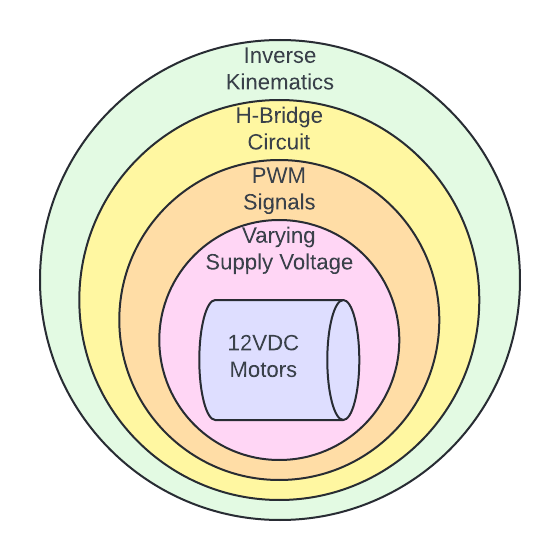
**Lesson 3 – Basic Motion**

In this lesson we’ll spend a bit less time on theory than before and we'll finally begin writing some code. First, we’ll discuss the layers of abstraction behind robot motor control, then we’ll set up a GitHub repository for our robot code, and finally we’ll write a basic motion script for our robot and drive it around!

**Layers of Abstraction**

Often, the best way to solve a problem is to divide it into many smaller problems you can tackle one at a time, and that's exactly how we’ll approach controlling our motors. An Arduino microcontroller is a digital device that interacts with the world by turning 5V pins on and off, but we control motor speed by varying the supply voltage the motors receive. It can be difficult to bridge the gap between these two systems, but we can break it down into a few *layers of abstraction* to simplify the problem.



**Figure 3-1:** Layers of abstraction involved in motion control

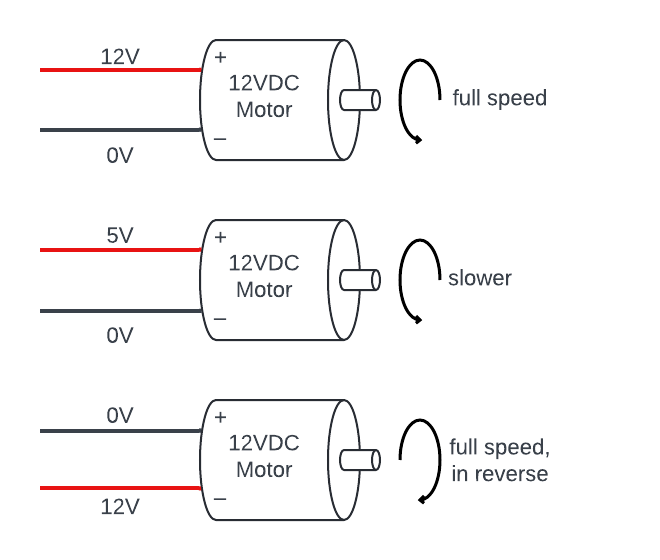
We’ll talk about each of these layers in succession and use them to solve the problem of getting our robot to move.

**Supply Voltage and DC Motors**

There are two common types of DC motors: brushed and brushless. Brushless motors (BLDC motors) are excellent for high-speed applications and produce less electromagnetic noise but are more complicated to operate, and neither of these advantages are necessary for our rover. For those reasons, we’ll stick to the humble and reliable brushed DC motor (BDC motor). You can learn [how a brushed DC motor works here](https://en.wikipedia.org/wiki/Brushed_DC_electric_motor), but we’ll focus on how we control the speed and direction of the motor.

Controlling the speed of a BDC motor is extremely simple. We have 12VDC motors, which means our motors are designed to run at full speed when 12V is applied to them. If we apply more than 12V we may break our motors, but if we apply less, they’ll just run slower. That’s how we control the speed of our motors: we vary the voltage between 0V and 12V.

Controlling the direction of a BDC motor is also easy. A BDC motor just wants there to be some sort of voltage across its terminals; it doesn’t care which terminal is hot and which is ground. By switching the direction of the voltage, we change the direction the motor spins.



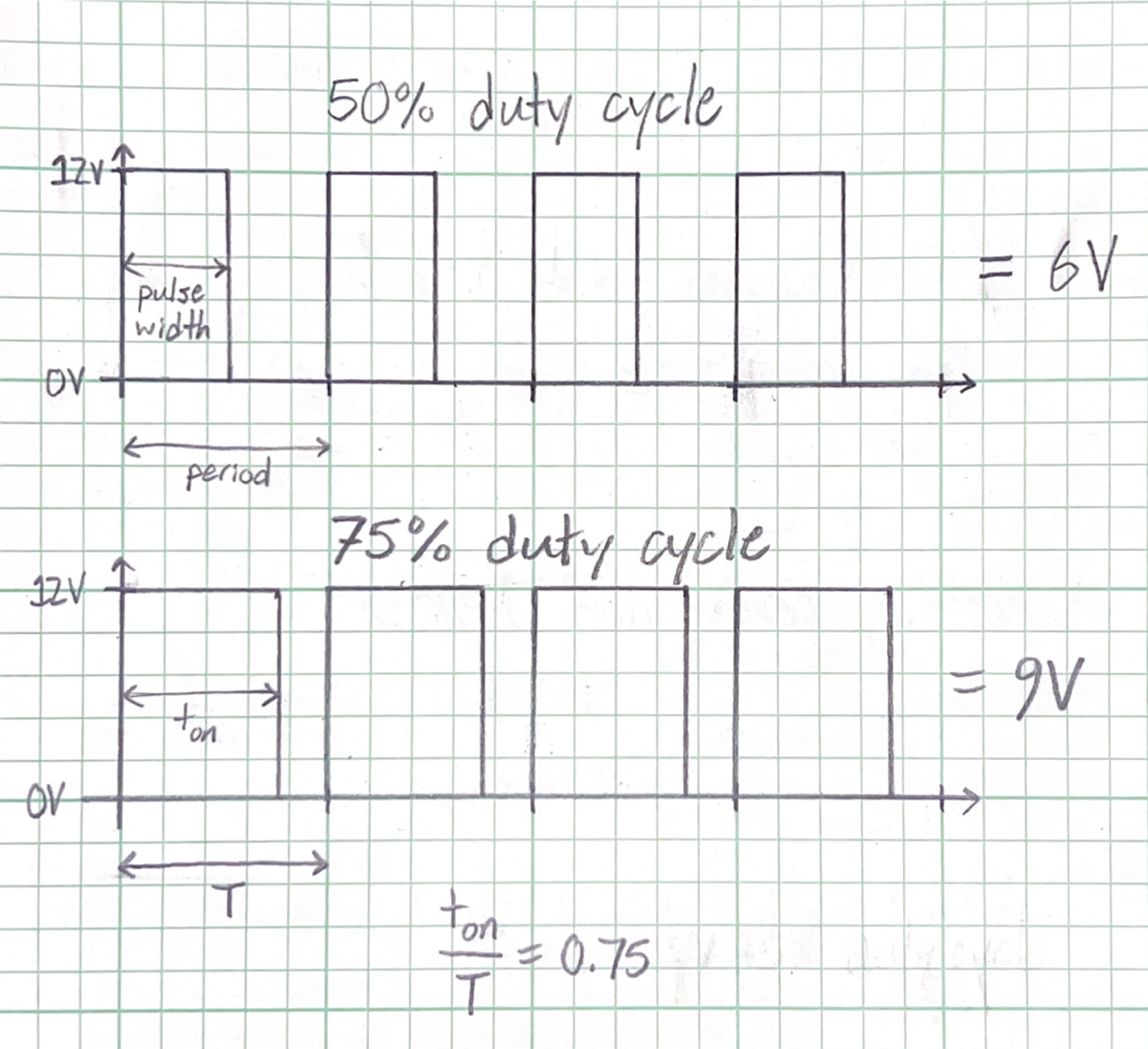
**Figure 3-2:** Basic operation of a brushed DC motor

So, we control the speed of the motors by varying the voltage, and the direction by switching the direction of the voltage.

**Varying Supply Voltage With** **PWM**

We’ve established that we control BDC motor speed by varying voltage, but the motors are connected to a battery that supplies a (relatively) constant 12V. We can’t exactly put our thumb over the battery connecter and control the flow of electricity like water coming out of a hose, so what do we do?

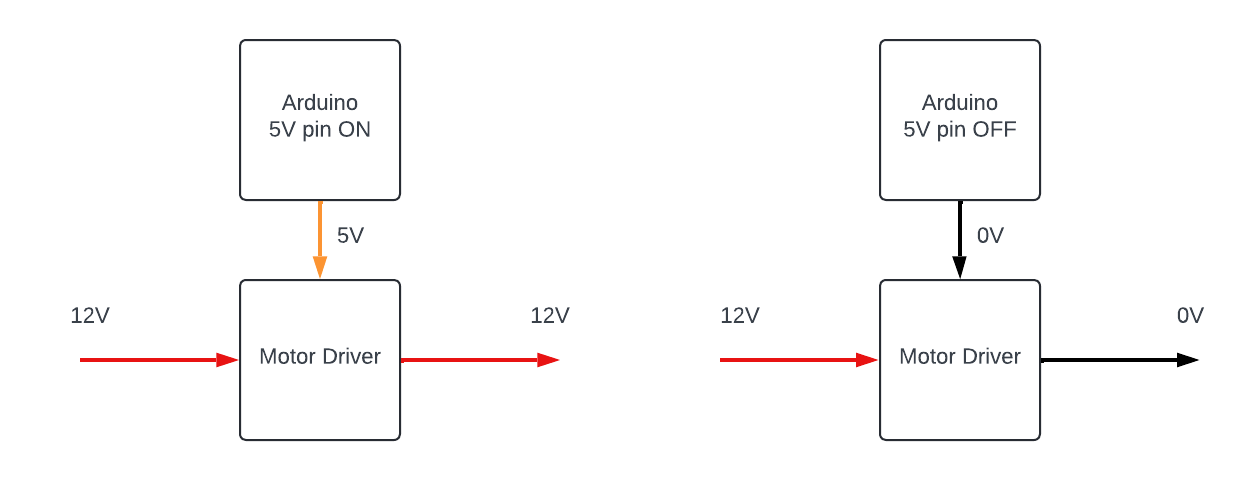
The answer lies in that motor driver we introduced in lesson 2 and something called Pulse-Width-Modulation (PWM). Our battery outputs a steady 12V, but if we were able to turn that voltage on and off again a few hundred times every second, that output would appear lower to our motor since it can’t respond to changes in voltage that fast. We could change the amount of time the voltage stays on vs off to change how much voltage the motor experiences. That’s what PWM is, and it’s how we vary the voltages the motors see to control their speeds.



**Figure 3-3:** How PWM duty cycle relates to apparent voltage

Let’s call the time between the start of each pulse the period, and the time in each period that our voltage is on the pulse width. The duty cycle, then, is simply the pulse width divided by the period. If our pulse width is ¾ of our period, then we have a 75% duty cycle and the voltage the motor sees will be 75% of our battery voltage. With our 12V battery, the motor would see 9V.

How do we turn the battery voltage on and off to generate these pulses though? That’s where the motor driver and Arduino come in. The motor driver acts as a sort of valve ([through the magic of transistors](https://www.build-electronic-circuits.com/how-transistors-work/)) which the Arduino can open and close by turning its 5V pins on (HIGH) and off (LOW).



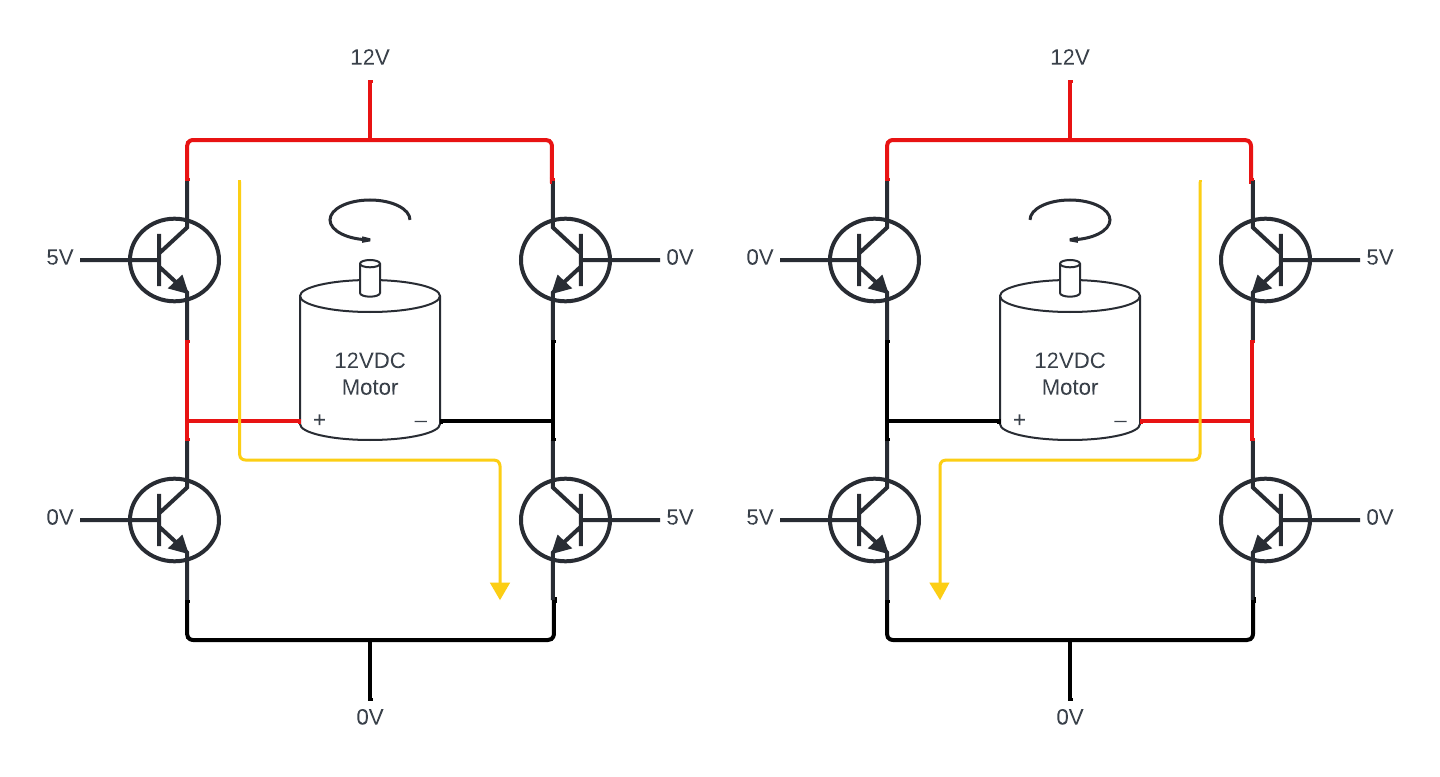
**Figure 3-4:** Simple example of motor driver operation

Certain pins on the Arduino are designed to produce the high-frequency pulses required for PWM, so by pulsing a pin on the Arduino at the proper duty cycle, we can turn our 12V supply voltage on and off just like we described above and precisely control the voltage the motor sees.

**Reversing Motor Direction With an H-Bridge**

At this point we can control our motor speeds with a signal we can produce from an Arduino. That’s a great start, but for our differential drive system, our wheels will frequently need to move backwards. We established before that our motors will turn backwards if we reverse the voltage applied to them, but how do we do that without stopping our robot and manually swapping the motor wires?

The answer is the second feature the motor driver offers: an H-bridge circuit. An H-bridge circuit consists of four *transistors*, [which are essentially switches](https://www.build-electronic-circuits.com/how-transistors-work/) that can be opened and closed by applying a voltage to their *base pin*.



**Figure 3-5:** How an H-bridge circuit reverses motor direction

By applying a voltage to opposite and opposed transistors, we can change the direction of the voltage across the motor and create a path for current (the yellow arrow) to flow through the circuit. Motor drivers may look fancy, but all they really do is package this circuit up into a neat and tidy chip for our convenience.

**Creating a GitHub Repository**

Before we discuss the last of our layers of abstraction, we’ll take a break to get into some programming. If we’re going to write code, it's a good idea to use a system to store and manage that code. Because of its ubiquity and usefulness as a collaboration tool, we’ll use GitHub to save and manage our code.

The first step will be to [install Git](https://github.com/git-guides/install-git) on the computer you’ll use to develop your robot code. If in doubt about any settings when using the installer, use the default option. You’ll know you have Git installed when you can run `git version` in your terminal and get a version number back rather than an error.

Next, create a [GitHub account](https://github.com/home?ef_id=_k_Cj0KCQjwvb-zBhCmARIsAAfUI2uWnnPpka1VOcYa5f6h_2tjjZPn53fP-GDVByNAWVQjMpnIiETKwKsaAjR2EALw_wcB_k_&OCID=AIDcmmcwpj1e5v_SEM__k_Cj0KCQjwvb-zBhCmARIsAAfUI2uWnnPpka1VOcYa5f6h_2tjjZPn53fP-GDVByNAWVQjMpnIiETKwKsaAjR2EALw_wcB_k_&gad_source=1&gclid=Cj0KCQjwvb-zBhCmARIsAAfUI2uWnnPpka1VOcYa5f6h_2tjjZPn53fP-GDVByNAWVQjMpnIiETKwKsaAjR2EALw_wcB) (if you don’t already have one). Creating a GitHub account should be quick and easy, but this next step may be a bit more challenging. For your computer to securely connect to GitHub, you need to [generate an *SSH key* and add it to your GitHub Account](https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account?platform=windows). The broad strokes for how to do this are:

1. In the terminal (or Git Bash on windows) check for existing ssh keys with `ls –al ~/.ssh`. If you get a “no such file or directory” error, move on to the next steps. Otherwise, you can either use an existing key or continue with the next steps.
2. Generate a new SSH key with `ssh-keygen -t ed25519 –C "your\_email@example.com"`. Follow the prompts to finish generating your SSH key.
3. Add your newly generated SSH key to the ssh-agent. The process for this varies depending on your operating system.
4. Copy the *public* SSH Key and add it to your GitHub account. Again, this process varies a bit with operating systems.

Next, follow this link to the Robotics [Workshop](https://github.com/Whitw-pers/my_rover) Template Repository. Click on the big green “Use this template” button and select create a new repository. You can keep the repository name the same or change it to something you prefer. Once you’ve created this repository, you’re going to clone it to a development workspace on your dev computer:

1. When you [install the Arduino IDE](https://www.arduino.cc/en/software) on your computer, it usually creates a folder called “Arduino” in your documents folder where it stores code you write in the Arduino IDE. Navigate there in your terminal [using cd and ls commands](https://tutorials.codebar.io/command-line/introduction/tutorial.html). The command will probably be something like `cd path\to\your\documents\Arduino`.
2. Clone the repository by clicking the large green “Code” button, selecting SSH, copying the URL, and running (in the Arduino directory) `git clone [copied URL]`.

Congratulations! This process is a bit tedious, but you now have a local workshop code repository on your computer, which is connected to a remote code repository on GitHub which you can use to share code with collaborators, store it safely outside your computer, and keep track of changes you make to your code.

**Reading The Basic Motion Code**

Enter the repository you just cloned and open the “lesson\_3\_basic\_motion.ino” file in the Arduino IDE (probably easiest to do in the graphical file explorer/finder app). You’ll see a lot of the bones of the basic motion script are already in place. We’ll quickly read through this script to understand how it will work and then our own additions to complete it.

/\*

By Whit Whittall

COSGC New Robotics Workshop code for lesson 3 basic motion

Finds wheel speeds based on input direction vector and drives brushed DC motors with dual H-bridge motor driver

left and right directions referenced in comments are from the robot's perspective

\*/

At the top of the script there is a block comment that explains what the code is for, how it works, and a few key points for reading and running it properly. It's good practice to include such comments in your code.

//--------------------rover geometry parameters--------------------

// motor\_controller() uses these parameters to calculate wheel velocities

const float r = 0.030; // radius of drive wheels in meters

const float L = 0.146; // width separating the drive wheels in meters

//--------------------declare motor pins--------------------

// setup() and drive() use these variables to control Arduino pins

// declare pins to control right motor

const int R1 = 8; //AI1 -> D8

const int R2 = 7; //AI2 -> D7

const int pwmR = 6; //PWMA -> D6

// declare pins to control left motor

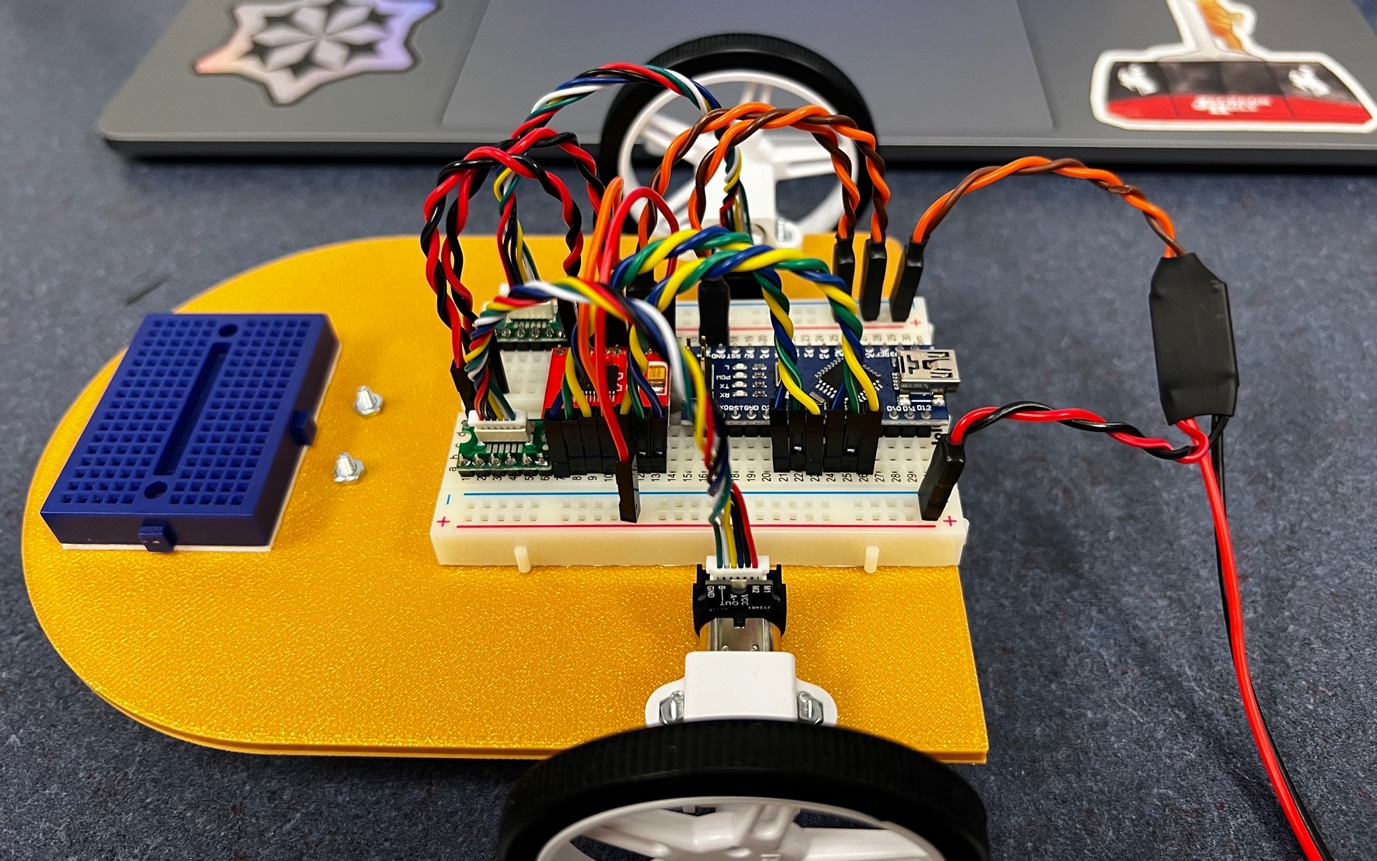
const int L1 = 5; //BI1 -> D5

const int L2 = 4; //BI2 -> D4

const int pwmL = 3; //PWMB -> D3

After our introductory comments, we declare our variables. In C++, [variables need to be declared](https://www.w3schools.com/cpp/cpp_variables.asp) (specify the type, variable name, and assign a value) before they can be used. Declaring these variables up here, outside of a function, makes these *global variables*. That means every function can see and use these variables.

While we’re thinking about which pins our Arduino uses to control the motors, let's make these connections on our robot. Under declare motor pins above, you can see which pins on the motor driver we need to connect to which pins on the Arduino. Let's do the right motor first. Take three jumper cables in different colors and twist them together. Plug them into the breadboard, making the following connections: A13 goes to A24, A12 goes to A25, and A11 goes to A26. Repeat the same exact process for the left motor, but this time you’ll make a different set of connections: A7 goes to A21, A8 goes to A22, and A9 goes to A23.



**Figure 3-6:** Motor logic pins (blue/green/yellow wires) properly connected

void setup() {

// put your setup code here, to run once:

//--------------------setup motor pins--------------------

pinMode(R1, OUTPUT);

pinMode(R2, OUTPUT);

pinMode(pwmR, OUTPUT);

pinMode(L1, OUTPUT);

pinMode(L2, OUTPUT);

pinMode(pwmL, OUTPUT);

}

Our microcontroller interacts with the world through its pins. For it to do so, we need to tell it which pins it’ll use and what it’ll use them for. For that, we use the pinMode() command. We don’t yet have any sensors, so all our pins will be outputs.

void motor\_controller(float v, float w) {

// determines required wheel speeds (in rad/s) based on linear and angular velocities (m/s, rad/s)

// maps required wheel speeds to PWM duty cycle

// expects -0.346 < v < 0.346 m/s, -4.73 < w < 4.73 rad/s

// motors will saturate if desired velocity vector is too large, best to keep desired velocities low

}

Before we talk about the main loop, we’re going to discuss our custom function. Custom functions are simple to write in Arduino C++, and we’ll see many more of them in this workshop. The `void` means that this function doesn’t return any sort of data when it’s called. The name of the function could be anything, but we’ll call it `motor\_controller` because that's a simple, descriptive name. Finally, it expects two *arguments* (another word for the inputs to a function) v and w, both of which must be *float* types. This all creates a function we can call with something like “motor\_controller(0.1, 1);” if we wanted to go forward at 0.1 meters per second and turn left at 1 radian per second. Right now, though, running this line wouldn’t do anything, since there are only comments inside the function.

**The Final Layer: The Motor Controller**

We need to add some code inside the motor\_controller(v, w) function that converts our desired velocity vector (a vector with components v and w) into the required left and right wheel speeds. Luckily, we’ve already solved this problem. Remember the inverse kinematics equations for a differential drive robot that we derived in lesson 1?

These equations find the wheel speeds necessary to achieve a desired velocity vector with components v and w. All we need to do is program these equations into our motor\_controller(v, w) function:

float dphi\_L = (v/r) - (L \* w)/(2 \* r);

float dphi\_R = (v/r) + (L \* w)/(2 \* r);

This gets us the wheel speeds that we want, but to turn the wheels, we need to map this to a PWM signal. In our case, we’re going to assume there is a linear relationship between motor supply voltage and motor speed (a good enough assumption in this case). The [Arduino breaks its PWM duty cycle down into 255 increments](https://docs.arduino.cc/learn/microcontrollers/analog-output/), i.e. if you PWM an Arduino pin at 255, then you have a 100% duty cycle. If you PWM at 153, that corresponds to a 60% duty cycle. 51 corresponds to a 20% duty cycle, and so on. 11.52 rad/s is the maximum speed (in forward and reverse) our motors can reach ([from the datasheet](https://www.pololu.com/product-info-merged/5225)), so we’ll use the [map()](https://www.arduino.cc/reference/en/language/functions/math/map/) function to map the top speed of our motors to the 100% duty cycle setting (negative numbers denote reverse, positive is forward). This is a simple example of [feed-forward control](https://en.wikipedia.org/wiki/Feed_forward_(control)):

// use the constrain function to keep dphi\_L and dphi\_R within certain boundaries

// this prevents unintended behavior of the map function

dphi\_L = constrain(dphi\_L, -11.52, 11.52);

dphi\_R = constrain(dphi\_R, -11.52, 11.52);

// need to confirm map() behaves well when given non-int input

// map() uses integer math, returns only integers which is not a problem in this case

// would be a problem if it misbehaves with float input

int duty\_L = map(dphi\_L, -11.52, 11.52, -255, 255);

int duty\_R = map(dphi\_R, -11.52, 11.52, -255, 255);

drive(duty\_L, duty\_R);

Finally, [we need to include some logic](https://docs.arduino.cc/built-in-examples/control-structures/ifStatementConditional/) to set all the pin outputs required to turn the wheels. For readability’s sake, we’ll do this in a different function, drive(duty\_L, duty\_R) which gets the PWM setting for each motor from motor\_controller() and [sets the H-bridge pins to generate the correct motor direction and sends the proper PWM signal](https://learn.sparkfun.com/tutorials/tb6612fng-hookup-guide) to set the motor speed:

void drive(int duty\_L, int duty\_R) {

// based on PWM duty cycle setting, assigns motor driver pin values

// expects duty\_L and duty\_R to be between -255 and 255

// left motor

if (duty\_L > 0) { // left motor forward

digitalWrite(L1, HIGH);

digitalWrite(L2, LOW);

}

if (duty\_L < 0) { // left motor backward

digitalWrite(L1, LOW);

digitalWrite(L2, HIGH);

}

if (duty\_L == 0) { // left motor stop

digitalWrite(L1, LOW);

digitalWrite(L2, LOW);

}

// right motor

if (duty\_R > 0) { // right motor forward

digitalWrite(R1, HIGH);

digitalWrite(R2, LOW);

}

if (duty\_R < 0) { // right motor backward

digitalWrite(R1, LOW);

digitalWrite(R2, HIGH);

}

if (duty\_R == 0) { // right motor stop

digitalWrite(R1, LOW);

digitalWrite(R2, LOW);

}

analogWrite(pwmL, abs(duty\_L));

analogWrite(pwmR, abs(duty\_R));

}

Using functions in this manner to organize our code makes for nice, readable code. Moreover, it makes our code modular. Later, we can write decision-making algorithms to determine the direction our rovers should drive. Because we made our motor\_controller() its own function, all we need to do is copy it over and provide it with the information it needs, and it’ll do its job just the same.

**Driving Our Robot**

Finally, each of the layers involved in controlling our motors is in place. We’ve created a modular system which can move our motors with a single command, now all we need to do is call that command in our main loop.

float v = 0.346; // linear velocity (0.346 is forward full, -0.346 is back full)

float w = 0; // angular velocity (4.73 is rotate left full, -4.73 is rotate right full)

motor\_controller(v, w);

delay(2000);

The motor\_controller() function we wrote needs to be fed a velocity vector to calculate what the wheel speeds should be. With our two degree-of-freedom differential drive system, that velocity vector is made up of a linear component (v) and an angular component (w). Like any variables, we need to declare them before we can use them. Declaring these variables within the main loop makes them *local variables*; they can only be seen and used within the main loop. We use variables with an identical name in the motor\_controller() function, so making v and w local variables within the main loop prevents variable scope conflicts between the functions.

Once we’ve assigned reasonable velocity variables to v and w, we can turn our motors with `motor\_controller(v, w)`. On its own inside the loop, this line will drive our robot along a consistent velocity vector. But what if we only want to move in a direction for a few seconds, and then change direction? For that, we can add delay() statements. The input to delay() is the time you want the robot pause before moving on to the next instruction in milliseconds. This code, for example, will drive the robot forward two seconds, then backwards two seconds, then turn left two seconds, and finally turn right two seconds:

// declaring v and w here limits their scope to loop() so there aren't conflicts with motor\_controller()

float v = 0.346; // linear velocity (0.346 is forward full, -0.346 is back full)

float w = 0; // angular velocity (4.73 is rotate left full, -4.73 is rotate right full)

motor\_controller(v, w);

delay(2000);

v = -0.346;

w = 0;

motor\_controller(v,w);

delay(2000);

v = 0;

w = 4.73;

motor\_controller(v,w);

delay(2000);

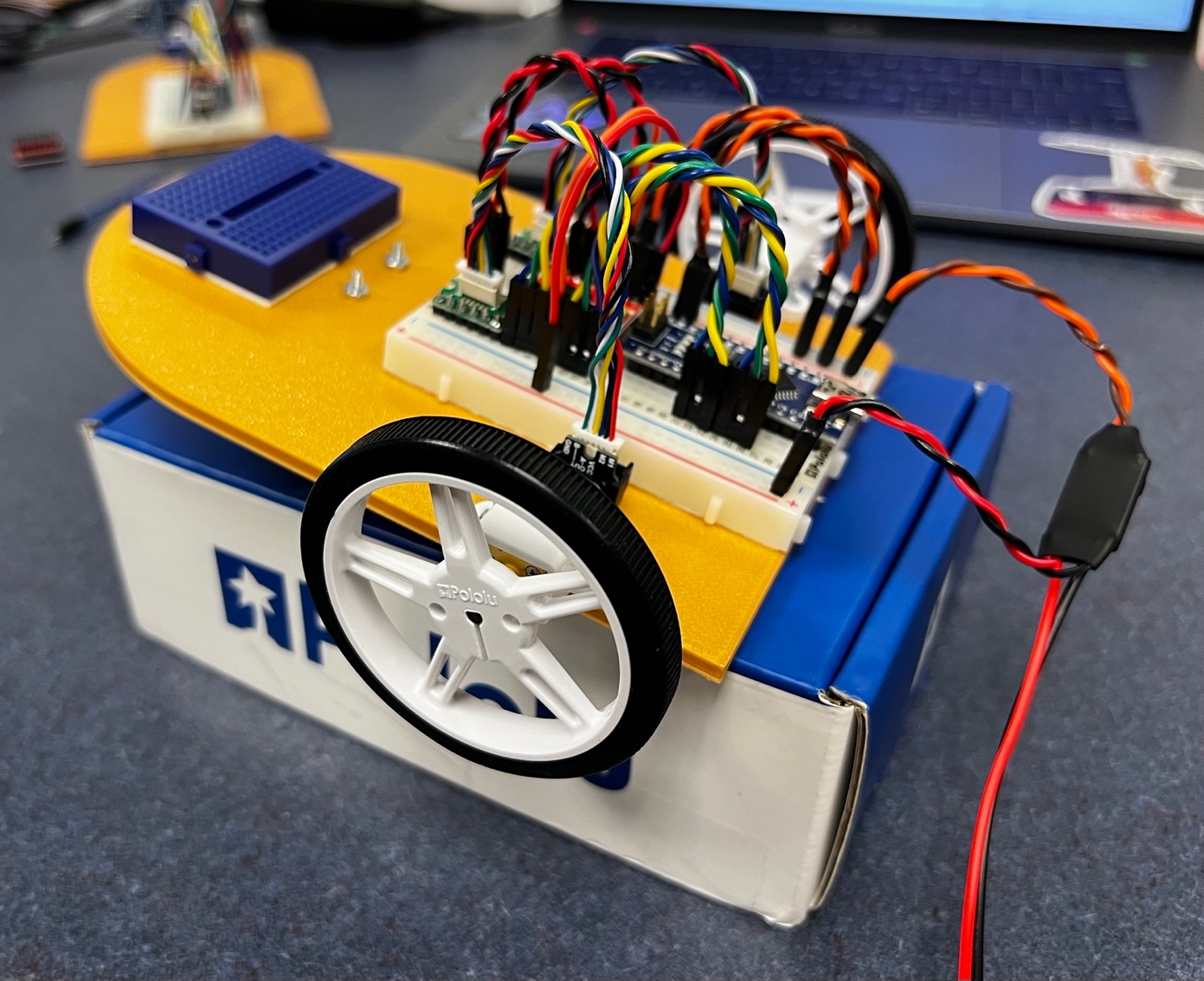
v = 0;

w = -4.73;

motor\_controller(v,w);

delay(2000);

Be careful when you first upload any driving code. The robot will immediately start driving away, and if you’ve got it set up on a workbench there's a good chance it may drive right off the edge of the bench and fall to the floor. It's smart to set the robot on a box any time you first test drive code to prevent the robot from doing its best baby bird impression. This also gives you a chance to troubleshoot any motor direction problems. If you know your code is telling the robot to drive forwards but one or both wheels are spinning backwards, a simple fix is to swap the twisted red and black motor wires at the motor controller. This will reverse the motor direction.



**Figure 3-7:** The rover resting on a box to prevent it running away during troubleshooting

Play around with the velocity vectors and delay statements and see what you can get the rover to do. Try combining linear and angular velocities to see what happens.

**Saving Code Changes in GitHub**

In this lesson we cloned a GitHub repository with useful code that has helped us control our rover. Since we cloned the repository, however, we’ve made a lot of changes to this code. To back up our work, manage our versions, and make our updates available to collaborators, we need to commit these changes to our local repository and then push them to our remote repository on GitHub. There is a simple process and set of terminal commands to do this:

1. In the terminal, navigate to your workspace within the Arduino folder. The command will be something like `cd path\to\documents\Arduino\my\_rover` depending on what you’ve named your workspace
2. Once in your workspace, type and run `git status`. The result should be a message with a list of changes not staged for commit.
3. Next, stage your changes for commit by typing and running `git add .` (the “.” is important, it tells git to add the changes in your current workspace)
4. You can check you’ve added your changes properly by running `git status` again. It should now tell you “Your branch is up to date with ‘origin/main’.”
5. Now type `git commit –m “[insert your commit comments here]”`. A good commit comment is short but describes the changes you’ve made recently. Committing your changes saves them to your local repository.
6. Finally, save your changes to your remote repository on GitHub by typing and running `git push`

Congratulations! You’ve now written some very useful rover code that we’ll build on in the next lessons and saved that code to GitHub. This process of committing and pushing changes may be a bit intimidating at first, but it becomes routine quickly and is useful for working on code as a group.