction Code Snippet

```
---------------------------------- snip ----------------------------------
#include <Servo.h>    // Load "Servo" library
Servo servomotor;     // Define servo name
const int bump = 4;   // Bumper Pin 4
int bumpstate = 0;    // Set Bump State value
void setup() {
        servomotor.attach(2); // Set servomotor to pin 2
        Serial.begin(9600);   // Setup Serial Monitor for monitoring
                              // switch
        pinMode (bump, INPUT_PULLUP); // Set bump to input
with pullup resistor
}
void loop() {
        bumpstate = digitalRead(bump);      // Read bumper state
        Serial.println (bumpstate, DEC);    // Print bumperstate in
                                            // Serial Monitor
        if (bumpstate == 0) {
                stop();
```

```
} else {
        servomotor.write(180);
}
}
void stop() {
        servomotor.write(90);
}
```
---------- snip ----------

Learn the Arduino

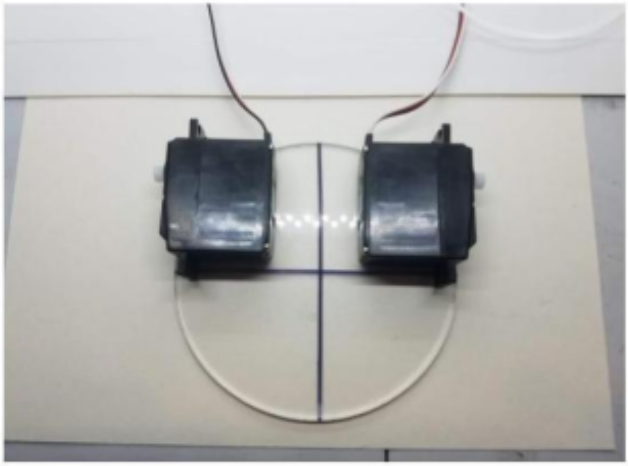
Arduino Tutorial
https://www.tutorialspoint.com/arduino/index.htm

Notice when your computer is plugged into the Arduino, it is powered, but before we can run the Arduino by itself, we will need to provide power from a 9-volt battery. I recommend investing in rechargeable 9-volt batteries to save money in the long run. You do plan on building more robots, right? You will also need a harness to connect the 9-volt battery to the Arduino. You can make one if you have the parts. Otherwise, I'd recommend buying one. You are looking to connect a 9-volt battery to the 2.1mm coaxial socket on the Arduino. If you make your own, be sure and observe polarity! The positive (red) or anode connection goes to the internal tip. The outer ring or sleeve is ground (black). Simply Google or Amazon search "9-volt battery snap" and "2.1mm coaxial power plug" to buy one. You can also put a switch on it or simply plug and unplug the coaxial power plug.

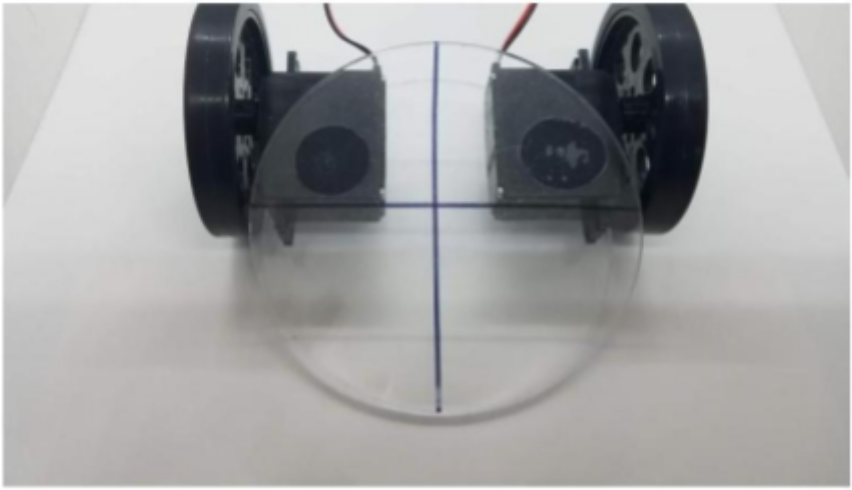
The Arduino board has what is called a voltage regulator on board. This drops the 9-volts, which powers the board overall, to 5 volts (the 5V socket) for your peripherals (servos, sensors, etc.).

Building the PROTOBot

Now you will need to build a decent robot body. I've elected to use two 4-inch rounds for this build (you can buy these through Amazon). You can use anything: cardboard boxes, old toys, and plastic containers. Just find something and start. Save



I use 4-inch round clear acrylic discs for the body. Using a ruler draw two perpendicular lines to help glue the servos with the drive-gears towards the front and and straight.



Screw the wheels on to the servo gears. If you don't have nice servo wheel you can homebrew your own with toy or RC plane wheels, even jar lids. Just glue them to the servo horns that always come with servos.

code as you go along, especially when you reach a breakthrough (gold). I generally increment the name: program_2, program_3, etc. Keep just a version that just reads the sensors. You could use it for troubleshooting.

I will be gluing the servos to one of the 4-inch plastic rounds. Before I do that, I will mark the rounds to center the servos. Use a Sharpie permanent marker and ruler to make a crosshair. I use various gluing techniques, including 5-minute epoxy and E6000 (a sealant popular with the arts and crafts community).

I like 5-minute epoxy because it takes only five minutes to cure. However, it's not that shock absorbent. E6000 is great! It's very shock absorbent, but it takes twenty-four hours to cure. Sometimes I take a dab of 5-minute epoxy and encircle it with E6000. This way, I get the best of both worlds. The connection cures and holds together in five minutes to work with it, and the E6000 cures overnight, so the connection becomes shock absorbent. On this build, I will use hot glue, a quick adhesive. Glue the

InstaMorph is a lightweight thermoplastic which acts like clay when warm, but when it cools, it's a strong plastic.

two servos to one side of the 4" round. Be sure to center it toward the front, and be sure the output gear shaft protrudes beyond the body as you want to attach wheels.

Next, let's attach the wheels to the output shafts. Screw the wheels into the output shaft with the provided Phillips head screws. If you do not have the premade continuous rotation wheels, you can make them by gluing wheels (even jar lids and such) onto the servo horns that come with the servo.

InstaMorph

Then, we need a tail drag to balance the robot. I would like to introduce you to a marvelous building material called "InstaMorph." Technically, it is a thermal polymer. You take beads of this stuff, put it into the water, and microwave it for a few minutes. It becomes this moldable plastic that you can mold into any shape. When the plastic cools, it hardens and keeps that

This is the tail skid made with the Instamorph. Glue it to the bottom rear of the tabletop robot.

shape. Reheat it and reshape it if you don't get the result you want the first time. It's a poor man's 3D printer. If you don't have InstaMorph, bend a wire or find some block that's just the right height for the tail-skid. I have a friend who made a TABLEBot using a Mott's Applesauce cup. It's called the "MottBot." Remember, this is homebrewed robotics, so be creative and have fun. Use what you have around the house!

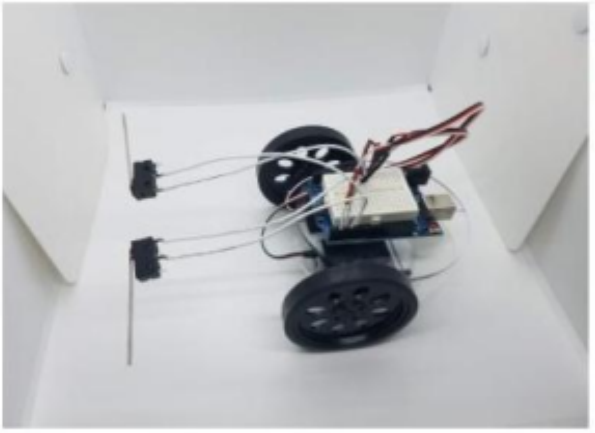
Use a popsicle stick for fishing the InstaMorph out. If you use something plastic, it will stick, and don't use the good silverware because, well, it's the good silverware.

Also, this stuff is HOT when you pull it out, so be careful. Carefully make a tail skid or runner for your tabletop robot. Let it cool, then glue it to the chassis when you have the right shape and height.

- Heating InstaMorph above 150°F increases the risk of burns.
- When warm, InstaMorph may stick to certain materials.
- Keep finished creations away from extreme heat.

Servo-Driven Wheels

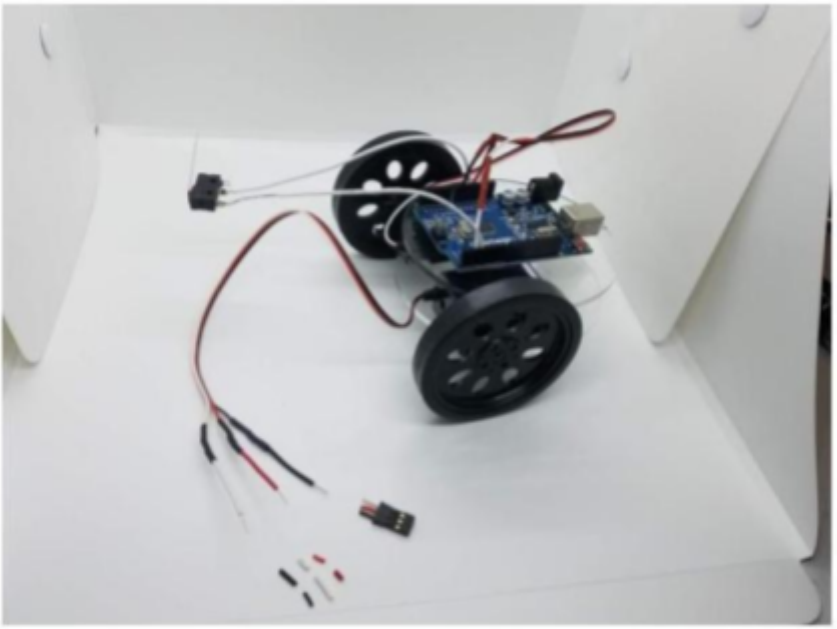
Let's place the Arduino on the body. I simply Velcro it down. There are two parts to Velcro: hook and loop. 3M Velcro on both sides. The loop is soft and fuzzy,



Make a second bump switch like the first.

whereas the hook is prickly and plasticky. I generally put the loop type on the base and the hook on whatever I'm attaching. That way, if I want to put something else on the body or move things around, the part on the body is always loop, and what I'm attaching (microcontroller, battery, sensor, etc.) is always hook.

Alright, we already have one servo and one bumper-switch wired up. Let's do the other servo and bumper-switch. Do like we did with the first servo and cut off the 3-conductor connector and solder 22 AWG solid core wire (red, black, and white if you have it, otherwise, use whatever color(s) you have) to the stranded wires. This makes the homebrewed plug explained above (see photos). You can plug these into the Arduino, except that the Arduino connections are limited. There are only two GND connections and



Create the same homebrew plugs on the second servo. Cut the 3-pin connector (not too short!), cut 22-AWG solid core wires to use as plugs, solder stripped stranded wire to solid core wires, heat shrink or wrap the exposed wire with electrical tape to prevent short circuits.

one 5V connection. This is where the solderless breadboard comes in. One thing about homebrewed robots is that you have to figure things out as you go along, so I've placed a small solderless breadboard in the center of the Arduino Uno. Fortunately, the solderless breadboard comes with double-stick foam tape on the back, so you only need to remove the release liner and stick the board to the Uno or whatever version of the Arduino you are using. Then jumper the GND to one row of sockets and 5V to another. Now you can plug in the grounds from the servos and switches and the 5V power for the servos. Now let's build one more bumper using the snap-action switches.

After building the second snap-action switch bumper, hook it up like the previous one. That is, one connection is connected to GND, and the other is connected to one of the digital connections, in this case, D5. Glue a popsicle stick or whatever mechanical means you have to put the bumper switches in place to actuate when the robot bumps into something.

Programming the PROTOBot

We have all the physical and electronic parts to create a fully functional mobile robot. Open your Arduino IDE, and let's add to the program we've already created. When we last left off, we had a program that ran the servo motor, and when the snap-action switch was pressed, it would stop the motor.

First, let's create some subroutines or functions; these are blocks of code called for by the main routine "loop" for the Arduino that describe the robot’s motions. Our PROTOBot uses what is called "differential drive." A drive and steering system that uses two motorized wheels and is generally balanced by casters or tail skids. You have nine possible motion patterns with a differential drive system, including "stop." Look at the below code, and you should be able to discern the motion pattern with the commands.

Right after you declare the name "servoleft," declare "servoright." Also, attach "servoright" the same way you did "servoleft," except servo right is plugged into D3 instead of D2, so the command would read, "servoright.attach(3);". Before we get ahead of ourselves by running both motors, let's program the other bumper. First, create the integer "rightbumper" and make it "0" the same way you created the "leftbumper." Set the bumper socket D5 to input and use the INPUT_PULLUP option, which allows you to make a bumper without using a physical pull-up resistor.

Now clear out your "loop" routine, as we will create subroutines for the behaviors. Subroutines for stop, left wheel forward, left wheel reverse, right wheel forward, right wheel reverse, forward, reverse, clockwise, and counterclockwise. Keep in mind, the way to actuate servo motors with the Arduino and servo library is to specify "0" or "180" to make the motor go forward and reverse and "90" to make it stop. With wheels facing outward 180 degrees, they turn in opposite directions to go forward and backward. We will make nine subroutines for each motion pattern.

Go through each subroutine to verify your motion codes, replacing the loop function call with each subroutine and verifying the robot moves in that way. Namely: stop, left wheel forward, left wheel reverse, right wheel forward, right wheel reverse, forward, reverse, clockwise, and counterclockwise. Make sure the robot is making the correct motion for each subroutine.

Before we dive into more Arduino code, I'd like to introduce a concept called "pseudocode," a helpful technique for understanding your code as it becomes more complex.

Pseudocode is typing out your code in plain language. That is, explaining what the program does in sequence and in simple language, so it is understandable. Type as if you were

explaining your code to a small child. This is not code that runs on the computer! It is code that helps you and the casual observer understand what you are doing (or trying to do) and what is going on. Among other things, it lacks semicolons, curly brackets, and other unnecessary punctuation. I'll use the Arduino "//" command for comments on how the program works.

//
https://www.arduino.cc/reference/en/language/structure/further-syntax/singlelinecomment/

PROTOBot Pseudocode
---------- pseudo ----------
```
        //PROTOBot
BumpLeft        // Name left bump switch
BumpRight       // Name right bump switch
                // The switch's normal electronic state is "1"
                // It changes to "0" when depressed.
Loop()          // Main Loop function

If BumpLeft is the same as "0"
        Go to Reverse()
        Go to clockwise()
        Go to Stop()
If BumpRight is the same as "0"
        Go to Reverse()
        Go to CounterClockwise()
        Go to Stop()
        Forward()       // If neither of the bumps are "0," go forward.
Return to Loop()

Stop()
        leftmotor Stop
        rightmotor Stop
        delay (100)     // Determines how long to do action.
Return
```

```
Forward()
        leftmotor forward
        rightmotor forward
        delay(2) // Determines how long to do action.
Return

Reverse()
        leftmotor reverse
        rightmotor reverse
        delay(400) // Determines how long to do action.
Return
Clockwise()
        leftmotor forward
        rightmotor reverse
        delay(600)      // Note: Delay is longer than
                        // counterclockwise so the robot does
                        // not get caught in a corner.
Return

CounterClockwise()
        leftmotor reverse
        rightmotor forward
        delay(400) // Determines how long to do action.
Return
-------------------------------- pseudo --------------------------------

Below is the actual Arduino code:
-------------------------------- snip --------------------------------
        // First Version Wheel Test
#include <Servo.h>    // Load "Servo" library
Servo servoLeft;      // Left drive servo
Servo servoRight;     // Right drive servo
void setup() {
        servoLeft.attach(2);    // Set left servo to pin 2
        servoRight.attach(3);  // Set right servo to pin 3
```

```
}
void loop() {
forward();
}

void forward() {
        servoLeft.write(180);
        servoRight.write(0);
        delay(2);
}

void reverse(){
        servoLeft.write(0);
        servoRight.write(180);
        delay(400);
}

void clockwise() {
        servoLeft.write(180);
        servoRight.write(180);
        delay(400);
}

void counterclockwise() {
        servoLeft.write(0);
        servoRight.write(0);
        delay(600);
}

void left() {
        servoLeft.write(90);
        servoRight.write(0);
        delay (2);
}

void right() {
```

```
        servoLeft.write(180);
        servoRight.write(90);
        delay (2);
}
void stop() {
        servoLeft.write(90);
        servoRight.write(90);
        delay (100);
}
```
---------- snip ----------

After verifying the subroutines, we need to create our main loop function. In this case, we'd like our robot to move forward until it bumps into something. When it bumps into something with the left bumper, we want the robot to back up, turn right, and go forward again. When it bumps into something with its right bumper, we'd like it to back up, turn left, and go forward again.

We need to do the same thing we did when there were only one switch and one servo motor. That is, read the left switch and read the right switch. As long as both switches read "0," we're good. But when one or the other changes to "1," we want to do the behaviors mentioned above. First, let's check our switches to ensure they are functioning correctly. In the same way we checked the left bumper switch, let's check the right bumper switch. Put this code into the loop:

```
        leftbumper = digitalRead(4);
        rightbumper = digitalRead(5);
        Serial.println (leftbumper, DEC);
        Serial.println (rightbumper, DEC);
```

Download the "First Version Mobile Robot Code" at the end of this chapter. Check the bump switches in the Serial Monitor

under the tools menu. When you press the switch(s), the "1s" should turn to "0s."

In this robot iteration, we control how long or far the robot reverses and turns by using the "delay" command in the subroutines.

delay()
https://www.arduino.cc/reference/en/language/functions/time/delay

This is to keep it simple, but it's not that flexible. In other words, the PROTOBot has fixed distances for particular motion patterns:

```
void reverse(){           // Reverse subroutine
        servoLeft.write(0);
        servoRight.write(180);
        delay(400);
void clockwise() {        // Clockwise subroutine
        servoLeft.write(180);
        servoRight.write(180);
        delay(400);
}
void counterclockwise() {       // Counterclockwise subroutine
        servoLeft.write(0);
        servoRight.write(0);
        delay(600);
}
```

In the TABLEBot chapter, we will learn to use the "while" command to make these motion patterns more flexible. The Parts List is at the end of the book. The full PROTOBot code can be found here: https://github.com/cpeavy2/PROTOBot

Chapter 2:

The TABLEBot

The author Camp Peavy and Buggy the TABLEBot.

After building the PROTOBot, let's have some fun and get this robot doing something besides bump-and-go. Since 2003 at the HomeBrew Robotics Club of Silicon Valley, we've had the TABLEBot Challenge event. The end goal of the challenge is to play tabletop soccer. Although playing tabletop soccer is way beyond this chapter's purpose, the pathway toward this goal (and any complicated undertaking for that matter) is to do it in phases.

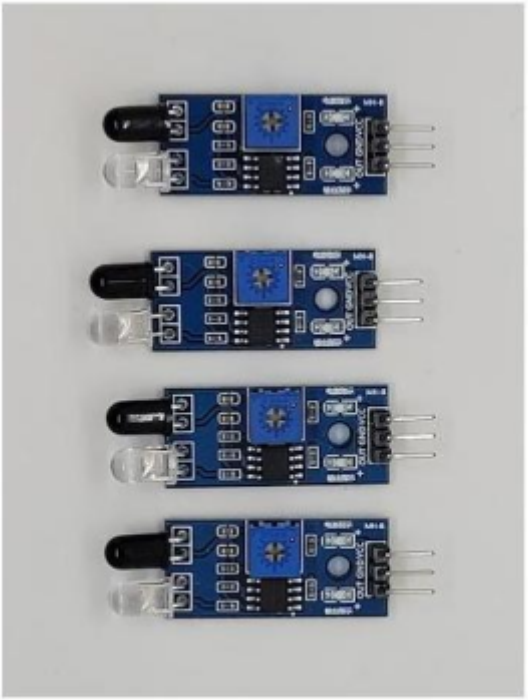
TABLEBot Challenge Phases are as follows:

- Phase I: Go from one end of the table to the other and back.

- Phase II: Find and Push a Block off the edge of the table.
- Phase III: Find and Push the Block into a Shoebox mounted at the end of the table.

The "Phase I" TABLEBot

Now that we have a basic bump-and-go robot (the PROTOBot), let's make it a TABLEBot. We first need drop-sensors so the robot can stay on the table. Remember, the TABLEBot is built in phases, and Phase I is to go from one end of the table to the other and back. We'll do this with another type of sensor; an "infrared" or IR sensor. IR sensors have an emitter and a detector. That is an IR LED (Light Emitting Diode) and an IR detector.

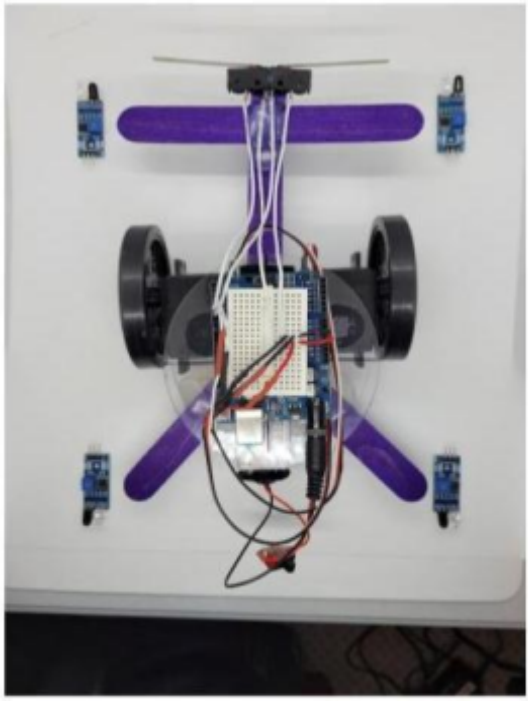


These are the infrared sensors we'll be using on the TABLEBot. Notice they have 3 connections just like the servos.

The IR detector is a transistor that switches on when it senses infrared light. On our TABLEBot, we're going to add four IR detectors. Two in the front and two in the back. This way, as the robot goes forward and backward, it monitors the drop detectors to ensure the

tablespace below. If there is no tablespace, the robot is about to fall off and needs to move back or forward to stay mobile and stay on the table. I'm using drop sensors available through Amazon, but a variety of small IR sensors are available.

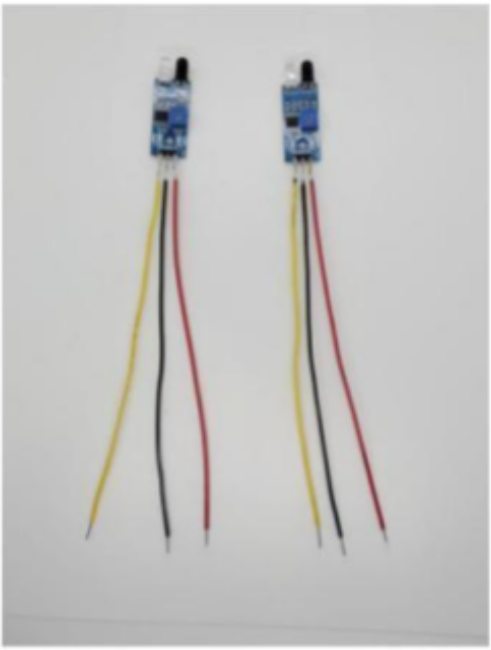
To physically place these sensors, we will use popsicle sticks. I've hot-glued the popsicle sticks into position to mount the front IR sensor. I will use a special kind of epoxy called "Sugru." Sugru is one of those homebrewed techniques I've mentioned to help you build faster, easier, and more creatively. I use Sugru in this instance because there isn't a way to formally mount this sensor to a popsicle stick, and it is tacky like clay and will harden like plastic and adhere like glue; well, it is epoxy.



Here we've glued the popsicle sticks into place and are preparing to attach IR drop sensors.

Be sure and leave enough space for the snap-action switches to be actuated. The switches will be used to verify the acquisition of the Block. Keep in mind the TABLEBot is built in three phases. Phase I is just to go from one end of the table to the other. In the next phase, you will push a Block off the edge. Then finally, in Phase III, the Block will be deposited in a Shoebox mounted at the end of the table.

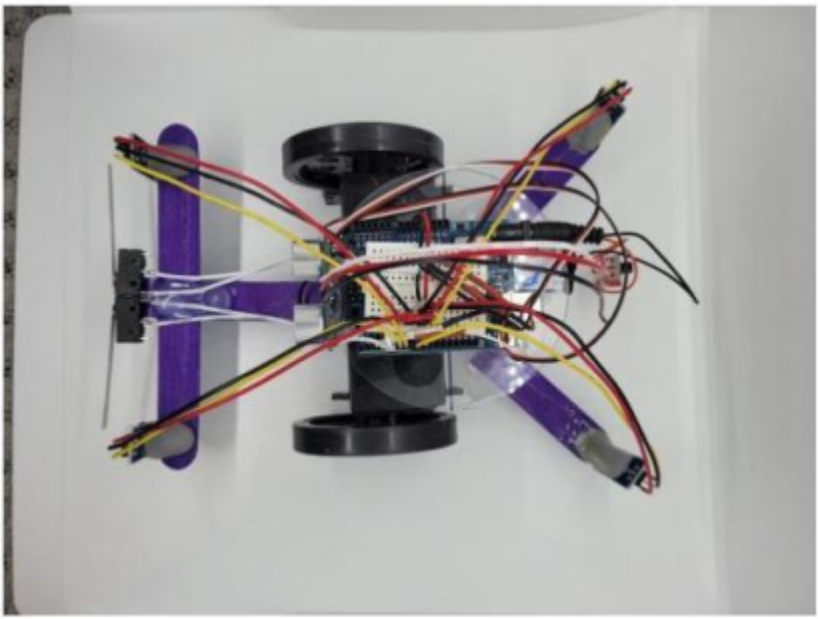
Notice the IR sensors that I am using have 3-pins or connections. 5V power, GND, and signal, kind of like the servos. Remember, for actuating motors, we talked about wiggling a wire, and for the sensor, we talked about listening to a wire to see if it's wiggling. Well, there are three wires; two are for power (5V and GND) and the third wire (signal) is the wire you monitor to detect something.



Solder ~6-inches of 22AWG Solid core wire to each IR ping. This makes a homebrew plug that you can plug into the solderless breadboard.

Cut twelve pieces of 22AWG solid wire ~6” in length (your measurement may vary. This is the same "homebrew plug" you used to plug in the servos. You may have to shuffle around the connections on the solderless breadboard to accommodate the additional plugs, but this is the advantage of the solderless breadboard. You can figure it out as you go along. If you need more sockets (for 5V and GND, for example), just jump one row to another, and you have a new row.

Make them black, red, and yellow (or a third color) if possible. These will be the power and signal connections for your



Adhere the IR sensors to the popsicle sticks with Sugru.
Plug the wires into 5V+, GND and D5, D6, D7 and D8.

sensor. I've adhered them with Sugru but have 5-minute epoxy handy if needed. Once the IR sensors are attached, power them on. The sensors I'm using have indicator lights. Yours may or may not. Other sensors might be analog; these are digital.

IR code snippet

```
---------------------------------- snip ----------------------------------
int val6 = 0;     // Initialize val6 value
void setup() {
 Serial.begin(9600);    // Setup serial monitor for debug
 pinMode(6, INPUT);   // Define pin 6 as input
}
void loop() {
 val6 = digitalRead(6);            // Read pin 6
```

```
 Serial.println(val6);              // Print pin 6 in Serial Monitor
}
---------------------------------- snip ----------------------------------
```

Now that we've physically attached the IR drop sensors and connected them electronically, let's write some software. With the bumper switches, we used the "delay" command in the subroutines Stop, reverse, clockwise, and counterclockwise to control the robot's movement the way we wanted. The command "delay" pauses the program for the time specified in milliseconds (1/1000 of a second). With the PROTOBot program, we set the Stop, reverse, clockwise and counterclockwise subroutines to "delay(1000)" so the robot would stop, reverse and turn clockwise or counterclockwise for one full second (1000 milliseconds). The subroutine "forward" was set at delay(2) or 2 milliseconds so the routine could loop around quickly and monitor the bumpers to react quickly.

The "While" Command

With the IR sensors, I'd like to introduce a new Arduino command, the "while" command:

while
https://www.arduino.cc/reference/en/language/structure/control-structure/while/

The while command will allow us to adjust the motions more flexibly instead of using "delay" for each motion. First off, let's set all the subroutine delays to 2 milliseconds, that is, delay(2);. Then for each motion, we'll specify "while" and tell the robot how long we'd like it to do that motion by specifying a number. For example:

```
---------------------------------- snip ----------------------------------
x = 0
```

```
while (x <100) {
 Forward();
 x++;
 }
```
---------- snip ----------

In this case, the program would run the "Forward" subroutine 100 times. "x++" tells the program to increment 1 unit every time it loops. This gives us much more flexibility in how long we run the subroutines over hardcoding time frames for each behavior.

Now let's move to reading the IR sensors. Like the servos, the IR sensor I am using has three wires: 5 volt (power) and Ground, and signal. The signal wires (four of them for four sensors) should be plugged into D6, D7, D8, and D9 on the Arduino. First thing, set the variables as integers and equal to "0."

---------- snip ----------
```
int val6 = 0;
int val7 = 0;
int val8 = 0;
int val9 = 0;
```
---------- snip ----------

Like the bump-switches, we make the IR pins inputs with the below commands:

---------- snip ----------
```
 pinMode(6,INPUT);
 pinMode(7,INPUT);
 pinMode(8,INPUT);
 pinMode(9,INPUT);
```
---------- snip ----------

Then in the loop routine, we read the IR sensors, which just like the bump-switches, read "1" in the serial monitor (in the Tools menu) when it senses the table and changes to "0" when it doesn't detect the table. We want to go forward until the robot senses the drop-off. When it senses the drop, we want the robot to stop, go backward, and then turn left or right, depending on which sensor is triggered. With the IR sensors, you have sensors in the rear to keep the robot from backing off the table. As mentioned with the bump-switches, we included the time element in the subroutine. We'll work the timing within the main loop routine with the "while" command with the IR sensor. The front left sensor will look something like this:

```
---------------------------------- snip ----------------------------------
{
x = 0;
while (x <100) {
 Stop();
 x++;
}
x = 0;
while (x <100) {
 Reverse();
 x++;
}
x = 0;
while (x <200) {
 Counterclockwise();
 x++;
}
}
---------------------------------- snip ----------------------------------
```

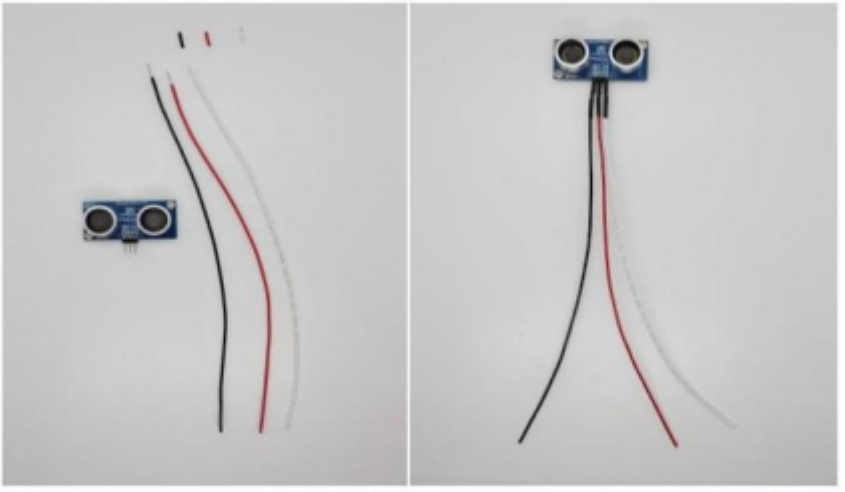
This series of commands make the robot stop, backup, and turn for ~¼ second each. Notice the "delay" in the subroutine is 2ms, and the count for "while" is 100. So for each motion, it

does the pattern for 2ms x 100 times. You can adjust the time for each motion by changing the "while" value.

This is the "Phase I" TABLEBot. It goes from one end of the table to the other and back without falling off... hopefully. The code can be found here: https://github.com/cpeavy2/TABLEBot_I.

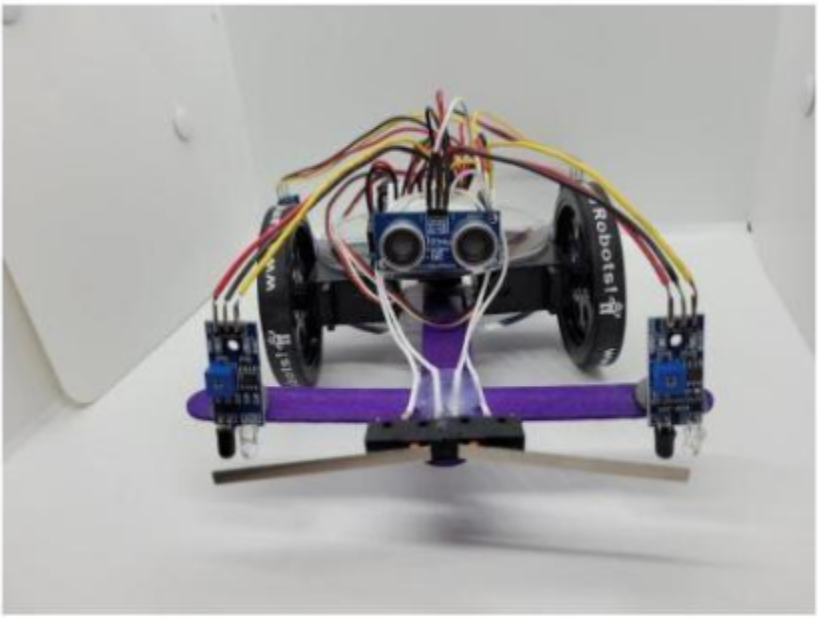
The “Phase II” TABLEBot

The next phase of the TABLEBot (Phase II) is to push a Block off the table's edge. To do this, we will use an ultrasonic sensor. An ultrasonic sensor works by sending an inaudible sound signal above 20KHz (ultrasonic) so that it can't be heard by human ears and then detecting its echo, like a bat. Normal healthy hearing frequencies are between 20 to 20,000Hz. Most of these sensors consist of a separate emitter and detector. One of the most popular is the "Ping" sensor from Parallax https://www.parallax.com/product/ping-ultrasonic-distance-sensor/.



The ultrasonic sensor has 3-wires like the servos and IR sensors. Power (Red and Black) and signal. Cut three 22-AWG solid wire to use as homebrewed plugs. Red goes into 5V, black goes into GND and the third wire (signal) goes into D10.

The Parallax Ping sensor has three wires like the servos and IR sensors. Two for power and a third signal wire.



Here's the completed Phase II TABLEBot with drop-sensors to keep it from falling off the table and an ultrasonic sensor to sense the block.

The signal wire on the Ping sensor functions as an output and an input. Basically, you "wiggle" the wire to send a pulse out and listen to the wire for a return signal. The circuitry on the board does the magic of sending the ultrasonic vibration from one of the transceivers (the little round aluminum knob) and receiving the signal from the other transceiver. The reading of the signal pin tells you the distance to the obstacle. The higher the number, the further the distance to the obstacle; the lower the number, the closer the obstacle. This application assumes there is nothing on

the table but the Block. Please do not stand in front of the robot, or the robot will interpret you as a Block.

To attach the Ping ultrasonic sensor physically, we will use hot glue. To attach it electronically, we will again use 22AWG solid wire. Cut three wires (red, black, and a different color) about 8"; better too long than too short. You can always cut it. If needed jumper to another row on the solderless breadboard. Here we're going to reintroduce our bump-sensor so the TABLEBot knows when it's touched the Block. First, let's get the ultrasonic sensor working. I'm using the "Ping" sensor from Parallax. If you have a different sensor, it might function differently.

Ping Ultrasonic Range Finder
https://www.arduino.cc/en/Tutorial/BuiltInExamples/Ping/

I'm going to clear the loop function again to get a handle on and understand the ultrasonic sensor.

Ping Ultrasonic Code Snippet

```
---------------------------------- snip --------------------------
const int pingPin = 10;
void setup() {
 Serial.begin(9600);
}
void loop() {
        long raw_distance;
        pinMode(pingPin, OUTPUT);
        digitalWrite(pingPin, LOW);
        delayMicroseconds(2);
        digitalWrite(pingPin, HIGH);
        delayMicroseconds(5);
        digitalWrite(pingPin, LOW);
        pinMode(pingPin, INPUT);
        raw_distance = pulseIn(pingPin, HIGH);
        Serial.print(raw_distance);
```

```
        Serial.println();
        delay(100);
}
---------------------------------- snip ----------------------------------
```

Detecting the Block

Look at the Serial Monitor again (under the Tools menu). You should see the raw reading of the Ping ultrasonic sensor. It should be reading ~16000, and when you put your hand ~6-inches in front of the sensor, the raw-reading value should drop to ~500. This is how you will detect the Block.

When you're programming robots, think in terms of modules and behaviors. Have your subroutines do particular things, read particular sensors, or do particular movements. Write down what you want the robot to do and then break it down into simple behaviors.

For example, in this case, you want the robot to turn 360° with its ultrasonics to try and "see" the Block. Of course, this environment is contrived, as nothing is expected to be on the table except the Block. Again, do not stand next to the table, or the robot will think you are the Block. In this example, we will have the robot turn in ~45° increments until it turns entirely around (eight turns or thereabouts), at which point the robot will go forward two inches and Ping again.

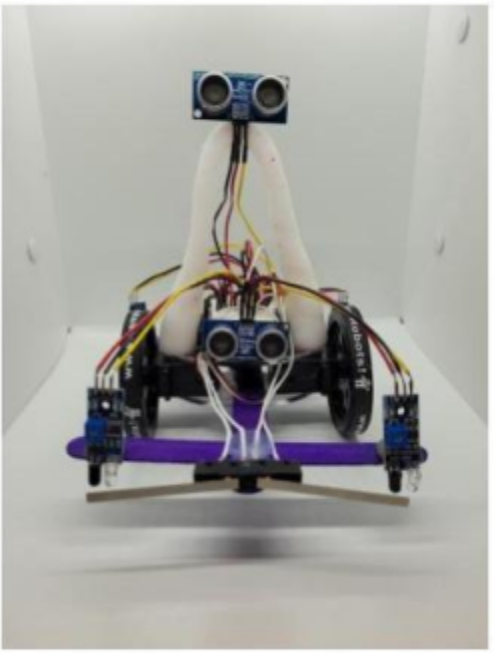
Find the complete Phase II TABLEBot code here: https://github.com/cpeavy2/TABLEBot_II

The “Phase III” TABLEBot

The final phase of the TABLEBot is "Phase III." In this phase, we will add another ultrasonic sensor so after the robot has

acquired the Block, it can then move it into a Shoebox mounted at the end of the table.

The other ultrasonic sensor will be higher than the sensor that detects the Block. This is so you can use it to see the Shoebox or goal. Again, the system has two ultrasonic sensors; a lower one and a higher one. The lower one is used to detect and acquire the Block. Once this has been accomplished, the higher ultrasonic sensor will detect and move the robot toward the Shoebox or goal. Once again, I have used Instamorph to create a mount for the upper servo. You might find another way to do it.



The box-sensing ultrasonic is placed above the block-sensing one as it will be used to find the shoebox goal after the robot has acquired the block.

Solder 22AWG solid core wire to the sensor pins to make the homebrewed plugs we've used for the servos, switches, and first ultrasonic sensor. You may need to jump the rows around on the solderless breadboard to make more connections for 5v(+) and GND(-) the signal wire will go into D10 on my system.

To move the Block, we need to acquire it, so I've added “fingers” to the front bumper of our TABLEBot. This is so that once

the bot has touched the Block, it can turn clockwise or counterclockwise without losing the Block.

So the basic strategy here is first to acquire the Block. So the robot searches the table (without falling off) for the Block using the lower ultrasonic sensor. If it detects something with its lower ultrasonic sensor, it needs to check the upper ultrasonic to verify it's not the Shoebox. If the lower sensor has a raw reading of ~700 and the upper sensor is greater than 1000, chances are you have acquired the Block.

I've exended the fingers with heatshrink tubing (unshrunk) so the tips are flexible. This is so once the robot acquires the block it can turn clockwise and counterclockwise without losing it.

Now the robot needs to look around for the Shoebox. Once it detects < 5000 (depending on the size of your table), it has probably seen the Shoebox. It needs to move toward the upper obstacle. If the raw readings drop below 5000, rotate clockwise or counterclockwise until it picks up the obstacle again. Move toward the obstacle, which again is presumably the Shoebox, until the raw sensor reads ~1000 on my robot (your mileage may vary). At this point, it has presumably deposited the Block into the Shoebox. That would be a "Phase III."

Shaping Behavior

We will start to granularize sensor readings and behaviors in the program below. We want a subroutine for each of the nine possible differential drive system motion patterns: Stop, left_forward, left_reverse, forward, left_reverse, counterclockwise, right_reverse, clockwise, and reverse. We have a subroutine for the Ping sensor, read_ping. We have a subroutine to read all the drop sensors and respond appropriately, ir_check. These subroutines or functions are prefaced with "void." This means that variables created within that function can only be used locally or within that function. Variables or declarations you want to use globally need to be defined outside these "void" functions or subroutines, like the part of the sketch above setup where the servo library is declared and various variables are defined. Defining them outside of the void functions allows them to be used globally. At this point, consider what it is you want your robot to do. Create and shape the behavior as you go along. Experiment! Try values, see what the robot does, and adjust accordingly.

The TABLEBot Challenge is an excellent example of "subsumption architecture," where simple behaviors layered on top of one another give the resemblance of intelligence. That is, first, you are moving and staying on the table. Next, you are moving and looking for the Block to push off the table and lastly, moving, looking for the Block, and moving it into the Shoebox mounted at the end of the table.

TABLEBot Phase III Pseudocode

```
---------------------------------- pseudo ----------------------------------
        // TABLEBot Challenge Phase III: Move the Block into the Box.
BumpLeft        // Name left bump switch
BumpRight       // Name right bump switch
                // The switches' normal electronic state is "1"
```

```
                // It changes to “0” when depressed.
PingPin1        // Name Ping #1 Sensor Pin (lower)
PingPin2        // Name Ping #2 Sensor Pin (higher)
Raw_distance1       // Value for Raw Distance Ping #1 (lower).
Raw_distance2       // Value for Raw Distance Ping #2 (higher).
LF_IR = 0       // Value for Left Front infrared (IR) sensor.
RF_IR = 0       // Value for Right Front infrared (IR) sensor.
y = 0   // Generic counter Variable
Loop() // Main Loop function
        If BumpLeft is the same as "0"          // The robot has
                                                // bumped
                                                // into the Block.
                Go to Goal()
        If BumpRight is the same as "0"         // The robot has
                                                // bumped
                                                // into the Block.
                Go to Goal()
        Read_ping1()                            // Read the Ping #1
(lower).
        If (Raw_distance1 < 700)                // Block acquired
                Go to Goal()
        If (Raw_distance1 <= 5000)              // This checks to
discern
                                                // Block from Box
        Read_ping2()            // Read the Ping #2 (higher).
                If (Raw_distance2 - 500) >= Raw_distance1
                                // Block is in front of Box.
                        Go to Forward()
                If (Raw_distance1 > 5000)   // Can’t find the Block.
Turn clockwise.
        Clockwise() x 100               // Use “while” command to
                                        // determine how long to
                                        // turn.
        Stop() x 100    // Use “while” command to determine
                        // how long to stop.
        y = y+1         // Timing loop.
```

```
        If (y > 10)       // If the robot has turned 10 times
                Forward() x 200       // Use “while” command to
                                      // determine how long to go
                                      // forward. This moves the
                                      // robot forward a little bit
                                      // searching for the Block.
        y = 0
Return to Loop()
Read_ping1()          // Read the lower Ping sensor
                      // Read 1st Ping sensor, store value in
Raw_distance1
Return
Read_ping2()          // Read the upper Ping sensor
                      // Read 2nd ping sensor
                      // Store value in Raw_distance2
Return
        // Go to the Box
Goal()
        Read_ping2()          // Read the 2nd (higher) Ping sensor
        If (Raw_distance2 > 5000)    // You don’t see the Box.
                Clockwise() x 100     // Use “while” command to
determine
                                      // how long to turn.
                Stop() x 100          // Use “while” command to
                                      // determine how long to
                                      // stop. . You will
                                      // be taking an
                                      // ultrasonic reading.
                      y = y+1         // Timing loop.
                      If (y > 10)     // If the robot has turned 10
                                      // times
                              Forward() x 200       // Use “while”
                                                    // command
                                                    // determine
                                                    // how long to
                                                    // go forward.
```

```
                                                    // This moves
                                                    // the robot
                                                    // forward a
                                                    //  little bit
                                                    // searching
                                                    // for the Box.
        y = 0
        If (Raw_distance2 < 5000)    // The TABLEBot sees the
                                     // Box.
                Forward()
        If (Raw_distance2 < 1500)
                Dance()              // Robot has successfully
                                     // deposited the Block
                                     // in the Box.
Return to Loop()
        // The robot has placed the Block in the Box. This is the
        // celebration dance.
Dance()
        Reverse() x 400      // Use “while” command to determine
                             // how long to reverse.
        Clockwise() x 400    // Use “while” command to determine
                             // how long to turn clockwise.
End()
        // This ends the routine
End()
        Stop()
        End()
Stop()
        leftmotor Stop
        rightmotor Stop
Return
Forward()
        leftmotor forward
        rightmotor forward
        IR_check()           // Check the IR sensors
Return
```

```
Reverse()
        leftmotor reverse
        rightmotor reverse
        IR_check()      // Check the IR sensors
Return
Clockwise()
        leftmotor forward
        rightmotor reverse
        IR_check()      // Check the IR sensors
Return
CounterClockwise()
        leftmotor reverse
        rightmotor forward
        IR_check()      // Check the IR sensors
Return
        // This is the subroutine where the robot checks the IR
        // sensors and reacts accordingly.
IR_check()
        Read IR sensor LF_IR
                If LF_IR is “High” or “1”
        Stop() x 100            // Use “while” command to determine
                                // how long to stop.
        Reverse() x 100         // Use “while” command to
                                // determine how long to
                                // reverse.
        Clockwise() x 200       // Use “while” command to
                                // determine how long to
                                // turn clockwise. Note: turns
                                // farther than RF_IR to avoid
                                // getting stuck in corner.
        Read IR sensor RF_IR
                If RF_IR is “High” or “1”
        Stop() x 100            // Use “while” command to determine
                                // how long to stop.
        Reverse() x 100         // Use “while” command to
                                // determine how long
```

```
                                    // to reverse.
        CounterClockwise() x 200    // Use “while”
                                    // command to
                                    // determine how long to
                                    // turn counterclockwise.
Return
---------------------------------- pseudo ----------------------------------
```

Try stuff! Get the basic behavior and then add to it. Simplify, simplify, simplify. First, make it work, then make it better. Good enough is good enough. If you get stuck on one aspect, work on something else.

Be sure and enjoy your pastime. Celebrate small victories! Let your robot run a bit. At the homebrew club, we have a saying: “We’re hard to impress but easily amused.” To a degree, you have to appreciate the simple things. Like a motor moving a robot around on the table, the IR sensors sensing the drop, and the robot running its program that YOU created; you may even experience emergent behavior. Shape the behavior you want.

Discuss the problem(s) you're having with others, especially those knowledgeable in this area. Ask questions on user groups or to manufacturers. The HomeBrew Robotics Club mailing list is a great resource:

HomeBrew Robotics Club
https://groups.google.com/g/hbrobotics

We have had much fun playing the "TABLEBot Challenge" at the HomeBrew Robotics Club over the years. As mentioned, we do not specify the size or color of the table, the Block, or the Shoebox. Nor do we specify the design of the robot. Use the robot you have or build something as I've outlined here. Keep in mind, many robot clubs are online now, so you'll have a chance to show off your creation. Maybe even at one of our annual challenge

meetings, which we now hold online. Find the complete Phase III TABLEBot code here: https://github.com/cpeavy2/TABLEBot_III

Chapter 3:

Machine Intelligence

I started robotics with a book by David L. Heiserman entitled "How to Build Your Own Self-Programming Robot" (1979 by TAB BOOKS). Heiserman describes an 8085 microprocessor controlled device with an 8-bit data bus, a 12-bit address bus, and a load button. You'd set the data, set the address, and punch the "Load" button to program the robot one byte at a time in binary!

My original 1987 “Rodney” from “How to Build Your Own Self-Programming Robot”... note the address and data toggle switches on top and relays on bottom in front.

Heiserman also had a theory on "machine intelligence," which involved learning through experience, an example of reinforced learning relevant to this day. The robot's name was "Rodney."

After wrestling with this graduate-level course in electronics for over five years — building the front panel, half the mainboard, auxiliary

power, and robot body — I finally managed to re-create "Alpha" level machine intelligence as described by Heiserman with a store-bought 386SX PC motherboard, relay I/O card, and GWBasic.

It's Alive!

This was one of those "It's ALIVE!" moments every robot-builder craves. I used a variant of this machine learning algorithm in my 1996–97 Robot Wars robot "Gladiator Rodney" and my Burning Man “dancing” Robot (1999–2005) "Springy Thingy.

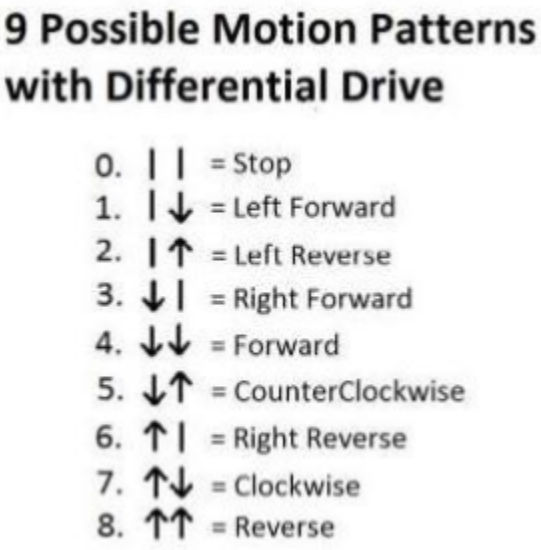
Gladiator Rodney: 1997

This chapter is about building a miniature version of Heiserman's "Rodney." Since it's a miniature version of the original, let's call him "Rodney Jr." But what does "Junior" do? Well, it moves. Yep, that's it; it moves. The deal with Rodney and his protégé "Junior" is he will try different things until he successfully achieves his goal (that is, moving), and then he becomes more and more efficient at achieving this goal (learns). Primitive, yes, but like an amoeba bouncing around in a drop of water from random motions, it learns the responses that move it the farthest sooner; not a bad strategy for a little machine

exploring its world. Besides, it's not what Rodney does but how he does it that's important!

It's Not What but How

In other words, "Junior" isn't just programmed to go forward and backward but programmed to monitor the mobility sensor (more about that later) and try random stuff until conditions meet what constitutes its "goal."



Let me explain. First, Rodney makes random motions. Then when he gets a successful response for a particular motion pattern, when in that pattern again, he tries the previously successful reaction. If the response is still successful, Rodney Jr. increments the confidence level of that response; if the response is no longer successful, the confidence level decrements. This robot will think for itself and learn from its experience. You might say he's smarter than the average robot!

According to Heiserman, Alpha-level machine intelligence can be considered a basic reflex, like the amoeba mentioned above bouncing around in a drop of water. In Heiserman's

Rodney, the robot's goal was simply to move. In a differential drive system, there are nine possible motion patterns, including "stop"; however, the only two patterns with any linear velocity are "forward" and "reverse" (Codes 4 and 8). All other motion patterns rotate around a point; they have angular velocity.

Rodney Jr. studying the 9 possible motion codes.


Spingy Thingy and Rusty at Burning Man early 21st century.

I quickly realized this when running the store-bought PC version of Rodney, where every pattern or Code (except for "stop") would detect motion so the robot would go clockwise (Code 7) or right wheel forward (Code 3) or whatever code was randomly selected forever because the mobility detector would always detect motion . . . not very interesting and not very useful.

What I did with "Gladiator Rodney" (the Robot Wars robot) was let the robot run for a limited time and then randomly change patterns regardless of whether or not the robot had stalled. This behavior was overridden when the robot detected the opponent's beacon. The robot would explore the arena randomly while looking for the target.

Same thing with "Springy Thingy" my Burning Man robot. I would let her go for a limited number of clicks and then change patterns (codes 0–8). I billed her as a "Dancing Robot." For Gladiator Rodney and Springy Thingy, the mobility detector was a spring-loaded gate-wheel from ACE Hardware with rare earth magnets glued (E6000) all around the perimeter. I glued a reed switch to the caster frame so that when the wheel spun (i.e., the robot was moving), the reed switch would switch off and on. It's funny the mind connects the random motion to music, and it appears the robot is reacting to the music (i.e., dancing). Springy Thingy is alive and well (twenty-three years young), albeit in a much-altered state (Radio and Arduino control).

With "Rodney Jr." the Alpha level machine intelligence tries random motion patterns until the robot discovers either forward or reverse, again, the only two motion patterns with linear velocity. When only one of the wheels (left or right) turns, or the robot turns clockwise or counterclockwise, this is termed angular velocity. That is, the robot rotates around a point.

You can think of this "goal" as the robot moving the maximum displacement or magnitude for a given area. Like a basic reflex trying random things to see what it can do to reach its goal or make it "happy." Not a bad plan for a little creature exploring its big world.

Alpha Level Machine Intelligence

"Alpha" can be described as random motion. The robot will randomly try motion patterns or codes until he stumbles upon 4 (forward) or 8 (reverse). Then, when he stalls or bumps into something, he will again try random motion patterns until he rediscovers forward or reverse. Alpha-level machine intelligence is hardly intelligent at all. That is, the motion patterns are basically random.

Alpha Pseudocode

```
------------------------------- pseudo ----------------------------------
        // Alpha
Loop()                          // Main Loop function
        Read mobility_sensor            // Read the mobility sensor.
        If Mobility_read is the same as Mobility_read_past
                Increment stall_ticks +1        // Accumulate
                                                // stall_ticks readings.
        Else stall_ticks = 0                    // the robot is moving reset
                                                // stall_ticks.
If stall_ticks > 100                // The robot has stalled.
        Go to Motion_Code()             // Get new random motion.
Else Mobility_read_past = Mobility_read     // Put previous
                                            // reading into
                                            // memory.
Go to loop()

Motion_Code()
        stall_ticks = 0             // Reset "stall_ticks" for new motion
```

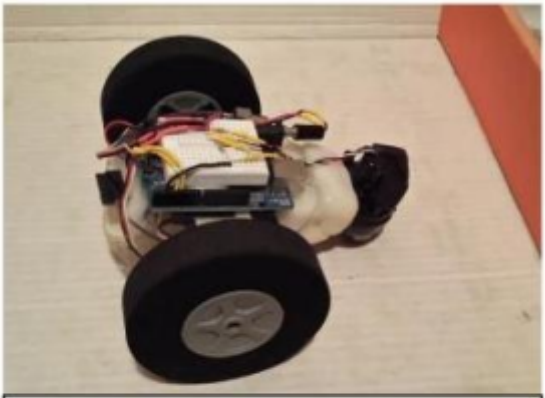
```
                                    // code.
MotionCode = random(9)      // Generate random motion code.
If MotionCode = 0
        Go to Stop()
If MotionCode = 1
        Go to LeftForward()
If MotionCode = 2
        Go to LeftRear()
If MotionCode = 3
        Go to RightForward()
If MotionCode = 4
        Go to Forward()
If MotionCode = 5
        Go to CounterClockwise()
If MotionCode = 6
        Go to RightRear()
If MotionCode = 7
        Go to ClockWise()
If MotionCode = 8
        Go to Reverse()
Go to loop()

// Motion pattern subroutines
Stop()
        leftmotor stop
        rightmotor stop
        delay (100)             // Determines how long to do action.
Return
LeftForward()
        leftmotor forward
        rightmotor pause
        delay(2)
Return
LeftRear()
        Leftmotor reverse
        rightmotor pause
```

```
        delay(2)
Return
RightForward()
        leftmotor pause
        rightmotor forward
        delay(2)
Return
RightRear()
        leftmotor pause
        rightmotor reverse
        delay(2)
Return
Forward()
        leftmotor forward
        rightmotor forward
        delay(2)
Return
Clockwise()
        leftmotor forward
        rightmotor reverse
        delay(2)
Return
Counterclockwise()
        leftmotor reverse
        rightmotor forward
        delay(2)
Return
Reverse()
        leftmotor reverse
        rightmotor reverse
        delay(2)
Return
------------------------------ pseudo ----------------------------------
```

Beta Level Machine Intelligence

"Beta" class machine intelligence features memory; that is, for a given motion pattern, when the robot discovers either forward or reverse, it remembers that successful response and uses it the next time when in that motion pattern or situation and encounters a stall condition. The program also increases confidence if the reaction continues to be successful and decrements the confidence level if it fails. Beta-level intelligence remembers successful responses and uses them when in the same circumstance.



This is “Rodney Jr” based on two servos sandwiched between two acrylic disks, driven by an Arduino. The key to Junior’s intelligence is the Roomba caster wheel in the rear. 1000 times simpler than the original “Rodney” which had to be programmed in binary.

Beta Pseudocode

```
------------------------- pseudo -----------------------------
        // Beta
Loop()          // Main Loop function.
Read mobility_sensor            // Read the mobility sensor.
        If Mobility_read is the same as Mobility_read_past
                Increment stall_ticks +1        // Accumulate
```

```
                                        // stall_ticks readings.
        Else stall_ticks = 0            // the robot is moving
                                        // reset.
                MemArray[PastMotion][0] = MotionCode
                        // Make MotionCode a successful response
                        // to PastMotion.

                MemArray[PastMotion][1] = +1
                        // Increment confidence level of response.

        If stall_ticks > 100            // The robot has stalled.
                MemArray[PastMotion][1] = -1        // Decrement
                                                    // confidence
                                                    // level of
                                                    // response.
                Go to Motion_Code() // Get new random motion.
        else mobility_read_past = mobility_read     // Put previous
                                                    // mobility
                                                    // reading into
                                                    // memory.
Go to Loop();

Motion_Code()
        Stall_ticks = 0;        // Reset "stall_ticks" for new motion
code.
        PastMotion = MotionCode // Commit previous motion to
memory.
If MemArray[MotionCode][1] > 3      // If the confidence level of
the response is
                                    // greater than 3.
        MotionCode = MemArray[MotionCode][0]    // Make the
                                    // high-confidence
                                    // response the
                                    // new motion code.
        Go to Loop();
Else MotionCode = Random(9)         // Generate random number
```

```
                                        // for motion pattern.
        If MotionCode = 0
                Go to Stop()
        If MotionCode = 1
                Go to LeftForward()
        If MotionCode = 2
                Go to LeftRear()
        If MotionCode = 3
                Go to RightForward()
        If MotionCode = 4
                Go to Forward()
        If MotionCode = 5
                Go to CounterClockwise()
        If MotionCode = 6
                Go to RightRear()
        If MotionCode = 7
                Go to ClockWise()
        If MotionCode = 8
                Go to Reverse()
Go to loop()

// Motion pattern subroutines (Stop, LeftForward, LeftRear,
// RightForward, etc.) are the same as
// Alpha Pseudocode above.
------------------------------- pseudo ----------------------------------
```

Gamma Level Machine Intelligence

"Gamma" class machine intelligence generalizes this information for heretofore unknown circumstances. That is, when the robot encounters a stall condition in a motion pattern for which it has no successful memory (that is, no Beta responses), it tries a high-confidence Beta level response from other motion patterns

before reverting back to Alpha level behavior (random/reflex responses).

So with each incremental level of intelligence: Alpha, Beta, and Gamma, the robot learns the motion patterns, maximizing displacement sooner and sooner. When you come up with a good solution, you remember it, and then you can apply that solution elsewhere, it's likely because of your past experience that you're going to come up with that solution sooner.

Gamma pseudocode

```
------------------------------- pseudo ----------------------------------
        // Gamma
Loop()          // Main Loop function
        Read mobility_sensor                    // Read the mobility
                                                // sensor.
        If mobility_read is the same as mobility_read_past
                Increment stall_ticks +1        // Accumulate
                                                // "stall_ticks"
                                                // readings.
        Else stall_ticks = 0                    // the robot is moving
reset.
        MemArray[PastMotion][0] = MotionCode    // Make
                                        // MotionCode a successful
                                        // response to PastMotion.
        MemArray[PastMotion][1] = +1            // Increment
                                                // confidence level of
                                                // response.
If stall_ticks > 100                            // The robot has
                                                // stalled.
        MemArray[PastMotion][1] = -1            // Decrement
                                                // confidence level of
                                                // response.
        go to Motion_Code()  // Get new random motion.
else mobility_read_past = mobility_read         // Put previous
                                                // mobility reading into
```

```
                                        // memory.
Go to Loop();

Motion_Code()
        stall_ticks = 0                 // Reset "stall_ticks" for new
                                        // motion code.
        PastMotion = MotionCode         // Commit previous motion to
                                        // memory.
        if MemArray[MotionCode][1] > 3  // If the confidence
                                        // level of the
                                        // response is
                                        // greater than 3.
        MotionCode = MemArray[MotionCode][0]    // Make the
                                        // high-confidence
                                        // response the
                                        // new motion code.
Go to loop()

Else gamma();
Gamma()             // If confidence is >50 than try this pattern
                    // before reverting to Alpha.
        If MemArray[0][1] > 50          // if this motioncode
                                        // confidence is >50
                                        // then use it.
        MotionCode = MemArray[0][0]
Go to loop()
        If MemArray[1][1] > 50
                MotionCode = MemArray[1][0]
                Go to loop()
If MemArray[2][1] > 50
                MotionCode = MemArray[2][0]
                Go to loop()
If MemArray[3][1] > 50
                MotionCode = MemArray[3][0]
                Go to loop()
If MemArray[4][1] > 50
```

```
                MotionCode = MemArray[4][0]
                Go to loop()
If MemArray[5][1] > 50
                MotionCode = MemArray[5][0]
                Go to loop()
If MemArray[6][1] > 50
                MotionCode = MemArray[6][0]
                Go to loop()
If (MemArray[7][1] > 50
                MotionCode = MemArray[7][0]
                Go to loop()
If (MemArray[8][1] > 50
                MotionCode = MemArray[8][0]
                Go to loop()

Else MotionCode = random(9)        // No high confidence
                                   // responses use random
                                   // motion pattern.
If MotionCode = 0
        Go to Pause()
If MotionCode = 1
        Go to LeftForward()
If MotionCode = 2
        Go to LeftRear()
If MotionCode = 3
        Go to RightForward()
If MotionCode = 4
        Go to Forward()
If MotionCode = 5
        Go to CounterClockwise()
 If MotionCode = 6
        Go to RightRear()
 If MotionCode = 7
        Go to ClockWise()
 If MotionCode = 8
        Go to Reverse()
```

Go to loop()

// Motion pattern subroutines (Stop, LeftForward, LeftRear,
// RightForward, etc.) are the same as
// Alpha Pseudocode above.
---------- pseudo ----------

How many projects start slow but speed up once you get the hang of it? You try random stuff at first, some of which is stupid, and sometimes without even doing anything, you run it through your head. That's learning! You think through random stuff, and you'll have a memory, if not a direct memory, a realization that this is similar to something you've done before! That is generalizing!

This is the underside of the robot. The 9-volt battery and two servos are sandwiched between the clear acrylic disks. The two servos are adhered with foam double-stick tape. The Roomba caster wheel is mounted with Shapelock. It only spins when going forward or reverse.

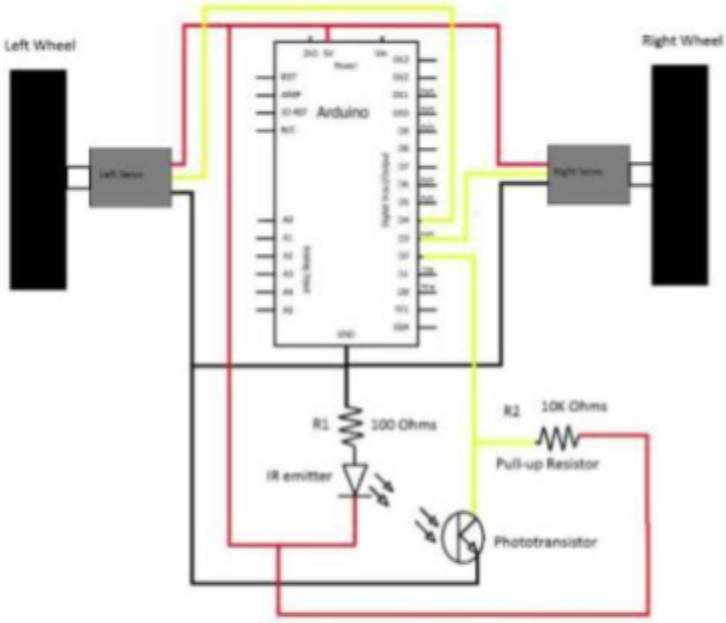
Like a child randomly reaching for a goal, we learn a lot just by being in the environment. You have to get your robot “out there” before it can begin to have its own experiences from which to learn. Isn’t this the way you

learned things; random and haphazard at first, and then you have some memory of what to do (and what not to do)? Finally, you generalize and use this information in other situations or environments. This is how we learn; machines can learn the same way; through experience!

To build “Rodney Jr,” you could start with a PROTOBot as described in a previous chapter. The basic design is two continuous rotation servos sandwiched between two 4” plastic disks from TAP Plastics (you know . . . the fantastic plastic place) controlled by an Arduino Uno and powered by a 9-volt battery; pretty typical setup as far as tabletop robots go. The wheels are from a radio-controlled plane (Lite-Flite). Still, they could easily be one of those Servo Wheels from Parallax or Pololu. You could even screw the servo horns onto jar lids with rubber-band treads or any wheel; just make sure it's not too slippery so you can get traction!

The back half of the robot is made from InstaMorph. If you have yet to use this stuff, you're in for a treat. InstaMorph (or ShapeLock or Friendly Plastic, or other name brands) is a Low-Temperature Thermoplastic. It comes in little beads that you put into water and microwave for a minute or two. It comes out moldable (Hot! but moldable); after it cools, it becomes a hard plastic that can be reheated and remolded if necessary. I used this stuff to glom the Roomba stall sensor onto the rear of the chassis.

The beauty of this project is you can do it with almost any Arduino-based tabletop robot. HomeBrewed, BoeBot, LEGO, mBot, etc . . . you're just adding the Roomba stall detector (although in this application, I prefer to call it a mobility sensor) and monitoring to see whether the robot is traveling forward or backward.



This is a typical IR LED phototransistor pair circuit. If the original iRobot sensor is not available, you can homebrew one. Most any IR emitter/detector pair will do (see Parts List).

The mobility sensor (if you will) in Rodney Jr. is connected to Pin 2. The servo signals lines are connected to Pins 3 and 4. The robot looks at the status of the mobility sensor (Pin 2), either high or low, to determine whether or not the mobility detector is changing. If it is changing from a 1 to a 0, continue the current motion pattern; if not, try another random motion code (0–8), look at the status of the mobility detector (Pin 2) again (either high or low), and determine whether or not the mobility detector is changing again. If it is oscillating, continue the current motion pattern; if not, try another pattern (0–8) randomly, over and over and over.

This can be thought of as the little creature considering change over time. It looks at its current state, applies that to memory, now looks at its current state again, compares that to its past, and does something based on whether or not these two are the same. In the Beta stage, Junior remembers his successful response instead of using random actions and Gamma generalizes those actions.

You'll want to get the Roomba caster wheel with the IR sensor (if only because it comes with the bracket). They're generally available through eBay for about twenty-five dollars. The heart of the Rodney Jr. system is this sensor. However, don't worry if you have the caster wheel with no sensor; most any infrared emitter/phototransistor pair will do. Basically, arrange the emitter and detector to detect the rotation of the caster/encoder wheel as it changes from black to white. You can see this with "serial monitor" in the Arduino IDE. Mine is mountedHomeBrew style (that is, with gaffer's tape).

R1 (the resistor for the emitter) should be adequate to limit the current so you don't kill the LED. Generally, the formula is to subtract your source voltage from the forward voltage drop, in this case, 5V-1.5V = 3.5V (the amount of voltage the resistor needs to drop) and divide that by the continuous current (which in this case is 150mA (a lot of current for a little diode; generally LEDs are ~20mA). If you crunch the numbers, the resistor value comes out to 23.33 ohms. I tried a 100 ohm resistor and could see the IR light; also, it triggered the detector when I rotated the wheel, so what the heck . . . save a little energy... 30 instead of 150mA. R2 (the resistor for the detector) should be the maximum resistance of the detector (that is, when it is in the dark) . . . usually ~10K ohms. The one from Radio Shack measured 13K ohms, so I tried a 10K ohm, and as mentioned above, it worked, so good enough is good enough! Also note that the phototransistor (i.e., the detector)

is reverse biased. That is, the cathode is connected to "+" and the anode "-."

Hurray for Arrays!

If you look at the code, you'll see an array called "MemArray[9][2]" that has nine indices . . . 0–8 (the nine possible motion codes) and two elements (response and confidence). Some more work needs to be done probing these elements to see exactly what Rodney is thinking. As a matter of fact, this is brain surgery! An array is a data structure that can store a fixed-sized collection of elements of the same type. In this case, there are nine motion codes, so we'll have nine indices in which to file and count successful responses to particular motion patterns.

array
https://www.arduino.cc/reference/en/language/variables/data-types/array/

You won't be able to see the IR LED (because it's below red on the infrared spectrum). However, you can view it through a digital camera (i.e., your phone). As far as the software is concerned, the first thing you do is load the Servo library, then initialize your variables (standard Arduino stuff). Set the "mobility_sensor" to digital Pin 2. Initialize the "mobility_read" variable. Initialize "mobility_read_past." Initialize "stall_ticks." Initialize "MotionCode." Initialize "PastMotion." Initialize "Confidence," and finally, initialize the "MemArray." If you look at the sample code https://github.com/cpeavy2/Machine_Intelligence and walk through it, you can literally see what the robot thinks and how it learns.

So the first thing your program should do is read the current state of the mobility sensor; either a "1" or a "0." Then run the wheels a random pattern or "code" . . . look at the mobility

sensor again and see if it's changed . . . if the sensor did not change increment "stall_ticks" one unit until it reaches some threshold (mine is set at "100"). Eventually, it will discover forward or backward (the only two motion codes with linear velocity) and hopefully have some room to maneuver so the mobility sensor oscillates between "0" and "1" (it's moving).

As long as the mobility sensor is spinning, the robot is "happy." You can test this by picking up the robot and spinning the caster wheel . . . whatever motion pattern the wheels are spinning will stay in that pattern until you stop spinning the wheel. You can also view the mobility sensor's pin status with "serial monitor" in the Arduino IDE. It should oscillate between "0" and "1" when you spin the wheel.

Find Relevant Sensor Data

Granted, the "goal" here is contrived, but the process is legitimate. The trick is to find relevant sensor data. To build up a confidence level, memorize successful responses, and finally generalize this information using high confidence Beta responses before reverting to random actions.

Is this serious? Is it real intelligence? Who knows? Who cares? It's a real autonomous cybernetic creature. Have fun exploring the questions, and as Heiserman says, " . . . if someone asks you what your robot does and they don't buy your answer, it's their problem." Rodney is an adaptive machine. More advanced than preprogrammed robots, it has integrated reflexes, and decision-making and goal-setting capabilities. It's not so much what he does but how he does it.

When you learn something that works or moves you toward your goal, you keep doing it. That is the strategy of little

Junior. In Rodney's world, he looks at the mobility sensor, waits a millisecond or two, and then looks again to see if he's moved. If the sensor has changed from black to white (or vice versa), that means the robot is moving back or forth; our definition of success! When the robot has success or achieves its goal (in this case, motion), he needs to recall the previous motion code, so a facility has been created to remember that motion pattern code. That is, the robot transfers its current motion pattern (the variable "MotionCode") and reassigns that value to another variable, "PastMotion" (PastMotion = MotionCode), so if the newly selected random code is successful . . . Junior can remember the motion

The heart and mind of the “Rodney Jr” system is the Roomba mobility sensor... all the robot wants to know is whether it is moving or not. He eventually learns the motion patterns which provide the most magnitude or displacement.

pattern from which it came and increment the confidence level of that response.

Try something, observe the effect; if the effect is positive (that is, it moves you toward your goal), the next time you are in that situation and, of course, desire the same goal, you simply remember what you did. This is the way we work. Why can't robots self-program the same way?

One of the bugs in developing the Rodney code was that it only needed one tick of the mobility sensor for Junior to think he was moving, so sometimes you got false positives and the robot would think that one wheel in one direction or the other (codes 1 or 2 for example) were in-motion-with-magnitude patterns (so to speak). It was interesting as to one motion pattern Left Reverse (2), for example, might have been a false trigger but the robot would nonetheless Left Reverse (2), go forward (4), Left Reverse (2), go forward, over and over and over again so that Junior managed to move around the playpen anyway . . . Rodney finds a way!

To squelch this false trigger I have added a timing loop called "Length" that makes the mobility sensor trigger twice (adjustable) before calling the motion pattern response "success." No real change in the theory; just conditioning the sensor.

I'm also using "randomSeed(analogRead(0));" This is so that Junior's random number is seeded from the floating (neither high nor low) port "0." Otherwise, the numbers would be pseudo random and eventually repeat.

Working with Rodney is like the proverbial box of chocolates . . . you never know what you're going to get. Sure, you've programmed in the code, but it's up to Rodney how he interprets it. Other features I'd like to integrate into Rodney Jr. are battery sensing and automatic charging, light sensor, microphone,

speaker (audio output) . . . maybe make a new goal like following a ball, a line, or a wall and incorporate these things into the learning situation where the robot is keeping track of more variables than just the mobility detector . . . and more outputs than just the drive motors . . . it's not so much programming it but interacting with this intelligent little creature on his own terms.

Rodney Jr. demonstrates Beta level Machine Intelligence
https://youtu.be/R_2WAwZtMCQ

Machine Intelligence as a way of life

Along the way, I encountered another David L. Heiserman fan by the name of Gary Gaulin. He has created a whole theory of life around machine intelligence and the origin of life. His theory is basically that everything is a process of guessing, remembering successful responses, and generalizing those responses. Below, find a link to Gary's blog.

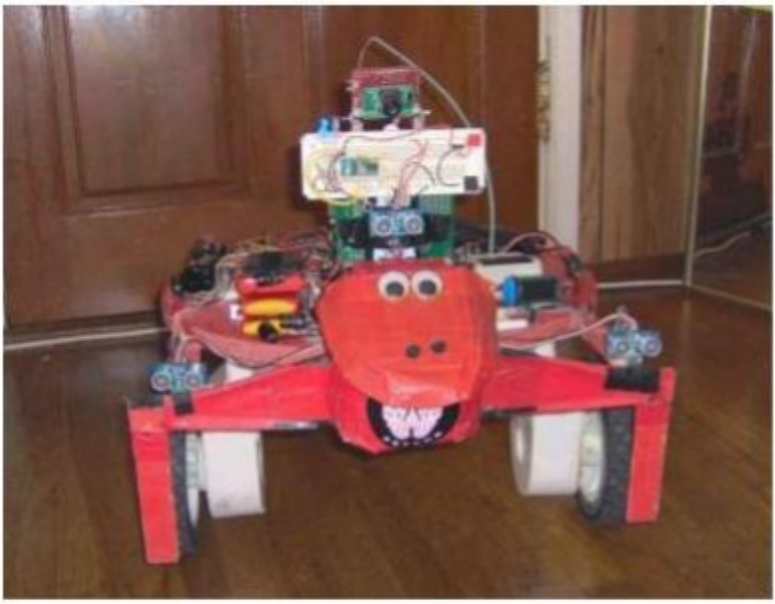
Origin Of Intelligent Life
http://originofintelligentlife.blogspot.com/

The Parts List is at the end of the book. The code can be found here: https://github.com/cpeavy2/Machine_Intelligence

Chapter 4:

RoboMagellan Trials

RoboMagellan is a contest that the Seattle Society of Robotics created in 2004. It is one of the best robotic contests because it gets you off the table and into a full-size robot; plus, it's a useful application! Think of it . . . you can have something delivered within ten to fifteen feet of any spot in the world. GPS is one of the most underrated sensors for mobile robotics.



Rusty the RoboMagellan robot ~2005

In the RoboMagellan contest, a 19" OSHA orange traffic cone is placed on a particular GPS coordinate. The Target Coordinate is announced thirty minutes before the competition. The course is

approximately a hundred yards long (think football field) and conducted on actual urban terrain. That is curbs, trees, benches, people, natural landscapes, etc. Safety is considered by requiring a "dead man's switch." That is a switch you must actively press for the robot to move.

RoboMagellan
http://robogames.net/rules/magellan.php

Blue Dog headed to Goal Cone at RoboGames 2012. Blue Dog took the Bronze medal in RoboMagellen by touching the orange goal Cone in a little over 5-minutes.
Photo credit: Tim Craig

In the first RoboMagellan at RoboGames (2005), I entered a red robot named "Rusty." When building Rusty, there was some chatter on the HomeBrew Club's mailing list about "sine this" and "cosine that," the “curvature of the earth,” etc. I was building Rusty with dead reckoning odometry, which worked terribly—that's when it hit me. RoboMagellan is just an X/Y coordinate problem.

If you want to plot a straight line from one GPS coordinate to another, the key is to compare your GPS coordinates and figure out the direction or bearing. In geometry, this would be the slope of the line.

If the Target Latitude is higher than the Actual Latitude, you must travel North. If your Target Longitude is higher than your Actual Longitude, you must travel East. Now that you know which direction you should go, orient yourself in that direction and go forward!

One of the many cool things about homebrewed robot-building is making robots from other stuff. Please don't ask me what I'm looking for at the hardware store. I'll know it when I see it. I started this project with a PowerWheels ride-on toy but abandoned it because it had only one drive wheel. Instead, I've chosen the Huffy Green Machine; a differential drive ride-on toy.

The Huffy Green Machine 360 Ride-On for Kids makes a good RoboMagellan platform.

This chapter is about building a RoboMagellan robot, as mentioned, a robot designed to go to a particular GPS location and touch an orange traffic cone.

I judged RoboMagellan at RoboGames for twelve years (2005–2017) and competed as well (it is an autonomous competition). My RoboMagellan robot became three different robots over time. It was first named "Rusty." It was red and had a funny face with googly eyes. Some years later, I painted it blue and called it "Blue Dog." In 2012 Blue Dog took the Bronze medal in RoboMagellen by touching the orange Goal Cone in a little over five minutes.

As of late, my old RoboMagellan base has been painted green and named "Frogger." I had it at Burning Man in 2018 with a skeleton (Homer) when the theme was "I Robot." After going to Burningman for six years with robots, I was finally “in theme.”

The system in this project (and my previous RoboMagellan robots) uses two Arduinos. One to read the GPS and the other to control the robot. It uses differential drive, so the two motorized wheels propel and steer the robot.

In RoboMagellen, the basic idea is to get the robot to navigate to a known GPS location and touch a 19" orange traffic "Goal Cone." Optional Bonus Cones are placed along the course, which, if touched, award fractional multipliers, which decrease the robot's overall runtime for a lower score. If you do not touch the Goal Cone, your score is our distance from the Goal. The lowest time (Score) wins. If no one touches the Cone, the robot closest to the Goal wins. RoboMagellan events take on the persona of golf competitions as large groups of people follow the robot like a famous golfer.

I originally started this project with an Ackerman steering Power Wheel but abandoned it because the rear drive train used only one motor, and it couldn't drive over mildly rough terrain.

How to find a GPS Coordinate

RoboMagellan is an X/Y linear equation problem. The path to the destination could be plotted using simple math. Again, here's how it works: if you have a Target Latitude and you know your current location (and you do because there is a GPS onboard), then if the Target Latitude is higher than the Actual Latitude, you need to travel North; if it is lower you need to travel South. If your target longitude is higher than your actual, you need to travel east (keep in mind, west of the prime meridian (USA), the longitude is negative; a larger negative number is lower). If your target longitude is lower than the actual, you need to travel west.

This is Homer riding on Frogger at Burning Man 2018. Homer would dance, and frogger would travel North, avoiding obstacles.

You might need to drive forward a bit to determine your bearing, but the GPS has a compass and will show your direction under "course" or 'bearing." I'm using the Adafruit "Ultimate GPS." First, install the prerequisite "SoftwareSerial library" and "TinyGPS library" into your Arduino IDE. In the Arduino "Examples," there will be a sketch called "Kitchensink," which prints out everything! You can pick the Latitude and Longitude from that program or something similar (gps.location.lat() and gps.location.lat()).

y

The bearing will be expressed in degrees (0°–359°). You'll also need obstacle avoidance sensors; ultrasonic, for example. Lastly, you'll want a bumper for when the robot touches the goal Cone. I've done something interesting with my RoboMagellan robots. I use two separate Arduinos. One collects GPS data and decides what direction the robot needs to go in.

Motor Drive Pin Codes
---------- snip ----------
// 0000 = halt
// 1000 = left forward
// 0100 = left reverse
// 1100 = right forward
// 0010 = forward
// 1010 = counterclockwise
// 0110 = right reverse
// 1001 = clockwise
// 0101 = reverse
// 1101 = veer_left
// 0011 = veer_right
---------- snip ----------

One could use other codes to communicate with the Drive Arduino.

Two Arduinos are better than One

When reading GPS data into the Arduino, there isn't enough bandwidth to step away and take ultrasonic readings and pulse wheels. This is why I dedicated an Arduino to just reading the GPS. We'll call that Arduino1 or the GPS Arduino. It's capable of comparing target and actual locations, figuring out what direction to go, and telling the other Arduino (the Drive Arduino) about it with four wires. The other Arduino (Arduino2) controls motors and senses obstacles and bumps.

Below, find the Arduino1 to Arduino2 pin connections I used:
---------- snip ----------
GPS Arduino -> Drive Arduino
2 -> 10
3 -> 11

```
   4      ->    12
   5      ->    13
---------- snip ----------
```

This could probably be done with one Arduino and interrupts, but it represents my heuristic form of building where you work toward something and discover all sorts of barriers. You work around the barriers to accomplish the goal. First, make it work; then make it work better.

I'll be comparing the Actual Latitude with the Target and the Actual Longitude with the Target. This will determine the direction in which you need to travel. I've split the robot's travel into eight different directions: North, Northeast, East, Southeast, South, Southwest, West, and Northwest. The course or bearing on the GPS will be expressed in degrees. 0° degree is North, 180° degree is South, 90° is East, and 270° is West.

I've included the links below to help explain the GPS Arduino code:
https://github.com/cpeavy2/RoboMagellan/blob/main/GPS_Arduino_final_90_degree_mod.ino

Language Reference
https://www.arduino.cc/reference/en/

If (Control Structure)
https://www.arduino.cc/reference/en/language/structure/control-structure/if/

|| (Logical OR)
https://www.arduino.cc/reference/en/language/structure/boolean-operators/logicalor/

&& (Logical AND)

https://www.arduino.cc/reference/en/language/structure/boolean-operators/logicaland/

So this Huffy Green machine is a differential drive system that will be controlled by two Arduinos and a Roboclaw motor controller. The motor controller I've chosen is the Roboclaw 2x7A.

Motor Controller
https://www.basicmicro.com/Roboclaw-2x7A-Motor-Controller_p_55.html

RoboClaw Downloads - Arduino Library and Examples
https://www.basicmicro.com/downloads

I originally programmed the robot just to go North to keep it simple. After which, I included the component that compared the Latitudes and Longitudes and oriented the robot toward the Goal Cone coordinate.

RoboMagellan Pseudocode
---------- pseudo ----------
// RoboMagellan
Loop()
Set Target Latitude and Longitude.
Read Actual GPS Latitude and Longitude.
Compare Target Latitude with Actual Latitude.
Compare Target Longitude with Actual Longitude.
If the Target Latitude is greater than the Actual Latitude and the Target Longitude is the same as the Actual Longitude, travel north. (0°)
if (Bearing >= 316 or Bearing <= 45) go forward
if (Bearing > 45 and Bearing <= 180) veer_left
if (Bearing < 316 and Bearing > 180) veer_right

If the Target Latitude is greater than the Actual Latitude and the Target Longitude is greater than the Actual Longitude, travel North East. (45°)

if (Bearing >= 0 and Bearing <= 90) go forward
if (Bearing > 90 and Bearing <= 225) veer_left
if (Bearing < 360 and Bearing > 225) veer_right

If the Target Latitude is the same as the Actual Latitude and the Target Longitude is greater than the Actual Longitude, travel East. (90°)

if (Bearing >= 46 and Bearing <= 135) go forward
if (Bearing < 46 or Bearing >= 270) veer_right
if (Bearing > 135 and Bearing < 270) veer_left

If the Target Latitude is less than the Actual Latitude and the Target Longitude is greater than the Actual Longitude, travel Southeast. (135°)

if (Bearing >= 91 and Bearing <= 180) go forward
if (Bearing < 91 or Bearing >= 315) veer_right
if (Bearing > 180 and Bearing < 315) veer_left

If the Target Latitude is less than the Actual Latitude and the Target Longitude is the same as the Actual Longitude, travel South. (180°)

if (Bearing >= 136 and Bearing <= 225) go forward
if (Bearing < 136 and Bearing >= 0) veer_right
if (Bearing > 225 and Bearing <= 359) veer_left

If the Target Latitude is less than the Actual Latitude and the Target Longitude is less than the Actual Longitude, travel South West. (225°)

if (Bearing >= 181 and Bearing <= 270) go forward
if (Bearing < 181 and Bearing >= 45) veer_right
if (Bearing > 270 or Bearing < 45) veer_left

If the Target Latitude is the same as the Actual Latitude and the Target Longitude is less than the Actual Longitude, travel West. (270°)

if (Bearing >= 226 and Bearing <= 315) go forward
if (Bearing < 226 and Bearing >= 90) veer_right
if (Bearing > 315 or Bearing < 90) veer_left

If the Target Latitude is greater than the Actual Latitude and the Target Longitude is less than the Actual Longitude, travel North West. (315°)

if (Bearing >= 271 and Bearing <= 360) go forward
if (Bearing > 0 and Bearing <= 135) veer_left
if (Bearing < 271 and Bearing > 135) veer_right

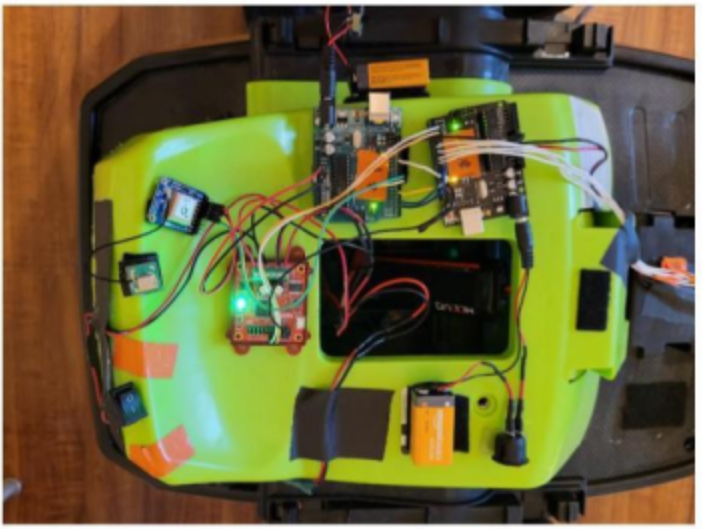
If the Target Latitude is the same as the Actual Latitude and the Target Longitude is the same as the Actual Longitude, Target found, Stop!

Travel toward GPS goal avoiding obstacles using ultrasonic sensors and PixyCam. When the bumper is depressed, you have touched the cone.
Go to Loop()
---------- pseudo ----------

You have to get used to programming two different Arduinos with one computer. Keep track of your Arduino type and com port. To that end, I've built tools to help me along the way.

One little sketch WRITES to the four wires (HIGH or LOW)
https://github.com/cpeavy2/RoboMagellan/blob/main/write_drive_pins.ino

Run the above code through the GPS Arduino (Arduino1). Write another sketch to READ the pins. Run the below code



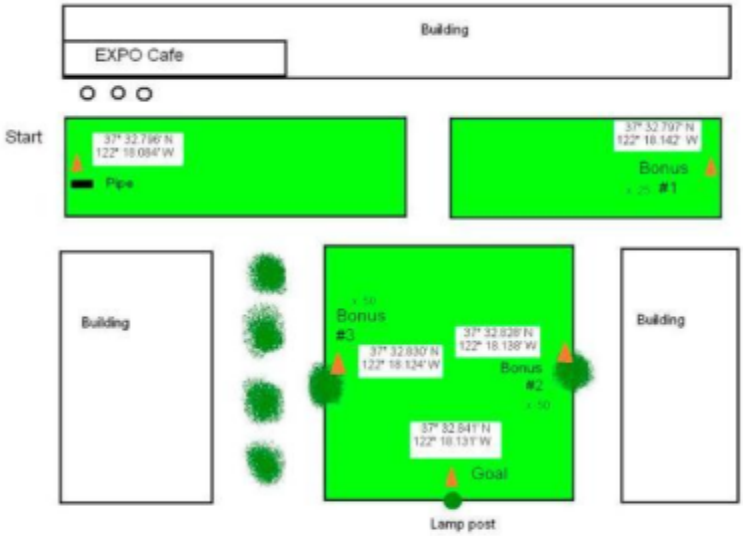
Kerbit with two Arduino Unos (one for GPS, two for motors and sensors), Adafruit "Ultimate GPS" with antenna and RoboClaw 2x15A Motor Controller.

through the Motor Arduino (Arduino2) https://github.com/cpeavy2/RoboMagellan/blob/main/read_drive_pins.ino

This way, you will see Arduino1 write and verify Arduino2 can see it. Step through both boards, writing to the pins from the GPS Arduino and reading from the Drive Arduino. I would get two different color USB cables and consistently plug them into the same port and order. When you plug into one port, it latches to a particular USB port, ACM0, for example, and the next one connects to ACM1. So you'll need to know what which Arduino is plugged into which port.

It takes about two minutes for the GPS to connect, so I power up the two Arduinos and let them sit for a few minutes before switching on the RoboClaw motor driver.

One of the valuable skills you get from building things is troubleshooting, as nothing will work out as you expect. When you have a problem, try breaking it into different parts to track down the problem. For example . . . one wheel's not working like the other . . . it only turns a few seconds and then stops. Switch pins in the Arduino . . . pins 2 to 3. If the problem goes away, it is some electrical or programming issue with the Arduino side of the equation. If, on the other hand, the problem follows the wheel, it's a problem with its controller (ESC) or perhaps setup. Try things! You never know what you might discover. Use the “Serial Monitor” to poke around inside the Arduinos and see what they’re thinking . . . and oh yeah, don’t forget to have fun.

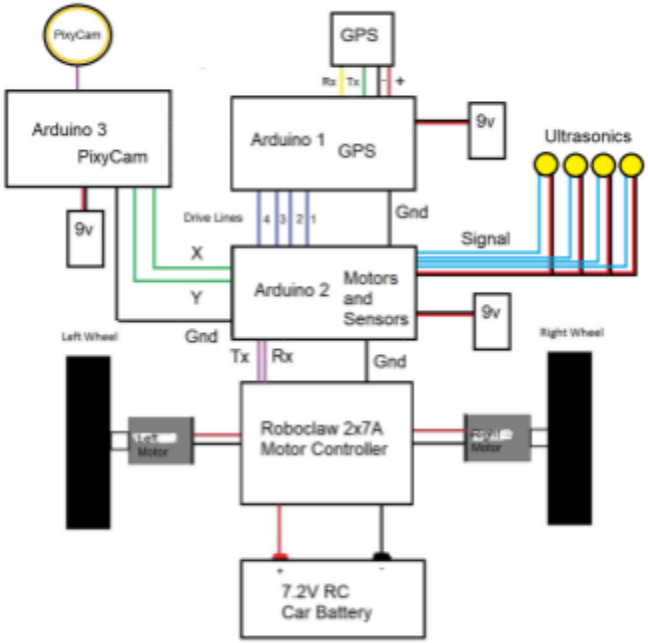


A course map is distributed 30 minutes before the event, and the contestants can walk the course, but not robots!

When you find a problem or a solution, put the system back the way it was and confirm the problem returns. Find your next step and do it! Have a deadline. Give yourself an award or a break when you complete a task. You have to be into this. Work through the most essential element. Ask questions. When troubleshooting, have spare parts remark out (//) questionable statements in your software to understand where the problems lie.

Here are some sample GPS locations (latitude, longitude) that you can use. See how your wheels drive when one is used as the Target coordinate.

New York City: 40.712776, -74.005974
Mexico City: 19.432608, -99.133209
Kansas City: 39.099724, -94.578331
Seattle: 47.606209, -122.332069



Miami: 25.761681, -80.191788

This is Degrees and Decimal Minutes (DMM) notation, where latitude and longitude are expressed in degrees, the minutes are 0–59, and the seconds are described decimally. Run the "Kitchensink." Look at all the data from the GPS. We're interested in three variables: Latitude, Longitude, and Course. When you start modifying functioning code, save often and increment its filename number so when things stop working, you can go back to when they did. You have to want this and want it for a long time. You work at things and figure it out.

Avoiding Obstacles

Hook up ultrasonic sensors as you did with the TABLEBot, except this time, instead of finding objects, you will be avoiding them, which is generally easier to do.

Motors and Ultrasonics Arduino
https://github.com/cpeavy2/RoboMagellan/blob/main/Motor_ultra_Arduino_0822.ino

I used a tabletop robot to develop the Pixy2 portion of the code.

Seeing the Orange Cone

To "see" the Cone, I'm using the CMUCam5 Pixy. When you think about it, this is the apex of the project as everything

beforehand puts you within 15' of the Goal. The camera is what takes you to the Goal.

I developed the camera software on a small tabletop robot; it is basically the same code on a differential drive base. Find the color Orange, center it on the screen, go forward. I'll put a snap-action bump switch on the nose so when he touches the Goal Cone, it triggers, and the robot stops.

I literally taped (gaffer's) the small robot to the nose of KerBit. I then displaced the servos on the small tabletop robot with LEDs so I could see when the left and the right pin was high or low. Use the same pins as the servos. I then jumpered those pins to the motor_aduino so if and when the robot sees OSHA orange the robot meanders in that direction.

I taped the tabletop robot to the big robot and displaced the servo with an LED and jumped it to the drive_arduino.

Arduino Library and API
https://docs.pixycam.com/wiki/doku.php?id=wiki:v2:arduino_api

Color Connected Components
https://docs.pixycam.com/wiki/doku.php?id=wiki:v2:color_connected_components

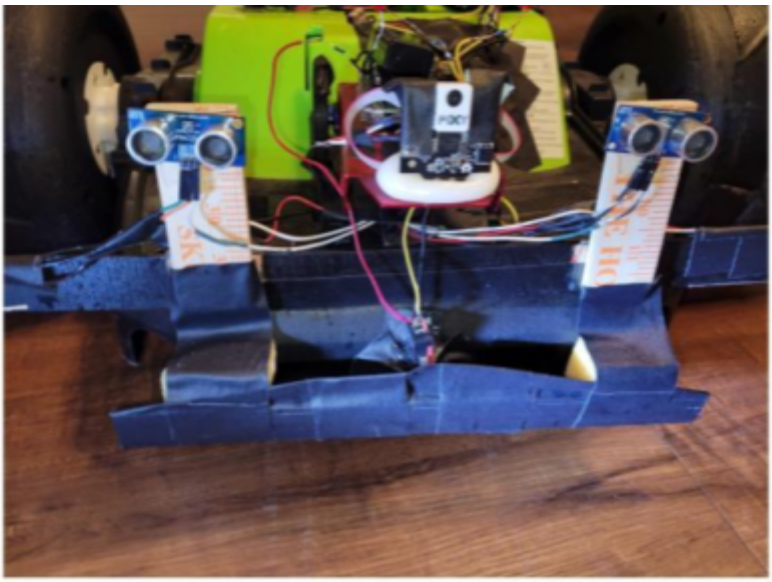
Hooking up Pixy to a Microcontroller (like an Arduino)
https://docs.pixycam.com/wiki/doku.php?id=wiki:v2:hooking_up_pixy_to_a_microcontroller_-28like_an_arduino-29

I developed the Pixy robot code with a tabletop robot like the ones in the PROTOBot and TABLEBot chapters. The core of the code is the below snippet where if the color is detected in the center of the screen, go forward. If the color is detected on the left . . . move right wheel forward. If the color is detected on the right . . . move left wheel forward. This way the robot will continuously track the Orange Goal Cone.

```
------------------------------ snip ------------------------------
{
        Serial.print("Detected ");
        Serial.println(pixy.ccc.blocks[i].m_x);
        if (pixy.ccc.blocks[i].m_x < 200 && pixy.ccc.blocks[i].m_x >
100) {
        forward();
        }
        if (pixy.ccc.blocks[i].m_x > 200) {
        right_forward();
        }
        if (pixy.ccc.blocks[i].m_x < 100) {
        left_forward();
}
}
---------------- snip ----------------
```

The complete KerBit parts list is at the end of the book. The code can be found here:
https://github.com/cpeavy2/RoboMagellan

Finally the switch. This is so that when the robot touches the Orange Goal Cone, it stops. If you knock the cone over, the score doesn't count.



I glued a big snap-action switch to the front of the robot using E6000 and gaffer's tape. I attached a 12" ruler to the switch with gaffer's tape and balanced it with foam rubber from a big sponge. The switch will actuate wherever it is depressed along the ruler (not to be confused with the yardstick I chopped up for the Ping ultrasonic sensors).

I made the bumper with a big snap-action switch, a 12" ruler, glue, and some foam from a big sponge. If you can dream it, you can build it.

This still needs work but I hope to compete KerBit in RoboGames 2023. Check my website for updates: www.camppeavy.com

Chapter 5:

Robot Operating System (ROS)

This chapter will focus on creating an Ubuntu Workstation from an old Windows computer and installing ROS, the standard

Ubuntu workstation created from a Windows machine purchased at Goodwill for $150.

framework for robotics. Ubuntu is a distro (distribution) of Linux.

I learned about ROS (Robot Operating System www.ros.org) shortly after it began in late 2006. Folks at the HomeBrew Robotics Club (www.hbrobotics.org) were early adopters, plus some members worked at Willow Garage, the creators of ROS. As mentioned, ROS is the standard framework

for robotics. It's not really an Operating System but middleware because it facilitates communication between various systems.

At its core, ROS is a publish-and-subscribe architecture with many standardized packages for driving motors, monitoring sensors, mapping, navigating, and visualization. This chapter is based on the humble version of ROS2. Long story short, ROS1 is End Of Life, so unless your project is anchored in ROS1, you should start with ROS2. And if your project is ROS1-based, you should be moving to ROS2.

Willow Garage alumni Melonee Wise and PR2 at RoboGames 2008.

First off, there is no learning curve to ROS. It is a learning cliff. A healthy knowledge of Linux would be good (Ubuntu is a version of Linux), but do not worry. If you work through this and become the least bit productive in ROS, you will learn Linux. You

have been warned! Although ROS2 supports Windows and Macintosh, it is still a Linux platform. And, frankly, when you are playing around with something as prototypish as ROS, it's nice to be on the standard platform; plus, when you add in the Raspberry Pi for robot compute, it becomes a Linux platform.

I didn't fall for ROS until the "Neato" package put together by Mike Ferguson came out in 2010 http://wiki.ros.org/neato_robot. ROS was developed on a $400,000 robot called the PR2 (Personal Robot 2). PR2's claims to fame were that it could plug itself in (an important feature for a mobile robot), it could fetch a beer from the refrigerator (the holy grail of mobile robotics), and it could also fold clothes (twenty minutes per towel, but by the end of the day, the laundry was folded). I've installed Ubuntu on several old Windows computers.

You can create a bootable USB Ubuntu thumb drive from this site: https://ubuntu.com/#download.

I'd recommend Ubuntu 22.04 LTS. The "LTS" stands for Long Term Support, which in this case means five years. This is good timing, as ROS "Humble Hawksbill" was released in April 2022 and also has a five-year LTS designation. These systems (OS and ROS) should be good until 2027. The next chapter (Neato Turtle) will feature a physical robot created from the Neato robot vacuum cleaner.

For the Neato Turtle, you will make another Ubuntu/ROS computer using the Raspberry Pi, but first, you need a Workstation or a remote computer to control the robot and create maps within which you will navigate. Again, the focus of this chapter is to make THAT Workstation, plus there are also some useful tutorials before you get to the physical Neato Botvac, particularly the "turtlesim" exercise.

Two Words: Open Source

As a network administrator at Intuit during the ’90s, I was lucky enough to have been locked in a server room and forced to learn as much Unix as possible. I used a Sun workstation that cost over $8,000 and it was attached to an $80,000 tape backup system. One of the best things about ROS is that it introduced me to Linux. Linux is Unix, except that it is free! I have warned you that there will be plenty of Linux, right?

As of this writing the latest version of ROS.

When I talk with kids, I tell them two words: “Open Source.” Corporate America will not give you an ice cube’s chance in Hell, but with Linux and Open Source software, you can make stuff that corporate America hasn't even considered. Moreover, you can sell it without paying anyone or getting anyone's permission. Technically you are selling your services to support it, but you get the idea. Open Source is THE opportunity of the future!

Creating a ROS Workstation

I've upgraded several old Windows workstations with Ubuntu and rarely have problems. Of course, this is no guarantee of success. If you're looking for a recommendation, I'd recommend the Dell Latitude E5440 or 7490 with the i7 processor if possible, but it'll run okay on an i5 or i3. Use what you have. I see Dell E5440s going for ~$200 on eBay.

First, you'll have to create a bootable USB flash drive. Select "22.04 LTS" under "Ubuntu Desktop" from the aforementioned download page: https://ubuntu.com/#download.

The "iso" (International Standards Organisation) image will download into your downloads folder. This will take a while, so take a break. The file is 3.6GB. Once the downloaded is finished, you'll want to burn it onto a USB thumb drive. Windows 10 has a utility called "Disk Management," but I went around the horn with this utility and had no success. Instead, I downloaded Rufus (https://rufus.ie/en_US/), which worked great.

Basically, you will want to select the Ubuntu iso file (it’s probably in your download folder) as your “Boot selection.” Select your USB drive as the “Device.” BE SURE AND SELECT THE USB DRIVE! YOU DO NOT WANT TO ERASE YOUR

COMPUTER'S HARD DRIVE AT THIS POINT. Copying the ISO file will take about an hour.

Once the thumb drive has been created, you'll want to boot it from the target Workstation (which might be the very Workstation you downloaded the image from). In order to do this, you'll have to make the thumb drive (USB device) your bootup device on the Workstation. To do this you'll have to get into the computer's BIOS or Basic Input/Output System.

With Dell computers, "F2" is the System Setup key. If you're not using a Dell, ask Google. Whatever the key, tap it a couple of times when the keyboard lights up. If you hold the key down, the computer might interpret it as a stuck key. Getting into the BIOs can be fiddly. Try several times tapping the key as the system boots.

As mentioned, your computer might use a different system setup key (F10, F12, F1, Esc or DEL, etc.). Again, Google what system key to press on your computer to get into the BIOS.

Once you're into the BIOS or system setup screen, you'll want to configure it so that it boots off the USB drive. On later-model computers (Windows 10), you may need to go through the UEFI (Unified Extensible Firmware Interface).
Type and search "Change advanced startup options" in the Windows search bar. Navigate to "Troubleshoot" => "Advanced options" => "UEFI Firmware Settings." Sometimes EUFI needs to be enabled. Again, Google is your friend. Search the make and model of your computer and ask how to get into the BIOS and bootup options. For example, "How do I get into BIOS on Lenovo L440." Good luck!

Once you've booted the thumb drive, you can just run Ubuntu from the thumb drive to get a feel for it, or you can overwrite your hard drive with Ubuntu so that it becomes an

Ubuntu Workstation. So when you're ready, boot the USB drive and select "Install Ubuntu." You'll want to connect to your local Wi-Fi network and install all the updates. Depending on your computer and network connection, this will take at least an hour. Once you get your Ubuntu Workstation booted . . . It might be a good time to review some standard Linux commands.

A lot of ROS and Linux is Command-Line Interface (CLI). These commands are typed into a terminal shell known as the BASH (Bourne Again SHell) Shell. This is how you open a terminal window in Linux. You'll be doing this a lot.

Ctrl+Alt+t

After opening the terminal window, let's see what's in there by typing "ls" or list. The "ls" command shows files and directories. Currently in your home folder you should see: Desktop, Documents, Downloads, Music, Pictures, Public, snap Templates, and Videos.

Let's dive into the root directory or "/". Note this is a forward-slash… that is it goes uphill from left to right. To do this we will use the "cd" command or Change Directory.

cd /

Now type "ls" and you'll see files and directories in the root directory.

cd ~

To get back to your "home" or "user" directory, type "cd ~". You can do this from any directory and it will take you back to your home directory. This is where you would keep your personal files.

With networking (and there's lots of networking in ROS), you'll commonly be fiddling with the "hosts" file, which basically associates an IP address with a name. Here's the common format:

```
------------------------------ snip ------------------------------
        sudo vi /etc/hosts
------------------------------ snip ------------------------------
```

sudo vi

This gets you into the "hosts" file as "superuser" (sudo) so that you can edit and save it. "vi" (pronounced "vee-eye") is your text editor. "vi" is very codish. For example:

i = insert
Hit "Esc" to escape insert.
Very important! How does one exit "vi?"
Type ":" for command mode so that you can quit (q), quit, and save (wq), and force quit (q!).

You'll also find yourself editing the .bashrc file (the "." makes it invisible). Type "vi .bashrc" in the home directory.

Here is a guide that you might find useful:

A Beginner's Guide to Editing Text Files With Vi
https://www.howtogeek.com/102468/a-beginners-guide-to-editing-text-files-with-vi/

Below are a few more tips for navigating Linux:

Ctrl+c

A common way to stop or "kill" a running program in a Linux terminal window is Ctrl+c.

Ctrl+Shift+c and Ctrl+Shift+v

You can copy from a terminal window by highlighting text and pressing Ctrl+Shift+c and paste with Ctrl+Shift+v.

Alt+Tab

To switch between windows.

Tab

Also, use the Tab key to finish commands! It will even fill in filenames! Take advantage of shortcuts!

Other useful things in Linux

At the Linux terminal command prompt type “hostname” to see the name of your computer. You can change its name by editing the hostname file.

---------- snip ----------

```
sudo vi /etc/hostname
```

---------- snip ----------

Remember how to save and exit in vi. “:” to get into the command mode and then w, wq, or q! to write, write and quit or just quit.

ifconfig is a useful networking tool. You’ll need to:

---------- snip ----------

```
sudo apt install net-tools
```

---------- snip ----------

May as well install ssh while we’re at it:

---------- snip ----------

```
sudo apt install openssh-server
```

---------- snip ----------

“ssh” (Secure Shell) is used to host a terminal session. It’s text based and makes it like you are typing on a keyboard connected to the remote computer. In the Botvac chapter a Rapberry Pi will be that computer inside the dirtbin.

As mentioned another useful feature of Linux is the manual is all online (man man). If you have questions about any Linux commands, just type “man” followed by the command. For example:

---------- snip ----------

```
man ls
    man mv
    man rm
    man mkdir
```

---------- snip ----------

Here is a Linux Cheatsheet that you might find useful:

Linux Commands Cheat Sheet
https://www.pcwdld.com/linux-commands-cheat-sheet

Do the ROS2 Desktop Install

Once you've created the Ubuntu 22.04 Workstation, it's time to install ROS2. The instructions are here:

Ubuntu (Debian)
https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html

We'll be installing ROS2 via Debian Packages. That means we use the "apt get" command instead of installing it in a source folder (/src) and compiling it. Source has advantages in terms of flexibility, but the Debian install is more straightforward. Carefully

step through the setup and installation instructions. After successfully running the “talker” and “listener” nodes, modify your Bash file (vi .bashrc) to include the environmental variables by using the below command in a terminal window from your home directory. Remember, type “cd ~” from any directory to go to your home directory (/home/user).

```
---------------------------------- snip ----------------------------------
echo 'source /opt/ros/humble/setup.bash' >> ~/.bashrc
---------------------------------- snip ----------------------------------
```

This will make it so that the environmental variables will be properly sourced every time you launch a Bash shell or terminal window. Otherwise, you'd have to execute "source/opt/ros/humble/setup.bash" every time you open a terminal window, and you will be opening a lot of terminal windows in ROS.

Colcon Build

After installing ROS, install "colcon" and other development tools to build ROS packages. Colcon is a build system that allows you to compile code from source (/src). Supplementary packages are likely to be from "source" rather than "Debian." So you will need colcon to compile them.

Development tools:

```
---------------------------------- snip ----------------------------------
        sudo apt install ros-dev-tools
---------------------------------- snip ----------------------------------
```

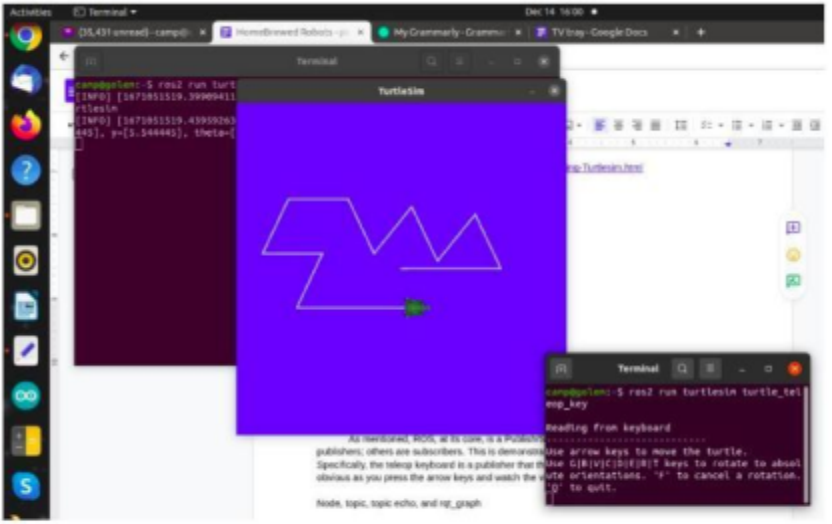
Now that you've installed Ubuntu and ROS plus the development tools, start in on the tutorials.

ROS2 Tutorials

Tutorials
https://docs.ros.org/en/humble/Tutorials.html

In particular, take a look at the "turtlesim."

Introducing turtlesim and rqt
https://docs.ros.org/en/humble/Tutorials/Turtlesim/Introducing-Turtlesim.html



The Turtlesim node (center blue) subscribes to Teleop_key (lower right, foreground), the publisher.

Launch the Turtle Sim and Teleop keyboard nodes:
---------- snip ----------
ros2 run turtlesim turtlesim_node
ros2 run turtlesim turtle_teleop_key
---------- snip ----------

As mentioned, ROS, at its core, is a Publish/Subscribe architecture. Some nodes are publishers; others are subscribers. This is demonstrated clearly with the turtlesim package. Specifically, the teleop keyboard is a publisher that the turtlesim node subscribes. This is obvious as you press the arrow keys and watch the virtual turtle respond to your commands.

Node, topic, topic echo, and rqt_graph

With turtlesim running let's take a look at some ROS command lines. Namely node, topic, topic echo, and rqt_graph. Open a new terminal window and type in the following commands. These commands are helpful when troubleshooting ROS.

- ros2 node list
- ros2 topic list
- ros2 topic echo /turtle1/cmd_vel

The first command, node list, will list the currently running ROS nodes (teleop_turtle and turtlesim). Multiple nodes can be loaded with what's known as a "launch file."

```
---------------------------------- snip ----------------------------------
        user@computer:~$ ros2 node list
        /teleop_turtle
        /turtlesim
---------------------------------- snip ----------------------------------
```

Topic list will list the running ROS topics. Topics are messages that are published or subscribed to by nodes. One of the published topics is "/turtle1/cmd_vel."

```
---------------------------------- snip ----------------------------------
        user@computer:~$ ros2 topic list
        /parameter_events
        /rosout
        /turtle1/cmd_vel
```

```
        /turtle1/color_sensor
        /turtle1/pose
---------------------------------- snip ----------------------------------
```

You can add “echo” into your topic command and the actual message will be printed in the terminal. When you type "ros2 topic echo /turtle1/cmd_vel" nothing will happen until you make the teleop terminal active (click on that window) and start pressing the arrow keys, at which point you'll see the little turtle start moving, and the "ros2 topic echo /turtle1/cmd_vel" window will come to life with x,y,z messages.

Linear vs angular velocity

The "x/y" variables are linear velocities, expressing meters per second while the "z" variable is the angular velocity, expressing radians per second. In other words, when you press the up arrow, the simulated turtle will go forward, and the x variable will register "2 meters per second." When you press the down arrow, the robot turtle goes reverse and the x variable registers "-2 meters per second." This is the linear velocity of the turtle robot going forward and backward.

When you press the left and right arrow keys, you'll see the robot turtle turn clockwise and counterclockwise and the "z" variable register "2 radians per second" and "-2 radians per second." This is the angular velocity or pose of the robot turtle.

```
---------------------------------- snip ----------------------------------
user@computer:~$ ros2 topic echo /turtle1/cmd_vel
linear:
 x: 2.0
 y: 0.0
 z: 0.0
angular:
 x: 0.0
 y: 0.0
```

z: 0.0
---------- snip ----------

The teleop keyboard publishes through the cmd_vel topic. The turtlesim subscribes to this topic. With "ros2 topic echo /turtle1/cmd_vel," you can see this message echoed. This is a useful troubleshooting tool to verify your publisher is actually functioning.

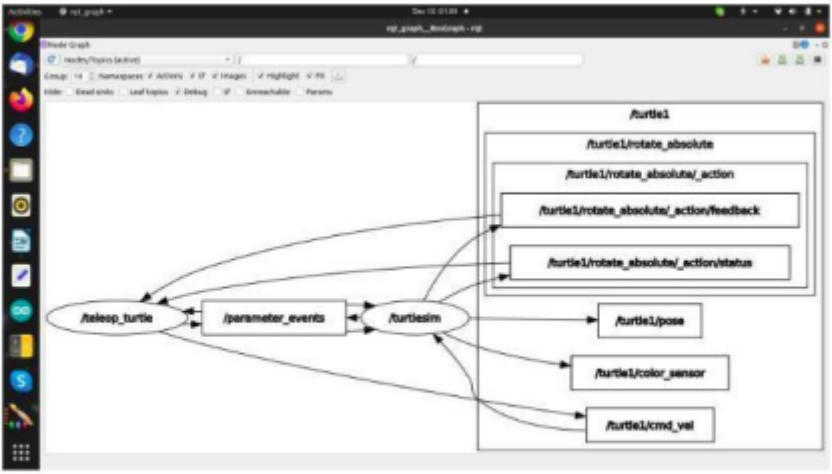
rqt_graph

rqt_graph is another tool that allows you to see the relationship between the nodes and topics.

---------- snip ----------

rqt_graph

---------- snip ----------



rqt_graph allows you to visualize your application's ROS nodes, topics, and the relationship between them.

rqt

You can also access information about nodes, topics, etc (including “rqt_graph” (Plugins > Introspection > Node Graph)) through rqt. It is a graphical form of tools and interfaces in the form of plugins.

---------- snip ----------

rqt

---------- snip ----------

Overview and usage of RQt
https://docs.ros.org/en/humble/Concepts/About-RQt.html

Talker and Listener

In this tutorial, you will create nodes that pass information in the form of string messages to each other over a topic. The example used here is a simple "talker" and "listener" system; one node publishes data, and the other subscribes to the topic so it can receive that data. Talker and Listener demonstrate the essence of the ROS framework. Talker is the publisher, and Listener is the subscriber. Generally, sensors “talk” and actuators “listen” and react in the ROS world.

---------- snip ----------

```
source /opt/ros/humble/setup.bash
ros2 run demo_nodes_py talker
```

---------- snip ----------

In a separate window run “listener.”

---------- snip ----------

```
source /opt/ros/humble/setup.bash
ros2 run demo_nodes_py listener
```

---------- snip ----------

Creating a package
https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-Your-First-ROS2-Package.html

Writing a simple publisher and subscriber (Python)
https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Publisher-And-Subscriber.html

This simple publisher package is the key to using the ROS system. You want to take programs written in Python or C++ and convert them into ROS messages via publishing them. In a later chapter (Smarty Head), we will take a Python ultrasonic package and insert the output into the simple publisher and change the message type from "text" to "range" so the system understands that this is an ultrasonic sensor and recognizes what to do according to its characteristics.

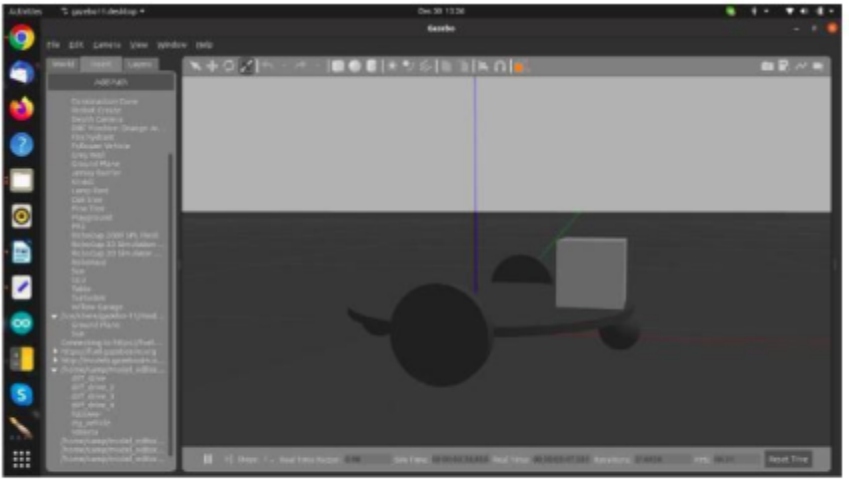
Gazebo

There is also a virtual adaptation of ROS called Gazebo. Here one can run simulations of your robot and test code and environments without actually building the physical robot.

Beginner: Overview
https://classic.gazebosim.org/tutorials?cat=guided_b&tut=guided_b1

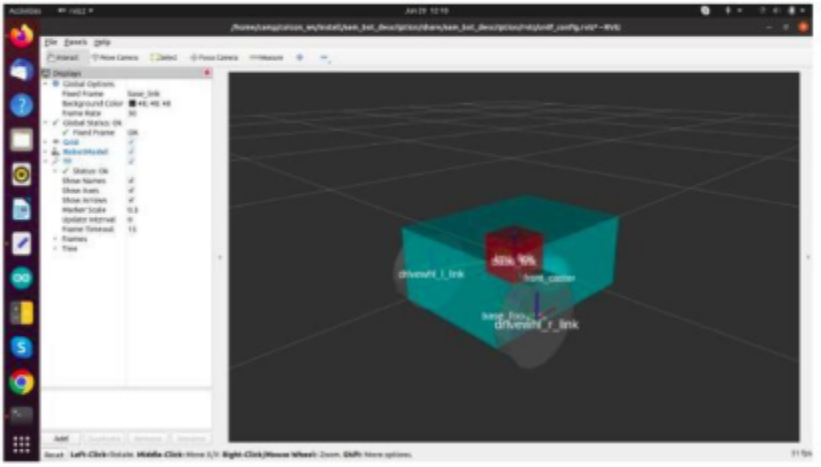
Once installed, this is a great way to get into Gazebo.

Beginner: Model Editor
http://gazebosim.org/tutorials?cat=guided_b&tut=guided_b3



I created this differential drive model with Gazebo's "Model Builder." The box on the front is a depth camera sensor with a "follower" plugin that will make the model move toward anything placed in front of it.

It is great because it teaches you some of the basic concepts and language. Below is a great introduction to URDF (Unified Robot Description Format) and XML (Extensible Markup



This is the "sam_bot." It is part of the ROS2 tutorial that shows you how to create a URDF (Unified Robotics Description Format) for RViz (ROS Visualizer).

Language) and shows the relationship between Gazebo and rviz (ROS Visualizer). Gazebo is the simulator and rviz is the visualizer. Gazebo represents a physical world within which simulations can be run. rviz is the visualizer and it displays the world as the robot sees it. The next chapter features a URDF that models the Neato robot.

Setting Up The URDF
https://navigation.ros.org/setup_guides/urdf/setup_urdf.html

The standard ROS2 rviz model is a robot called the "sam_bot" (ros2 launch sam_bot_description display.launch.py). The tutorial shows how the model is created with XML code.

And finally, the linorobot2 project ties it all together nicely with a virtual and physical identity. It's modular and scalable.

linorobot/linorobot2
https://github.com/linorobot/linorobot2

The last chapter in the book, Looking forward, will focus on the Linorobot project. You now have an Ubuntu Workstation with ROS2 installed. This will be the Workstation used for mapping and navigating with the Neato Turtle.

Chapter 6:

Neato Turtle

This project is built from a robot vacuum cleaner called the Neato Botvac. You could use the earlier Neato model, known as the "XV," but the Botvac is better because the Raspberry Pi (a small single-board computer) fits perfectly into the dirt bin. On the earlier XV model, the mini-USB connector is on the rear of the robot, so you'd have to put the RasPi on top of the lidar dome and run a cable to the connector; on the Botvac, the micro USB is inside the dirt bin.

Neato Botvac D80 with Raspberry Pi in the dirt bin.

Both robots use the same application programming interface (API): https://neatorobotics.com/wp-content/uploads/2020/07/XV-ProgrammersManual-3_1.pdf. The latest models (D8, D9, and D10) do not use this API, but the D3, D4, D6, and D7 do; as do previous models 70, D75, D80, D85, and the XV series. Refurbished Botvacs can be had on eBay for less than $200. First, run it like a vacuum cleaner to ensure it’s fully operational.

The Neato Botvac makes a great experimental ROS platform because of the “lidar” or laser radar. The Botvac features a two-dimensional lidar that spins at 5Hz (five times a second) and takes 360 distance measurements per revolution. This is how the Botvac creates the map within which to navigate.

When you get your Botvac use it as a vacuum cleaner! If for nothing else to check that the system functions properly (battery, motors, encoders, lidar, etc.). If it cleans the floor to your satisfaction all systems are go!

ROS isn't just about running motors and monitoring sensors; well, it is, except that now the sensing and the output can be put toward mapping, navigating, and visualizations, which is a quantum leap beyond bump-and-go robotics. ROS is about shaping behavior at a very high level! The real power behind ROS is that virtually everyone in the robotics business supports it. It might not be the end result but it is definitely the starting place for robot navigation.

The first thing to do using the Ubuntu 22.04 ROS Workstation that you created in the last chapter (ROS) is download the 64-bit Raspberry Pi version of the Ubuntu Desktop 22.04 LTS:

Install Ubuntu on a RasPi (Raspberry Pi).
https://ubuntu.com/download/raspberry-pi

Raspberry Pi version of Ubuntu

Install "Ubuntu Desktop 22.04 LTS." The full Ubuntu Desktop will give you a browser and other graphical applications and tools.

It might take a while to download (2.1GB). When the download finishes, you can find it in the "Download" folder named "ubuntu-22.04-preinstalled-desktop-arm64+raspi.img.xz" **Only download the file once! If you download it again it will be renamed "ubuntu-22.04-preinstalled-desktop-arm64+raspi.img(1).xz" and will not work!**

Insert a microSD card into the computer; you'll probably have to use a miniSD adapter. Use at least a 16GB card. Use Gnome Disk Image Writer to restore the ISO image to the microSD card.

Left-click on the Ubuntu xz image in the download folder, so the options pop down. The default option should be Gnome Image Writer. If not, choose it through "Open with Another Option." Select the microSD card that you've inserted into the computer. BE SURE AND SELECT THE microSD CARD AND NOT YOUR HARD DRIVE! After restoring the image, take the card out of the adapter and put it into the Raspberry Pi (at least a Raspberry Pi 3B).

You'll want to have a monitor, keyboard, and mouse to configure the card for "headless" operation. Headless operation means you can use the Pi without a

MicroSD card with adapter.

monitor (text-based, of course, not graphical). Remember, if your Linux command line skills aren't so good, they will be by the time you complete this project.

Boot the Raspi with microSD Card

Since this is the desktop version of Ubuntu, it simplifies things, including connecting to the network and installing ROS2. As mentioned, you will want a monitor, keyboard and mouse to get

Raspberry Pi with monitor, camera, mouse, keyboard and battery.

started. After which, we will be running the Pi headless in the dirt bin of the Neato Botvac.

The Ubuntu Desktop version will take you through “Twenty Questions.” Connect to the Wifi network. Where are you? Who are you? What is your computer’s name? Choose a Password, etc. REMEMBER YOUR PASSWORD! The system will need to go through all sorts of configurations and applying changes . . . give it a minute. Once the system is up, I would shut it down and bring it up again for good measure.

After the reboot, if the Pi doesn’t automatically ask you to connect to the network, go to the Power symbol (circle with a line through it) in the upper right-hand corner. Select your Wi-Fi network and input the Wi-Fi password.

If you have problems with the Desktop version of Ubuntu it could be because of your monitor. Consider installing the Server version of Ubuntu and editing the “netplan” file as described here:

Ubuntu Server 20.04: Connect to WiFi from command line
https://linuxconfig.org/ubuntu-20-04-connect-to-wifi-from-command-line

You know you are on your wifi network when you see the current date and time. Note the time will be the current local time in UTC (Coordinated Universal Time).

Get the Robot and Workstation Talking to Each Other

Now let’s get our ROS Workstation and the RasPi talking! First, let’s get the IP address of each machine. The IP address (Internet Protocol) is how computers locate and talk with each other on a network. You can use the Linux “ip a” command in a terminal window (Ctrl+Alt+t) to see the IP addresses of the various Network Interfaces. The one you are interested in, the Wi-Fi

adapter is probably named “wlp2s0” or “wlan0.” Regardless, you’d be looking for an IP address that starts with “192.168.” or “10.0.” depending on your router. Write down the IP address of BOTH the ROS Workstation AND the Raspberry Pi. On the RasPi, ping the ROS Workstation . . . type “ping 192.168.x.x,” for example, using the IP address of the Workstation, and type “ping 192.168.x.x” using the IP address of the Raspberry Pi from the ROS Workstations. They should reply to each other in a matter of milliseconds. Congratulations, your robot’s Pi and ROS Workstation are now talking with each other!

Now that you’ve got the Pi and Workstation talking let's understand how these computers are named and how they can refer to each other by name. Linux computers get their name from the “hostname” file, which lives in the /etc. (etcetera) directory. You’ll need superuser privileges to change the name because it is below your home directory.

```
--------------------------------- snip ---------------------------------
        sudo vi /etc/hostname
--------------------------------- snip ---------------------------------
```

Name your computer and Raspberry Pi whatever you like, although they were already named during your setup. Remember to keep it short and sassy so it’s fast and easy to remember. To save the file in vi, type “:” (for command mode) and “wq” to save and quit. If you make a mistake type “q!” to force quit and try again. Reboot for the host names to take effect.

“hosts” file

Now you will want to match the IP address to the computer's name. To do this you edit the “hosts” file, which also lives in the /etc subdirectory. You’ll need to use sudo again as this file is not in your user directory. Again, vi is the standard text editor.

---------- snip ----------

sudo vi /etc/hosts

---------- snip ----------

This makes it easier to communicate between computers. The name is easier to remember than an IP address. Restart for changes to take effect.

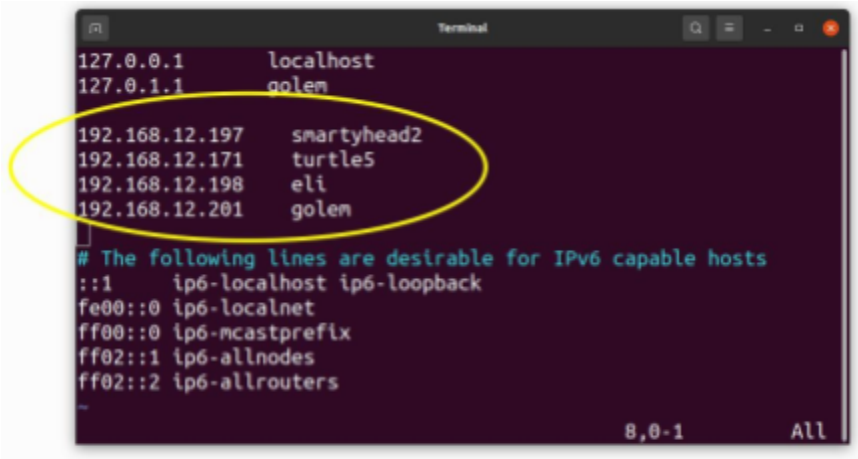
You type the IP address followed by the name of the computer. In my case, the Workstation is called “golem” and the Raspberry Pi “turtle5.”

---------- snip ----------

192.168.12.201 golem

192.168.12.171 turtle5

---------- snip ----------



This is the "hosts" file. Note the IP address is alongside the computer's name. This allows you to refer to the laptop or Pi by name rather than the IP address.

Thereafter, when you type “ping hostname” (hostname being the name of the Pi or Workstation), the reply would be as if you typed the IP address; a lot easier to remember! In my case, the Workstation is called “golem” and the Raspberry Pi “turtle5.” For example, on the Workstation:

---------- snip ----------
ping turtle5
---------- snip ----------

And on the Raspberry Pi:
---------- snip ----------
ping golem
---------- snip ----------

ssh

Now that we can “ping” the Pi via name from the Workstation and vice versa, let's "ssh" (Secure Shell) to your Pi from the Workstation. This will be the first step toward a headless Workstation. With “ssh” you will be able to type on the Pi from your Workstation as the Pi will eventually be in the dirt bin of the Botvac and not have a keyboard or monitor.

If “ssh” isn’t there, you may need to install it with:
---------- snip ----------
sudo apt install openssh-server
---------- snip ----------

Now you should be able to “ssh” to the Pi via either IP address or name like below:
---------- snip ----------
ssh ubuntu@192.168.12.171
or
ssh ubuntu@turtle5
---------- snip ----------

Keep track of the username and the computer’s name as the user trying to log into a computer MUST be a user on that system. Once you've ssh'ed to your Pi, you can work with it headless as if you were typing on the Pi with a keyboard.

arp

Be aware you are using a DHCP (Dynamic Host Control Protocol) IP address, which could change periodically and probably will at the worst possible moment (like when you want to demo). The below command will give your router's IP address (you will need to identify it).

---------- snip ----------

```
arp -a
```

---------- snip ----------

nmap

Plug your router's address into "nmap" to see if you can find the wayward device.

---------- snip ----------

```
sudo nmap -sn 192.168.12.1/24
```

---------- snip ----------

Here's a useful link for finding devices on your network:

How to Find What Devices are Connected to Network in Linux
https://itsfoss.com/how-to-find-what-devices-are-connected-to-network-in-ubuntu/

minicom

This is not necessary but while we're at it, a useful old-school telecommunications program "minicom" would be interesting to install as it will allow you to query the Botvac and run commands directly; drive the wheels, read the lidar scans, etc.

---------- snip ----------

```
sudo apt-get install minicom
```

---------- snip ----------

You'll need to configure it for port ttyACM0. Launch "sudo minicom -s" and configure it for "ttyACM0" through serial port setup.

Here is a webpage that explains how to use Minicom:

Linux / UNIX minicom Serial Communication Program
https://www.cyberciti.biz/tips/connect-soekris-single-board-computer-using-minicom.html

Once connected to the Botvac, type "help." The system will show the commands, including "getchar" and "getldsscan." getchar will display the robot's battery charge level. getldsscan will display the lidar scans. Type "help" followed by any of the commands and it will give you details and options. These commands have been ROSified in the Botvac package. Minicom is not necessary for ROS. Remember the Programmer's Manual?

Programmer's Manual
https://neatorobotics.com/wp-content/uploads/2020/07/XV-ProgrammersManual-3_1.pdf

Now let's install ROS2

Since we were using the Desktop version of Ubuntu and probably still have the monitor, keyboard, and mouse attached, that is you're not already running it headless, you can launch a Foxfire Web Browser (top left-hand corner) on the Pi and copy and paste through the ROS2 instructions:

Ubuntu (Debian)
https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html

Carefully step through all the ROS2 install instructions linked above and do the “Desktop Install.” It will include RViz, demos and tutorials.

---------- snip ----------
sudo apt install ros-humble-desktop
---------- snip ----------

Be sure and set your environment by adding “source /opt/ros/humble/setup.bash” to your .bashrc file. This can be done by typing the below command:
---------- snip ----------
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
---------- snip ----------

You’ll also want to add the developement tools including colcon, ROS’s compiler:
---------- snip ----------
sudo apt install ros-dev-tools
---------- snip ----------

Creating a workspace

Next, you will want to create a workspace for your programs installed from “source.” Compiling from source means that you’ll be assembling a text listing of commands into an executable computer program.

Creating a workspace
https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-A-Workspace/Creating-A-Workspace.html
---------- snip ----------
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
---------- snip ----------

Here we've gone for the default workspace name (ros2_ws). You can name your workspace whatever you want, but I'd suggest going with the standard name to get started.

The ROS2 Botvac Package

A lot of people have built and worked on this package. In 2010, Mike "Fergs" Ferguson (a Willow Garage Engineer) created the Neato Driver, ROS wrapper, and configuration files for running the ROS1 navigation stack. In 2015 Ralph Gnauck and others with the SV-ROS Meetup group, including Greg Maxwell and Dr. Alan Federman (AKA Dr. Bot), added the URDF (Universal Robot Description Format) and launch files (which allows you to launch multiple nodes with one command). In 2020 James Nugen updated the driver and the wrapper to Python 3 to run on ROS2.

In order to use GitHub, you will need to install the app.

---------- snip ----------

```
sudo apt install git
```

---------- snip ----------

Now go to the Botvac GitHub:

cpeavy2/botvac_node
https://github.com/cpeavy2/botvac_node

You will want to install the Botvac package on both the Ubuntu Workstation and Raspberry Pi. First, install these packages:

---------- snip ----------

```
sudo apt install build-essential
sudo apt install ros-humble-xacro
sudo apt install python3-rosdep2
```

---------- snip ----------

Just execute the commands in a terminal window. Be sure and create a workspace and source directory for your ROS2 source builds (mkdir ~/ros2_ws/src/). Check these repos into that workspace/source directory as follows:

---------- snip ----------

```
cd ~/ros2_ws/src
git clone https://github.com/cpeavy2/botvac_node.git
git clone https://github.com/cpeavy2/neato_robot.git
git clone https://github.com/kobuki-base/cmd_vel_mux.git
git clone https://github.com/kobuki-base/kobuki_velocity_smoother
git clone https://github.com/stonier/ecl_tools
```

---------- snip ----------

Go back to the workspace and run “rosdep” which identifies and installs package dependencies.

---------- snip ----------

```
cd ~/ros2_ws/
rosdep update
rosdep install --from-paths src --ignore-src -r -y
```

---------- snip ----------

Install the Navigation2 package on the Ubuntu Workstation only!

On the Workstation go to your workspace source directory (ros2_ws/src) and install Nav2 from source:

---------- snip ----------

```
cd ~/ros2_ws/src
git clone https://github.com/ros-planning/navigation2.git
```

---------- snip ----------

ros-planning/navigation2
https://github.com/ros-planning/navigation2.git

Now go back to your workspace directory and compile the code on both Pi and the Workstation.

```
------------------------- snip ------------------------
        cd ~/ros2_ws
        colcon build
------------------------- snip ------------------------
```

You do not need to install the Navigation2 package on the Raspberry Pi, just on the Ubuntu Workstation because you will be running mapping (slam_toolbox) and navigation (Nav2 and AMCL) on the Workstation. If you get error messages, don't panic. Read the message and try and figure out what it is saying. Perhaps there is another package that needs to be installed, or perhaps it is an environmental problem. Check .bashrc to make sure you have sourced your environment (source /opt/ros/humble/setup.bash). Google the error message to see possible solutions.

Power Pi with battery bank and put into dirt bin on Botvac. Connect Pi to micro USB socket in the dirt bin and make sure the robot is on.

At this point you should have verified the functioning of your Neato Botvac by vacuuming the floor. We shall now make the transformation from floor-care device to educational robot. If you still want a robot vacuum-cleaner… you may need to buy another one.

First remove the brush and

squeegee (rubber blade) to make the robot less resistant. There are numerous websites and videos online that explain how to disassemble the Neato Botvac.

Tutorial: How to disassemble teardown a Neato Botvac
https://youtu.be/BpncgDs3I_k

You will also want to remove the speaker as the robot complains about floor-care things that an educational robot does not care about. Also, pay no attention to the LCD screen as it too is complaining about things relative to floor care. You will be driving the robot with ROS2 through the USB ports. These commands override internal programming regarding the speaker and LCD screen. Note: you will need a long-handled Torx (star) #10 to open the case.

Remove the filter and dirt bin from the robot. Pop the clear plastic dirt bin from the cover with a wide-blade screwdriver. You may need to go from side-to-side. Connect the Pi to the Robot and power the RasPi with a cell phone battery bank. Put everything into the dirt bin and close the cover.

Remove the filter and pop the dirt bin off with a wide-blade screwdriver. Be careful, wedge the blade between the top of the bin and the clear plastic bin… twist and pop. You may need to go from side-to-side. You're breaking the plastic

welds that connect the bin to the cover. When you have the bin off you'll be able to put the RasPi and cell phone battery bank into where the bin was and cover it.

You'll need a short USB type A to Micro USB cable to connect the Pi to the Neato Robot and a short USB-A to USB-C to connect the battery to the Pi4 Power supply.

Connect the Pi to the micro USB socket in the dirt bin. Make sure the robot is on! As mentioned, use as short a micro USB cable as possible. Now power the Raspberry Pi with the cell phone battery bank and put it into the dirt bin of the Botvac.

First "ping" your Pi from the Workstation to verify they are communicating. Then "ssh" to the Raspberry Pi from your Ubuntu Workstation and launch the below command:

---------- snip ----------

```
ros2 launch botvac_node botvac_base.launch.py
```

---------- snip ----------

You should see the Pi connect to the Neato Botvac port ttyACM0. Make sure the Botvac is "on". If it has permission issues, type:

---------- snip ----------

```
sudo adduser user $(stat --format="%G" /dev/ttyACM0 )
```

---------- snip ----------

Next, open another terminal window on the Ubuntu Workstation (not ssh to the Raspberry Pi) and launch the the Nav2 package and slam_toolbox mapper with this long single line command:

---------- snip ----------

```
ros2 launch nav2_bringup bringup_launch.py
use_sim_time:=False autostart:=True
```

map:=/home/user/ros2_ws/src/navigation2/nav2_bringup/maps/map.yaml slam:=True

---------- snip ----------

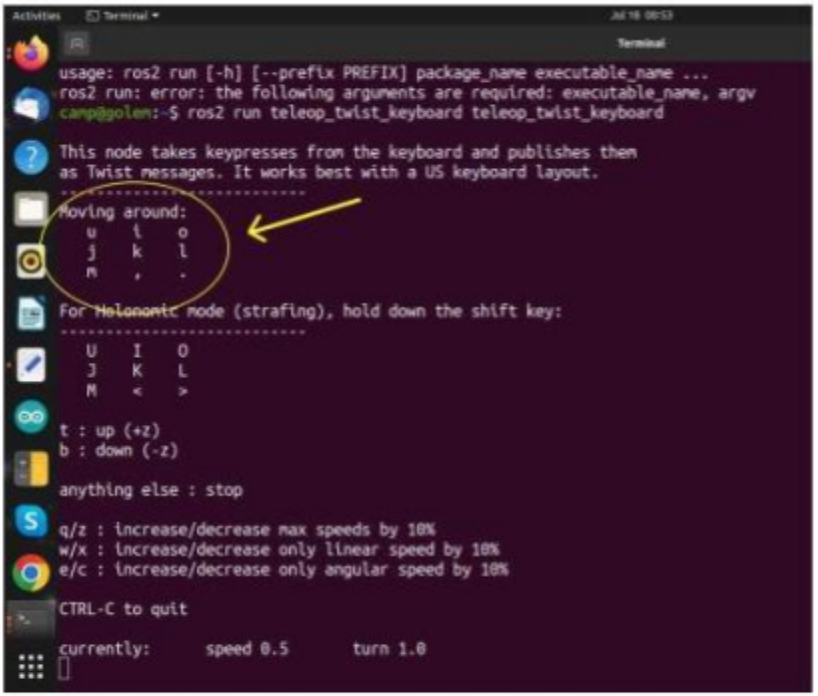
This is a long command (one line). You are launching the “bringup_launch.py” package with Slam Toolbox *on* (slam:=True). Also be sure and use YOUR username in the home directory.

Open another terminal window on the Workstation (not Pi) and launch rviz, the ROS visualizer.

---------- snip ----------

ros2 launch nav2_bringup rviz_launch.py

---------- snip ----------



Teleop keyboard allows you to drive the robot around with your computer.

And finally, open one more Workstation window and launch the teleop keyboard package.

```
---------------------------------------- snip -------------------------------
ros2 run teleop_twist_keyboard teleop_twist_keyboard
---------------------------------------- snip -------------------------------
```

There are also packages that allow you to use standard PC joysticks instead of the keyboard.

Now you should be able to drive the robot around using the keyboard. Press “i” to go forward, “,” to go backward, “l” for clockwise, and “j” for counterclockwise. THE CONTROLLER IS CASE-SENSITIVE!

After you’ve driven around, you will notice the robot has created a map. You will want to save this map, kill slam_toolbox, and load Nav2 with the saved map so that you can navigate around within the static map. The eventual goal is lifelong mapping where the robot continuously maps unknown space. The problem is if you leave on mapping and the robot becomes slightly delocalized, this creates a huge problem due to the compounding effect, long story short . . . it screws up the map.

Again, the concept is to drive around and create a good map, save the map, load the map, and reload Nav2 without slam_toolbox. The robot will then be able to navigate autonomously within the map you created. It can also navigate around obstacles placed within the map.

Stopping Mapping and Starting Navigating!

You should have driven the robot around and seen a map on your Ubuntu Workstation. What we want to do now is save that

map. Open another terminal screen and launch the below command. Note: your directory structure might be different.

---------- snip ----------

```
ros2 run nav2_map_server map_saver_cli -f
~/ros2_ws/src/navigation2/nav2_bringup/maps/map --free 0.196
--ros-args -p save_map_timeout:=5000.0
```

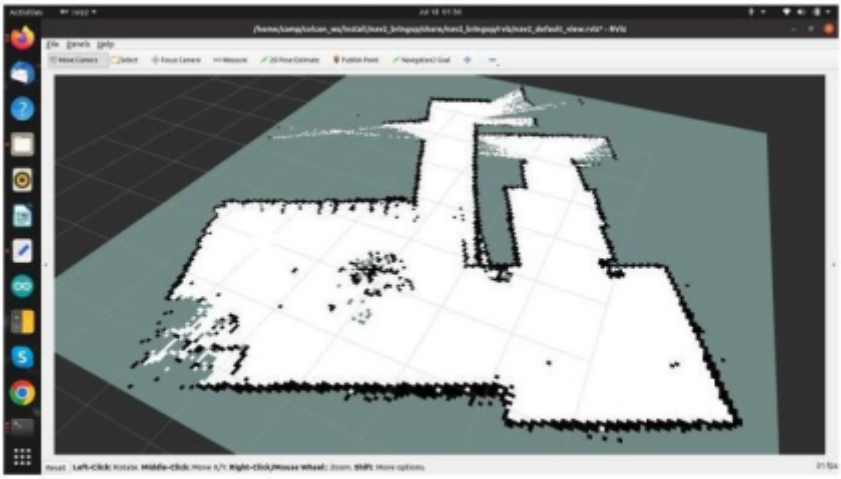
---------- snip ----------

This launches the map saver and saves the map! You might consider saving the map periodically as you go along.

Kill Slam Toolbox!

Now that you've saved the map, you can safely kill the Nav2/slam_toolbox and rviz terminal windows (Ctrl+c).

BE SURE AND SAVE YOUR MAP BEFORE KILLING SLAM_TOOLBOX!



This is the global map. Select 2D Pose Estimate and place the robot on the map.

If you haven't figured it out by now, the proper way to exit the terminal window is with the "exit" command. So when we relaunch Nav2 (without SLAM) and rviz, you can either open new terminal windows or use the ones that are open.

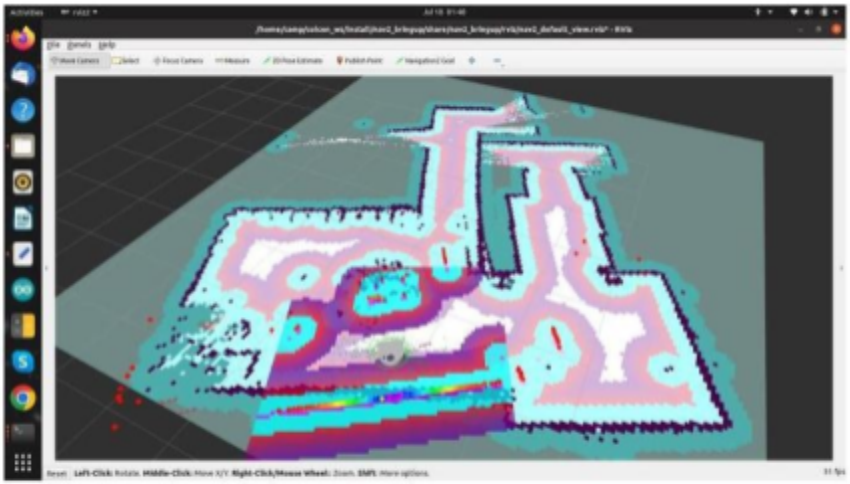
After killing slam_toolbox and rviz, we want to reload Nav2 with the map we just saved. The command is as below (one line). Change the "user" directory to your user name.

---------- snip ----------

```
ros2 launch nav2_bringup bringup_launch.py
use_sim_time:=False autostart:=True
map:=/home/user/ros2_ws/src/navigation2/nav2_bringup/bringup/
maps/map.yaml
```

---------- snip ----------

Note: this is the same command as used above when mapping without "slam:=True". "slam:=True" basically loads slam_toolbox.



Here the robot has been placed on the map and you can see the global and local costmaps.

Also relaunch rviz.

```
--------------------------------- snip ---------------------------------
        ros2 launch nav2_bringup rviz_launch.py
--------------------------------- snip ---------------------------------
```

Now you should have a map that looks just like the one you saved. Click “2D Pose Estimate” and place the robot on the map analogous to where it is physically. You should see the global and local costmaps spring up.

Costmaps are grid maps where each cell is assigned a specific value or cost. It represents the cost (difficulty) of traversing different areas of the map. The different colors represent known space, unknown space, obstacles, and inflation layers.

Now you should be able to select “Navigational Goal” from the menu bar and click any known space on the map, and the robot will autonomously navigate there.

That’s it! You are now mapping and navigating with the Neato Botvac. In the next chapter, we will add a “head,” which features a camera and ultrasonics. The concept is a “Smart Table” or a tray table that can reliably get from one place to another for generic delivery.

When done with the robot, remember to stop the botvac_node with Ctrl+c. This will get the robot out of test mode and stop the lidar from spinning.

Chapter 7:

Smarty Head

I'm obsessed with something I call the "Smart Table." A Smart Table is a tray table that can reliably navigate from one place to another. I think this would be useful in a variety of ways for both home and business. It's not perfect; it can't navigate stairs, open doors, or fetch you a beer unassisted. It's a human assisted delivery system. The payload needs to be loaded by someone in one location, and with a tap on a map (computer or phone) or a verbal command, the robot navigates to the destination to be unloaded, again by a human. It could also help carry things; that is, put what you want on the robot and command it to "follow me."

The original TV tray table.

A TV tray table, TV dinner tray, or personal table is a type of collapsible furniture that functions as a small and easily portable folding table. These small tables were initially designed to be a surface from which one could eat a meal while watching television.

A Tray Table that can Navigate in an Unstructured Environment

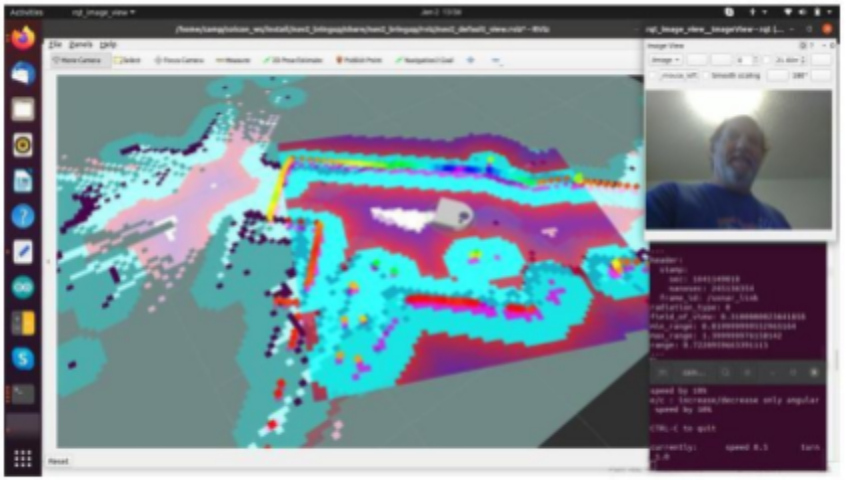
As a matter of factI need my computer to come to me. I need my music to follow me around. I need to look around the house when I am away. In the previous chapter, we made the Neato Turtle mobile base that can map and navigate. To make the Neato Turtle into the Smart Table, we need to add a two-foot-high structure to make the Turtle into a small table. I've used an old TurtleBot superstructure that is made from aluminum standoffs and plywood. You can build the tabletop structure with either aluminum standoffs or

Author Camp Peavy with LocoMo (Crazy Motion), the Smart Table.

wooden dowels. Glue, screw, tape, or use whatever you have. You're basically putting as lightweight a tray table as possible on top of the Botvac. Aim for ~20 inches in total height.

To begin we'll need to make another Raspberry Pi with Ubuntu and ROS2 as we did for the Neato Turtle:

Install Ubuntu on a Raspberry Pi
https://ubuntu.com/download/raspberry-pi



Robot mapping and navigating plus video and ultrasonics.

As before use a monitor, keyboard and mouse to configure the Pi before going headless. Get the second Pi (I'm calling it "smartyhead2") pinging your Workstation and your Workstation pinging "smartyhead" as we did with the Neato turtle. Edit "hostname" and "hosts" in the /etc directory. Use "ip a" to get IP addresses. Edit the "hosts" file so the Pi and the Workstation can ping each other by name. When you get an error message, literally, copy and paste the error message to Google!

ssh again

ssh to “smartyhead”. You probably need to install ssh on “smartyhead”.

```
------------------------------- snip -------------------------------
    sudo apt install openssh-server
------------------------------- snip -------------------------------
```

I told you if you stick with this long, you will learn Linux! Now let's install ROS2 . . . again!

Ubuntu (Debian)
https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html

Carefully step through each of these instructions and do the “Desktop Install” as you did with the Neato Turtle.

```
------------------------------- snip -------------------------------
    sudo apt install ros-humble-desktop
------------------------------- snip -------------------------------
```

Again install ROS’s development tools which include colcon so you can compile from source code:

```
---------------------------------- snip ----------------------------------
    sudo apt install ros-dev-tools
---------------------------------- snip ----------------------------------
```

We’ll be using two Raspberry Pi’s for the Smart Table; one for the Botvac mapping and navigating as outlined in Chapter 6; the other for the Smarty Head camera and ultrasonics.

Remote Viewing

To begin with, we want to put a camera on the second Pi. The standard camera for the Raspberry Pi is the PiCam;

unfortunately, the PiCam is not currently compatible with 64-bit Ubuntu, although I expect it to be shortly. Regardless, after banging my head on this for quite a while (did I mention robots are hard), I've elected to go with a more standard USB webcam, the Logitech C270. I'm sure you can use other webcams, but be aware some could draw quite a bit of current from the USB port,

Camera and ultrasonic sensors being developed on separate Raspberry Pi's.

perhaps requiring a powered hub.

After you've created the Ubuntu 22.04 Raspberry Pi boot disk and booted the Pi (again use a keyboard, monitor, and mouse to get started), install the camera. Shut down the system, and plug the camera into one of the two blue USB 3.0 ports. The 3.0 ports allow more speed and power than the black USB 2.0 ports. Reboot the Pi!

Create your Workspace

Open a terminal window, and create a workspace directory. The workspace directory contains a subdirectory that holds source code. Source code is a plain text list of commands to be compiled into executable code.

```
-------------------------------- snip --------------------------------
        mkdir -p ~/ros2_ws/src
        cd ~/ros2_ws/src
-------------------------------- snip --------------------------------
```

Install ros2_v4l2_camera

```
-------------------------------- snip --------------------------------
        sudo apt-get install ros-humble-v4l2-camera
-------------------------------- snip --------------------------------
```

When playing around with this, I found a way to control the number of frames per second (this changes it from 30 to 5):

```
-------------------------------- snip --------------------------------
        v4l2-ctl -d /dev/video0 -p 5
-------------------------------- snip --------------------------------
```

Now launch the Video for Linux2 (v4l2) package.

```
-------------------------------- snip --------------------------------
        ros2 run v4l2_camera v4l2_camera_node
-------------------------------- snip --------------------------------
```

There are some control problems, but the camera has started.

Open another terminal session and launch “ssh” with the graphics and compression option:

```
-------------------------------- snip --------------------------------
        ssh -YC ubuntu@smartyhead
-------------------------------- snip --------------------------------
```

Install the rqt_viewer:

```
------------------------------- snip -------------------------------
sudo apt-get install ros-${ROS_DISTRO}-rqt-image-view
------------------------------- snip -------------------------------
```

And launch the rqt_image viewer in this window.

```
------------------------------- snip -------------------------------
ros2 run rqt_image_view
------------------------------- snip -------------------------------
```

A video stream from the smartyhead Raspberry Pi should appear. If not, check the Image View drop down menu and select the Image_Raw topic. This will give us remote viewing as we drive the robot around.

The Simple Range Publisher

"Smarty Head" is a Raspberry Pi4 with a camera and ultrasonic sensor. The ultrasonic sensor system is made with a modified ultrasonic Python package and the simple publisher from the ROS2 tutorial:

Writing a simple publisher and subscriber (Python)
https://docs.ros.org/en/foxy/Tutorials/Writing-A-Simple-Py-Publisher-And-Subscriber.html

First, I would suggest you hook up the Raspberry Pi HC-SR04 distance sensor and work through the instructions in "Using a Raspberry Pi distance sensor (ultrasonic sensor HC-SR04)," making it work before you turn it into a ROS2 package.

Using a Raspberry Pi distance sensor (ultrasonic sensor HC-SR04)

https://tutorials-raspberrypi.com/raspberry-pi-ultrasonic-sensor-hc-sr04/

First of all, you should install the Python GPIO library.

```
----------------------- snip -----------------------
        sudo apt install rpi.gpio-common
----------------------- snip -----------------------
```

Make yourself a member of the dialout group!

```
----------------------- snip -----------------------
        sudo adduser "${USER}" dialout
----------------------- snip -----------------------
```

And for good measure . . .

```
----------------------- snip -----------------------
        sudo reboot
----------------------- snip -----------------------
```

You should have already worked through the “Simple Publisher” and “Simple Subscriber” tutorials in the ROS chapter. This would be an excellent opportunity to review.

At the end of the chapter, find the modified Python code. We are modifying the publisher to publish a “range” topic instead of a “string” message. The Simple Publisher package publishes the “Hello World” text message. We have converted that into a ROS2 “range” message that carries the range attributes, including a field of view, minimum range, maximum range, and the actual distance measurement. The distance calculation is changed from the original code as the original code expresses distance in centimeters, and ROS expects the range message to be expressed in meters.

You can find the modified ultrasonic Python package and the modified simple publisher at https://github.com/cpeavy2/Smarty_Head

First, create the "simple publisher" (and subscriber) per instructions and make sure it works. That is you type “ros2 run py_pubsub talker” and it replies, “hello world.”

Then replace "publisher_member_function.py" with the modified version and place the modified "ultrasonic_distance.py" in the same directory. Compile and run "ros2 run py_pubsub talker" to start the publisher and "ros2 topic echo /distance" to see the distance being published as a range topic. If you take a look at the original file and compare it to the modified version, you’ll see the first thing is you import the py_pubsub.ultrasonic_distance as “p.” Then you need to import the Range sensor_msgs.

The String message and message data are remarked out (#) and replaced with rp=p.distance() so that you are publishing a Range message instead of a String message.

This is LocoJo. It uses a TurtleBot 1 superstructure on the Neato Botvac. You can make the same type of chassis with plywood and standoffs.

Code Changes:

```
------------------------------- snip -------------------------------
import rclpy
from rclpy.node import Node

import py_pubsub.ultrasonic_distance as p

from std_msgs.msg import String

from sensor_msgs.msg import Range

class MinimalPublisher(Node):

                def __init__(self):
                super().__init__('minimal_publisher')
                self.publisher_ = self.create_publisher(~~String, 'topic~~
Range, 'distance', 10)
                timer_period = ~~0.5~~ 0.1  # seconds
                self.timer = self.create_timer(timer_period,
self.timer_callback)
                self.i = 0

        def timer_callback(self):
                #msg = String()
                # msg.data = 'Hello World: %d' % self.i
                # self.publisher_.publish(msg)
                # self.get_logger().info('Publishing: "%s"' %
msg.data)
                # self.i += 1
        rg = p.distance()
        r = Range()
        r.header.stamp = self.get_clock().now().to_msg()
        r.header.frame_id = "/sonar_link"
        r.radiation_type = 0
        r.field_of_view = .31
        r.min_range = .02
```

```
        r.max_range = 1.4
        r.range = rg
        self.publisher_.publish(r)

def main(args=None):
        rclpy.init(args=args)

        minimal_publisher = MinimalPublisher()

        rclpy.spin(minimal_publisher)
        # Destroy the node explicitly
        # (optional - otherwise it will be done automatically
        # when the garbage collector destroys the node object)
        minimal_publisher.destroy_node()
        rclpy.shutdown()


if __name__ == '__main__':
        main()
```
---------- snip ----------

Here is a summary of message types available in ROS:

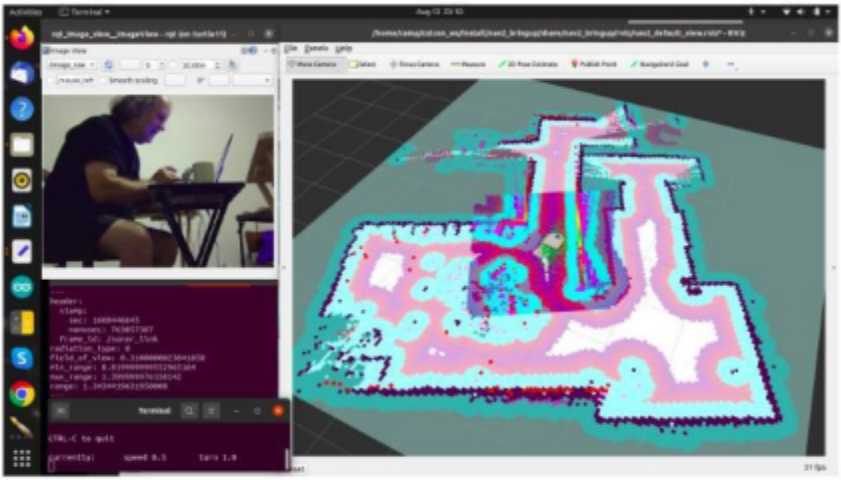
common_msgs
http://wiki.ros.org/common_msgs

Now put the camera and ultrasonics on the tabletop superstructure. Still needs a lot of work, including registering the ultrasonics as an obstacle so that when the robot sees it while navigating, it will path plan around, but it is a starting place for the smart table.

Here is a summary cheatsheet of the command sequence you would use to bring up the Botvac, remote vision, and ultrasonics. Move the windows around on your screen the way you like them. **Change the map location to fit your home directory**

and ROS workspace. Also the username and computer name will be yours.

This is the sequence to bring up the Botvac. “turtle5” is the Raspi inside the dirt bit of the robot. The Pi’s name is “turtle5” and the username “ubuntu.”



After launching the various packages, arrange the windows to create a Robot Dashboard.

---------- snip ----------
ping turtle5
ssh ubuntu@turtle5
ros2 launch botvac_node botvac_base.launch.py
ros2 launch nav2_bringup bringup_launch.py
---------- snip ----------

On the Ubuntu Workstation launch Nav2 with slam_toolbox:
---------- snip ----------
map:=/home/camp/colcon_ws/src/navigation2/nav2_bringup/bringup/maps/map.yaml slam:=True
---------- snip ----------

And launch rviz2 (the ROS Visualizer) again on the Workstation:

```
------------------------------ snip ----------------------------
ros2 launch nav2_bringup rviz_launch.py
------------------------------ snip ----------------------------
```

And finally the Teleop Keyboard to drive the robot around and create the map:

```
------------------------------ snip ----------------------------
ros2 run teleop_twist_keyboard teleop_twist_keyboard
------------------------------ snip ----------------------------
```

Once you've created a nice map (like you did with the Neato Turtle in Chapter 6) save it with the below command:

```
---------------------------------- snip ----------------------------------
ros2 run nav2_map_server map_saver_cli -f
~/ros2_ws/src/navigation2/nav2_bringup/bringup/maps/map --free
0.196 --ros-args -p save_map_timeout:=5000
---------------------------------- snip ----------------------------------
```

At which point you can kill the Nav2 and slam_toolbox terminal (make sure you save the map before killing slam_toolbox!. Then launch the the below command to reload Nav2 and load the map that you just created. Be sure and change to your home directory and ROS workspace.

```
---------------------------------- snip ----------------------------------
ros2 launch nav2_bringup bringup_launch.py
map:=/home/camp/colcon_ws/src/navigation2/nav2_bringup/bring
up/maps/map.yaml
---------------------------------- snip ----------------------------------
```

Place your robot in the map with 2D Pose Estimate. At this point you should see the robot URDF and the lidar scans. You are ready to navigate.

This is the sequence of commands to bringup the remote vision camera. The Raspi attached to the Logitech camera is “smartyhead”.

```
---------------------------------- snip ----------------------------------
ssh ubuntu@turtle14
v4l2-ctl -d /dev/video0 -p 5
ros2 run v4l2_camera v4l2_camera_node
---------------------------------- snip ----------------------------------
```

Open another terminal window and execute the below commands:

```
---------------------------------- snip ----------------------------------
ssh -YC ubuntu@turtle14
ros2 run rqt_image_view
---------------------------------- snip ----------------------------------
```

This should give you a video image from the Smarty Table.

And finally here is the sequence of commands to bringup the ultrasonic sensor. “smartyhead2” is the Raspi connected to the ultrasonic sensor.

```
---------------------------------- snip ----------------------------------
ssh ubuntu@smartyhead2
ros2 run py_pubsub talker
---------------------------------- snip ----------------------------------
```

Open another window to view the /distance toping being published.

```
---------------------------------- snip ----------------------------------
ssh -YC ubuntu@smartyhead2
ros2 topic echo /distance
---------------------------------- snip ----------------------------------
```

These are useful command sequences to clear local costmaps and global costmaps.

```
---------------------------------- snip ----------------------------------
```

ros2 service call /local_costmap/clear_entirely_local_costmap nav2_msgs/srv/ClearEntireCostmap
---------- snip ----------
ros2 service call /global_costmap/clear_entirely_global_costmap nav2_msgs/srv/ClearEntireCostmap
---------- snip ----------

We've done this with two different RaspiPis. The camera and ultrasonics are done with one Pi and the Botvac with another. In the next chapter, I will discuss my future plans using the Linorobot2 package to build a Smart Table with 5" wheels, which I believe will be very helpful in making the table navigate on carpet.

The complete Smarty_Head code can be found here: https://github.com/cpeavy2/Smarty_Head

Chapter 8:

Looking forward . . .

Growing up, some of us were fortunate enough to have the Encyclopedia Britannica. Now, with the smartphone, most kids have all the information in the world in the palm of their hands. For better or worse, we have no idea what this can and will lead to.

SV Magazine - 1976, Steve Jobs and Steve Wozniak, founders of Apple Computer.

Steve Wozniak, the co-founder of Apple Inc., was just building the computer he wanted. He and Steve Jobs famously showed off their homebrewed computer at Silicon Valley's Homebrew Computer Club in 1976. Mobile robots are computers

with eyes, arms, and legs. Ultimately, I hope this book will help and inspire you to build the robot you want. That is the essence of HomeBrewed Robots; building the robot you want.

Is it worth the trouble? I've observed that many older folks (mainly men) are involved in homebrewed robot building. Even when I was in my thirties, I noticed that, for the most part, the HBRC was made up of old, gray-haired guys, not to say women can't be involved too. I figured this bodes well for doing this late in life. We're decades later into the twenty-first century, and it is high time for the mobile robot revolution! The tools and technology are better than ever. Short of family and friends, I can think of no better place personally and professionally for one to spend time than robot-building. So much potential and so much fun!

Robotics is an art whose technology is quickly ripening. We're in uncharted territory creating the blueprint as we go along; the golden age is still decades away. What a marvelous time, a blank canvas. Robotics is a physical medium, a logical extension of the Internet. It's a medium where you reach out, touch, and it pushes back. It's a medium that encompasses all that is humanity and, ultimately, a medium that will be our progeny.

If you're involved or want to be involved with robotics, it is important to interact with like-minded individuals. You can find them everywhere, Facebook, LinkedIn, etc. Look to find a local club or start one! Join us at the HomeBrew Robotics Club! We always have Show-and-Tell.

HomeBrew Robotics Club
www.hbrobotics.org

As I wrap this up, I want to start again. My vision is the smart table and the way there is through Linorobot2. I do believe the next step is carpet navigation, which I think can be done better

with 5” wheels. In other words, a vacuum-cleaning base isn’t going to do it. “Talino” means intelligence in Tagalog.

Linorobot2

linorobot/linorobot2
https://github.com/linorobot/linorobot2

Linorobot in Gazebo; ROS's 3D simulator.

Here are the sequence of commands to install Linorobot2

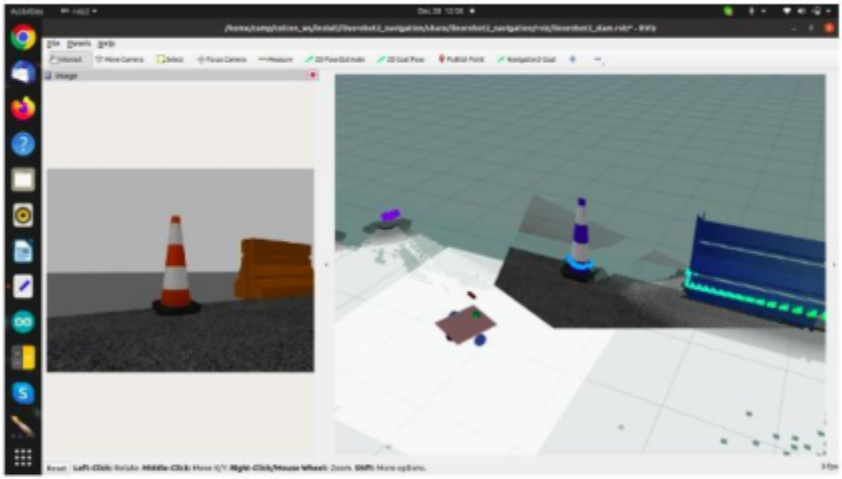
---------- snip ----------

```
source /opt/ros/<ros_distro>/setup.bash
cd /tmp

wget
https://raw.githubusercontent.com/linorobot/linorobot2/${ROS_DIS
TRO}/install_linorobot2.bash
bash install_linorobot2.bash <robot_type> <laser_sensor>
<depth_sensor>
source ~/.bashrc
```

---------- snip ----------

Note: You will need to define your robot_type, laser_sensor, and depth_sensor. Look at the Linorobot2 GitHub for options. I am choosing 2wd, xv11, and realsense.



RViz is the ROS Visualizer. It represents how the robot sees the world.

And these commands to install the Gazebo world so you can run your Linorobot virtually on your computer.

---------- snip ----------

```
cd <host_machine_ws>
git clone -b $ROS_DISTRO https://github.com/linorobot/linorobot2 src/linorobot2
rosdep update && rosdep install --from-path src --ignore-src -y --skip-keys microxrcedds_agent --skip-keys micro_ros_agent
colcon build
source install/setup.bash
```

---------- snip ----------

Linorobot is a great framework from Gazebo to a physical robot. It is modular and scalable. It is relatively easy to drill down

into the URDF and change the size of the wheels, the platform, and the location of sensors. In other words, you're building the robot you want.

Go to the URDF directory:

---------- snip ----------

~/<host_machine_ws>/src/linorobot2/linorobot2_description/urdf

---------- snip ----------

In the urdf folder you'll see the configuration files including: 2wd_properties.urdf.xacro. Open "2wd_properties.urdf.xacro" and you'll see the stock base_length at 0.24 meters (~10") and wheel_radius at .275 meters (~3").

2wd_properties.urdf.xacro

---------- snip ----------

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
 <xacro:property name="base_length" value="0.24" />
 <xacro:property name="base_width" value="0.275" />
 <xacro:property name="base_height" value="0.003" />
 <xacro:property name="base_mass" value="1" />
 <xacro:property name="wheel_radius" value="0.045" />
 <xacro:property name="wheel_width" value="0.01" />
 <xacro:property name="wheel_pos_x" value="0.0" />
  <xacro:property name="wheel_pos_y" value="0.13" />
 <xacro:property name="wheel_pos_z" value="-0.03" />
 <xacro:property name="wheel_mass" value=".1" />
 <xacro:property name="wheel_torque" value="20" />
 <xacro:property name="front_caster_wheel" value="true" />
 <xacro:property name="rear_caster_wheel" value="true" />
 <xacro:property name="laser_pose">
        <origin xyz="0.12 0 0.33" rpy="0 0 0"/>
 </xacro:property>
 <xacro:property name="depth_sensor_pose">
```

```
        <origin xyz="0.14 0.0 0.045" rpy="0 0 0"/>
 </xacro:property>
</robot>
---------------------------------- snip ----------------------------------
```

It's relatively easy to change the base_length value to 0.50 and the base_width value to 0.50 so you have a ~20" base (.5 meters).

```
---------------------------------- snip ----------------------------------
<xacro:property name="base_length" value="0.50" />
<xacro:property name="base_width" value="0.50" />
---------------------------------- snip ----------------------------------
```

Change your wheel_radius from 0.045 to 0.0635 and you have 5" wheels.

```
---------------------------------- snip ----------------------------------
<xacro:property name="wheel_radius" value="0.0635" />
---------------------------------- snip ----------------------------------
```

I also flipped the location of the camera and the lidar. I prefer the camera up high and the lidar down low.

```
---------------------------------- snip ----------------------------------
<xacro:property name="depth_sensor_pose">
                <origin xyz="0.12 0 0.33" rpy="0 0 0"/>
</xacro:property>
<xacro:property name="laser_pose">
                <origin xyz="0.14 0.0 0.080" rpy="0 0 0"/>
</xacro:property>
---------------------------------- snip ----------------------------------
```

Of course virtual robots are fun but really? Linorobot has standard robot hardware including standard motor drivers, IMUs (Inertial Measurement Unit), lidars, and cameras.: https://github.com/linorobot/linorobot2_hardware

The following list of commands will launch gazebo, launch the nav2 stack with mapping, launch rviz, and lastly teleop keyboard:

```
---------------------------------- snip ----------------------------------
ros2 launch linorobot2_gazebo gazebo.launch.py
---------------------------------- snip ----------------------------------
ros2 launch linorobot2_navigation slam.launch.py rviz:=true sim:=true
---------------------------------- snip ----------------------------------
ros2 launch nav2_bringup navigation_launch.py
---------------------------------- snip ----------------------------------
ros2 run teleop_twist_keyboard teleop_twist_keyboard
---------------------------------- snip ----------------------------------
```

You should now be able to drive around, create a map, and navigate within the map.

We are combining the mechanics of the industrial age with the electronics of the information age and making intelligent machines. This is a never-ending story. The end is the beginning. Enjoy the journey. Keep building those HomeBrewed Robots!

Parts List

PROTOBot:

Continuous rotation Servos:
https://www.robotshop.com/en/parallax-futaba-continuous-rotation-servo.html

Arduino Uno:
https://www.amazon.com/Arduino-A000066-ARDUINO-UNO-R3/dp/B008GRTSV6/ref=asc_df_B008GRTSV6/

22 AWG solid core wire
https://www.amazon.com/TUOFENG-Hookup-Wires-6-Different-Colored/dp/B07TX6BX47

4-inch Acrylic discs:
https://www.amazon.com/dp/B09VTD852M/ref=sspa_dk_detail_2

Instamorph:
https://www.instamorph.com/

Popsicle Sticks
https://www.amazon.com/Colored-Popsicle-Sticks-Crafts-Multi-Purpose/dp/B09SGRMVMQ/

Snap Action Switches
https://www.amazon.com/gp/product/B07P4CJ8TV/ref=ppx_yo_dt_b_asin_title_o03_s00

Rechargeable 9V batteries:
https://www.amazon.com/EBL-USB-Rechargeable-Lithium-Batteries/dp/B086GTFGPP/

TABLEBot:

IR Infrared Sensor Module
https://www.amazon.com/Gikfun-avoidance-Reflective-Photoelectric-Intensity/dp/B07FJLMLVZ/ref=asc_df_B07FJLMLVZ/

Hot Glue Gun:
https://surebonder.com/collections/glue-guns

PING))) Ultrasonic Distance Sensor:
https://www.parallax.com/product/ping-ultrasonic-distance-sensor/

snap-action switch:
https://www.amazon.com/gp/product/B07P4CJ8TV/

Alien Tape:
https://www.amazon.com/Alientape-Multipurpose-Removable-Adhesive-Transparent/dp/B09H2RXXD9/

Gaffer's tape:
https://www.amazon.com/dp/B07K1WW8WW

E6000
https://www.amazon.com/gp/product/B004O7EXNU/

5-Minute Epoxy
https://www.amazon.com/dp/B0002BBV46

Machine Intelligence:

Roomba Caster Wheel:
https://www.ebay.com/sch/i.html?_from=R40&_trksid=p2380057.m570.l1313&_nkw=Roomba+caster+wheel&_sacat=0

940nm LEDs Infrared Emitter and IR Receiver
https://www.amazon.com/dp/B01HGIQ8NG/ref=sspa_dk_detail_6

RoboMagellan Trials:

Green Machine 360, 6V
https://www.huffybikes.com/green-machine-360-6v-19998/

Adafruit Ultimate GPS:
https://www.adafruit.com/product/746

Motor Controller
https://www.basicmicro.com/Roboclaw-2x7A-Motor-Controller_p_55.html

Charmed Labs Pixy2 Smart Vision Sensor
https://www.amazon.com/gp/product/B07D1CLYD2/ref=as_li_tl

Neato Turtle:

Neato Botvac
https://www.ebay.com/sch/i.html?_from=R40&_trksid=p2380057.m570.l1313&_nkw=neato+botvac&_sacat=0

1’ MicroUSB cable
https://www.amazon.com/Android-Compatible-Smartphones-Charging-Stations/dp/B095JZSHXQ/

1’ USB-C cable
https://www.amazon.com/etguuds-Charging-Charger-Braided-Compatible/dp/B08933P982/

Power Bank External Battery
https://www.amazon.com/dp/B09Z6R5XZ7/

Torx T10H T10 Tamper Proof Security Screwdriver
https://console5.com/store/torx-t10h-t10-tamper-proof-security-screwdriver.html

Tools:

Multimeter
https://www.amazon.com/dp/B07SHLS639

Safety Glasses
https://www.mcmaster.com/magnifying-glasses/safety-glasses-with-magnifiers-7/

Soldering Iron (includes hot air tool)
https://www.amazon.com/gp/product/B000ICBX8S/

Wire stripper
https://www.jensentools.com/greenlee-pa1118-stripper-wire-progrip-20-ergonomic-30-20-awg/p/618pl1118

Electrical Tape
https://www.amazon.com/3M-35-PACK-Scotch-Electrical-Packs/dp/B000PHGM14/

Solder
https://www.amazon.com/KESTER-SOLDER-32117-24-6040-0027-Diameter/dp/B00068IJPO/

Battery tester
https://www.amazon.com/D-FantiX-Battery-Universal-Checker-Batteries/dp/B014FEM0X6/

Heat Shrink Tubing
https://www.amazon.com/560PCS-Heat-Shrink-Tubing-Eventronic/dp/B072PCQ2LW/

References:

Neato Robotics
https://neatorobotics.com/

HBRC Challenge
http://www.hbrobotics.org/robot-challanges/

RoboGames
http://robogames.net/index.php

Arduino IDE
https://www.arduino.cc/en/Main/Software

Arduino Tutorial
https://www.tutorialspoint.com/arduino/index.htm

delay()
https://www.arduino.cc/reference/en/language/functions/time/delay/

Servo
https://www.arduino.cc/reference/en/libraries/servo/

PROTOBot Code
https://github.com/cpeavy2/PROTOBot

while
https://www.arduino.cc/reference/en/language/structure/control-structure/while/

Phase I TABLEBot Code
https://github.com/cpeavy2/TABLEBOT_I

Ping Ultrasonic Range Finder
https://www.arduino.cc/en/Tutorial/BuiltInExamples/Ping

The HomeBrew Robotics Club mailing list
https://groups.google.com/g/hbrobotics

Phase II TABLEBot Code
https://github.com/cpeavy2/TABLEBot_II

Phase III TABLEBot Code
https://github.com/cpeavy2/TABLEBot_III

Rodney Machine Intelligence Code
https://github.com/cpeavy2/Machine_Intelligence

array
https://www.arduino.cc/reference/en/language/variables/data-types/array/

Rodney Jr. Beta level Machine Intelligence
https://youtu.be/R_2WAwZtMCQ

Origin Of Intelligent Life
http://originofintelligentlife.blogspot.com/

RoboMagellan
http://robogames.net/rules/magellan.php

KerBit GPS RoboMagellan Code
http://robogames.net/rules/magellan.php

Arduino Language Reference
https://www.arduino.cc/reference/en/

If (Control Structure)
https://www.arduino.cc/reference/en/language/structure/control-structure/if/

|| (Logical OR)
https://www.arduino.cc/reference/en/language/structure/boolean-operators/logicalor/

&& (Logical AND)
https://www.arduino.cc/reference/en/language/structure/boolean-operators/logicaland/

RoboClaw Downloads
https://www.basicmicro.com/downloads

Write Drive Pins
https://github.com/cpeavy2/RoboMagellan/blob/main/write_drive_pins.ino

Read Drive Pins
https://github.com/cpeavy2/RoboMagellan/blob/main/read_drive_pins.ino

Motors and Ultrasonics Arduino
https://github.com/cpeavy2/RoboMagellan/blob/main/Motor_ultra_Arduino_0822.ino

Arduino Library and API
https://docs.pixycam.com/wiki/doku.php?id=wiki:v2:arduino_api

Color Connected Components
https://docs.pixycam.com/wiki/doku.php?id=wiki:v2:color_connected_components

Hooking up Pixy to a Microcontroller (like an Arduino)
https://docs.pixycam.com/wiki/doku.php?id=wiki:v2:hooking_up_pixy_to_a_microcontroller_-28like_an_arduino-29

KerBit RoboMagellan Code
https://github.com/cpeavy2/RoboMagellan

Create Ubuntu USB Thumb Drive
https://ubuntu.com/#download

Linux Commands Cheat Sheet
https://www.pcwdld.com/linux-commands-cheat-sheet

ROS Install on Ubuntu (Debian)
https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html

Colcon Installation
https://colcon.readthedocs.io/en/released/user/installation.html

ROS2 Tutorials
https://docs.ros.org/en/humble/Tutorials.html

Using turtlesim and rqt
https://docs.ros.org/en/humble/Tutorials/Turtlesim/Introducing-Turtlesim.html

rqt_graph
https://docs.ros.org/en/humble/Concepts/About-RQt.html

Creating a ROS2 package
https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-Your-First-ROS2-Package.html

Gazebo Beginner: Overview
https://classic.gazebosim.org/tutorials?cat=guided_b&tut=guided_b1

Beginner: Model Editor
http://gazebosim.org/tutorials?cat=guided_b&tut=guided_b3

Setting Up The URDF
https://navigation.ros.org/setup_guides/urdf/setup_urdf.html

Neato Programmers Manual
https://help.neatorobotics.com/wp-content/uploads/2020/07/XV-ProgrammersManual-3_1.pdf

Install Ubuntu on a RasPi (Raspberry Pi).
https://ubuntu.com/download/raspberry-pi

A Beginner's Guide to Editing Text Files With Vi
https://www.howtogeek.com/102468/a-beginners-guide-to-editing-text-files-with-vi/

How to Find What Devices are Connected to Network in Linux
https://itsfoss.com/how-to-find-what-devices-are-connected-to-network-in-ubuntu/

Linux / UNIX minicom Serial Communication Program
https://www.cyberciti.biz/tips/connect-soekris-single-board-computer-using-minicom.html

Creating a workspace
https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-A-Workspace/Creating-A-Workspace.html

cpeavy2/botvac_node
https://github.com/cpeavy2/botvac_node

ros-planning/navigation2
https://github.com/ros-planning/navigation2.git

Writing a simple publisher and subscriber (Python)
https://docs.ros.org/en/foxy/Tutorials/Writing-A-Simple-Py-Publisher-And-Subscriber.html

Using a Raspberry Pi distance sensor (ultrasonic sensor HC-SR04
https://tutorials-raspberrypi.com/raspberry-pi-ultrasonic-sensor-hc-sr04/

Modified Ultrasonic Python package and Modified Simple Publisher
https://github.com/cpeavy2/Smarty_Head

Using a Raspberry Pi distance sensor (ultrasonic sensor HC-SR04
https://tutorials-raspberrypi.com/raspberry-pi-ultrasonic-sensor-hc-sr04/

common_msgs
http://wiki.ros.org/common_msgs

linorobot/linorobot2
https://github.com/linorobot/linorobot2

Linorobot2 Hardware
https://github.com/linorobot/linorobot2_hardware